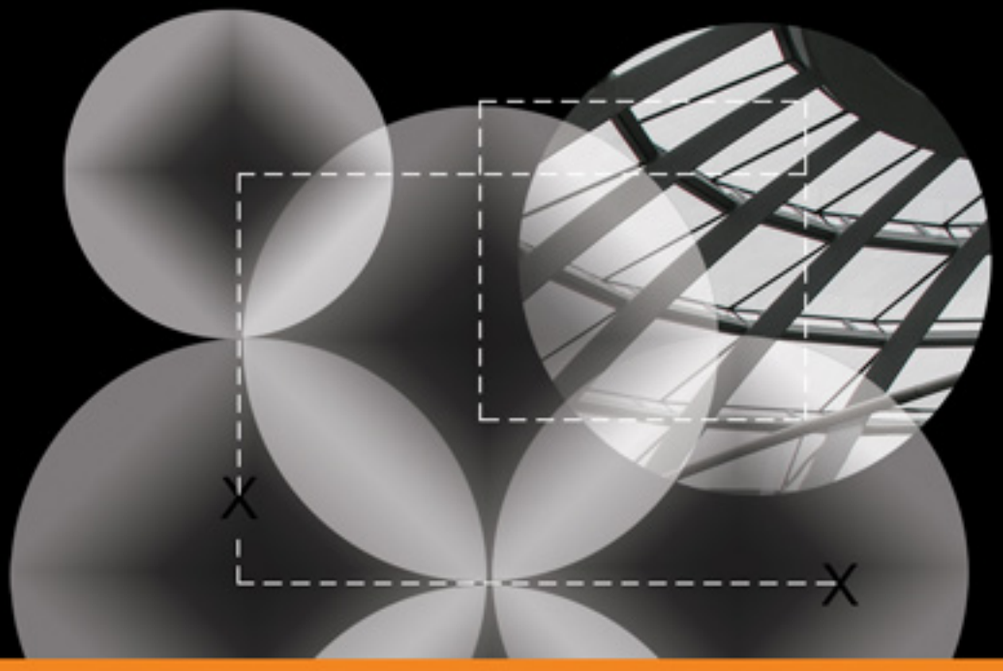
Иван Краљевски Ѓорги Какашевски



ОПЕРАТИВНИ СИСТЕМИ

И. Краљевски, Ѓ. Какашевски

О П Е Р А Т И В Н И С И С Т Е М И

Скопје, 2007

Рецензенти: вонр. проф. д-р Симе Арсеновски
Факултет за информатика
Европски универзитет - Скопје

вонр. проф. д-р Зоран Гацовски
Факултет за информатика
Европски Универзитет - Скопје

Лектор: проф. Емилија Арсеновска

Дизајн на корица: доц. Гордана Вренцоска

Краљевски Иван, Какашевски Ѓорги,
Оперативни системи, 2007

С О Д Р Ж И Н А

Предговор

Don't imagine you know what a computer terminal is. A computer terminal is not some clunky old television with a typewriter in front of it. It is an interface where the mind and body can connect with the universe and move bits of it about.

Douglas Adams

Компјутерите без сомнение предизвикаа револуција во човечката историја еднаква на пронаоѓањето на тркалото, барутот и парната машина. Првите електронски сметачи се појавиле веќе пред повеќе од половина век, а последната половина од тој период станаа забрзувачи на технолошкиот развој на човештвото. Без разлика дали се обични персонални сметачи кои се користат за едноставни задачи како што се пишување на електронско писмо или големи суперкомпјутери способни да предвидуваат глобални климатски промени или проучување на „алгоритамот“ на животот – генетскиот код, секојдневието на човештвото не може да се замисли без нив.

Еден компјутерски систем претставува само збирка од егзотични електронски уреди која не може ништо сработи, под услов да не постои рецепт т.е. низа од команди преку кои ќе се определи која компонента во кој момент што ќе работи. Впрочем компјутерските системи и служат за извршување на поставените задачи, иако понекогаш тоа не испаѓа како што го замислил програмерот. За да се извршат замислите на програмерите, постои колекција на програми кои работат со хардверот и ја олеснуваат задачата на програмерите да напишат код кој ќе биде јасен и ефикасен. Збирката на овие програми е позната како оперативен систем и претставува дел од системскиот софтвер во еден компјутерски систем.

Оперативните системи се темата на обработка на овој труд. Во текстот се објаснети поимите и концептите врз кои се изградени оперативните системи, без разлика дали се користат на големи суперкомпјутери или на преносни повеќефункционални уреди. Авторите се потрудиле колку што е можно повеќе претставените концепти да бидат и практично илустрирани преку примери и графикони. Концептите се презентирани на начин со кој се овозможува запознавање на лесен и едноставен начин со градбените елементи на модерните оперативни системи. Книгата е наменета пред сè за изведување на курсот по истоимениот предмет и за студентите на Факултетот за информатика на првиот приватен универзитет – Европски универзитет, Република Македонија. Како и за сите други заинтересирани за запознавање на основите на компјутерската наука, оперативните системи и системскиот софтвер.

Првото поглавје е воведно дава дефиниција на поимите поврзани со оперативните системи и нивните функции, историјата на развитокот на

оперативните системи, параметрите кои се земаат в предвид при дизајнот, видови на оперативни системи според начинот на обработка, функционалните особини, видови на окружување каде што работат, видот на структурата и т.н. Даден е и кус преглед на концептите на оперативните системи кои поединечно се обработуваат во посебни поглавја.

Во второто поглавје е претставен поим за јадрото на оперативниот систем и корисничка програма. Детално е опишан концептот на процеси, нивно создавање и завршување, хиерархија на процеси и состојби низ кои поминуваат во текот нивното извршување. Механизмите за имплементација на концептот на процеси и интерпроцесна комуникација, како и концептот на „лесни“ процеси и нивната споредба со обичните процеси.

Третото поглавје се однесува на начините за управување со процеси, нивото однесување, видови на распоредувачи на процесите. Даден е и преглед на познати алгоритами за распоредување на процесите на извршување во процесорот.

Интерпроцесната комуникација и синхронизација на процесите е опишана во четвртото поглавје преку објаснување на поимите за состојба на натпревар, критична секција, можни решенија за решавање на проблемите на конкурентност. На крајот од поглавјето се дадени и примери за класични проблеми од оваа област како и различни методи за нивно решавање.

Во петтото поглавје се објаснети концептите на ресурси во еден систем, поим за застои (блокади) како и причините кои доведуваат до нивно создавање кај оперативните системи. Претставени се и алгоритамите за справување со блокадите, нивна детекција, закрепнување и превенција од нивна појава.

Во шестото поглавје објаснета е улогата на меморијата во еден компјутерски систем, начинот на поврзување на логичкиот и физичкиот адресен простор, заштита на делови од меморијата од влијанието на конкурентни процеси, мемориската организација при различни начини на доделување на меморијата.

Проширувањето на концептот на меморијата кон виртуелна меморија и концептот на страничење е даден во седмото поглавје. Опишани се компонентите на овој подсистем, алгоритамите за страничење, а претставен е и концептот на сегментација на меморијата.

Потсистемот за управување со влезно-излезните уреди е опишан во осмото поглавје. Прво е дадена нивната класификација, архитектурата на В/И системот, начини на комуникација со уредите, основите и структурата на В/И софтвер.

Во деветтото поглавје се опишани поимите и концептите за системот на датотеки, поимот за датотеки и директориуми, операциите кои може да се изведат врз нив и нивните атрибути. Опишана е и реализација на системите на датотеки, доделување и управување со слободниот простор како и сигурноста и заштитата на системот на датотеки. На крајот од поглавјето се опишани

неколку значајни системи на датотеки кои се употребуваат во различни изведби на оперативни системи.

Во десеттото поглавје е даден опис на системот за управување со секундарни мемории – дискови, нивната структура, организацијата и справување со грешки, алгоритамите за движење на раката за читање и пишување. Опишани се и нивоата RAID изведби на системи за безбедно складирање на податоците.

Во единаесеттото и дванаесеттото поглавје се претставени практични изведби на оперативни системи кои се популарни и широко се користат. Нивното опишување е во светло на претходно опишаните парадигми и концепти, што овозможува подетален увид на начинот како се изведени и ја олеснува нивната меѓусебна споредба. Дадени се како пример, оперативните системи од фамилијата UNIX/Linux и Windows.

Тринаесеттото поглавје претставува прилог и во него се опфатени вежби и задачи кои се однесуваат на материјалот во целата книга.

1. Вовед

1.1. Поим и дефиниции за Оперативен систем

1.1.1. Видови софтвер

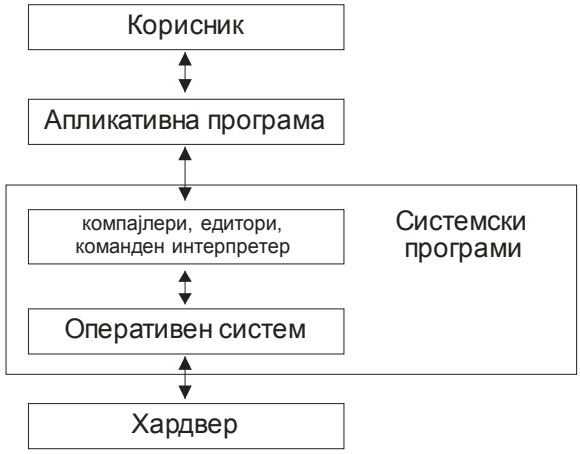
Поимот софтвер означува колекција на програми, кои му овозможуваат на компјутерот да изврши одредени задачи, како што се прибирање и обработка на податоци, пребарување по Интернет, обработка на слика, звук и т.н. Поимот софтвер е во контраст со поимот на компјутерски хардвер, кој се однесува на физичките врски и уреди потребни за складирање и извршување на софтверот. Во компјутерските системи софтверот се внесува во главната меморија (*англ. Random Access Memory - RAM*) и се извршува во централна процесорска единица (*англ. Central Processor Unit - CPU*). Гледано од најниското ниво, софтверот се состои од инструкции на машински јазик кои се специфични за одреден процесор. Машинскиот јазик се состои од групи на бинарни вредности кои означуваат одредени процесорски инструкции. Софтверот претставува низа од инструкции преку кои се менува состојбата на компјутерскиот хардвер по одреден редослед. Обично е напишан во виши програмски јазици кои се полесни и поефикасни за користење од страна на корисниците. Вишите програмски јазици се преведуваат или интерпретираат во машински јазик. Софтверот може да биде напишан и во асемблер, односно мнемоничка претстава на машинскиот јазик преку користење на алфанумерички симболи. Според намената софтверот може да се подели на два основни вида:

- *Системски програми*, овозможуваат работа на хардверот на компјутерот и на компјутерскиот систем. Тука спаѓаат оперативните системи, драјвери на уреди, дијагностички алатки, сервери, апликации за кориснички интерфејси. Намената на системскиот софтвер е да се изолира што е можно повеќе програмерот или корисникот од деталите на користениот компјутерски систем, меморија и други хардверски детали како што се комуникациите, печатарите, мониторите и т.н.
- *Апликативни програми*, им овозможуваат на корисниците да изведуваат една или повеќе специфични активности кои не се поврзани со хардверот на компјутерскиот систем. Типични апликации може да бидат за автоматизација на одредени процеси, бизнис програми, медицински софтвер, податочни бази, компјутерски игри и други. Секоја област од човековиот живот и работење на некој начин користи некоја форма на апликативен софтвер, со кој може да се автоматизираат одредени функции и операции.

1.1.2. Поим за оперативен систем

Најосновната системска програма е оперативниот систем. Модерните компјутерски системи се составени од: еден или повеќе процесори, меморија, дискови, печатачи, тастатури, монитори, мрежни интерфејси и други влезно/излезни уреди. Пишувањето на програми кои покрај својата намена уште треба и да вршат управување и следење на сите овие компоненти е многу сложено и тешко.

Поради оваа причина, компјутерските системи поседуваат софтверски слој, наречен оперативен систем, чија задача е да управува сите овие уреди и да му обезбеди на корисникот и неговите апликации поедноставен интерфејс кон хардверот.



Слика 1.1: Приказ на компјутерски систем

Локацијата на оперативниот систем во состав на еден компјутерски систем е прикажана на Слика 1.1. На најниското ниво се наоѓа хардверот, составен од три подслоја, најнискиот слој претставува множество на физички уреди, интегрирани кола, жици, електрично напојување и други слични уреди. Над него се наоѓа слојот на микроархитектурата на системот, каде што физичките уреди се групираат во функционални целини и обично содржи интерни регистри на процесорот, податочниот пат (*англ. datapath*) кој ја содржи аритметичко-логичката единица. Работата на податочниот пат ја управува софтвер наречен микропрограма или директно - хардверски кај некои машини. Над овие два слоја се наоѓа слојот на машинските инструкции кои го формираат инструкциското множество на процесорот или машинскиот јазик.

Машинскиот јазик обично се состои помеѓу 50 и 300 инструкции, поголемиот број од нив служи за поместување податоци во системот, изведување на аритметички операции и споредба на вредности. Во овој слој, влезно-излезните уреди се контролираат со помош на поставување на одредени вредности во интерните регистри на уредите. На пример, дискот може да биде

управуван за операција на читање (*англ. read*) преку издавање на вредноста на адресата на блокот на главната мемориска адреса, бројот на битови, со директно читање или пишување во интерните регистри. Реално, потребни се многу повеќе параметри и статусот кој се враќа по завршување на операцијата е многу посложен.

За да се прикрие сложеноста на системот се користи системски софтвер, односно слој на оперативниот систем. Тој се состои од слоеви на софтвер, со кои делумно се прикриваат деталите на хардверот и на програмерот - корисникот му презентира соодветни погодни инструкции со кои може да работи. На пример, како би се вршело управувањето во диск единица кога не би постоел оперативен систем? Во тој случај програмерот треба при дизајнирањето на програмата да ги земе в предвид и специфичностите при изведување на читање од дискета како што се придвижување на моторот, промена на аголот на механичката рака за позиционирање на главите. Со тоа се зголемува можноста да се направат поголеми грешки при програмирањето кои би довеле до неоптимален код. Спротивно на оваа, управувањето со диск единицата би го презела системска програма преку која програмерот ќе повикува одредени процедури за управување со уредот и нема потреба од познавање на деталите за работата на дискот. Овој пример ја опишува апстракцијата на хардверот која е и една од функциите на оперативниот систем.

Над оперативниот систем се наоѓа остатокот од системскиот софтвер, како што се командни интерпретери, преведувачи, уредувачи и други апликациско зависни програми. Треба да се разбере дека овие не се интегрален дел од оперативниот систем. Оперативниот систем се извршува во така наречен кернел (јадро - *англ. kernel*) начин или супервизорски начин на работа. Системските програми пак работат во кориснички начин и тие може да бидат заменети во други или подобрени верзии, што не е можно со процедурите на јадрото-кернелот. Над системските програми, се наоѓаат апликативните програми. Овие програми се купуваат или се пишуваат од корисниците за решавање на одредени проблеми или задачи, на пример уредувачи на текст, програми за табелирање, инженерски апликации, податочни бази.

1.1.3. Дефиниција и функција за оперативен систем

Оперативниот систем ги обединува разнородните делови на компјутерскиот систем во една целина и ги сокрива деталите за функционирањето на оние кои не се битни за програмерите или корисниците. Оперативниот систем извршува две главни функции:

- Обезбедува работно окружување за крајниот корисник на компјутерот, преку криење на непотребните детали за работа на хардверот. На овој начин, функцијата на оперативниот систем е на корисникот да му

претстави еквивалент на „продолжена машина“ или „виртуелна машина“ која е полесна за манипулација, отколку самиот хардвер.

- Модерните компјутерски системи се состојат од процесори, меморија, тајмери, електронски картички, печатачи, мрежни интерфејси и други уреди. Оперативниот систем во својата друга улога се јавува како обезбедувач на контролирано доделување на процесорот, меморијата и В/И уреди помеѓу програмите што се натпреваруваат за нив.

Значи, оперативниот систем претставува програма што посредува помеѓу компјутерскиот хардвер и корисникот, при што целта на оперативниот систем е да обезбеди околина во која корисникот може удобно и ефикасно да ги врши своите обврски (програмирање, обработка на податоци). Тој обезбедува различни сервиси кои можат да ги користат програмерите преку користење на специјални инструкции, наречени системски повици, кои подетално ќе бидат разгледани во продолжението на поглавјето.

Задачата на оперативниот систем е да води сметка за ресурсите на системот, односно да ги задоволи побарувањата на програмите, да следи кој од корисниците кои ресурси ги користат. При оваа архитектура на компјутерите, програмирањето е многу усложнето и оперативниот систем обезбедува т.н. виртуелна машина, односно единствен поглед на системот од страна на корисникот, која е независна од конфигурацијата на системот, конкретниот хардвер и самата архитектура.

Оперативниот систем е множество на системски програми и може да се опише со повеќе дефиниции:

- Алатка која го олеснува животот на програмерите,
- Управувач со ресурси каде што треба да обезбеди:
  - еднаков критериум на доделување на процеси од иста класа,
  - дискриминација помеѓу класи на процеси со различни побарувања,
  - максимизирање на протокот на податоците и минимизирање на времето на одзив.
- Контролна програма (политика на заштита и сигурносни мерки),
- Алатка за ефикасно работење на компјутерскиот систем,
- Виртуелна машина која е лесна за разбирање и програмирање.

Функцијата на оперативниот систем е да посредува помеѓу корисникот и хардверот и неговата цел е да:

- Извршува кориснички програми и да го олесни решавањето на корисничките проблеми,

- Го прилагоди користењето на системот за корисникот,
- Овозможи што поефикасно искористување на системскиот хардвер.

Оперативниот систем е еден од најважните и најкомплексните делови на компјутерскиот систем и се состои од повеќе независни целини. Не постојат цврсто дефинирани правила со кои се регулира распределбата на функциите на оперативниот систем по слоеви.

1.2. Историја на оперативни системи

Затоа што оперативните системи биле блиску поврзани со архитектурите на компјутерите на кои се извршувале, нивната еволуција може да се поврзе со компјутерските генерации. Пресликувањето на генерациите на оперативните системи кон генерациите на компјутерите обезбедува одредена структура за следење на нивниот развој, иако можеби во одредени периоди имало и преклопување.

1.2.1. Прва генерација: (1945-1955): Вакуумски цевки

Развојот на дигиталните компјутери започнал во Втората светска војна, каде повеќе истражувачи од повеќе нации успеале да создадат сметачки машини, кои првично биле засновани на механички релеи, а подоцна биле користени вакуумски цевки. Овие системи зафаќале огромен простор и биле изградени од десетици илјади вакуумски цевки кои често се расипувале. Системите работеле милион пати побавно од наједноставните денешни персонални компјутери. Во тоа време одредена група на оператери, ги дизајнирале, програмирале, управувале и одржувале овие машини. Сето програмирање се изведувало на машински јазик преку приклучување на приклучоци на табла слична на тогашните телефонски централи. Не постоеле програмски јазици, дури ни асемблер, ниту пак некаков вид на оперативни системи. Проблемите кои се решавале обично биле нумерички пресметки, кои се изведувале со помош на операторот кој управувал со системот. Пресметките се изведувале во тек на неколку часа, со надеж дека за тоа време ниту една од 20-те илјади цевки нема да прегори.

Во раните 50-ти години програмирањето е унапредено преку користење на дупчени картички, иако процедурата на извршување на програмите во суштина останала иста. Во оваа прва генерација, операторот на компјутерскиот систем имал комплетна контрола со управувањето на системот и тој можел да ги заврши сите потребни активности со време со оглед на бавноста на системот. Системот извршувал само една програма едновремено. Искористувањето на ресурсите на системот не било оптимално, најголемиот дел од времето се трошел на влезно-излезните активности на операторот, а многу помал дел на работата на централната процесорска единица. Иако овој

систем бил крајно неефикасен, односот на временските периоди бил во прифатливи граници заради малата брзина на процесорот.

1.2.2. Втора генерација (1955-1965): Транзистори и групна обработка

Со појавата на транзисторот во средината на 50-тите години од минатиот век, компјутерите станале многу подоверливи и поевтини и трошеле значително помалку енергија. Тоа довело до нивна комерцијализација и за првпат постоела јасна поделеност помеѓу дизајнерите, конструкторите, операторите, програмерите и одржувачите. Системите биле базирани околу концептот на голем централен компјутер кои бил наречен и „*mainframe*“. Само големите корпорации, државните агенции или универзитетите можеле да си дозволат ваков систем чија цена изнесувала неколку милиони долари.

Начинот на работа се состоел во тоа што програмерот ќе напишел програма во Фортран, ќе издупчел картички и ќе ги предадел на операторот, кој требало да ја подготви машината за извршување на програмата преку поставување на множество на картички за Фортран компајлер по што следела и самата програма. Заради губењето на време кое се трошело во активности за подготвувањето на системот за извршување на програмата се појавило решение за или групна обработка (*англ. batch)*. Идејата за овој концепт се состои во собирање на повеќе програми во подготвителна просторија каде што се запишувале на магнетната лента со користење на помал и поевтин компјутер како што бил во тоа време IBM 1401, кој бил соодветен за влезно-излезни операции, снимање на лента и печатење на резултатите, но не и за извршување на пресметките кои ги вршел поспособен и поскап систем какошто е IBM 7094.

Слика 1.2: Пример за групна обработка

По едночасовно прибирање на задачите (*англ. job*), магнетната лента се премотувала од почетокот и се носела во машинската соба. Операторот внесувал специјална програма (предок на денешните оперативни системи) која ја полнела и извршувала првата задача од лентата. Резултатите биле запишувани на друга магнетна лента или биле печатени на печатар. Потоа се полнела и следната задача. По извршувањето на целата група на програми, лентата со резултатите се носела во посебна соба за печатење.

Големите компјутери од втората генерација најчесто се употребувале за научни и инженерски пресметки, како пресметување парцијални диференцијални равенки. Типични оперативни системи од ова време се FMS (*Fortran Monitor System*) и IBSYS, IBM-ов оперативниот систем на IBM 7094. Во втората генерација на компјутерските системи се зголемува брзината на процесорите, капацитетот на главната меморија и се појавиле побрзи В/И уреди. Програмите се пишуваат на симболички машински јазик, асемблер или виши програмски јазици. Операторот веќе не е во состојба ефикасно и благовремено да го опслужува системот и решението се наоѓа во префрлување на контролните функции на самиот систем односно на посебни контролни програми. Тие се вклучуваат во одредени ситуации како на пример, подготовка за изведување на програмите, контрола на В/И уреди и активности околу полнење на програмите и презентација на резултатите од обработката. Во компјутерите од втората генерација се разликуваат два вида на програми: *контролни* и *кориснички*.

Слика 1.3: Компјутерски центар за групна обработка

1.2.3. Трета генерација (1965–1980): Интегрирани кола

За оваа генерација на компјутери е карактеристична појавата на IBM System/360, како одговор на потребата од помали, покомпактни и поевтини системи одошто комбинацијата на IBM 1401 и 7094. Серијата 360 претставувала низа од софтверски компатибилни машини со перформанси во опсег до IBM 1401 до IBM 7094. Сите претставници на серијата се разликувале во однос на цената и перформансите, но имале иста архитектура и инструкциско множество. Компјутерите биле ориентирани за нумерички пресметки во науката и инженерството и карактер-ориентирани комерцијални компјутери кои се користеле за сортирање на ленти и печатење во банки и осигурителни компании.

Идејата се состоела во тоа, целиот постоечки софтвер заедно со оперативниот систем да биде способен да работи на сите постоечки модели на компјутери. Таков оперативен систем требало да работи и на малите машини за копирање на дупчени картички на лента, како и на поголемите системи за изведување на сложени пресметки. Тој требало да биде добар и за системи и со малку периферни уреди и за системи со многу периферни уреди, да обработува комерцијални пресметки и научни пресметувања и притоа да биде ефикасен за сите поставени задачи.

Оперативниот систем со ознака OS/360 користен на системите од оваа серија, не можел да ги покрие сите поставени барања. Резултат на ова е голем и комплексен оперативен систем со неколку милиони линии на асемблерски код напишан од неколку стотици програмери, со многу грешки, кои се провлекувале и понатаму во следните негови верзии.

Иако по својата конструкција гломазен, овој тип на оперативен систем и други слични на него, ги задоволувале потребите на тогашните корисници.

Слика 1.4: Систем OS/360

Тука се претставени нови техники коишто не постоеле во Втората генерација оперативни системи, од кои најважна е техниката на мултипрограмирање (*англ. multiprogramming*). Заради избегнување на загубата од 80-90% од процесорското време при изведување на В/И операции кај комерцијалните програми, воведена е поделба на меморијата на неколку делови (партиции), со по една задача (*англ. job* – програма или множество на програми) во секој од нив. Додека еден чекал за В/И податоци, друг го користел процесорот. Ако постоел доволен број на програми кои се чуваат во главната меморија процесорот можел да биде зафатен скоро цело време. Постоењето на програми во делива меморија условувало постоење на специјален хардвер кој имал за задача заштита од нивното меѓусебно мешање и интеракција.

Друга главна особина претставена во третата генерација на оперативните системи е можноста за запишување на програмите од картичките на дискот

веднаш штом тие ќе бидат донесени од компјутерската соба. Потоа, секогаш кога некоја работа ќе биде завршена, оперативниот систем можел да побара нова програма од дискот во слободната мемориска партиција и да започне да ја извршува. Оваа техника се нарекува „спулинг“ (*англ. spooling*) и е употребена и за добивање излез од овие системи.

Поделбата на времето (*англ. time-sharing*) претставува техника која овозможува секој корисник интерактивно да работи со системот преку посебен терминал кој истовремено е и влезно-излезен уред за корисникот. Поделбата на времето е посебна форма на мултипрограмирање каде што на секој терминал му се доделува одредено процесорско време. По истекувањето на временскиот квантум, процесорот се доделува на друг терминал. Доколку терминалот е блокиран заради извршување на В/И операција процесорот и пред истекувањето на квантумот се доделува на друг терминал. Кај третата генерација на компјутерите треба да се истакне и појавата на првите два оперативни системи MULTICS и UNIX. Проектот MULTICS (*MULTIplexed Information and Computing Service*) е неуспешна идеја на компаниите MIT, Bell Labs и General Electric да се создаде моќен компјутерски систем и оперативен систем за работа со голем број на терминали. Основната идеја беше преземена од моделот на дистрибуција на електричната енергија, каде би било доволно секој корисник кој ќе посака да употреби некој електричен уред едноставно да го приклучи на електричната мрежата. Сличен модел требало да биде применет и кај предложената идеја, каде во секој град би постоел централен компјутер, а корисниците во своите домови би имале терминали со кои би се приклучувале кон него преку користење на модеми.

Другиот оперативен систем UNIX, е поедноставена варијанта на MULTICS системот која добила и практична реализација, развој и употреба и до денешно време. *Ken Thompson*, еден од научниците и програмерите на компанијата Bell Labs, кој работел на развојот на MULTICS, напишал мини верзија на системот за компјутерот PDP-7. Потоа настанал UNIX (UNI= еден, X = CS =Computing Service).

Компјутерските системи од третата генерација се нарекувале и мини компјутери, прв компјутер е PDP-1 на Digital Equipment Corporation, до тогаш најмалиот и најевтин мини компјутер кој тогаш чинел 120.000 долари.

Во третата генерација, заради појавата на мултипрограмирањето и зголемувањето на брзините, големината на меморијата и бројот на В/И уреди, сè повеќе контролно-управувачки функции се префрлаат од човекот на оперативниот систем. Програмерите се ослободуваат од низа сложени рутински работи и им се овозможува поголем ангажман на креативниот дел од работата. Покрај контролно-управувачките програми, во системите од третата генерација развиени се и низа на услужни кориснички програми кои имаат за задача уште повеќе да ја олеснат употребата на компјутерските системи. Тука

може да се направи поделба на софтверот според намената на: *системски* и *кориснички* односно апликативен софтвер.

1.2.4. Четврта генерација (1980-1990): Персонални компјутери

Со развитокот на LSI (*Large Scale Integration)* кола, започнале да се произведуваат чипови со 1000-ци транзистори на 1 cm^2 силициум. Во поглед на архитектурата персоналните компјутери (првично наречени микрокомпјутери) не се разликувале многу од PDP класата, освен во цената, која овозможила поголема достапност за компаниите и универзитетите.

Во тоа време се појавил и првиот оперативен систем за микрокомпјутери базирани врз Интеловиот 8-битен микропроцесор со ознака 8080 и 8-инчен флопи диск. Диск базираниот оперативен систем бил наречен CP/M (*Control Program for Microcomputers*) и подоцна станал сопственост на компанијата Digital Research. Во 1977, Digital Research го преработил CP/M за да го прилагоди за извршување на повеќе микрокомпјутери кои го користеле Интел 8080, Zilog Z80 и други процесорски чипови. Биле напишани многу апликативни програми за работа на CP/M, овозможувајќи му комплетна доминација во светот на микрокомпјутерите во период од 5 години.

Во раните 1980-ти години за потребите на IBM-овиот персонален компјутер, Бил Гејтс (сегашниот сопственик на Microsoft) понудил пакет кој го содржел неговиот интерпретер за BASIC и диск оперативен систем DOS (*Disc Operating System*). По направените измени, системот се преименувал во MS-DOS. Во 1983, на пазарот се појавил IBM PC/AT со Интел 80286 процесор, кој го користел MS-DOS, како и неговите следбеници 80386 и 80486. Иницијалната верзија на MS-DOS е релативно едноставна, а натамошните верзии вклучувале понапредни особини, некои преземани и од UNIX оперативниот систем.

Оперативните системи CP/M, MS-DOS употребувани на раните микрокомпјутери беа базирани врз принципот за внесување на корисничките команди преку тастатура. Концептот на графичкиот кориснички интерфејс, бил измислен уште во 1960-тите години, а бил прифатен и користен во страна на Xerox PARC истражувачкиот центар во нивниот микрокомпјутер Alto а подоцна и во комерцијалната верзија наречена Star.

По углед на Xerox-овиот оперативен систем, настанал оперативниот систем за Apple Macintosh, кој доживеал голем комерцијален успех заради комфорот при користењето на компјутерот дури и за неискусни корисници (*англ. user friendly*).

Наследникот на MS-DOS е создаван под големо влијание на успехот на Apple, што довело до појава на графички кориснички интерфејс наречен Windows, кој оригинално се извршувал над MS-DOS, односно претставувал негова школка. Тоа останало така и во подоцнежните верзии сè до 1995 година, како што се

Windows 95, 98 кои сè уште содржеле големо количество на асемблерски код од оригиналниот MS-DOS.

Друг оперативен систем на Microsoft е Windows NT (*NT – New Technology*), кој до одредено ниво е компатибилен со Windows 95, но внатрешно е целосно обновен. Тој во целост претставува 32-битен систем. Microsoft очекуваше дека првата верзија на NT во целост ќе ги замени MS-DOS и претходните верзии на Windows, заради супериорноста на системот. Но дури со верзијата Windows 4.0, се направи пробив на пазарот особено кај корпоративните мрежи. Верзијата 5 на Windows NT беше преименувана во Windows 2000 во почетокот на 1999. Намерата беше Windows 2000 да биде наследник и на Windows 98 и на Windows NT 4.0. Бидејќи тоа во целост не се оствари Microsoft се појави со уште една верзија наречена Windows Me (*Millenium Edition*). Врз основа на Windows 2000 понатаму се развиени и верзиите на XP, а во поново време и Windows Vista каде што кодот на системот претрпе поголеми измени .

Друг значаен оперативен систем во светот на персоналните компјутери е UNIX, кој има голема примена и се користи на работни станици и мрежните сервери. Неговиот дериват, Linux пак станува сè популарна замена на Windows оперативните системи како за студенти, така и за сè повеќе на корпоративните корисници. Сите UNIX-ови системи, иако примарно создадени за користење со командно линиски интерфејс имаат поддршка и за графички систем со прозорци наречен X Window кој овозможува основно управување со прозорците и овозможува на корисникот да креира, брише, преместува и да ја менува нивната големина користење на глушец.

Покрај класичните оперативни системи се појавуваат и два нови вида, *мрежни оперативни системи* и *дистрибуирани оперативни системи*.

Преку користење на мрежни оперативни системи, корисниците може да се поврзат со други оддалечени машини и да префрлуваат податоци од една машина на друга или стартуваат апликации со помош на соодветни протоколи. Оперативните системи може да бидат и различни, неопходен е само заеднички протокол за комуникација. Секоја машина има сопствен локален оперативен систем и има свој локален корисник (или корисници).

Дистрибуираните оперативни системи за разлика од мрежните, покрај делењето на датотеки и печатачи, овозможуваат и делење на процеси односно програми. На корисниците им се претставува како традиционален еднопроцесорски систем, иако е составен од повеќе процесори кои се поврзани преку компјутерска мрежа. Корисниците не мора да знаат од каде нивните програми функционираат и каде се складирани нивните податоци. Со сето тоа автоматски и ефикасно се справува оперативниот систем и корисниците го гледаат системот како еден компјутер.

1.3. Функција и концепти на оперативни системи

Функциите кои треба да ги извршува оперативниот систем може да се изведат врз очекуваните функции за еден сметачки систем, како што се:

1. Автоматско работење на компјутерскиот систем (без интервенција на операторот, полнење и извршување на програмите). Оперативниот систем треба да обезбеди функционалност на компјутерскиот систем без интервенција на операторот, заради бавноста на човекот. Во активностите кои оперативниот систем треба да ги извршува без интервенција на операторот спаѓаат: полнењето на програмата во меморијата од мемориски уреди и извршувањето на еден или повеќе програми.
2. Планирање и распоредување на работите и задачите (*англ. scheduling*), каде што се определува која задача или процес ќе се извршува во процесорската единица. Заради автоматска контрола на редоследот на извршување, оперативниот систем користи наредби за управувањето со работата на целиот компјутерски систем.
3. Мултипрограмирање. Техника за извршување на повеќе програми истовремено на еден систем, каде што секој добива дел од главната меморија, а процесорот се доделува според одлуките на распоредувачот на процесите.
4. Елиминирање на зависности на В/И операциите (техники на канал и прекини). Затоа што В/И операциите се многу побавни од процесорот, системот треба да изврши нивна изолација од процесорот. Заради оптимизација на користењето на ресурсите, потребно е В/И операциите да се комбинираат со извршувањето на другите процесорски задачи. Тоа се изведува преку имплементација на две хардверски техники: *канал* и *механизам за прекини*. Под канал се подразбира уред кој контролира еден или повеќе периферни уреди и е способен да пренесува податоци помеѓу уредите и меморијата без интервенција на процесорот. Како пример на овој пристап е DMA контролер (*Direct Memory Access*). Механизмот на прекини е универзална метода за известување на оперативниот систем за одредени настани, како што е завршување на В/И операција или барање за внимание на процесорот од страна на некој дел од системот. Прекините обезбедуваат високи перформанси и изолираност на В/И уредите: DMA пренесува податоци по каналот, а со соодветниот сигнал за прекин се известува процесорот за завршувањето на В/И активноста.

Според дефиницијата на оперативниот систем и очекуваната функционалноста на системот, оперативниот систем ги има следниве задачи:

- Стартување и прекинување на програмите и управување со процеси,
- Управување со меморија,

- Управување со влезно/излезни уреди,
- Управување со грешки и прекини,
- Управување со ресурси,
- Заштита на ресурсите од напади, грешки,
- Овозможување на повеќе-кориснички пристап,
- Комуникација со другите компјутери во мрежа,
- Обезбедување на интерфејс на операторите и корисниците.

1.3.1. Карактеристики на оперативните системи

Оперативните системи се карактеризираат со следниве особини:

1. *Конкурентност* е постоење на повеќе симултани или паралелни активности, како што се на пример преклопување на В/И операции и пресметковни операции или држење на повеќе програми во меморијата. Конкурентноста предизвикува одредени проблеми при преминување од една активност кон друга, проблемите со заштита на влијанието на едната активност врз другата и синхронизација на меѓусебно зависни активности.
2. *Делење на ресурси*. Конкурентните операции може да побаруваат делење на заеднички ресурси или информации заради обезбедување на доволно ресурси за задоволување на извршувањето на процесите.
3. *Постоење на трајна меморија.* Потребата за делењето на програмите и податоците доведува до потреба за трајно складирање на податоците со можност за брз пристап (*англ. long term storage*). Тоа го овозможуваат мемориски уреди со голем капацитет, односно секундарните мемории. Притоа треба да се решат повеќе проблеми како што се обезбедувањето на едноставен пристап на податоците, заштита на интегритетот на податоците од грешки со најразлично потекло.
4. *Недетерминизам.* Оперативниот систем треба да биде детерминистички ориентиран, што значи кога една програма се извршува под исти услови со исти податоци, треба да дава ист резултат, без разлика на времето на извршувањето. Од друга страна оперативниот систем треба да одговори и на недетерминистичко однесување на компонентите на системот. Треба да се предвидат сите ситуации во кои може да се најде системот и да биде подготвен да одговори на сите можни случувања.

1.3.2. Посакувани особини на оперативните системи

Настрана од веќе презентираните особини на оперативните системи, критериумите кои треба да ги задоволи еден оперативен систем за да одговори на целите и функциите спаѓаат:

1. Ефикасност на оперативниот систем може прецизно да се претстави само преку користење на повеќе критериуми каде што значењето на секој критериум зависи од видот и намената на оперативниот систем:
   - Средно време помеѓу задачите
   - Време на неискористеност на процесорот
   - Време на поминување на пакетска обработка (*англ. turn-around time*)
   - Време на одзив (*англ. response time*)
   - Искористеност на ресурсите
   - Пропусна моќ (број на задачи по време)

2. Мерка за ефикасност претставува односот на времето кога процесорот работи нешто корисно и вкупното време на работењето на процесорот за извршувањето на програмата: $e=t\_korisno/t\_vkupno$ каде што вредноста на e се движи помеѓу 0 и 1. ($0<e<1$)
3. Мерка за загуба на време (*англ. overhead*) е односот на времето кое процесорот го троши на изведување на системски работи и вкупното процесорско: $o=t\_odrzuvanje/t\_vkupno$ и притоа важи $0<o<1$ и $e+o=1$.
4. Високо ниво на доверливост, идеално гледано оперативните системи треба да се извршуваат без грешки, иако во реалноста тоа не е така, мерката која се воведува е изразена како средно време на појава помеѓу две грешки.
5. Едноставност во одржување, каде што ќе бидат вклучени што помал број на оператори. Затоа оперативниот систем треба да поседува модуларна структура, прецизно дефинирани интерфејси со детална придружна документација.
6. Прифатлива големина, каде што оперативните системи треба да заземаат што помал простор во меморијата. Иако со зголемувањето на достапната меморија и намалувањето на нивната цена се овозможува и работа и на поголеми системи, тие се посложени и подложни на појава на грешки и нивниот развој трае повеќе.

1.3.3. Видови на оперативни системи

Според намената, оперативните системи се делат на *системи општа намена*, кои извршуваат разни задачи како што е обработка на текст и слика и оперативни *системи за специјална намена*. Според начинот на обработка на задачите (процесите) се делат на:

- Системи со групна обработка (*англ. batch*), сериска или пакетска, е начин на работа каде што корисниците ги предаваат своите програми на извршување кои понатаму се извршуваат во низа по редоследот на пристигнување. Корисниците немаат можност за комуникација со своите програми. Времето на одзив е во граници од неколку минути.
- Интерактивни системи (*англ. time-sharing*), се карактеризираат со постоење на терминал за секој корисник, преку кои корисници задаваат програми и комуницираат со нив. Паралелноста во работата се постигнува со тоа што на секој кориснички процес се доделува одреден фиксен временски период од работата на главниот процесор. По завршувањето на временскиот рок процесот се прекинува и процесорот се доделува за извршување на друг процес од редица на чекање. Времето на одзивот е во граници од неколку секунди.
- Комбинирани системи, имаат особина на истовремено извршување на интерактивни и пакетски задачи. На пример, на еден систем корисникот во позадината може да започне неколку програми кои бараат процесорско време за извршување но не и интеракција со корисникот и во преден план програми кои бараат интеракција.
- Системи кои работат во реално време. Кај овие системи се изведуваат критични операции каде што од најголемо значење е одзив на системот. На некој процес му се овозможува одзив на системот во строга временска рамка изразена најчесто во милисекунди или секунди во зависност од самиот процес. На таквите процеси им се овозможува да преку генерирање на сигнал за прекин го добијат централниот процесор на располагање.

1.3.4. Класификација според функционалните особини

Според функционалните особини и видот на системите на кои се поставуваат и работат, оперативните системи може да се класифицираат и во следниве категории.

Оперативни системи за големи сметачки системи

Големите компјутерски системи (*англ. mainframes*) се разликуваат од персоналните компјутери според нивниот В/И капацитет. На пример, тие системи може да имаат и 1000 дискови и илјадници гигабајти податоци. Тие

уште се користат и како моќни сервери за WEB, електронски бизнис од поголем обем и сервери за бизнис - бизнис трансакции.

Тие се ориентирани за обработка на повеќе задачи истовремено кои побаруваат огромни В/И ресурси. Овозможуваат три вида на услуги: групна обработка, процесирање на трансакции и поделба на процесорско време. Групната обработка извршува низа од процеси без интеракција на корисникот, додека обработка на трансакциите опфаќа справување со голем број на краткотрајни барања. Ако процесите се мали, системот треба да биде во состојба истовремено да обработува илјадници од нив. Системите со временска поделба овозможуваат истовремено извршување на програми од повеќе оддалечени корисници, како на пример, пребарување на податочна база. Пример за ваков оперативен систем е OS/390 наследникот на OS/360, нивните следбеници zOS, Linux on System z и други.

Серверски оперативни системи

Едно скалило под претходно наведените оперативни системи за големи сметачки системи, се наоѓаат серверските оперативни системи. Се извршуваат на сервери, компјутери кои имаат помоќни карактеристики во однос на персоналните компјутери, или на персонални компјутери, па дури на големите системи. Серверите овозможуваат разни услуги како што се печатење, делење на датотеки или WEB сервис. Типични серверски системи се UNIX, Windows 2000/2003, Linux и други.

Повеќепроцесорски системи

Начин за да се зголеми моќноста на процесирањето е да се поврзат повеќе процесори во еден систем. Во зависност од начинот на поврзување, овие системи за нарекуваат паралелни компјутери, мулти-компјутери или мулти-процесори. Овие системи имаат потреба од специјализирани оперативни системи, но тоа обично се варијанти од серверските оперативни системи со посебни функции за комуникација и поврзување.

Оперативни системи за персонални компјутери

Задачата на овие оперативни системи е да обезбедат добар интерфејс кон крајниот корисник. Се користат за поддршка на апликациите како што се уредувачи на текстот, табеларни пресметки, пребарување по Интернет и други. Познати системи за десктоп компјутери се DOS, MS-DOS, UNIX, Linux, Windows и други.

Оперативни системи за работа во реално време

Кај овие оперативни системи, следењето на времето е клучен параметар. Обично се користат за управување на индустриски процеси каде се прибираат податоци и се врши контрола на машините и каде постојат прецизни временски рокови кои треба да се почитуваат. Ако акцијата апсолутно треба да се изврши во дадената временска рамка овие системи се нарекуваат „тврди системи во реално време“ (*англ. hard real-time*). Друг вид на системи за реално време се нарекуваат „меки“ (*англ. soft real-time*) каде што пропуштањето на временскиот рок е префатливо, но не и пожелно. Познати оперативни системи од овој вид се VxWorks (користен во мисиите на NASA, „*Spirit*“ и „*Opportunity*“ за истражување на планетата Марс) и QNX.

Интегрирани оперативни системи

Овие оперативни системи (*англ. Embedded Operating Systems*), се извршуваат на уреди кои не изгледаат како компјутери во права смисла, како што се PDA уреди, мобилни телефони, микробранови печки, телевизиски системи. Овие обично ги имаат особините на системи кои работат во реално време, но имаат ограничувања во поглед на меморијата, големината, напојувањето кои ги чини особени. Примери за вакви оперативни системи се PalmOS и Windows CE, Symbian и други.

1.3.5. Видови на окружување во кое работат оперативните системи

Според окружувањето во кое работат оперативните системи може да се поделат на:

- Традиционално. Еднокориснички или системи со поделба на времето каде што корисниците се приклучуваат преку своите терминали на серверите.
- Базирано на WEB базирано окружување. Умрежувањето базирано на принципите на глобалната мрежа WEB каде што постојат WEB сервери и клиенти во вид на персонални компјутери и мали мобилни уреди.
- Вградено окружување е типично за вградени системи, обично тврдите реални системи каде што не постојат монитори, тастатури или дискови. Влезните податоци се презентираат преку уреди како што се сензори додека статусните и излезни операции прикажуваат на LED диоди или помал дисплеј.

1.4. Концепти на оперативни системи

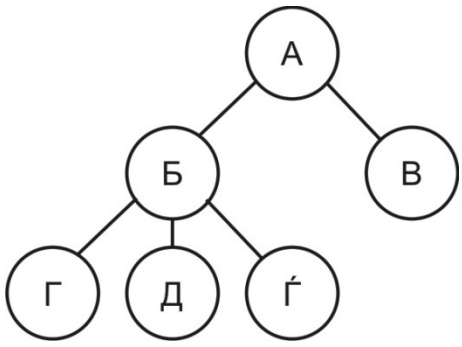
Сите оперативни системи имаат одредени основни концепти како што се процеси, меморија и датотеки, нивното разбирање е од најголема важност. Во продолжението на текстот ќе бидат накратко претставени основните концепти, а понатаму во книгата тие ќе бидат обработени во поширок обем.

1.4.1. Процеси

Основен концепт во сите оперативни системи е процес. Процесот претставува програма во извршување и е создаден од страна на јадрото како околина во која се извршува програмата. На секој процес му се придружува адресен простор, односно листа на мемориски локации од кои процесот може да врши читање и запишување.

Адресниот простор ги содржи самата програма која се извршува, нејзините податоци и стекот (*англ. stack*). Кон секој процес е придружено и множество од регистри, вредноста на програмскиот бројач, стек-бројачот и други хардверски регистри.

При своето извршување процесот може да биде запрен од страна на оперативниот систем и подоцна пак да биде стартуван. При суспендирањето на процесите сите информации за процесот се зачувуваат во табела на процеси (*англ. process table*). Суспендираниот процес во мирување се состои од адресниот простор и информацијата во која се наоѓа во табелата на процеси.



Слика 1.5: Процесно стебло

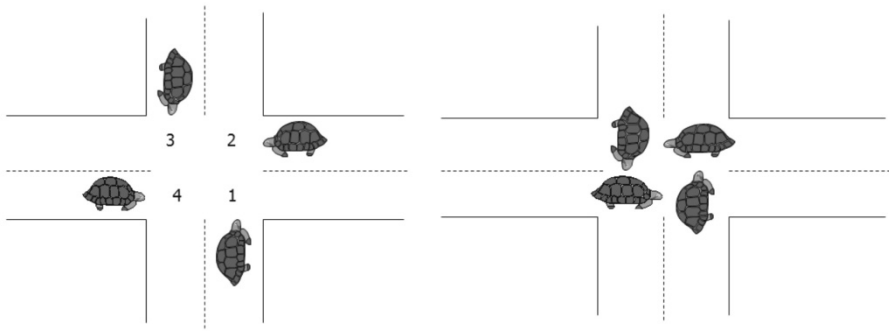
Процесот може да создаде еден или повеќе процеси, кои се наречени процеси – деца (*англ. child processes*), кои пак понатаму можат да создаваат свои деца – процеси и на тој начин да се формира стебло на процеси – хиерархија на процеси (Слика 1.5). Поврзаните процеси кои соработуваат, обично меѓусебно комуницираат и ги синхронизираат своите активности, тоа се нарекува интер-процесна комуникација.

Секој авторизиран корисник на оперативниот систем добива дескриптор со ознака *uid* (*англ. User Identification*) од системскиот администратор и секој

процес стартуван го носи *uid* од својот корисник. Процесите – деца го носат *uid* од процесот – родител, а корисниците можат да бидат членови на кориснички групи и да имаат дескриптор *gid* (*англ. Group Identification*).

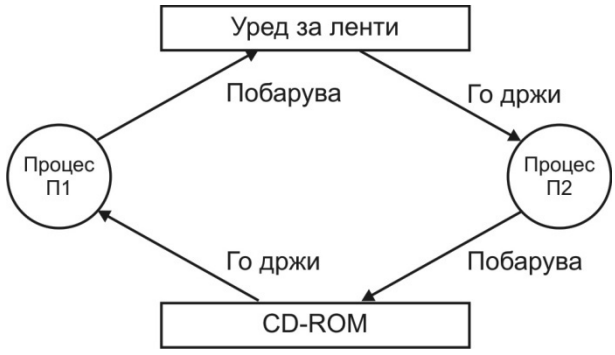
1.4.2. Застои (Deadlocks)

Кога два или повеќе процеси имаат интеракција, понекогаш можат да се доведат во безизлезна состојба, наречена блокада или застој (*англ. deadlock*) и ниеден од процесите не може да направи прогрес. Ова ситуација најдобро може да се опише преку пример од реалниот живот. На Слика 1.6 е опишан пример каде што четири „брзи возила“ истовремено налетуваат на раскрсница каде што нема сообраќајна регулација и согласно правилата за почитување на првенство на премин сите четири се заглавуваат во безизлезна состојба.



Слика 1.6: Појава на застој

Секој го блокира секој и ниту еден од нив не може да направи прогрес што евентуално би довело до излез од состојбата.



Слика 1.7: Застој во компјутерски систем

Друг пример, два процеси имаат за задача да префрлат податоци од CD-ROM на магнетна лента. Процесот 1 бара и добива право на користење на CD-ROM-от, истовремено процесот 2 бара и добива право на користење на магнетна лента. Потоа процесот 1 бара право на користење на магнетната лента и се суспендира сѐ додека процесот 2 не го врати уредот. И процесот 2 се суспендира очекувајќи да заврши процесот 1 и да го врати заземениот ресурс назад. Очигледно е дека и од оваа ситуација не постои излез.

1.4.3. Управување со меморија

Секој компјутерски систем поседува главна меморија која се користи за чување на програмата која се извршува. Во пример за едноставен оперативен систем само една програма едновремено се наоѓа во меморијата која треба да се замени за да се изврши друга. Покомплексните оперативни системи овозможуваат истовремено повеќе програми да се наоѓаат во меморијата. За да се спречи нивното меѓусебно влијание (како и влијанието врз самиот оперативен систем) потребно е да постои соодветен заштитен механизам контролиран од системот. Тоа се однесува на управување и заштита на главната меморија и на адресниот простор на процесите.

Обично секој процес поседува множество на адреси кои може да ги користи (од 0 до максимумот). Во наједноставниот случај максималното количество на адресниот простор е помал од капацитетот на главната меморија, така што целиот адресен простор на процесот ќе се содржи во главната меморија.

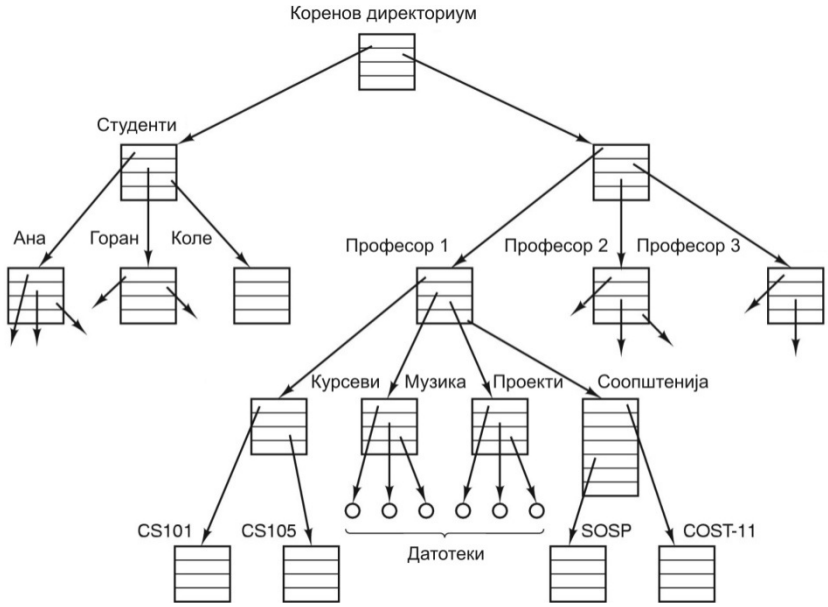
Треба да се најде решение за случајот каде процесот има поголем адресен простор отколку капацитетот на главната меморија на компјутерскиот систем. Кај првите компјутери, било невозможно да се извршуваат такви процеси, а во денешно време тоа е можно со користење на техниката наречена „*виртуелна меморија*“. Со помош на таа техника оперативниот систем чува еден дел од адресниот простор во главната меморија, а друг дел на единица од надворешна меморија (на пример, диск) и врши префрлување на деловите од една во друга меморија кога за тоа ќе се јави потреба.

1.4.4. Влез/Излез

Сите компјутери поседуваат физички уреди за преземање на влез и произведување на излез. Постојат огромен број на В/И уреди во состав на еден компјутерски систем (тастатури, монитори, принтери и т.н.), а задачата на оперативниот систем е да врши управување со овие уреди. Секој оперативен систем поседува В/И подсистем за управување и поддршка на В/И уредите. Одреден В/И софтвер може да биде независен од видот на В/И уредот и подеднакво добро да се употребува за повеќето или сите уреди. Од друга страна одредени делови како што се управувачките програми (*англ. device drivers*) се специфични за одредени В/И уреди.

1.4.5. Системи на датотеки

Главната функција на оперативниот систем е да ги апстрахира деталите и специфичностите на дискот и другите В/И уреди и на програмерот или корисникот да претстави разбирлив и јасен апстрактен модел од датотеки (*англ. files*) независни од типот на уредот. Со користење на системски повици се создаваат, бришат, се врши читање од и запишување во датотеките.



Слика 1.8: Стеблеста хиерархиска структура на системот на датотеки

Датотеките заедно, според нивниот вид се групираат во директориуми. На пример, може да се создаде еден директориум каде што би се сместиле програмите, друг за чување на текстуални датотеки трет за електронска пошта и т.н. Преку системските повици се врши создавање и бришење на директориумите, како и вметнување и вадење на постоечка датотека во директориумот. Директориумите се организираат хиерархиски во форма на стебло каде што секоја датотека е одредена и може да се референцира според нејзината патека (*англ. path name*) од врвот на стеблото од коренскиот (root) директориум.

За дадената слика за да се пристапи до датотеката со ознака CS101 се користи следнава патека:

```
/Fakultet/Profesor1/Kursevi/CS101.
```

Кога на почетокот од името на патеката постои „/“ со тоа се означува дека таа патеката е апсолутна и започнува од коренскиот директориум. Секој процес што се извршува има свој тековен работен директориум, на пример ако:

```
/Fakultet/Profesor1/
```

е работен директориум тогаш со користење на патеката со име *Kursevi/CS101* се добива истата датотека како и апсолутната патека дадена претходно. Процесите имаат можност со системски повик да ги менуваат своите работни директориуми.

Датотеките и директориумите се заштитуваат од страна на оперативниот систем, во UNIX тоа се врши со придружување на 9-битен заштитен код кон датотеката или директориумот. Овој код се состои од три 3-битни полиња, едно за сопственикот, едно за членовите на групата во која припаѓа сопственикот и едно за сите останати. Секое поле има бит за дозвола за читање, запишување и извршување (rwx – битови). На пример,

```
rwxr-x---x
```

Пред да може да се чита и запишува во датотеката, првин таа треба да се отвори при што се проверува заштитниот код. Ако пристапот е дозволен, системот враќа целобројна вредност наречена дескриптор на датотеката (*англ. file descriptor*) за користење во наредните операции, во случај ако пристапот е оневозможен тогаш се враќа код за грешка.

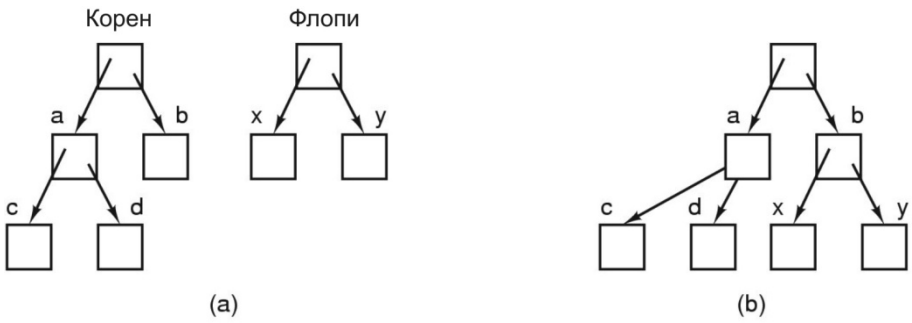
Друг важен концепт се специјални датотеки, тие овозможуваат В/И уредите да бидат претставени како датотеки. Така операциите за читање и запишување на уредите се вршат преку истите системски повици за читање и запишување на датотеки.

Постојат два вида на специјални датотеки: *блок специјални датотеки* и *карактер специјални датотеки*. Блок специјалните датотеки се користат за моделирање на уреди кои се состојат од множество на блокови со случаен адресен пристап како што се дисковите. Со отворање на блок специјална датотека и читање на блок 4, програмата може директно да пристапи кон 4-от блок на уредот без разлика на структурата на системот на датотеки.

Слично, карактер специјални датотеки се користат за моделирање на уреди како што се принтери, модеми и други кои прифаќаат или испраќаат низа од карактери (*англ. stream*).

Друг важен концепт кај UNIX е прикачувањето (*англ. mount*) на еден кон друг систем на датотеки. Сите персонални компјутери имаат една или повеќе

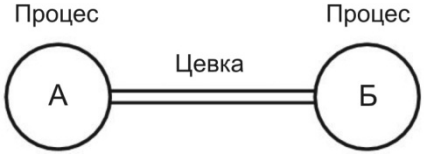
надворешни диск единици, флопи или оптички уреди. За обезбедување на едноставен начин за управување со овие уреди, кај кои се вметнуваат медиумите, UNIX и Linux овозможуваат прикачување на коренскиот директориум на уредите кон одредена точка на стеблото на системот на датотеките.



Слика 1.9: а) Пред прикачување (mounting), датотеките на флопито се недостапни, б) по прикачување, датотеките од дискетата се дел од хиерархијата на системот

Овој начин овозможува користење на системот на датотеките на уредот на која било позиција на коренскиот директориумски систем според барањата на одреден процес.

Цевка (*англ. pipe*) претставува „псевдо“ датотека која се користи за меѓусебно поврзување на два процеси.



Слика 1.10: Псевдо датотека - цевка

Кога процесот А сака да испрати податоци кон процесот Б, ги запишува на „цевката“ како во излезна датотека, а процесот Б ги чита податоците од неа како што би се читале од влезна датотека. Комуникацијата помеѓу процесите во UNIX (POSIX) наликува на обично читање и запишување во датотеки. Процесот може да открие дали запишува во обична датотека или „цевка“ само преку користење на специјален системски повик.

1.4.6. Команден интерпретер (Shell)

Оперативниот систем е кодот кој ги прифаќа и изведува системските повици. Уредувачите на програмски код, преведувачите (*англ. compilers*), асемблерите и командните интерпретери не се дел од оперативниот систем, иако се корисни и важни за работата на истиот.

Школката или *„Shell“*претставува команден интерпретер на оперативниот систем, кој иако не е дел од истиот користи добар дел од можностите на оперативниот систем и служи како одличен пример за користење на системските повици. Тој е примарен интерфејс помеѓу корисникот пред терминалот на својот компјутер и самиот оперативен систем, во случај кога не се користи графичка околина за работа. Постојат повеќе видови на командни интерпретери за UNIX и Linux како што се *sh, csh, ksh* и *bash*. Сите имаат слична функционалност која произлегува од оригиналниот интерпретер *shell* (*sh*).

Кога корисникот ќе се најави (*англ. login*) на системот се стартува shell-от и се испишува промпт знакот (кај UNIX тоа е знакот $). Со тоа се означува дека интерпретерот очекува команда од корисникот. Ако корисникот, на пример ја внесе следнава команда:

```
$ date
```

Процесот на командниот интерпретер создава процес -дете и го извршува програмот *date* како негово „дете“. Процесот родител (shell-от) чека сè додека не се заврши процесот дете, а потоа пак го испишува промпт знакот и се обидува да ја прочита следната влезна линија. Корисникот може да изврши пренасочување на стандардниот излез кон одредена датотека

```
$ date > file
```

Корисникот може да се изврши и пренасочување на стандардниот влез,

```
$ sort <file1 >file2
```

при што се повикува програмата *sort* со влез од *file1* а резултатот од извршувањето на програмата се запишува во *file2*.

Излезот од една програма може да биде искористен како влез во друга програма преку нивно поврзување со преку *pipe* цевката. Така,

```
$ cat file1 file2 file3 | sort > /dev/lp
```

програмата *cat* ги спојува трите датотеки и го пренесува излезот како влез во програмата *sort* за сортирање. Излезот се пренасочува кон датотеката */dev/lp* кое е типично име за специјална карактер датотека за печатарот. Ако корисникот ја внесе следната секвенца со додаден *&* (амперсанд) знак на крајот

```
$ cat file1 file2 file3 | sort > /dev/lp &
```

командниот интерпретер не чека да се завршат процесите - деца туку веднаш го враќа промптот, на тој начин се стартува програмот да се извршува во позадина, а корисникот може нормално да продолжи со својата работа.

1.4.7. Режими на работа на оперативни системи

Оперативните системи во себе вклучуваат два (или повеќе) начини на работа: *кориснички мод* (*англ. user mode*) и *системски*, нагледен или привилегиран начин (*англ. supervisor, system, privileged mode*) на работа. Поддршката за дуалниот начин на работа е изведена хардверски преку еден бит, наречен мод-бит кој е во составот на статусниот регистар на процесорот (*PSW – Processor Status Word*) како индикатор за начинот на работа на самиот компјутер, односно процесорот:

- нагледен (0) (*англ. kernel – supervisor, monitor, system*);
- кориснички (1) (*англ. user*);

Со овој мод – бит може да се разграничи кога одредена задача (*англ. task*) се извршува за потребите на оперативниот систем, а кога се извршува корисничка задача при што оперативниот систем секогаш работи во нагледен (контролен) начин, а корисничките програми секогаш работат во кориснички начин.

Во нагледниот начин се достапни инструкциите кои имаат големо влијание врз работата на системот, особено ако не се изведуваат правилно. Тие инструкции ги вклучуваат читањето и запишувањето на содржината на регистрите за специјални намени на процесорот, како што се регистри користени за одредување на пристапот кон одредени мемориски локации за да се изврши изолација на симултано активни апликации една од друга. Понатаму операцијата за пресликување на мемориски страни кон адресниот простор на

одреден процес, инструкции за поставување на нивото на приоритетите на прекините, инструкции за активирање на В/И уредите и други.

Премин од кориснички во контролен мод се јавува при појава на прекини кога е потребна услуга од некој В/И уред, при појава на посебни околности, како на пример во случај кога при извршувањето на корисничката програма се појавила грешка (делење со 0, грешка на магистралата и т.н.). Или при издавање на системски повик кога корисничката програма бара услуга од оперативниот систем.

Модерните оперативни системи се водени од прекини (*англ. interrupts*). Ако нема процес што се извршува, нема В/И побарувања и нема најавени корисници, оперативниот систем мирува и чека нешто да се случи, односно се извршува празен, бескорисен циклус (*англ. idle loop*).

Прекините се начин на кој хардверот го информира оперативниот систем за посебните околности на кои тој треба да посвети внимание. Појава на прекин условува процесорот да не пристапи кон извршување на следната инструкција од процесот што тековно се извршува, туку контролата ја превзема оперативниот систем и понатаму одлучува за избор на следен процес за извршување.

Случките (*англ. events*) секогаш се најавуваат со прекин (*англ. interrupt*) или инструкција за стапица (*англ. trap*). Кога е предизвикан прекин на работата на процесорот, процесот застанува со тоа што го работел при што:

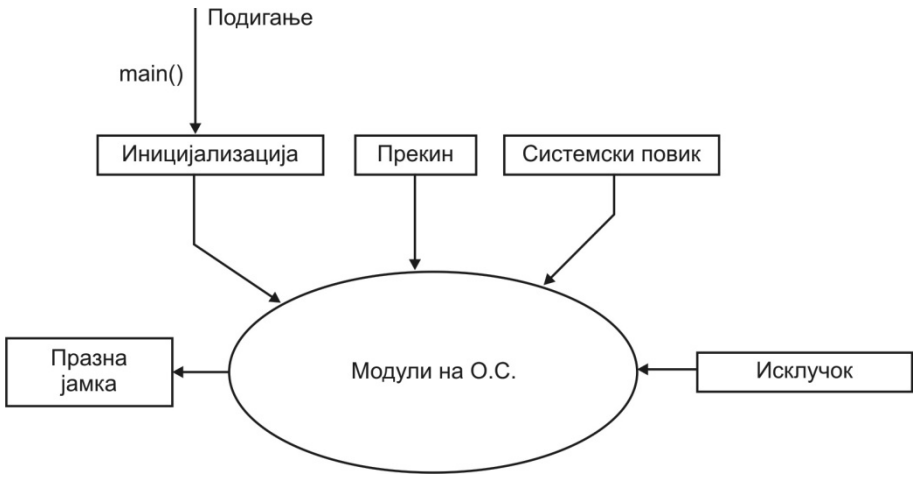
- Процесорот влегува во нагледен режим
- Ја дава контролата на ракувач со прекини (*англ. interrupt handler*) – сервисна рутина
- Адресата на ракувачот се наоѓа преку користење на бројот на прекинот како индекс во табелата на вектори на прекини
- Сервисната рутина се извршува и потоа оперативниот систем ја враќа запомнетата состојба на програмата.
- Корисничката програма продолжува од местото каде што била прекината.

Постојат два вида на прекини и тоа: *асинхрони прекини* кои создаваат од страна на надворешните уреди во непредвидливи временски моменти и *синхрони прекини* (интерни) кои се генерирани синхроно од страна на процесорската единица како резултат на појава на вонредна состојба, состојба на грешка или некој привремен проблем.

Стапица (*англ. trap*) претставува софтверски генериран прекин предизвикан од грешка на корисничката програма, при што, ако се појави инструкција која предизвикува стапица, оперативниот систем:

- ја идентификува причината за околноста;
- ја поставува состојбата на корисничката програма така што таа го повикува кодот за водење на околноста, ако е специфициран;
- ја враќа контролата на програмата.
- ако не е постои код за водење на околноста, оперативниот систем ја прекинува и повикува друга програма.
- мод-битот во PSW се поставува на 1 (*user mode*).

Недостаток од дуалниот мод кој е интегриран во хардверот може да предизвика сериозни оштетувања на оперативниот систем. На пример, MS-DOS напишан за Интел 8088 архитектурата не содржи дуален мод-бит. Корисничката програма што содржи грешка во кодот може да го избрише (или само дел) од оперативниот систем со запишување податоци во неговите мемориски локации или пак повеќе кориснички програми може да бараат достап до уред од компјутерскиот систем во исто време со можни катастрофални последици. Понапредните верзии на Интел процесорите, како што е процесорите со Pentium архитектурата и неговите наследници, содржат хардверската заштита со дуален мод. Како резултат на ова, помодерните оперативни системи Windows 2000/2003/XP/Vista и IBM OS/2 со употреба на оваа можност, обезбедуваат поголема заштита на самиот оперативен систем.



Слика 1.11: Тек на контролата на извршувањето на оперативниот систем

Оперативниот систем е само програма која има главна функција- main(), што се повикува само еднаш (при стартувањето – *boot* процес). Како секоја програма, тој користи ресурси (пример, меморија) и може да предизвика непредвидени случаи (се појавува исклучителна околност). Оперативниот систем е програма која се карактеризира со неколку особености како што се: внесена е од посебна локација, може да биде повикана симултано од повеќе

различни настани (пр. системски повик и прекин), не е предвидено да заврши, може да извршува било која наредба во машината.

1.4.8. Системски повици

Системските повици обезбедуваат интерфејс помеѓу корисничките програми (процесите) и оперативниот систем. Системските повици кои се наоѓаат во составот на некој интерфејс се различни од систем до систем. Апликативните програми комуницираат со оперативниот систем преку системските повици кои се реализираат со механизмот на прекини, како што е опишано подолу во текстот.

Некои системи дозволуваат системскиот повик да биде направен директно од програма на повисок програмски јазик, каде тие се претставени со множество предефинирани функции или повици на потпрограми.

Во процесорот може едновремено да се извршува само еден процес и тоа една инструкција во еден момент. Ако процесот е корисничка програма во извршување во кориснички начин на работа и има потреба од услуга на системот, треба да изврши системски повик со што ќе ја предаде контролата на оперативниот систем. Оперативниот систем треба да разбере што бара процесот кој издава системски повик преку проверка на понудените параметри кои се дадени како аргументи во повикот.

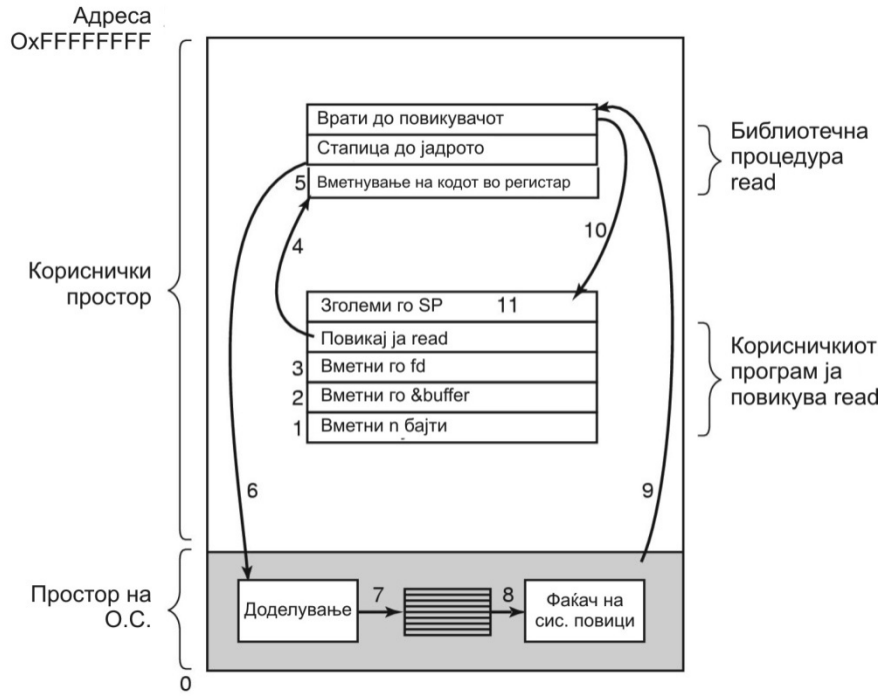
За да овој концепт стане појасен даден е пример за системски повик за читање на датотека (*англ. read*). Повикот има три параметри: референцата на датотеката, покажувач кон привремената меморија (баферот) и третиот е бројот на бајтите кои ќе се прочитаат.

```
count = read (fd, buffer, nbytes)
```

Системскиот повик го враќа бројот на прочитаните бајтови во променливата *count* и обично треба да е еднаква на *nbytes*, но може да биде помал во случај да се прочитани знаци до појава на *„end-of-file“* знакот. Ако се појави грешка заради невалиден карактер или грешка при читање на дискот, вредноста на *count* се поставува на -1.

Системски повици се извршуваат во низа од чекори. За дадениот пример, програмата која се подготвува за да издаде системски повик првин ги снима параметрите на стекот (чекор 1-3). Вредностите се поставуваат по обратен редослед, а по нив следува називот на инструкцијата (чекор 4). Оваа команда е обичен повик за процедура како и во секој виш програмски јазик.

Библиотечната процедура која обично е напишана во асемблер, го поставува бројот на системскиот повик во одреден регистар (чекор 5).



Слика 1.12: Извршување на системски повик

Со тоа се издава инструкција „стапица“ за префрлување од кориснички во режим на јадро и започнува со извршување на фиксна позиција во јадрото на оперативниот систем (чекор 6). Кодот на јадрото ги испитува бројот на системскиот повик и го дава вистинскиот ракувач со повикот преку консултирање на табела со покажувачи индексирани со бројот на повикот (чекор 7). Со тоа започнува извршувањето на рутината на ракувачот со повикот. Откога ракувачот на системскиот повик ќе ја заврши работата, контролата се враќа во корисничкиот режим на работа на инструкцијата која следува по *trap* командата (чекор 9). Понатаму се враќа на извршување на процедурата до кај стигнала (чекор 10). Вредноста на покажувачот на стекот се зголемува доволно колку да ги отстрани поставените параметри на стекот (чекор 11).

Некои виши програмски јазици, како што се C, C++, Perl, се дефинирани за да го заменат асемблерскиот јазик за употреба при системско програмирање. Овие јазици овозможуваат системските повици да бидат директно создадени и користени од кодот. На пример, системски повик кај оперативниот систем UNIX директно се изведува во C и C++. За модерните Microsoft Windows платформи, системските повици се дел од Win32 API функциите (*Application Programmer Interface*), кои може да се употребуваат со сите компајлери напишани за Microsoft Windows. На Слика 1.13 е даден споредбен приказ на некои покарактеристични системски повици за UNIX и Win32.

UNIX	Win32	Опис
fork	CreateProcess	Креира нов процес
waitpid	WaitForSingleObject	Чека процесот да заврши
execve		CreateProcess=fork+execve
exit	ExitProcess	Завршува процесот
open	CreateFIle	Креира или отвара постоечка датотека
close	CloseHandle	Затвора датотека
read	ReadFile	Чита од датотека
write	WriteFile	Запишува во датотека
lseek	SetFilePointer	Го поставува покажувачот кон датотека
stat	GetFileAtributesEx	Ги чита атрибутите на датотеката
mkdir	CreateDirectory	Креира директориум
rmdir	RemoveDirectory	Бриши празен директориум
link		Креира линк
unlink	DeleteFile	Бриши датотека
mount		Монтира уред
umount		Демонтира уред
chdir	SetCurrentDirectory	Го променува тековниот директориум
chmod		Го променува модот на датотеката (дир.)
kill		Убива процес
time	GetLocalTime	Го чита локалното време

Слика 1.13: Некои системски повици кај различни оперативни системи

1.5. Структура на оперативни системи

Управувањето со основните ресурси на компјутерскиот систем обезбедува повеќе делови или функционални групи на програми кои се наменети за:

- управување со процеси;
- управување со меморија;
- управување со датотеки (систем на датотеки);
- управување со В/И систем;
- управување со надворешна меморија;
- систем на заштита;
- мрежи;
- кориснички интерфејс.

Тие функционални групи обезбедуваат околина за извршување на програмите и обезбедуваат сервиси кои се користат да овозможат полесно програмирање, извршување на програмите, В/И операции, манипулација со датотеки, комуникација, детекција на грешки и други. Понекогаш е тешко прецизно да

се каже што се подразбира под делови на оперативниот систем, односно дали некои програми припаѓаат или не кон оперативниот систем.

Процесорот е еден од најважните ресурси на компјутерскиот систем иако во одредени случаи некои други ресурси може да бидат побитни. Управувањето со процесорот може да се подели на два нивоа: нивото на непосредно доделување на процесорот на некој процес и нивото за разрешување на приоритетот, односно избор кој од процесите кои чекаат на извршување има приоритет за да му се додели процесорот.

Под управувањето со меморијата се подразбира управувањето со работната односно главната меморија на компјутерите, кон која директно пристапува процесорот заради преземање на инструкции или податоци. На ова ниво се врши реализација на одредена стратегија за доделување на меморијата, редоследот и начинот за доделување и спроведување на одредена стратегија ослободување на меморијата.

Делот за управувањето со В/И уредите обезбедува независност на уредите, ефикасно работење и управување со В/И уредите, реализација на доделување на уредите на користење на процесите и физичко доделување на уредите, контролните единици и канали на процесите, како и стратегија за ослободување на уредите по завршувањето на процесите.

Делот за управување со податоците обезбедува средства за организација и пристап на податоците на начин соодветен за корисникот. Тука се реализираат следниве функции како што се креирање и бришење на датотеките, читање и запишување во датотеките, симболичко именување на датотеките, заштита од неавторизиран пристап и делењето на датотеките помеѓу повеќе процеси (корисници).

1.5.1. Системски програми

Системските програми како што е веќе наведено не претставуваат дел од оперативниот систем иако се дел од нивото на системскиот софтвер. Постојат голем број на различни системски програми кои помагаат при:

- Управување со датотеките каде што овие програми креираат, бришат, копираат, преименуваат, печатат, односно управуваат со датотеки и директориуми;
- Овозможуваат увид и следење на статусни информации и тоа се програми што даваат информации за состојбата на системот - датум, слободна меморија на дискот, број на корисници и сл. Оваа информација потоа се форматира и се печати на терминалот или на друг излезен уред или датотека;
- Системски програми за модификација и уредување на содржината на датотеки како на пример текст едитори;

- Поддршка на програмски јазици: компајлери, асемблери и интерпретери за некои програмски јазици (C, C++, Java, Visual Basic, Perl);
- Вчитување и извршување на програми (*англ. loading and execution*), системски програми што овозможуваат на асемблирана и преведена програма да биде вчитана во меморија и извршена;
- Системски програми со кои се овозможува комуникација: обезбедуваат виртуелна врска помеѓу процеси, корисници и различни компјутерски системи. Тие овозможуваат корисниците да испраќаат меѓусебни пораки на екран, да сурфаат WEB страни, да се најавуваат од далечина, да пренесуваат датотеки од една на друга машина.

Најважната системска програма на еден оперативниот систем е командниот интерпретер, чија главна функција е да ја земе и изврши наредната команда зададена од корисникот.

1.5.2. Системска структура

За големи и компликувани системи како што се модерните оперативни системи, потребно е внимателно да се конструираат за да функционираат исправно и да овозможат полесна модификација и надградба. Постојат повеќе концепции за проектирање на оперативните системи, од кои монолитна организација (*англ. monolithic systems*), слоевита организација (*англ. layered systems*), архитектура на микројадра (*англ. microkernel*), виртуелни машини (*англ. virtual machines*) и клиент-сервер системи.

Монолитна организација (Monolithic Systems)

Оперативниот систем е напишан како множество на процедури, каде секоја може да повика било која друга ако има потреба од тоа. Секоја од процедурите поседува добро дефиниран интерфејс во однос на параметрите и резултатите и може слободно да повика било која друга ако таа обезбедува корисна акција за првата која ја повикала.

За да се изгради ваков вид на оперативен систем, сите процедури прво се компајлираат и потоа се поврзуваат во еден објект со користење на системски поврзувач (*англ. linker*). Дури и кај монолитен систем можно е да постои некаква структура. Системските повици се повикуваат преку поставување на параметрите на повикот на точно дефинирани места и се повикува инструкција за стапица. Инструкцијата го префрла режимот на работа од кориснички во нагледен режим. Оперативниот систем тогаш ги прифаќа параметрите и определува кој системски повик треба да се изврши. Потоа ја индексира табелата и го наоѓа покажувачот кон процедурата која треба да го одработи

системскиот повик. Ова покажува како може да изгледа структурата за овој едноставен вид на оперативен систем:

- Главна програма која бара сервисна процедура.
- Множество на сервисни процедури за обработка системскиот повици.
- Множество на процедури кои помагаат при извршување на сервисната процедура.

Во овој модел, за секој системски повик постои една процедура која го обработува, а процедурите кои помагаат на сервисната процедура обично се грижат за операции како што се преземање на податоци од корисничката програма.

Многу комерцијални оперативни системи немаат добро дефинирана структура. Нивниот развој започнал како мали, едноставни и ограничени системи, а потоа како што се појавувала потреба така биле надградувани. Оперативниот систем MS-DOS е таков пример, кој во почетокот бил дизајниран и имплементиран како едноставен систем и не било познато дека тој ќе стане доста популарен. Напишан е да обезбеди најголема можна функционалност во најмал простор (поради ограниченият хардвер на кој работел), па затоа не е разделен во модули и има монолитна структура.

Оперативниот систем UNIX е друг пример за систем кој во почетокот бил ограничен со хардверската функционалност. Тој се состои од два дела, јадро и системски програми. Јадрото е поделено во серија од интерфејси и драјвери на уреди, коишто се проширувани и додавани во текот на еволуцијата на UNIX. Јадрото го обезбедува функционирањето на системот на датотеки, доделувањето на процесорот, управувањето со меморијата, како и други функции кои се користат преку системските повици.

Овој пристап претставува големо оптоварување за само едно ниво. Затоа оперативниот систем UNIX е тежок за надградба и модификација бидејќи промена во една секција може да значи упад во функционалноста на друга секција. API функциите на UNIX се дефинираат со системските повици. Корисничкиот интерфејс е дефиниран од достапното множество системски програми. Програмскиот и корисничкиот интерфејс ги дефинираат потребите што треба да ги поддржува јадрото. Новите верзии на UNIX се дизајнирани со цел за инсталација и работа на понапреден хардвер.

Организација со повеќе слоеви (Multilayer Systems)

Преку генерализирање на обидот за структурирање кај монолитните системи се добива организација на хиерархија на слоеви, наредени еден над друг. Овој пристап овозможува поголема контрола над компјутерот и над апликациите што се извршуваат, затоа што структурата на еден оперативен систем може да се разбие на помали делови. Поделбата на модули, дозволува на дизајнерите да

вршат прикривање на информации и со тоа да имплементираат рутини на пониско ниво така што надворешниот интерфејс на тие рутини и резултатот на таа рутина остануваат неизменети.

Значи концептот може да се сумира:

- Оперативен систем со монолитна структура се состои од множество на процедури без било какво групирање и хиерархија.
- Оперативниот систем со слоевита структура се состои од повеќе слоеви од кој секој има прецизно дефинирана функција (управува со одредени ресурси) и се потпира врз функциите од пониските нивоа кон кои пристапува со користење на системските повици.

Воопшто, оперативните системи со слоевита хиерархија се делат на повеќе слоеви од кој секој се гради на основа на претходен слој. Најнискиот слој (со ознака 0) го претставува хардверот додека највисокиот (слој N) го претставува корисничкиот интерфејс.

Секој слој претставува имплементација на апстрактен објект кој е енкапсулација на податоци и операции кои можат да манипулираат со тие податоци.

Ваква структура овозможува повикување (користење) на операции на создавање или уништување на процеси само од слојот од хиерархијата кој се наоѓа над сите слоеви во хиерархијата на оперативниот систем. Тоа значи дека постењето на процесите исклучиво е поврзано за корисничкиот слој.

Иако сите процеси се наоѓаат во корисничкиот слој, тие меѓусебно се раздвоени со тоа што секој од процесите поседува сопствен адресен простор – кориснички простор (*англ. user space*). На сличен принцип се базира и раздвојувањето на процесите и оперативниот систем. Раздвојувањето на корисничкиот простор од системскиот простор, оневозможува повиците за операциите на оперативниот систем да се базираат врз користењето на потпрограми. Затоа е и потребно користење на системските повици кои овозможуваат преминување од кориснички во системски простор заради повикување на операции на оперативниот систем. Системските повици бараат користење на специфични асемблерски команди и заради тоа се прикриваат во самите системски потпрограми. Некои од овие потпрограми покрај системските повици, содржи и специфична обработка на податоците како што е на пример, претворање од знаковен во бинарен облик и обратно кај форматираниот влез и излез. Секој системски потпрограм е наменет за повикување на една од операциите на оперативниот систем наменети за корисничкиот слој. Овие операции се нарекуваат системски за да се разликуваат од внатрешните (интерни) операции на оперативниот систем наменети само за користење во состав на оперативниот систем.

Системските потпрограми формираат системска библиотека, според тоа повикување на системските операции се сведува на повикување на потпрограмите од системската библиотека. Системската библиотека се испорачува со оперативниот систем и се користи при постапката на поврзување (*англ. linking*). Во текот на поврзувањето, библиотечните програми се врзуваат за објектен облик на корисничката програма заради создавање на извршната верзија на корисничката програма. Благодарејќи на системските потпрограми, системската библиотека, оперативниот систем претставува дел од корисничката програма иако не е интегрален дел од неа. Затоа може да се смета дека извршувањето на системските операции претставува составен дел на извршувањето на програмата односно процесорот.

Со модуларноста се поедноставува дебагирањето и системската верификација. Првото ниво може да се дебагира без да се води сметка за остатокот од системот, бидејќи по дефиниција тоа го користи само основниот хардвер за имплементирање на своите функции. Со негово дебагирање, точноста на неговите функции може со сигурност да се претпостави за дебагирање на второто ниво и т.н. Затоа се вели дека дебагирањето е поедноставено кога системот е раздвоен на нивоа.

Најголемата тешкотија на овој пристап е дефинирањето на самите слоеви, бидејќи еден слој може да ги користи само услугите на слоевите под него. На пример, кога корисничката програма извршува В/И операција, таа извршува системски повик од В/И слојот, кој го повикува слојот за управување со меморија, кое пак го повикува слојот за распоредување на процесорот и на крај повикот се пренесува на хардверот. Кај секој слој параметрите на В/И операција може да се менуваат, а може да се појави и потреба за пренос на податоци и т.н. Ова резултира со потреба од повеќе време за извршување на операциите отколку кај системите со едноставна монолитна структура.

Организација со микројадра (Microkernel systems)

Во текот на развитокот и со проширувањето на UNIX оперативниот систем, јадрата станале преголеми и сложени за управување. Во средината на 80-тите, развиен е оперативен систем наречен Mach, кој го модуларизира јадрото, со користење на микро јадра (микрокернели). Концептот на микројадро претставува современ концепт во реализација на модерните оперативни системи. Основната идеја е да се создаде минимално јадро со високи перформанси, а сите други функции на јадрото да се потиснат во корисничкиот простор. Со овој метод од јадрото се отстрануваат сите несуштински компоненти и тие се имплементираат како системски и кориснички програми.

Корисничките модули може меѓусебно да комуницираат со испраќање на пораки (*англ. message passing*). Типично, микројадрата обезбедуваат

минимално управување со процесите и меморијата, а имаат побогати можности за комуникација.

Добрата страна на овој пристап е полесното проширување на оперативниот систем. Сите нови сервиси се додаваат во корисничкиот простор и не бараат измена на дотогашното јадро. Кога ќе се укаже потреба за менување на јадрото, измените не се толку големи, бидејќи микројадрото не е многу обемно.

Таквиот оперативен систем е полесен за инсталирање на различни хардверски платформи. Микројадрата, исто така обезбедуваат поголема сигурност и доверливост, бидејќи најголемиот број сервиси се одвиваат во корисничкиот простор, а не како процеси во јадрото.

Организација со виртуелни машини (Virtual machines)

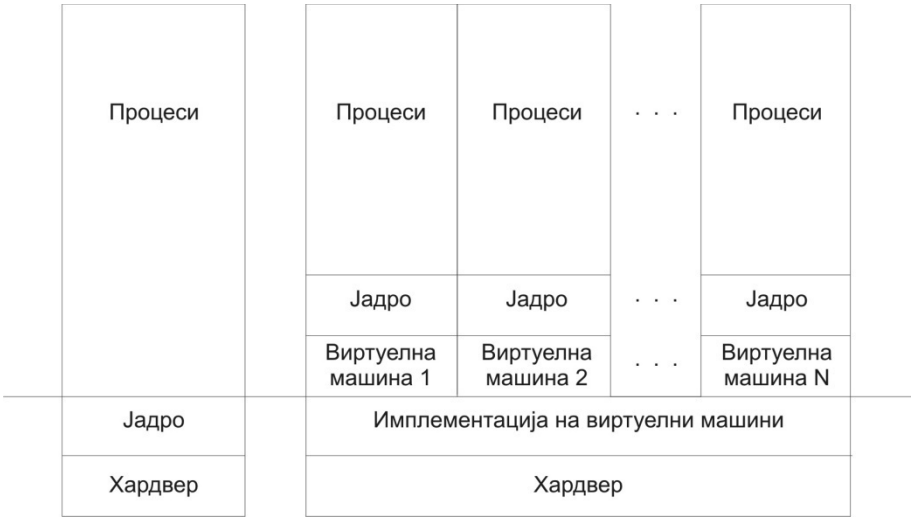
Концептуално, еден компјутерски систем е составен од слоеви. Кај секој систем хардверот се наоѓа на најнискиот слој. Јадрото што се наоѓа на следното ниво ги користи хардверските инструкции за да креира множество системски повици, кои пак се користат од страна на повисоките слоеви. Системските програми над јадрото може да ги користат системските повици или хардверските инструкции и на некој начин тие не прават разлика помеѓу нив.

Некои оперативни системи ја прошируваат оваа шема, така што овозможуваат апликациските програми лесно да ги повикуваат системските програми. Повторно, иако системските програми се наоѓаат на еден слој повисоко од останатите рутини, апликациските програми можат сè она што се наоѓа под нивното ниво да го третираат како дел од самата машина (па и системските програми). Овој начин на третирање на слоевита структура е искористен во конципирање на виртуелните машини (ВМ). Структурата на виртуелните машини се дефинира на следниов начин: на најниското ниво се наоѓа хардверот, а над хардверот монитор на виртуелни машини, посебен систем кој обезбедува низа од виртуелни машини (точни копии на хардверот). На тие виртуелни машини можат да се инсталираат различни оперативни системи. Мониторот на виртуелните машини ги прима системските повици од корисничките програми, а хардверските операции кои тие оперативни системи ги испраќаат кон своите виртуелни машини и ги реализира во согласност со постоечкиот хардвер.

Кај виртуелните машини, оперативниот систем може да создаде илузија дека еден процес има свој процесор и своја (виртуелна) меморија. Пристапот на виртуелни машини не обезбедува дополнителна функционалност, но дава интерфејс кој е идентичен со хардверот. Секој процес е обезбеден со (виртуелна) копија на компјутерот на кој се работи.

Физичкиот систем ги дели своите ресурси за да се креира виртуелната машина. Делењето на процесорот може создаде илузија дека секој корисник има свој процесор.

Голема тешкотија претставува пристапот на виртуелната машина со диск системите (на пример, проблем со користење на 3 диска за 7 виртуелни машини). Решението е создавање на виртуелни дискови.



Слика 1.14: Организација со виртуелни машини

Користењето на виртуелните машини дава предност при овозможување на комплетна заштита на системските ресурси. Виртуелните машини имаат големо ниво на сигурност каде што секоја машина е комплетно изолирана од останатите, со што се оневозможува појава на сигурносни проблеми. Друго, виртуелните машини дозволуваат развој на системот без да се наруши нормалниот тек на системските операции. Имаат голема популарност за решавање на проблемот на компатибилност на системи. На пример, апликациите напишани за Windows на Intel процесорите, може да се користат на Sun Microsystem машини со креирање на виртуелна Intel машина над Sun-овиот процесор. Притоа, Интеловите инструкции се преведуваат во инструкциски сет познат за процесорот-домаќин.

2. ПРОЦЕСИ

Концептот процес зазема централно место во секој оперативен систем. Тој претставува програма во извршување и сите други механизми и операции кои ги извршува оперативниот систем зависат од овој концепт. Затоа е многу значајно дизајнерите на оперативни системи (студентите) во потполност да го совладаат поимот и да ја разберат улогата на процесите во функционирањето на еден оперативен систем при извршување на најразлични операции.

Основна улога на оперативниот систем е да овозможи извршување на низа од операции зададени од страна на програмерот и да ги презентира резултатите или да предизвика некоја акција. Истовремено системот може да пројави потреба да извршува и помошни операции кои директно се ставени во функција на извршувањето на задачата, односно програмата. На пример, додека се извршува една програма, системот може да врши „истовремено“ читање или пишување на дискот, или да активира или исчитува некои надворешни уреди. Во еден компјутерски систем заснован на еден процесор, едновремено може да се извршува само една програма, иако во претходниот пример изгледа дека едновремено може да се извршуваат и повеќе операции. Во суштина тие операции се извршуваат преку наизменично заземање на процесорот во кратки временски интервали, при што во процесорот секој пат се извршува само една низа на команди од зададената програма или задача.

Ова овозможува да изгледа дека поголем број на програми се извршуваат паралелно, без да се користат повеќе инстанци на хардверски ресурси. Овој концепт се нарекува псевдопаралелизам и е спротивен на вистинскиот хардверски паралелизам, каде при извршување на програмите на повеќе процесори се дели заедничката меморија.

Заради решавањето на поставените предизвици како што е следењето на измената на извршувањето на програмите во процесорот и тешкотиите кои произлегуваат од тоа, со тек на времето во областа на дизајнот на оперативните системи се развил концепт наречен секвенцијални процеси или само процеси. Пред да се претстави концептот на процесите ќе биде дадено појаснување на поимот на јадро, компонента на оперативниот систем чија основна функција е управување со процесите.

2.1. Поим за јадро

Јадрото е основна компонента на секој оперативен систем. Во хиерархискиот модел на оперативниот систем, јадрото се наоѓа најблиску до хардверот на системот и најчесто се надоврзува директно на хардверот и останатите слоеви на оперативниот систем.

Програмите и процесите можат да се извршуваат и на компјутерски систем без јадро во составот на оперативниот систем, доколку програмерот е подготвен да го заобиколи процесот на апстрахирање на хардверот и услугите на оперативниот систем и самостојно да ја напише програма на најниско програмско ниво со користење на команди на машински јазик. Притоа, во процесот на програмирање треба да се земат в предвид сите детали на системот како и начинот на користењето на ресурсите.

Овој начин на програмирање и извршување на програми бил вообичаен на постарите компјутерски системи каде што се изведувало монопрограмирање, а мултипрограмирањето се изведувало со прекинување на извршувањето на една програма преку ресетирање и полнење на друга програма за извршување. Притоа се користеле програмите за откривање на грешки (*англ. debuggers*) и апликации со кои се вршело полнењето на програмата во системот *(англ. loaders)*. Овие алатки биле основа за понатамошниот развој на јадрото на оперативниот систем.

Функцијата на јадрото е управување со процеси, односно воспоставување на околина која овозможува постоење на концептот на секвенцијални процеси, доделување на процесорот на процесите и обезбедување на механизми за интерпроцесна комуникација. Затоа што процесорот е неделив ресурс и може едновремено да се извршува само еден процес, јадрото одредува кој процес и кога ќе го добие процесорот на користење.

За да ја оствари својата функција, потребно е да постојат одредени хардверски компоненти:

1. Механизам на прекини;
2. Механизам на заштита на адресирањето на меморијата;
3. Множество на привилегирани инструкции;
4. Часовник на реално време;

Механизмот на прекини (*англ. interrupts*) обезбедува извршување на управувачката програма (сервисна рутина), односно префрлување на контролата на извршување од корисничката програма на управувачката програма. Бидејќи се прекинува извршувањето на корисничката програма во произволен момент, потребно е да се изведат и дополнителни активности, за да по завршувањето на сервисната рутина, корисничката програма продолжи со извршувањето. По појава на прекинот, механизмот за прекини треба да ја сочува вредноста на програмскиот бројач на корисничката програма и да ја активира сервисната рутина од дадена фиксна локација од меморијата. Сервисната рутина го определува изворот на прекинот и изведува одредени операции соодветни на типот на прекинот.

Заштитниот механизам за адресирање на меморијата оневозможува погрешно адресирање, односно оневозможува еден процес да ги впише своите податоци

во делот на меморијата кој е доделен на друг процес. Овој механизам автоматски го чува интегритетот на процесот и податоците кои се наоѓаат во работната меморија.

Множество на привилегирани инструкции го сочинуваат сите инструкции кои се достапни на оперативниот систем, но не и на корисничката програма. Овие инструкции му овозможуваат на оперативниот систем да ги маскира прекините, да го додели процесорот на други процеси, да пристапи кон заштитените регистри во меморијата, да изврши влезно-излезна операција или да го запре процесорот. При извршувањето на привилегираните инструкции оперативниот систем се наоѓа во системски режим на работа (*англ. supervisory mode*). Корисничката програма не може директно да извршува привилегирани инструкции, туку исклучиво со помош на системски повици. Корисничката програма со помош на системските повици (*англ. system call*) бара од оперативниот систем да изврши привилегирана инструкција, по што тој преминува во системски режим и ја извршува таа инструкција.

Со помош на часовникот за реално време (*англ. real-time clock*) се контролира и следи користењето на ресурсите на системот за сите поединечни процеси. Овој механизам можи да се користи и за распоредување и закажување за извршување на различни задачи (*англ. tasks*).

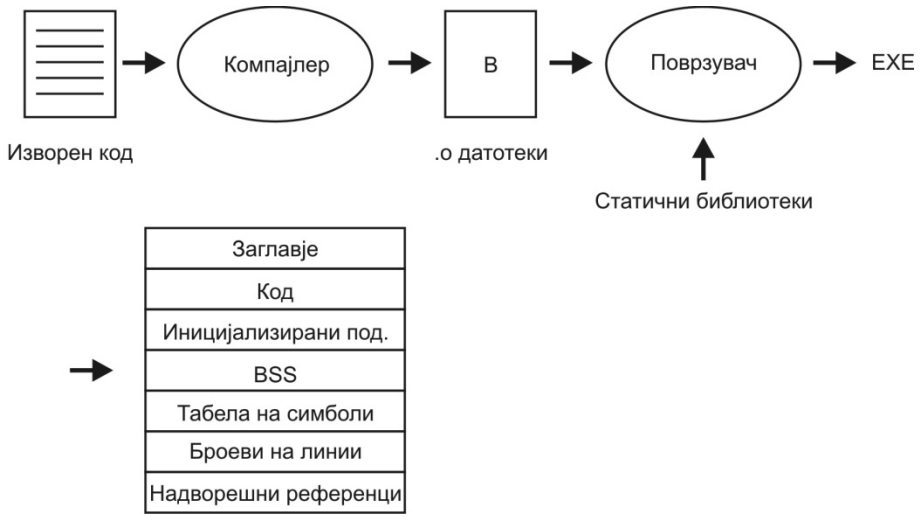
2.2. Поим за програма

Поимот за компјутерска програма може да се однесува на изворен код напишан во некој програмски јазик, или форма на извршна датотека од изворниот код. Извршната датотека претставува датотека чија содржина треба да се изврши од страна на компјутерскиот систем. Во понатамошниот текст за поимот на извршна датотека ќе се користи името „програма“. Програмата содржи бинарна репрезентација на машинските инструкции карактеристични за одреден процесор и дополнителни податоци.

Извршната датотека се добива по изведување на процесот на преведување на програмата од некој симболички програмски јазик – компајлирање или асемблирање, при што се добиваат објектни датотеки кои содржат машински инструкции, заедно со информација за реалокација, програмските симболи (имиња на променливите и функциите) и можно и податоци за дебагирање (Слика 2.1). Со користење на програма за поврзувње (*англ. linker*), се поврзуваат повеќе објектни датотеки и статички библиотеки во една извршна датотека – програма.

Значи, во составот на една програма влегуваат програмскиот код односно низата од машински наредби, податоци меморирани и управувани во меморијата класифицирани во предефинирани променливи, неиницијализирани, односно динамичко алоцирани променливи (преку користење на инструкциите *malloc, new*) која се нарекува и хип (*англ. heap*)

меморија и стек променливи како и информација за динамичко поврзување со надворешни библиотеки (*DLL – библиотеки*) код и податоци што не биле поврзани во крајна извршна датотека.



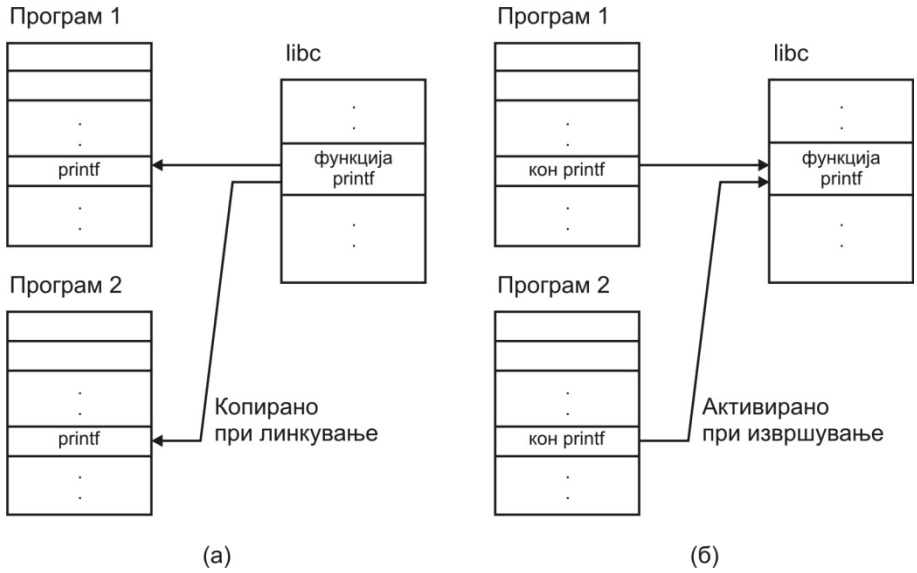
Слика 2.1: Процес на создавање на извршна верзија на програма

Модерните оперативни системи дозволуваат динамичко поврзување што всушност означува одложување на разрешувањето на одредени недефинирани симболи сè до моментот до кога програмата ќе започне да се извршува. Тоа значи дека извршната датотека содржи недефинирани симболи, со листа на објекти и библиотеки кои го овозможуваат нивното дефинирање. Со вчитувањето на програмата се вчитуваат и овие објекти/библиотеки и се изведува нивно поврзување. Статичките библиотеки уште се нарекуваат и архивски, а динамичките – делени библиотеки. При користењето на архивските библиотеки референцата кон симболите (на пример, повик кон функција) се дефинира со вклучувањето на рутината во извршната програма, додека кај користење на делива (динамичка датотека) во програмата постои покажувач кон рутината, која се повикува во времето на извршување на програмата (таа не се вклучува во извршната верзија на програмата).

Овој пристап нуди две предности:

- Често користените библиотеки (како што се стандардните системски датотеки) треба да се складираат на само една локација и не се удвојуваат за секоја извршна датотека.
- Ако се изврши модификација на функција во делена датотека или се изврши нејзината замена, сите програми кои ја користат ќе ја имаат вклучено промената по нивното рестартирање. Додека за програмите

кои ја вклучуваат функцијата преку архивски датотеки и статичко поврзување треба повторно да се изврши процесот на поврзување.



Слика 2.2: Користење на а) архивски и б) деливи библиотеки

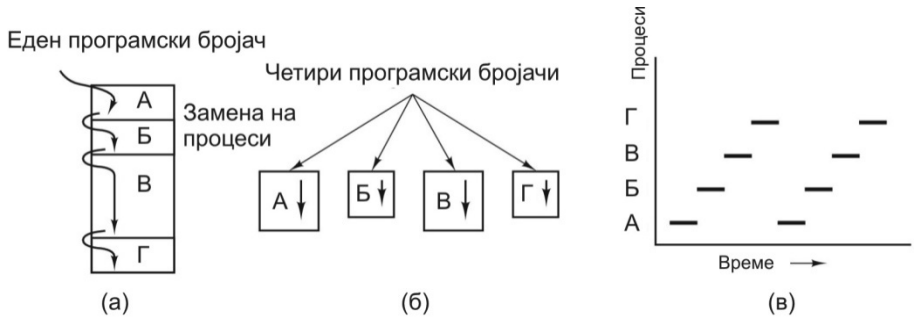
2.3. Поим за процес

Еден сметачки систем кој поседува само еден процесор е способен да извршува само една инструкција во еден момент. Оперативниот систем треба на сите корисници на системот и на сите програми да им обезбеди пристап на процесорот при што не смее да дозволи на никој да го заземе произволно долго време.

Процесот е еден од најважните концепти на оперативниот систем, најосновна апстракција што ја обезбедува оперативниот систем. Целиот извршен софтвер на компјутерот, вклучувајќи го и оперативниот систем е организиран во секвенцијални процеси односно процеси. Процесот е програма или дел од програма која се наоѓа во состојба на извршување. Програмата (извршната датотека) е пасивен објект кој се наоѓа на дискот. Кога програмата ќе се внесе во главната меморија и се додели процесорот за извршување, процесот станува активен објект кој има свои ресурси како што се регистри и меморија.

Концептуално секој процес има сопствен виртуелен процесор, но во реалноста постоечкиот процесор изменува процес по процес при извршувањето. Моделот може да се претстави како множество на процеси кои се извршуваат псевдо-паралелно. Измената на процесите се нарекува и мултипрограмирање како што е објаснето во претходното поглавје. На следната слика е прикажан пример на мултипрограмирање на 4 процеси, секој со сопствена контрола на

извршувањето (сопствен логички програмски бројач) и каде што секој се извршува независно од другите. Постои само еден физички програмски бројач, така што секој пат кога ќе се извршува еден процес се вчитува неговиот логички бројач во физичкиот програмски бројач. При завршувањето на извршувањето од било која причина вредноста на физичкиот бројач се снима во локалниот бројач на процесот во меморијата. На Слика 2.3 може да се види дека сите четири процеси се извршуваат и создаваат прогрес во текот на времето, меѓутоа во еден момент само еден процес се извршува во процесорот.



Слика 2.3: Извршување на процесите, а) со еден и б) повеќе програмски бројачи, в) распределба на процесорот на процесите

Друга карактеристика на процесот е што се врши изолација на контекст за секоја апликација. Тоа значи дека како во дадениот пример четирите корисници кои извршуваа активност (на пример, извршуваат корисничка апликација), ќе бидат претставени со четири различни процеси. Тоа исто така значи дека едната програма, која самостојно ќе се подели на два дела кои работат истовремено, ќе биде претставена како два различни одделни процеси.

Во контекст на еден процес се содржат три основни делови:

- Програмска или текстуална секција, која не се менува (*англ. read only*) и која содржи програмски код.
- Стек секција (*англ. stack section*), која содржи привремени податоци (параметри за процедурата, повратни адреси, локални променливи).
- Секција со податоци (*англ. data section*), која содржи глобални променливи.

Како што текстуалната секција не се менува, повеќето оперативни системи ќе формираат заеднички текстуален сегмент за сите процеси кои настануваат по започнувањето на извршување на една програма. Овој модел е ефикасен бидејќи на тој начин се заштедува меморијата. Притоа секој од новонастанатите процеси ќе го дели текстуалниот дел, но ќе има засебен стек сегмент со податоци. Освен мемориската секција, процесот ги опфаќа и вредностите на останатите важни регистри на процесорот. Процесот ги опфаќа

и влезно-излезните ресурси, кои евентуално ќе ги користи како што се датотеки и разни видови на влезно-излезни уреди.

Разликата помеѓу процесот и програмата е мала, но значајна. Најдобро може да се разјаснат поимите преку пример каде еден готвач за подготовка на роденденска торта користи книга со рецепти. Во овој случај, рецептот преставува програма, односно алгоритам за подготовка на тортата во облик кој е разбирлив за готвачот. Готвачот го претставува процесорот кој го чита рецептот по ред како што е напишан и презема влезни податоци – состојки потребни за тортата. Активностите што ги извршува готвачот како што се читање на рецептот, додавање на состојките, печење на колачот се аналогни на концептот на процес. Во еден момент настанува прекин, каде што доаѓа синот на готвачот и се жали на повреда, каснување од пчела. Готвачот запомнува до каде стасал со читање на рецептот (состојбата на тековниот процес сочувана). Зема книга за прва помош и започнува со читање на тие инструкции (процесорот преминал од еден процес на друг, поприоритетен процес). Секој од овие процеси има своја програма (рецепт за колач наспроти книгата за прва помош). Кога ќе ја санира повредата, готвачот се враќа на правењето на колачот, точно таму каде што застанал во претходната активност.

Клучната идеа е дека процес е некој вид на активност, има програма, влез, излез и состојби. Еден процесор може да се дели меѓу повеќе процеси преку користење на некој алгоритам за распоредување преку кој ќе се определи кога да се запре извршувањето на еден процес и да се опслужи друг.

2.3.1. Претставување на процеси

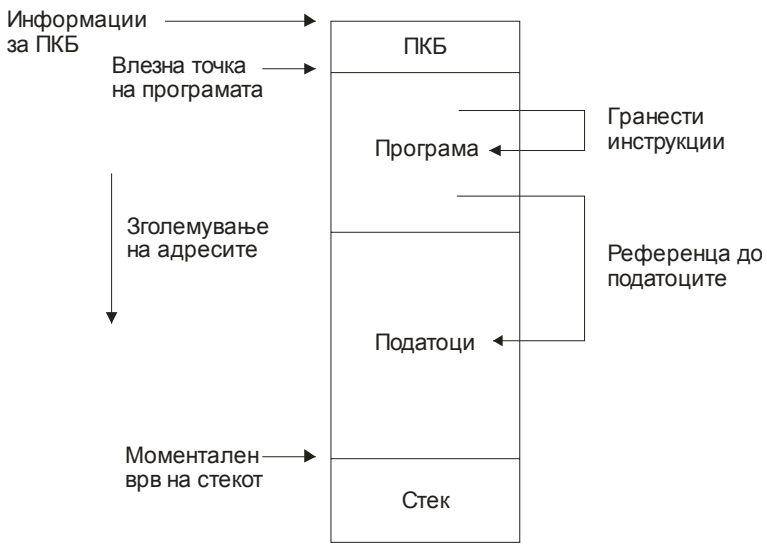
Како што е веќе претходно наведено процес е активност и претставува програма во извршување. Дел од процесот се и податоците кои ја опишуваат таа активност, односно податоците неопходни за управување со процесот. Овие податоци ги генерира и ги користи оперативниот систем односно делот наречен диспечер во контролен блок на процесот (*PCB - Process Control Block*), вектор на состојбата или дескриптор на процесот (*англ. process descriptor*).

На пример, нека се разгледува повеќе-процесен оперативен систем со два активни процеси (П1 и П2). Процесот П1 кој се извршува во еден момент се доведува во состојба на чекање на ресурс. Потоа се извршува процесот П2 кој по некое време исто така ќе дојде во состојба на блокиран. Во меѓувреме се ослободува ресурсот неопходен за извршување на процесот П1, така што процесот П1 продолжува понатаму да се извршува. За да знае оперативниот систем каде треба да продолжи со извршувањето, на секој процес му се додаваат дополнителни информации, односно единствен контролен блок. Контролниот блок е дел од работната меморија т.е. мемориска структура со основни информации за процесот, кој оперативниот систем ги користи за

управување со тој процес. Благодарение на контролниот блок, извршувањето на програмата може да се прекинува и да продолжува повеќе пати.

Во информации од контролниот блок спаѓаат:

- име или единствен идентификатор на процесот (PID);
- контекст (опкружување) на процесот;
- приоритет на процесот;
- информации за меморијата на процесот;
- листа на отворени датотеки;
- статусот на зафатените влезно-излезни ресурси;
- моменталната состојба на процесот.



Слика 2.4: Контекст на процес со процесниот контролен блок

Контекстот на процесот го чинат податоци кои се чуваат во случај на одземање на процесорот, а нив ги создава самиот хардвер: програмскиот бројач, вредностите на регистрите и покажувачи на дел од меморијата каде се наоѓа програмата. Блокот на процесот во кој се чува контекстот уште се нарекува и хардверски контролен блок на процесот или хардверски дескриптор на процесот (*англ. hardware process descriptor*).

Шематски контролниот блок се прикажува како придружен на самиот код т.е. процесот, но во реалниот оперативниот систем никогаш не се наоѓа во истото мемориско подрачје. Во понатамошниот текст, подетално ќе бидат објаснети

начините на имплементација на процеси како и компонентите на блокот на контрола на процесот.

2.3.2. Креирање на процеси

Еден процес може да се креира на различни начини, во еден едноставен систем може да се случи при вклучувањето сите потребни процеси да бидат присутни во меморијата и да се активни, нивното креирање е предизвикано од самата иницијализација на системот. Во зависност од намената процесите може да се креираат и терминираат и при изведувањето на операцијата, односно извршувањето на програмата.

Постојат 4 основни настани кои доведуваат до создавање на процеси:

1. Иницијализација на системот
2. Создавање на процес преку системскиот повик за креирање на процес
3. Барање на корисникот за создавање на процес
4. Започнување на задача за групна обработка (*англ. batch*).

При иницијализација на системот (*англ. boot*) се креираат одреден број на процеси кои можат да работат во преден план и да вршат интеракција со корисникот или да се извршуваат во позадина и повремено да се активираат и да извршуваат одредени задачи.

Покрај овие процеси кои се создаваат при подигањето на системот, може да се создадат и нови процеси преку издавање на системски повици за креирање на еден или повеќе нови процеси како помош при извршувањето на задачата на процесот – креатор. Ова е особено корисно кога задачата која треба да ја изврши процесот може да се раздели на помали задачи кои може да се извршуваат независно едни од други, ако системот е повеќе-процесорски секоја од тие може да се извршува засебно на одделни процесори и задачата да се изврши побрзо. Во интерактивните системи, корисниците може да стартуваат програма преку пишување на команда или преку активирање на икона. И двата начини создаваат нов процес преку која се извршува програмата.

И во последниот случај каде што се создаваат процесите се однесува на „batch“ системите за секвенционална обработка на задачи кои се наоѓаат во големите системи со централни компјутери - „*mainframe*“. Тука корисниците ги приложуваат своите програми во системот. Кога системот ќе увиди дека има слободни ресурси за стартување на нов процес, избира програма од влезната редица на чекање и креира нов процес и ја извршува програмата.

Во сите овие случаи нов процес се создава од страна на постоечки процес преку извршување на системски повик за креирање на процес. Тој процес може да биде, постоечки кориснички процес, системски процес повикан од

тастатурата или глувчето или управувач со *"batch"* процеси. Системскиот повик му кажува на оперативниот систем да креира нов процес и покажува директно или индиректно која програма го иницирала. Односот процес-процес родител и дете може да се најде во следниве релации:

- Процесите родител и дете ги делат сите ресурси;
- Процесите родител и дете ги делат ресурсите на родителскиот процес;
- Процесите родител и дете не ги делат ресурсите.

Тоа значи дека по креирањето, процесот-дете може да побара од оперативниот систем нови ресурси, дел од ресурсите на родителскиот процес или сите ресурси кои припаѓаат на родителот. Од ресурсите, секој процес за својата работа ги користи регистрите на процесорот, мемориските секции, датотеките и влезно-излезните уреди. Особено е чувствителен меморискиот адресен простор, за чие користење се применуваат следните техники:

- Процес - дете го копира адресниот простор на родителот;
- Адресниот простор на детето се генерира (преобразува) според програмата која се полни во адресниот простор.

Според начинот на извршување, процесот со својот родител може да се најде во следниве релации:

- Процесот родител продолжува да се извршува независно и конкурентно со детето - процес.
- Процесот родител се блокира и чека додека процесот дете не заврши со своите активности.

На пример, во оперативниот систем UNIX при подигањето на системот се создава првиот процес (*sysproc*), потоа се создава и вториот процес наречен *init* кој понатаму ги креира сите програми наречени *getty* кои го создаваат и прикажуваат промптот и процесот на логирање и ако логирањето е успешно ќе се повика и корисничкиот посредник - *shell*. Иницијалниот процес не е предвидено да заврши, туку се терминира само преку интервенција од страна на корисникот.

Во UNIX системите постои само еден системски повик за создавање на нов процес – *fork*, кој создава идентична копија на процесот кој го извршил системскиот повик. И процесот родител и процесот дете имаат потполно иста околина на извршување, меморија и отворени датотеки. Обично процесот дете ја извршува наредбата „*execve*" или сличен системски повик за промена на мемориската содржина (машинските инструкции) за да започне со извршување на нова програма. Во Windows постои Win32 системски повик „*CreateProcess*" кој истовремено и креира процес и ја полни мемориската мапа со нова програма.

На UNIX системите, при креирањето секој процес добива универзален идентификатор, PID (*англ. process identificator*). Процесот добива копија од адресниот простор на родителот. Со тоа се овозможува лесна интерпроцесна комуникација. И двата процеси (и родителот и детето) продолжуваат со своето извршување по повикувањето на системскиот повик *fork*, но притоа секој од нив добива различен PID: детето добива вредност PID=0, а на родителот вредноста на PID е различна од 0.

Освен доделувањето на интерните PID броеви, наредбата *fork* го копира адресниот простор на родителот и го доделува на детето. При системскиот повик *fork*, адресниот простор на двата процеси е наполнет со иста програма (код), но идентификаторите на процесите се различни. Процесот дете потоа извршува системски повик *execve*, кој впишува нова програма во адресниот простор доделен на детето и го извршува. Процесот родител извршува системски повик *wait*, со што се блокира додека процесот дете не ги заврши своите активности и не врати статус на родителскиот процес.

Следната програма во јазикот C, го илустрира начинот на создавање на процес под UNIX-систем, а примерот одговара на случајот на командниот интерпретер *bin/sh* кој ја извршува командата *ls*. Истиот код ќе го извршува и процесот родителот и процесот дете. Родителот прави нов процес со помош на системскиот повик *fork*, кој на родителот и на детето му доделува различен PID. Процесот дете чиј PID е 0, извршува системски повик *execve* кој во доделуваниот адресен простор ќе ја вчита програма *ls* и ќе го изврши. Процесот родител чиј PID е различен од 0, чека да командата *ls* заврши (системски повик *wait*). Кога процесот дете ќе заврши со активностите, родителот ќе излезе од системскиот повик *wait*.

```
/* fork - drug proces*/
       pid = fork ();

       if (pid = = 0 )
       {                                    /*proces dete*/
              execve("/bin/ls", "ls", NULL);
       }

       else   /*proces roditel*/
       {
              /*roditelot ceka na zavrsuvanje na deteto */
              wait(NULL);
              printf("deteto se izvrsilo");
              exit (0);
       }
}
```

Слика 2.5: Пример за користење на системскиот повик *fork* за креирање на нов процес (извршување на *ls* командата).

И кај Windows и UNIX, по креирањето на процесот и процесот родител и детето имаат свој сопствен изолиран адресен простор. Ако било кој процес изврши измена на содржината на сопствениот мемориски простор тоа нема да биде видливо за другиот. Во UNIX иницијалниот адресен простор е копија на просторот на родителот, но тоа претставува потполно засебен адресен простор, не постои заедничка делена меморија. Можно е новосоздадениот процес да дели некои други ресурси од својот родител како што се отворените датотеки. Кај Windows адресниот простор кај родителот и кај детето процес од самиот почеток се потполно различни.

2.3.3. Завршување на процеси

По креирањето на процесот, тој ја извршува својата задача одредена според програмскиот код. Процесот завршува порано или подоцна заради некоја од следниве состојби:

- нормален излез (доброволно);
- излез со грешка (доброволно);
- фатална грешка (неволно);
- убиен (*англ. killed*) од друг процес (неволно).

Поголемиот дел од процесите завршуваат затоа што си ја завршиле работата. На пример, еден компајлер по завршувањето на операцијата компајлирање издава системски повик кон оперативниот систем дека завршил со работата. Повикот е *exit* кај UNIX, или *ExitProcess* кај Windows.

Втората причина за терминација на процес е кога процесот при неговото извршување ќе открие грешка. На пример, ако корисникот издаде команда за отворање на директориум:

```
$ cd user
```

а таков директориум не постои, процесот јавува грешка и се терминира. Кај графичките кориснички интерфејси на корисникот му се презентира дијалог прозорец каде му се нуди избор повторно да се обиде (во зависност од видот на грешката).

Третата причина за завршување на процесот е грешка предизвикана од самиот код, грешка при програмирањето. Некои од примерите кои може да се наведат се, извршување на илегална инструкција, референца кон непостоечка мемориска локација или делење со нула.

Четвртата причина за престанување на работа на процес е кога еден процес извршува системски повик кон оперативниот систем за терминација на други процеси. Кај UNIX системскиот повик е *kill*, а соодветната Win32 функција е *TerminateProcess*. И во двата случаи процесот треба да има авторизација за издавање на ваков системски повик.

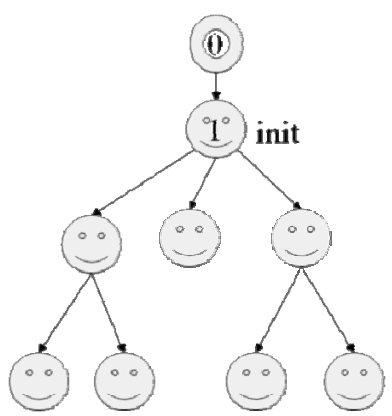
По нормалното завршување на процесот-дете, тој враќа излезни податоци на родителскиот процес со помош на системскиот повик *wait*. Потоа се ослободуваат сите ресурси кои ги зазел процесот, како што се физичката и виртуелната меморија, датотеките и бафери на влезно-излезните уреди.

Како што е наведено претходно еден процес може да предизвика и присилно завршување на работата на друг процес. Притоа родителот може да го прекине извршувањето на процесот што го создал и тоа обично се прави од следниве причини:

- Процес дете при користење на користи некој ресурс (меморија, системско време) ја надминал предвидената квота;
- Активностите што процесот дете ги извршувал повеќе не се потребни;
- Процес родител ја завршил активноста пред своите децата, што се смета за ненормална ситуација која оперативниот систем не ја дозволува и сите деца процеси треба присилно да се уништат, (*англ. cascade termination*).

2.3.4. Процесна хиерархија

Кај некои системи кога еден процес создава друг процес, процесот родител и процесот дете продолжуваат да бидат поврзани на некој начин. Самиот процес-дете може самостојно да креира повеќе процеси со што се креира хиерархија на процеси, при што секој процес има само еден родител.



Слика 2.6: Процесна хиерархија

Кај UNIX, процесот и сите негови деца и идните наследници заедно формираат процесна група. Кога корисникот испраќа сигнал преку тастатурата, сигналот се испорачува кон сите членови на групата кои се придружени кон тастатурата. Секој процес може да го преземе сигналот и соодветно да реагира или да го игнорира. Спротивно на тоа кај Windows не постои концепт на процесна хиерархија и сите процеси се еднакви.

2.4. Состојби на процес

Иако секој процес е независен ентитет, со свој сопствен бројач и внатрешна состојба, постои потреба од нивна меѓусебна интеракција. Еден процес може да генерира излезни податоци кои што на друг процес може да му служат како влезни податоци. На пример командата:

```
$ cat file1 file2 file2 | grep info
```

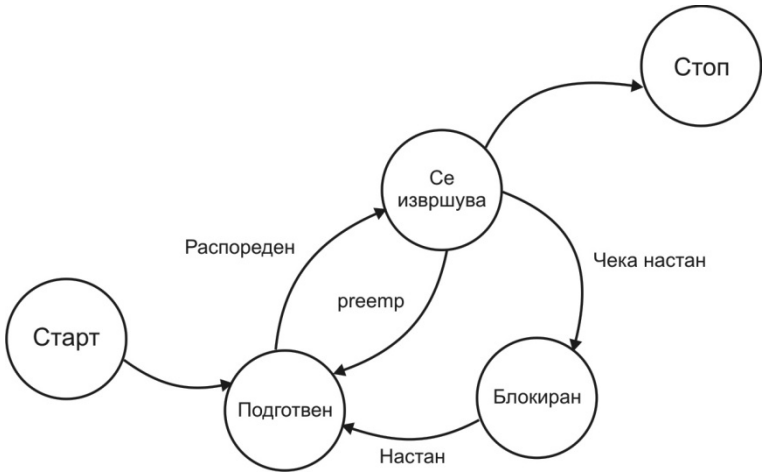
првиот процес ја извршува наредбата *cat* и ги спојува трите датотеки во една, вториот процес кој ја извршува наредбата *grep* ги селектира сите линии кои го содржат зборот "*info*". Во зависност од релативните брзини на двата процеси може да се случи процесот *grep* да биде подготвен, но да нема влезни податоци и ќе мора да чека во состојба блокиран (*wait*). Кога процесот се блокира тоа обично се случува затоа што не може да продолжи со работата од причина што чека на влезни податоци или што оперативниот систем одлучил да го прекине процесот и да го додели процесорот на друг процес. Овие две состојби се целосно различни, во првиот случај суспендирањето на процесот е заради самиот проблем (не може да се процесира командата која не е отчукана на тастатурата), а вториот случај е чисто техничка операција од страна на самиот систем.

Сите процеси кои влегуваат во компјутерскиот систем минуваат низ неколку состојби при нивното извршување. Состојбата на процесот опишува што се случува со процесот во еден даден момент. Преминувањето на процесот од една во друга состојба се извршува од страна на оперативниот систем. Процесот може да се најде во неколку состојби (пет или седум), а следните три се најзначајни:

- Се извршува (го користи ЦПЕ во тој момент) - RUN
- Подготвен (извршен, привремено чека на процесорот) - READY
- Блокиран (не може да продолжи да се извршува се додека не се случи надворешен настан) - WAIT

Првите две состојби се слични и во двата случаи процесите се подготвени за извршување, само што нема достапен процесор за да му се додели на процесорот. А третата состојба се разликува од првите две во тоа што е

оневозможено извршувањето на процесот, иако е можно да има достапен процесор во моментот.



Слика 2.7: Состојба на процеси

При управувањето со процесите, диспечерот на оперативниот систем ја следи и испитува состојбата во контролните блокови на процесите, врз основа на тоа процесите можат да се додаваат во ред за чекање на заземање на процесорот. Додавањето во ред на чекање е симболичко - процесот физички не се поместува од една на друга локација, туку се менуваат само податоците во контролните блокови и покажувачите со чија помош се формираат редови. Секој процес започнува со работата во состојба START, која претставува еден вид на подготовка, по што се доведува во состојба READY односно во состојба на чекање на доделување на процесот во процесорскиот ред. Ова е прва транзиција во дијаграмот на состојбите.

По тоа процесот поминува низ дополнителни транзиции помеѓу состојбите:

1. Процесот се блокира - чека на влез (*wait* премин RUN-WAIT)
2. Распоредувачот избира друг процес за извршување (*preempt* RUN - READY)
3. Распоредувачот го бира тој процес за извршување (*schedule* READY-RUN)
4. Влезот станува достапен (*event done* WAIT-READY)

RUN-WAIT. Одземањето на процесорот од процесот доколку ресурсот кој е потребен за извршувањето на процесот е заземен. Оваа транзиција е возможна и во едно-процесни и во повеќе-процесни оперативни системи. До оваа транзиција може да дојде и кога процесот чека резултат од операцијата (извршување) која се извршува од друг процес. До ова транзиција доаѓа и ако

процесот чека одреден момент кој е однапред програмиран и во кој може да продолжи извршувањето.

RUN-READY. Одземањето на процесорот по истекувањето на периодот во кој му е доделен. Ако оперативниот систем е со пред-испразнување (*англ. pre-emptive*), процесорот може да се одземе и при појава на процес со поголем приоритет и овој премин е возможен само во повеќе процесни системи.

READY-RUN. Преминот се случува кога процесорот му се доделува на процес чиј е ред за извршување. Процесот преминува од состојбата на чекање во состојбата на извршување по прекинувањето на претходниот процес и започнување на извршување на нов процес.

WAIT-READY. Процесот се враќа на крајот од процесорската редица на чекање по ослободувањето на ресурсот кој е неопходен за неговата работа. Транзицијата WAIT-RUN не е можна во повеќе-процесорските оперативни системи. Во таквите системи, процесот секогаш се доведува во состојба READY.

Во состојба RUN во еден момент можат да се најдат најмногу онолку процеси колку што има процесори во системот. Процесите кои се наоѓаат во состојба RUN се тековни процеси. Во повеќе-процесни системи процесот не може да се задржи во состојбата на извршување (RUN) до своето завршување. Затоа процесорот се доделува на процесот одредено време, според даден алгоритам на распоредување. На тој начин се спречува еден процес да црпи ресурси од системот, додека пак други процеси чекаат во редица на неговото завршување. Доколку оперативниот систем овозможува пред-испразнување, процесорот може да се одземе и во случај да наиде процес со поголем приоритет со што се обезбедува навремено извршување на критични процеси.

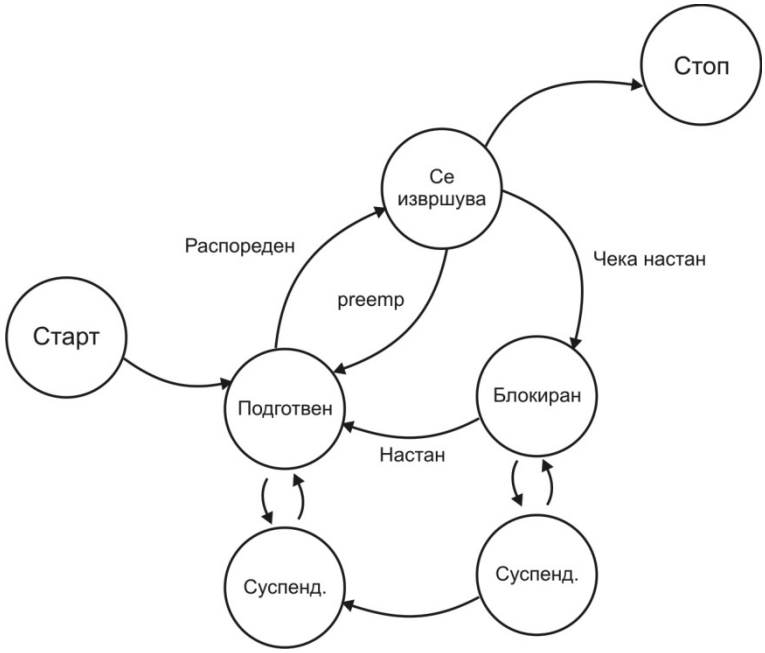
2.4.1. Проширен дијаграм на состојби на процес

Во некој оперативни системи постои можност за привремено прекинување на извршувањето на процесот. Процесот чие извршување е привремено прекинато е отстранет (суспендиран) и престанува да се натпреварува за ресурси. Овој модел е карактеристичен за оперативниот систем UNIX. Основната претпоставка е дека корисникот кој иницира процес има право привремено да го запре неговото извршување.

Исто така корисникот може во одреден момент да продолжи со извршување на суспендираниот процес. Самиот оперативен систем може, доколку постои потреба одреден број процеси да ги донесе во состојба на суспендирање со што се спречува застој и појава на ефект на заситување заради поголемо количество на кеширани податоци. Суспендирањето на процес може да се појави и како последица на *swap* просторот (просторот на дискот за привремено сместување на неактивни процеси заради ослободувањата што поголема количина на RAM меморијата за извршување на други процеси).

Извршувањето на процесот може да се прекине само во состојби WAIT и READY. На тој начин, се формираат две дополнителни состојби:

- SUSPENDED-READY- во оваа состојба процесот пристигнува доколку е суспендиран во состојба на чекање на процесорот.
- SUSPEND-WAIT- во оваа состојба процесот пристигнува доколку е суспендиран во состојба на чекање на ресурс.



Слика 2.8: Проширен дијаграм на состојби на процеси

READY - SUSPENDED READY. Доделување на процесот од состојба на подготвен во суспендирана состојба. Процесот поминува низ ова транзиција ако на системот постојат премногу процеси во состојба READY, треба да се избегне застојот или корисникот експлицитно го суспендира процесот.

WAIT - SUSPENDED WAIT. Доделување на процесот во суспендирана состојба од состојба на чекање WAIT.

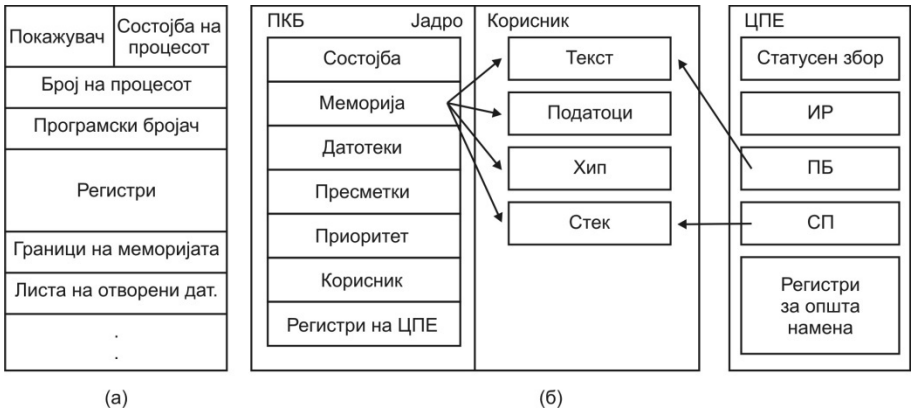
SUSPENDED WAIT - SUSPENDED READY. Ресурсот кој е неопходен за извршувањето на процесот станал достапен, но процесот и понатаму се наоѓа во состојба суспендиран.

SUSPENDED READY - READY. Процесот е одмрзнат и се реди на крајот на редицата на чекање на процесорот. Оваа транзиција е можна само на експлицитно барање на корисникот.

SUSPENDED WAIT - WAIT. Процесот е одмрзнат, но ресурсот кој е неопходен за негово извршување не е ослободен. Оваа транзиција е можна само на експлицитно барање на корисникот.

2.5. Имплементација на процеси

За да се имплементира претставениот модел на процеси, оперативниот систем треба да одржува табела (низа од податочни структури), наречена табела на процеси. За секој процес кој е активен на системот се наоѓа еден влез во табелата. Овие влезови се нарекуваат контролни блокови на процесите (*PCB - Process Control Blocks*) и ги содржат информациите за состојбата на процесот, неговиот програмски бројач, покажувачот на почетокот на стекот, доделувањето на меморијата и статусот на отворените датотеки, следењето на искористувањето на ресурсите (пред сè процесорското време) и сите други информации потребни при измена на состојбата на процесот од извршување (RUN) во подготвен (READY) или блокиран (WAIT), за да може да биде повторно доведен во состојбата на извршување како и да не бил прекинат. Со користењето на PCB се овозможува ефикасен и централизиран пристап до сите информации во врска со процесот. Контекст на извршување на еден процес ги вклучува податоците содржани во PCB структурата. Редиците на чекање за доделување на процесорот всушност ги содржат покажувачите кон PCB податочните структури.



Слика 2.9: а) Контекстот на процес, б) содржина на регистрите во состав на контекстот на процесот за управување со меморијата

Контролен блок на процесот (PCB), содржи повеќе различни информации:

- Информација за распоредување на процесите, покажувачи за редовите на чекање, приоритет на процеси;

- Информации за управување со меморија, опис на локацијата на типичните мемориски сегменти од адресниот простор на процесот.
- Регистри на процесорот, нивната содржина се чува за да се запази состојбата при прекин за да може процесот да продолжи каде што застанал:
  - *Process Status Word* (PSW) – регистарот, го опишува режимот на работа на процесорот, статусната информација со последните резултати при пресметките, ниво на прекини;
  - Инструкциски регистар (IR), кодот на тековната инструкција што се извршува;
  - Програмски бројач (PC), ја содржи адресата на следната машинска инструкција што треба да се изврши за дадениот процес;
  - Stack покажувач (SP), локација на тековната позиција на стекот;
  - Регистри за општа намена;
- Адресен простор:
  - Текстуална секција, го содржи програмскиот код на процесот;
  - Податоци, предефинирани податоци (иницијализирани при компајлирањето);
  - Heap, динамичко алоцирани податоци;
  - Стек, дел од меморијата за поддршка на прекините и на системските повици.
- Околина:
  - Надворешни ентитети, како што се отворени терминални сесии, отворени датотеки;
  - Комуникациски канали, воспоставени локално и со други системи.

Измена на контекстот (*англ. context switch*) се изведува кога ќе се појави потреба за доделување на процесорот на друг процес, при појава на прекин, системски повик или промена на режимот на работа на процесорот (заштитен начин). Иако за измена на контекстот се користат брзи и кратки асемблерски рутини тоа е сепак сложена и скапа операција во поглед на дополнително трошење на процесорско време. При измената јадрото на оперативниот систем го снима контекстот на стариот процес во неговиот PCB и го вчитува контекстот на новиот процес кој треба да се извршува. Претставува битен фактор за ефикасноста на оперативниот систем и цената на операцијата се зголемува со забрзувањето на процесорот, кај посложените оперативни системи треба да се изведат повеќе операции при измената на контекстот на процесите.

Извршувањето на повеќе секвенцијални процеси на систем со еден процесор и повеќе влезно-излезни уреди се изведува преку користење на претходно споменатите механизми (прекини, заштита на адресниот простор, привилегирани инструкции, часовник на реално време). Со секоја класа на влезни-излезни уреди (дискови, терминали, модеми и т.н.) се придружува мемориска локација (обично на крајот од меморискиот простор) кој е наречен вектор на прекин (*англ. interrupt vector*). Тој ја содржи адресата на сервисната потпрограма за одреден тип на прекини.

Нека се претпостави дека еден кориснички процес се извршува сè до моментот кога се појавува прекин предизвикан од дискот. Со помош на хардверот на механизмот на прекинот, програмскиот бројач, програмскиот статусен збор (*PSW - Program Status Word*) и најверојатно еден или повеќе регистри на процесот се поставуваат на тековната позиција на стекот. Потоа системот скока на адресата одредена со векторот на прекините за дискот. Оттука контролата ја презема софтверот, односно сервисната рутина. Сите прекини започнуваат со снимање на содржината на регистрите, обично во влезот на табелата на процесите за дадениот процес. Овој процес се изведува со помош на асемблерски рутини кои се идентични за сите прекини без разлика за нивното потекло.

По завршувањето на работата на рутината, се повикува процедурата напишана во виш програмски јазик (обично во C) за завршувањето на работата за одреден тип на прекин. По завршувањето на прекинот и сервисната рутина, контролата се предава на распоредувачот на процесите кој определува кој од наредните процеси ќе биде ставен во состојба на извршување. Следува пак префрлање на контролата на асемблерски рутини за пренесувањето на содржината на регистрите и меморискиите мапи за новиот процес и започнување на неговото извршување.

Целиот процес може да се опише со следните чекори:

- Хардверот го префрла програмскиот бројач на стекот;
- Хардверот вчитува нов програмски бројач за прекинот;
- Асемблерска рутина ги снима регистрите;
- Асемблерска рутина поставува нов стек;
- Се извршува сервисна рутина на прекинот (го чита влезниот бафер);
- Распоредувачот одлучува кој процес ќе го извршува понатаму;
- Асемблерска рутина поставува нов процес.

2.6. Интерпроцесна комуникација

Рутините за интерпроцесна комуникација треба да се имплементираат во јадрото на оперативниот систем како би биле достапни за сите процеси и како би имале директен пристап на распоредувачот на процеси. Интерпроцесната комуникација може да се оствари на повеќе начини: семафорски техники, испраќање на пораки, користење на делива меморија (бафери).

2.6.1. Независни и кооперативни процеси

Во однос на меѓусебна зависност во текот на извршувањето, процесите можат да се поделат на:

- независни процеси и
- кооперативни процеси.

Процесот е независен доколку не влијае директно на извршувањето на другите процеси и доколку на неговото извршување директно не влијае друг процес. Независни се оние процеси кои не делат никакви податоци со други процеси. Кооперативните процеси се оние кои влијаат едни на други, а тоа се сите процеси кои делат податоци или било какви ресурси. Кооперативните процеси нудат повеќе погодности заради следниве причини:

- Делење на податоци. Бидејќи повеќето процеси можат да бидат заинтересирани за исти податоци, на пример за исти датотеки, потребно е да се обезбеди околина која ќе овозможи правилен конкурентен пристап кон таквите податоци.
- Забрзување на работата. доколку би се поделила некоја работа на повеќе делови и ако би тие делови се извршувале конкурентно, ќе се изврши подобрување на перформансите. Ова има смисла само во повеќе-процесорски системи (како што е случај со RAID системот).
- Модуларност. Системските функции можат модуларно да се поделат на одвоени процеси и нишки.
- Погодност. Корисникот може истовремено да извршува повеќе работи.

Конкурентно извршување на кооперативните процеси бара механизам за комуникација помеѓу процесите и синхронизација на нивните акции. Проблемот со синхронизацијата можи да се објасни на примерот со произведувач-потрошувач (*англ. producer-consumer*) кој претставува заедничка парадигма на кооперативните процеси.

Производителот е процес кои произведува информации, а потрошувачот е процес кој троши односно ги користи тие информации. Тие два процеси можат да работат конкурентно, доколку постои привремена меморија (бафер) која ќе го полни произведувачот а ќе го празни потрошувачот.

Правила за синхронизација се дека потрошувачот не може ништо да земе од баферот доколку произведувачот на ставил нешто во него.

Во идеален случај, баферот има бесконечен капацитет (*англ. unbounder buffer*) односно има бесконечен број на места и никогаш не може да се наполни. Капацитетот на реалниот бафер е ограничен на N елементи, односно податоци (*англ. bound buffer*). Со ограничувањето на капацитетот се воведува ново правило за синхронизација кое вели дека доколку баферот е полн, произведувачот не може постави информација сé додека потрошувачот не земе нешто од баферот. Баферот може да се реализира преку интерпроцесна комуникација или преку делена меморија, прикажана на следниот пример.

Произведувачот и потрошувачот ги делат следните променливи :

```
#define BUFFER_SIZE N
typedef struct
{
       . . .
} item;

item buffer [BUFFER_SIZE];
int in = 0;          //прва слободна позиција на баферот
int out = 0;         //прва полна позиција на баферот
```

Деливиот бафер е реализирана како циркуларен кој се состои од *N* елементи *[0, N-1]* со два покажувачи *in* и *out*. Покажувачите *in* и *out* го определуваат првото слободно и прво зафатено место во баферот. Баферот е празен кога *in=out*, а полн кога *(in+1) mod N=out*. Даден е програмскиот код што одговара за процесите на произведувачите и потрошувачките. За произведувачите:

```
item nextProduced;
 int in = 0;         //прва слободна позиција на баферот
 int out = 0;        //прва полна позиција на баферот
while (1)
       {
       while (((in + 1) % BUFFER_SIZE) == out)

              ; /* баферот е полн, не прави ништо */

       buffer[in] = nextProduced;

       in = (in + 1) % BUFFER_SIZE;

       }
```

За потрошувачите :

```
item nextConsumed;
       int in = 0;          //прва слободна позиција на баферот
       int out = 0;         //прва полна позиција на баферот
              while (1)
              {

              while (in == out)          //празен
                     ; /* не прави ништо */

              nextConsumed = buffer[out];

              out = (out + 1) % BUFFER_SIZE;
              }
```

Решението е коректно само во случај кога во баферот можат да се најдат најмногу *N-1* податоци. За бафер каде можат да се најдат сите *N* податоци треба да се обезбеди дополнителна синхронизација на процесите.

2.7. Модел на нишки (Threads)

Кај оперативните систем, секој процес има сопствен адресен простор и една нишка на контрола. Постојат ситуации каде што е пожелно да се има повеќе нишки на контрола во ист адресен простор и кои се извршуваат квази-паралелно, како да се посебни процеси. Моделот на процеси досега е објаснет на два независни концепти: групирање на ресурси и извршување, но понекогаш е потребно да се разделат и тука се користат нишките (*англ. threads*).

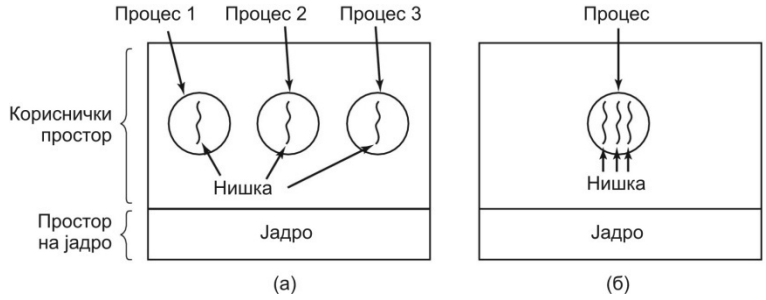
Еден начин на гледање на процесите е начинот на групирање на поврзаните ресурси, процесот има адресен простор кој содржи текст и податоци како и други ресурси. Овие ресурси може да вклучуваат отворени датотеки, процеси – деца и други информации. Ако се здружат во форма на процес може полесно да се управуваат. Другиот концепт е дека еден процес има една нишка на извршување, која скратено се нарекува само нишка. Нишката има програмски бројач кој води сметка која инструкција да ја изврши, користи и регистри каде се наоѓаат работните променливи, поседува и стек.

Нишките и процесите се различни концепти и треба засебно да се разгледуваат. Процесите се користат за групирање на ресурси, додека нишките се ентитети кои се распоредуваат за извршување во процесорот. Нишките овозможуваат повеќекратни извршувања во иста околина на процесот. Повеќе нишки кои се извршуваат во еден процес е аналогно на повеќе процеси кои се извршуваат на еден систем. Во последниот пример, процесите ги делат адресниот простор, отворени датотеки и други ресурси. Затоа што процесите имаат големи сличности со процесите обично се нарекуваат и како „лесни

процеси“. На Слика 2.10 се дадени три обични процеси, секој со сопствен адресен простор и единична нишка на контрола. На втората слика може да се види еден процес со три нишки на контрола. И во двата случаи постојат три нишки додека во последниов случај трите го делат истиот адресен простор.

Со измената на извршување на процесите во процесорот, еден повеќе-процесен систем дава илузија на истовремено извршување на повеќе процеси, со измената на извршување на нишките во еден процес се добива илузија на истовремено извршување на нишките иако на процесор со помали перформанси (ако имаме три нишки на секоја од нив отпаѓа една третина на процесорското време).

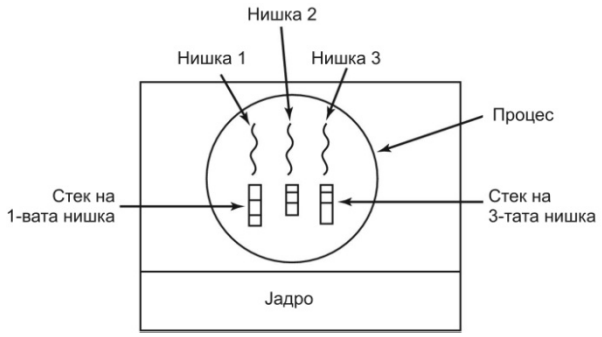
Нишките меѓу себе не се потполно независни како што се процесите, туку делат ист адресен простор и други ресурси и една нишка може да влијае врз извршување на другите со измена на заедничките ресурси. Заштита од оваа ситуација не е можна ниту пак потребна, затоа што нишките припаѓаат на еден процес од еден корисник и се претпоставува дека нишките се програмирани за соработка а не за натпревар за ресурси.



Слика 2.10: а) Повеќе процеси, б) повеќе нишки во состав на еден процес

Како и кај процесите, нишките може да се најдат во една од трите состојби, нишката може да се извршува, да биде блокирана, подготвена или завршена. Нишката која го има зафатено процесорот е моментно активна, онаа која чека на некој надворешен настан или акција од друга нишка е блокирана, подготвената пак чека на доделување на процесорот. Премините помеѓу овие состојби се потполно исти како и кај процесите.

Треба да се сфати дека секоја нишка има сопствен стек затоа што извршува различни делови од програмата, повикува различни процедури и историјата на извршувањето на стекот е различна за нишките. Нишките може со повикување на соодветни системски повици да предизвикаат создавање на нови нишки кои може да бидат поставени во хиерархиски однос или пак, што е почест случај да бидат меѓусебно еднакви.



Слика 2.11: Секоја нишка си има сопствен стек

По завршувањето на работата нишките со повикување на системски повик се уништуваат и ги снемува од адресниот простор и веќе не се распоредуваат за извршување во процесорот. Во работата нишките се однесуваат во голема мерка како процеси, но со разлика дека тие кога ќе го заземат процесорот не постои механизам за прекинување, туку тие сами може доброволно да го отстапат на процесорот на другите нишки.

2.7.1. Користење на нишки

Главната причина за користење на нишките е тоа што при извршување на процес постои потреба од изведување на повеќе паралелни активности, каде што некои од овие може да се блокираат од време на време. Раздвојувањето на такви апликации на повеќе секвенцијални нишки кои се извршуваат квази-паралелно (како и процесите) го упростува програмскиот модел.

Нишките на процесите претставуваат паралелни ентитети кои го делат адресниот простор и оваа особина е суштествена за некои апликации каде што не може да се користат повеќе процеси со засебни адресни простори.

Друга причина за користењето на нишките е економичноста која што се отсликува во користење на делен простор и ресурси и заштеда на време кое има значително влијание на перформансите.

Како што е речено, нишките ја делат меморијата и сите останати ресурси кои припаѓаат на ист процес. Делењето на програмскиот код е корисно, затоа што овозможува една апликација да има одреден број на нишки кои се извршуваат користејќи ист дел на меморијата, а не различни делови на меморијата во која се наоѓа кодот на нишката.

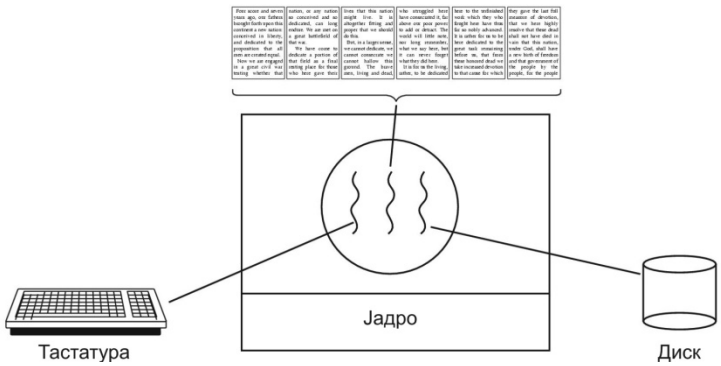
Нишките се создаваат многу побрзо од процесот затоа што користат веќе постоечки ресурси и префрлањето помеѓу нишките исто така е побрзо од префрлањето на контекстот помеѓу процесите. Создавањето на процесите на системот може да биде и до 100-тина пати побавно од креирањето на нишките, како на пример креирањето на нишките на системот Solaris 2 е 30 пати

побавно од изработката на нишки, додека пак префрлувањето на контекстот е 5-пати побавно.

Понатаму, нишките овозможуваат подобро искористување на повеќе-процесорската архитектура. Било која нишка може истовремено да се извршува со другите на различни процесори. Доколку имаме само еден процес во системот, едно-нишната концепција дозволува извршување на тој процес само на еден процесор, додека пак другите процесори би останале неискористени.

Наједноставен начин да се согледа значајноста на нишките е да се презентираат преку примери. Како прв пример, да се претпостави апликација за уредување на текст, каде што текстот се уредува и прикажува на екранот таков каков што ќе изгледа кога ќе се направи печатена верзија.

Ако се претпостави дека корисникот избрише една реченица од страницата 1 од документ кој има 800 страници. И истовремено сака да го види изгледот на 600-тата страница при што на апликацијата и задава команда да отиде на неа. Програмата мора првин да изврши реформатирање на документот од почетокот до крајот затоа што не е познато како ќе изгледа страницата по измената на првата страница, а потоа корисникот може да зададе команда. Може да има поголемо временско задоцнување при прикажувањето на 600-тата страница, што доведува до некомфорно користење на апликацијата и незадоволство на корисникот.



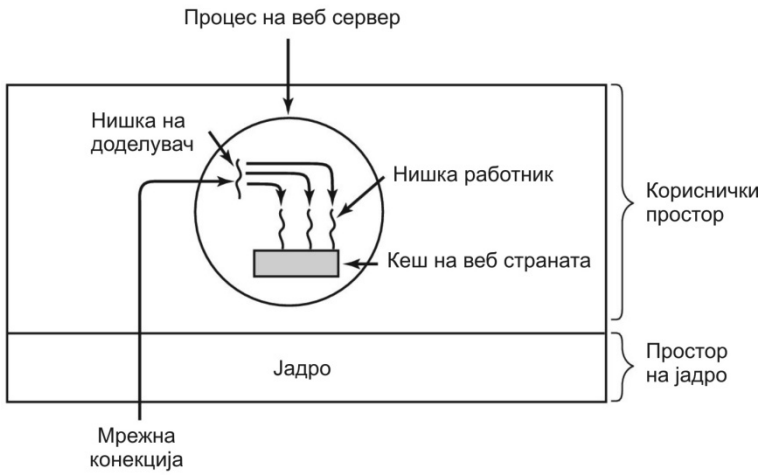
Слика 2.12: Пример со Word процесор

Ако апликацијата е напишана така што содржи две нишки, едната извршува интеракција со корисникот а другата се грижи за реформатирањето. При измената на првата страница, едната нишка започнува со процесот на реформатирање, додека другата продолжува да чека на командите на корисникот. Така процесот на реформатирање може да заврши и пред корисникот да внесе команда. Примерот може да се прошири произволно и со

трета нишка која повремено би го снимала документот на дискот, примерот е даден на Слика 2.12.

Користењето на повеќе процеси наместо нишки во овој пример нема да функционира затоа што сите три активности се изведуваат на еден ист документ. Користењето на нишките овозможува делење на заедничка меморија и сите имаат пристап кон истиот документ.

Друг пример за користење на нишки би бил сервер на WEB страници (Слика 2.13). Клиентите упатуваат барања кон серверот за испраќање на страницата. Кај повеќето Web сајтови на некои страници почесто им се пристапува отколку кон другите. Серверите го користат тоа за подобрување на перформансите со тоа што често посетуваните страници ги чуваат во главната меморија за да се избегне нивното барање и преземање од дискот (множеството на страниците се на нарекува кеш - *cache*). Серверот може да се организира на следниов начин, постои нишка наречена диспечер чија задача е очитување на барањата кои пристигнуваат од мрежата. По прегледувањето на барањето се избира неактивната (блокирана) работна нишка и се предава барањето. Работната нишка по будењето проверува дали барањето може да се задоволи со користењето на кешот до кој имаат пристап сите нишки. Ако не, нишката започнува со операција на читање од дискот и се блокира чекајќи на резултатот. За да се изврши поголем обем на работа, додека работната нишка е блокирана се избира диспечерот за очита ново барање или пак друга работна нишка.



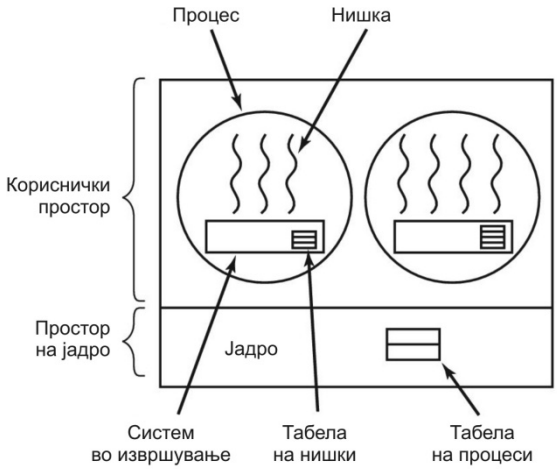
Слика 2.13: Пример со WEB сервер

Овој модел овозможува серверската апликација да биде опишана со секвенционални нишки, каде нишката диспечер работи во бескраен циклус очитувајќи ги барањата и давајќи ги на работната нишка. Секоја работна нишка пак се извршува во бескраен циклус проверувајќи дали барањето може да се најде во кешот, ако да тогаш се враќа страницата кон диспечерот и

нишката се блокира, ако не се стартува читањето од дискот. Ако серверот не користи нишки, едната варијанта е да има еден процес кој се извршува на тој начин што се додека не се опслужи барањето не може да се преземе следното, додека се чека на читањето на дискот процесот „врти во празно“ т.е. троши процесорско време. Ова доведува до помал број на обработени барања и неефикасно искористување на процесорот. Овие примери покажуваат како одредени активности може да бидат изведени со помош на нишки.

2.7.2. Имплементација на ниво на корисници

Постојат два начини за имплементација на нишките: во корисничкиот простор и во јадрото, а постои и хибридна имплементација. Првиот начин е сите нишки целосно да се постават во корисничкиот простор и јадрото нема никакво познавање на постоење на нишки во состав на одреден процес. Предноста се состои во тоа што нишките може да се имплементираат и кај оперативни системи кои не ги поддржуваат.



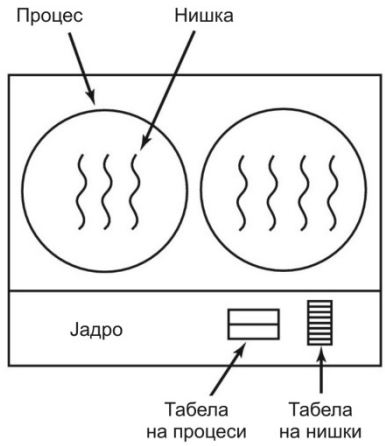
Слика 2.14: Имплементација на корисничко ниво

Кога нишките се управувани во корисничкиот простор, секој процес треба да има сопствена табела на нишки за да се следат во процесот (Слика 2.14). Оваа табела е аналогна на процесната табела во јадрото, со таа разлика што во табелата се чуваат карактеристиките на нишките како што се програмскиот бројач, покажувачот на стекот, регистрите, состојбата и т.н. Овие податоци се користат кога нишките поминуваат од состојбата на блокирана или подготвена во извршување при што од табелата се вчитуваат содржините на бројачот, регистрите и се воспоставува околина за нејзиното извршување. И обратно, кога една нишка преминува од состојбата на извршување во блокирана, сите претходно наведени податоци се снимаат во табелата на нишките во процесот.

Предностите на користењето на нишките во корисничкиот простор се состои од мали побарувања при измена на контекстот на нишките и чување на табелата, користење на сопствен алгоритам за распоредување на нишките. Од друга страна постојат и поголеми проблеми како што се имплементација на системските повици за блокирање за да се избегне влијанието на блокираната нишка врз другите. Ако се испрати повик за блокирање кон една нишка, бидејќи јадрото не знае за нивното постоење, ќе се блокира целиот процес и сите нишки кои влегуваат во неговиот состав. Друг проблем се состои во тоа што кога една нишка се извршува, не постои механизам за нејзино прекинување освен ако таа самата доброволно не ја предаде контролата на другите нишки. Постојат и други проблеми кои се во врска со страничењето на меморијата и имплементација на нишките кај активности кои често се блокираат. Постојат решенија за наведените проблеми, но обично се бара интервенција во кодот на јадрото за модификација на системски повици, но со тоа се губи предноста за имплементација и кај оперативни системи кои не подржуваат концепт на нишки.

2.7.3. Имплементација на ниво на јадрото

Кај овој пристап, јадрото управува со нишките и постои една табела на нишки која ги следи сите нишки од сите процеси во системот. Создавањето и уништувањето на нишките се врши со системски повици кон јадрото, при што се обновува и ажурира табелата на нишките.



Слика 2.15: Имплементација на корисничко ниво

Табелата во јадрото ги содржи сите податоци кои ги содржи и табелата на нишки во еден процес имплементирана на корисничко ниво (Слика 2.15). Покрај табелата на нишки, јадрото одржува и традиционална табела за процесите во системот.

Сите повици кои може да блокираат нишка се имплементирани како системски повици, а не како системска процедура во процесот. При блокирањето на нишката, јадрото бира или друга нишка од истиот процес или нишка од друг процес, за разлика од нишките од корисничкиот простор, каде што нишките се извршуваат во состав на еден процес сè додека не му се одземе процесорот. На тој начин подобро се искористува процесорот, држејќи го зафатен поголемиот дел од времето.

Нишките од јадрото не бараат нови не-блокирачки системски повици, ако една нишка во процесот предизвика „*page fault*“ настан, јадрото може многу лесно да провери дали процесот има други нишки кои може да се извршуваат и ги активира се додека бараната мемориска страница не биде донесена од дискот. Главниот недостаток на овој пристап е големата цена на системските повици за операции на нишките како што се создавањето, терминацијата и т.н. и создавањето на поголем „*overhead*“ заради поголемиот број на измени на контексти при поголем број на нишки.

2.7.4. Нишки наспроти процеси

На крајот предностите и недостатоците на користењето на нишките во однос на процесите може да се резимира преку следнава табела.

Табела 2.1. Споредба на нишки и процеси

Нишки	Процеси
• Нишките немаат податочен сегмент или хип • Нишката не може да живее самостојно, туку постои во рамките на еден процес • Повеќе нишки можат да бидат во ист процес, а првата ја повикува *main* функцијата • Евтина измена на контекст • Кога конецот умира се ослободува стекот	• Процесот има код/податоци, хип и други сегменти • Мора да има барем една нишка по процес • Нишките во процесот делат код/податоци/хип, В/И но секој има свој стек и регистар. • Скапо креирање • Кога процесот умира, ресурсите се ослободуваат и сите нишки умираат

3. Управување со процеси

Во систем кој користи мултипрограмирање, постојат повеќе процеси кои се натпреваруваат за процесорот во исто време. Овој случај се појавува кога имаме два или повеќе процеси кои се наоѓаат во состојба на подготвеност (READY). Ако системот располага со само еден процесор тогаш треба да се донесе одлука кој од процесите да се избере за доделување на процесорот. Во состав на оперативен систем се наоѓа дел кој се нарекува распоредувач и користи одреден алгоритам за распоредување на процесите. Во продолжението ќе бидат претставени алгоритамите за распоредување на процеси кои во голема мерка може да се применат и за распоредување на нишките.

Порано во системите кои користеле групно процесирање, алгоритамот за распоредување бил доста едноставен, да се извршува следната задача која се наоѓа на магнетната лента. Кај системите со временска поделба распоредувањето станало посложено, затоа што постојат повеќе корисници кои чекаат на процесирање односно услуга од централниот - „*mainframe*“ компјутер. Затоа што процесорското време е многу значаен и ресурс, делот за распоредување на процесите и алгоритамот кој се користи може да има големо влијание врз перципираната перформанса и задоволство на корисникот. Со појавата на персоналните компјутери настанала нова ситуација во поглед на користењето каде што обично во еден момент се користи една апликација или неколку апликации (наспроти повеќе корисници во систем со временска поделба) и потребните ресурси ги има во доволен број и никогаш скоро и да нема недостиг од нив. Од друга страна кај работните станици и серверите во мрежна околина постојат повеќе процеси кои се натпреваруваат за процесорот. Тука изборот на алгоритамот за распоредување може да има големо влијание врз прикажаните перформанси.

Дополнително, при изборот на вистинскиот процес за извршување, распоредувачот треба да го земе и в предвид и чинењето изразено во процесорско време на операцијата на измената на контекстот. Изведување на поголем број на измени на контексти по единица време може да доведе до неповолен однос на корисно потрошено процесорско време врз време за изведување на измената на контекстот.

3.1. Однесување на процесите

Сите процеси при извршувањето наизменично побаруваат користење на процесорот и влезно – излезни уреди. Обично процесорот се извршува без прекин извесно време, додека не се појави системски повик за читање или запишување кон некоја датотека. По завршувањето на системскиот повик, процесорот пак продолжува да се извршува сè додека пак не се појави потреба за пристап кон некој В/И уред. Изведувањето на В/И операција го поставува

тековниот процес во состојба на блокирање до завршувањето на системскиот повик, при што процесорот останува неискористен доколку не се додели на друг процес.

Треба да се забележи дека некои процеси поминуваат повеќе време во користењето на процесорот за пресметки додека другите пак поминуваат повеќе време во чекање на В/И операции. Извршувањето на еден процес се состои од еден или повеќе циклуси на извршување на процесорот (*CPU burst*) и еден или повеќе циклуси на чекање на влезно-излезни операции (*I/O burst*). Процесот започнува со циклусот на извршување на процесорот и потоа може повеќе пати да се менуваат овие циклуси или состојби. Врз основа на видот на ресурсите кои доминантно се користат т.е. бројот и траењето на циклусите, процесите се делат на:

- Процесот кој доминантно го користи процесорот (*англ. CPU bound*)
- Процесот кој доминантно користи влезно-излезни операции (*англ. I/O bound*)

Најголем дел од времето на извршување В/И интензивните процеси отпаѓа на влезно-излезни циклуси, додека на употребата на процесорот и меморијата се троши релативно малку време. Сосема е природно овие процеси да можат релативно набргу да се блокираат и да чекаат на завршување на В/И операција (влез во состојба WAIT). ЦПЕ интензивниот процес генерира многу малку влезно-излезни барања, а најголем дел од своето време троши на користењето на процесорот.

Со динамичниот развиток на процесорите и зголемувањето на процесирачката моќ, кај процесите станува доминантно користењето на В/И операциите, кои стануваат значаен фактор при процесот на распоредување. Основната идеа е дека кога В/И интензивен процес бара доделување на процесорот, тоа треба да се изврши што побргу за да се издаде повик за пристап кон дискот и да го држи зафатен поголем дел од времето, заради поефикасен трансфер на податоците.

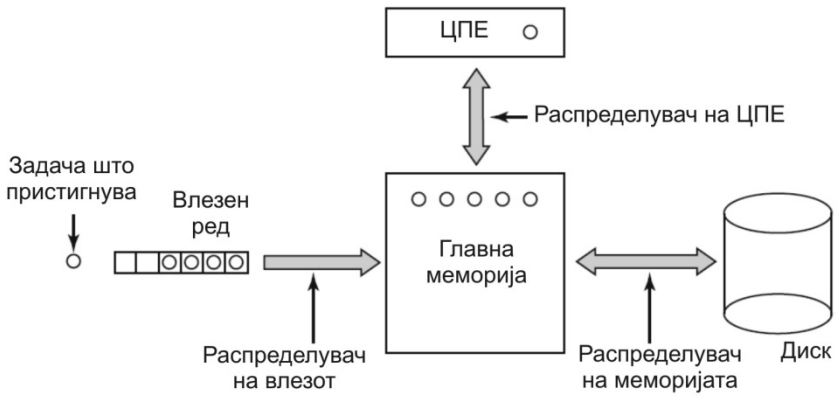
Потребата за распоредување на процеси се појавува во неколку случаи. При креирање на процес, треба да се изврши распоредување на процесот родител и процесот дете каде и двата се наоѓаат во состојбата на подготвени. Потоа, избор на процес треба да се изврши и кога еден процес се терминира, тој процес не може веќе да се извршува па е потребно да се избере друг од редицата на чекање на процесорот. Друг случај е кога процесот се блокира на В/И повик, на семафор или од некоја друга причина друг процес се избира за извршување. И на крајот може да се појави случај кога ќе се иницира хардверски прекин и треба да се донесе одлука за распоредување на процесите. Распоредувачот треба да одлучи дали по сервисирањето на прекинот ќе се извршува нов процес од редицата на чекање, процесот што беше прекинат или пак некој трет процес.

3.2. Распоредувачи

Како што веќе е опишано, процесот во своето траење поминува низ разни состојби и редови на чекање. Програмите за распоредување (*англ. schedulers*) одлучуваат за тоа кога процесот ќе влезе во некој ред на чекање или ќе го напушти тој ред. Распоредувањето може да се изведува на три нивоа (Слика 3.1). Кога се поднесува задача (*англ. job*) во системот, иницијално се сместува на дискот (кај „*batch*“ системи). Пристапниот распоредувач (*англ. admission scheduler*) одлучува кои од работите да ги внесе во системот, а другите се чуваат во влезната редица на чекање на дискот сè додека не бидат изберени. Типичен алгоритам за ова ниво на распоредување би бил да се избираат комбинација од процесор интензивни и В/И интензивни процеси. Или друг начин би бил, да се изберат првин пократките процеси, а подолгите да чекаат. Пристапниот распоредувач има слобода да избира кои процеси ќе ги внесе во системот а кои да ги задржува на дискот. Пристапниот распоредувач во хиерархискиот систем се наоѓа над јадрото, ги објавува наведените функции:

- ги дели работите на процеси;
- врз основа на одредени алгоритами им доделува приоритети на процесите.
- ги доделува процесите во ред на чекање кај процесорот.

Пристапниот распоредувач (долгорочен распоредувач, планер на работи) практично се повикува само кога ќе се појават нови процеси или кога еден или повеќе процеси ќе ја извршуваат активноста. Тој го регулира степенот на мултипрограмирање преку бројот на процесите кои се пренесени од дискот во меморијата. Од него се очекува да прави добра селекција на процесите кои ќе добијат пристап во меморијата за да системот што поефикасно функционира. Ова ниво на распоредување потекнува од големите сметачки системи (mainframe) кои ги карактеризира групна обработка на програмите.



Слика 3.1: Распоредувачи на процеси

Еднаш кога задачата е внесена во системот, за неа се креира процес и започнува да се натпреварува за доделување на процесорот. Но, можно е да се случи, бројот на процесите сместени во главната меморија толку да нарасне што нема повеќе место за сите нив. Во тој случај некои од тие процеси ќе бидат суспендирани на дискот (*англ. swapped out*). Второто ниво на распоредување донесува одлука кои процеси да се задржат во меморијата а кои на дискот. Овој распоредувач се нарекува мемориски распоредувач или среднорочен распоредувач.

Овој распоредувач треба почесто да врши проверка за да им овозможи на процесите на дискот да ја добијат услугата од процесорот. Меѓутоа, пренесувањето на процеси од и кон дискот е скапа операција и ако содржината на меморијата често се пренесува се троши голем пропусен опсег на дискот со што се забавуваат В/И операциите со датотеките. Исто така, среднорочниот распоредувач има улога во определување на степенот на мултипрограмирање преку регулирањето на бројот на процесите кои се наоѓаат во меморијата. Тој има увид во природата на процесите дали се процесорски или В/И интензивни и треба да се обидува да одржува смеса на овие процеси во меморијата.

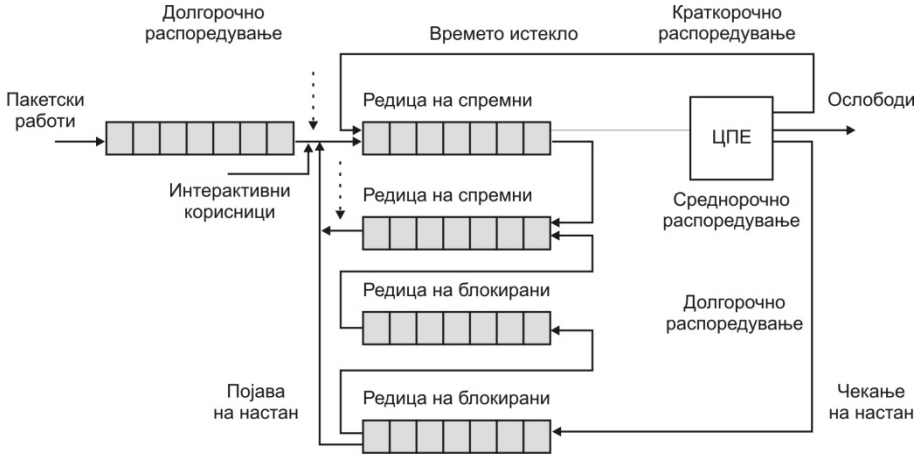
За да донесува одлуки, меморискиот распоредувач периодично го прегледува секој процес на дискот за да го избере или не. Критериумите според кои се води среднорочниот распоредувачот се времето поминато од префрлањето на процесот, колку процесорско време потрошил последен пат, колку е голем процесот и колку е значаен.

Третото ниво на распоредување всушност ги избира процесите присутни во меморијата за да им го додели процесорот, се нарекува краткорочен или процесорски распоредувач. Тој може да користи било кој од алгоритамите за распоредување кои ќе бидат претставени во продолжението на текстот.

3.2.1. Редици на чекање

Процесот својот живот минува низ неколку редови на чекање (*англ. scheduling queues*). По креирањето процесот се вметнува во влезена редица на чекање за (*англ. job queue*), кој ги опфаќа сите постоечки процеси на системот. Процесите можат да се најдат во различни состојби и на различни локации (во меморија, на дискот, делумно во меморија, делумно на дискот). Сите процеси кои се подготвени за работа и се наоѓаат во работната меморија се чуваат во редицата на чекање на процесорот, т.е. во редот на чекање на подготвени процеси (*англ. ready queue*). Редиците на чекање на процесорот по правило се реализираат како поврзани листи, формирани од контролните блокови на процесите со дефиниран редослед за извршување на процесите. Редоследот се задава преку листата на заглавја (*англ. queue header*) која ги содржи информациите за почетниот и завршниот контролен блок во листата и на покажувачот на следниот контролен блок.

Оперативниот систем воведува и посебен ред на чекање за секој влезно-излезен уред (*англ. I/O queue*). Секоја редица на чекање на уредот (системот) содржи поврзана листа на контролни блокови на процесите кои го побаруваат уредот. На Слика 3.2. е прикажана техниката за распоредување на процеси (*англ. scheduling*).



Слика 3.2: Редици на чекање и распоредување на процесите

Новиот процес иницијално се поставува во редицата на чекање подготвени процеси во кој се чека доделување на процесорот, по што го напушта редот и започнува да се извршува. Процесот кој се наоѓа во состојба на извршување може да:

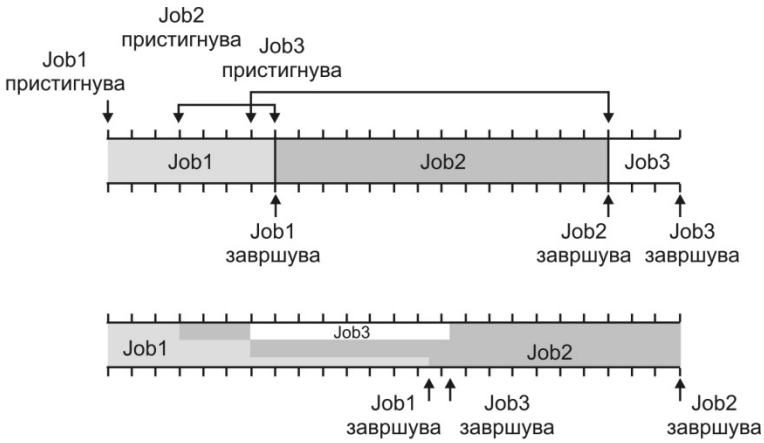
- остане без процесорот кога ќе му истече временскиот квантум
- да се создаде нов процес и да чека во блокирана состојба, додека новиот процес да се изврши;
- да се остане без процесор кога ќе се случи прекин ;
- да се постави В/И барање, по што се префрлува во редот на чекање на влезно-излезен уред т.е. станува блокиран.

Процесот се враќа во редот на чекање на процесор се додека не се изврши после што ги ослободува сите зафатени ресурси.

3.3. Алгоритами за распоредување на процесите

За да се дизајнира алгоритам за распоредување, треба да се постават цели кои треба да ги исполни. Целите зависат од околина во која се извршуваат процесите („*batch*“, интерактивни или системи кои работат во реално време). Една цел е сите процеси кои имаат споредливи карактеристики и барања да добијат и соодветна услуга, спротивно на тоа одредени процеси од критично

значење треба да имаат поголем приоритет при избор во процесот на распоредување. Друга општа цел е да се држат ресурсите во состав на системот цело време зафатени, ако и процесорот и влезно-излезните уреди цело време работат ќе бидат обработени повеќе задачи во единица време отколку некои компоненти да бидат неактивни. Одлуката за избор на процес на кој ќе му се додели процесорот треба да ја подобри искористеноста на процесорот (користење на ресурси), но и да го подобри времето на одзив (задоволување на корисникот), при што треба да се запази и поедначков третман на процесите. Одлуката се базира неколку критериуми од кои најзначајни се два параметри: користењето на процесорот кое се опишува како односот на времето кое процесорот работел на процесот и вкупното време на работа и времето на одзив кое се определува како просечна вредност на разликата од времето на завршување и времето на започнување на процесот (поставување во редот на чекање). Двата параметри се спротивставени еден на друг и не може едниот да се подобри без штета на другиот. На Сликите 3.3. и 3.4 се дадени примери за како да се подобрат и двата параметри.

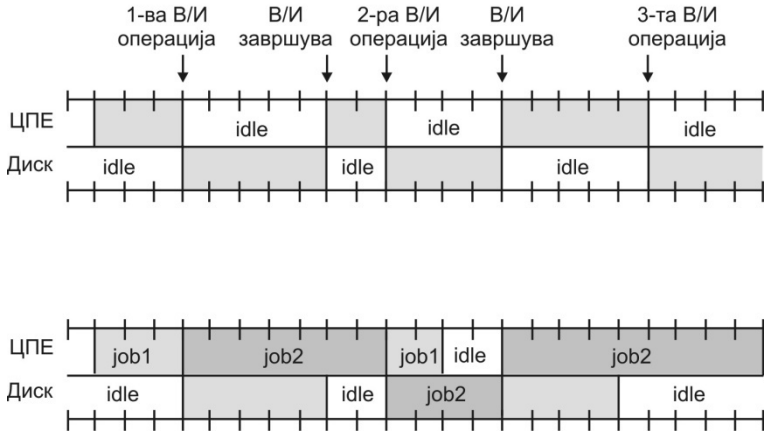


Слика 3.3: Одзив (*англ. responsiveness*)

На Слика 3.3 е прикажан преминот од секвенционално извршување на процеси каде што 3 задачи (програми) пристигнуваат во системот во различни моменти еден по друг. Кога системот би бил со монопрограмирање, сите процеси би се извршувале еден по еден што би довело до зголемено просечно време на одзив и нарушена интерактивност. Ако се користи систем кој работи квазипаралелно како што пристигнуваат задачите односно процесите така се сместуваат во редица на чекање и се избираат за извршување. Во прикажаниот случај сите се третираат поедначкво и очигледно е дека интерактивноста значително се подобрила, иако на штета на времето на извршување на поедини процеси.

Другиот пример претставува процес кој повремено се блокира за извршување на В/И операција, во систем со монопрограмирање очигледно е неефикасното искористување на процесорот, каде скоро половина од времето на

процесирање процесорот е неактивен додека процесот е блокиран. Со воведување на мулти-програмирањето и доделување на процесорот на процесите кои чекаат во ред на чекање секогаш кога тековниот се блокира се зголемува искористувањето на процесорското време. Секако на штета на времето на извршување на поединечни процеси.



Слика 3.4: Искористувањето на процесорот

Конечно може да се набројат критериумите за распоредување на процесите:

- искористеноста на процесорот (*англ. CPU utilization*)
- пропусната моќ на системот (*англ. throughput*)
- време потребно за завршување на процесот (*англ. turnaround time*)
- време на чекање (*англ. waiting time*)
- време на одзив (*англ. response time*)

Искористеноста на процесот се претставува како однос на корисното време врз вкупно потрошеното време на работата на процесорот, каде корисното време е времето во кое процесорот бил доделен на некој процес.

Пропусната моќ се изразува во бројот на задачите (процесите) по единица време кои системот ги извршува. Општо гледано, подобро е да се извршат повеќе програми за што пократко време.

Времето за извршување на процесот е статистички просечен временски период од моментот кога задачата е поставена во системот до моментот на нејзиното завршување. Добро време на извршувањето не мора да значи и голема пропусна моќ и обратно, на пример ако има множество на кратки и долги процеси и распоредувачот секојпат ги избира пократките отколку подолгите, може да се оствари одлична пропусна моќ но по цена на многу лошо време на извршување на подолгите процеси.

Времето на чекање се изразува како временскиот период од моментот кога процесот влегол во редицата на чекање се до моментот кога бил распореден за извршување.

Времето на одѕив претставува времето од издавањето на командата до добивање на некаков резултат.

Во зависност од намената на системот, понекогаш алгоритамите се оптимизирани за одредени услови, каде што понекогаш се форсираат минималните или максималните вредности, а понекогаш и средните вредности. На пример, кај интерактивните системи побитно е да се минимизира времето на одѕив отколку било кој друг критериум.

Битен фактор во систем со мултипрограмирање е дали е дозволено прекинување на процес во извршување и негова измена со друг процес кој чека во редицата на подготвени за извршување. Алгоритам за распоредување без испразнување (*англ. non-preemptive*) избира процес за извршување каде тој го зафаќа процесорот сè додека доброволно не го препушти на друг или не се блокира на В/И операција дури и да се извршува со часови.

Спротивно на ова алгоритам за распоредување со предиспразнување (*англ. preemptive*) избира процес и му дозволува да се извршува само одредено времетраење, ако сè уште се извршува по истекот на временскиот квантум распоредувачот го суспендира и се избира друг процес под услов да има таков. За овозможување на распоредувач со алгоритам со предиспразнување, системот треба да има имплементирано часовник на реално време и механизам на прекини, односно прекини генерирани од часовникот. Ако не постои часовник, единствена опција е користење на алгоритами без испразнување.

Покрај ова, алгоритамите за распоредување се делат и според видот на системот на кој се имплементираат. Кај „*batch*“ системите не постојат корисници кои се поврзуваат на терминали и очекуваат брз одѕив. Следствено и алгоритамите без испразнување и со испразнување со долги периоди за извршување на процесите се прифатливи. Во околина со интерактивни корисници, пред-испразнување е од суштинско значење за оневозможување на монополизација на процесорот од страна на еден корисник.

Во системите со критични процеси кои треба да запазат одредени временски ограничувања (*англ. real-time*), понекогаш не се потребни распоредувачи со пред-испразнување, затоа што процесите релативно кратко за извршуваат и бргу се блокираат. За разлика од интерактивните системи овие се користат за специјални намени, отколку за општа намена каде што се извршуваат произволни апликации кои што може и меѓусебно да си наштетат.

Прво ќе бидат преставени алгоритамите за распоредување во системи со групна обработка:

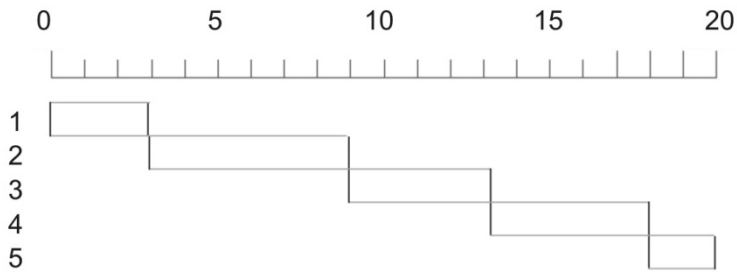
- Прв пристигнат, прв услужен (*First Come, First Served - FCFS*)
- Најкратката задача прва (*Shortest Job First - SJF*)
- Задача со најкратко преостанато време (*Shortest Time Remaining Next - STRN*)

А потоа и алгоритамите кои се користат во интерактивни системи и системи со мултипрограмирање:

- Кружно доделување (*Round-Robin - RR*)
- Распоредување со приоритети (*Priority Scheduling - PS*)
- Најкраткиот процес следен (*Shortest Process Next - SPN*)
- Распоредување со извлекување (*Lottery scheduling - LS*)
- Подеднакво распоредување (*Fair-Share Scheduling - FSS*)
- Распоредување во *Real Time* системи

3.3.1. Прв пристигнал – прв услужен (FCFS)

Наједноставен алгоритам за распоредување без испразнување е „прв пристигнал – прв услужен“ (*англ. first-come first-served*). Со овој алгоритам на процесите им се доделува процесорот по редоследот како што го побаруваат. Постои една редица на чекање на подготвени процеси и како што првата задача влегува во системот преку долгорочниот распоредувач, веднаш започнува да се извршува сè додека не заврши, како што наидуваат други процеси се вметнуваат на крајот на редицата на чекање. Ако процесот кој се извршува се блокира, следниот од редицата на чекање се распоредува, а кога блокираниот процес излегува од состојбата на блокирани се сместува на крајот од редицата на подготвени како ново создаден процес.



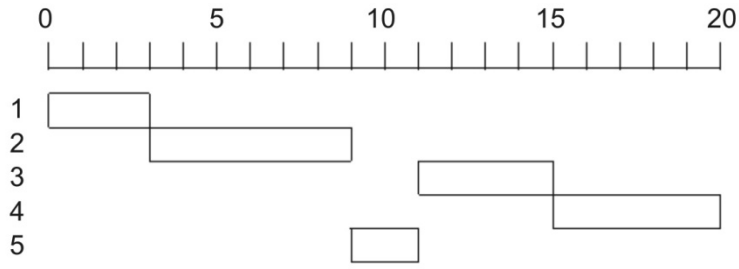
Слика 3.5: Гантограм за FCFS алгоритам за распоредување

Најголемата предност на овој алгоритам е дека е едноставен за разбирање и за имплементација. Но постои и голем недостаток, на пример, нека се

претпостави дека имаме процесорски интензивен процес кој се извршува во времетраење од 1 секунда и многу В/И интензивни процеси кои малку го користат процесорот, но на секој му треба 1000 циклуси на читање од дискот за да се заврши. Значи, првиот се извршува за 1 секунда по што врши читање на еден блок од дискот. По него се извршуваат сите други В/И интензивни процеси последователно. Се дава предност на процесорски интензивни процеси, додека В/И процесите мора да чекаат додека ЦПЕ процесите не завршат, слично како ефект на конвој во редица за В/И. Резултат од ова е што процесот кој е процесорски интензивен ќе го зафати процесорот сè додека не се изврши (алгоритамот е без испразнување), додека во редицата се натрупуваат В/И процеси кои кратко би го држеле процесот и бргу би се блокирале на чекање на В/И операција. Алгоритамот FCFS општо се карактеризира со голема вредност на средното време на чекање и зависи од траењето на процесите како и од редоследот по кој доаѓаат во редицата на чекање. Затоа што алгоритамот е со не-испразнување не е погоден за користење во интерактивни системи.

3.3.2. Наредeн најкус процес (SJF)

Овој алгоритам користи политика на не-испразнување и однапред е потребно познавање на времетраењето на извршување на процесите. Од редицата на подготвени процеси се избира оној процес кој има најмало очекувано време на обработка при што е потребно да се изврши процена на времето.

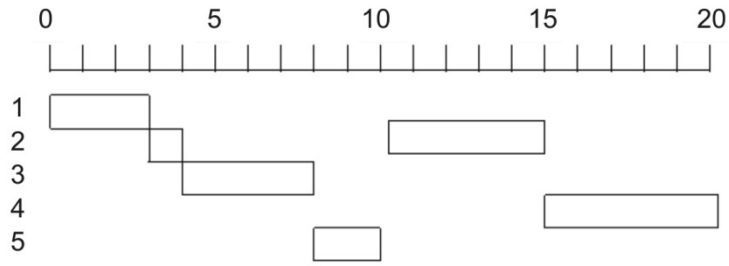


Слика 3.6: Гантограм за SJF алгоритам за распоредување

Овој алгоритам дава предност и ги фаворизира кусите процеси, а ако двата се со исто времетраење се користи FCFS алгоритамот за распоредување. Оптимален во случај кога сите процеси пристигнуваат во редицата на подготвени симултано во ист временски момент. Недостаток е што тешко може да се направи правилна проценка за траењето на процесот, обично се земаат вредности на периодот на извршувањето на процесот до блокирањето за В/И операција (*англ. CPU burst time*).

3.3.3. Најкусо преостанато време (STRN)

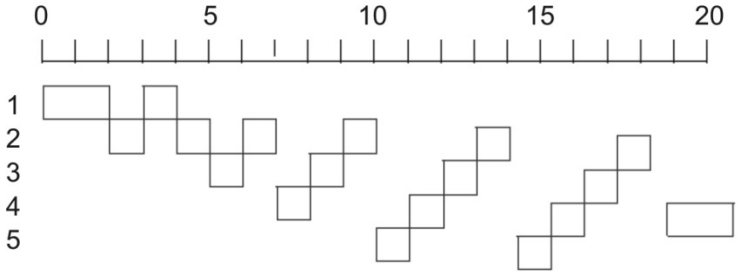
Верзија на SFJ со политика на испразнување е алгоритамот за избор на најкусото преостанато време за извршување. Распоредувачот секој пат го избира процесот чие преостанато време на извршувањето е најмало. Нов процес дојден во редицата на подготвени процеси може да има покусо траење од тој што се опслужува и процесорот се доделува на новиот процес, а подоцна се довршува претходниот. Ова овозможува новите процеси кои имаат кратко времетраење да добијат благовремена услуга со доделувањето на процесорот. И за овој важи како за претходниот алгоритам, дека е потребно да се направи проценка за времето на процесирање.



Слика 3.7: Гантограм за STRN алгоритам за распоредување

3.3.4. Кружно доделување (Round-Robin)

Еден од најстарите, наједноставните и најкористените алгоритами кој обезбедува подеднаков третман на процесите е кружно доделување или „round robin“. На секој процес му е доделен временски интервал кој е наречен квантум, во кој му е дозволено да се извршува.



Слика 3.8: Гантограм за RR алгоритам за распоредување

Ако процесот сè уште се извршува по истекувањето на квантумот, тој се испразнува од процесорот и истиот се доделува на друг процес. Ако при извршувањето во квантумот процесот се блокира или се изврши, префрлувањето на друг процес се врши и пред да заврши квантумот. Овој алгоритам е едноставен за имплементација, распоредувачот треба да има листа

на подготвени процеси од каде ги избира еден по еден, а како што истечува квантумот на процесот тој се става на крајот од листата.

Основен параметар на овој алгоритам е должината на временскиот квантум, префрлувањето од еден процес на друг бара извесно време за да се извршат сите попратни активности, снимањето и вчитувањето на регистрите и мемориската мапа и други. На пример, нека се земе дека измена на контекстот трае 1 милисекунда и квантумот е поставен на 4 милисекунди. Со овие параметри процесот 4 милисекунди од времето извршува корисна работа, а 1 милисекунда се т.н. административни активности, што е премногу. За да се подобри искористувањето на процесорот, квантумот се поставува на пример на 100 милисекунди и тогаш административното трошење (*англ. overhead*) на процесорското време изнесува само 1%. Меѓутоа, ако на пример на системот се врзуваат десет корисници и сите истовремено задаваат команда со што се генерираат процеси кои се поставуваат во листа на подготвени процеси, временскиот квантум од 100 милисекунди е определен за секој корисник, тогаш првиот процес веднаш се извршува, вториот по 100 милисекунди а десеттиот по цела секунда што не е задоволителен одзив за корисникот.

Друг фактор е ако квантумот е поставен на подолго време од времетраењето на користењето на процесот (*англ. CPU burst*), пред-испразнувањето нема да се случува често, туку процесите ќе изведуваат блокирање пред истекот на квантумот што предизвикува измена на контекстот. Со елиминирање на пред-испразнувањето се подобруваат перформансите затоа што измената на процесите се случува само тогаш кога е потребно, кога процесот ќе се блокира и не може да продолжи.

Поставувањето на временскиот квантум на краток период предизвикува поголем број на измени на процеси и ја намалува ефикасноста на процесирањето, од друга страна поставувањето на квантумот на подолг период пак предизвикува слаб одзив и интерактивност за кратки процеси. Се карактеризира со долго средно време на чекање.

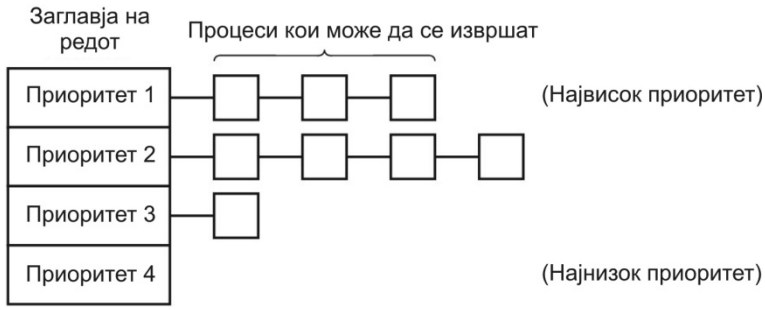
3.3.5. Распоредување со приоритети (Priority Scheduling)

Кај претходниот алгоритам (*Round Robin*) е направена претпоставка дека сите процеси се подеднакво значајни. Ако се земаат в предвид и одредени надворешни фактори за карактерот и значајноста на процесите, треба да се користи алгоритам кој под одредени услови ќе фаворизира одредени процеси. Таков алгоритам се нарекува распоредување по приоритети, каде што на секој процес му е доделен приоритет и сите процеси со највисок приоритет и е дозволено да се извршуваат.

За да се спречи процесите со висок приоритет бесконечно да се извршуваат, распоредувачот го намалува приоритетот со секој прекин на часовникот (*англ. clock interrupt*). Ако приоритетот падне под нивото на следен по приоритет

процес, се врши измена на контекстот. Од друга страна на секој процес може да се додели максимално време на извршување, по чиј истек се дава можност за извршување на друг процес со најголем приоритет.

Приоритетите можат да бидат доделувани статистички и динамички. Динамички се доделуваат кога треба да се постигнат одредени цели. На пример, некои процеси се В/И интензивни и поголемиот дел од времето поминуваат чекајќи на В/И операции. Секојпат кога ваков процес ќе го побара процесорот, веднаш треба да му се додели за да ја стартува В/И операцијата која ќе продолжи понатаму да се извршува во паралела на следниот распореден процес. Ако еден В/И интензивен процес чека долго време на процесорот, тоа значи дека чека во меморијата непотребно долго време. Едноставен алгоритам за опслужување ма овие процеси е да им се додели приоритет со вредност од *1/f* каде што *f* е делот од последниот квантум што го користел тој процес. Процес кој искористил 1 милисекунда од 50 милисекунди квантум ќе добие приоритет 50, додека процес кој работел 25 милисекунди ќе добие приоритет 2, а цел квантум приоритет 1.



Слика 3.9: Распоредување на процеси по приоритет

Обично процесите се групираат во класи на приоритети (Слика 3.9) каде што се користи распоредување со приоритети помеѓу класите, а во склоп на една класа се користи кружно доделување (*Round Robin*). Ако приоритетите повремено не се прилагодуваат, може да се случи класите со помал приоритет да „изгладнат“ чекајќи на доделувањето на процесорот.

3.3.6. Повеќекратни редици на чекање (Multiple Queues)

Поефикасно е да се додели поголем квантум наеднаш на процесите кои се процесорски интензивни наместо помал квантум почесто (се намалува измената на контекстот). Од друга страна, ако на сите процеси им се даде поголем квантум тоа ќе значи слабо време на одзив како што е веќе претходно објаснето. Едно решение е да се користат класи на приоритети каде што процесите од повисоките класи добиваат еден квантум, во втората класа два

квантуми, по третата 4 квантуми и така понатаму. Како што процесот ќе ги потроши доделените квантуми се преселува во подолната класа.

На пример, ако се разгледа процес кој треба да се извршува 100 квантуми, иницијално ќе му биде доделен еден квантум па потоа ќе се замени. Следниот пат ќе му се доделат 2 квантуми, па 4, 8, 16, 32 и 64 квантуми иако ќе искористи 37 од доделените 64. Во овој случај ќе бидат извршени само 7 замени на контекстот наместо 100 кога би се користело кружното доделување (*Round-Robin*). Како што процесот оди во подолните класи ќе се извршува сè поретко овозможувајќи им пристап на пократките интерактивни процеси. Идејата е да се оневозможи ситуацијата кога процес кој треба долго да се извршува но станува интерактивен подоцна да биде казнуван бесконечно.

3.3.7. Најкраткиот процес следен (Shortest Process Next)

Затоа што алгоритамот за избор на најкраткиот процес секој пат предизвикува најмало средно време на чекање за „*batch*“ системите, би било добро да се искористи и во интерактивните системи. Проблемот што беше споменат е како да се изврши проценка на времето на извршување од процесите во редицата на подготвени и да се избере најкраткиот. Следното време на опслужување со процесорот на еден процес може да се предвиди според времето на опслужување на процесот во минатото.

Нека T_n е должина на n-то опслужување на тој процес (најсвежа информација, скора историја - *recent history*), F_{n+1} е нашата предвидена вредност за следната должина на процесот (F_n е историја од минатото – *past history*).

Се дефинира рекурентна врска:

$$F_{n+1}= \alpha \cdot T_n + (1- \alpha) \cdot F_n,$$

Каде што α го контролира влијанието на скорешната и мината историја. Проценка на времетраење на следно извршување на процес: Нека има две претходни вредности за времето на извршување на еден процес, T_0 и T_1.

$$F_1=T_0,\ F_2= \alpha \cdot T_1 + (1- \alpha) \cdot F_1,$$

$$F_K = \alpha \cdot T_{K-1} + (1- \alpha) \cdot F_{K-1},\ K=3,\ 4,\ \ldots$$

за вредност на $\alpha= 1/2$ скората и историјата од минатото имаат еднаква тежина.

$$T_0,\ T_0 /2+T_1/2,\ T_0 /4+T_1/4+ T_2/2,\ T_0 /8+ T_1/8 + T_2 /4+ T_3/2,\ldots$$

Со изминувањето на неколку извршувања вредноста на T_0 паднала на 1/8 и предизвикува минатата историја сè помалку влијае врз иднината. Техниката за проценка на следната вредност во серијата со земање на средната вредност од тековната измерена и претходните процени се нарекува „стареење“ (*англ. aging*) и се применува во повеќе ситуации каде што треба да се направи предикција врз основа на претходните вредности. SJF е единствено оптимален (според времето на чекање) кога сите процеси доаѓаат во ист момент.

3.3.8. Распоредување со извлекување (Lottery Scheduling)

Основната идеа која стои зад овој алгоритам е да се издаваат „лото“ тикети за разни ресурси на системот како што е на пример, процесорско време. Секој пат кога треба да се направи распоредување, случајно се избира еден тикет и процесот кој го поседува тикетот го добива ресурсот на користење. Кога се применува на распоредување на процесорот, системот може да извлекува тикети 50 пати во секунда така што победникот во извлекувањето добива по 20 милисекунди од времето. Позначајните процеси може да добијат и повеќе тикети за да се зголемат шансите за нивно распоредување. Во овој случај е правилото за доделување на приоритетот е: процес кој поседува одреден број на тикети од вкупниот број добива исто толкав обем од ресурсот на користење. Ако од 100 постоечки тикети еден процес поседува 20 ќе добие 20% процесорско време.

Распоредувањето со извлекување има неколку интересни карактеристики: процесот при секое распоредување има шанса да биде извлечен пропорционално на бројот на тикетите кои ги држи, т.е. овој алгоритам има голем одзив. Процесите кои меѓусебно соработуваат може да си ги разменуваат тикетите, на пример кога клиентски процес испраќа барање кон серверот, се блокира и при тоа ги дава сите свои тикети на серверот со што му ја зголемува шансата за распоредување. Серверот по извршувањето и задоволувањето на барањето ги враќа тикетите на клиентот.

Алгоритамот за распоредување може да се користи за решавање на проблеми кои тешко се реализираат преку други методи. Еден пример е сервер на видео, каде што неколку процеси го даваат потокот на видео сигналот кон нивните клиенти но со различен број на рамки по секунда. Ако се претпостави дека процесите бараат 10, 20 и 25 рамки по секунда и со доделување на соодветен број на тикети 10, 20, 25 автоматски се дели процесорското време со бараниот сооднос 10:20:25.

3.3.9. Подеднакво распоредување (Fair-Share Scheduling)

Досега беше претпоставено дека секој процес се распоредува самостојно, без водење сметка на кој корисник му припаѓа. Како последица на тоа, ако еден

корисник стартува 9 процеси, а друг корисник само 1 и се користи кружно доделување, првиот корисник ќе добие 90% од времето а вториот само 10%.

За да се избегне ситуацијата, некои системи земаат в предвид чиј е процесот пред да се изврши распоредувањето. Секој корисник му се доделува дел од процесорот и процесите се избираат според дадениот начин. Така ако на еден корисник му се доделува 50% од процесорското време тој ќе го добие без разлика на бројот на процесите кои ги поседува. На пример, на два корисници им се доделува по 50% од процесорското време. Корисникот 1 ги има овие процеси, A, B, C и D, а корисникот 2 само еден процес E. Ако се користи распоредување со кружно доделување врз наметнатите ограничувања на основа на корисникот, секвенцата би изгледала вака. Со користење на RR алгоритамот и 50% процесорско време за секого:

A E B E C E D E A E B E C E D E

Ако на корисникот 1 има доделено 2-пати поголемо процесорско време тогаш секвенцата на извршување на процесите е:

A B E C D E A B E C D E

Постојат и други варијанти кои може да се искористат во зависност од односот доделен на корисниците.

3.3.10. Распоредување во Real Time системи

Времето во системите со реално време игра клучна улога. Обично еден или повеќе надворешни физички уреди поврзани со компјутерот кои создаваат влез на кој компјутерот мора соодветно да реагира во одреден временски рок. Еден пример би бил компакт диск уред кој ги презема битовите како што се читаат од дискот и треба да се конвертираат во музика во многу краток временски интервал. Ако пресметките од било која причина се оддолжат или забават, продуцираниот звук ќе биде деградиран. Други такви системи на се на пример, за следење на пациенти во болница, авто-пилот во авион или контрола на роботи во фабрика. Во сите овие случаи немањето на благовремен одговор е полошо од воопшто и да не се добие одговор на барањето од уредот кој работи во реално време.

Вакви системи се категоризираат како тврди (*англ. hard real time*) каде што дадените временски рокови треба да се почитуваат, или меки (*англ. soft real time*) каде што пречекорувањето на ограничувањето не е посакувано, но може да се толерира. И во двата случаи, однесувањето во реално време се постигнува преку делење на програмата на процеси каде што за секој постои предвидливо однесување кое е однапред познато. Овие процеси се

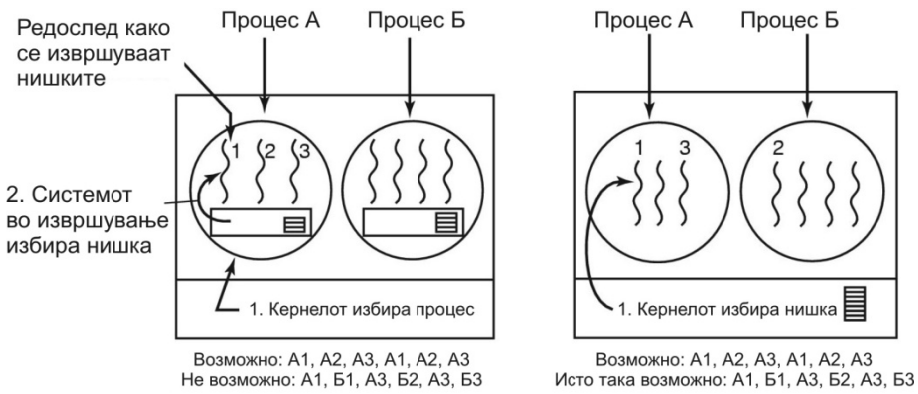
краткотрајни и бргу се извршуваат. Кога ќе се детектира надворешен настан, распоредувачот има за задача да ги распореди процесите на таков начин да бидат запазени временските рокови. Настаните може да се поделат и на периодични, кои се појавуваат во одредени интервали и на апериодични, чие појавување не може да се предвиди. За еден систем кој работи во реално време се вели дека може да врши распоредување на процесите ако е задоволен критериум, каде што постојат m периодични настани и настанот i се појавува со периода P_i и бара C_i секунди процесорско време за опслужување на секој процес:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Распоредувањето во реално време може да биде статичко или динамичко. Кај статичко одлуките за распоредувањето се донесуваат уште пред да се стартува системот, а кај динамичкото се донесуваат при самото извршување.

3.4. Распоредување на нишки

Кога повеќе процеси имаат повеќе нишки, постојат два нивоа на паралелизам: процеси и нишки. Распоредувањето на овие системи се разликува во зависност дали се поддржани нишки на корисничко ниво или на системско или и на двата нивоа. Во случајот на нишки имплементирани на корисничко ниво, јадрото не знае за нивно постоење и распоредувањето се врши врз основа на процеси, а во самиот процес распоредувачот на нишките одлучува која нишка да се извршува. Како што не постојат прекини системскиот часовник за мултипрограмирањето на нишките, нишката може да се извршува колку што сака, ако го искористи процесниот квантум, јадрото ќе го прекине процесот и ќе избере друг за извршување. Кога процесот пак ќе се распореди нишката ќе продолжи да се извршува се додека не заврши.



Слика 3.10: Можен начин за распоредување на нишки од корисничко ниво а) и од нивото на јадрото
б) 50-msec процесен квантум, нишката се извршува 5 msec/CPU burst

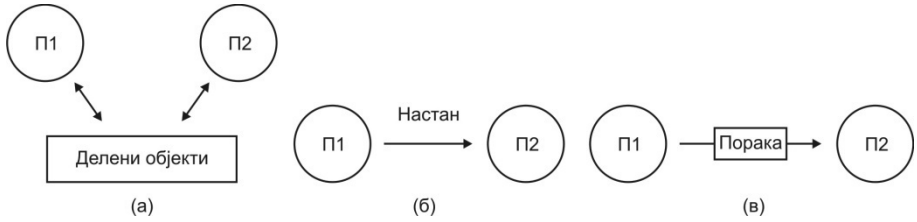
Алгоритамот за распоредување може да биде било кој од претходно наведените, обично се користат RR и распоредување по приоритети. Другиот случај е со нишки имплементирани во јадрото, тука јадрото избира одредена нишка за извршување, така можна е наизменична промена на нишките кој разни процеси како што е дадено на Слика 3.10 б), за разлика од претходниот случај каде што нишките се распоредуваат во состав на еден процес - Слика 3.10 а). Главната разлика помеѓу двата вида на имплементации на нишките се перформансите, изведувањето на измената на нишка на корисничко ниво бара помал број на инструкции, додека во јадрото бара целосна измена на контекстот на начин опишан во претходната глава. Од друга страна блокирање на нишка имплементирана во јадро нема да го блокира целиот процес.

4. Интерпроцесна комуникација и синхронизација

Процесите често комуницираат еден со друг, како што беше прикажан примерот со користење на цевковод (*англ. pipe*) во командниот интерпретер, каде што излезот од првиот процес се пренасочува како влез на вториот процес. Затоа е потребен добро структуриран начин за комуникација помеѓу процесите која се нарекува интерпроцесна комуникација (*Interprocess Communication – IPC*) и кој не би го користел механизмот на прекини.

Постојат три случаи за кои е потребна интерпроцесна комуникација, првиот е на кој начин еден процес може да ја предаде информацијата на друг процес. Второто е како да се обезбеди два или повеќе процеси да не си пречат еден на друг кога изведуваат критични активности и третиот случај е како да се изведе прередување кога постојат меѓусебни зависности, како во случај кога еден процес чека на резултатите од извршувањето на друг процес. Оттука следуваат и трите начини за интерпроцесна комуникација како и проблемите кои можат да произлезат од тоа:

- *Делење.* Се бара взаемно исклучување на пристапот кон одреден ресурс заради оневозможување на меѓусебно влијание на процесите.
- *Синхронизација.* Еден процес известува друг процес за одредени настани и вториот процес чека на нив.
- *Комуникација.* Еден процес испраќа пораки кон вториот процес, пример за ова е парадигмата произведувач-потрошувач. Првиот процес се блокира кога баферот за пораките е исполнет, а пак вториот процес се блокира кога баферот е празен и чека на појава на некоја порака.



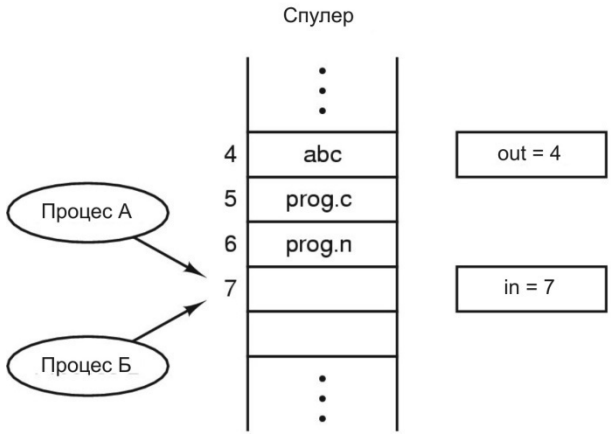
Слика 4.1: Три начини за интерпроцесна комуникација

4.1. Состојба на натпревар

Во одредени оперативни системи, процесите кои работат заедно можат да делат заеднички ресурси како што е мемориски склад или глобална променлива во кој може истовремено да запишуваат или читаат. Можат да се појават одредени проблеми поврзани со интерпроцесната комуникација заради истовремената работа со процесите.

Еден услов е да постои мултипроцесирање каде што два или повеќе процеси може да менуваат иста променлива симултано. Како и при опслужувањето на прекините, кога процесот ќе прими сигнал при промена на глобалната променлива, а сервисната рутина на сигналот ја менува истата променлива. И во самото јадро на оперативниот систем може да се појават одредени проблеми, јадрото се извршува како процес со повеќе нишки, каде што при појава на прекини, може да се појави измена на структурата на податоците во јадрото.

Последица од ваквиот конкурентен пристап кон ресурсите или податоците може да доведе до неконзистентност на податоците во мемориските локации или во датотеките. Конкурентноста како принцип е посакувана особина на оперативните системи и тоа заради мултипрограмирањето, каде што повеќе процеси работат на заеднички проблем, овозможува симултано извршување на програмите, како и извршување на програми во позадина. Од друга страна, конкурентноста претставува непријатност за оперативниот систем, затоа што се врши пристап до делени податочни структури и се појавува можност за блокада заради побарување на заедничките ресурси.



Слика 4.2: Проблем на синхронизација при печатење

Проблемот со синхронизацијата на конкурентните процеси може да се опише преку примери. Првиот е за пристап кон „спулерот“ на еден печатар, каде што еден процес сака да печати датотека, ја сместува во неговиот „спул“ директориум. Друг процес (даемонот на печатарот) периодично проверува дали има датотека за печатење и ако има ги печати и ги отстранува од директориумот. Ако се претпостави дека печатарот има „спул“ директориум со голем број на прегради каде секоја може да содржи име на датотека и ако се претпостави дека постојат две променливи со кои се обележуваат датотеките кои треба следни да се печатат (*out*) и променлива која покажува на следната слободна преграда во директориумот. Во еден момент, местата во директориумот од 0 до 3 се празни, а 4 и 6 се пополнети со имиња на датотеки

кои чекаат на печатење. Во исто време процесите А и Б стават датотеки за печатење.

Притоа процесот А ја чита вредноста на променливата *in*, во овој случај 7 и ја сместува како информација во својата локална променлива. Веднаш потоа настанува прекин генериран од системскиот часовник и процесот се заменува со процесот Б. Процесот Б исто така ја чита вредноста на променливата *in* која ја има вредноста 7 и исто така ја сместува во својата локална променлива и го запишува името на својата датотеката на позиција 8. Во тој момент и двата процеси имаат информација дека наредното слободно место во редицата за чекање во спулерот е позиција 8. Потоа процесот А повторно се извршува и го запишува името на својата датотека на позиција 8 и го препишува името на датотеката на процесот Б. Со тоа процесот Б нема да ја заврши својата задача и ќе продолжи да очекува излез од печатачот.

Ваквата ситуација каде што два процеси читаат или запишуваат во делени променливи или податоци и резултатот зависи од редоследот на извршувањето (спротивно од онаа што се очекува од оперативниот систем, да се однесува детерминистички) се нарекува состојба на натпревар (*англ. race condition*).

4.2. Модел на критична секција

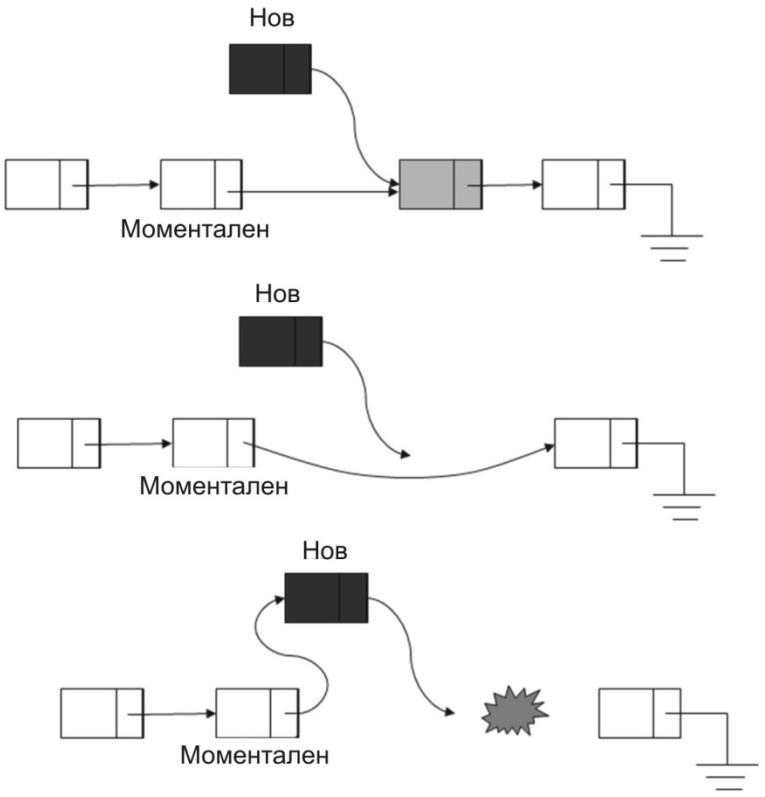
За да се избегне состојбата на натпревар, треба да се направи взаемно исклучување на пристап на процесите кон делените податоци. За да се подобро објасни концептот за взаемно исклучување даден е пример со операции врз членови на поврзана листа.

За да се вметне член во поврзана листа се користи процедурата *insert_after (current, new)*, при што првин покажувачот на членот што треба да се вметне добива вредност на покажувач на следниот член во листата, а потоа покажувачот од претходниот член се поставува да покажува на новиот член. Бришење на членот од листата се врши со тоа што покажувачот он претходниот член од оној што треба да се брише, се поставува на оној член од листата кој се наоѓа по оној што се брише.

Проблемот може да настане кога два конкурентни процеси се обидуваат истовремено да пристапат кон еден член од листата, при што едниот процес се обидува да вметне член пред него, а другиот да го избрише елементот. Како што може да се види од сликата, настанува проблем при поставувањето на покажувачите и раскинување на поврзаната листа.

```
insert_after(current,new):
      new->next=current.next
      current.next=new
```

```
remove_next(current):
       tmp=current.next;
       current.next=current.next.next;
       free(tmp);
```



Слика 4.3: Пример за критична секција кај поврзана листа

Овој проблем може да се надмине со користење на взаемно исклучување, односно исклучиво еден процес да пристапува и манипулира со членовите. Едно општо решение е да се користат таканаречени атомични операции, каде што сите низа операции се извршуваат како една, односно не може да настане прекин се додека не се извршат. Така што решението за примерот со поврзана листа би можело да биде:

`insert_after(current,new) remove_next(current),` или

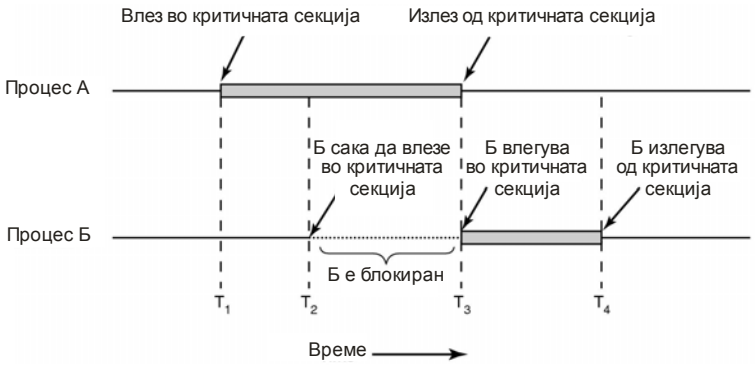
`remove_next(current) insert_after(current,new)`

Поимот критичен сегмент преставува одреден дел од програмата во кој се врши пристапување до делени податоци (променливи), датотеки и други ресурси. Критичниот сегмент (секција) претставува една апстракција што се состои од број на последователни програмски наредби што се извршуваат без прекин или грешки. Критичните сегменти се користат во оперативните системи за заштита на податочните структури како што се поврзани листи, делени прекини, сервисните рутини за прекините и други.

Во услови кога постои систем од n процеси, $P_0,\ldots,P_{n-1}$, не може да се претпостават релативните брзини или нема синхронизирани часовници, така што не може да се прецизира начинот како да се распоредат процесите правилно за да не се доведе во системот во состојба на натпревар. Тоа е и согласно на недерминистичкиот модел на извршување на процесите со кој треба да се справи оперативниот систем. Секој процес има сопствен критичен сегмент каде што менува вредности на променливи, табели, запишува датотеки кои се заеднички со други процеси и не постојат претпоставки за процесната активност во кој момент ќе се извршуваат во критичните сегменти.

Правилото за избегнување на состојба на натпревар може да се сведе на взаемното исклучување: кога еден процес работи во својот критичен сегмент, ниту еден друг процес нема дозвола да извршува во неговиот критичен сегмент. Иако предложениот концепт е коректен тој не е доволен за оптимално решавање на проблемот на натпревар. Условот на взаемно исклучување може да се прошири на четири услови:

1. Во критичниот регион може да има само еден процес;
2. Не се направени претпоставки за брзината и бројот на процесори;
3. Ниту еден процес кој се извршува надвор од критичната секција може да блокира друг процес;
4. Ниеден процес не смее да чека бесконечно за да влезе во својата критична секција.



Слика 4.4: Модел на критична секција

Моделот е прикажан на Слика 4.4, каде што еден процес влегува во својот критичен регион во одредено време, а малку подоцна друг процес се обидува да влезе во својот критичен регион но не успева затоа што таму веќе се извршува процес.

Следствено, вториот процес привремено се суспендира сè до моментот кога првиот процес ќе го напушти својот критичен регион, при што на вториот процес му е овозможено да влезе во својот критичен регион. По напуштањето и на вториот процес на критичната секција, системот доаѓа до првичната состојба кога во секцијата нема присутно ниту еден процес. Потребно е да се дизајнира протокол за соработка на процесите преку модел на критична секција кој содржи уште и влезна и излезна секција, кој ќе овозможи паралелни процеси ефикасно да се извршуваат и да пристапуваат кон делени податоци и ресурси. Примерот со поврзана листа може да се претстави со моделот на критична секција на следниов начин:

```
do{                     do{                     do{
Entry section           Entry section           Entry section

  Critical section      new->next=current.next  tmp=current.next
                        current.next=new        current.next=
Exit section                                    current.next.next
                        Exit section            free(tmp)
  Remainder section
} while(1)                Remainder section     Exit section
                        }while(1)
                                                  Remainder section
                                                }while(1)
```

Слика 4.5: Модел на критична секција кај поврзана листа

4.3. Решенија

Постојат повеќе решенија за имплементација на конкурентност и синхронизација на критичните сегменти на процесите. Покрај хардверскиот приод, постојат повеќе софтверски решенија кои може да се поделат на решенија со работно чекање (чекање на ред, заклучување и нивната комбинација), исклучување на прекините, блокирање (семафори, критични региони и монитори) и размена на пораки.

Секоја од можните имплементации треба да ги задоволи условите каде што само еден процес може да го извршува критичниот сегмент во еден момент, влегувањето и излегувањето од критичниот сегмент мора да се изврши брзо и ефикасно и на крајот добрата имплементација треба да биде флексибилна и да дозволува максимална конкурентност и има максимално можно малку ограничувања.

Коректното решение треба да ги има следниве својства:

1. *Взаемна исклучивост*, ако еден процес работи во критичен сегмент, ниеден друг на смее да работи во својот критичен сегмент (најмногу еден).
2. *Прогрес*, ако нема процес што моментално работи во критичен сегмент, а има процеси што сакаат да работат во критичниот сегмент, тогаш еден од нив добива дозвола (најмалку еден).
3. *Ограничено чекање*, мора да постои граница на бројот на дозволи која ја добиваат други процеси за влез во својот критичен сегмент пред да биде услужен даден процес што има барање да влезе во критичен сегмент (ниту еден процес не смее да чека бесконечно долго во својот критичен сегмент).

Концептуално, постојат три начини за да се задоволат барањата за конкурентност при имплементацијата, софтверски приод: каде што самите процеси да ја имаат одговорноста, системски приод преку обезбедување на поддршка во оперативните системи или програмските јазици и хардверски приод каде што се користат машински инструкции со специјална намена.

4.3.1. Взаемно исклучување со работно чекање

Взаемно исклучување со работно чекање претставува решение каде што еден процес е зафатен со пристапување кон делената меморија во својот критичен регион, така што ниту еден друг процес не може да влезе во својот критичен регион и евентуално да создаде непријатности.

Оневозможување на прекини

За системи кои не користат нишки, прекините се единствен извор на проблеми поврзани со конкурентност и синхронизација на процеси и нивните критични секции. Наједноставно решение за имплементација на критичните региони се состои во оневозможување на прекините пред влегување во критичниот сегмент и повторно вклучување по излезот од критичниот сегмент. Процесот може единствено да биде прераспореден од користењето на процесорот при појава на надворешен или внатрешен прекин (системски часовник). Со исклучени прекини, нема да постои временско ограничување на извршувањето на процесот во процесорот, значи ако нема појава на прекини нема да има ниту измена на процеси. Процесорот нема да биде доделен на користење на друг процес и тековниот процес може да ја чита и запишува во делената меморија без опасност од влијанието на друг процес. Ова решение не е најсоодветно затоа што на корисничките процеси не треба да им се даде можност да управуваат со прекините.

Исклучувањето на хардверските прекини може да биде опасно по системот, при што може да дојде до одложување на одговор на важни настани, во друга

рака пак, јадрото на оперативниот систем често ја користи ова техника додека ги обновува своите листи на променливи. Ако извршувањето на процесорот во критичниот сегмент е долго, може да се случи времето на одзивот на прекинот соодветно да се менува и може да се пропушти важен прекин. Потоа, во повеќепроцесорски систем критичниот сегмент може да биде извршен од друг процесор, исклучувањето на прекините се врши врз база на процесор и не може да се запре појава на прекин кај другиот процесор. Ако се појави грешка при извршувањето на процесот кој ги исклучил прекините и не се изврши повторно вклучување на прекините, системот може да се заглави во бесконечен циклус а бидејќи механизмот на прекини е тој кој го овозможува распоредувањето на процесите, ниту еден процес да не може да продолжи да се извршува. Сè на сè може да се каже иако јадрото го користи овој начин, тој не е соодветен како генерална техника за кориснички процеси за постигнување на взаемно исклучување.

Заклучување на променливите

Како друг начин може да се користи софтверско решение при што се користи единична делена променлива за заклучување *lock* иницијално поставена на вредност 0. Кога процесот сака да влезе во својот критичен регион ја тестира вредноста на *lock* ако е нула ја поставува на 1 и влегува во критичниот сегмент. Ако вредноста е веќе 1 процесот чека се додека не стане 0 (нема процес). Меѓутоа и овој пристап го содржи истиот проблем како што беше прикажан во примерот со „спул“ директориумот на печатачот.

Се појава на состојба на натпревар, каде еден процес чита вредност 0 и додека да ја постави вредноста на 1, друг процес се извршува и поставува вредност 1. Кога првиот процес доаѓа на ред за извршување и тој поставува вредност 1 и двата процеси се наоѓаат во критичниот сегмент истовремено.

Алгоритам на стриктна алтернација

Кај овој пристап процесите делат заедничка целобројна променлива *turn* со иницијална вредност 0 и преку неа се следи кој процес е на ред да влезе во критичниот сегмент. Првин еден процес со ознака 0 ја испитува променливата, наоѓа дека е 0 и влегува во својот критичен сегмент. Другиот процес со ознака 1 ја проверува променливата и затоа што вредноста на променливата *turn* е 0, чека во циклус при што цело време ја проверува променливата за да кога ќе стане 1 да влезе во секцијата. Чекањето во циклус и проверувањето на променливата троши процесорско време. Затоа овој пристап уште се нарекува работно чекање и треба да се избегнува заради трошење на времето на процесорот.

```
Shared: enum{0,1}turn;
Process Pi;

do{
turn=i;

while(turn!=i);
    Critical section
  Do nothing
    Remainder section
}while(1)
```

а.

```
Shared: enum{0,1}turn;
Process Pi;

do{

while(turn!=i);
    Critical section
  turn=1-i;
    Remainder section
}while(1)
```

б.

Слика 4.6: Пример за алгоритам за стриктна алтернација

Решението обезбедува дека само еден процес ќе влезе во својот критичен сегмент. Не е задоволен вториот критериум за прогрес. На пример, ако *turn=P1* и процесот *P2* е подготвен да влезе во критичниот домен, сепак P2 нема да може да го стори тоа, иако *P1* не е веќе во критичниот сегмент туку ги извршува инструкциите по него - Слика 4.6 а). Затоа секој процес пред излегувањето од својот критичен регион треба да ја постави вредноста на променливата за вториот процес.

Кога процесот 0 ќе го напушти регионот ја поставува променливата на 1 за да влезе процесот 1 во својот критичен регион. И овој процес (1) го напушта својот регион при што сега и двата процеси се вон регионот и вредноста на променливата е 0. Процесот 0 пак влегува во регионот го напушта и ја поставува *turn* на 1. Сега и двата процеси се наоѓаат вон своите региони. Процесот 0 се завршува и пак оди на влезот на регионот каде што не му е овозможено да влезе заради вредноста 1 а процесот 1 се уште го извршува кодот вон критичниот регион. Двата процеси имаат различна брзина на извршување, но правилото за стриктна алтернација го блокира извршувањето на процесот кој е побрз. Со тоа се нарушува 3-тиот услов за взаемно исклучување, процес е блокиран од процес кој не се наоѓа во својот критичен регион.

Критична секција без строга алтернација

Недостаток на претходниот пристап е тоа што тој не содржи доволно информации за состојбите за секој од процесите туку само на кој процес му е дозволено да влезе во својот критичен сегмент. Може да се направи модификација на алгоритамот со воведување на низа од Булови променливи со кои се искажува подготвеност за влез на некој процес во критичната секција.

На пример, наједноставно може да се опише за два процеси:

```
Shared: boolean flag[2]
Process Pi;

do{
flag[i]=true;

while(flag[1-i]);
    Critical section
  flag[i]=false;
    Remainder section
}while(1)
```

Се воведува низата *Boolean flag[P1..P2]*;

Ако *flag[Pi]* е *true*, значи дека i-тиот процес *Pi* е *подготвен* да влезе во критичниот сегмент;

i-тиот процес проверува дали j–тиот процес не е подготвен да влезе во својот критичен сегмент;

Ако и *flag[Pj]* е *true*, тогаш *Pi* чека додека *flag[Pj]* стане *false*, па тогаш *Pi* ќе влезе во критичниот сегмент;

Во овој случај не е задоволено второто барање за взаемно исклучување за прогрес на извршувањето на процесите. Можна е појава на застој (*англ. deadlock*) кога вредностите *Flag[P1]=true* , *Flag[P2]=true* што доведува до појава на бесконечен циклус и чекање на двата процеси.

Dekker-Peterson-ово решение

Со комбинација на алгоритамот на стриктна алтернација и заклучување на променливите се добива софтверско решение за взаемно исклучување кое не бара стриктна алтернација при што алгоритамот ги исполнува сите поставени барања. Променливата *flag* покажува дека некој процес има намера да влезе во својата критична секција, додека променливата *turn* покажува одредува кој процес има предност при влез во секцијата.

```
Shared: boolean flag[2]; enum{0,1}turn;
Process Pi;

do{
flag[i]=true;
turn=1-i;

while(flag[1-i] && turn==1-i);
    Critical section
  flag[i]=false;
    Remainder section
}while(1)
```

Иницијално ниту еден процес не се наоѓа во својот критичен регион. За да влезе во критичниот сегмент, процесот *Pi* го поставува *flag[Pi]* на *true* и го поставува *turn=Pj*, со што означува дека ако другите процеси можат да влезат во критичниот сегмент ако имаат намера. Процесот *Pj* чека да влезе во својата

критична секција само ако *flag[j]=1* и на процесот му е дадена шанса за влегување во секцијата *turn=j*.

```
#define FALSE       0
#define TRUE        1
#define N           2      //број на процеси

int turn;           //чиј ред е
int interested[N];  //иницијално сите се 0 (FALSE)

void enter_region(int process)    //процесот е 0 или 1
{
       int other;                 //број на друг процес

       other=1-process;           //обратно од процесот
       interested[process]=TRUE;  //покажува дека е заинтересиран
       turn=process;              //го поставува знаменцето
       while(turn==process&&interested[other]==TRUE)
}

void leave_region(int process)    //процес: кој напушта
{
       interested[process]=FALSE; }
```

Слика 4.7: Имплементација на Петерсоновото решение

Ова решение овозможува исполнување на сите услови, се почитува взаемното исклучување, заради променливата *turn* во критичната секција ќе биде присутен само еден процес. Не постои стриктна алтернација и процесот кој порано завршил со извршувањето на кодот вон критичната секција повторно може да влезе во секцијата без чекање на другите процеси и под претходно наведените услови. И не постои заглавување во бесконечен циклус затоа што не може да се појави ситуацијата истовремено да бидат и *turn=0* и *turn=1*, односно не постои можност и двата процеси да останат во јамката while. Дадена е имплементација на Петерсоновто решение за *N* процеси (Слика 4.7).

Хардверски примитиви

За хардверска изведба на взаемно исклучување со оневозможување на прекините потребни се специјални машински инструкции кои се применуваат во системите со повеќе процесори. Тие претставуваат елементарни блокови способни за неделиво изведување на одреден чекори во еден временски момент и тие треба да бидат универзални за да овозможат решавање на различни синхронизациони проблеми.

Една од таквите хардверски примитиви е TSL (*Test and Set Lock*), која ја чита содржината на мемориската локација *lock* во внатрешен регистер на процесорот и поставува не-нулта вредност на мемориската адреса *lock*.

Операцијата на читање и запишување на меморискиот збор е гарантирано неделива каде ниту еден друг процес (или процесор) не може да пристапи кон локацијата. Процесорот кој извршува TSL инструкција ја заклучува магистралата и оневозможува пристап кон меморијата се додека не се изврши.

За да се користи TSL инструкцијата, потребна е делена променлива *lock* за координација за пристапот кон меморијата. Кога вредноста на променливата изнесува 0, било кој процес може да ја постави на 1 со TSL инструкцијата и потоа да чита или запишува во деленатa меморија. По завршувањето процесот пак је враќа вредноста на 0 со обична инструкција за движење на податоците (*mov*). Даден е пример за користење на инструкцијата во виш програмски јазик со која се објаснува функционирањето на инструкцијата, едната функција ја поставува вредноста пред влегување во критичната секција, а функцијата *reset* ја поставува иницијалната вредност.

```
boolean test-and-set (boolean &lock)
{
       temp=lock;
       lock=TRUE;
       return temp;
}
```

```
reset (boolean &lock)
{
       lock=FALSE;
}
```

Моделот на критичната секција може да се престави на следниот начин со користење на дадените функции:

```
do {
       while (test-and-set(&lock));
              critical section;
       reset (&lock);
              reminder section;
} while(1);
```

Со *while* циклусот се испитува променливата *lock* со користење на *test-and-set* функцијата, ако е еднаква на нула содржината се пренесува во привремена променлива *temp*, ја поставува вредноста на *lock* на единица. Ако вредноста е при тестирањето на променлива е 1, тогаш постои процес кој се извршува во критичната секција и потребно е да се чека на промена на вредноста на *lock* на 0. На тој начин се оневозможува пристап на друг процес во критичната секција. Враќањето на вредноста на *lock* е едноставно со директно поставување на 0 во *lock*. Онаа што е различно од пристапот со заклучувањето на променливите објаснет претходно во текстот, е што хардверски е оневозможена појава на состојба на натпревар при пристапот кон заедничката променлива.

Користењето на хардверските примитиви - инструкциите *Test-and-Set (TS)* ги задоволуваат условите за взаемно исклучување и за прогрес на процесите, но не го задоволува условот за ограничено чекање (*англ. fairness*), додека не

излезе процесот од критичната секција, другите процеси треба да чекаат. Овозможува взаемно исклучување помеѓу неограничен број на процеси каде што идентификаторите на процесите и нивниот број е не е потребно да се познава. Меѓутоа најголем проблем останува работното чекање, односно, трошење на процесорските циклуси додека процесот е блокиран и ја проверува вредноста на заедничката променлива.

Предности и недостатоци на взаемно исклучување

За предностите при користењето на предложените методи може да се каже дека се применливи на било кој број на процеси на единичен или повеќе процесори при делење на главната меморија. Овозможуваат едноставна имплементација и верификација и може да се користат за повеќе критични региони.

Од друга страна се појавуваат и одредени недостатоци, од кои најзначајниот е тоа што предложените решенија користат работно чекање и трошење на процесорски циклуси. Можна е и појава на „изгладнување“ (*англ. starvation*) кога процес ќе го напушти својот критичен регион, додека чекаат повеќе од еден процес на влез во своите критични секции.

Критичните сегменти се несоодветни за програмирање, а се појавува и нарушување на перформансите заради работното чекање и се добива груба синхронизација. Можна е и појава на застој во ситуација наречена „инверзија на приоритети“. Ако процес со помал приоритет влезе во критичниот регион и постои процес со повисок приоритет тој ќе го добие процесорот на користење. Но тој не може да влезе во критичната секција заради процесот со помал приоритет се наоѓа во секцијата, но не може да излезе од неа бидејќи не му е доделен процесорот. Во тој случај, настапува застој и двата процеси чекаат, еден да го добие процесорот, а другиот извршува бесконечен циклус чекајќи на влез во критичниот регион. Користењето на хардверските примитиви пак директно резултира во непреносliв код.

4.3.2. Примитиви за интерпроцесна комуникација (Sleep, Wakeup)

И Петерсоновото решение и користењето на TSL инструкциите се исправни, но и двата користат работно чекање и не само што се трошат процесорски циклуси, туку и може да се појават неочекувани ефекти како проблемот на инверзија на приоритетите. Затоа се користат примитиви за интерпроцесна комуникација со кои се блокираат процесите наместо да се изведува работно чекање додека не им се дозволи влез во своите критични секции. Една од наједноставните примитиви е парот *sleep* и *wakeup*. Функцијата *sleep()* го суспендира и блокира процесот се додека некој друг процес на го разбуди со изведување на инструкцијата *wakeup()*. Функцијата *wakeup()* има еден параметар, а тоа е процесот кој треба да го разбуди и да го одблокира.

Проблемот на „производител-потрошувач“

Како пример каде може да се користат овие примитиви, е проблемот на производител-потрошувач (или познат како проблемот на ограничен бафер). Два процеси делат заеднички бафер со фиксна големина. Еден од процесите, производителот поставува елемент во баферот а потрошувачот го презема (концептот е објаснет во претходното поглавје). Проблемот настанува кога производителот сака да стави елемент во полн бафер и кога потрошувач сака да земе елемент од празниот бафер. Едно решение кое може да се користи е да се стави производителот во состојба на „спиење“ (*sleep*) односно блокиран и да биде разбуден од процесот потрошувач тогаш кога ќе се ослободи место во баферот со отстранување на еден или повеќе елементи. Слично важи и за потрошувачот, производителот ќе го разбуди тогаш кога во баферот ќе има еден или повеќе елементи со користење на функцијата *wakeup*. За да се следи состојбата на баферот се воведува променлива *count*. Ако максималниот број на елементи кои може да ги чува баферот е *N*, производителот врши проверка дали *count* е еднаков на *N*. Ако е процесот оди на спиење се додека не биде разбуден од другиот.

```
#define N 100   //број на слотови во баферот
int count = 0;  //број на производи во баферот

void producer(void)
{
 int item;

 while(TRUE){              //се повторува бесконечно
  item = produce_item();   //генерира следен производ
  if(count == N) sleep();
  insert_item(item);       //го вметнува во баферот
  count = count + 1;
  if(count == 1) wakeup(consumer);//дали баферот е празен?
 }

}

void consumer(void)
{
 int item;

 while (TRUE){             //се повторува бесконечно
  if(count == 0) sleep(); //ако баферот е празен, чекај
  item = remove_item();    //отстрани го производот
  count = count - 1;
  if(count == N - 1)wakeup(producer); //дали бафефор е празен?
  consume_item(item);      //печати го производот
}}
```

Можна е појава на состојба на натпреварување затоа што пристапот кон променливата *count* не е ограничен. Можна е појава на следнава ситуација. Баферот е празен и потрошувачот ја чита променливата *count* да види дали е еднаква на 0. Во тој момент распоредувачот го доделува процесорот на процесот произведувачот. Тој става елемент во баферот, ја зголемува вредноста на *count*, гледа дека е 1 и издава системски повик *wakeup* на потрошувачот. Потрошувачот не е логички во состојба на спиење, тој е само прераспореден од користењето на процесорот и *wakeup* сигналот се губи. Последица на ова е тоа што процесот производител ќе го наполни баферот и двата процеси заспиваат и никој не може да ги разбуди, односно настанува состојба на застој.

4.3.3. Семафори

Решенијата со работно чекање тешко се генерализираат за покомплексни проблеми и затоа се користи нова метода за синхронизација на процесите наречена семафор пронајдена од страна *Edsger Dijkstra* во 1968. Семафор е целобројна променлива со ненегативна вредност со која се заштитува некој ресурс и овозможува комуникација помеѓу процесите (механизмот на взаемно исклучување). Семафорот се состои од променлива со ознака *v* и листа на процеси *L*. Вредноста на *v* определува дали ресурсот кој е заштитуван од семафорот е слободен или заземен. Ако вредноста на $v = 0$ ресурсот е заземен, а ако $v > 0$ ресурсот може да се додели. Секој семафор има своја почетна вредност и врз него може да се извршат две неделиви операции (примитиви):

- Функција *Down* (*wait, sleep*) која го блокира процес (во текстот означена како *Up()*)
- Функција *Up* (*signal, wakeup*) – „буди“ процес (во текстот означена како *Down()*).

Функциите Up и Down може да се претстават на следниов начин:

```
Down (v)
{
       While (v>0);
       v--;
}
```

```
Up (v)
{
              V++;
}
```

Операцијата *down* ја намалува вредноста на семафорот само ако таа е поголема од нула. Во случај кога е еднаква на нула, процесот кој ја извршува оваа операција се блокира и чека вредноста да стане позитивна. Операцијата *up* ја зголемува вредноста на семафорот за еден. Иако се прикажани како функции со повеќе инструкции нивното извршување е атомично и тие се неделиви. Иницијалната вредност на семафорот покажува колку идентични инстанци на

критичниот ресурс постојат. А семафорот кој е иницијализиран на вредност 1 е наречен *mutex* (*англ. mutual exclusion*).

Функционирањето на семафорите може да се објасни со едноставен пример. Ако се замисли паркинг со наплатна рампа со одреден број на места за паркирање. Бројот на местата за паркирање може да се претстави со вредноста на семафорот. Ако има слободни места, вредноста v еднаква на бројот на местата, а кога сите места се зафатени вредноста на е $v=0$. На рампата постои светлосна сигнализација која свети зелено ако има слободни места, а црвено ако нема. Автомобил (процес) влегува на паркингот само ако свети зелено светло при што бројачот на слободни места се намалува за еден (операција *Down*). Ако светлото е црвено, автомобилите чекаат во редица за ослободување на ресурсот (паркинг позицијата). При излегување од паркингот (операција *Up*) се зголемува бројот на местата (променливата v) и светлосната сигнализација овозможува влегување на друг автомобил (процес).

Операции на семафорите

Дадена е можна имплементација на семафорите за решавање на проблемот на критичната секција и синхронизација на процесите. Имплементација за повеќе ресурси може да се прикаже со следниов код:

```
Initialization(val)  Down(s) (sleep)       Up(s) (wakeup)
{                    v--;                  v++;
       v = val ;     If (v <= 0)           if (v > 0)
       L = ⊥;        {                     {
}                    add process to L;     Remove a process P from L;
                     block;                Unblock(P);
                     }                     }
```

Кодот на овие функции се извршува атомично, што значи додека се одвива операцијата во семафорот, ниту еден друг процес нема достап до ресурсот кој се штити од страна на семафорот. Вметнувањата и отстранувањата од листата на процеси кои чекаат во листата L може да се прават случајно, но практично најчесто се користи FIFO.

Бинарни семафори

Поедноставена верзија на семафорот, каде што постои само една инстанца од ресурсот кој се дели, се нарекува бинарен семафор или *mutex.* Со *mutex* најдобро се изведува взаемно исклучување на процесите при пристап во критичната секција или некој друг делен ресурс. *Mutex* е променлива која може да зазема две состојби „заклучен“ или „отклучен“ и доволно е користење на еден бит за опис на состојбата.

Процес кој сака да влезе во својата критична секција повикува *down* на семафорот, ако вредноста на *v=0* тој се блокира и чека се додека друг процес не излезе од критичната секција и не направи повик за деблокирање на процесот и поставување на променливата на вредност 1. Процесот кој чекал досега се „буди“ и влегува во критичниот регион и пак ја поставува вредноста на v на 0.

Соработка помеѓу процесите чии активности се синхронизирани може да се обезбеди со користење на семафор „*proceed*“ со иницијална вредност 0. Механизмот за синхронизацијата може да се објасни преку примерот на два процеси 1 и 2. На првиот процес во точката *s1* му е потребна информација кој ја обезбедува процесот 2 кога ќе ја помине точката *s2*. Синхронизацијата може да се изведе на следниов начин.

```
Process 1                    Process 2
…                            …
…                            …
Down (proceed)               S2
S1                           Up (proceed)
…                            …
…                            …
```

Проблеми со семафорите

Користењето на семафорите може да предизвика и одредени проблеми. Лошата имплементација или програмерските грешки може да доведат до ситуација каде што еден или повеќе процеси бесконечно долго чекаат на настани кои нема да се случат. Тие процеси се наоѓаат во состојба на застој. Нека постојат процеси 1 и 2 кои следствено ги извршуваат следниве операции на семафорите *S* и *Q*, со иницијални вредности 1:

```
Process 1                    Process 2
Down (S)                     Down (Q)
Down (Q)                     Down (S)
…                            …
Up (S)                       Up (Q)
Up (Q)                       Up (S)
```

Овој редослед на извршување на операциите ќе доведе до појава на застој. Кога процесот 1 ќе ја изврши командата *Down (S)* вредноста на семафорот станува нула, како и за семафорот *Q* од страна на процесот 2. Процесот 1 чека на командата *Up (Q)* која може да се изврши само од страна на процесот 2 и обратно се чека на зголемувањето на вредноста на семафорот *S* од страна на

процесот 1. Оттука и двата процеси бесконечно чекаат еден на друг за зголемување на соодветниот семафор со што системот се доведува во состојба на застој.

4.4. Критични региони и монитори

Иако семафорите се добро решение на проблемот за синхронизација на процесите, сериозните грешки што може да бидат направени од страна на програмерот како што е покажано може да доведат до застој на системот или дури и појава на два процеси во критичната секција, што не е дозволиво.

Грешките може да се избегнат со користење на решенија со воведување на конструкции карактеристични за програмските јазици од повисоко ниво (*англ. high level language constructions*). Претпоставката е дека процесите се состојат од локални податоци и програмски код што оперира со податоците. Локалните податоци може да се пристапуваат само преку програмата што е енкапсулирана во самиот процес односно еден процес не може директно да пристапи кон локални податоци на друг процес.

Една таква концепција се нарекува критични или условни региони. Тие се состојат од взаемно исклучива (*mutex*) променлива *v* која се дели помеѓу процесите, условна променлива *B* и низа од наредби *S*. Кон променливата *v* може да се пристапи само во регионот на низата на наредби S преку изразот:

```
region v when B do S ;
```

Оваа конструкција означува: додека се извршуваат наредбите *S*, ниту еден друг процес не може да пристапи на променливата *v*. Изразот *B* е Boolean променлива и го управува пристапот да критичниот регион, кога процесот се обидува да влезе во критичниот регион, се проверува изразот *B* и ако тој е точен, се извршуваат наредбите *S*. Во случајот *B* да не е точен, процесот се откажува од променливата *v* и се блокира сè до оној момент додека *B* не стане точен. Кога *B* станува точно, блокираниот процес се буди и ако ниту еден друг процес не е во регионот поврзан со *v*, се извршуваат наредбите *S* во критичниот регион. Даден е пример каде што е опишан редоследот на извршување на наредбите и пристап кон променливата *v*. За почетна вредност $x = 0$ и дадени изрази за критични региони:

```
region v when x > 1 do S1;
region v when true do S4;
region v when x < 1 do S2;
x = 4;
region v when true do S3;
```

Се добива редоследот на извршување: *s4, s2, s1, s3*.

Проблемите кои може да се појават се: оперативниот систем треба да го најде моментот кога условот ќе стане точен и повторно да го пресмета така што процесот што чека да може да продолжи и со работа. Тоа е речиси невозможно да се направи ефикасно со денешната технологија каде што имплементацијата бара повторна евалуација на изразот *B* за секој процес што чека при излез на некој друг процес од критичниот регион.

4.4.1. Монитори

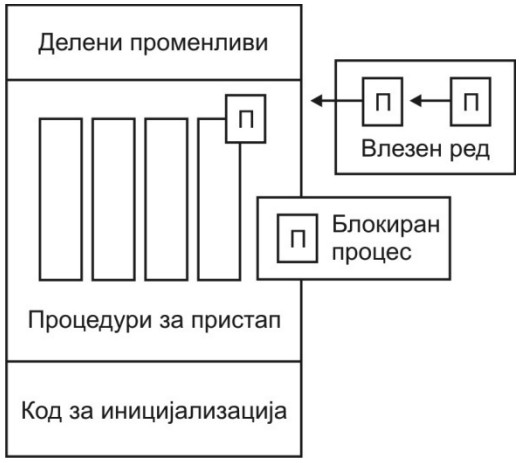
Монитори претставуваат синхронизациона примитива од повисоко ниво. Монитор е множество на процедури, променливи и податочни структури групирани во специјален вид модули. Процесите може да го повикаат мониторот во секој момент, но не можат директно да пристапат до внатрешните податочни структури на мониторот со процедури декларирани надвор од мониторот. Мониторите имаат битна карактеристика со која се постигнува взаемно исклучување: само еден процес може едновремено да биде активен во мониторот. Мониторите се конструкции од виши програмски јазици, така што преведувачот (*англ. compiler*) знае дека се посебни и повиците кон процедурите во мониторот се опслужуваат различно од другите повици кон процедурите. Обично кога се повикува процедура во мониторот, првите неколку инструкции од процедурата извршуваат проверка дали било кој друг процес е активен во мониторот. Ако има, процесот кој ја повикува процедурата се суспендира сè до моментот кога другиот процес ќе ја напушти процедурата.

Секој монитор се состои од: општи променливи (*a, b, c*) со кои се опишува делен ресурс, односно објектот чии вредности ја дефинираат состојбата на мониторот. Локалните променливи се видливи само во мониторот. Методи, множество на процедури (*fn1(...), fn2(...); fnm (...);)* преку кои се пристапува на објектот. Иницијален кон, конструктор кој се извршува само еднаш при креирањето на објектите и условни променливи *(x,y,z).*

```
Monitor M

{
        int a, b, c ;
        condition x, y, z ;
        fn1(…);
        fn2(…);
        …
        fnm(…);
}
```

Само еден метод (процес) во мониторот може да биде активен во еден момент (взаемно исклучување). Програмерот не треба експлицитно да програмира ограничувања заради синхронизација тоа го врши компајлерот и со тоа се

елиминираат проблемите го грешките при програмирањето. Со додавање на сите критични сегменти како процедури (функции) на мониторот, нема да има можност за појава на конкурентно извршување на процесите во критични сегменти.



Слика 4.8: Монитор

Доколку се потребни дополнителни синхронизациони механизми, програмерот треба сам да ги дефинира променливите со кои ќе се реши дополнителниот проблем. Ако дополнителниот синхронизационен систем е условна конструкција се дефинираат условни променливи. Со додавање на условни променливи, се збогатува управувањето на процесите. Над условните променливи може да се изведат две операции:

- *x.wait (),* каде што процесот кој ја повикува оваа операција се блокира (чека во ред), се додека некој друг метод (процес) не ја сигнализира променливата.
- *x.signal ()*, врши деблокирање на процес кој што чека на променливата.

Доколку нема процеси за деблокирање, повикот не работи ништо, за разлика од операцијата *signal* кај семафорите која секој пат ја зголемува вредноста на семафорот. Процесот кој ја продуцирал операцијата сигнал, го напушта мониторот (се блокира), а се активира процес кој чека на условната променлива *x*. Ако условната променлива е сигнализирана, а никој не чека на неа, сигналот се губи. Секој пат пред *signal* треба да се употреби *wait*.

Потребата од синхронизација помеѓу процесите се зголемува кога тие пристапуваат кон делени ресурси или податочни структури. Мониторот го енкапсулира ресурсот или податочната структура и присилува на взаемно исклучување на сите методи за пристап.

Процесот може да се блокира внатре во методот за пристап – може да чека на условна променлива:

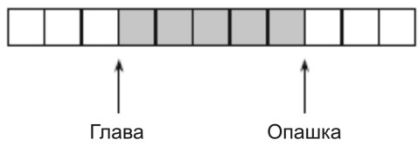
- кога тоа ќе се случи, друг процес може да влезе во мониторот;
- кога процесот сигнализира условна променлива, мониторот го избира првиот процес од редицата за таа условна променлива да продолжи;
- ново-разбудениот процес мора да чека на првиот процес да го напушти мониторот пред да влезе внатре.

4.5. Класични проблеми на интерпроцесна синхронизација

Сите наведени методи за интерпроцесна комуникација и синхронизација може да се претстават преку имплементација на три најпознати класични проблеми. Проблемот на произведувач/потрошувач (*англ. producer/consumer*) уште познат како проблем на ограничен бафер. Филозофите што вечераат и проблемот на читач/пишувач (*англ. reader/writer).* За сите три методи во понатамошниот текст ќе бидат презентирани начините на имплементација со класично блокирање, семафори, монитори и критични региони.

4.5.1. Проблем на ограничен бафер

Проблемот на ограничен бафер со неговото решение кое користи класично блокирање беше прикажан претходно. Примерно се користи за моделирање на работата на мрежни интерфејси, В/И контролери, размени на пораки и т.н.



Слика 4.9: Циркуларен бафер

Следниот пример го илустрира решавањето на проблемот произведувач-потрошувач со употреба на семафори. Синхронизацијата се обезбедува со бројачки семафори со вредности во интервалот *[0, N-1]* каде што N е капацитетот на баферот:

- *empty* (дали има слободно место во баферот, за да може произведувачот може да постави информација)
- *full* (да ли има податок на баферот, за да потрошувачот преземе информација од него)

Се воведува и дополнителен семафор *mutex* со кој се штити баферот како неделив ресурс. На тој начин се спречува можноста за забуна со покажувачите

која настанува кога еден процес чита информации од баферот, додека другиот запишува во него.

```
#include <prototypes.h>
#define N 100 // broj na slotovi
typedef int semaphore;
semaphore mutex=1, empty=N,
full=0;

void producer (void)
{
int item;

while (TRUE)
       {
       produce_item(&item);
              // sleden item
       Down (&empty); // poln?
       Down (&mutex); // kr. seg.
       enter_item (item);
       Up (&mutex); // kr. seg.
       Up (&full);
       }
}
```

```
void consumer (void)
{
int item;

while (TRUE)
       {
       Down (&full);
                     // poln?
       Down (&mutex);// kr. seg.
       remove_item(&item);
              // zemi item
       Up (&mutex); // kr. seg.
       Up (&empty);
       consume_item(item);
              // pecati
       }
}
```

Проблемот може да се реши и со користење на критични региони, кодот на функциите е прикажан на примерот, каде што произведувачот влегува во својата критична секција со тестирање на условот *count* < *N* и потоа се извршува низата од наредби со кои се поставува нов елемент во баферот и бројачот *count* се зголемува за еден.

```
Proizveduvac:

while (TRUE)
{
       region v when count < N do
       produce_item(&item);
       buf[tail++ % N] = item;
       count++;
}
```

```
Potrosuvac:

while (TRUE)
{
       region v when count > 0 do
       item = buf [head++ % N];
       count--;
       consume_item(item);
}
```

Следниов пример го претставува решението на проблемот произведувач и потрошувач со користење на монитори. Со условните променливи *full* и *empty* се контролира влезот во критичната секција. Произведувачот не може да постави елемент во баферот ако тој е полн туку се блокира сè до оној момент додека не биде сигнализиран од страна на потрошувачот (*full.wait* и *full.signal*). Слично и за потрошувачот, со променливата *empty* се блокира сè до оној момент кога производителот не сигнализира дека во баферот постои елемент кој може да биде преземен.

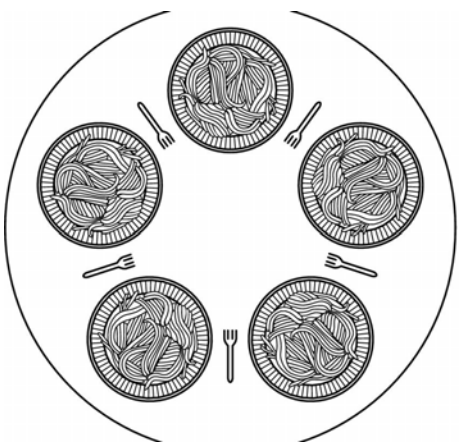
```
Monitor
BoundedBuffer
char buf [N];
int count;
int tail, head,
item;
condition empty,
full;
//inicijalizacija
count = tail =
head =0;
```

```
Enter(item)
{
if (count == N)
        full.wait;
buf [tail++ % N]=item;
count++;
empty.signal;
}

void produce ()
{
while (TRUE)
{
        produce_item(&item);
        Enter(item);
}}
```

```
Remove(item)
{
If (count == 0)
        empty.wait;
item = buf [head++ % N];
count--;
full.signal;
}

void consume()
{
while (TRUE)
{
        Remove(item);
        consume_item(&item);
}}
```

4.5.2. Филозофи што вечераат

Проблем на синхронизација опишан преку пример на филозофи кои вечераат предложен и решен од страна на *Dijkstra 1965*. Пет филозофи седат на кружна маса и пред секој од нив има чинија со шпагети. За да јадат, на секој од нив им требаат две виљушки, а помеѓу секоја чинија има само една.



Животот на филозофите се состои од наизменични периоди на јадење и размислување. Кога филозофот ќе стане гладен, се обидува да ги земе левата и десната виљушка и да јаде извесно време, а потоа да го врати приборот на масата и да продолжи да размислува. Проблемот се состои во овозможување на секој филозоф да јаде, без да се случи да се заглави само во размислување затоа што не може да пристапи кон потребните ресурси, во примерот виљушките. На примерот е опишано решение со користење на семафори.

Кога филозофите размислуваат ги оставаат стапчињата десно и лево. А кога јадат, ги земаат стапчињата лево па десно, при што се зема едно стапче во еден момент. Ако стапчето не е на маса, мора да се чека соседот што јаде да заврши и почне да размислува.

```
Semaphore stapce[5] = {1,1,1,1,1};
Semaphore mutex = 1;

void philosopher (int i)
{
while(TRUE)
{
       think();       // прво мисли
       Down(&mutex);
       Down(stapce[i]);
       Down(stapce[(i+1) % 5]);
       Up(&mutex);
       eat();              // јаде
       Down(&mutex);
       Up(stapce[i]);
       Up(stapce[(i+1) % 5]);
       Up(&mutex);
} }
```

Решението гарантира дека два соседи нема да јадат симултано. Без користење на наредбите *Down(&mutex);* и *Up(&mutex);* се добива најочигледното решение кое доведува до можна состојба на застој. Ако се претпостави дека сите пет филозофи ја земаат својата лева виљушка:

Филозоф 1 го зема стапчето 1

Филозоф 2 го зема стапчето 2

Филозоф 3 го зема стапчето 3

Филозоф 4 го зема стапчето 4

Филозоф 5 го зема стапчето 5

Сите чекаат еден на друг за ослободување на виљушка, при тоа не се наоѓаат ниту во состојба на размислување ниту во состојба на јадење. Затоа се користат семафорите а може да се подобри решението со кога еден филозоф ќе го земе левото стапче, ако десното е зафатено, да го врати левото и да чека некое случајно време. Решението е исправно, но не е оптимално затоа што само еден филозоф може јаде во еден момент, а можат и двајца кога има пет виљушки. Во следниов пример е дадено решението со користење на критични региони:

```
enum {Eating, Thinking} state[5];
for(i = 0; i < 5; i++)
       state[i] = Thinking;
// Entry(i):

region v when state[(i + 1) % 5] == Thinking && state[(i - 1) % 5] ==
Thinking do state[i] = Eating;
// Exit(i):
region v when TRUE do state[i] =Thinking;
```

Иницијално сите пет филозофи се доведуваат во состојба на размислување, кога некој од нив сака да јади, влегува во критичниот регион преку проверка на условот дали двете соседни виљушки се слободни, односно дали неговите соседи се наоѓаат во состојба на размислување. Решението обезбедува взаемно исклучување и не може да дојде до застој и изгладнување, но критичните региони се тешки за имплементација.

```
monitor dp {
enum {razmisuva, gladen, jadi}
sostojba[5];
condition filozof [5];

void pickup (int i)
{
       sostojba [i] = gladen;
       test[i];
       if (sostojba [i] !=jadi)
       filozof[i].wait();
}
void release (int i)
{
       sostojba[i]=misli;
       test((i+4)%5);
       test((i+1)%5);}
```

```
void test (int i )
{
If (sostojba[i]==gladen) &&
(sostojba[(i+4)%5]==jadi) &&
(sostojba[(i+1)%5]==jadi)
       {
              sostojba[i]=jadi;
              filozof[i].signal();
       }
}

void init()
{
 for (int i=0; i<5; i++)
       sostojba[i] = misli
}}
```

Во примерот со решението со користење на мониторите првин се дефинира структура на податоци *sostojba* која ја претставува моменталната состојба на секој филозоф. Секој филозоф може да се најде во една од трите можни состојби. Филозофот може да јади само ако неговите соседи размислуваат. Дополнително се воведуваат и условни променливи *sostojba [i]* која ќе го блокира филозофот кој е гладен а не може да ги земе двете виљушки.

Најпрвин во мониторот, сите филозофи се доведуваат во состојба на размислување (*misli*). Секој филозоф треба да ги изврши следниве две операции:

dp.pickup(i*)* и

dp.release(i)

Секој филозоф пред да започне да јаде, треба да ја повика процедурата *pickup* од мониторот *dp*. Во состав на таа процедура, филозофот може да ги земе двете виљушки или ќе биде блокиран со операцијата *filozof [i].wait*. Ако ги земе и двете виљушки, филозофот јаде, а потоа ги спушта на масата. Потоа се повикува процедурата *release*, која ги ослободува и двете виљушки и по потреба ако има такви и блокираните процеси.

4.5.3. Проблем на читање/пишување

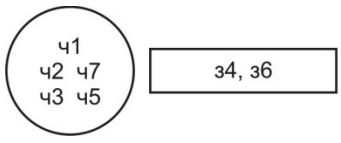
Проблемот на филозофите кои вечераат е корисен за моделирање на процеси кои се натпреваруваат за ексклузивен пристап на ограничен број на ресурси како што се на пример В/И уреди. Друг познат проблем е проблемот на читачи и запишувачи со кој се моделира пристапот кон податочни бази. Кај податочните бази прифатливо е да постојат повеќе процеси кои истовремено пристапуваат за читање на базата, но ако само еден процес запишува во базата ниту еден друг не смее да има пристап дури ни за читање.

Се користи при креирање на решенија за изведување на трансакции како на пример продавање на авио билети, банки и т.н. и кај системи на датотеки.

При решавање на овој проблем треба да се земат в предвид неколку услови. Прво да се изведат два типа заклучување, деливо за читање и ексклузивно за пишување. Притоа треба да се внимава кога се прави ексклузивно заклучување, ниеден друг процес на смее да има пристап, а кога се прави деливо заклучување, други процеси можат да читаат, но никој не може да добие ексклузивно заклучување (да запишува).

Постојат и два начини на работа каде што може да се преферира читање: ако дојде нов за читање тој влегува ако нема тековно запишување, т.е. тој влегува иако други веќе чекаат за запишување. И начин кога се преферира запишување: ако дојде нов за запишување, тој се опслужува колку е можно побрзо.

Пример. Пристигнуваат: *r*1 *r*2 *r*3 *w*4 *r*5 *w*6 *r*7



Се преферира читање

Се преферира запишување

Дадено е решение со користење на семафори. За пристап кон базата, процесот читач изведува *down* на семафорот *db*. Последователните читачи само го зголемуваат бројачот *rc*. По заминувањето на читачот, бројачот се намалува и последниот од нив изведува *up* на семафорот, при што се овозможува пристап на процес запишувач ако постои таков. За да се избегне ситуацијата каде што со константно појавување на процеси читачи, процесот запишувач би чекал многу време се додека сите читачи не завршат, се прави мала измена. Кога ќе пристигне читач и постои процес запишувач кој чека, тој ќе се суспендира наместо веднаш да влезе во критичниот ресурс.

```
semaphore mutex = 1;
semaphore db = 1;

int rc = 0;

void writer (void)
{
       while(TRUE)
       {
              ……...
              down(&db);
              write_db ();
              up(&db);
       }
}
```

```
void reader (void)
{
       while(TRUE) {
              down(&mutex);
              rc++;
              if (rc == 1)
              down(&db);
              up(&mutex);
              read_db ();
              down(&mutex);
              rc--;
              if (rc == 0) up(&db);
              up(&mutex);
              ……
       }
       }
```

Проблемот може да се реши и со користење на критични региони каде што е прикажан начинот кој преферира запишување.

```
Read()
{
       region v when !writing && !waiting do reading++;
       // read
       region v when TRUE do reading--;
}

Write()
{
       region v when TRUE do waiting++;

       region v when !writing && !reading do { waiting--;
       writing = 1}
       // write
       region v when TRUE do writing = 0;
 }
```

И прикажан е начинот кој преферира читање.

```
Write()
{
       region v when TRUE do waiting++;
       region v when !writing && !reading do { waiting--;
       writing = 1 }
       // write
       region v when TRUE do writing = 0;
}

Read()
{
       region v when TRUE do waiting++;
       region v when !writing do {waiting--; reading++;}
       // read
       region v when TRUE do reading--;
 }
```

4.6. Заклучок

Процесите може да комуницираат меѓусебно со користење на интерпроцесни примитиви како што се, семафори, монитори или пораки. Овие примитиви се користат за да обезбедат дека два процеси нема да бидат истовремено во својот критичен регион, ситуација која може да доведе до проблеми. Недозволен конкурентен пристап до делени податоци води кон неконзистентност. Конкурентност е голем извор на софтверската недоверливост. Взаемно исклучување е основната техника за која обезбедува дека однесувањето на системот е резултат сериско преплетување на логички операции врз делените податоци. Конкурентни системи бараат внимателен дизајн и валидација и се екстремно тешки за валидација со тестирање.

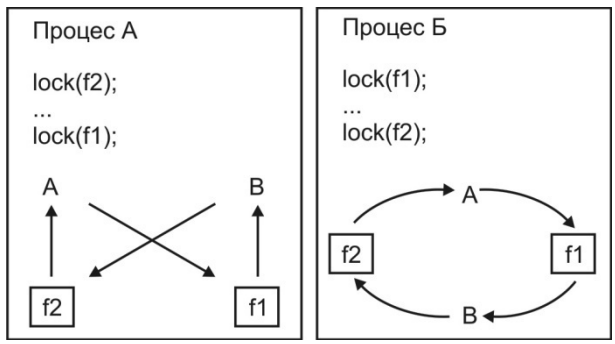
Процесот може да се извршува, да биде подготвен или блокира и може да ја промени својата состојба кога друг процес извршува некоја од примитивите за интерпроцесна комуникација. Овие примитиви може да се искористат за решавање на класичните проблеми како што се: произведувач-потрошувач, филозофи кои вечераат, читачи-запишувачи и друг. Дури и со користење на овие примитиви треба да се внимава за да не дојде до појава на грешки и застои.

5. Застои

На процесите им требаат хардверски и софтверски ресурси за да можат да се извршуваат, делови на компјутерскиот систем, процесори, меморија, диск и т.н. Процес кој чека на доделување на ресурс не може да го комплетира своето извршување сè додека ресурсот не стане достапен и се доведува во состојба на блокирање.

Компјутерските системи се карактеризираат со конечно количество на ресурси кои можат да бидат користени само од еден процес едновремено. Примери за такви ресурси се печатачите, магнетните ленти и други. Истовремено запишување на печатач од два процеси доведува до грешки и недоследности. Запишување на податоци во табелата на системот на датотеки истовремено од повеќе процеси доведува до оштетен систем на датотеки. За повеќе апликации, еден процес може да има потреба од ексклузивен пристап не само на еден туку на повеќе ресурси.

Во систем со мултипрограмирање вкупното побарување за ресурси од сите активни и конкурентни процеси го надминува количеството на достапни ресурси. Може да дојде до ситуација кога два или повеќе процеси (нишки) неограничено се чекаат меѓусебно. Множество процеси (нишки) се блокирани кога секој од нив чека за настан што може да биде предизвикан само од некој друг процес од множеството процеси.



Слика 5.1: Пример за блокада

Ако се претпостави дека два процеси сакаат да извршат снимање на скениран документ на оптички медиум. Процесот А бара и го добива на користење CD снимачот и бара дозвола за заземање на скенерот. Во меѓувреме процесот Б го зазел скенерот и бара дозвола за користење на CD снимачот. Во тој момент и двата процеси се блокирани и остануваат во таа состојба бесконечно време. Оваа ситуација се нарекува застој – блокада (*англ. deadlock*). Често се појавува кога се користат взаемни исклучувања како што е работа со критични региони

и тогаш може многу лесно да дојде до “циклично чекање” еден на друг процес. Застоите можат да се случат и помеѓу различни компјутерски системи, како на пример побарување на делени ресурси (печатачи, дискови итн.) преку локална компјутерска мрежа, а можат да се случат и на хардверските и на софтверските ресурси (пр. зафаќање на записи од податочна база).

5.1. Ресурси

Застои се случуваат кога процесите добиваат ексклузивен пристап на уреди, датотеки и т.н.. Под поимот ресурс се дефинира ентитет што може да се користи само од страна на еден процес (нишка) во даден момент. Ресурсите можат да бидат хардверски уреди или софтверски односно одредена информација или податок. Компјутерскиот систем има повеќе видови на ресурси кои може да се заземат, а за некои од нив може да постојат и повеќе инстанции кои може да се доделуваат. Со поголем број на идентични инстанции од еден ресурс, може да се задоволат повеќе истовремени барања од повеќе процеси.

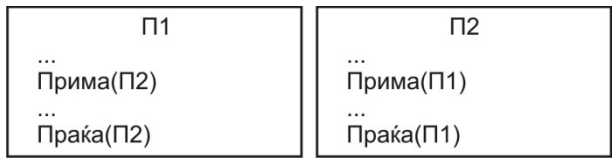
5.1.1. Категории на ресурси

Постојат два вида на ресурси: оние што можат да се испразнат (*англ. preemptable recourses*), односно кои можат да се одземат од процесот кој ги поседува во моментот без појава на негативни ефекти (пример за таков ресурс е главната меморија).

И ресурси што не може да се испразнат (*англ. nonpreemptable recourses*), односно кои не може да се одземат од процесот без да пропадне неговото извршување. Општо, блокадите се однесуваат на ова втората категорија. Потенцијалните блокади во случај на ресурси кои може да се испразнат може да се решат со доделување на ресурсите од еден процес кон друг. Може да се даде и друга дефиниција за категории на ресурси и тоа повторно искористливи кои се користени од еден процес едновремено и не се трошат при користењето. Можат да се користат и од други процеси и можат постојат повеќе инстанци, примери за овие се: процесори, меморија, дискови, магнетни ленти и т.н. И потрошливи ресурси кои се создадени (произведени) и уништени (потрошени) од самиот процес, како што се: прекини, сигнали, пораки и податоци во бафери на В/И уреди.

Слика 5.2: Блокада кај повторно искористливи ресурси

Застој може да се случи ако секој процес држи еден ресурс и побарува друг Како на примерот прикажана на Слика 5.2 каде што постои слободен простор за алокација на 200K. Друг пример е кога блокада се појавува ако пораката „*receive*“ се блокира (Слика 5.3).



Слика 5.3: Блокада кај потрошливи ресурси

Низа од настани кои се потребни при користење на ресурс:

- Побарај ресурс (*request*);
- Искористи го ресурсот (*use*);
- Ослободи го ресурсот (*release*);

Ако ресурсот не е достапен во моментот на побарување, процесот е принуден да чека. Во некои оперативни системи процесот автоматски се блокира и продолжува кога ресурсот пак ќе стане достапен. Во други системи пак барањето враќа код на грешка и процесот треба пак да се обиде по извесно време.

5.2. Аквизиција на ресурси

За некои видови ресурси, како што се записи во податочни бази, од самите кориснички процеси зависи управувањето со ресурсите. Еден начин е да се дозволи корисничко управување со ресурсите преку придружување на семафор за секој од ресурсите. Тие се иницијализирани на 1, а може да се користат и *mutex*-и. Трите чекори кои претходно се наведени се имплементирани преку користење на семафори за барање и заземање на ресурсите, нивно користење и на крај и нивно ослободување.

Понекогаш процесите имаат потреба од два или повеќе ресурси. Тие можат да бидат заземани секвенцијално како што е прикажано во следниот пример - Слика 5.4 б). Ако е потребни повеќе од два ресурси тие се заземаат еден по друг. Во ситуација кога има два процеси А и Б и два ресурси. И двата процеси ги бараат ресурсите по ист редослед во првиот пример, а по различен редослед во вториот пример, иако изгледа дека се исти сепак постои разлика. Кај првиот пример на Слика 5.4 а) еден од процесите го зазема првиот ресурс пред тоа да го стори вториот процес. Потоа, процесот успешно ќе го заземе и вториот ресурс. Ако вториот процес се обиде да го заземе ресурсот 1 пред првиот процес да го ослободи, ќе се блокира сè до моментот кога ќе стане слободен.

```
typedef int semafore
semafore resource_1;

void process_A (void)
{
       down(&resource_1);
       use_resource_1( );
       up(&resource_1);
}
```

а.

```
typedef int semafore
semafore resource_1;
semafore resource_2;

void process_A (void)
{
       down(&resource_1);
       down(&resource_2);
       use_resource_1( );
       use_resource_2( );
       up(&resource_2);
       up(&resource_1);
}
```

б.

Слика 5.4: Еден процес со еден ресурс и два ресурси

Во вториот пример од Слика 5.5, ситуацијата е поинаква. Може да се случи да еден од процесите да ги заземе и двата ресурси и да го блокира другиот процес сè до неговото завршување. Меѓутоа може да се случи процесот А да го заземе ресурсот 1, а процесот Б да го заземе ресурсот 2. Секој се блокира чекајќи еден на друг да го ослободи својот ресурс. Тоа доведува до појава на застој. Значи само мала промена во стилот на програмирањето може да доведе до застој кој тешко се детектира и проблеми при извршувањето на програмата.

```
typedef int semafore
semafore resource_1;
semafore resource_2;

void process_A (void)
{
       down(&resource_1);
       down(&resource_2);
       use_both_resources( );
       up(&resource_2);
       up(&resource_1);
}
void process_B (void)
{
       down(&resource_1);
       down(&resource_2);
       use_both_resources( );
       up(&resource_2);
       up(&resource_1);
}
```

а.

```
typedef int semafore
semafore resource_1;
semafore resource_2;

void process_A (void)
{
       down(&resource_1);
       down(&resource_2);
       use_both_resources( );
       up(&resource_2);
       up(&resource_1);
}
void process_B (void)
{
       down(&resource_1);
       down(&resource_2);
       use_both_resources( );
       up(&resource_1);
       up(&resource_2);
}
```

б.

Слика 5.5: Појава на застој заради промена на редоследот на извршување

5.3. Поим за застои

Застој (блокада) може да се дефинира преку: Множество на процеси е блокирано ако секој процес во него чека на настан кој може да биде предизвикан од друг процес во множеството. Бидејќи сите процеси чекаат, никој од нив нема да предизвика настан на кој чекаат останатите членови од множеството и така продолжуваат да чекаат во бесконечност. За овој модел се претпоставува дека сите процеси имаат само една нишка и дека не постојат прекини кои би го разбудиле процесот. Во повеќето случаи, настанот на кој чекаат блокираните процеси е ослободување на некој ресурс кој е заземен од друг блокиран процес. Ниту еден од процесите не може да се извршува, никој од нив не може да заземе ресурс и ниту еден не може да се разбуди. Бројот на процесите и бројот и видот на ресурсите кои се бараат и се заземени не е воопшто битен. Овој резултат (застој) ќе се појави за било кој тип на ресурси и хардверски и софтверски.

5.3.1. Услови за застој

Постојат четири услови кои треба да се исполнети за да дојде до појава до застој:

- Взаемно исклучување (*Mutual Exclusion*)
- Држи-и-чекај (*Hold-And-Wait*)
- Нема изпразнување (*Nonpreemption*)
- Кружно чекање (*Circular Wait*)

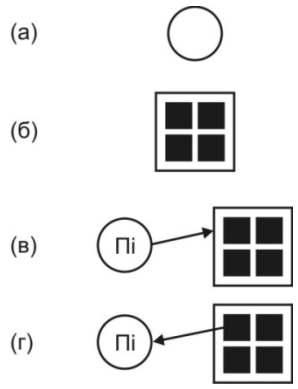
Сите четири услови мора да бидат истовремено исполнети за да дојде со застој. Условот за взаемно исклучување може да се објасни кога само еден процес во еден момент може да го користи ресурсот. Условот држи и чекај се карактеризира преку состојбата каде што процес кој држи најмалку еден ресурс чека да заземе дополнителни ресурси зафатени од други процеси.

Без испразнување се однесува на случајот каде ресурсот може да се ослободи само доброволно од процесот кој го држи, откога процесот ќе заврши со работата. За да се исполни условот за кружно чекање треба да постои множество $\{P_0, P_1, \ldots, P_n\}$ на процеси кои чекаат и тоа P_0 чека на процесот кој го држи P_1, P_1 чека на процесот кој го држи P_2, …, P_{n-1} чека на процесот кој го држи P_n и P_n чека на процесот кој го држи P_0. Сите четири услови треба да бидат исполнети за да се појави застој. Ако некој од нив не е, тогаш нема да може да дојде до состојба на застој.

5.3.2. Моделирање на застои

Овие 4 услови можат да бидат моделирани со користење на директни графови на доделени ресурси. Графовите можат да имаат два вида на јазли, процеси

(прикажани како кругови) и ресурси (квадрати). Стрелка од ресурсот кон процесот покажува дека ресурсот е побаран, дозволен и заземен од процесот. Стрелка од процесот кон ресурсот покажува дека процесот е блокиран чекајќи на ослободување на ресурсот.



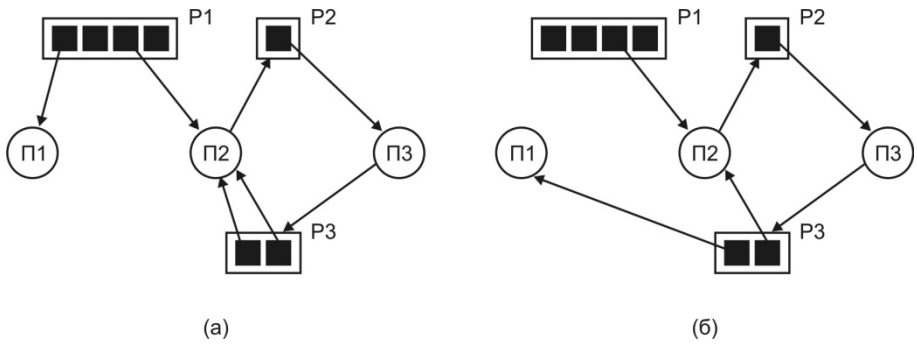
Слика 5.6: Процес, ресурс со 4 инстанци, Пі побарува инстанца од Р, Пі држи инстанца од Р

Графовите се состојат од множество на објекти и множество на стрелки *E*. Множеството на објекти содржи:

- Множество на сите активни процеси во системот $P=[P_1,P_2,...P_n];$
- Множество на расположливи ресурси $R=[R_1,R_2, ... R_n];$

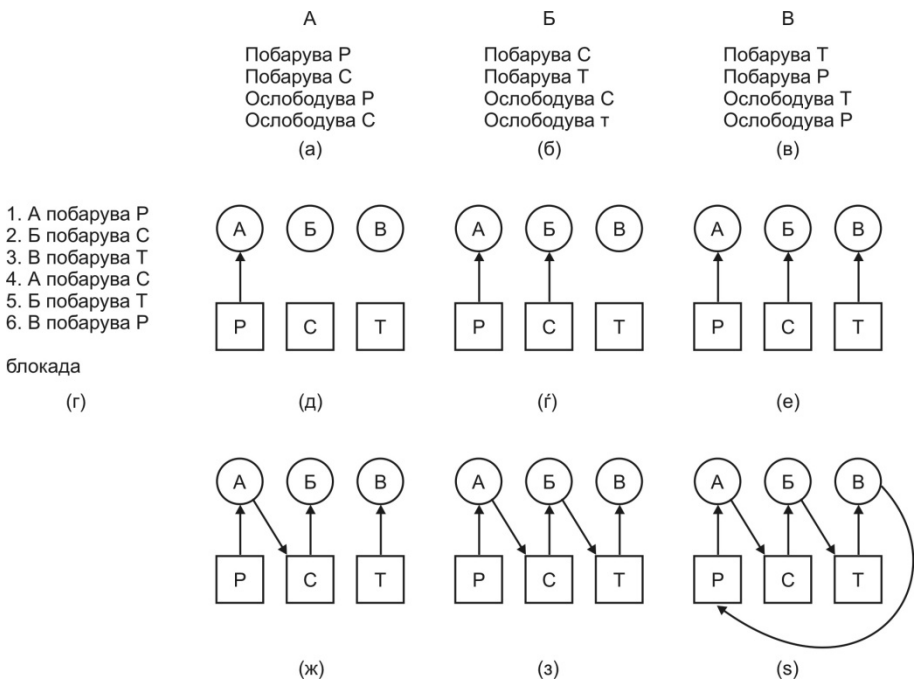
Стрелката за побарување (P_i->R_j) покажува дека процесот P_i бара една инстанца од ресурсот R_i и чека на неа. Стрелката на доделување $(R_j$->$P_i)$ покажува дека ресурсот R_i е доделен на P_i. Графот на доделени ресурси се создава со тоа што се вметнува стрелка во графот секој пат кога еден процес бара ресурс, а стрелката на доделување секој пат кога еден ресурс ќе се додели на процес.

Од графот може лесно да се согледа состојбата на застој: Ако графот не содржи кружен ток, сигурно не постои застој. Ако графот содржи најмалку еден кружен тек, можна е појава на застои. Кога сите ресурси кои имаат една инстанца припаѓаат на кружниот тек, се случил застој. Ако ресурсите во кружниот тек имаат повеќе инстанци може и да не се случи застој, како што е илустрирано на следниве два примери.



Слика 5.7: а) Кружно чекање, б) нема кружно чекање

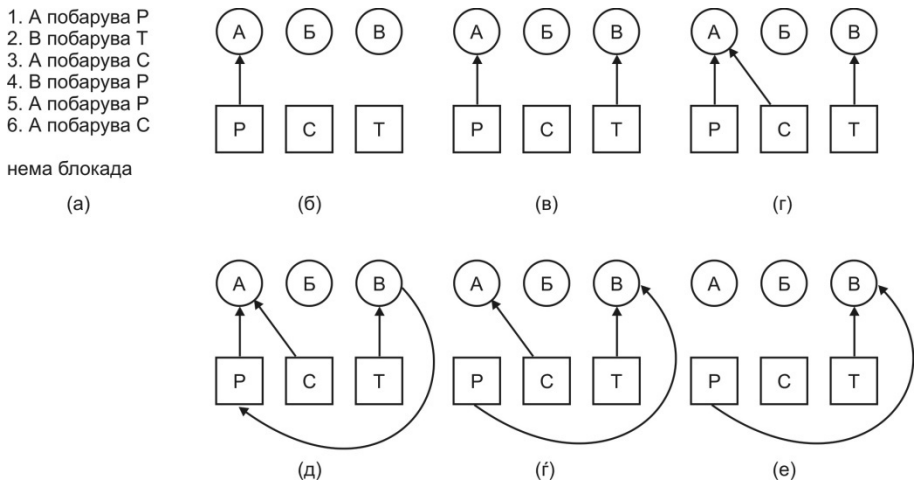
На следниот пример (Слика 5.8) е покажано како може да се одбегнат застоите со измена на редоследот на распоредувањето на процесите. Ако се претпостави дека процесите изведуваат и пресметки и В/И операции, како алгоритам за распоредување може да се земе *Round-Robin*. Побарувањето на ресурсите може да се одвива како на сликата. Постојат 6 барања за ресурси кои се извршуваат последователно, со што се добива графот. По изведувањето на 4–тото барање, процесот *А* се блокира заради чекање на ресурсот *С*. Во следниве два чекори *Б* и *В,* исто така се блокираат, со што се доведува да циклус и до појава на застој.



Слика 5.8: Избегнување на застоите со измена редоследот

Како што е веќе споменато, од оперативниот систем не се бара да извршува процеси по одреден редослед. Ако со дозволување на одредено барање за ресурс од страна на процес се предизвикува застој, оперативниот систем може да го суспендира процесот сè до момент кога е безбедно да се задоволи барањето.

Во вториот дел од примерот на оперативниот систем му е познато дека ќе се појави застој и може да го суспендира процесот *Б* наместо да му го додели ресурсот *С*. Со извршувањето на процесите *А* и *В*, се ослободуваат нивните ресурси, со што овој редослед на извршувањето предизвикува графот на доделувањето на ресурсите да нема кружна патека. На процесот *Б* понатаму може да му се дозволи заземање на ресурсот *С*, дури и да се појави блокирање на *Б*, доволно е да се почека да се заврши процесот *В*.



Слика 5.9: Избегнување на застоите со измена редоследот

5.4. Справување со застои

Графовите на доделувањето на ресурсите се алатка со која може да се види дали секвенцата на побарување или ослободување на ресурсите доведува до застој. Се изведуваат чекорите за барања и ослободувања еден по еден и по секој чекор се проверува дали графот содржи циклуси. Ако постојат, тогаш се случил застој, ако не тогаш до тој момент нема застој во системот. Постојат 4 стратегии за справување со застоите:

1. Не прави ништо, која најчесто се користи во реалноста, *„ако го игнорирате проблемот, може и тој да ве игнорира вас“*.
2. Детекција и опоравување, да се дозволи појава на застој, истиот да се детектира и да се преземат соодветни мерки.

3. Динамичко одбегнување, одбегнување на појава на застој со внимателно доделување на ресурсите кон процесите.
4. Превенција, да се реализираат проблемите на конкурентност во оперативниот систем, така што застој не може да се случи, преку избегнување на некој од условите за појава на застоите.

5.4.1. Алгоритамот на нојот (The Ostrich Algorithm)

Наједноставното решение е „алгоритамот на нојот“ (*англ. ostrich algorithm*): да се забие главата во песокот и да се претвора дека проблемот воопшто и не постои. Различни профили на стручњаци имаат различен став за овој пристап. Математичарите наоѓаат дека е потполно неприфатлив и дека застоите треба по секоја цена да бидат спречени. Инженерите се прашуваат колку често се појавува проблемот, колку често се руши системот поради тоа и колку е сериозна појавата на застој. Ако застојот се појавува еднаш во пет години, а компјутерскиот систем испаѓа од функција заради хардверски проблеми, грешки при компајлирањето и багови во оперативниот систем еднаш неделно, повеќето инженери не би сакале да ги нарушат перформансите на системот заради додавање на дополнителен код за елиминација на застоите. Некои оперативни системи страдаат од појава на застои кои дури не може ни да се детектираат. На пример, бројот на процесите кои се извршуваат во системот е ограничен од големината на процесната табела. Според тоа процесната табела е конечен ресурс. Ако системскиот повик *fork* не успее затоа што процесната табела е исполнета, разумно би било програмата да почека извесно време и пак да се обиде. Ако се претпостави дека еден UNIX систем има можност за извршување на 100 процеси. Десет програми се извршуваат од кои секој има потреба да креира 12 подпроцеси (процеси –деца).

Секој процес креирал 9 процеси, значи 10 оригинални процеси и 90 процеси деца ја пополниле табелата. Секој од 10 оригиналните процеси врти во бесконечен циклус обидувајќи се да го изврши системскиот повик *fork* и паѓа и се појавува застој. Веројатноста ова да се случи е многу мала, но можноста постои.

Слично на ова може да се види и кај системите на датотеки, каде што максималниот број на отворени датотеки е ограничен од големината на i-јазлите, пак може да се појави проблем. Во суштина секоја табела во системот претставува ограничен и конечен ресурс. Повеќето оперативни системи како што се UNIX и Windows го игнорираат проблемот, претпоставувајќи дека корисниците се подготвени да доживеат и по некој застој по цена на перформансите. Справувањето со застоите претставува правење на компромис помеѓу комфорноста и коректноста на работата на компјутерскиот систем. Кое решение ќе се користи зависи најмногу од потребата која треба да се задоволи.

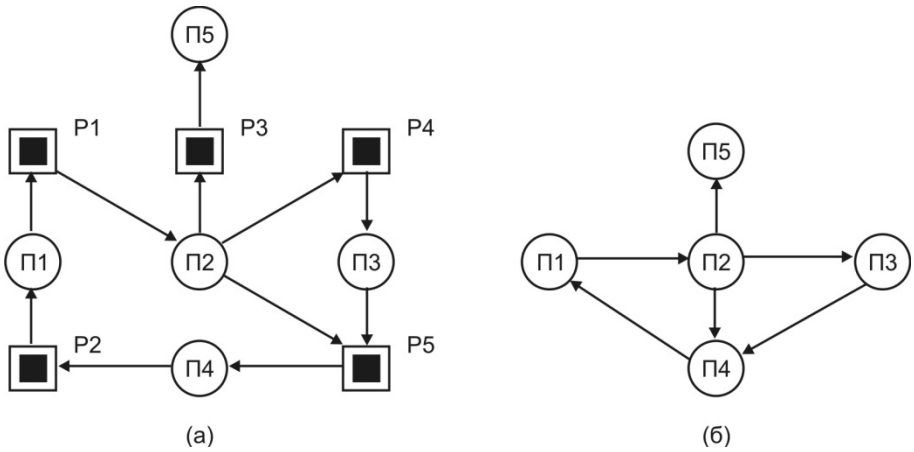
5.4.2. Детекција и закрепнување

Втората техника е детекција и закрепнување. Системот не се обидува да изврши превенција на појавување на блокадите. Наместо тоа, се анализира ситуацијата, се детектира блокадата и се преземаат чекори за закрепнување од настанатата ситуација, како на пример да се одбере процес која е член на кружниот тек на графот на доделување на ресурсите и да се терминира. При изведувањето на техниката за детекција и закрепнување се поставуваат повеќе прашања, како на пример, како да се препознае појавата на застој. Потребен е формален алгоритам за детектирање циклуси во ориентиран граф, или некој матричен алгоритам и како да се препознае кој од процесите се во застој за да се преземе соодветна акција.

Детектирање на застој со еден ресурс од секој тип

Наједноставниот случај е кога постојат по еден ресурс од секој тип. Таков систем може да има еден скенер, една магнетна лента, оптички уред, но не повеќе од еден примерок од секоја класа на уреди (по една инстанца). За ваков систем се конструира граф за чекање на ресурси (*англ. wait for graph*) од графот на доделување на ресурсите. Ако во графот на чекање постои кружен тек, системот е во застој, ако не постојат, системот не е блокиран. Правилата за конструирање на графот се следниве:

- Јазлите на графот се само процеси (не постојат ресурси);
- Стрелките се поставуваат само помеѓу процесите кои чекаат еден на друг за да ги добијат потребните ресурси.



Слика 5.10: Конструирање на граф за детекција на блокади

Периодично се повикува алгоритамот за да се испита дали постојат циклуси, за што е потребно n^2 операции за граф кој содржи n стрелки. Постојат повеќе

алгоритами за детекција на циклуси во графот за чекање. Даден е пример за еден едноставен алгоритам каде што се испитува графот и алгоритамот се завршува кога ќе се најде циклус, или кога воопшто нема да се најде дека постои таков. Се користи податочна структура *L* како листа од јазли. При извршувањето на алгоритамот, сите стрелки ќе бидат одбележени со што се покажува дека веќе биле проверени за да се избегне повторување на проверките.

Алгоритамот функционира преку изведување на следниве чекори:

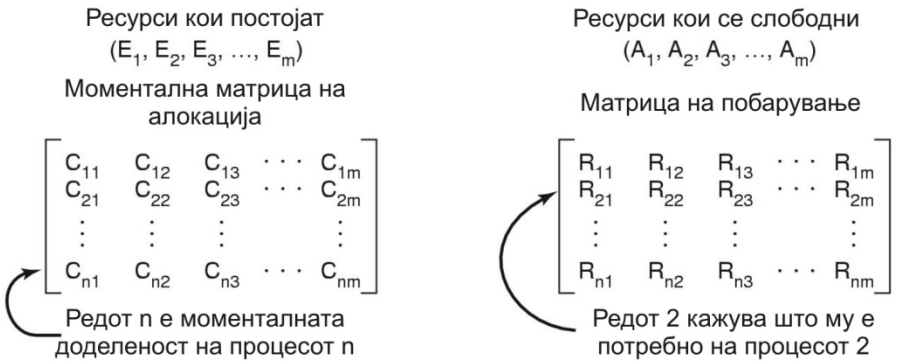
1. За секој јазол *N* во графот, да се поминат овие 5 чекори со *N* како почетен јазол.
2. Иницијализација на *L* како празна листа и означување на сите стрелки како неозначени.
3. Додавање на тековниот јазол на крајот од *L* и проверка дали се појавува во листата два пати. Ако се појавува. Графот содржи циклус и алгоритамот се завршува.
4. За дадениот јазол, да се провери дали има неозначени стрелки кои потекнуваат од него. Ако има се скока на чекорот 5, ако не се оди на чекорот 6.
5. Се избира по случаен пат неозначена стрелка и се одбележува, се поминува кон нов тековен јазол и се оди на чекорот 3.
6. Се стигнува до крајот, се отстранува од листата и се оди на претходниот јазол кој станува тековен и се оди на чекорот 3. Ако јазолот е иницијален, графот не содржи никакви циклуси и алгоритамот се завршува.

Алгоритамот може да се илустрира на дадениот пример на Слика 5.10. Редоследот на изведувањето на алгоритамот е произволен и може да се земе насока од лево кон десно и од горе кон долу, со изведувањето на алгоритамот врз јазолот P_5. Па потоа P_1, P_2, P_3 и P_4. Ако се детектира циклус, алгоритамот завршува.

Се започнува со P_5 и се иницијализира L на празна листа. Се додава P_5 на листата и бидејќи нема стрелки кои потекнуваат, алгоритамот тука завршува и се поминува на следниот процес P_1. Листата се пополнува и изгледа вака $L=\{P_1,R_1,P_2\}$ и тука се прави случаен избор на немаркирана стрелка. Се избира на пример, стрелката кон R_4 и тука листата продолжува сè до P_1 и се добива листата $L=\{ P_1, R_1, P_2, R_5, P_4, R_2\}$, бидејќи сите стрелки оттука понатаму се маркирани, алгоритамот се враќа назад кон P_2 и се избира од немаркираните стрелки, нека биде кон R_4 и патеката се додава кон крајот на листата $L=\{ P_1, R_1, P_2, R_5, P_4, R_2, R_4, P_3, R_5\}$, ресурсот R_5 се појавува два пати во листата со што се покажува дека постои циклус во графот и алгоритамот се прекинува.

Детектирање на застои со повеќе ресурси од ист тип

Алгоритамот за детекција со помош на графови на чекање не е соодветен кога ресурсите имаат повеќе од една инстанца. Во тој случај се користи алгоритам со матрично претставување за детекција на застој помеѓу n процеси од P_1 до P_n. Бројот на класи на ресурсите нека биде m и се воведува низа E наречена вектор на постоечки ресурси, каде што членовите се E_1- класа 1, E_m- класа m (на пример класа 1 е магнетна лента и $E_1=2$ означува дека системот има 2 ленти).



Слика 5.11: Матриците и векторите кои се користат при детекција на застои со повеќе вида на ресурси

Во секој момент некои од ресурсите се доделени на процеси и не се достапни. Со A се означува вектор на достапни ресурси, каде што со A_i се означува бројот на инстанците кои се расположливи, на пример ако и двете магнетни ленти се користат, тогаш A_1 е еднаков на 0. Потребни се и две низи C, матрица на заземени ресурси и R матрица на побарување. Редицата со ознака i на C покажува колку инстанци од секоја класа на ресурси се заземени од процесот P_i. Така C_{ij} е бројот на инстанци од класа j се заземени од процесот i. слично R_{ij} е бројот на инстанци на ресурсот j што ги бара процесот P_i.

За ресурсите важи дека тие може да бидат заземени или слободни, ова значи дека:

- $E_j = A_j + \Sigma C_{ij} (i=1..n)$
- $C_{kj} \leq E_i$: ниеден процес може да држи повеќе од вкупниот број на ресурсите во системот
- $R_{kj} \leq C_{ki}$: ниеден процес не побарува повеќе отколку што првично изјавил дека има потреба

Алгоритамот за детектирање на блокада функционира преку споредба на вектори. Се дефинира релација $A \leq B$ за два вектори A и B која значи дека секој елемент од A е помал или еднаков соодветниот елемент на B, $A_i \leq B_i$ за $1 \leq i \leq m$.

Секој процес иницијално не е маркиран. Со работа на алгоритамот, оние процеси што може да се извршат и не се во застој, се маркираат. Алгоритамот е даден преку следниве чекори:

1. Се бара немаркиран процес P_i, за секоја i-та редица од R е помала или еднаква на A.
2. Ако е пронајден таков процес, се додава редицата i од C кон A се маркира процесот и се оди на чекор 1.
3. Ако нема такви процеси алгоритамот завршува.
4. Оние процеси што ќе останат немаркирани се блокирани.

Алгоритамот всушност ги бара процесите кои може да работат до крај на нивното извршување. Процесот има побарувања за ресурси кои може да бидат задоволени со моментно достапните ресурси. Селектираниот процес се извршува до крај и ги ослободува сите ресурси кон множеството на достапни ресурси во системот. Потоа се маркира како комплетиран, ако некои од нив не може да се завршат тие се блокирани. Без разлика на нивниот редослед на извршување резултатот секој пат е ист. Даден е пример за работа на овој алгоритам за детекција на застои. Постојат три процеси и четири класи на ресурси,

Слика 5.12: Пример за алгоритам на детекција за застои

Процесот 1 држи еден скенер, процесот 2 зазема 2 магнетни ленти и CD-ROM. Процесот 3 еден плотер и еден скенер. Секој од процесите бара дополнителни ресурси како што е покажано во матрицата R. Алгоритамот бара процес чии барања може да бидат задоволени. Првиот не може да биде задоволен, затоа што нема слободен CD-ROM уред. И вториот не може да биде задоволен, затоа што нема слободен скенер. Третиот може да биде задоволен и се извршува до

крај со што ги враќа своите ресурси при што се добива *A=(2 2 2 0),* потоа се отвора можност да се заврши вториот процес со што ги враќа своите ресурси и се добива вектор на достапни ресурси *A =(4 2 2 1)* и последниот процес може да се изврши, така што во системот не се случил застој.

Примената на алгоритамот зависи и од фреквенција на појавата на застои и од бројот на процесите кои се опфатени во застојот. Алгоритамот може да се повикува секогаш кога ќе се побара ресурс што внесува дополнително оптоварување и трошење на процесорско време. Или може да се повикува на одредено време, или кога оптоварувањето на процесорот се спушти под некоја граница, затоа што повеќето процеси се во блокада, па многу малку се користи процесорот .

Закрепнување од застој

Откако се препознае застој треба да се изврши закрепнување на системот, односно да се преземат одредени активности. Двете најкористени методи се:

- Привремено одземање на ресурси од некој процес.
- Прекинување на процесот кој е во застој

Елиминација на застои со присилно прекинување на извршувањето на заглавените процеси може да се изведе на повеќе начини, а во секој случај се ослободуваат сите ресурси кои биле заземени од процесот. Постојат две техники:

1. Прекинување на сите заглавени процеси, процедурата може долго да трае и да има негативни ефекти. Добро е да се „убие“ процес што нема има проблеми при повторно стартување, како на пример компајлирање, но не ажурирање во база на податоци. Ако на пример еден процес додава вредност 1 на некој запис и ако потоа тој се терминира, при неговото повторно стартување, ќе додаде уште еднаш вредност 1 на записот, што е некоректно.
2. Прекинување на извршување на процесите до разбивањето на кружниот тек. Оваа метода е подобра, но потребно е да се изберат вистинските процеси и редоследот на прекинувањето на процесите. Во најдобриот случај се прекинува само еден процес. При изборот на процесот за прекинување, треба да се земат в предвид и одредени фактори како што се приоритет на процесот, колку долго работи и колку време останало до неговото завршување, кои ресурси ги користи, кои ресурси се бараат и дали тоа се интерактивни или групни процеси.

Елиминацијата на процесите со одземање на ресурсите не го прекинува извршувањето на процесите. На одреден процес (или на процеси) му се одземаат ресурсите. Се бира процес каде што одземањето на ресурсите нема да

предизвика поголема штета. Процесот на кој му се одземени ресурсите останува во состојба на блокиран, се чува безбедната состојба за да подоцна процесот продолжи понатаму со извршувањето. Треба да се избегне изгладнување на процесот на кој ќе му се одземе ресурсот, затоа што неговото извршување ќе трае подолго време од другите.

5.4.3. Одбегнување на застои

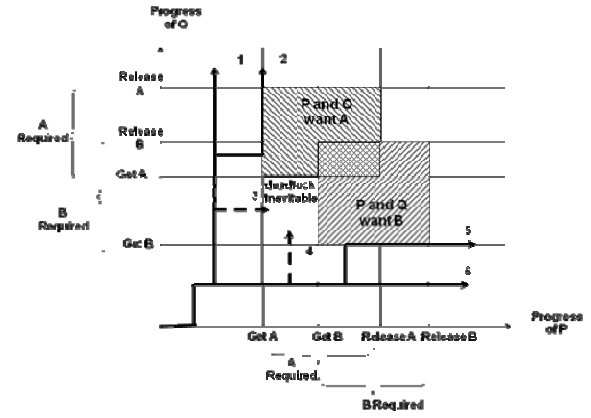
Кај најголемиот број системи ресурсите се побаруваат еден по еден, а кај техниката за детекција на блокади се претпоставува дека сите процеси ги побаруваат сите ресурси наеднаш. Оперативниот систем треба да одлучи дали е сигурно да се додели некој ресурс. Ресурсите треба да се доделуваат на начин со кој се обезбедува дека точката на блокада никогаш нема да биде достигната. Постои алгоритам со кој секогаш може да се избегнуваат блокади, но за да се реализира, потребно е да се знаат однапред одредени факти (кои и колку ресурси се потребни по процес).

Безбедна состојба

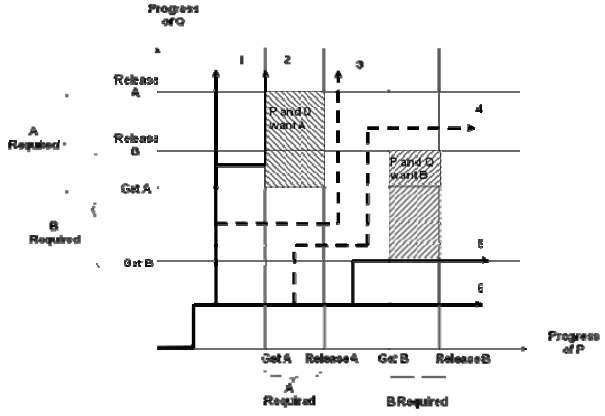
Главните алгоритами за одбегнување на застоите се засновани на концепт на безбедни состојби. За да стане поразбирлив, концептот е објаснет преку пример со два процеси и два ресурси. Хоризонталната оска го претставува бројот на инструкциите изведен од процесот *P*, а вертикалната оска бројот на инструкциите изведен од страна на процесот *q*. Во моментот *l1* процесот *P* го бара ресурсот *A*, а подоцна го бара и ресурсот *B*.

И двата ресурси се ослободуваат по извесно време. Слично, процесот *Q* првин го бара ресурсот *B*, а потоа ресурсот *A* и ги ослободува по истиот редослед по некое време. Секоја точка во дијаграмот претставува заедничка состојба на процесите, а со црни стрелки се прикажани патеките на извршувањето на инструкциите, во примерот, првин *P* извршува една инструкција, потоа *Q* една и така понатаму. Можни се повеќе патеки во зависност на редоследот на измената на извршувањето на процесите.

Засенчените области ги претставуваат временскиот моменти кога и двата процеси го зазеле ресурсот, што е невозможно заради условот на заемно исклучување. Влезот во делот покажан со стрелките 3 и 4 не е безбедна состојба, затоа што секој од процесите го држи ресурсот кој го бара другиот и не треба да се влезе во него.



а.



б.

Слика 5.13: Безбедна состојба и одбегнување на застои

Решението би било да се продолжи со извршување на процесот *P* (патека 5 или 6) сé до ослободување на ресурсот *A*, или продолжување на извршувањето на процесот *Q* (патека 1 или 2). Системот треба да одлучи дали ќе додели ресурс кон процесот во еден момент со тоа што ќе испита дали состојбата во која потоа ќе се најде системот е безбедна или не. На вториот графикон е покажано како може да се одбегне можната блокада со барање на ресурс од страна на процес и негово ослободување пред да биде побаран друг ресурс.

Пристап за одбегнување на застои

Одбегнување на блокади се изведува со тоа што, не се исклучува ниеден од 4 услови за застој *a priori*. Туку, одлуката се донесува зависно од случајот кога се цени дека процесот води кон можен застој.

Може да се раздвојат 2 пристапи за одбегнување на застои:

- Забрана за иницијализација на процесот: ако се процени дека е можен застој воопшто да не се стартува процесот кој може да го предизвика
- Забрана за доделување на ресурс: се забранува алокација на ресурсот кон процесот со што може да доведе до иден застој

Одлуката за доделување на ресурсите се донесува динамички и е заснована врз податочните структури опишани кај пристапот за детектирање на застои, вкупниот број од достапните ресурси (*E*), моментно достапни ресурси (*A*), побарувањето на ресурсите од страна на процесите (*R*) и моменталните доделувања на ресурсите кон процесите (*C*). Согласно ознаките, забраната на иницијализација на процесот се извршува под условот:

$$E_j \geq A_j + \sum C_{ij} \quad (i=1..n),$$

Што значи дека процесот ќе се стартува само под условот дека збирот на сите барани ресурси и доделените ресурси ќе биде помал или еднаков на вкупниот број на достапните ресурси во системот. Овој начин не е оптимален од причина дека процесите не ги објавуваат сите потребни ресурси при иницијализација, туку во текот на извршувањето може да се побараат и дополнителни ресурси. Кај пристапот за забрана за доделување на ресурси, при секое побарување на ресурсот, се проверува дали неговото доделување носи потенцијален ризик од блокада. Стандардниот алгоритам за проверка на овој тест (*Dijkstra*) е познат како банкарски алгоритам и е продолжување на алгоритамот за детектирање на блокада.

Банкарски алгоритам (Banker's algorithm)

Алгоритамот се моделира според пример на банка каде што банкарот може да издаде готовина за да ги кредитира и задоволи сите барања на клиентите. Алгоритамот проверува дали доделувањето на ресурсот доведува до небезбедна состојба и според тоа го доделува или не ресурсот. Банкарскиот алгоритам може да се искористи доколку се исполнети следниве услови:

- Ресурсите имаат повеќе инстанци;
- Секој процес треба однапред да го објави најголемиот број на инстанци од секој ресурс кој сака да го користи;
- Кога процесот побарува ресурси, системот проценува дали по тоа ќе се најде во стабилна состојба. Ако останува во стабилна состојба, на процесот му се доделува ресурсот, во спротивно процесот мора да чека на ослободување на ресурсите од други процеси.

- Процесот кој добива ресурс треба да ги ослободи во конечно време.

Матриците на достапните и доделените ресурси се исти како во претходниот случај. При извршувањето на банкарскиот алгоритам, кога процесот побарува ресурси првично се смета дека барањето е дозволено, соодветно се врши обновување на системот. Потоа се одредува дали резултантата состојба е безбедна. Ако е безбедна, се врши и вистинското доделување на ресурсите. Во спротивно се блокираат процесите сé до оној момент кога состојбата ресурсите ќе стане безбедна. Алгоритамот може да се изведе преку извршување на следниве чекори:

1. Се бара редица *P* од процесите каде што побарувањата на останатите ресурси е помала или еднаква на векторот *A* (моментно достапни ресурси). Ако не постои таква редица, системот се наоѓа во застој.
2. Ако се претпостави дека избраниот процес, ги бара сите ресурси наеднаш и се извршува. Се маркира овој процес како завршен и неговите ресурси се ослободуваат и се додаваат на векторот на достапни (*A*).
3. Се повторуваат чекорите сè додека сите процеси не бидат маркирани и завршени, со што се покажува дека иницијалната состојба била безбедна, во спротивно не била.

Алгоритамот може да се претстави преку следниов пример (Слика 5.14), ако се претпостави дека постојат 3 вида на ресурси со даден број на нивни инстанци: *P(1) = 9, P(2) = 3, P(3) = 6*, треба да се утврди дали тековната состојба е безбедна, односно дали постои секвенца со која ќе се извршат сите процесите до крај.

Во дадениот пример, се покажува дека ако се извршуваат процесите по редоследот *П2->П1->П3->П4* нема да се појави застој на системот. Процесот *П2* првично барал *(6 1 3)* од сите класи на ресурси, во текот на извршувањето му биле доделени *(6 1 2),* што значи дека му се потребни уште *(0 0 1)* инстанци од ресурсите што е помало или еднакво на векторот на достапните чија тековна вредност е *(0 1 1)*. Процесот се маркира како завршен и неговите ресурси се враќаат на услуга на другите процеси. Истите чекори се повторуваат и со другите процеси.

Банкарскиот алгоритам теориски задоволува и е одличен за избегнување на застои, но практично е неупотреблив затоа што процесите не знаат колку и какви ресурси ќе им бидат потребни при нивното извршување. Покрај тоа и бројот на процесите во системот може да варира, како што се најавуваат и одјавуваат корисниците. Ресурсите можат да станат достапни и да ги снема заради разни причини (расипување или исклучување). Практично многу малку системи го користат Банкарскиот алгоритам за одбегнување на застоите.

	Побарува			Алоцирано			Вкупно		
	P1	P2	P3	P1	P2	P3	P1	P2	P3
П1	3	2	2	1	0	0	9	3	6
П2	6	1	3	6	1	2			
П3	3	1	4	2	1	1	Слободно		
П4	4	2	2	0	0	2	0	1	1

	Побарува			Алоцирано			Вкупно		
	P1	P2	P3	P1	P2	P3	P1	P2	P3
П1	3	2	2	1	0	0	9	3	6
П2	0	0	0	0	0	0			
П3	3	1	4	2	1	1	Слободно		
П4	4	2	2	0	0	2	6	2	3

	Побарува			Алоцирано			Вкупно		
	P1	P2	P3	P1	P2	P3	P1	P2	P3
П1	0	0	0	0	0	0	9	3	6
П2	0	0	0	0	0	0			
П3	3	1	4	2	1	1	Слободно		
П4	4	2	2	0	0	2	7	2	3

	Побарува			Алоцирано			Вкупно		
	P1	P2	P3	P1	P2	P3	P1	P2	P3
П1	3	2	2	1	0	0	9	3	6
П2	0	0	0	0	0	0			
П3	3	1	4	2	1	1	Слободно		
П4	4	2	2	0	0	2	6	2	3

	Побарува			Алоцирано			Вкупно		
	P1	P2	P3	P1	P2	P3	P1	P2	P3
П1	0	0	0	0	0	0	9	3	6
П2	0	0	0	0	0	0			
П3	3	1	4	2	1	1	Слободно		
П4	4	2	2	0	0	2	7	2	3

Слика 5.14: Банкарски алгоритам за одбегнување на блокади

5.4.4. Превенција на застои (Deadlock prevention)

Потребно е да се дизајнира систем кој нарушува еден од четирите услови за појава на застој:

1. Спречи држи-и-чекај (*англ. hold and wait*): Оваа ситуација може да се избегне доколку: секој процес ги побара сите ресурси во почетната точка на извршување и да чека се додека сите ресурси не бидат

достапни. Ако може да се избегне чекањето на дополнителни ресурси на процеси што веќе држат ресурси тогаш нема да има застои. Проблемот е што многу процеси не знаат колку ресурси ќе им требаат до почетокот на извршувањето. И друг начин, кога процес може да бара ресурс само ако не држи други ресурси, што значи дека процесот може да држи само еден ресурс едновремено или прво привремено да ги ослободи ресурсите кои веќе му се доделени. На овој начин се намалува значително користењето на В/И уредите.

2. Спречи взаемно исклучување: Ако се избегне ексклузивно доделување ресурс на некој процес нема да постои можност од застој. Взаемното исклучување е задолжителен услов за сите неделиви ресурси односно ресурси кои може да ги користи само еден процес едновремено. На пример, печатарот е таков ресурс. Принципот на взаемно исклучување не треба да се применува кај деливите ресурси како што се датотеки отворени за читање. Една метода за превенција на застои е да се избегнува условот на меѓусебно исклучување за сите деливи ресурси. Општо за овој начин може да се каже дека треба да се избегнува назначување ресурс освен кога е апсолутно потребно и да се води сметка што помалку процеси да го бараат тој ресурс.

3. Одземање на ресурси: Според условот дека нема испразнување на ресурсите, само процесот кој го држи ресурсот, може и да го ослободи. Оваа е најнезгоден услов за превенција на застои, на пример ако на процес му е доделен на пр. принтер и се наоѓа во среде работа, насилно одземање на принтерот затоа што моментално не е слободен исто така потребниот плотер не с соодветно па може да биде и невозможно. Ова може да се изведе и на следниов начин: секој процес кој бара нов ресурс а не може да го добие да ги ослободи сите ресурси и да помине во состојба на чекање. Процесот може да помине во состојба на подготвен дури кога ќе ги поврати своите ресурси кои ги држел и ресурсите кои ги барал за користење. Процесот не може да се наоѓа во состојба на блокиран со заземени ресурси. Друг начин е кога процесот бара ресурси, а не се достапни, да се провери дали тие се доделени на друг процес кој исто така чека на дополнителни ресурси. Ако се, ресурсите насилно се одземаат од блокираниот процес и му се доделуваат на процесот кој моментно ги побарува.

4. Спречи кружно чекање: Еден начин за елиминација на циклусот е да еден процес може да зафати само еден ресурс едно времено. Ако има потреба од друг да го ослободи првиот. Друг начин е со дефинирање на линеарно подредување на сите ресурси. Процесот кој држи некој ресурс може да бара само ресурси со повисоки броеви. Се подредуваа ресурсите по индекс: *R1, R2,* . . . и се присилуваат процесите да ги бараат ресурсите во дадениот редослед. Случајот каде што *P1* држи *Ri* и бара *Rj* и *P2* го држи *Rj* и бара *Ri* е невозможен, бидејќи $i < j$ и $i > j$.

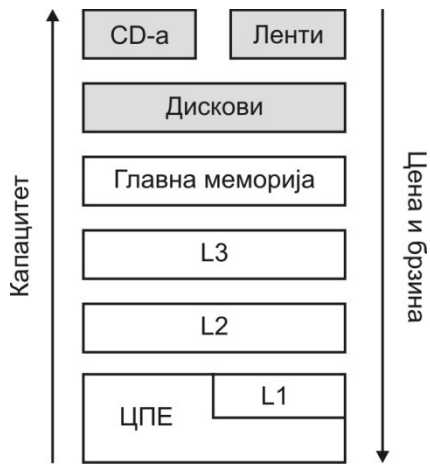
Процес што држи некој ресурс, не може да побара ресурс со понизок реден број, сè додека не го ослободи својот ресурс. Понекогаш е изводливо и постои сценарио во кое сите процеси завршуваат со што застој никогаш нема да се случи, но не е секогаш ефикасен. Не може да се најде редослед по кој ќе се задоволат сите процеси, особено ако постои поголем број на ресурси за различна намена.

6. Управување со меморијата

6.1. Улога на меморијата

Меморијата е важен ресурс во еден компјутерски систем, исто како и процесорот. Процесот може да се извршува само ако е сместен во главната меморија и го користи процесорот. Од најголемо значење е ефикасно и оптимално управување со меморијата, затоа што тоа има големо влијание врз перформансите и работата на самиот компјутерски систем. Иако, модерните системи имаат неколку десетици илјади пати поголем мемориски капацитет, меморијата е ресурс кој секојпат е недоволен. Идеална меморија, гледано од аспект на програмерите би требала да се одликува со бесконечно голем капацитет, бесконечно голема брзина и да биде постојана, (т.е. при губењето на напојувањето не се губи нејзината содржина) и нејзината цена да биде што помала.

Развојот на технологијата не нуди решение за овие барања, така што во постоечките системи, меморијата се организира во мемориска хиерархија, каде што се комбинираат типови на мемории со различни капацитети и брзини и соодветно цени. Самиот оперативен систем ја има таа улога за управување со мемориите. Оперативниот систем треба да овозможи повеќе програми да ја користат меморијата истовремено, да врши заштита на програмите и податоците едни од други и ефикасно да ја користи со цел да се сместат што повеќе процеси на извршување во главната меморија.



Слика 6.1: Хиерархија на мемории

Делот од оперативниот систем кој управува со мемориската хиерархија се нарекува „мемориски управувач“ (*англ. memory manager*) и неговата улога е да следи кои делови од меморијата се користат а кои не, да врши доделување на меморијата кон процесите кога за тоа ќе се појави потреба и одземање на меморијата кога процесите ќе завршат и да управува со размената на делови помеѓу различни видови мемории. Покрај тоа врши и динамичка реалокација на програмите кои се сместуваат во меморијата, делењето на меморискиот простор помеѓу повеќе процеси и заштита на адресниот простор на процесите од меѓусебно влијание и пристап.

При дизајнот на модулот за управување со меморијата, потребно е да се задоволат неколку барања. Едно барање е времето на пристап кон мемориската локација да биде што помало и тоа е зависно од хардверскиот и софтверскиот дизајн. Друго барање е да се максимизира капацитетот на меморијата, односно да се направи транспарентен пристап кон мемориски уреди од различно ниво во мемориската хиерархија. На крај уредот за управување со меморијата треба да биде и ефикасен и да има помал удел во вкупниот процент од цената на не еден компјутерски систем.

Системите за управување со меморијата можат да бидат поделени во две класи: едни кои ги поместуваат процесите помеѓу главната меморија и дискот при извршувањето (размена - *swap* и страничење - *paging*) и оние кои не врши поместување кои се и поедноставни за имплементација. Изборот на методата за управување со меморијата зависи и од хардверската поддршка односно од процесорската архитектура.

6.2. Поврзување на меморискиите адреси

За да се изврши една програма, т.е. да стане процес, потребно е нејзиниот код да биде сместен во главната меморија. Програмата се наоѓа на дискот во форма на бинарна датотека која може да се изврши. Програмата од дискот се вчитува во меморијата во адресниот простор на новосоздадениот процес. Самата меморија е составена од мемориски зборови каде што секоја локација има сопствена мемориска адреса, читањето и запишувањето во меморискиите локации го врши процесорот.

Процесорот со читање на вредноста на регистарот за програмскиот бројач, одредува од која мемориска локација ќе изврши читање на содржината и ќе ја преведе во инструкција или операнд. Еден циклус на извршување на една инструкција се состои од неколку чекори, првиот е преземање инструкција од меморијата (*англ. fetch*), нејзиното декодирање и преземање на операнди од меморијата, ако е потребно. По извршувањето на инструкцијата, резултатите може пак да се зачуваат во меморијата.

Во системите со мултипрограмирање, повеќе процеси ја делат главната меморија. Системот не може да знае кои процеси во одреден момент ќе се

наоѓаат во меморијата ниту пак кои мемориски локации ќе бидат слободни. Како што процесот при неговото извршување, во зависност од методата на доделување на меморијата се користи, повеќе пати се движи на релацијата меморија – диск, така неговото полнење нема да биде на одредена фиксна локација во меморијата. Затоа се користат релативни, симболички адреси кој оперативниот систем, односно модулот за управување со меморијата ги преведува логичките во физички мемориски адреси, т.е. врши нивно поврзување.

Слика 6.2: Преведување на програмата и поврзување

Пред самото извршување, една програма поминува низ повеќе фази каде што мемориските адреси се претставени различно. Програмскиот код се преведува од преведувач – компајлер.

Изворниот код се компајлира во поврзливи објектни модули (*англ. linkable object modules*) каде што мемориските адреси се дадени како релативни поместувања (релокатибилни адреси). Програмата може подоцна да се смести било каде во меморијата, при што библиотеките содржат објектни модули.

Објектните модули заедно се поврзани во извршни модули (*англ. loadable module*) кои што полначот на програми (*англ. loader*) ги поставува во меморијата. Релокатибилните адреси се претвораат во апсолутни и притоа се оставаат неразрешени референците кон динамичките библиотеки. Извршните модули се вчитуваат во меморијата и се извршуваат во процес. Оперативниот систем помогнат од хардверот врши пресликување на логичкиот адресен простор во физичка меморија.

Значи поврзувањето на релативните адреси на инструкциите и податоците кон мемориските адреси може да се случи во три различни фази:

- *Време на компајлирање*: Ако мемориската локација е позната од претходно (*a priori*), може да се генерира апсолутен код и тој треба да се рекомпајлира ако се изврши промена на почетната локација на адресниот простор.
- *Време на вчитување*: Треба да се генерира код кој може да се реалоцира ако стартната мемориска локација не е позната во времето на компајлирањето.
- *Време на извршување*: Адресното поврзување се одложува до извршувањето ако процесот може да биде поместуван во текот на извршувањето од еден мемориски сегмент во друг. Потребна е хардверска поддршка за адресното мапирање (на пр., *base* и *limit* регистри).

6.2.1. Логички и физички адресен простор

При управување со пристапот кон меморииите, одредени податоци може да се наоѓаат на разни мемориски уреди. На пример, може да се најдат во податочните регистри кои имаат мал капацитет и работат со брзина на процесорот, или пак може да се најдат и во главната меморија која пак работи со брзина 1.5 - 4 пати побавно. Самата единица за управувањето со меморииите гледа само поток од мемориски адреси при што не е познато како се генерирани (со користење на програмски бројач, индексирање, апсолутна адреса и т.н.).

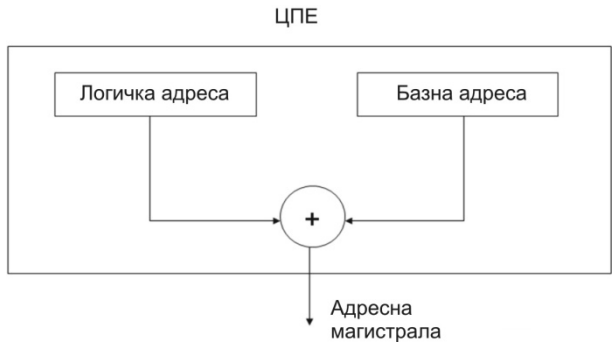
Адресата која се генерира од процесорска инструкција е логичка адреса, а адресата на самата мемориска единица е физичка. И физичките и логичките адреси во фазата на преведување и во фазата на вчитување на програмата се исти, но се разликуваат во фазата на извршувањето на програмата (логичките адреси во фазата на извршување се нарекуваат и како виртуелни адреси). Множество на сите логички адреси кои се генерирани од програмата се нарекува логички или виртуелен адресен простор, а множеството на сите физички адреси кои се соодветни на нив се нарекува физички адресен простор. Секој пристап кон меморијата, односно пресликувањето од виртуелниот кон физичкиот простор се опслужува хардверски со единицата за управување со

меморија (MMU). При процесот на пресликување на мемориските адреси може да се појави проблем на реалокација на програмата. На пример, да се претпостави дека првата инструкција е повик кон процедура со апсолутна физичка адреса 100 во бинарната датотека создадена од поврзувачот (линкерот), ако програмата се вчита така што првата инструкција започнува од нулта мемориска адреса, тогаш повик кон физичката адреса 100 ќе ја даде посакуваната инструкција. Во случај кога првата инструкција од програмата (процесот) ќе биде вчитана на некоја друга локација, повикот кон апсолутна адреса со вредност 100 ќе резултира со пристапување кон локација вон адресниот простор на процесот. Начините за пресликување може да бидат многу сложени, затоа е даден само едноставен пример како се изведува. Физичката адреса се добива како сума од вредноста базната – основна адреса и вредноста на логичката адреса, како што е дадено на примерот.

Пример:

```
CALL 100 се претвора во

        C = A+B = 100 +100 = 200

        A – базна адреса = 100
        B – логичка адреса = 100
        C – физичка адреса = 200
```

Базната адреса е дефинирана со вредноста на така наречен реалокациски регистар и со неа се дефинира адресата на физичкиот почеток на програмата. Секоја логичка адреса која се генерира од програмата се собира со вредноста на овој регистар и на тој начин се добива физичката адреса. Програмата секој пат започнува од нулта логичка адреса и треба само да се води сметка за горната граница на програмата (*0-max*). Логичкиот адресен простор се пресликува во физичкиот простор во опсегот од (*R+0, R+max*), каде што *R* е вредноста на реалокациониот регистар односно физичката адреса на почетокот на програмата.



Слика 6.3: Добивање на физичката адреса

Логичкото адресирање овозможува повеќе програми да можат да работат истовремено во меморијата, да се користи препокривање (*англ. overlaying*) за да се зголеми достапниот адресен простор над физичките ограничувања и да се користат делени библиотеки и динамички библиотеки.

6.2.2. Заштита на меморијата

Заштитата на оперативниот систем од корисничките процеси и меѓусебна заштита на корисничките процеси по прашање на пристапот кон мемориските секции може да се реализира со помош на два регистри:

- реалокационен регистар, кој ја содржи најниската адреса на процесот *(R+0)*
- регистар на ограничување, кој го содржи најголем опсег на логички адреси на процесот

Единицата за управување со меморијата ја пресликува секоја логичка адреса динамички, така што најпрво проверува дали логичката адреса е помала од вредноста на регистарот на ограничувањето, ако е, тогаш се додава вредноста на регистарот за реалокација.

На тој начин се заштитува содржината на меморијата, бидејќи ако се користат апсолутни наместо релативни мемориски адреси, не постои начин да се спречи програма да изврши инструкција која чита или запишува било каков збор во меморијата. Малициозна програма секогаш може да конструира инструкција и ја изврши.

Регистарот за реалокација и регистарот за ограничување се процесорски регистри кои се полнат со вредности тогаш кога процесот ќе го добие процесорот на користење. Од контекстот на процесот, распоредувачот на процесите ги чита вредностите за овие два регистри и ги поставува при активирањето на процесот. Со помош на овие два регистри процесите успешно ги штитат своите мемориски локации, затоа што секоја логичка адреса која е релативна во однос на нулата треба да помине преку обработка од овие два регистри.

6.3. Доделување на меморијата

Главниот проблем при управувањето со меморијата е доделувањето (алокација) на мемориски простор на одреден процес кој треба да се извршува. При проектирање на програмите кои заради нивната големина не може да се сместат целосно во главната меморија, програмерите користат техники со кои привидно може да се зголеми достапниот мемориски простор.

Техниките за доделување на меморијата генерално може да се поделат на два вида:

- *Континуирана алокација* (*англ. contiguous allocation*), каде што и физичкиот и адресниот простор се состојат од континуирана низа на мемориски адреси, при што мемориските партиции кои се доделуваат кон процесите можат да имаат иста или различна големина.
- *Не континуирана алокација* (*англ. discontiguous allocation*), каде што физичкиот адресен простор на процесите не е континуирана низа од мемориски локации. Ваквата алокација ги опфаќа и техниките на страничење, сегментација и страничење со сегментација.

Техниките за доделување на меморијата може да се поделат и според начинот на програмирање и тоа, во случај кога се користи монопрограмирање, привидно зголемување на достапниот мемориски простор се изведува со користење на техниката на препокривање (*англ. overlaying*). А во случај на околина со мултипрограмирање, се користат техники за партиционирање, размена на делови од меморијата (*англ. swapping*) и виртуелна меморија. Дадена е категоризација на техниките според начинот на доделување на меморијата и начинот на програмирање.

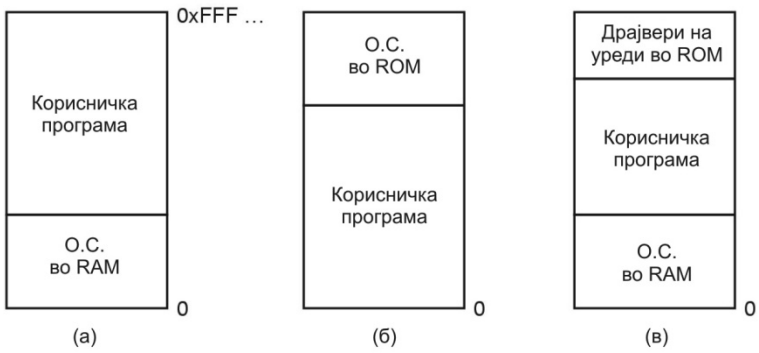
Табела 6.1. Поделба на техниките за доделување на меморијата

Континуирано доделување	**Дисконтинуирана алокација**
Моно-програмирање	Страничење (Paging)
Обично	
Препокривање	Сегметација (Segmentation)
Мултипрограмирање	
MFT (со фиксни партиции)	
MVT (со варијабилни партиции)	

6.3.1. Мемориска организација кај монопрограмирање

Ова е наједноставен начин за управување со меморијата каде што се извршува само една програма која едновремено ја дели главната меморија со оперативниот систем. Ваквите системи се карактеризираат со мало количество на меморија каде што не е имплементирана реалокација и заштита на меморијата. Оперативниот систем може да се наоѓа на дното од RAM меморијата како што е покажано на сликата, или може да биде сместен во ROM на врвот од меморијата, или пак драјверите на уредите може да бидат сместени во ROM-от а оперативниот систем во RAM-от. Првиот модел бил користен на големите „*mainframe*“ системи и миникомпјутери, но веќе многу

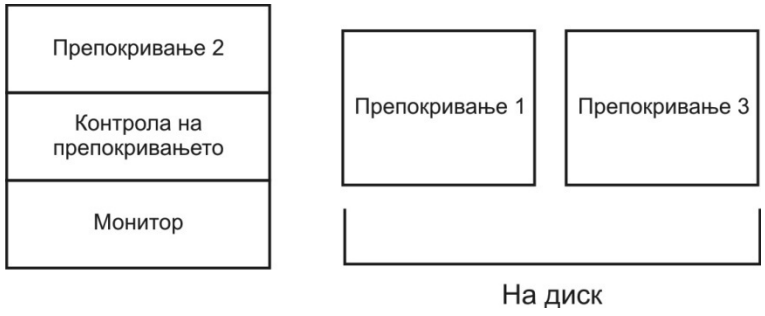
поретко се користи. Вториот модел се користи на некои преносни компјутери (*palmtop*) и вградени системи, а третиот модел бил користен кај поранешните персонални компјутери (пример, MS DOS оперативниот систем) каде што делот во ROM меморијата е наречен BIOS (*Basic Input Output System*). Ако системот е организиран на ваков начин, само еден процес може да биде сместен во меморијата и да се извршува. Штом корисникот ќе внесе команда, оперативниот систем ја внесува програмата во меморијата, ја извршува и по завршувањето, контролата се враќа на корисникот и со внесување на нова команда пак се повторува процедурата од почеток.



Слика 6.4: Организацијата на меморијата кај монопрограмирањето

6.3.2. Препокривање (Overlaying)

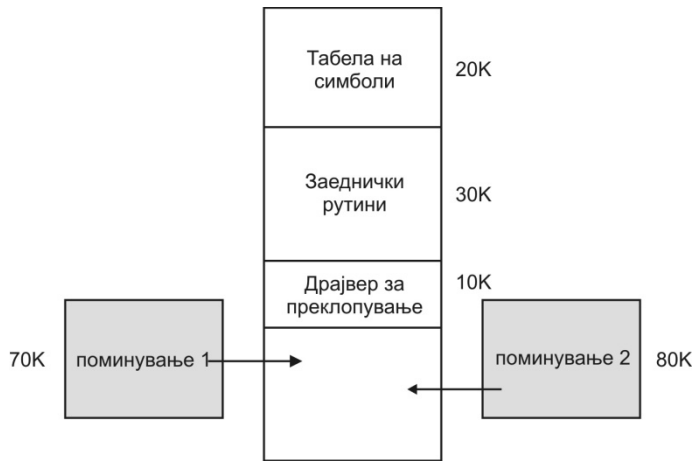
За да се овозможи процесот да биде поголем од количеството меморија што му се доделува се користи техниката на преклопување. Со неа се овозможува во меморијата да се чува само дел од програмата која е потребна во моментот. Кога одреден дел од програмата веќе не е потребен се исфрла од меморијата и се заменува со друг. Проблемот настанува од потребата програмата да се пишува на делови со кои ќе биде овозможено непречено и ефикасно изведување на препокривањето.



Слика 6.5: Изведба на препокривање

Даден е пример за препокривање со програма која се состои од две релативно независни компоненти кои се изведуваат една по друга. Програмерот може самостојно да ја изврши поделбата на изворниот код на два дела. Ако се претпостави дека големината на компонентите е следнава: првиот дел (70 KB), друг дел (80 KB), заеднички податоци (20 KB), заеднички рутини (30 KB).

За извршувањето на програмата се потребни најмалку 200 KB меморија не земајќи ги в предвид и резидентните делови на оперативниот систем. Тоа значи дека програмата не може да се извршува во претпоставениот мемориски простор од 150 KB. Меѓутоа, при извршувањето на првиот дел од програмата, вториот дел и не мора да биде вчитан во меморијата. Исто, така вториот дел од програмата може да се извршува доколку првиот не е во меморијата. Така се дефинираат две преклопувања, каде што првото се состои од заедничките податоци и рутини и кодот од првиот дел и изнесува 120 KB, а вториот дел со заедничките рутини и податоци изнесува 130 KB.



Слика 6.6: Пример за препокривање

Кон програмата е потребно да се додаде и драјвер со кој се управува со преклопувањата (претпоставка дека зазема 10 KB). Така во првото поминување програмата може да се изврши во простор од 130 KB, а во второто во простор од 140 KB што е помало од достапното количество на работната меморија.

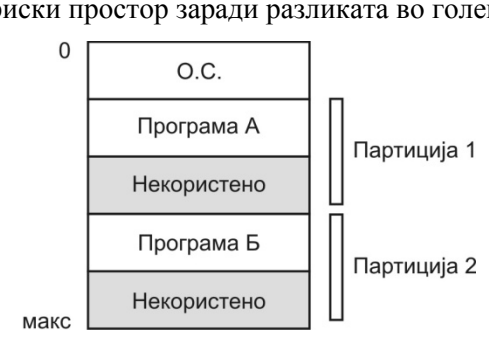
Техниката на препокривање не бара специјална поддршка од оперативниот систем туку деловите комплетно се имплементираат од страна на програмерот со едноставни податочни структури. Проблемот е што програмерот треба внимателно да ја дизајнира структурата за препокривањето, при што е потребно комплетно познавање на структурата на програмот, кодот, податочните структури што може да биде многу сложено за поголеми програми, а баш за нив е и потребно да се врши препокривање.

Техниката го забавува работењето на системот, затоа што деловите се внесуваат во меморијата во повеќе итерации но со неа се постигнува привидно зголемување на работната меморија на системи микрокомпјутери и системи што имаат ограничени ресурси.

6.3.3. Мулти-програмирање со фиксни партиции (MFT)

Освен кај вградените системи монопрограмирањето не се веќе користи. Како што мемориите станувале сè поголеми и сè подостапни станувало се повозможно извршувањето на повеќе програми во исто време, што пак доведува до подобро искористување на системските ресурси (процесорот, меморијата и т.н.)

Извршување на повеќе процеси истовремено значи дека кога еден процес се блокира на чекање на В/И операција за да се изврши, друг процес може да го користи процесорот и на тој начин се подобрува искористувањето на самиот процесор. Наједноставен начин да се оствари мултипрограмирањето е главната меморија да се подели на одреден број на партиции со еднаква или различна големина. Во секоја партиција може да се најде по еден процес кој се извршува. Во оваа организација степенот на мултипрограмирањето е еднаков на бројот на партициите. Кога во системот ќе пристигне процес на извршување може да се постави во редица на чекање на најмалата или еднаква по големина на него партиција. Ако сите партиции се еднакви по големина, разликата во големината на процесот и партицијата ќе биде изгубен мемориски простор. Во случај кога партициите се со различна големина, за секоја партиција ќе има засебна редица на чекање каде што се сместуваат процеси кои може најдобро да се сместат. Проблемот настанува кога редицата на чекање за помала партиција е полна, додека редицата за чекање на поголема партиција е празна. Што значи дека постои количество на меморија за задоволување на потребите но не може да биде искористено. За да се реши овој проблем може да се користи една редица на чекање, каде што секој пат кога една партиција ќе се ослободи од редицата се избира програмата која е на ред, иако можеби ќе се изгуби многу мемориски простор заради разликата во големините.



Слика 6.7: Организација на меморијата со фиксни партиции

Друг начин за пооптимално искористување на просторот е од редицата на процеси да се избере оној кај најдобро се сместува слободната партиција.

Со овој начин се прави дискриминација на малите процеси во однос на користење на големи партиции, дури и каде што е пожелно на помалите процеси да им се даде најдобра а не најлоша услуга. Онаа што е битно да се потенцира во овој случај е дека не може два процеси да бидат сместени во една партиција.

И двата вида на организацијата на редиците на чекање имаат проблем со интерна фрагментација, односно неискористениот мемориски простор во состав на партицијата.

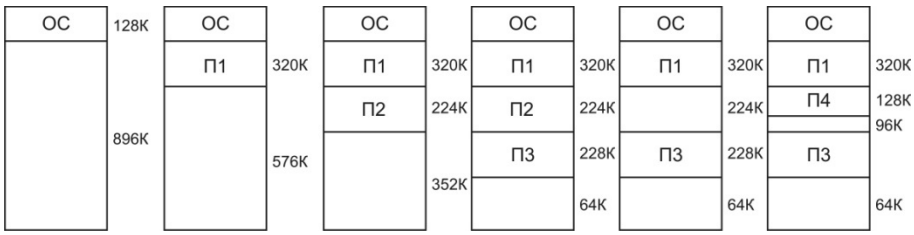
Мултипрограмирањето со фиксни партиции се користело кај системите со групна обработка (*batch*), таа е крајно непогодна за интерактивни системи заради постоење на поголем број на процеси со променлива големина кои се појавуваат по случаен распоред. Заради тоа оваа метода не се користи, но е вредно да се спомне заради дефинирање на нова метода – мултипрограмирање со партиции со променлива големина.

6.3.4. Мултипрограмирање со променливи партиции (MVT)

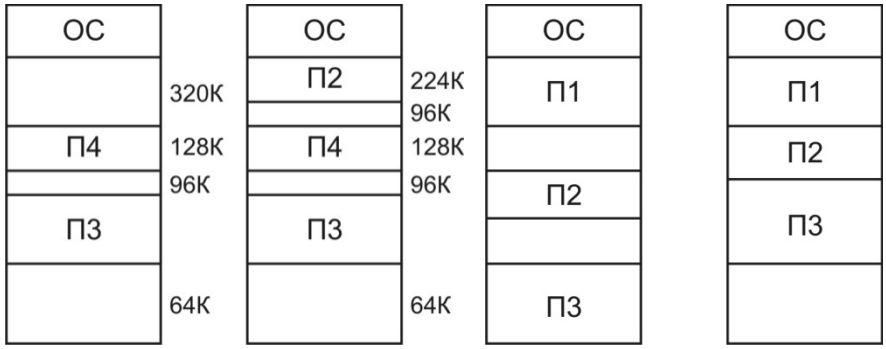
Наместо користење на фиксни партиции меморијата се дели динамички, а секоја дупка (*hole*), односно, слободен континуиран дел од меморијата, може да се искористи за сместување на процеси – под услов да биде доволно голема. На процесите им се доделува меморија по потреба и на почетокот, целата меморија е достапна за процесите како еден голем блок меморија.

Бројот и големината на партициите се променливи со текот на времето и оперативниот систем за управување со слободниот простор користи табела за слободни/зафатени делови (со помош на битмапа или листа). Дупките динамички се создаваат и динамички ги снемува, заедно со процесите, можат да бидат било каде во меморијата и да имаат било која големина, што пак одговара на процесите ви интерактивните системи.

Кога во системот ќе наиде процес, се бара доволно голема мемориска дупка која може да го смести. Ако не се заземе целиот простор, остатокот ќе претставува нова мемориска дупка во која може евентуално да се смести и друг процес. Оваа појава се нарекува како надворешна фрагментација која со тек на време може да создаде голем број на мали дупки каде што не може да се смести ниту еден процес, иако вкупното количество на меморијата е поголемо од потребите. Потребно е да се употреби механизам за збивање со цел процесите да се непрекинати и слободната меморија да обликува посебен блок.



Слика 6.8: Процесот на доделување на меморијата



Слика 6.9: Процесот на збивање на меморијата

На Слика 6.8 е даден пример за користење на динамички партиции, каде што на почетокот целиот мемориски простор е слободен и како што наидуваат процесите се сместуваат, еден до друг. По терминацијата на одреден процес (процес 2), се ослободува меморијата и се појавува дупка, која понатаму може да биде искористена за сместување на друг процес (процес 4). Од примерот може да се заклучи дека по одредено време се појавуваат полем број на дупки кои не можат да ги задоволат барањата за сместување на нови процеси. Затоа е потребно да се врши збивање (Слика 6.9), при што наједноставен начин е сите процеси да се придвижуваат на едниот крај од меморијата а сите дупки да се придвижуваат на другиот крај на меморијата. Прашањето е кога да се врши збивањето, дали во одреден момент оперативниот систем да го изведе тоа или пак да се врши во текот на извршувањето на корисничките процеси, од причина што целата процедура трае и бара поголемо процесорско време. На пример, на систем кој има 256 MB и кој може да копира 4 бајти во 40 nsec, потребно е околу 2.7 секунди за збивање на целата меморија.

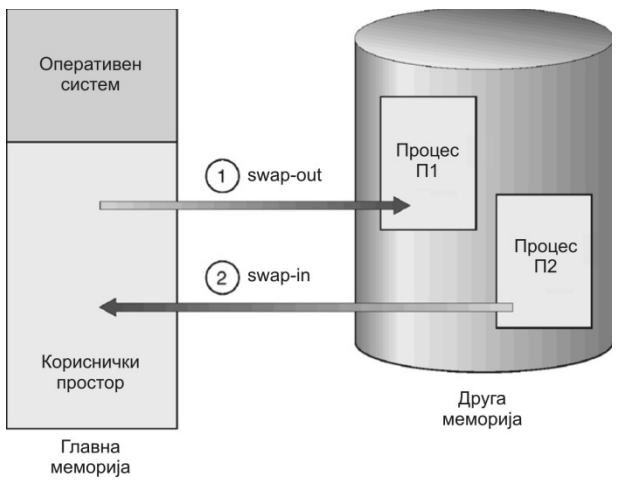
6.3.5. Смена (Swapping)

За систем организиран со фиксни мемориски партиции организирањето на меморијата е едноставно и ефикасно. Секоја програма се вчитува во партиција и таму останува се додека не се заврши. Кај системите со временска поделба и

интерактивните системи со графички кориснички интерфејс, може да се појави недостаток за меморија за да се опслужат сите активни процеси, така што некои од нив е потребно да се чуваат на дискот и да се внесуваат во главната меморија динамички кога за тоа ќе се појави потреба. Постојат две решенија за управување со меморијата кои можат да бидат користени во зависност од хардверските карактеристики на системот. Наједноставниот начин е наречен смена (*swapping*) и функционира на тој начин што ги вчитува процесите во целост од дискот во главната меморија, ги извршува и потоа пак ги враќа на дискот. Другиот начин е со користење на виртуелна меморија, со која се овозможува извршување на процесите дури и ако се делумно во главната меморија. Овој метод е објаснет во посебно поглавје од книгата.

Смената на процесите се изведува преку две фази, како што е прикажано на сликата:

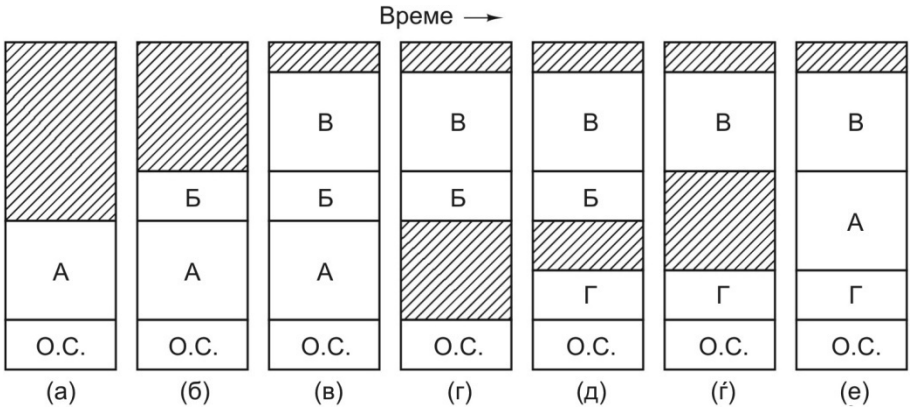
- Swap-out: се врши суспендирање на процесот и се врши копирање на неговата меморија од главната меморија на дискот
- Swap-in: се врши копирање на меморија на процесот од дискот на главната меморија и продолжува со неговото извршување



Слика 6.10: Измена на процесите

Операциите за измена се опишани и на следниов пример, иницијално во меморијата е само еден процес. Потоа процесите *Б* и *В* се создаваат или се внесуваат од дискот. Процесот *А* се отстранува од меморијата на дискот, а процесот *Г* наидува и процесот *Б* се отстранува од меморијата. Процесот *А* пак се внесува во главната меморија меѓутоа на различна локација од претходниот пат. Затоа е потребно да се примени реалокација на адресите било од софтверот при измената или од хардверот при извршувањето на програмата.

При извршување на процесите, нивните податочни сегменти може да нараснат преку динамичка алокација на меморијата. Ова во системите со фиксно партиционирање претставува проблем, затоа што може да не постои доволен простор за зголемување на меморијата. Во системите со динамичко партиционирање и таму каде што се користи измена на процесите, ако до процесот кој е сместен постои дупка таа може да се искористи и да се додели на процесот за зголемување на адресниот простор. Од друга страна ако процесот се наоѓа до друг процес и зголемувањето не е можно, потребно е тој да се премести во поголем слободен мемориски простор или еден или повеќе процеси треба да се исфрлат на дискот за да се ослободи доволен простор. Во најлошиот случај ако процесот не може да расне во главната меморија и нема слободен простор на дискот, процесот ќе мора да чека или да биде терминиран.

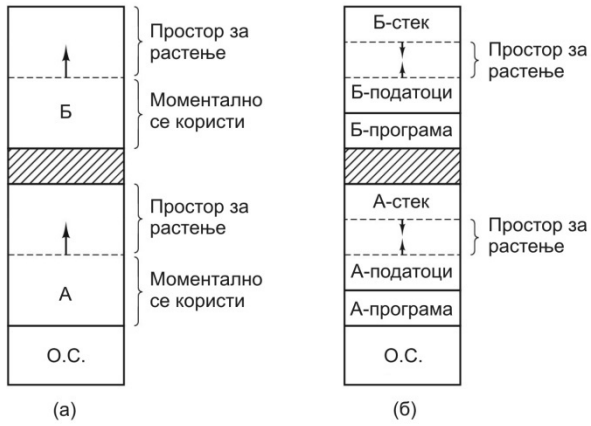


Слика 6.11: Пример за смена на процесите

Се очекува повеќето процеси да раснат при нивното извршување и затоа е потребно да се алоцира повеќе меморија секој пат кога процесот ќе се вчита или отстрани на дискот.

На Слика 6.12 е даден пример за доделување на поголеми партиции со цел да се овозможи задоволување на потребите за меморија од страна на процесите. Едната варијанта е да се овозможи зголемување во еден правец до исполнување на слободниот простор, а другиот случај е нараснување на стекот и податочниот сегмент еден кон друг. Меморијата помеѓу нив може да се искористи од било кој сегмент, а ако се случи да се исполни просторот, процесот треба да се премести или да се отстрани од меморијата на дискот.

Измената на процесите (*angl. swapping*) овозможува поголем степен на мултипрограмирањето како и подобро искористување на меморијата а како ефект се добива и помало трошење на процесорот за процесот за збивање.



Слика 6.12: Растење на процесите во главната меморија

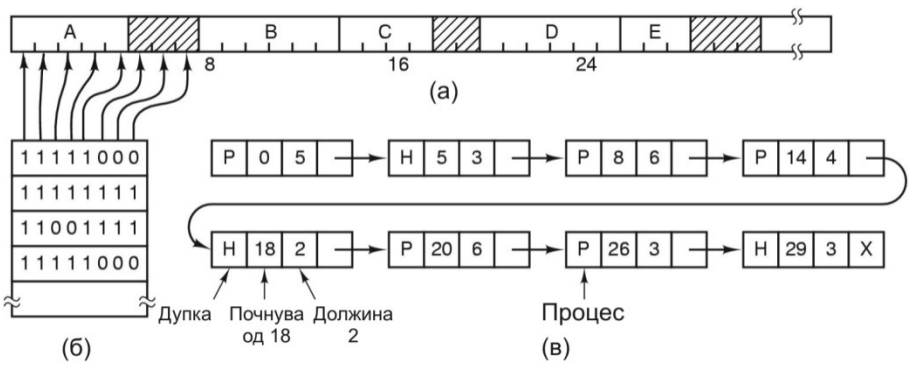
Главните недостатоци на овој пристап се тоа што виртуелниот простор е многу помал од реалниот адресен простор, а целата програма треба да биде сместена во меморијата за да може да се извршува и тоа во континуиран мемориски простор. Од овие причини се врши пресликување на континуираниот виртуелен адресен простор во дисконтинуирани парчиња од физичката меморија. Со тоа се решава и проблемот за потребата од збивање. Општо проблемите со користење на динамички партиции и измената на процесите може да се решат со користење на концептот на виртуелната меморија.

6.3.6. Управување со меморијата со бит-мапи

Во случај кога меморијата се доделува динамички, потребно е да се управува од страна на оперативниот систем и тоа на два начини: со користење на бит-мапи или поврзани листи. Со помош на бит-мапа, меморијата се дели на единици за доделување во опсег од неколку зборови па до неколку килобајти. На секоја алокациска единица во бит-мапата одговара еден бит, кој може да добие вредност 0 ако е слободна или 1 ако е зафатена (или обратно).

На Слика 6.13 б) е даден пример на опис на зафатеноста на меморијата со користење на бит-мапи. Големината на алокациската единица е битен фактор при дизајн на ваков систем за следење на меморијата. Ако единицата е помала, доаѓа до зголемување на бит-мапата. Дури и со големина на единицата од 4 бајти, каде што за 32 – бита од меморијата се користи еден бит, бит мапата ќе користи 1/33 од меморијата. Ако од друга страна се земе единицата да биде поголема, бит-мапата ќе биде помала, но се губи поголем дел од меморијата при алокација ако големината на процесот е помала или не е цел број пати од големината на единицата за алокација. Бит-мапата е едноставен начин за следење на мемориските локации со фиксна меморија, затоа што големината

на мапата зависи од бројот на локациите и големината на алокациската единица.

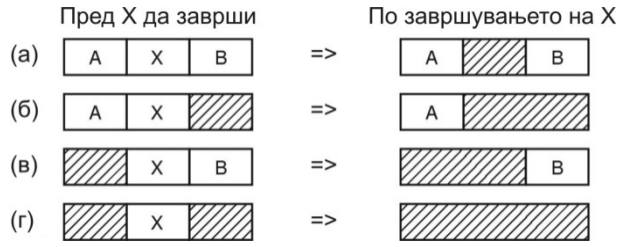


Слика 6.13: Управување со меморијата со б) бит-мапи и в) поврзани листи

Проблемот се состои во тоа што кога се врши вчитување на процес со k единици во меморијата, управувачот со меморијата треба да ја пребара бит-мапата за да најде последователна низа од k нули. Тоа претставува бавна операција и е главен аргумент против користењето на бит-мапите за управување со меморијата.

6.3.7. Управување со меморијата со поврзани листи

Друг начин за следење на зафатеноста на меморијата е да се користи поврзана листа на алоцирани и слободни мемориски сегменти, каде што сегментот е процес или дупка помеѓу два процеси.



Слика 6.14: Спојување на соседни членови во листата

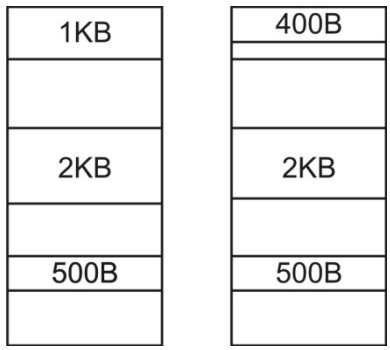
Меморијата од Слика 6.13 в) е претставена како поврзана листа од сегменти, каде што секој влез во листата преставува процес или дупка, адресата на која започнува и со која должина е, како и покажувач кон следниот член во листата. Во дадениот пример, листата на сегментите е сортирана и со тоа се добива предност во случаите кога еден процес се терминира или се отстранува

од меморијата на дискот, односно обновувањето на листат е едноставно. Кога еден процес се терминира, обично има два соседи, процеси или дупки, на Слика 6.14 се претставени сите комбинации.

6.4. Алгоритами за доделување на меморија

Кога еден процес влегува во системот, се сместува во редица на чекање на доделување на мемориски простор во главната меморија а со тоа и извршување во процесорот. Оперативниот систем води сметка за меморискиte потреби на секој процес и количеството слободна меморија, тој одлучува за доделувањето каде што процесот се сместува во меморијата и се натпреварува за доделување на процесорот. При завршување се ослободува доделената меморија и оперативниот систем ја доделува ослободената меморија на друг процес. Алокацијата на простор во меморијата на нов процес во многу зависи од тековната слика на доделувањето на меморијата, односно распоредот на пополнувањата и празнините. По правило постојат повеќе празнини кои би биле кандидати за сместување на процесот а со тоа и соодветни алгоритами за сместување.

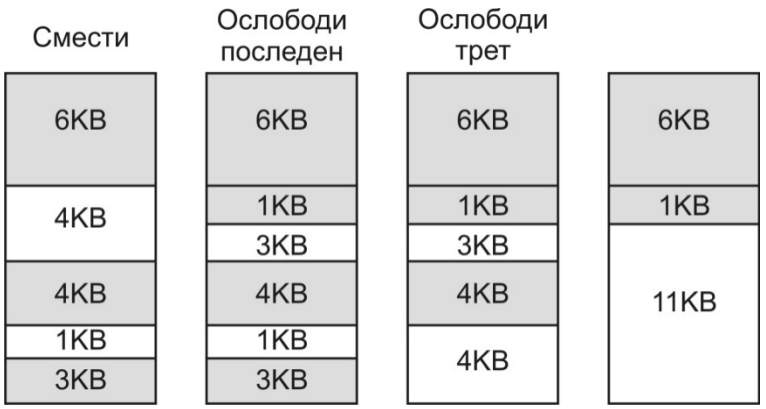
Наједноставниот алгоритам е „прв избор“ (*angl. first fit*). Управувачот со меморијата врши пребарување на листата на сегментите се додека не се пронајде празнина која е доволно голема за дадениот процес. Потоа, празнината се дели на два дела, од кои едниот го содржи процесот, а другиот дел остатокот од меморијата која останува непополнета, освен во случајот кога постои потполно совпаѓање на големината на процесот и празнината.



Слика 6.15: Пример за сместување *n* бајти, се користи првиот расположив слободен блок поголем од *n*, во случајот за сместување на 400 бајти, го користиме првиот расположив блок

Овој алгоритам се одликува со едноставност на имплементација и брзо пребарување, но потребна е листа на слободни блокови сортирани по адресите. Сместувањето бара пронаоѓање на соодветна партиција, а ослободувањето бара проверка да се види дали ослободената партиција може да биде споена со соседната слободна партиција (ако постои). На примерот (Слика 6.15) е

прикажана секвенца на сместување на процес со големина од 100 KB, во главната меморија со одредени празнини. Алгоритамот го сместува процесот во првата празнина од 400 KB иако не е целосно искористена и со тоа се создава помала празнина која понатаму ќе биде потешко да се пополни со поголем процес. При ослободувањето, две празнини се спојуваат во една поголема, што доведува со тек на времето до тенденција на појава на слободни блокови кон крајот на адресниот простор. Друг недостаток е и појава на надворешна фрагментација.



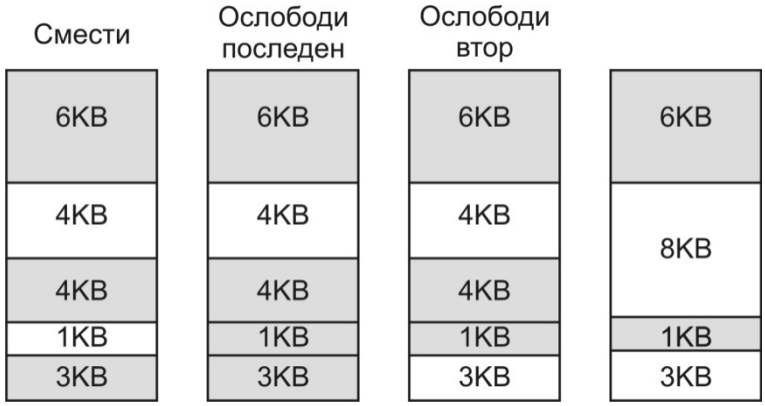
Слика 6.16: Пример за збивање кај *first fit*

Слика 6.17: Пример за сместување на *n* бајти, се користи најмалиот расположив слободен блок поголем од *n*, во случајов за сместување на 400 бајти, го користиме третиот блок (најмалиот)

Варијанта на алгоритамот за прв избор е „следен избор“ (*англ. next fit*). Тој функционира на ист начин како и прв избор освен што води сметка каде се наоѓа при изнаоѓање на соодветна празнина. Следен пат кога ќе се повика алгоритамот за наоѓање на празнина, пребарувањето започнува од последната

позиција наместо од првата. Се покажало дека овој алгоритам дава малку послаби перформанси во однос на прв избор.

Друг познат алгоритам е „најдобар избор“ (*англ. best fit*), тој врши пребарување на целата листа и ја одбира најмалата празнина која е доволна за сместување на процесот. Наместо да се дели поголема празнина која може да биде потребна подоцна, се избира празнина која е најблиска до потребната големина. Најдобриот избор е побавен од првиот избор, затоа што треба да ја пребара комплетната листа секој пак кога ќе биде повикан. Изненадувачки, резултира и со поголем обем на изгубена меморија од претходно наведените, затоа што је полни меморијата со мали бескорисни празнини во кои не може да се смести ниту еден процес. Идејата за користење на овој алгоритам е да се одбегне фрагментацијата на големите слободни блокови и да се минимизира надворешната фрагментација. Меѓутоа, за имплементација на алгоритамот е потребна листа на слободни блокови сортирани по големина, при сместувањето се бара избор на соодветна партиција, а со ослободувањето се врши проверка да се види дали ослободената партиција може да се спои со соседната слободна партиција (ако постои таква).

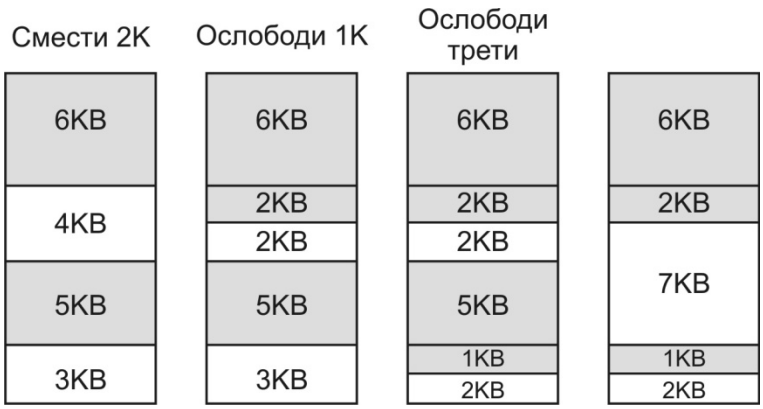


Слика 6.18: Ослободување кај алгоритамот *best fit*

Сè на сè овој алгоритам добро работи под услов кога најголемиот дел од сместувањата се со мала големина и неговата имплементација е релативно едноставна. Од друга страна и тука се појавува надворешна фрагментација, ослободувањето е бавно и постои тенденција на создавање на многу бескорисни мали фрагменти. За да се надмине поделбата на празнината на процеси и нова мала празнина, може да се користи алгоритамот наречен „најлош избор“ (*англ. worst fit*), кој за разлика од претходно наведените пак ја избира најголемата празнина, со цел по тоа пак да создаде доволно голема празнина (Слика 6.19).

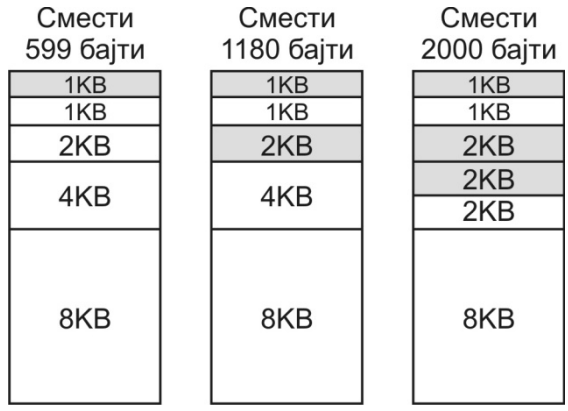
Слика 6.19: Пример за сместување на *n* бајти, се користи најголемиот расположив слободен блок поголем од *n*, во случајов за сместување на 400 бајти, се користи средниот блок (најголемиот)

Алгоритамот се карактеризира со брзо сместување и најдобро работи кога сместувањата се со средна големина. Иако изразена е надворешната фрагментација, каде што се фрагментираат големите слободни блокови и непотребно се трошат големите партиции.



Слика 6.20: Ослободување на меморијата кај најлош избор

Симулациите покажуваат дека прв избор и најдобар избор се подобри по перформансите и по искористувањето на меморијата од методот на најлош избор. Што се однесува до првите две методи, првиот избор е најбрз а кој од двата е подобар е дискутабилно. Додека сето тоа пак зависи од самата распределба на процесите во однос на времето. Постои и друг алгоритам за сместување со динамички партиции кој е поврзан и со управувањето со меморискиот простор, кој се нарекува систем на здружени парови или „пријателско сместување“ (*англ. Buddy Allocation)*. За сместување на n бајти, се врши делење на слободниот простор во два блока (*англ. buddies*). Потоа рекурзивно се дели првиот блок на истиот начин се додека најмалиот блок има големина која е поголема од *n*. На пример, за сместување на 599 бајти во блок од 16K опишан е процесот на Слика 6.21.



Слика 6.21: Пример за сместување на 599 бајти

За управување со меморијата се користи по една листа за секој блок слободна меморија со големина од 2^n бајти (1, 2, 4, 8, 16, …). На пример, ако имаме меморија со големина од 1 MB, тогаш ќе постои 21 листа (2^0=1, 2^2=2, ... 2^{20}= 1 MB). На почетокот, целата меморија е празна и во листата на празнини со големина од 1 MB имаме еден запис, додека останатите листи се празни. Значи за мемориски простор со големина *n,* потребно е пребарување низ *log n* листа и алгоритамот се карактеризира со многу брзо сместување. При ослободувањето на еден блок, можно е спојување на соседните блокови со иста големина и рекурзивно се врши спојувањето се додека е можно.

Недостаток на овој алгоритам е сместување на процес со големина кој е малку поголем од половина од големината на тековната партиција. Како што е прикажано на сликата, за сместување на 599 бајти се троши цела партиција од 1 килобајт. Исто важи и за сместување на 1180 бајти во партиција со големина од 2 килобајти. Ако во даденив пример се направи анализа колку мемориски простор зафаќаат процесите, 599+1180+2000=3779 бајти и колку е потрошена меморијата за нивно сместување, 1000+2000+2000+5000, се доаѓа до заклучок дека се појавува проблем со внатрешна фрагментација (меморија што е доделена во партицијата, но не е искористена) исто како проблем со надворешна фрагментација. Оттука следува дека овој алгоритам е најдобар во случајот кога процесите имаат големина која е за малку помала од некој блок – степен на два отколку кога се за малку поголеми.

6.5. Внатрешна и надворешна фрагментација

При доделувањето на меморијата на процесите како што веќе беше споменато се појавуваат два вида на фрагментација на меморијата:

- Внатрешна (интерна) фрагментација, која се појавува доколку на процесот се додели мемориска партиција поголема од меморијата која барана од процесот и останатиот дел е неупотреблив.
- Надворешна (екстерна) фрагментација, која се појавува кога меморискиот простор не е континуиран па понекогаш не може да се задоволи барањето за сместување на процес иако има доволно слободна меморија.

Намалување на надворешната фрагментација може да се оствари со збивање на меморијата (*англ. compaction*), каде целта е да се изведе преместување на содржината на меморијата за да се добијат поголеми празнини односно, континуиран мемориски простор.

Збивањето е можно само ако реалокацијата на програмите кои се извршуваат е динамичка и се изведува во времето на извршување, но не и ако се врши при преведувањето или полнењето на програмата. Збивањето ги деградира перформансите на системот, затоа што системот треба да ги прекинува процесите и привремено да ги префрла на дискот со што се губи време. Во зависност од вкупната меморија и просечната големина на процесите, надворешната фрагментација може да биде голем или мал проблем, кој може да се реши и преку примена на техники за дисконтинуирано доделување на меморија.

7. Виртуелна меморија и страници

Проблемот за доделување на мемориски простор од главната меморија кој се појавува во случајот кога програмите се поголеми од вкупното количество на примарната меморија се решавал со тоа што се вршела поделба на програмата во делови кои се преклопувале. Овој пристап иако го решавал проблемот имал одредени недостатоци кои се спомнати во претходното поглавје, иако за измената на деловите се грижел самиот оперативен систем, задачата на програмерот била да изврши соодветна поделба на програмскиот код. Процесот е комплексен и бара многу време и затоа се поминало на автоматска поделба на меморискиот простор кој би го зафатила програмата.

Од друга страна дури и да постои доволен вкупен простор за сместување на процесот во некои од партициите, остануваат проблемите со фрагментацијата и потребата од збивање на меморијата, динамичкиот раст партициите како и не постоење на победничка политика на сместување/ослободување. Бидејќи процесите имаат потреба од континуиран мемориски блок што тешко се обезбедува, како решение на проблемот е да се даде илузија на континуиран мемориски простор иако во реалноста, сместувањето не би било континуирано.

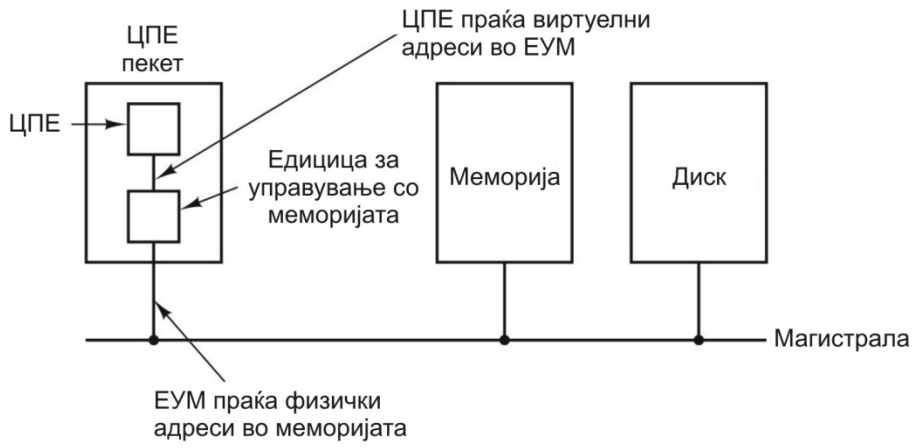
Концептот е наречен „виртуелна меморија“ и основната идеја се состои во тоа што вкупната големина на програмата, податоците и стекот може да ја надмине вкупната достапна физичка меморија. Оперативниот систем ги сместува деловите од виртуелниот мемориски простор и во главната меморија и на секундарната меморија – дискот. Виртуелната меморија може да се користи и во околина со мултипрограмирање, каде што во главната меморија се чуваат делови од различни програми, а кога ќе се појави потреба од дискот да се преземе друг дел, додека процесот чека на В/И операција, процесот може да биде прераспореден и процесорот да му се додели на друг процес.

7.1. Страници

За да се оствари илузија на доволно голема и континуирана виртуелна меморија оперативниот систем користи мемориски страници, односно техника наречена страничење *(англ. paging)*. Процесната меморија (виртуелниот и физичкиот адресен простор) е поделена на делови со фиксна и еднаква големина наречени страници (*англ. pages)* кои се пресликуваат во рамки (*англ. frames)* во главната меморија.

Адресите кои се програмски генерирани се нарекуваат виртуелни мемориски адреси кои го формираат виртуелниот адресен простор. Во системите со виртуелна меморија, при референцирањето на виртуелна адреса, таа не се поставува директно на адресната магистрала туку единицата за управување со

меморијата (*MMU – Memory Management Unit*) врши пресликување на виртуелната адреса во физичка мемориска адреса.



Слика 7.1: Виртуелен и физички адресен простор

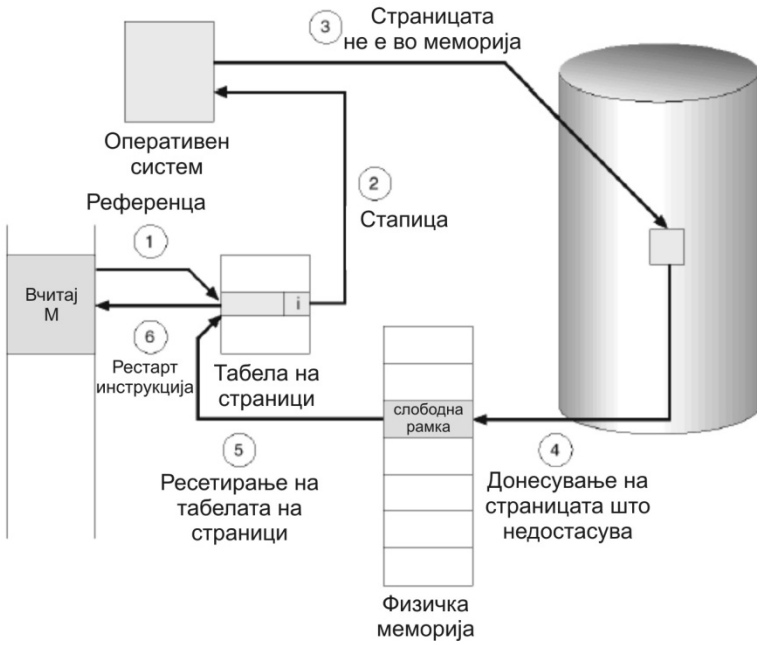
Процесите повеќе не ги "гледаат" вистинските физички адреси и процесните страни можат да бидат сместени насекаде низ главната меморија, а со тоа се отстранува и надворешна фрагментација. Виртуелната страница нема сопствена меморија (таа е само концепт) и потребно е таа да биде сместена во физичка рамка (реална страница) што обезбедува реална меморија со што се остварува илузијата дека континуиран виртуелен мемориски простор не е и физички континуиран и физичкиот адресен простор е независен од виртуелниот.

На Слика 7.2 е даден пример за пресликување каде што е претпоставен компјутер кој може да генерира 16-битни адреси, од 0 до 64 K. Системот поседува физичка меморија од само 32 KB, иако може да се пишуваат програми со големина до 64K тие не можат целосно да бидат сместени во меморијата, туку се сместуваат на дискот. Виртуелниот адресен простор се дели на страници, а соодветните единици на физичката меморија се наречени рамки на страниците (*англ. page frames*). И двете единици имаат иста големина која изнесува степен на 2 (може да се движи од 512 бајти па до 64 KB), во примерот е дадено дека се 4 KB. Во претставениот систем постојат 16 виртуелни страници и 8 физички рамки. Преносот од дискот во меморијата се врши во единици со големина на страницата.

Слика 7.2: Пример за пресликување на виртуелниот во физички адресен простор

На овој начин не се решава проблемот со големината на виртуелната меморија во однос на физичката. Бидејќи постојат само 8 физички страници, може во нив да се сместат само 8 виртуелни страници, оние кои се прикажани на сликата со ознаката „X“ не се пресликани.

Ако се случи да биде повикана мемориска адреса која припаѓа во опсег на страница која не е пресликана во физичката меморија, MMU го препознава настанот и оперативниот систем прави софтверски прекин на извршување на процесорот (*trap*), за да се доведе потребната страница во меморијата. Овој прекин се нарекува „испад на страница“ (*англ. page fault*), оперативниот систем тогаш ја зема страницата од меморијата која е најмалку користена и ја запишува на дискот, а потоа оттаму ја презема реферираната страница и ја сместува во слободната рамка, го изменува пресликувањето и повторно ја рестартира прекинатата инструкција, таа која извршила референцирање на виртуелната адреса (Слика 7.3).



Слика 7.3: Опслужување на Page Fault настан (Page Fault Handler)

7.2. Табела на страници

Табелата на страниците изведува пресликување на процесните страници во рамки како што е опишано. Секоја адреса генерирана од процесорот (виртуелна адреса) се дели на 2 дела, број на страница (*англ. page number*) – *p*, кој го опишува бројот на виртуелната страна (повисоките битови) и поместување на страница (*англ. page offset*) – *d*, односно пониските битови. На пример, со 16-битна адреса и големина на страница од 4 KB, повисоките 4 бита специфицираат една од 16-те виртуелни страници, а 12-те пониски бита го одредуваат поместувањето во селектираната страница (0-4095).

Виртуелниот број на страницата се користи како индекс во табела на страници за да се најде влезот за таа виртуелна страница. Од вредноста на влезот во табелата, се наоѓа бројот на физичката рамка, ако воопшто постои, односно е извршено доделување на рамката. Потоа се врши додавање на бројот на рамката однапред кон вредноста на поместувањето со што се добива физичката адреса која се референцира од меморијата.

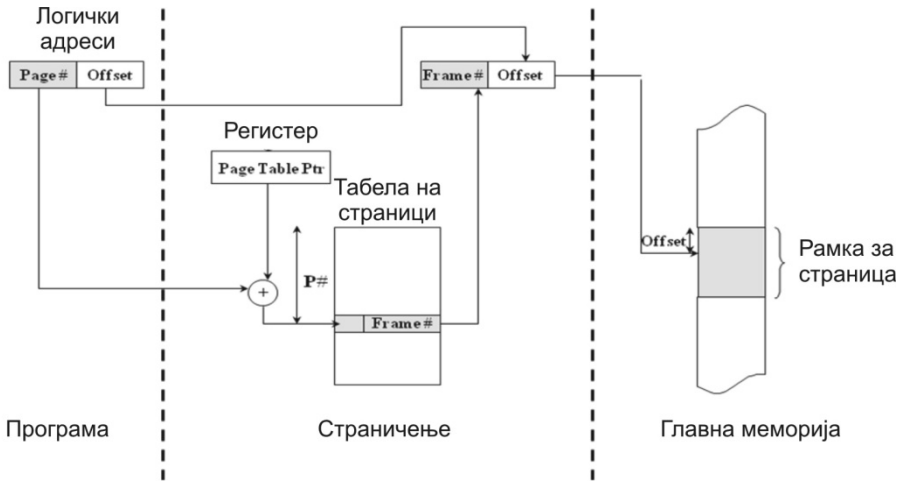
Пример.

Големина на страница 8 KB = 8 · 1024=8192

MOVE REG, 104852 (104852 = 12·8192 + 6548)

Се проверува 13-тата редица во табелата на страници
Ако v = 0 (виртуелната страница не е во рамка) page fault
Ако v = 1 се чита бројот на рамката во физичката меморија за 13-тата виртуелна страница - нека 7
Содржината на регистерот се праќа во мемориска локација

7 ·8192 + 6548 = 63892 преку магистралата



Слика 7.4: Користење на табела на страници

На адресната магистрала се добива само барање за запис во локацијата 63892, без да се знае за било каква претходна активност на единицата за управување со меморијата. Намената на табелата на страници е да се изврши пресликување на виртуелните страници во физички рамки *Forward-mapped page tables (FMPT).*При тоа се можни два проблеми, едниот е во врска со големината на табелата на страници а другиот е поврзан со брзината на пресликувањето.

Денешните компјутери користат виртуелни адреси со најмалку 32-бита (од поново време и од 64 бита). Ако се претпостави големина на страница од 4 KB, тогаш 32-битен адресен простор има 1 милион страници, а 64-битен адресен простор има број на страници кој многукратно ги надминува ресурсите за сместување на една табела на страници. За секоја страница е потребен одреден мемориски простор за сместување на влезот во табелата на страници, а дури и секој процес поседува сопствена табела на страници во сопствениот виртуелен адресен простор.

Вториот проблем произлегува од фактот дека при секое референцирање на меморија е потребно да се направи пресликување. Зависно од типот на инструкцијата бројот на пресликувањата може да биде 2, 3 или повеќе пати, а ако на пример инструкцијата трае околу 4 наносекунди, пресликувањето треба да се изведе под вредноста од 1 наносекунда за да системот работи ефикасно.

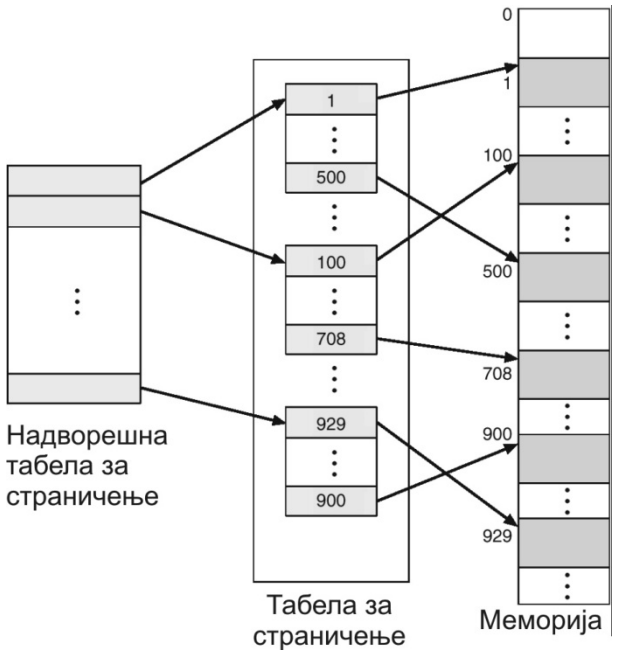
Наједноставен начин за имплементација на страничењето е да постои единствена табела на страници која се состои од брзи хардверски регистри со влез за секоја виртуелна страница индексирана со бројот на виртуелната страница (Слика 7.4). Кога ќе се стартува процес, оперативниот систем ги вчитува регистрите со табелата на страниците од копијата која се чува во главната меморија. При извршувањето на процесот, нема потреба од референцирање на меморијата за табелата на страници при пресликувањето, затоа што сите потребни вредности се наоѓаат во брзите хардверски регистри. Овој пристап е едноставен но од друга страна може да биде потенцијално скап заради големината на табелата и има ефект врз перформансите заради вчитувањето на табелата при секоја измена на контекст на процесот.

Друга начин е да се чува табелата на страниците целосно во главната меморија и доволен е еден регистар кој ќе ја чува вредноста на почетокот на табелата на страници. Со тоа се овозможува измена на мемориската мапа при измена на контекстот со едноставно вчитување на еден регистар. Недостатокот е што е потребно да се рефренцира меморијата повеќе пати за читање на влезовите од табелата на страници при изведување на една инструкција. За да се надминат овие проблеми, се користат варијанти на последниот пристап со кои се овозможуваат многу подобри перформанси, како и повеќе нивоа *FMPT* и инвертирани табела на страници (*Inverted page tables - IPT)*.

7.2.1. Табели на страници со повеќе нивоа

За да се надмине проблемот со чувањето на целата табела на страници во меморијата, се прибегнува кон хиерархиска организација, при што во главната меморија се чува само едно ниво од хиерархијата додека останатите се сместени на дискот. Овој начин користи една табела на страници која покажува на страници во кои се сместени вистинските табели на страници (Слика 7.5). Се користат битови од виртуелната адреса за индексирање на хиерархијата на табелите на страници, на пример ако имаме 12 бита за поместувањето и 20 бита за индекс на страницата и ако се користат две нивоа за страничење како што е прикажано на сликата, тогаш овие 20 бита може да се поделат на битови кои покажуваат на надворешни и внатрешни табели во произволен однос (на пример 10 со 10). Тогаш логичката адреса ќе ја има следната интерпретација: 10 највисоки битови претставуваа индекс со кој од надворешната табела се избира една од внатрешните табели со вистинските табели со страници. Следните 10 битови од адресата покажуваат на позиција

од одберената табела од која се чита покажувачот на соодветната рамка за страници од физичката меморија.



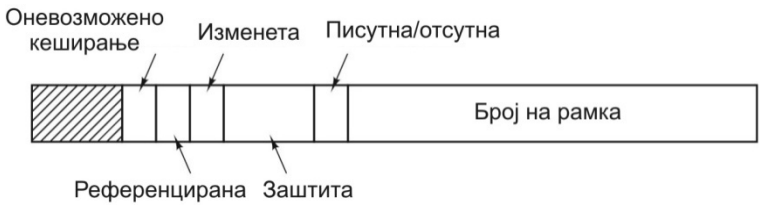
Слика 7.5: Табела на страници со повеќе нивоа

Треба да се забележи дека иако адресниот простор содржи милиони страници, всушност потребни се само четири табели на страници во меморијата и тоа: страницата од највисокото ниво и трите страници второто ниво. Хиерархискиот систем на табелата на страници може да се прошири и на 3, 4 или повеќе нивоа на табелата со што се добива поголема флексибилност но и поголема сложеност.

7.2.2. Структура на табелата на страниците

На Слика 7.6 е даден приказ на записот во табелата на страници, вистинскиот изглед во многу зависи од хардверот иако информациите кои се претставени се заеднички за сите изведби. Големината на влезот варира од систем до систем, но 32 бита е вообичаена вредност. Најзначајното поле е бројот на рамката на страницата (*англ. page frame number*), со кое се врши пресликувањето. Потоа постои бит кој е индикатор за присутноста на страницата (*present/absent bit*), ако тој е 1 влезот во табелата е валиден и може да се користи, ако е 0 виртуелната страница на која и припаѓа овој бит моментно не се наоѓа во меморијата. Референца кон невалидна страница генерира настан за грешка (*page fault*) кој се обработува од страна на оперативниот систем.

Битот за заштита (*Protection bit*) покажува кој вид на пристап кон страницата е дозволен, ако полето содржи 1 бит тогаш со вредност 0 се означува дозволено пишување/читање а со 1 само дозволено читање. Посложен начин е да се користат 3 бита, секој за овозможување на читање, пишување и извршување на страницата, соодветно.



Слика 7.6: Битовите во PTE (Page Table Entry)

Битовите за измена и референцирање *(modified, referenced)* го следат користењето на страницата. Кога се врши запишување во даден страница, автоматски битот за измена се поставува на 1. Со тоа се означува дека страницата е „валкана“ (*dirty bit*) и треба при нејзината измена да се запише на дискот, ако страницата е неизменета, односно останала „чиста“ тогаш едноставно се отфрла затоа што копијата на дискот е се уште валидна.

Битот за референцирање покажува дали страницата е пристапувана во скоро време било за читање или запишување. Се користи од страна на алгоритамите за замена на страниците при појава на „*page fault*“ настан. Страниците кои не се користени се подобри кандидати за замена од оние кои се користени скоро време.

7.2.3. Делење исти рамки

Една од предностите на страничењето е можноста за делење на заеднички код помеѓу повеќе процеси, што е многу погодно за интерактивни системи. Нека се земе за пример систем на кој повеќе корисници истовремено извршува уредувач на текст. Кога не би било можно делењето на страниците, секој кориснички процес покрај своите податоци би требало да вчита во меморијата и засебна копија од кодот на уредувачот на текст. Ако кодот при текот на извршувањето не се менува, делењето на страниците може да се искористи многу ефикасно, една копија во меморија може да ја користат сите процеси и нивните табели на страници ќе покажуваат на ист мемориски простор на кој се наоѓа кодот. На сликата е илустриран пример за два процеси чии табели на страници покажуваат на иста физичка рамка за страниците.

Слика 7.7: Делење на исти мемориски страници

Оваа техника е особено погодна за користење во случаи кога има програми кои често се користат од страна на повеќе процеси, како на пример, компајлери, графички системи, системски библиотеки и бази на податоци, при што се добива заштеда на мемориски ресурси и заштеда на време за полнење на често користените програми. Заштитата на интегритетот на овие делени страници се постигнува со фино подесување на битот-вите за заштита.

7.2.4. Асоцијативна меморија

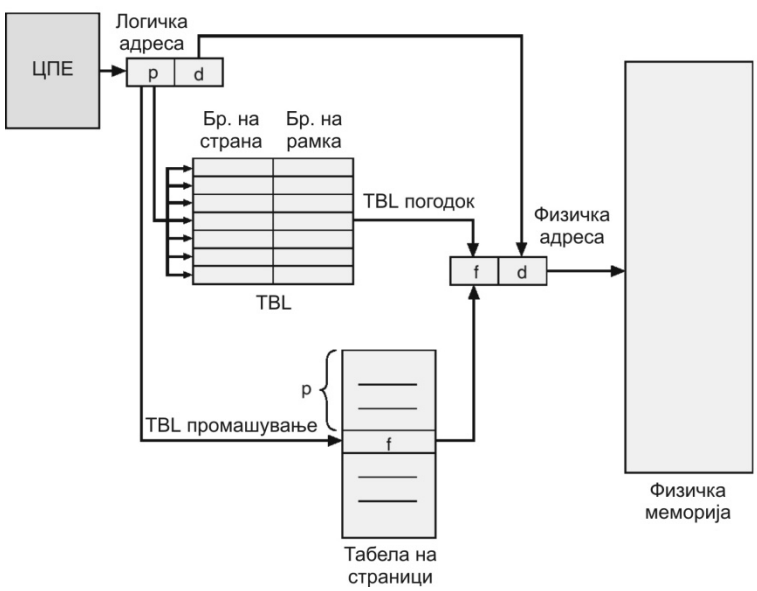
Во повеќето од претставените методи на страничење, табелите на страниците се чуваат во меморијата, или во комбинација главна меморија – диск. Ваквата реализација доведува до нарушување на перформансите и страничењето е бавно и неефикасно. Ако страниците се мали тогаш табелата на страниците ќе биде голема, покрај тоа за секој процес се формира табела на страници. Времето потрошено за одржување на сите тие табели на страници претставува некорисно трошење на време (*англ. overhead*). Друг проблем е и самото референцирање на мемориските локации, на пример, ако воопшто не постои страничење при извршување на една инструкција се прави само една референца кон меморијата за преземање на инструкцискиот код. Со страничењето, потребни се и дополнителни мемориски референци за пристап кон табелата на страници, со што практично се намалуваат перформансите на мемориските циклуси за околу 50%.

Бидејќи програмите имаат тенденција да користат голем број пристапи до мал број мемориски страници може да се реши проблемот на хардверско ниво на самиот процесот преку воведување на специјална кеш структура наречена TLB (*Translation Lookaside Buffer*). Таа претставува мала и брза асоцијативна меморија сместена во процесорот, поточно во единицата за управување со меморијата – MMU, каде што се содржани неодамнешните резултати на пресликувањата (од виртуелна во физичка) и типично содржи 8 до 64 влеза. Ако пристапите до меморијата се "локални" методот работи многу добро но содржината на асоцијативната меморија треба да биде сменета при измена на контекстот (*context switching*).

Валидна	Виртуелна страна	Модифицирана	Заштита	Рамка за страна
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Слика 7.8: Изглед на TLB меморија

Секој влез се состои од два дела: клуч и вредности односно информации за една страница, со бројот на виртуелната страница, бит кој е поставен кога страницата е изменета, ознаката за заштита (*rwe* - дозволи) и физичката рамка на страницата. Овие полиња се соодветни кон полињата во табелата на страници а постои и бит кој покажува дали страницата е валидна, односно дали се користи или не. Во асоцијативната меморија сите излези истовремено го споредуваат својот клуч со влезниот клуч и ако постои идентификација асоцијативната меморија враќа вредност (*TLB hit*).



Слика 7.9: Користење на TLB меморијата

Ако не се пронајде референца односно се случи промашување (*TLB miss*), тогаш физичката страница се бара преку табелите на страници во меморијата како што е веќе опишано и е поедначкво бавно. Освен тоа, треба да се изврши и обнова на TLB баферот со измена и внесување на влезот на страницата кај

која се случило промашувањето. На тој начин, ако во скоро време таа пак ќе се користи тогаш ќе има погодок наместо промашување. Влезот за страницата која се отстранува од баферот се копира назад во табелата на страници во меморијата.

И овој систем како и сите кеш системи брзо работи во случај на погодоци, а многу побавно кога има промашување. Општо ако односот на погодоците и промашувањата е голем системот работи побрзо од систем каде што не е имплементирана асоцијативна меморија за чување на делови од табелата на страници. Голема предност е и хардверската изведба на системот со што се растоварува оперативниот систем од управувањето со работата на TLB баферот.

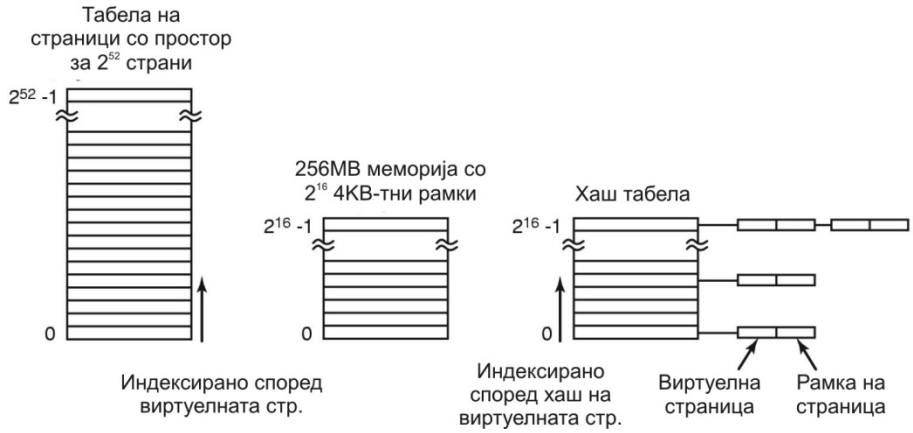
7.2.5. Инвертирана табела на страници

Традиционалните табели на страници бараат еден запис во табелата по виртуелна страница, затоа што се индексирани со бројот на виртуелната страница. Според тоа табелите на страници по процес зависат од големината на виртуелната меморија која станува се поголема.

Кај 32-битните системи, со 4 KB големина на страна постојат 2^{20} влезови во табела на страни, 4 B за еден влез = 4 MB за табела на страни за само еден процес процес. Ситуацијата е уште подрастична за 64-битни системи, со 4 KB големина на страна постојат 2^{52} влезови во табела (4 квадрилиони) и ако секој запис е 8 бајти, табелата ќе изнесува околу 30 милиони гигабајти, што не може да се изведе во постоечките па и идните системи.

Едно решение е инверзна табела на страници (*IPT – Inverse Page Table*) каде што табелата на страници се организира околу физичката меморија, наместо виртуелната меморија, така што во табелата ќе има по еден индекс за една физичка страна. На тој начин за 64-битен адресен простор, со големина на страница од 4 KB и 256 MB RAM , табелата на страници ќе има само 65536 записи. Иако се добива заштеда на количеството на простор, овој пристап има и сериозен недостаток, а тоа е процесот на пресликување од виртуелна во физичка меморија станува многу посложен. Кога еден процес *n* ќе референцира виртуелна страница со ознака *p,* се врши пребарување на целата инвертирана табела за да се најде парот *(n, p)*. Освен тоа, ова пребарување треба да се изврши за секое референцирање на меморијата не само за „*page fault*“ настани. Еден начин да се подобри ситуацијата е да се користи TLB, каде што ако во асоцијативната меморија се содржани страниците кои почесто се користат, преведувањето ќе се случи исто толку брзо како и кај обичната FMPT табела на страници. Меѓутоа, ако се случи промашување, тогаш инверзната табела треба да се пребарува софтверски. Еден начин е да се изврши „хеширање“ (*англ. hash*) на виртуелните адреси. Сите виртуелни страни кои имаат иста хеш вредност се поврзуваат во поврзана листа, така што пребарувањето ќе се врши за помал број на записи, само оние кои се во

листата. Кога ќе се пронајде бројот на физичката рамка, нов пар (виртуелен, физички) се внесува во TLB кешот.



Слика 7.10: Инверзна табела на страници

7.3. Алгоритами за замена на страници

Виртуелна меморија како што беше објаснето е техника која дозволува извршување на процес каде не е потребно сите процесни страни да бидат резидентни во меморијата. Страничењето овозможува изведба на виртуелна меморија и пресликувањето на адресите да се врши динамичко при референцирање на мемориска локација. Со тоа се проширува физичката меморија на диск а само потребните мемориски страници се чуваат во физичката меморија и непотребните страници се "прегазуваат" според одредени критериуми. Голема улога при реализација на систем со виртуелна меморија има политика на замена односно алгоритамите за замена страниците, која од страниците кои се наоѓаат во главната меморија да се отстрани и замени со друга.

Погрешна политика на замена на страници може да доведе до состојба на отпадност (*англ. trashing)* кога поголемиот дел од времето системот е зафатен со префрлување на страниците од дискот кон меморијата и обратно. Друг момент кој треба да се запази е и изборот на големината на страниците. Ако се земат помали страници, се постигнува подобра апроксимација на локалноста, се добиваат и големи табели на страници и неефикасен пренос од дискот во главната меморија, но и помала внатрешна фрагментација и помал број на испади (*англ. page faults*). Поголемите страници, пак се карактеризираат со помала табела на страници, подобро покривање со TLB, поефикасен пренос од и кон дискот како и поголема внатрешна фрагментација. Повеќето модерни системи подржуваат различни големини на страници и тој параметар е конфигурабилен. Сите овие карактеристики на имплементација на систем со

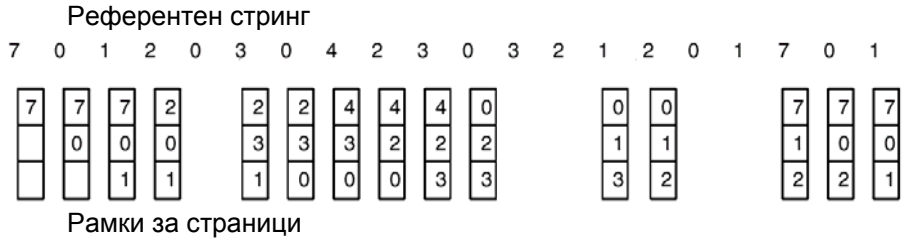
виртуелна меморија се меѓусебно зависни и перформансите на системот зависат од нив како и од изборот на алгоритамот за замена на страниците.

Кога ќе се појави настан „*page fault*“, оперативниот систем треба да избере страница која ќе ја отстрани од меморијата за да направи место за страница која треба да се внесе од дискот. Ако страницата која треба да се отстрани била изменета, таа се запишува на дискот за да се обнови копијата. Ако не била изменета, копијата на дискот е „најнова“ и не е потребно повторно запишување, туку страницата едноставно се отстранува. Иако може да се избере страница за замена по случаен избор, подобро би било да се избере страница која не била многу користена. Во спротивно, ако се избере страница која често се користи, многу бргу пак ќе се појави *„page fault“* настан. Во понатамошниот текст ќе бидат објаснети алгоритамите за избор на страница за замена.

- Прв влезен - прв излезен (*FIFO*)
- Втора шанса (*Second Chance*)
- Часовник (*Clock*)
- Најскоро користена (*LRU*)
- Некористена во скоро време (*NRU*)
- Оптимална (*OPT*)

7.3.1. Прв влезен - прв излезен (FIFO)

First-in First-out е еден од наједноставните алгоритами кој функционира на следниов начин, оперативниот систем одржува листа на сите страници кои моментно се наоѓаат во меморијата, на почетокот со страницата која е најмногу време во листата и на крајот страницата која скоро пристигнала во листата.



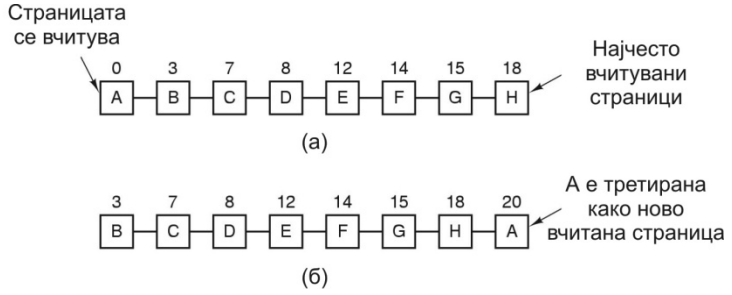
Слика 7.11: Пример за користење на FIFO

При појава на грешка (*англ. page fault*), страницата на почетокот се отстранува и на крајот се додава нова страница. Не се води никаква сметка за користеноста на страницата и можно е да се појави грешка веднаш по

исфрлањето на страницата од почетокот на листата, таа биде повторно референцирана.

7.3.2. Втора шанса

Претставува едноставна модификација на FIFO алгоритамот со кој се избегнува проблемот со отфрлање на страницата која често се користи преку проверка на битот за референцирање. Ако тој е 0, страницата е и стара и некористена и може веднаш да се замени. Ако битот е поставен на 1, тој се брише и страницата се поставува на крајот од листата како нова страница. Потоа се продолжува со барање на страница со бит за референцирање поставен на 0. Функционирањето на алгоритамот – Втора шанса е прикажана Слика 7.12.



Слика 7.12: Алгоритамот „Втора шанса“

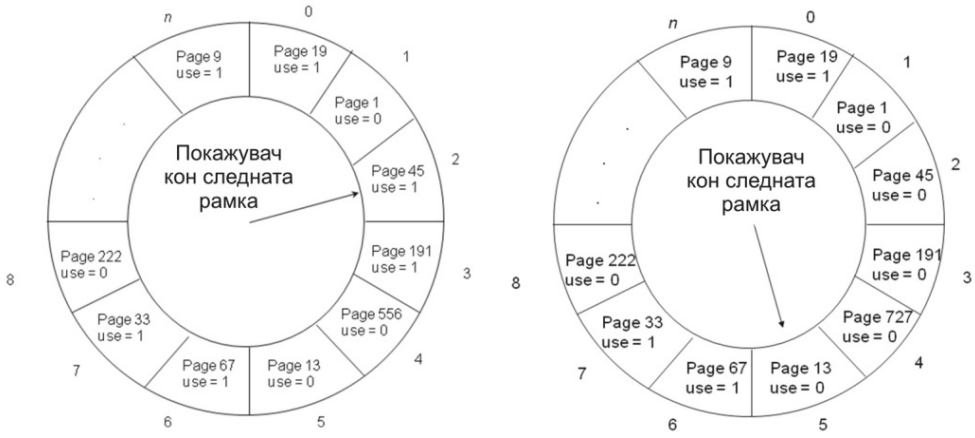
Ако се претпостави дека се појавил *„page fault“* настан во временски момент 20, се гледа дека во листата најстарата страница е со ознака А која пристигнала во временски момент 0. Ако битот за референца е еднаков на нула тогаш страницата *A* се отстранува од меморијата или со запишување на дискот (ако била изменета – „извалкана“) или со едноставно бришење ако била „чиста“. Ако битот за референцирање е поставен, тогаш страницата се поставува на крајот од листата со време на полнење еднакво на 20. Се продолжува со барањето на соодветна страница од остатокот од листата (*B*).

Овој алгоритам всушност бара најстара страница која не била референцирана во одреден временски интервал, ако сите страници биле референцирани, алгоритамот втора шанса се трансформира во *FIFO* и алгоритамот секој пат дава резултат и избира страница за отстранување.

7.3.3. Алгоритам за замена на страници - часовник

Втора шанса е алгоритам е непотребно неефикасен затоа што страниците цело време се префрлуваат од почетокот на крајот од листата. Подобар пристап е страниците да се организирани во форма на циркуларен бафер со покажувач кој покажува на најстарата страница. Кога ќе се појави промашување, се

испитува страната на која покажува поинтерот, ако битот за референцирање е еднаков на еден, се брише битот и покажувачот се поместува на следната страница, ова се повторува сè додека не се најде страница која не била користена. Овој алгоритам овозможува едноставна измена и мал „*overhead*“, исто работи како и политика на втора шанса само поефикасно имплементирана бидејќи нема потреба од преместување во листа.



Слика 7.13: Пример, се бара страницата 272,
и изгледот на баферот по извршената замена

„*Clock*“ алгоритамот покрај битот за референцирање може да го користи и битот за модификација за да се преферира замена на неизменети страници (*англ. clean*) од изменетите (*англ. dirty*) страници. Според тоа, можни се вкупно четири класи на страници:

- u=0; m=0: некористена, неизменета
- u=0; m=1: некористена, изменета
- u=1; m=0: користена, неизменета
- u=1; m=1: користена, изменета

Изменетиот алгоритам работи на следниов начин, при првото поминување се бара страница со вредности на битовите (0,0) при што не се врши измена на битот за референцирање, ако се најде таква се врши нејзина замена. При второ поминување се бара страница со вредности на битовите (0,1) страница и се поставува битот за користење на 0 за секоја помината страна, ако се најде (0,1) тогаш се врши замена. Ако не се успее во наоѓањето на страницата повторно се повторува процедурата и во овој случај сигурно ќе се најде соодветна страница затоа што битовите за референцирање при последното поминување биле поставени на нула. Оваа модификација е слична на алгоритамот „Некористена во скоро време“ - NRU.

7.3.4. Некористена во последно време (NRU)

Кај алгоритамот за замена „*Not Recently Used Page*“ се користат битовите за референцирање и измена за креирање на едноставен алгоритам за замена на страници. При стартувањето на процесот и двата бита за сите страници се поставуваат на вредност 0 од страна на оперативниот систем. Периодично (на пример при секој интерапт на системскиот часовник) се врши бришење на битот за референцирање кај сите страници за да се направи разлика која од страниците не била користена во скоро време. При појава на промашување на страницата (*page fault*), оперативниот систем ги испитува сите страници и врши нивна категоризација по класи, исто како и во претходниот пример.

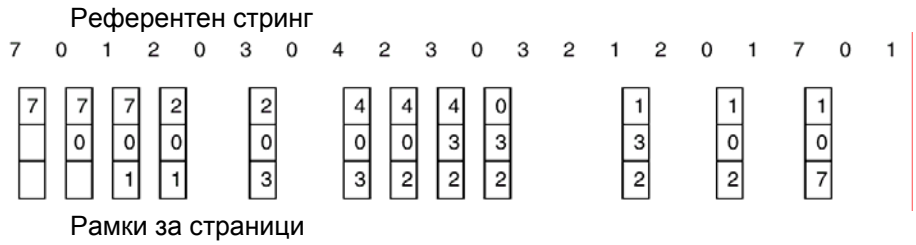
- Класа 0, не и е пристапено, не е променета
- Класа 1, не и е пристапено, променета
- Класа 2 и е пристапено, не е променета
- Класа 3 и е пристапено, е променета

На прв поглед изгледа дека, класата 1 не е можна, но веќе е споменато дека системот врши периодично ресетирање на вредноста на битот за референцирање на 0. Битот за измена не се ресетира, од причина што е потребен при одлучување дали страницата која се отстранува ќе се запишува на дискот или не.

Алгоритамот NRU врши отстранување на страниците по случаен избор од најмалата класа која има членови. Битно за овој алгоритам е дека подобро е да се отстрани измената страница која не била референцирана во одреден временски интервал, отколку неизменета страница која многу често се користи. Добрата страна на овој алгоритам е што е ефикасен за имплементација, едноставен за разбирање и дава перформанси кои иако не се оптимални се прифатливи.

7.3.5. Најмалку употребувана во последно време (LRU)

Алгоритамот „*Least Recentlu Used“* врши замена на страницата што не била користена најдолго време. Кон секоја страница се доделува време на последно користење кое се обновува при секое користење на страницата. За целосна имплементација потребно е одржување на врзана листа на сите страници во меморијата со најскоро користената страница на почетокот и најнекористената страница на крајот од листата. Проблемот е што листата треба да се обновува при секое референцирање на меморијата. Наоѓањето на страницата во листата, нејзиното бришење и поместување на почетокот на листата е операција која троши време и системски ресурси.



Слика 7.14: Пример за LRU алгоритамот

На сликата е прикажан *LRU* на ист пример како и за *FIFO* и може да се види дека предизвикува помал број на промашувања (12 наспроти 15). Алгоритамот заради своите добри страни често се користи и има можност за хардверска имплементација. Може да се реализира на два начини:

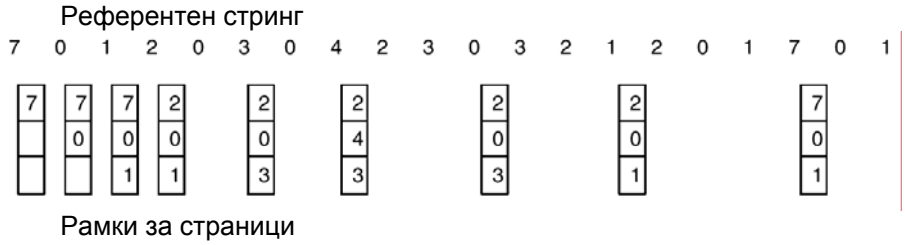
1. Реализација со помош на бројач каде што процесорот има бројач на мемориски инструкции. Секој пат кога ќе се стартува системот, бројачот се поставува на 0. По секоја мемориска инструкција бројачот се зголемува за еден. Секоја рамка си има сопствен интерен бројач и секој пат кога ќе се пристапи кон рамката содржината на бројачот на инструкциите се копира во интерниот бројач на рамката. Кога ќе се случи промашување, алгоритамот ги прегледува сите бројачи и ја одбира таа страница чиј број е најмал (што значи дека најдолго време не била користена).
2. Реализација со помош на стек, слично како и кај алгоритамот *FIFO*, се формира стек кој го опишува пристапувањето кон страниците. По појава на прекин заради промашување, се отфрла страницата на врвот од стекот а на нејзиното место се поставува страницата кон која и се пристапува. Оваа операција е бавна затоа што е потребно стекот да се обновува при секое референцирање на меморијата.

7.3.6. Оптимална замена (OPT)

Оптималниот алгоритам е оној кој дава најмал број на промашувања (*page fault*) и никогаш не доведува до Беладиевата аномалија. При замената се отстранува страницата која најдолго време нема да се користи со што максимално се одолжува времето до појава на следно промашување. Проблемот е непознавањето на временските моменти, односно бројот на инструкциите по кои ќе се појави потреба од пристап кон една страница.

Случајот е идентичен како кај распределувањето на процесите, каде што требаше да се процени колку еден процес ќе го користи процесорското време во иднина.

На сликата е прикажан пример за оптималниот алгоритам, за разлика од *FIFO* или *LRU* критериумот за избор на страница за замена се формира од параметри од иднината и кои не секогаш се познати.

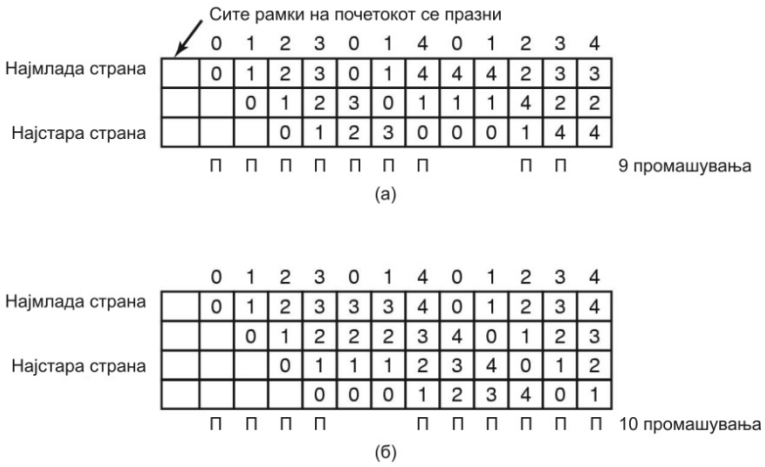


Слика 7.15: Оптимален алгоритам

Овој алгоритам е најдобар можен, но за жал не е можно да се имплементира, затоа што предикцијата за идните вредности не е можна особено кај интерактивните системи. Затоа, најчесто се користи за споредбени анализи и најблиску до оптимална е стратегијата е алгоритамот *LRU*.

7.3.7. Беладиева аномалија (Belady's Anomaly)

Разумно е да се очекува дека без оглед на оптеретеноста, бројот на промашувањата (*page faults*) не би требало да се зголеми ако се зголеми бројот на рамките. Меѓутоа ова не важи секојпат, се покажало дека *FIFO* алгоритамот предизвикува повеќе промашувања со четири рамки наместо со три рамки.



Слика 7.16: Беладиева аномалија

Ова ситуација е наречена Беладиева аномалија и на Слика 7.16 е даден пример за програма со 5 виртуелни страници, означени со 0 до 4. Страниците се референцирани по следниов редослед:

```
0 1 2 3 0 1 4 0 1 2 3 4.
```

На сликата може да се забележи дека со три рамки се предизвикани 9 промашувања, додека за 4 рамки предизвикани се 10 промашувања.

7.3.8. Ефект на заситување

Процес бара работна меморија како ресурс, доколку бројот на достапните рамки падне под минималната вредност, потребно е процесот да се доведе во состојба на чекање, затоа што нема слободен ресурс и тој не може да се извршува. Ако бројот на рамките за еден процес е мал, процесот може да се извршува но почесто се појавуваа промашувања на страници. Тогаш процесот многу често или постојано врши замена на своите страници, што вклучува и голем број на В/И операции со дискот. Појавата на честа измена на страниците, кој е последица од голем број на активни процеси, т.е. висок степен на мултипрограмирањето се нарекува ефект на заситување или отпадност (*англ. thrashing*).

Слика 7.17: Ефект на заситување

Оперативниот систем го следи степенот на искористувањето на процесорот, ако е мал, воведува нови процеси за извршување, односно го зголемува степенот на мултипрограмирањето. Со тоа се зголемува и бројот на промашувањата на страници (*page fault*), што пак резултира во зголемување на В/И операции со што се намалува степенот на искористување на процесорот. Ако оперативниот систем не би бил во состојба да го препознае овој тренд и продолжи да активира нови процеси, тоа би довело до паѓање на перформансите и испад на системот.

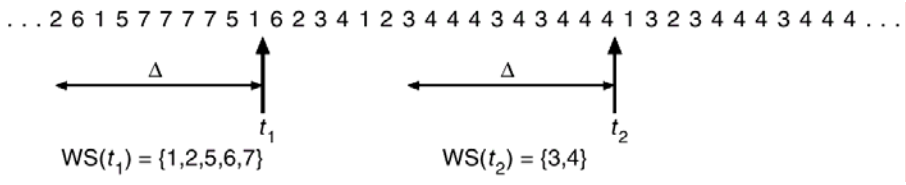
Степенот на искористувањето на процесорот расне со зголемувањето на бројот на процесите до еден момент после кој нагло опаѓа. Заситувањето може да се избегне доколку оперативниот систем престане да го зголемува степенот на мултипрограмирањето.

7.3.9. Работно множество страници

Ефектот на заситување може да се ограничи со користење на локална или приоритетна замена. Најголемиот дел од процесите покажуваат тенденција на локалност, каде што релативно мал број страници се користат. Овие страници формираат "работно множество" на процесот кое во текот на извршувањето се менува. На пример, секој повик на потпрограма формира нова локалност, а по излегувањето од потпрограмата таа се менува. Ефектот на заситување се појавува кога сумата на сите локалности за активните процеси е поголема од физичката меморија.

Ако работното множество е во меморијата, ќе има помал број промашувања (*page faults*) и тоа само кога множеството се менува, ако работното множество не е во меморијата ќе има повеќе промашувања.

Имплементација на работното множество се изведува преку дефинирање на прозорец на работното множество Δ што претставува време во кој ќе се извршат одреден број на инструкции и сите референцирани страници во тој период го сочинуваат работното множество за Δ. На Слика 7.18 се дадени две работни множества за 5 мемориски референци *WS(t1)* ги опфаќа страниците 1,2,5,6,7, а *WS(t2)* страниците 3 и 4.



Слика 7.18: Пример за две работни множества

Најзначајно е големината на прозорецот, се подесува бројот на страните доделен за секој процес според големината на работното множество. Ако вредноста е поголема, ќе постои преклопување на локалноста, а ако вредноста е помала нема да биде соодветна и ќе предизвикува промашувања. Оперативниот систем го следи збирот на локалностите на сите процеси и во оној момент кога ќе стане поголем од достапната меморија се престанува со воведување на нови процеси.

7.3.10. Локална или глобална замена

Во сите претходно опишани алгоритами за замена на страници не беше дефинирано кон кој процес биле доделени рамките за користење. Ако се земе в предвид и овој фактор, може да се дефинираат два вида на замени на страници:

- Глобална замена, каде што сите страници се кандидати за замена. Процес кој промашува може да изврши замена на страници од други процеси. Автоматски ги подесува големините на процесите кон нивните тековни барања, но може да предизивика ефект на заситување. Можно е и дефинирање на приоритети за замена каде што процес со висок приоритет може да заменува свои рамки или рамки на процес со низок приоритет.
- Локална замена: На алгоритамот му се овозможува да избере кандидат за замена од мемориските страници од процесот кој промашува. Бројот на рамките не се менува со текот на времето.

Локалните замени го ограничуваат ефектот на заситување кај еден процес, додека глобалните замени го зголемуваат, ако нема појава на заситување генерално подобра е глобалната замена. Покрај тоа, блокиран процес кај локалната замена не може да ги отстапи своите рамки на другите процеси и активен процес треба да ги жртвува сопствените страници.

7.4. Сегментација

За да се приближи раководењето со меморијата до начинот на кој програмите ја користат (да се ослободи програмерот од грижата за растот на логичките целини на процесите како што се: стекот, табелите, колекциите на променливи итн.) се дефинираат логички целини во изворната програма кои понатаму се преведуваат во мемориски сегменти од страна на преведувачот:

- сегмент на кодот
- сегмент на податоци
- сегмент на стекот
- сегмент на деливите библиотеки
- сегменти дефинирани од корисникот

Сегментација е метода на управува со меморијата која подржува логички приказ на меморијата, каде што логичкиот адресен простор се состои од множество на сегменти, а секој сегмент има единствено име и должина. Логичката адреса се состои од два дела:

- име на сегментот
- поместување во самиот сегмент

При страничењето, логичката адреса е еднодимензионална и при раздвојувањето на деловите на адресите и пресликувањето го извршува хардверот.

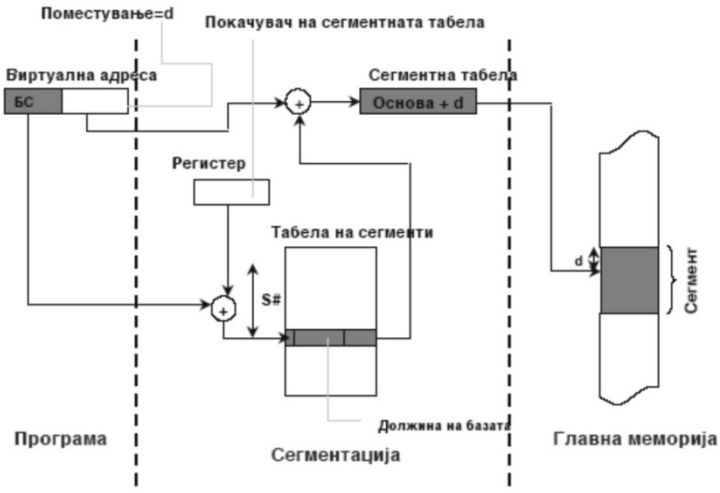
Слика 7.19: Влез во табела на сегменти

При сегментацијата, како и при програмирањето со партициите со променлива должина се јавува екстерна фрагментација. Слободната меморија не може да се искористи за сместување на сегментите доколку не постои доволно голема празнина, без разлика на вкупната големина на слободна меморија. Проблемот на надворешната фрагментација, која зависи од големината на сегментите и распределбата на воведувањето на процесите може да се намали со процесот на збивање на меморијата.

7.4.1. Табела на сегменти

Користењето на сегментите е илустрирано на сликата. Кај методата на сегментација, корисничките адреси се дводимензионални, но треба да се пресликаат во еднодимензионални физички адреси. Ова пресликување се извршува преку табела на сегменти каде што секој запис во табелата опишува точно еден сегмент и содржи два параметри:

- почетната адреса на соодветниот сегмент во главната меморија
- и должината на сегментот

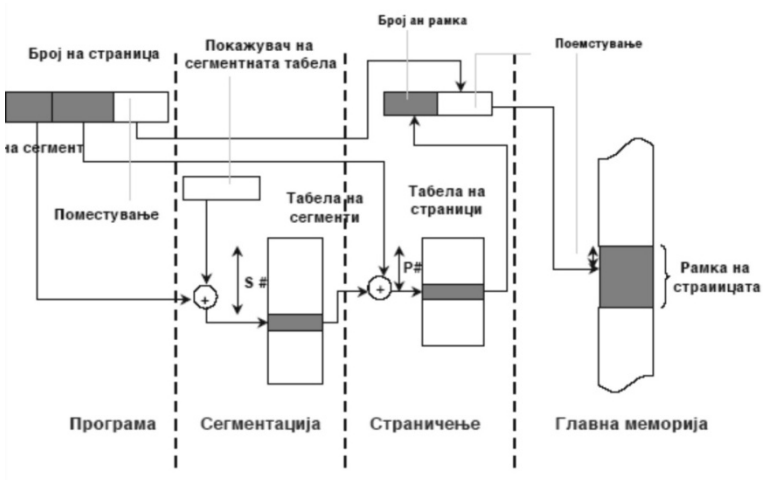


Слика 7.20: Користење на табела на сегменти

Бројот на сегментот се користи како влез во табелата на сегментите, од кој се читаат базната адреса на сегментот и ограничувањето. Поместувањето треба да биде помало од ограничувањето на сегментот. Во табелата на сегментите се содржани и други информации. Со дополнителен бит се проверува (означува) дали сегментот се наоѓа во работната меморија, а потребен е и бит што означува дали е дојдено до промена во сегментот откако тој е сместен во меморијата. Може да се содржи и дефиниција на заштита од модификација, па така сегментите со кодот може да се дефинираат како непроменливи, а може и да бидат поделени со што ефикасно се штеди на меморијата.

7.4.2. Комбинација на сегментација и страничење

Страничењето е транспарентно за програмерот и го отстранува проблемот на надворешна фрагментација, од друга страна пак сегментацијата е видлива и достапна за манипулација од страна на програмерот. Покрај тоа сегментацијата овозможува динамичко растење на податочните структури, модуларност и обезбедува поддршка на делењето и заштитата. Но сегментите може да се преголеми и не може да се чуваат во целост во меморијата. Некои системи имаа вградена поддршка и за сегментација и за страничење (Intel 80x86 и Pentium), што овозможува примена на комбинирани методи за неконтинуирана алокација на меморијата.



Слика 7.21: Комбинација на страничење и сегментација

Секој сегмент се дели во страници со постојана големина, логичката адреса се состои од идентификатор на сегмент (селектор) и поместување во рамките на еден сегмент. Од табелата на сегменти се чита адресата на логиката страница во табелата на страници. Страничењето е реализирано на два нивоа, надворешната табела се нарекува каталог на страници, а внатрешната табела на страници и двата дела имаат иста големина (пример 10 бита).

Слика 7.22: Влезовите во табелите за страничење и сегментација и виртуелната адреса

8. Влезно/Излезни уреди

Една од главните функции на еден оперативен систем е да ги контролира сите влезно излезни уреди од компјутерскиот систем. Тој треба да издава наредби до уредите, да прифаќа прекини и да се справува со евентуални грешки. Исто така треба да обезбеди и интерфејс помеѓу уредите и остатокот од системот кој е едноставен и лесен за употреба. Дури ако е возможно, интерфејсот треба да биде ист за сите уреди со што се обезбедува независност од уредите. Кодот од влезно-излезниот (В/И) систем претставува важен дел од оперативниот систем.

Поголемиот дел од уредите кои се приклучуваат на компјутерскиот систем може да се сместат во одредена општа категорија според намената на уредот. Така може да се издвојат уреди за долготрајно складирање на податоци (дискови и ленти), уреди за пренос на податоци (мрежни интерфејси и модеми), уреди кои обезбедуваат интерфејс спрема корисникот (монитори, тастатури и глувчиња). Овие уреди се разликуваат во многу аспекти, како и по намената така и по множеството на операции кои може да се изведат, брзината на пренос на податоците, количеството на податоците кон кои се пристапува, начин на пристап, деливоста на уредот и множеството на состојби за кои се смета дека се грешки во работата на уредот.

Оперативниот систем треба да обезбеди поддршка за поширок спектар на В/И уреди чии карактеристики се разликуваат од уред до уред од ист тип, како и со конкретниот модел на одреден производител на хардвер. Како последица на тоа, отежнат е дизајнот и програмирањето на В/И подсистемот на оперативниот систем. Главните функции на В/И делот од еден оперативен систем:

- ги контролира В/И уредите;
- задава команди кон В/И уреди;
- работи со прекини и грешки на уредите;
- обезбедува интерфејс помеѓу уредите и остатокот од системот.

Кодот со кои се опишани функциите на В/И подсистемот зафаќа поголем дел од кодот на самиот оперативен систем.

Заради производство на огромен број на различни хардверски уреди од ист или различен тип и карактеристики, поддршката за одредени уреди или група на сродни уреди е содржана во програмите за управување со уредите, кои се нарекуваат драјвери (*англ. drivers*). Тие претставуваат единствен интерфејс помеѓу В/И подсистемот и самиот хардверски уред.

Од главен интерес за дизајнерите на оперативни системи е програмирање на В/И уреди, а не нивно дизајнирање, при што програмерот треба да е запознаен

со принципите на В/И хардверот, слоевита организација на В/И каде секој слој да има добро дефинирана задача. При развојот треба да се внимава да се запазат основните услови и цели за програмирање на еден В/И подсистем:

- *Независност од уредите*. Различни модели на уреди од ист вид треба да се разгледуваат подеднакво, В/И системот не смее да биде оптоварен од познавањето на самиот модел на уредот од одреден тип, на пример конкретен модел од некој производувач. Втор услов е независност на програмата од конкретен вид на уредите. На пример треба да се постигне системот да врши читање на податоците на ист начин и од дискот и од оптичкиот медиум. Како последица на овие услови, програмите работат врз основа на виртуелни уреди преку кои се изведуваат сите В/И операции.
- *Независност од знаковниот код*. Корисникот не треба да има детално познавање на знаковниот код кој е користен во конкретни уреди. В/И подсистем ги презема обврските за препознавање на одреден знаковен код и претставување на податоците на корисникот во стандарден облик. Затоа постои внатрешен знаковен код односно униформни претстави на сите знакови во уредот и механизам за преведување на кодот како на пример преведувачка табела или помала програма.
- *Ефикасност*. Секој уред е значително побавен од работната меморија и процесорот, што значи В/И уредите се тесно грло на системот и операциите на д нив треба да се извршат на што поефикасен начин. Покрај тоа, подсистемот треба да обезбеди конкурентност во работата со уредите.
- *Униформен интерфејс* кон апликациите и корисникот. Треба се овозможи униформно ракување со цел да се олесни работата на корисниците. Заради разновидноста на уредите, оваа цел е тешко да се постигне, постојат начини за справување со овој проблем како претставување на уредите како датотеки (кај Linux).

8.1. Класификација на уредите

В/И уредите се разликуваат во многу карактеристики кои зависат од одреден модел. Уредите генерално може да се класифицираат според следниве критериуми.

1. Според намена уредите се категоризираат во:
   - За долготрајно чување на податоци (*англ. storage devices*), дискови и магнетни ленти.
   - За пренос на податоци (*англ. transmission devices*), мрежни картички и модеми.

- Уреди кои обезбедуваат интерфејс кон корисникот (тастатура, глувчиња, монитори).

2. Според насока на пренос:
   - Влезни (глувче, скенер);
   - Излезни (печатач, монитор);
   - Влезно-излезни (мрежна картичка);
3. Според методи на пристап:
   - Уреди со секвенцијален пристап (модем, магнетна лента);
   - Уреди со случаен пристап (дискови, оптички уреди).

Секвенцијалните уреди ги пренесуваат податоците по фиксен редослед кој го определува самиот уред, додека кај уредите со директен пристап редоследот на преносот е произволен и одреден од самата апликација.

4. Според деливост на уредите, се категоризираат во *деливи* и *неделиви*. Деливи уреди може да се користат од страна на повеќе процеси истовремено (на пример дискови), додека неделивите само еден процес (пример тастатура).
5. Според брзината на уредите. Уредите се делат на *брзи* и *бавни* и оваа поделба е релативно во однос на големите варијации во брзините на уредите од неколку бајти во секунда до неколку гигабајти во секунда (Табела 8.1), дури и во однос на моделот и производителот за уреди од ист вид.
6. Уредите според можност за запишување се делат на оние кои извршуваат читање и запишување (дискови, флеш меморији), исклучиво за читање (CD-ROM, DVD-ROM) и исклучиво за запишување (монитор, графичка картичка).

Настрана од претходно наведената поделба, В/И уредите може грубо да се поделат во две категории, *блок уреди* (*англ. block devices*) и *карактер уреди* (*англ. character devices*).

Блок уредите се оние кои што складираат информации во блокови со фиксна големина, секој со сопствена адреса. Вообичаените големини на блоковите се движат од 512 бајтни па до 32.768 бајтни, па и поголеми. Значајна карактеристика на блок уредите е дека е невозможно независно да се чита или пишува секој блок од сите други. Дисковите се карактеристични блок уреди.

Табела 8.1. Видови на уреди и нивните брзини на пренос на податоците

Уреди	брзина на пренос
Тастатура	10 bit/sec
Глушец	100 bit/sec
56 модем	7 KB/sec
I.1.1. Телефонски канал	I.1.2. 8 KB/sec
I.1.3. ISDN линија	I.1.4. 16 KB/sec
Ласерски принтер	100kb/sec
скенер	400kb/sec
класичен Етернет	1.25Mb/sec
USB	1.5Mb/sec
Дигитална камера	4Mb/sec
IDE диск	5Mb/sec
40x CD ROM	6Mb/sec
Fast ethernet	12.5Mb/sec
ISA bus	16.7Mb/sec
FireWire (IEEE 1394)	50Mb/sec
XGA монитор	60Mb/sec
SONET OC-12 мрежа	78Mb/sec
SCI ultra 2 диск	80Mb/sec
Gigabit Ethrnet	125Mb/sec
Ultrium tape	320Mb/sec
PCI bus	528Mb/sec
Sun Gigaplane XB backplane	20gm/sec

Друг тип на В/И уреди се карактер уредите. Карактер уредот дава или прифаќа низа од карактери, без обрнување внимание на некаква блок структура во податоците. Тие уреди не се адресибилни и не постои операција со која се врши пребарување (*англ. seek*).

Оваа класификација за некои видови на уреди не може да примени, како на пример системскиот часовник, тој не се блок адресибилен, ниту генерира или прифаќа карактери. Неговата улога е да предизвикува прекини во прецизно дефинирани интервали. Сепак, моделот со блок и карактер-уредите е доволно општ за може да биде искористен како основа за дизајн на некој од програмите на оперативниот систем и се справува со В/И уредите независно. Датотечниот систем работи со апстрактни блок уреди и го остава делот зависно од уредите на пониските софтверски слоеви на В/И подсистемот.

В/И уредите, како што беше веќе споменато покриваат широк опсег на брзини што придонесува кон усложнувањето на условите во кои треба да работи софтверот. Во табелата се прикажани некои видови на уреди, каде за повеќето од нив се очекува да станат и понапредни и побрзи.

8.2. Архитектура на В/И системот

Уредите што се поврзани на еден компјутерски систем комуницираат со него преку жични и безжични преносни медиуми, а на него се поврзани преку одреден конектор. Архитектурата на хардверот што поврзува еден В/И уред се состои од 4 делови. Магистралата (*англ. bus*) е врска на уредите со компјутерот каде што повеќе уреди ја делат и ја користат преку дефиниран протокол. Сериската врска на која е приклучен само еден уред каде од него се врши поврзување кон следниот во низа се нарекува (*англ. daisy chain*) и функционира на сличен начин како и магистралата.

В/И уредите се состојат од механички и електронски компоненти. Обично се врши поделба на две компоненти за да се оствари модуларност и генерализиран дизајн. Електронската компонента се нарекува контролер на уреди или адаптер. Кај персоналните компјутери, обично изгледа како електронска печатена плочка која се приклучува во слотовите за проширување (*англ. expansion slots*), а механичката компонента е самиот уред. Картичката на контролерот обично има конектор, во кој се поврзува кабелот за поврзување со уредот.

Многу контролери опслужуваат и повеќе уреди (два, четири, дури и осум идентични уреди). Ако интерфејсот помеѓу контролерот и уредот е стандарден интерфејс, или официјалниот ANSI, IEEE или ISO стандардот, тогаш компаниите може да ги прилагодат контролерите или уредите да одговараат на дадениот стандард за интерфејсот. Многу компании, на пример произведуваат диск единици кои одговараат на IDE или SCSI интерфејсот.

Интерфејсот помеѓу контролерот и уредот е многу често на ниско софтверско ниво. Дискот на пример, може да биде форматиран со 256 сектори од 512 бајти по една патека (трака). Всушност од дискот се очитува сериска низа од битови, која започнува со преамбула, па потоа 4096 битови во сектор и конечно сума за проверка (*checksum*), позната како код за корекција на грешки (*ECC - Error Correcting Code*). Преамбулата се запишува кога дискот е форматира и го содржи бројот на цилиндарот и секторот, големината на секторот и други податоци, како и информација за синхронизација.

Задачата на контролерот е да изврши конверзија на серискиот поток на битови во блокови од бајти и да изведе проверка и детекција на грешки. Блокот од бајтите се прибира, бит по бит, во привремената меморија, баферот во контролерот. По верификацијата на проверката и ако блокот нема грешки, може да се пренесе во главната меморија.

Контролерот за мониторот исто така работи како бит сериски уред на еднакво ниско ниво. Ги чита бајтите кои ги содржат карактерите кои што треба да се прикажуваат од меморијата и се генерираат сигнали кои вршат модулација на зракот од електронскиот топ да го поминува фосфоресцентниот екран првин хоризонтално па вертикално. Ако не постоеше контролерот, програмерот на оперативни системи би морал експлицитно да го програмира аналогното

скенирање на цевката на екранот. Со користење на контролерот, доволно е оперативниот систем да изврши иницијализација со неколку параметри како што се на пример бројот на карактери или пиксели од линија и бројот на линии од екранот и да му дозволи на контролерот да се грижи за движењето на електронскиот зрак.

8.2.1. Мемориско пресликани В/И уреди

Портата со која контролерот се поврзува на магистралата се наоѓа на уредот и служи за прифаќање на командите и нивните параметри од магистралата и за поставување на резултатите од бараната операција. Контролерот има и неколку регистри со кои се врши комуникацијата со процесорот. Со запишување на податоците во овие регистри, оперативниот систем управува со операциите на пренесување на податоци кон и од уредот, вклучување или исклучување на уредот или очитување на статусот.

Типично во состав на контролерот можат да се најдат следниве 4 регистри:

- Статусен регистар кој покажува дали уредот е зафатен, дали податоците се подготвени, дали има грешка при преземањето на податоците од уредот.
- Контролен регистар во кој се сместува кодот на командата која треба да се изврши (читање, запишување или некоја друга.
- Регистар за влезни податоци каде што се поставува податок што се испраќа од уредот кон процесорот)
- Регистар за излезни податоци кои што се испраќаат од процесорот кон уредот.

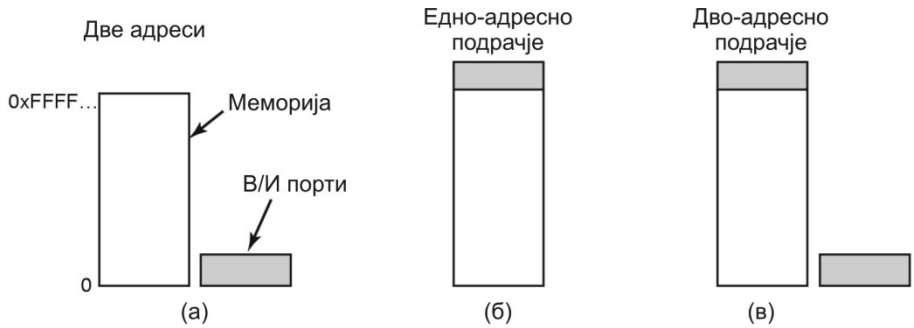
Според начинот на кој процесорот пристапува кон регистрите можни се два начини на организацијата во однос на меморијата. Ако регистрите се дел од регуларниот мемориски адресен простор, шемата се нарекува мемориски - мапиран В/И. Спротивно на ова, регистрите се наоѓаат во специјален адресен простор за В/И и им се пристапува со посебни машински наредби и се нарекува директен В/И.

Во првиот случај, контролните регистри се пресликуваат во меморискиот простор (Слика 8.1). На секој контролен регистар му се доделува единствена меморија адреса која не спаѓа во составот на главната меморија. Обично доделените адреси се наоѓаат на врвот од адресниот простор.

Во вториот случај на секој од регистрите им се доделува број на В/И порта, обично 8 или 16 битна целобројна вредност. Со користење на специјална В/И инструкција како што е IN REG, PORT, процесорот може да ја прочита содржината на регистарот PORT и резултатот да го смести во регистар на процесорот REG. Слично на тоа преку користење на OUT PORT, REG

процесорот ја запишува вредноста во регистарот од процесорот REG во регистарот на контролерот PORT. Кај овој начин на организација на В/И системот адресниот простор на меморијата и на В/И уредот се различни (Слика 8.1). Хибридната шема претставува комбинација од двете претходно наведени шеми. Пример за користење на оваа шема може да се најде кај системите базирани на Пентиум процесорот, каде адресниот простор од 640К до 1М е резервиран за податочните бафери, со дополнителен В/И адресен простор со вредности од 0 до 64К.

Во сите случаи, кога процесорот сака да чита податочен збор од меморијата или од В/И портата, ја поставува адресата на адресните линии на магистралата и вметнува сигнал за читање на контролната линија на магистралата. Се користи и сигнална линија со која се покажува дали се користи меморијата или В/И адресниот простор. Ако се однесува на меморискиот простор, меморијата одговара на барањето, односно се врши декодирање на локацијата и поставување на содржината на податочната магистрала. Ако се однесува на В/И уред, се извршуваат бараните операции.



Слика 8.1: Мемориско пресликани В/И уреди

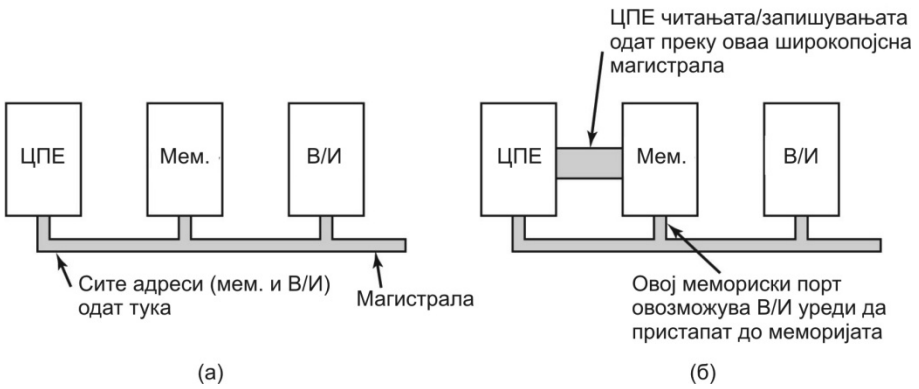
Ако постои само еден адресен простор, секој мемориски модул и В/И уред ја очитува информацијата на адресната магистрала и ги споредува и ако адресниот простор одговара (се декодира адресата) меморијата или уредот одговараат на барањето. Затоа што нема двојни адреси, нема двосмислености и можност за појава на конфликти.

И двете архитектури си имаат свои предности и недостатоци. Кај мемориско мапирани В/И, регистрите на уредот претставуваат обични променливи и лесно се пристапува кон нив. Што значи дека може да се користат и инструкции и од виши програмски јазици за читање и запишување во контролните регистри на уредите. На пример, еден драјвер за уред може комплетно да се изработи во програмскиот јазик С, инаку би требало да се користи и асемблерски код.

Друга предност кај мемориско мапирани В/И е што нема потреба од заштитен механизам при извршување на В/И операции од страна на корисничките процеси. Оперативниот систем не треба да дозволи давање на делот од

адресниот простор во контекстот на процесите. В/И регистрите на уредот можат да бидат во различни адресни простори и доделувањето кон кориснички процес се врши преку вклучување на мемориските страници кои го содржат адресниот простор на В/И во табела на страници за одреден процес. Трета предност се состои во тоа што секоја инструкција која може да референцира меморија може да референцира и контролен регистар на уредот не само инструкции за движење на податоците (како што е MOV).

Мемориско мапираните В/И уреди си имаат и свои недостатоци. Сите компјутерски системи имаат некоја форма на кеширање на мемориските зборови. Кеширањето на контролните регистри на уредите може во најмала рака да има непредвидливи последици. За да се надмине овој недостаток, хардверот треба селективно да го исклучи кеширањето врз основа на референцираните мемориски страници. Од друга страна тоа додава поголема комплексност и на хардверот и на оперативниот систем за селективно управување на кеширањето. Друго ако постои само еден адресен простор тогаш сите мемориски модули и В/И уреди треба да ги испитаат сите мемориски референци за да се види кој од нив треба да даде одговор. Ако компјутерот има само една магистрала, проблемот е прилично едноставен. Меѓутоа повеќето од новите системи имаат повеќе магистрали, каде што постои посветена мемориска магистрала со голема брзина заради оптимизација на мемориските перформанси, или дури и повеќе (3 надворешни магистрали како кај Пентиумот, меморија, PCI, ISA). Со користење на архитектури со повеќе магистрали се појавува проблем каде В/И уредите не можат да ја видат вредноста поставена на мемориската магистралата и не постои начин да одговорат на евентуалното барање насочено кон нив. Затоа треба да се преземат специјални мерки, едно решение би било да се пратат сите референци кон меморијата па ако барањето појави грешка, потоа да се проследи кон другите магистрали, но се усложнува хардверот.



Слика 8.2: Архитектура со една и повеќе магистрали

Друго решение кое може да се примени би било да се пренесат адресите кон сите магистрали и кон меморијата и кон В/И уредите. Проблемот е што В/И уредите работат на брзини многу помали од меморијата и не може благовремено да одговорат.

Трето решение е да се користи филтрирање на адресите во чипот на мостот (*PCI bridge*). Опсегот на не мемориските адреси се вчитува при процесот на подигањето на системот. На пример, 640K-1M е обележен како не-мемориски опсег и се насочува кон PCI магистралата наместо кон меморијата. Недостаток на овој пристап е проблемот да се одреди кои мемориски адреси спаѓаат во овој опсег при процесот на подигнувањето на системот.

8.2.2. Комуникација оперативен систем – В/И уред

Комуникацијата помеѓу оперативниот систем – процесорот со В/И уредот може да се опише според принципот произведувач - потрошувач, чија реализација може да се изведе со помош на два бита: зафатен (*англ. busy*) и очекува команда (*англ. command ready*). Битот зафатен во статусниот регистар на контролерот ја опишува состојбата на уредот. Ако е поставен на зафатен, контролерот нешто работи и не може да одговори на поставената команда. Ако е поставен на слободен тој е подготвен за прифаќање на команда.

Процесорот укажува на контролерот на присуство на нова команда преку битот за подготвеност на командата. Кога овој бит ќе биде поставен тогаш контролерот ја добил командата и може да започне со работа.

Техниката за комуникацијата може да се објасни преку пример на испраќање на еден бајт по сериски канал кој се одвива по следниот редослед:

1. Процесорот го очитува регистарот на контролерот (работно-чека) се додека статусот на контролерот не стане „*idle*“, односно *„busy“* битот не стане 0.
2. Процесорот поставува вредност за запишување во командниот регистар (*англ. status ready)* и ги запишува податоците во податочниот регистар (ако е излезна операција).
3. Процесорот го поставува битот за подготвеност на командата при што контролерот реагира на статусот *ready* и го поставува статусниот бит на *busy*.
4. Контролерот ја чита командата од командниот регистер и ја извршува испраќајќи или примајќи податоци од податочните регистри
5. По завршувањето на операцијата ако не се појавиле грешки, контролерот го брише битот за подготвена команда, го брише битот за грешка и го менува статусот во *idle* со поставувањето на битот *busy* на 0.

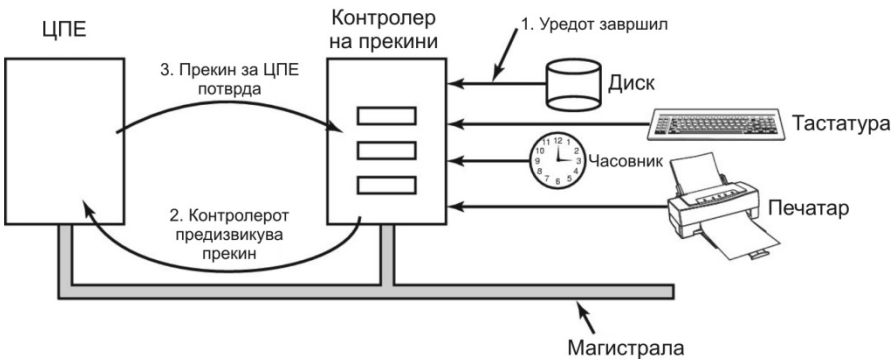
6. Процесорот кој до тогаш го испитува статусот на битот за подготвена команда го гледа новиот статус на контролерот и ги превзема податоците (ако е влезна операција).

Системот треба целата постапка да ја повтори за секој циклус на читање или пишување кон В/И уредот. Техниката се нарекува прозивка (*pooling*) или уште и работно чекање. Системот цело време врши проверка на статусниот регистар и го анализира битот за зафатеност сè додека неговата вредност на стане 0. Целиот процес може да трае кратко за брзи и подолго за побавните уреди.

8.2.3. Прекини (Interrupts)

Основен недостаток на техниката на прозивка е трошењето на процесорското време. Наместо процесорот да врши работно чекање додека уредот да стане слободен, уредот може да иницира прекин (*англ. interrupt*) на процесорот кога ќе ја комплетира претходно зададената операција. Тоа се извршува со вметнување на сигнал на соодветната за тоа линија на магистралата кој понатаму се детектира од страна на контролерот на прекини кој одлучува за понатамошна акција.

При појава на В/И иницран прекин, процесорот одредува кој уред го предизвикува прекинот, прекинува со извршувањето на тековниот процес за да го обработи прекинот. По завршувањето на обработката, процесорот продолжува со извршувањето на процесот кој бил прекинат.



Слика 8.3: Обработка на прекин

Бројот кој се појавува на адресната линија при појавата на прекинот го одредува уредот кој го предизвикал прекинот. Бројот се користи како индекс во табелата се нарекува вектор на прекин (*англ. interrupt vector*), преку кој се одредува нова вредност на програмскиот бројач соодветна на сервисната процедура. Локацијата на векторот на прекини може да биде препрограмирана

на чипот или да се наоѓа на било која мемориска локација. Обработката на прекинот е прикажана на Слика 8.3.

1. Кога В/И уред завршува со работа, предизвикува прекин со давање сигнал на својата интерапт магистрала, кој е детектиран од интерапт контролерот.
2. Контролерот го става бројот на адресната линија специфицирајќи кој уред бара прекин на процесорот.
3. Сервисната рутина по започнување со извршување го потврдува прекинот со запишување на одредена вредност во В/И портовите на интерапт контролерот.

Кај модерните оперативни системи, техниката на прекини овозможува и бројни софистицирани особини. Можност за одлагање на прекинот, додека процесот се наоѓа во критичната секција, се изведува со задоцнување на потврдата од страна на процесорот (3) со која се кажува дека процесорот е подготвен да прими нов прекин. Со тоа се избегнува појава на состојба на натпревар при повеќекратни симултани прекини.

Брза и ефикасна техника на одредување на уредот кој го поставил прекинот, за да не се врши анализа кој од уредите предизвикал прекин. Потоа вгнездување на прекини кој овозможува прифаќање на прекините и во периодот кога веќе се извршува сервисната рутина на прекин, притоа системот треба да има механизам за разликување на приоритети на прекини.

Поголемиот дел од процесорите имаат две засебни линии за сигналите за прекини. Едната служи за пренесување на немаскирачки прекин (*англ. non masking*), кој со секое време може да го прекине извршувањето на тековниот процес. И другата за пренос на маскирачки прекини (*англ. masking*), кои потекнуваат од нормални операции и стандардни грешки кои се појавуваат од влезно-излезните уреди.

8.2.4. Директен пристап до меморијата (DMA)

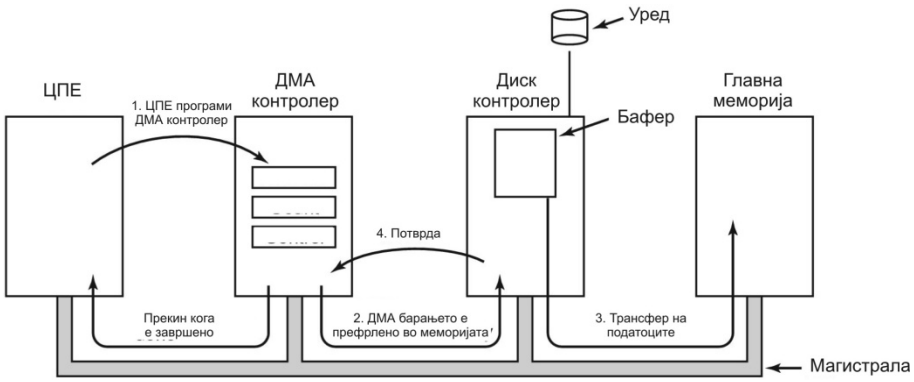
Без разлика дали е имплементирана мемориски пресликан В/И или не е, процесорот треба да го адресира уредот за да изврши пренос на податоци кон и од него. При поголеми трансфери на податоци (диск, CD, DVD) не е оптимално процесорот да биде работно ангажиран, затоа се користи друг начин за директен пристап до меморијата *DMA (Direct Memory Access)*. За да се користи DMA потребно е хардверот да има соодветен контролер, што е правило во поновите системи. Контролерот може да биде интегриран во диск или другите контролери, но ваквиот пристап бара посебен контролер за секој уред поединечно. Обично постои еден контролер вграден на матичната плоча за управувањето со преносот на податоците за повеќе уреди истовремено.

DMA контролерот без разлика каде се наоѓа, има директен и од процесорот независен пристап кон податочната магистрала (Слика 8.4). Содржи неколку регистри кои можат да бидат пристапувани од процесорот, мемориски адресен регистар, бројач на бајти и еден или повеќе контролни регистри. Контролните регистри ја специфицираат В/И портата која ќе се користи, насоката на преносот (читање или запишување), преносната единица (бајт или збор) и бројот на бајтите кои треба да се пренесат.

Во случајот ако не се користи DMA (PIO - програмиран В/И), операцијата за читање од дискот би се извршувала на следниов начин:

1. Контролерот го чита блокот од дискот сериски, бит по бит, сè додека целиот блок не се најде во внатрешниот бафер во циклус на контролерот;
2. Се врши проверка да не се случиле грешки при читањето со споредувањето на контролната сума (*checksum*);
3. Потоа контролерот предизвикува прекин;
4. Оперативниот систем во циклуси врши читање на бајт или збор од баферот на контролерот и го сместува во меморија (по еден бајт/збор во итерација).

Очигледно дека има непотребно губење на процесорско време за изведувањето на циклусите на читање, затоа е и создаден DMA, за да го ослободи процесорот од оваа обврска.



Слика 8.4: Операција за читање од диск со DMA

Кога се користи DMA изведба на уредот, тој има пософистициран контролер што може директно да запишува во меморијата наместо во регистрите за влез и излез од процесорот.

1. Процесорот го програмира контролерот преку поставување на вредности во неговите регистри, за да се знае што и каде треба да се пренесува. Му дава на DMA контролерот покрај диск-адресата на

блокот што се чита и уште два параметри: мемориска адреса каде што ќе се запише блокот и број на бајти што треба да се пренесат (бројач).

2. DMA контролерот иницира пренос преку издавање на барање на читање на магистралата до диск контролерот. Барањето е исто како и секое друго, така што диск контролерот не знае дали потекнува од процесорот или DMA контролерот. Обично мемориската адресата на која треба да се запишат податоците се наоѓа на адресната магистрала, така што диск контролерот знае каде да го изврши запишувањето.
3. Запишувањето во меморијата е друг стандарден циклус на магистралата (чекор 3).
4. По завршувањето на запишувањето, диск контролерот испраќа сигнал за потврда кон DMA контролерот преку магистралата. DMA контролерот потоа ја зголемува мемориската адреса и го намалува бројачот на бајти. Операцијата се повторува од чекорите 2 до 4 се додека бројачот не падна на нула.

По изведувањето на претходно наведените чекори, DMA контролерот издава прекин на процесорот и му става до знаење дека трансферот е завршен и оперативниот систем не треба да врши пренос, потребните податоци веќе се наоѓаа на предвидената мемориска локација. DMA контролерот пренесува податоци во два чекора од дискот во интерниот бафер и кон главната меморија. Додека податоците се пренесуваат од контролерот до меморијата, дискот ротира. Следниот блок за читање на дискот, ако физички е сместен веднаш до првиот заради времето потребно да се извршат DMA операциите не може да бие прочитан во внатрешниот бафер. Ќе треба да се чека уште една ротација за да се прочита и при што се губи време уште повеќе време. Затоа се воведува фактор на преплетување (*англ. interleave*) каде што соседните блокови на дискот физички се организирани преку една или повеќе позиции, на тој начин што кога ќе измине времето за читање и запишување од баферот во меморијата, главата за читање да се најде позиционирана на следниот блок кој треба да се прочита.

Усогласувањето помеѓу DMA контролерот и контролерот на уредот се изведува преку сигналите *DMA-request* и *DMA-acknowledge*. В/И контролерот иницира DMA циклус само ако има подготвен податок со издавањето на *DMA-request*. По примањето на овој сигнал DMA ја зазема системската магистрала и започнува мемориски циклус со кој бајтот од В/И контролерот се пренесува во меморијата. По завршувањето на циклусот DMA контролерот поставува сигнал *DMA-acknowledge* што означува крај на еден DMA циклус.

Со користење на DMA контролер, работата на процесорот се прекинува дури кога ќе заврши трансферот, притоа DMA контролерот и процесорот се натпреваруваат за системската магистрала, која може во еден момент да припаѓа само на еден од нив. Постојат хардверски сигнали за нивната

синхронизација, при што процесорот може да ги задоволи неговите барања од кеш меморијата. При тоа на релацијата DMA контролер – процесор се случува крадење на циклуси (*англ. cycle stealing*): DMA чека на магистралата да стане слободна и ја зазема, а за тоа време процесорот мора да чека ако сака пристап до меморијата. Без разлика на тоа, DMA го ослободува процесорот од преносот на податоци од меморија кон и од уредите, со што се подобруваат перформансите на системот.

Некои компјутерски системи користат физички адреси за DMA, додека другите пак користат виртуелни адреси, тие извршуваат и адресно мапирање и може да пренесе податоци помеѓу два мемориски пресликани уреди и без користење на процесорот или меморијата.

8.3. Принципи на В/И софтвер

Досега беше разгледуван начинот на работа на В/И хардверот, па подетално ќе бидат разгледани целите и концептите врз кои се базира софтверот за управување со В/И операциите во состав на еден оперативен систем. За него може да се каже дека е организиран во низа на нивоа каде што пониските ги кријат специфичностите на хардверот од повисоките нивоа а тие овозможуваа едноставен и разбирлив интерфејс кон корисниците, односно апликативните програмери.

Целите на В/И софтверот кои треба да се постигнат или запазат произлегуваат од претходно наведените цели при развојот В/И системот на оперативниот систем како целина.

1. Независност од уредот. Клучниот концепт при дизајн на В/И софтверот е независност од уредите, што значи дека треба да биде можно да се пишува софтвер кои ќе пристапуваат кон било кој В/И уред без да има потреба да се специфицира однапред. Програмите не треба да се менуваат за различните уреди, на пример командата за сортирање која го има обликот

   ```
   sort < input > output
   ```

   треба да работи на потполно ист на чин, независно дали се работи за датотеки што се наоѓаат на флопи, диск, или пак се праќаат или читаат од терминал.

2. Именување и лоцирање на уредите. Имињата на датотеките или уредите претставуваат низа од integer броеви кои не се зависат од уредот на било кој начин. Во UNIX оперативниот систем секој уред има своја датотека која се наоѓа во директориумот /dev/ чиј i-јазол во датотечниот систем содржи вредности наречени „*major*“ и „*minor*

device number“, каде што првиот служи за лоцирање на драјверот, а втората вредност претставува параметар за драјверот за да знае со кој уред се работи. На пример флопи диск може да се прикачи (монтира) на дел од директориумското стебло, па со копирање на датотека во тој директориум всушност се врши копирање на датотеката на флопи дискот. На овој начин сите датотеки и уреди се адресираат на ист начин: со името на зададена патека (*англ. path name*).

3. Раководење со грешките. Општо грешките треба да се решаваат на колку е можно пониско ниво до хардверот. Ако контролерот открие грешка треба да се обиде самостојно да ја исправи, на пример со повторно читање на блокот од дискот, бидејќи причина за грешката може да била прашина на тој дел од дискот па со повторување на операцијата причината може да ја снема. Ако пониските нивоа не се во состојба да ја решат грешката, информацијата за грешката се пренесува на повисоките нивоа.

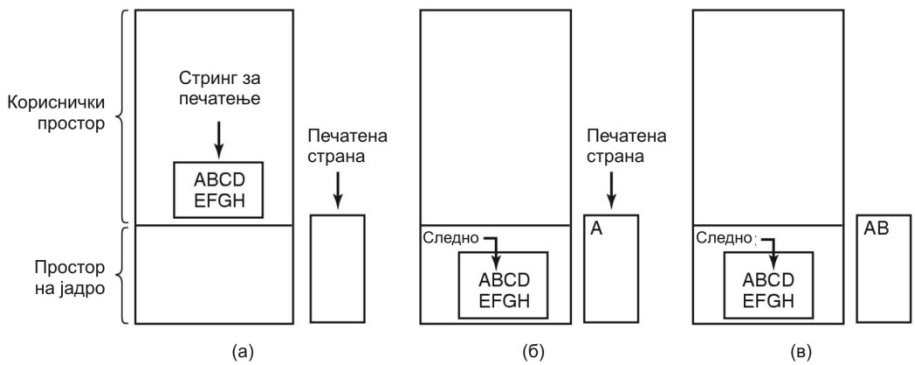
4. Прекини и блокирања. Друг клучен концепт се синхрони (блокирачки) наспроти асинхрони (прекини) трансфери на податоци. Повеќето В/И уреди се асинхрони, процесорот ќе започне операција на трансфер и ќе работи нешто друго до пристигнувањето на прекинот. Корисничките програми се полесни за пишување ако В/И операциите се блокирачки – по издавање на системски повик, програмата автоматски се суспендира сè додека податоците не станат достапни. Оперативниот систем тој што треба да создаде илузија кон корисникот дека операциите водени од прекини (*англ. interrupt driven*) се претставуваат како блокирачки трансфери (како системот да го чека само тој корисник да ги добие или прати податоците).

5. Баферирање. Обично податоците кои доаѓаат од уредот не можат да бидат сместени во конечната дестинација. На пример, ако се презема пакет од мрежата, оперативниот систем не може да знае каде треба да го упати сè додека некаде претходно не го смести и го обработи. Се користат техники како што се баферирање, кеширање или спулирање за да се овозможи ефикасен трансфер на податоци.

6. Контрола и распоредување на пристапот. Некои В/И уреди може да бидат пристапувани од повеќе корисници истовремено. На пример хард дискот претставува делив уред каде може да има повеќе корисници што истовремено читаат или запишуваат податоци, од друга страна пак уредите за магнетните ленти не се деливи. А уредите како што се печатари не може да се користат истовремено по случаен избор на процесите. Оперативниот систем треба да биде способен за да управува и со деливи и со посветени уреди (*англ. dedicated devices*) за да се избегнат проблемите кои може да се појават со нивното конкурентно користење.

8.3.1. Програмиран В/И

Постојат 3 начини на кои може да се изведат В/И операции. Наједноставниот е да се препушти на процесорот да ја врши сета работа околу изведувањето на В/И трансфер на податоци. Овој начин се нарекува програмиран В/И (*Programmed I/O - PIO*). Ако земеме на пример, еден процес кој сака да ги печати знаците „ABCDEFGH“ на печатар. Процесот првин ја создава низата од карактери во баферот во корисничкиот простор како што е прикажано на сликата. Потоа процесот го зазема печатарот преку издавање на системски повик за отворање (*open*). Ако печатарот се користи од друг процес, повикот нема да успее, ќе се врати порака за грешка или ќе се блокира се додека печатарот не стане слободен. По заземањето на печатарот, корисничкиот процес издава системски повик за печатење на низата знаци.



Слика 8.5: Пример за програмиран В/И трансфер

Потоа, оперативниот систем го копира баферот со низата на знаци во јадрото, каде што полесно се пристапува – јадрото не мора да ја менува мемориската мапа за да пристапува во корисничкиот простор. Потоа се проверува да ли печатарот моментно е слободен, ако е оперативниот систем го копира првиот карактер кон податочниот регистар на печатарот (во овој пример се користи В/И мапиран меморија). Со тоа се активира принтерот и се појавува карактерот, по што системот го одбележува наредниот карактер „B“ за печатење. Штом ќе го ископира првиот знак во печатарот, оперативниот систем проверува дали печатарот е слободен за наредниот карактер. Штом контролерот на печатарот ќе го процесира карактерот, покажува дека е достапен преку поставување на вредност во статусниот регистар. Од оваа точка оперативниот систем чека печатарот пак да стане достапен и процедурата пак се повторува за секој карактер посебно се додека не се печати целата низа по што контролата се враќа кон корисничкиот процес. Програмскиот код кој одговара на програмиран В/И е даден на примерот, каде е евидентно дека процесорот цело време го проверува статусот на печатарот, како што е веќе наведено, ваквото однесување се нарекува работно чекање или прозивка на В/И уредот.

```
copy_from_user(buffer, p, count);
for(i=0;i<count;i++){
    while(*printer_status_reg != READY);
    *printer_data_register=p[i];
}
return_to_user();
```

Слика 8.6: Печатење на низа на знаци (string) на принтерот со користење на програмирани В/И

Програмираниот В/И е едноставен но има недостатоци заради врзување на процесорот сè додека не се заврши В/И операцијата. Ова е неприфатливо и потребен е друг концепт за програмирање на В/И операции.

8.3.2. В/И водени од прекини

Ако го разгледаме примерот кога печатарот не ги баферира карактерите, туку ги печати по редоследот на пристигнувањето и ако секој карактер се печати за 10 милисекунди, тоа значи дека процесорот треба да чека исто време додека карактерот не се испечати и не му биде дозволено да запише друг во податочниот регистар на контролерот. А тоа време е доволно долго процесорот да изврши смена на контекст и да во меѓувреме извршува некој друг процес.

```
copy_from_user(buffer,p,count);
enable_interrupts();
while(*printer_status_reg!=READY);
*printer_data_register=p[0];
scheduler();
```

(а)

```
if(count==0){
  unblock_user();
}else{
  *printer_data_register=p[0];
  count=count-1;
  i=i+1;
}
acknowledge_interrupt();
return_from_interrupt();
```

(б)

Слика 8.7: Печатење на низа на знаци (string) на принтерот со interrupt водени В/И, а) кодот се извршува кога е направен системскиот повик за печатење, б) interrupt сервисна процедура

Користењето на механизмот на прекините овозможува процесорот да не губи време чекајќи на бавна В/И операција. Кога ќе се издаде системски повик за печатење на низата, таа се копира во бафер во јадрото и првиот карактер се испраќа на печатарот штом ќе стане слободен, исто како и во претходниот случај. Во тој момент распоредувачот го блокира процесот и врши измена на контекстот се до печатењето на целата низа. Кодот е даден на Слика 8.7.

Кога печатарот го испечатил карактерот и е подготвен да прифати друг издава сигнал за прекин. Се прекинува извршувањето на тековниот процеси и се зачувува состојбата. Потоа се извршува сервисната рутина на печатарот - Слика 8.7 б). Ако не постојат повеќе карактери за печатење, сервисната процедура го деблокира корисниот, инаку го дава следниот карактер, го потврдува прекинот и продолжува со извршувањето на процесот од местото каде бил претходно прекинат.

8.3.3. В/И со користење на DMA

Очигледен недостаток на претходниот концепт е што прекин се јавува за секој карактер. Изведувањето на прекинот одзема време и се троши процесорот за изведување на замена на контекстот и чување на состојбата. Друго подобро решение е да се користи DMA, каде што за зададениот пример DMA контролерот би ги испраќал карактерите еден по еден кон контролерот на печатарот без повикување на процесорот. Кодот за овој пример е даден на сликата.

```
copy_from_user(buffer,p,count);
set_up_DMA_controller();
scheduler();
```

(а)

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

(б)

Слика 8.8: Печатење на низата со користење на DMA, кодот се извршува кога е направен системски повик за печатење interrupt сервисна процедура

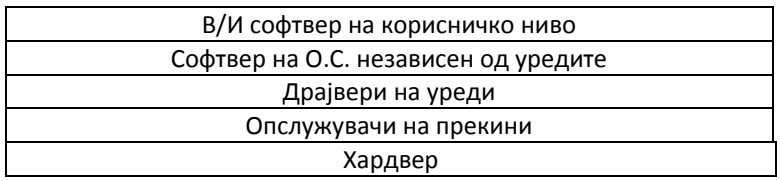
На овој начин се намалува бројот на прекините од еден по карактер со еден по испечатена содржина на баферот. Од друга страна DMA контролерот обично е побавен од процесорот и ако не е во состојба да го управува В/И уредот согласно неговата брзина, може да се искористат и некои од претходно предложени варијанти.

8.4. Структурирање на В/И софтвер

Една од наведените цели при дизајнирање на оперативен систем е формирање на униформен интерфејс кон апликациите и корисниците. Проблемот на униформноста на уредите се решава со методи на апстракција и повеќе-слоен софтвер во рамките на В/И подсистем. Типично софтверот е поделен на 4 слоја, каде што секој од слоевите има прецизно дефинирана функција и интерфејс кон соседните слоеви.

Функционалноста и интерфејсите се разликуваат од систем до систем но генерално се категоризираат во четири нивоа на В/И софтверот (од понизок кон повисок).

1. Водачи со прекини
2. Драјвери на уреди
3. Уредски-независен ОС софтвер
4. Софтвер за корисничко ниво



Слика 8.9: Слоеви на В/И софтверот

8.4.1. Водачи на прекини (Interrupt Handlers)

Механизмите за изведување на прекините треба да бидат скриени на најниското ниво, најблиску до хардверот. Најдобар начин за прикривање на прекините е секој процес (драјверот) што стартува В/И операција да се блокира (*DOWN, WAIT*), сè додека В/И не се комплетира и не се иницира прекин.

Кога ќе се случи прекинот, сервисната процедура го опслужува прекинот и по завршувањето го деблокира процесот (*UP, SIGNAL*) и тој продолжува со работа. Овој модел најдобро работи кога драјверите се структурирани како процеси во јадрото со состојби, стек и програмски бројач.

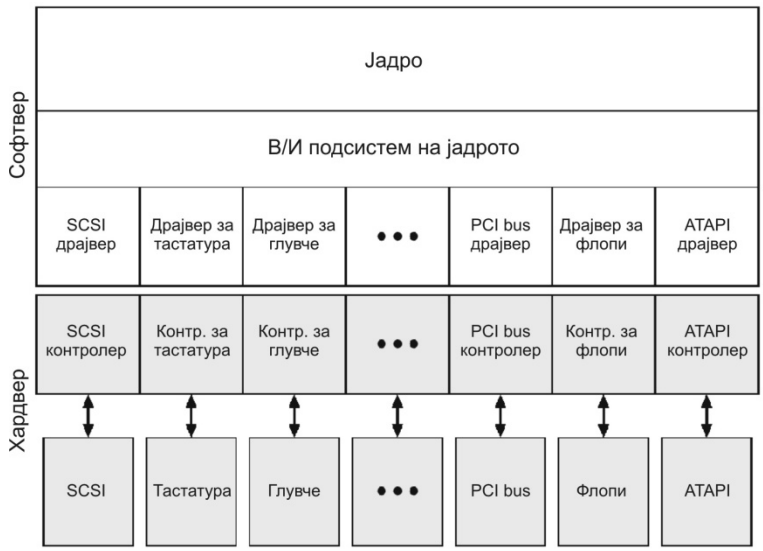
Во реалноста процесирањето на прекините не е едноставно и дадени се чекорите кои треба да бидат изведени во софтверот по завршувањето на хардверски прекин, при што треба а се забележи дека деталите и редоследот на чекорите може да се разликуваат од систем до систем:

1. Снимање на регистрите кои не се снимени од интерапт хардверот.
2. Се поставува контекст за сервисната процедурата за прекинот.
3. Се поставува стекот за сервисната процедурата за прекинот.
4. Се издава потврда (*ACK*) за контролерот на прекини, ако нема централизиран контролер на прекини да се овозможат повторно прекините (*англ. Reenable Interrupts*).
5. Да се копираат регистрите од кои се снимени во процесната табела.
6. Извршување на сервисната процедура со очитување на вредностите од регистрите на контролерот на прекините.
7. Се бира кој процес понатаму ќе се извршува.
8. Се поставува контекстот за да се изврши следниот процес.

9. Се вчитуваат регистрите за новиот процес.
10. Се стартува извршувањето на новиот процес.

8.4.2. Драјвери на уреди (Device Drivers)

Уредите одист вид, како што се дисковите се групираат во општи класи на уреди. Секоја класа се имплементира преку стандардно множество на функции, односно интерфејсот. Разликите кои постојат помеѓу уредите од иста класа се сокриени во специјалните модули во јадрото кои се нарекуваат управувачки програми или драјвери (*англ. device drivers*). Драјвери постојат за секој уред но кон корисникот се претставени како униформен интерфејс и се напишани од страна на самиот производител на уредот. Драјверите на уредите се единствен дел од оперативниот систем на кој му се познати карактеристиките и хардверските детали на уредите, како на пример што се: бројот на регистри на диск контролерот, нивната намена, организацијата на податоците на дискот во форма на сектори, траки, цилиндари и други.



Слика 8.10: Архитектура на драјвери на уреди

За да се вклопи кодот на одреден драјвер во состав на оперативниот систем, треба да постои архитектура која ќе овозможи таква инсталација. Тоа значи добро дефиниран модел на користење на драјверот и неговата интеракција со останатите делови од оперативниот систем.

Драјверите на уредите имаат неколку функции, најочигледната е да се прифатат апстрактните барања за читање и запишување од страна на слојот на софтверот независен од уреди (над слојот на драјверите). Тие задаваат команди на контролерите на уредите и проверуваат дали исправно се

изведени. Покрај тоа драјверот треба да изврши иницијализација на уредите, управувањето со потребите за напојувањето и да ги бележи настаните поврзани со него. Типичен драјвер започнува со проверка на влезните параметри дали се валидни, во случај да не се, се враќа порака за грешка. Ако се валидни, тогаш се врши преведување од апстрактни во конкретни операции. На пример за дискот тоа би било конверзија на логичкиот блок на дискот во број на глава, трака, сектор и бројот на цилиндарите на геометријата на дискот.

Потоа драјверот проверува дали уредот е достапен, ако е зафатен барањето се сместува во редица на чекање за подоцнежно процесирање. Ако уредот е достапен, се испитува хардверскиот статус за да се види дали може да се опслужи барањето. Може да биде неопходно претходно уредот да се подготви, како на пример кај дискот да се задвижи моторот и да се постигне работна состојба. Драјверот врз основа на барањето од повисоките слоеви, ја одредува секвенцата на команди кои треба да се извршат и ги запишува во регистрите на контролерот. Се проверува дали контролерот ја прифатил командата и е подготвен да ја прифати и следната.

По издавањето на командите може да се случат две ситуации. Во повеќето случаи драјверот на уредите чека додека контролерот ја изврши зададената задача и се блокира сè до моментот на издавање на прекин од контролерот. Во друг случај, операцијата е краткотрајна и се извршува без задржување така што драјверот нема потреба да се блокира. И во двата случаи, драјверот треба да изврши проверка за појава на грешки и ако сè е во ред резултатите ги предава на слојот на софтверот независен од уредите (*англ. device independent software*). И на крајот, може да врати и статусна информација за грешка, да избере други барање кое веќе чека или се блокира и чека на пристигнување на ново барање.

На драјверите не им е дозволено да издаваат системски повици и обично вршат интеракција со останатите делови од оперативниот систем, со повикување на одредени процедури во јадрото, како што се за управување со меморијата, тајмерите, DMA контролерот и други.

8.4.3. Уредски-независен софтвер (Device Independent I/O Software)

Одреден дел од В/И софтверот не зависи од карактеристиките на уредите. Границата меѓу слоевите 2 и 3, драјверите на уредите и независниот софтвер зависи од самиот систем, некои функции можат да бидат изведени и во двата слоја заради ефикасност или други причини. Основни функции на уредски-независниот софтвер се:

- изведување В/И операции заеднички за сите уреди;
- обезбедување униформен интерфејс кон софтверот на корисничко ниво;

- баферирање;
- пријава на грешките;
- заземање и ослободување на посветените уреди:
- обезбедување на големината на блокот независно од уредот.

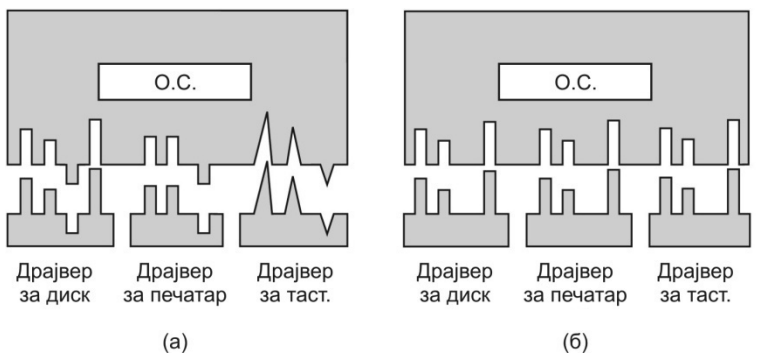
Основната функција на софтверот независен од уредите е да изведува В/И функции кои се заеднички за сите уреди и да обезбеди униформен интерфејс кон софтверот од корисничкиот слој.

Униформиран интерфејс за драјверите на уредите

Главен проблем во еден оперативен систем е како да се направи сите В/И уреди и драјверите да изгледаат повеќе или помалку исто. Ако дисковите, печатарите, тастатурите и т.н. сите се поврзани со интерфејси различни начини, секогаш кога ќе се приклучи нов уред, оперативниот систем ќе треба да се прилагоди за новиот уред.

Еден аспект на овој проблем е интерфејсот помеѓу драјверите за уредите и остатокот од оперативниот систем. На Слика 8.11 а), е прикажана ситуацијата во која секој драјвер за уред има различен интерфејс кон оперативниот систем. Ова значи дека функциите на драјверите достапни на системот се разликуваат од драјвер до драјвер. Тоа може исто да значи дека функциите на јадрото што му се потребни на драјверот исто така се разликуваат од драјвер до драјвер. Заедно гледано тоа значи дека поставување на секој драјвер во оперативниот систем би барал поголем програмерски ангажман што е непрактично.

Спротивно на оваа, на Слика 8.11 б) е прикажан различен пристап каде што сите драјвери имаат ист интерфејс. Многу полесно да се вклучи нов драјвер, доколку неговите интерфејс е соодветен на интерфејсот на драјверите во системот.



Слика 8. 11: а) Без стандарден драјвер на интерфејс,
б) со стандарден драјвер на интерфејс

Тоа исто така значи дека програмерите на драјвери знаат што се очекува од нив (на пр. кои функции тие треба да ги обезбедат и кои функции на јадрото тие можат да ги повикаат). Во пракса, не сите уреди се апсолутно идентични, но обично има само мал број на типови на уреди и дури овие ако се посматраат воопштено се речиси исти.

Друг аспект од постоење на униформиран интерфејс е именувањето на В/И. Софтверот независен од уредите се грижи за мапирањето на симболичните имиња на уредите соодветниот драјвер. На пример, кај UNIX оперативниот систем, име на уред како *dev/disk0* е единствена ознака на i-јазолот за специјална датотека и овој i-јазол го содржи главниот број на уредот (*major device number*) кој се користи за да се лоцира бараниот драјвер. Во датотечниот систем, i-јазолот на специјалната датотека исто така го содржи и минорниот број на уредот (*minor device number*), кој се пренесува како параметар на драјверот, со цел за да се специфицира единицата која треба да биде напишана или прочитана. Сите уреди имаат главни и минорни броеви и сите драјвери се пристапуваат со користењето на главниот број на уредот со кој се избира соодветниот драјвер. Блиску поврзано до именувањето е заштитата. Како системот ќе ги заштити корисниците од пристапувањето кон уреди кон кои не им е дозволен пристап?

Во двата оперативни системи UNIX и Windows, уредите се појавуваат во системот како именувани објекти, што значи дека вообичаените заштитни механизми за датотеките исто така може да се применат и на В/И уредите. На пример, UNIX, специјални датотеки придружени со В/И уреди се заштитени со битовите *rwx,* а систем-администраторот задава дозволи за достап до одредените уреди.

Баферирање

Користењето на привремен мемориски простор – бафер може да претставува проблем и за блок и за карактер базирани уреди. Баферот функционира по принципот произведувач-потрошувач и служи за чување на привремени податоци при пренос помеѓу два уреди и апликации. Се користи заради следниве причини:

- Ускладување на различни брзини помеѓу двата ентитети. Заради големата разлика во брзини, В/И операциите може да се преклопат и баферот треба да се подели на два дела ко наизменично работат и како произведувач и како потрошувач. Овој систем се нарекува двојно баферирање (се користи при кеширањето на дисковите).
- Прилагодување на различни големини на трансфери на податоци. Баферите често се користат во мрежно окружување, каде што податоците се делат на помали пакети. Во баферот се собираат пакетите по недетерминистички начин (во зависност од состојбата на

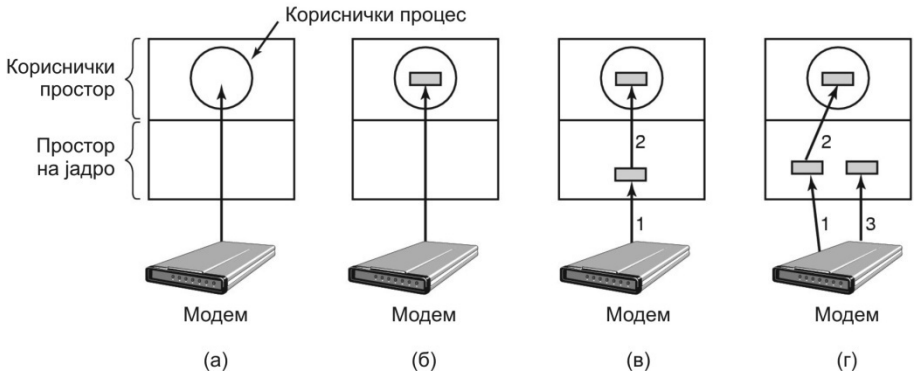
мрежата и евентуалните грешки) по што се врши реконструкцијата на оригиналната порака.

- Одржување на семантиката на копирање. Кориснички процес може да издаде системски повик за запишување, односно испраќање на податоци кон некој уред. Доколку повикот е асинхрон или неблокирачки, процесот има можност да ги смени податоците при испраќањето кон уредот. Значи процесот може да запише нешто друго наместо податоците што иницијално требало да бидат запишани. За да се избегне оваа ситуација, т.е. за да се задржи семантиката на копирање, оперативниот систем во јадрото дефинира бафери во кои се врши копирање на корисничкиот бафер, па потоа се префрла контролата на корисничкиот процес.

Една од стратегиите која може да се користи при користењето на баферите, е кога при пристигнување на карактерите кои треба да се прочитаат од уредот, процесот издава повик за читање и се блокира чекајќи на карактер. Секој карактер предизвикува појава на прекин и сервисната процедура го предава карактерот кон процесот и го деблокира процесот. По читањето на карактерот, процесот го обработува и пак се блокира. Проблемот со овој пристап се состои во тоа што процесот секој пат се стартува при пристигнувањето на секој карактер, а често и краткотрајно извршување на процесите не е ефикасно искористување на процесорското време (се губи при измена на контекстите при блокирањето на процесот - Слика 8.12 а).

Може да се направи подобрување преку користење на *n*-знаковен бафер во корисничкиот просторот (б). Сервисната процедура ги сместува карактерите во баферот сè додека не се наполни, а потоа го буди блокираниот процес. Овој пристап е многу поефикасен од претходниот, но има еден недостаток кој се појавува во случај кога мемориската страница каде што се наоѓа баферот се заменува во моментот кога пристигнува карактерот. Баферот може да биде заклучен во главната меморија, но со оглед на поголемиот број на процеси оваа решение не е оптимално.

Може да се создаде бафер во јадрото и карактерите да се сместуваат во него (в), кога тој ќе се наполни неговата содржина се префрла во баферот во корисничкиот простор. Иако е поефикасна и оваа стратегија има свој недостаток, а тоа е проблемот каде да се смести карактерот ако баферот е полн а се чека на донесување на мемориската страница која го содржи корисничкиот бафер. Може да се користи дополнителен бафер во јадрото кој ќе се користи за прифаќање на карактерите, додека содржината на другиот бафер се копира и обратно, ова се нарекува двојно баферирање (*англ. double buffering*).



Слика 8.12: а) Небафериран влез, б) баферирање во корисничкиот простор, в) баферирање во јадрото пратено со копирање во корисничкиот простор, г) двојно баферирање во јадрото

Управување со грешки

Уредите и В/И операциите можат да откажат од повеќе причини и тоа привремени (преоптоваруавњето на мрежата) или трајни (расипувањето на уредот). Во првиот случај оперативниот систем може да го повтори барањето за операцијата), а во вториот нема можност за поправка и единствено нешто што може да се стори грешката да биде пријавена. За тоа се користат одредени системски повици преку кои се опишува статусот на операцијата односно деталите за евентуално настанатата грешка.

Грешките може да бидат предизвикани од грешки при програмирањето како што се на пример, обид за запишување на уред кој може само да се чита (тастатура), користење на невалидна бафер адреса, или користење на невалиден уред и В/И грешки кои се појавуваат заради откажување на самиот уред, како на пример обид за запишување на оштетен диск блок.

При појава на грешка, самиот драјвер може да се обиде да ја поправи ако тоа е можно или да ги извести повисоките нивоа за настанатата ситуација, па тие да извршат управување на настанот. На кој начин ќе се справи В/И софтверот со појава на грешки, зависи во голема мерка од природата и причината за настанување на грешката. На пример, ако грешката се случила при читање корисничка датотека, се дава информација за појава на грешка при читање преку презентирање на порака на корисникот, каде што тој би имал избор да го повтори барањето преку некој дијалог прозорец или да се откаже. Од друга страна, ако грешката е поврзана со структурата на системот како што е на пример грешка при читање бит-мапа за слободен простор во датотечниот систем, оперативниот систем нема друг избор освен да испечати порака за грешка и да го терминира процесот.

Заземање и ослободување на посветени уреди

Некои уреди како што се оптичките снимачи, може да бидат користени само од еден процес. Оперативниот систем треба да ги обработи барањата од процесите за заземање на одреден ресурс и да ги прифати или одбие во зависност на достапноста на уредот. Еден начин би бил е процесите да извршуваат повик за отворање на специјалните датотеки на уредот (*open*), ако уредот не е достапен операцијата нема да биде успешна. Друг пристап е постоење на специјални механизми за барање и ослободување на посветените уреди. Повикот за користење на одреден уред го блокира повикувачкиот процес наместо да објави грешка ако уредот не е достапен. Блокираните процеси се стават во редица на чекање за уредот и порано или подоцна, кога уредот ќе стане достапен првиот процес во редицата го зазема и продолжува со извршувањето.

Големина на блокот независно од уред

Различни дискови, може да имаат различна големина на секторите. Обврска на слојот на софтверот независен од уредите да ги прикрие овие детали и да обезбеди униформна големина на блокот за повисоките нивоа. Со ова повисоките ниво работат со апстрактни уреди кои користат иста големина на блокот. Слично, одредени карактер уреди ги испорачуваат своите податоци еден бајт едновремено, додека другите пак може да испорачуваат и поголеми податочни единици и овие разлики не треба да се презентираат на програмерите на апликациите или корисниците.

8.4.4. Софтвер за корисничко ниво (User-Space I/O Softvare)

Иако В/И софтвер зазема поголем дел од кодот на оперативниот систем, мал дел од него се состои во форма на библиотеки, кои се врзуваат со корисничките програми (пр. В/И системски повици). А дури и цели програми се стартуваат и извршуваат надвор од јадрото. Системските повици, како и В/И повиците се изведуваат преку библиотечните процедури. Кога С програма го содржи повикот:

```
count=write(fd, buffer, nbytes)
```

библиотечната процедура *write* се поврзува со програмата и се содржи во бинарната презентација на програмата во меморијата при извршување на процесот. Множеството на сите овие библиотечни процедури се дел од В/И подсистемот. Форматирањето влезните и излезни податоци, исто така се врши од страна на библиотечни процедури. На пример, процедурата *printf***,** која зема форматиран стринг и некои променливи како влезни, создава ASCII стринг и потоа ја повикува процедурата *write* за запишување на излез.

```
printf("The square of %3d is %6d\n", i, i*i);
```

слична процедура за влез е scanf која го чита влезот и содржината ја сместува во одредена променлива во низа форматирана на сличен начин и синтакса како и *printf*. Стандардна В/И библиотека содржи поголем број процедури, кои ги вклучуваат В/И операциите и работат како дел од корисничките програми.

Систем за спулирање

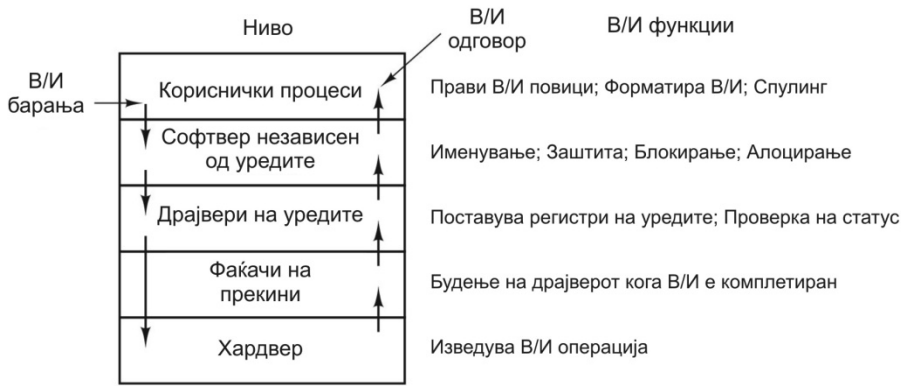
Не секој В/И кориснички софтвер содржи библиотечни процедури. Друга значајна категорија е систем за „спулирање“ (*англ. spooling system*). Спулер претставува бафер кој ги чува податоците наменети за некој неделив уред (пример, печатар) и претставува начин за справување со користење на овој вид на уредите кај системи со мултипрограмирање. Спулерот овозможува истовремено пристапување кон неделив уред на следниов начин, процесите ги запишуваат податоците наменети за уредот на дискот, а оперативниот систем ги опслужува барањата еден по еден. На пример, секој процес што сака да го користи принтерот, го поставува своето барање во спулерот, а процесот кој во позадината управува со него врши печатење на барањата по редоследот на пристигнувањето.

На пример, кај линиски печатач ако не се користи спулирањето, секој процес, без ограничување отвора специјална карактер датотека и со тоа го зазема печатачот, процесот може да седи без работа со часови при што друг процес до печатачот немаат воопшто пристап. Ако се користи спулирање, се креира процес демон и специјален спулинг директориум, процес ја запишува датотеката за печатење во спулинг директориумот, демонот одбира која од тамошните датотеки е на ред за печатење. Користењето на спулерот овозможува одредени предности како што се:

- елиминиран е проблемот на непотребно чекање на процесот, кој по користењето на спулерот, се ослободува за извршување на други задачи.
- неделивите уреди се користат привидно како деливи, со што се овозможува користење од страна на повеќе процеси истовремено.

8.5. Претставување на В/И систем

В/И системот може да биде претставен како на Слика 8.13, каде што се прикажани слоевите и основните функции на секој од нив. Ако се започне од доле, слоевите се хардверот, водачи на прекини, драјвери на уредите, софтвер независен од уредите и корисничките процеси.



Слика 8.13: Текот на контролата при барање и одговарање за В/И

Со стрелките е прикажан текот на контролата. Кога корисничката програма ќе се обиде да чита блок од датотека на пример, кон оперативниот систем се упатува системски повик, слојот независен од уред првин гледа дали блокот се наоѓа во својот кеш. Ако блокот не е внатре, се упатува системски повик кон драјверот на уредот за да испрати барање кон хардверот да го преземе блокот од дискот. Процесот е блокиран се додека не се заврши диск операцијата. Кога дискот ќе заврши, се генерира прекин. Водачот на прекини го открива уредот кој го испратил прекинот, го чита статусот на уредот и го буди блокираниот процес за да го заврши В/И барањето и да овозможи продолжување на работата на корисничкиот процес.

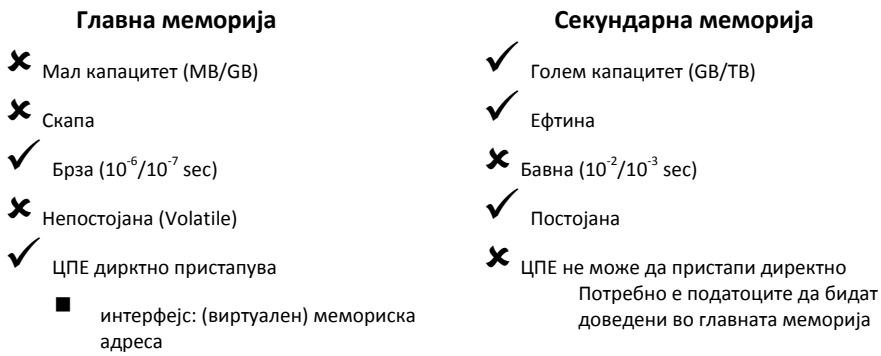
9. Систем на датотеки

9.1. Поим за систем на датотеки

Сите компјутерски апликации имаат потреба да чуваат и складираат податоци. Додека се извршува процесот, податоците се складираат во адресниот простор. Капацитетот на меморискиот склад е ограничен од големината на адресниот простор од виртуелната меморија. Втор проблем со складирањето на податоците во адресниот простор на процесот е што кога процесот ќе се терминира, податоците се губат. И третиот проблем се состои во тоа што процесите имаат потреба истовремено да пристапуваат кон делови од податоците. Ова се трите суштествени барања кои треба да ги исполни еден за долгорочен мемориски склад:

1. Да складира големо количество на информации;
2. Информациите да го преживеат завршувањето на процесот;
3. Повеќе процеси да можат да пристапуваат конкурентно кон податоците.

За имплементација на долгорочен мемориски склад се користат податочни структури наречени датотеки и директориуми, В/И уреди како што се дискови, магнетни ленти или оптички дискови. На Слика 9.2 е прикажана мемориската хиерархија, на врвот се наоѓа главната меморија, која се карактеризира со брз пристап и мал капацитет и непостојаност на податоците по терминација на процесот. На второ ниво се наоѓа секундарната меморија дисковите, кои имаат поголем капацитет за складирање на податоците, за неколку реда на големина побавен пристап и способност за зачувување на податоците. На најниското ниво се наоѓаат така наречени „*off-line*“ мемориски уреди за складирање, каде што пристапот е најбавен, но имаат голем капацитет и обично податоците се чуваат за архивирање.



Слика 9.1: Споредба на главна со секундарна меморија

Датотеките се управувани од страна на оперативниот систем. Како се структурирани, именувани, користени, пристапувани и заштитувани и имплементирани е задача дел од оперативниот систем познат како систем на датотеки (*англ. file system*).

Системот на датотеки е ниво помеѓу секундарната меморија и апликацијата и ја опишува секундарната меморија како множество на перзистентни објекти со единствени имиња – наречени датотеки (*англ. files*). Покрај тоа обезбедува механизми за пресликување на податоците помеѓу секундарната меморија и главната меморија.

Слика 9.2: Хиерархија на видовите на мемории

Најважниот аспект во однос на корисниците е начинот на претставување на датотеките, што се содржи во нив, кои операции се дозволени над нив, како изгледа стеблото од директориуми како и нивната заштитата. Еден пример на интерфејс кон корисникот би бил командите за пристап кон датотеките и нивната синтакса:

- *READ*: Го превзема специфицираниот податочен дел (*англ. chunk*) од датотеката директно во виртуелниот адресен простор на процесот
- *WRITE*: Го запишува специфицираниот податочен дел (*англ. chunk*) од виртуелниот адресен простор на дискот

Тоа се услугите кои ги дава системот на датотеки на корисникот, а од друга страна самата имплементација на системот на датотеки, како на пример се претставени податоците на дискот на најниско ниво (бит по бит), како се решаваат и другите проблеми како што се автоматско зголемување на датотеките, појава на дупки и постоење на отворени датотеки и т.н. не е од големо значење за корисниците освен на дизајнерите на оперативните системи.

9.2. Податочни целини: датотеки

Датотеките се именувани целини на меморирани информации на дискот. Тие обезбедуваат начин за складирање на информациите и нивно повторно повикување од дискот. Ова треба да биде потполно транспарентно за

корисникот, т.е. деталите како се организирани податоците треба да бидат скриени од него. Кога еден процес ќе создаде датотека и дава име, а по завршувањето на процесот датотеката продолжува да постои и може да биде пристапена од други процеси преку повикување на нејзиното име. Повеќето оперативни системи подржуваат дво-делни имиња на датотеки одвоени со точка, како на пример datoteka.txt. Делот кој следува по точката е нарекува екстензија на датотеката и дава дополнителна информација за типот на датотеката. На Слика 9.3 се дадени некои од најкористените екстензии на датотеки кои се користат во оперативните системи од фамилијата DOS/Windows. Кај UNIX/Linux се применува метода на одредување на типот на датотеката преку карактеристични записи во самата датотека – магичните броеви кои се наоѓаат на самиот почеток на датотеката. Оперативниот систем UNIX во командниот режим се води исклучиво според овие записи, додека Linux кој работи во графички режим се води од екстензиите на датотеките за придружување на соодветната апликација кон нив. И самите корисници може рачно да извршат придружување на екстензија кон одредена апликација која ќе се повикува при активирањето на датотеката.

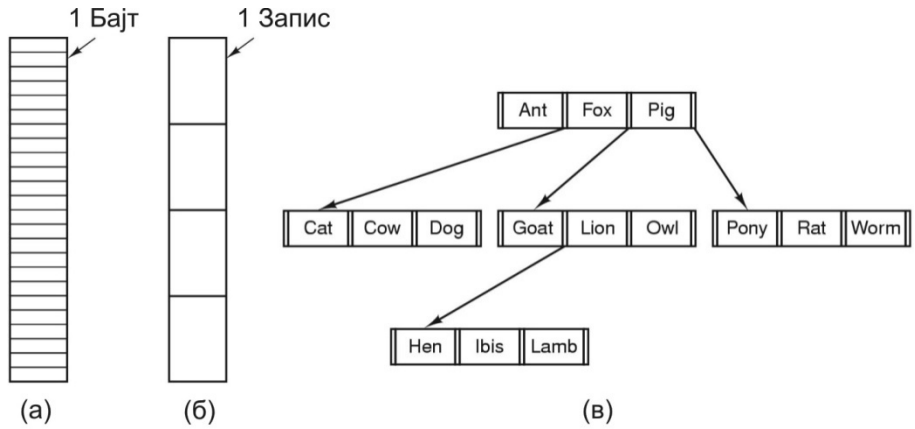
Наставка	Значење
file.bak	Архива
file.c	С програма
file.gif	GIF слика
file.hlp	Помошна датотека
file.html	Веб страна
file.jpg	JPEG слика
file.mp3	Музика
file.mpg	Видео
file.o	Објектна датотека (излез од компајлер)
file.pdf	Документна датотека
file.ps	PostScript датотека
file.tex	Датотека за TEX програмата
file.txt	Текстуална датотека
file.zip	Компресирана датотека

Слика 9.3: Екстензии на имињата на датотеките и нивното значење

9.2.1. Структура на датотеки

Датотеките може да бидат структурирани на неколку начини. Три можности се прикажани на Слика 9.4. Неструктурирани, секвенцијални датотеки кои се претставуваат како низа од бајти, каде кон податоците се пристапуваат со користење на вредност на поместување (*англ. offset)* во однос на почетокот на датотеката. И Windows и UNIX го користат овој пристап. Друг пристап претставува организирање на датотеките како низа од записи со одредена

должина. Идејата се состои во тоа што операцијата за читање врши очитување на еден запис и соодветно операцијата за запишување се однесува на еден запис.



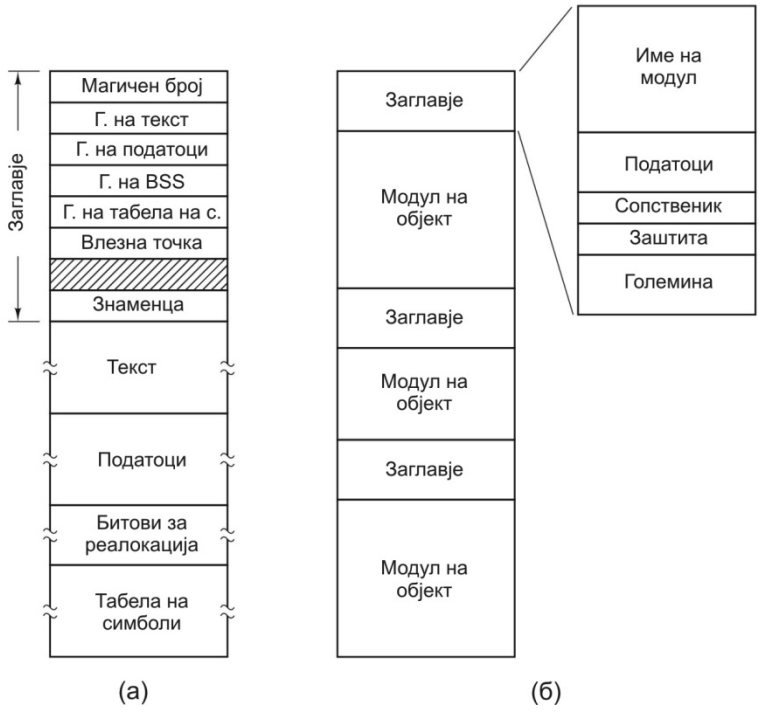
Слика 9.4: Структура на датотеки а) бајт, б) запис, в) стебло на записи

И третиот начин е претставување на содржината на датотеките во облик на стебло на записи, кои не мора задолжително да имаат иста големина, но имаат поле-клуч на точно одредена позиција во записот. Стеблото е сортирано според содржината на полето на клучот и овозможува брзо пребарување за одреден клуч. Податоците се повикуваат според дадениот клуч, независно од нивната локација во датотеката, односно со користење на асоцијативен пристап, како на пример да се најдат сите записи за кои "Name=Иван".Кога се додаваат нови податоци во датотеката самиот оперативниот систем се грижи за нивното сместување.

Првиот начин е најкористен во практиката, додека вториот многу ретко се користи и претставува историско наследство од времето кога програмите биле сместувани на дупчени картички со фиксен број на карактери. Додека третиот начин се користи кај големите "*mainframe"* системи за комерцијална обработка на податоци.

9.2.2. Типови на датотеки

Многу оперативни системи подржуваат повеќе типови на датотеки. UNIX и Windows имаат регуларни датотеки, а UNIX покрај тоа има и карактер и блок специјални датотеки. Регуларни датотеки се оние кои содржат корисничка информација. Директориумите претставуваат системски датотеки кои се користат за одржување на структурата на системот на датотеки. Карактер специјални датотеки се поврзани со влез/излез и се користат за моделирање на В/И уреди како што се терминали, печатачи и мрежи.



Слика 9.5: (а) Извршна датотека, (б) архивска

Блок специјални датотеки се користат за моделирање на дискови. Регуларните датотеки преставуваат бинарни или ASCII датотеки. ASCII датотеки се состојат од линии на текст, каде што секоја линија на текст завршува со карактер CR (*carriage return*) или LF (*line feed*). Предноста на ASCII датотеки е што може да се прегледуваат и уредуваат во било кој текст уредувач. Другите датотеки се бинарни и имаат одредена интерна структура која е позната на апликациите. На пример, на Слика 9.5 може да се види извршна бинарна датотека (*англ. executable binary file*) преземена од верзија на UNIX. Иако датотеката е само низа од бајти, оперативниот систем ќе ја изврши, односно ќе создаде процес само ако датотеката е во соодветен формат, да ги има петте секции: заглавје, текст, податоци, битови за реалокација и табелата на симболи. Заглавјето започнува со таканаречен магичен број, со кој датотеката се идентификува како извршна (со што се оневозможува случајно стартување на датотека која не е извршна). Потоа следуваат големината на различните делови од датотеката, адресата на која започнува извршувањето и одредени битови – знаменца. По заглавјето следуваат, текстот и податоците на самата програма кои се вчитуваат во меморијата и се преместуваат – реалоцираат со користење на битовите за реалокација, а табелата на симболи се користи за дебагирање.

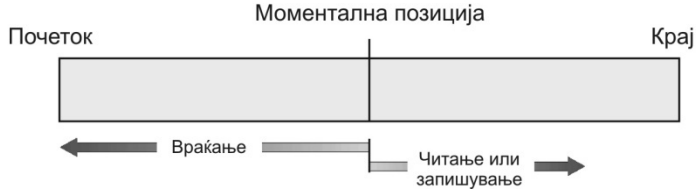
Другиот пример е бинарна датотека која е архивска, исто така од UNIX. Се состои од множество на библиотеки на процедури (модули) кои се

компајлирани, но не и поврзани. Секоја има заглавје со кое се означува нејзиното име, датата на креирање, сопственикот, заштитата и големината.

Можна е појава на проблеми кога корисникот ќе стори нешто неочекувано. На пример, еден програмер пишува С програма која е претставена со датотека со екстензија dat. Ако се даде на компајлирање, компајлерот може да ја одбие заради несоодветната екстензија.

9.2.3. Пристап кон датотеките

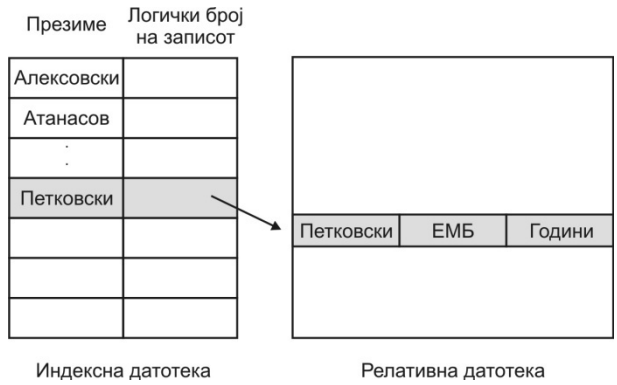
Постојат два главни начини за пристап кон содржините на датотеките: секвенцијален (*англ. sequential*) и случаен (*англ. random*) пристап. Кај секвенцијалниот пристап процесот може да ги чита сите бајти или записи од почетокот на датотеката по ред, но не може да прескокнува и да ги чита надвор од нивниот редослед, но може да се премота од почетокот. Нивното користење е посоодветно кога медиумот за складирање е магнетна лента, но не и диск.



Слика 9.6: Секвенцијален пристап кон датотеки

Кај дисковите може да се читаат бајтовите по произволен редослед или да се пристапува кон записите според клуч наместо по одредена локација. Датотеките чија што содржината може да се чита по било кој редослед се нарекуваат датотеки со случаен пристап. Тие се суштествени за многу апликации како што се на пример, податочни бази. Два методи се користат за да се одреди од која позиција да се започне читањето. Кај првата за секоја операција на читање се дава и позицијата од каде да се започне, додека кај другата се користи специјална операција за пребарување (*англ. seek*) да одбележување на тековната позиција, а потоа да се врши секвенцијално читање или пак прво читање па тогаш да се помести „*file*“ маркерот. Кај модерните оперативни системи секогаш се користат датотеки со случаен пристап, а кај постарите “mainframe”системи може да се определи начинот на пристап при креирањето на датотеката и според тоа да се користи соодветен мемориски склад.

На крајот може да се спомене и начинот на пристап со помош на индексни датотеки каде што на секоја датотека и е придружена и индексна датотека, уредена по некој критериум, каде што при читањето може многу бргу да најде соодветниот запис. Со запишувањето во датотеката се врши и обновување на податоците и во индексната датотека.



Слика 9.7: Индексни датотеки

9.2.4. Атрибути на датотеките

Секоја датотека најосновно се карактеризира со своето име и податоци што ги содржи, дополнително оперативните системи вршат придружување на други информации за секоја датотека, како на пример времето кога е создадена и нејзината големина. Овие податоци се атрибути на датотеката. Листата на атрибутите може да се разликува од систем до систем. На Слика 9.8 се дадени некои од можностите, но може да постојат и други.

Атрибут	Значење
Protection	Кој и како пристапува до датотеката
Password	Лозинка за пристап кон датотеката
Creator	ID на корисникот кој ја создал датотеката
Owner	Тековен сопственик
Read-only flag	0 за read/write; 1 само за read only
Hidden flag	0 за нормално; 1 за да не се листа
System flag	0 за нормални датотеки, 1 за системска
Archive flag	0 за веќе архивирана, 1 треба да се архивира
ASCII flag	0 за ASCII датотека, 1 за бинарна датотека
Random access flag	0 за секвенцијален пристап, 0 за случаен
Temporary flag	0 за нормално, 1 за бришење при излез на процесот
Lock flags	0 за незаклучена, било кој број за заклучена
Record length	Број на бајтите во записот
Key position	Поместувањето на клучот во секој запис
Key length	Број на бајтите во полето на клучот
Creation time	Дата и време кога е создадена датотеката
Time of last access	Дата и време кога последен пат е пристапена
Time of last change	Дата и време кога последен пат е изменета
Current size	Број на бајти во датотеката
Maximum size	Број на бајти до каде може да расне датотеката

Слика 9.8: Атрибути на датотеки

Првите четири атрибути се поврзани со заштитата на датотеката и покажуваат кој има дозвола да пристапи кон податоците и да ги користи. Кај некои системи, корисникот треба да внесе лозинка за да пристапи кон датотеката, во тој случај и лозинката преставува еден атрибут на датотеката. Знаменцата (*англ. flag*) се битови или кратки полиња кои контролираат или овозможуваат одредени карактеристики, како на пример криење на датотеките за да не се појават во листата на директориумот.

Архивскиот бит претставува знак дека датотеката треба да се сочува при следното backup-ување на системот. Големината за записите, позицијата на клучот и должината на клучот се користат само кај датотеките кои користат индексен пристап со помош на клучен збор. Временските атрибути ги чуваат информациите кога датотеката е создадена, пристапувана во скоро време или изменета во блиско минато. Тековната големина на датотеката се изразува во број на бајти од кои е составена, а максималната големина до која и е дозволено да расне.

9.2.5. Операции врз датотеките

Датотеките постојат за складирање на податоците и нивно подоцнежно читање. Различни системи обезбедуваат различни операции кои овозможуваат складирање и читање. Се дефинираат операциите кои може да се изведат над датотеките:

1. Креирај (*Create*). Се создава датотека без податоци. Намената на овој повик е да се додели простор на дискот за сместување на датотеката, да се запишат информации за датотеката во директориумот каде што ќе биде сместена и да се постават одредени атрибути.
2. Избриши (*Delete*). Кога датотеката веќе не е потребна, потребно е да се ослободи просторот на дискот. Датотеката не се брише физички од дискот туку само се отстранува контролниот блок на датотеката од директориумот и може со користење на специјален софтвер да се реставрира.
3. Отвори (*Open*). Пред да се користи датотеката, процесот првин треба да ја отвори. Намената на овој системски повик е процесот да ги преземе атрибутите и листата на адресите на дискот во главната меморија за побрз пристап.
4. Затвори (*Close*). Кога ќе се завршат сите пристапи во датотеката и адресите од дискот не се веќе потребни, датотеката се затвора заради ослободување на просторот на интерната табела. Многу системи го поддржуваат тоа преку ограничување на бројот на отворените датотеки по еден процес.

5. Читај (*Read*). Се читаат податоците од датотеката. Се читаат бајтовите од тековната позиција на покажувачот во датотеката. Повикувачот на операцијата треба да специфицира кое количество на податоци е потребно и мемориски бафер каде ќе се сместат.
6. Запиши (*Write*). Податоците се запишуваат во датотеката, на тековната позиција на покажувачот, ако тој покажува на крај од датотеката (*end of file*) големината ќе порасне. Ако позицијата е на средина од датотеката, податоците ќе бидат заменети и неповратно изгубени.
7. Приклучи (*Append*). Овој повик е ограничена форма од запишување. Податоците може да се додадат на крајот од датотеката.
8. Пребарување (*Seek*). Потребен е начин за да се изведе случаен пристап кон датотеката, за да се определи од каде да се земаат податоците. Се користи системски повик за пребарување со кој се преместува покажувачот на одредена позиција. По завршувањето на ова операција, може да се врши читање или запишување на таа позиција.
9. Преземање на атрибути (*Get Attributes*). Процесите имаат потреба за читање на атрибутите за да дознаат одредени информации за датотеката, како што е на пример времето на последната измена.
10. Поставување на атрибути (*Set Attributes*). Некои од атрибутите на датотеката може да се поставуваат од страна на корисниците и може да се менуваат по креирањето на датотеката. Пример за ова е поставување на битовите со кои се определуваат дозволите на пристап кон датотеката.
11. Преименување (*Rename*). Системски повик со кој се врши преименување на постоечка датотека.

Дадени се примери за изведбата на операциите над датотеките во состав на кориснички интерфејс кај оперативниот систем *UNIX:*

```
read(fd, buffer, count)- вчитува count следни бајти од
датотеката fd во buffer
write(fd, buffer, count)-запишува низа деф. со buffer во
датотеката fd
lseek(fd)-сетира покажувач кон бајт-позиција во датотека
дефинирана со fd
open(filename, mode)-отвора дат.
creat(filename)-креира датотека
link(filename)- прави врска кон дат.
unlink(filename)-брише врска кон датотека
```

Во продолжението е даден пример за програма под UNIX која врши копирање на една датотека од изворна во одредиште. Програмата има минимална функционалност и можности за известување за грешки, но дава пример како се

користат системските повици за работа со датотеки. Програмата *copyfile* се повикува преку командна линија:

```
copyfile abc xyz
```

при што се врши копирање на датотеката *abc* во датотека *xyz* која ако веќе постои се препишува, ако не тогаш се креира. Програмата се повикува со две имиња на датотеки како аргументи. Главната функција *main* првин ги проверува бројот на аргументите, ако бројот не е еднаков на 3 вклучувајќи ја и функцијата и двата аргументи, програмата излегува со код за грешка. Потоа се отвора првата датотека со системскиот повик *open* со поставена заштита само за читање (O_RDONLY), а другата се создава со наредбата *creat* и параметар OUTPUT_MODE, при што секоја датотека добива свој дескриптор *in-fd* и *out_fd*. Понатаму се врши истовремено читање и запишување од една во друга датотека со големина блок од 4096 бајти, сé додека вредноста на променливата *rd_count* која претставува број на успешно прочитани бајти не стане 0 во случај на крајот на датотеката или помала од 0 во случај на грешка.

```
/* File copy program. Error checking and reporting is minimal */

#include <sys/types.h>        /*include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[])     /*ANSI prototype*/

#define BUF_SIZE 4096                 /*use a buffer size of 4096*/
#define OUTUT_MODE 0700               /*protection bits for output file*/

int main (int argc, char *argv[])
{
        int in_fd, out_fd,rd_count, wt_count;
        char buffer[BUF_SIZE];

        if (argc!=3) exit (1);/*syntax error if argc is not 3 */

/* open the input file and create the output file */

in_fd = open(argv[1], O_REONLY);      /*open the source file*/
if (in_fd < 0) exit(2);               /*if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE); /*create the destination file*/
if (out_fd < 0) exit(3);              /*if it cannot be created, exit */

/*Copy loop*/
while (TRUE) {
        rd_count=read(in_fd, buffer, BUF_SIZE); /*read a block of data*/
if (rd_count <= 0) break;             /*if end of file or error, exit loop*/
        wt_count= write(out_fs, buffer, rd_count);   /*write data*/
        if(wt_count<=0) exit(4);      /*wt_count <=0 is an error*/
}
```

```
/*Close the files*/
        close(in_fd);
        close(out_fd);
        if(rd_count==0)                 /*no error on last read*/
                exit(0);
        else
                exit(5);                /*error on last read*/
}
```

Слика 9.9: Пример за програма за копирање на датотеки

Тоа уште значи дека датотеката треба да биде со големина на цел број пати од големината на блокот или последниот повик на за читање нема да се изврши. Системскиот повик *write* ја запишува вредноста од баферот (*buffer*) во излезната датотека *out_fd*. По завршувањето на јамката, двете датотеки се затвораат и програмата излегува со индикација дека командата успешно се завршила. Иако системските повици кај Windows се различни, структурата на командата за копирање на датотеки е во голема мерка слична.

9.3. Директориуми

За да се обезбеди следење на датотеките и нивното организирање, датотечните системи имаат директориуми, кои во одредени оперативни системи и самите претставуваат датотеки. Директориумите вршат групирање на датотеките заради организација и доделување на привилегии на пристап. Покрај тоа обезбедуваат пресликување од име кон датотека и можат да обезбедат дополнителни атрибути за датотеките. Различни се во однос на обичните датотеки, подржуваат и други видови на операции како што се *create, delete, list*. Спречуваат дуплирање на имиња, можат да бидат имплементирани како *hash* табела за поефикасно пребарување или со користење на балансирани стебла. Сите директориуми со своите датотеки го сочинуваат системоте на датотеки.

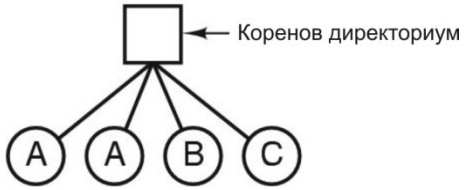
9.3.1. Логичка структура на директориумите

Во системот на датотеките директориумите може да се имплементираат на едно ниво, на две нивоа или повеќе нивоа во форма на стебло на директориуми. Најчестата структура е хиерархиската структура (стебло) која овозможува и поддршка за хиерархиски имиња на патеки (*англ. pathnames*).

Системи на директориуми со едно ниво

Наједноставната форма на директориумски сервис е да се има еден директориум кој ги содржи сите датотеки и обично се нарекува коренски директориум (*root*). Даден е пример на ваков тип на систем, каде што директориумот содржи 4 датотеки кои се одбележени со имињата на

сопствениците. Предноста на оваа шема е едноставноста и можноста датотеките лесно да се најдат, бидејќи се лоцирани само во еден директориум.

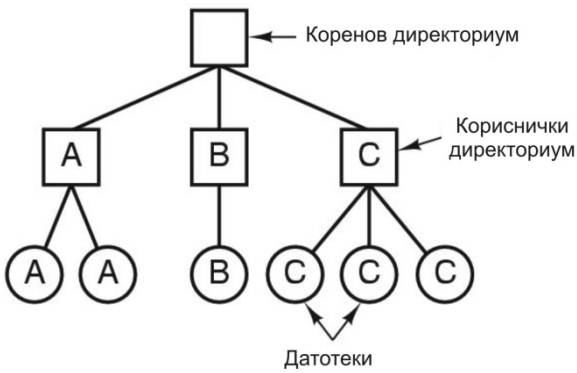


Слика 9.10: Пример за систем на директориуми со едно ниво содржи 4 датотеки сопственост на 3 различни корисници, A, B и C

Проблемот при користење на системи со повеќе корисници е именувањето на датотеките, каде што случајно може да се појават датотеки со исто име. Затоа овој вид на системи повеќе не се користи на повеќекориснички оперативни системи, а е погоден за користење во вградени (*англ. embedded*) системи.

Системи на директориуми со два нивоа

За да се избегнат конфликти предизвикани од различни корисници кои избираат исто име за сопствените датотеки, директориумите се организираат во повеќе нивоа. Едно решение е секој корисник да добие приватен директориум. Во тој случај нема да се појави судир во именувањето. На сликата е прикажан ваков систем, кој би можел да се користи на повеќе кориснички систем или сервер на датотеки на локална компјутерска мрежа.



Слика 9.11: Буквите ги обележуваат сопствениците на директориумите и датотеките

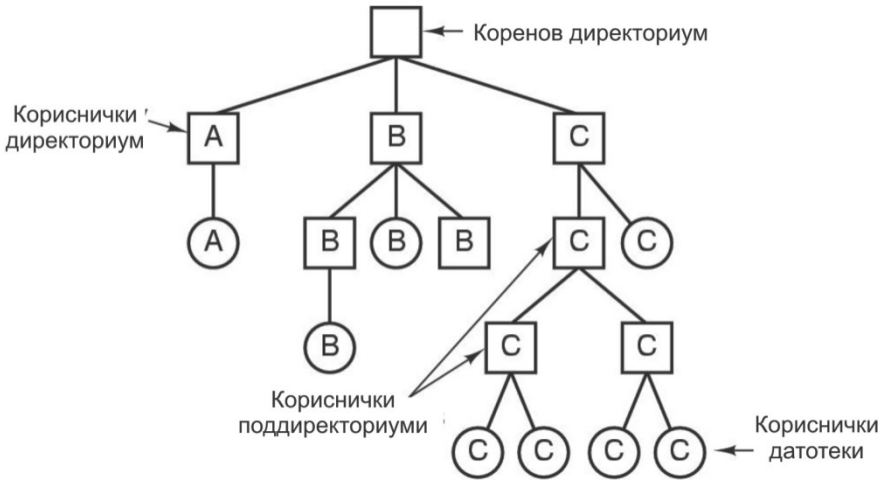
Како што еден корисник ќе се обиде да отвори датотека, системот веќе знае од кој корисник е и во кој директориум да ја бара. Затоа е потребна процедура за најава на системот (*logon*) за идентификација што не беше неопходно кај систем со едно ниво. Меѓутоа едно е кога корисниците ги чуваат своите архивски датотеки во еден директориум а друго за чување на извршни

датотеки што е крајно неефикасно затоа што ќе се повторуваат за повеќе корисници. Затоа барем треба да постои засебен системски директориум за чување на бинарните извршни датотеки.

Хиерархиски системи на директориуми

Хиерархијата со два нивоа ги елиминира конфликтите со имињата, но не задоволува во случајот кога корисниците имаат голем број на датотеки. Обично корисниците ги групираат датотеките на логички начини. На пример, може да се организираат според намената, професор кој пишува книга за еден предмет, сите негови датотеки и материјали би биле организирани во еден директориум, а презентациите за предавања по секој предмет во друг. Или на било кој друг начин одбран од страна на корисникот.

Затоа е потребно да се генерализира начинот на хиерархиското организирање во облик на стебло на директориумите за да датотеките бидат соодветно групирани. Еден пример е прикажан на Слика 9.12.



Слика 9.12: Хиерархиски систем на директориуми

Директориумите A, B и C се содржат во коренскиот директориум и припаѓаат на различни корисници, од кои два креирале поддиректориуми за своите датотеки. Можноста корисниците да создаваат произволен број на поддиректориуми обезбедува структуриран начин за организација на работата на корисниците. Треба да се истакне дека сите модерни датотечни системи се базирани на хиерархиски модел со поддиректориуми.

9.3.2. Имиња на патеки

Ако системот на датотеките е организиран како стебло на директориуми, потребен е начин за еднозначна идентификација и референцирање на датотеките. Се користат два начини, едниот каде што за секоја датотека е дадена со име на апсолутна патека која се состои од патеката од коренот до датотеката. На пример, *„/usr/ivan/kniga“* означува дека коренскиот директориум содржи поддиректориум *usr*, а тој еден поддиректорум означен со *ivan* и така понатаму. Апсолутните патеки се единствени. Компонентите на патеката се одделуваат со сепаратор кој кај различни системи е различен, како на пример:

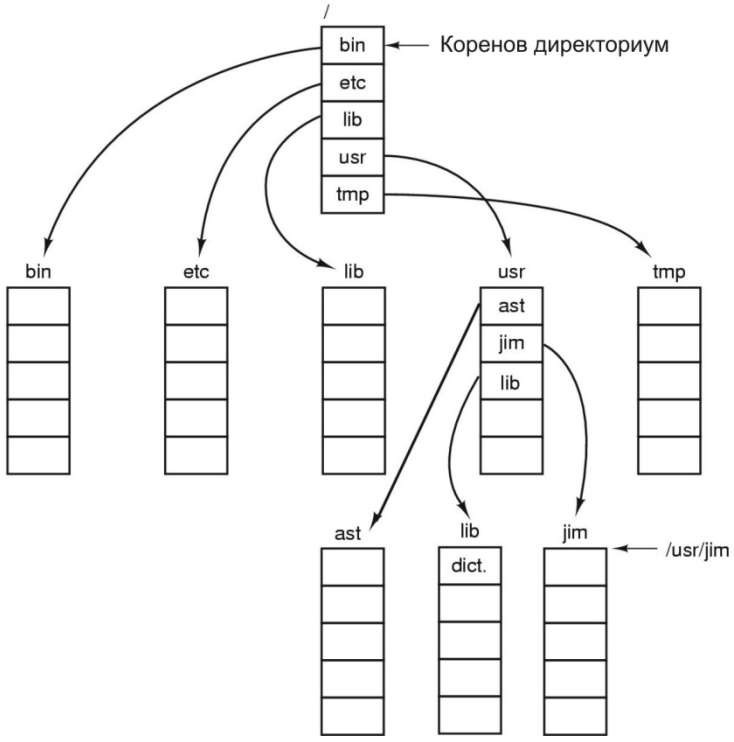
```
Windows     \usr\ivan\kniga
UNIX        /usr/ivan/kniga
MULTICS     >usr>ivan>kniga
```

Без разлика кој карактер се користи, ако првиот карактер од патеката е сепаратор, тогаш патеката е апсолутна. Специфицирањето на апсолутната патека не е соодветно за одредени ситуации, особено ако програмата содржи апсолутни патеки и има потреба да се пресели на друг компјутерски систем. Затоа се користи концептот на работен директориум (или тековен директориум). Датотеките се референцираат релативно во однос на тековниот работен директориум. На пример, ако тековниот директориум е */usr/ivan/* тогаш датотеката */usr/ivan/kniga* може да се референцира едноставно само со *kniga*. Со други зборови командата *cp /usr/ivan/kniga usr/ivan/kniga.bak* и командата *cp kniga kniga.bak* се потполно исти ако работниот директориум е поставен на */usr/ivan.* Некои програми имаат потреба за пристап кон одредена датотека без разлика на работниот директориум. Тогаш тие треба да ги користат апсолутните патеки и овој начин секој пат ќе биде валиден без разлика на поставеноста на тековниот директориум.

Секој процес има сопствен работен директориум и ако изврши негова промена ниту еден друг процес нема да почувствува последица поради тоа. Повеќето оперативни системи кои подржуваат хиерархија на директориуми имаат два специјални директориуми означени како „ . “ и „ .. “ каде што точка се однесува на тековен директориум, а две точки на директориумот родител. На Слика 9.13 е прикажан пример за користење на овие специјални директориуми. Одреден процес има поставено работен директориум /usr/ivan, со користење на .. се оди едно ниво погоре по стеблото. На пример може да се копира датотеката од */usr/lib/dictionary* во сопствениот фолдер со користење на командата.

```
cp ../lib/dictionary .
```

Првиот дел од патеката кажува на системот да оди едно ниво погоре, а потоа да се спушти во поддиректориумот *lib* и да ја најде датотеката со име *dictionary*.



Слика 9.13: Изминување на патеката до директориумот

Вториот аргумент (.) го означува тековниот директориум. Кога cp командата ќе го добие името на тековниот директориум ја запишува датотеката во него. Ова е потполно исто со командата *cp /usr/lib/dictionary . .*

9.3.3. Операции со директориуми

Операциите кои може да се изведат над директориумите повеќе се разликуваат од систем до систем отколку системскиот повици за операции на датотеките. Наведените операции даваат една претстава што се може да се изведе над директориумите, примерот е земен од оперативниот систем UNIX.

1. *Create.* Се создава директориум кој на почетокот е празен освен специјалните директориуми (.) и (..) кои автоматски се поставуваат.
2. *Delete.* Се брише директориумот, под услов тој да е празен. Празен е директориум кој ги содржи само директориумите (.) и (..).
3. *Opendir.* Директориумот се отвора за читање. На пример, за да се прикаже листата на датотеките во директориумот, програмата го отвора директориумот и ги чита имињата на сите датотеки.

4. *Closedir.* Кога директориумот ќе се прочита, тој треба да се затвори за да се ослободи просторот во внатрешната табела.
5. *Readdir.* Овој системски повик го враќа следниот запис во отворен директориум.
6. *Rename.* Директориумите се однесуваат слично како датотеките и според тоа може да се изврши и нивно преименување.
7. *Link.* Поврзување е техника која овозможува датотека да се појави во повеќе од еден директориум. Овој системски повик специфицира постоечка датотека и име на патека и креира врска од датотеката кон името специфицирано од патеката. На овој начин една датотека може да се појави и во повеќе директориуми. Врската од овој тип, со која се зголемува бројачот на i-јазолот на датотеката се нарекува „цврста врска“ (*hard link*)
8. *Unlink.* Се отстранува запис во директориумот. Ако датотеката на која и се раскинува врската е присутна само во еден директориум (што е вообичаено) се отстранува од системот на датотеки. Ако пак е присутна во повеќе директориуми се отстранува само името на патеката, а другите остануваат.

Наведените системски повици се најважните, иако постојат уште други со кои се управува на пример со заштитата на директориумите.

9.3.4. Врски

Како што беше веќе спомнато, во поголемиот број на системи датотеките може да се поврзат односно референцираат со други имиња од иста или различна позиција во стеблото на директориумот. Причината за користење на врските (*links*) е полесно пребарување и користењето на датотеките како и заштедување на просторот на дискот.

Неки системи на датотеки подржуваат повеќе видови на поврзување, на UNIX може да се формираат два вида на врски: *цврста врска* (*англ. hard link*) и *симболичка врска* (*англ. symbolic link*). Кога корисникот ќе ја повика датотеката по име, оперативниот систем го преведува симболичкото има на датотеката во интерно име кое го користи оперативниот систем.

Заради посебното интерно претставување корисниците на датотеките може да им доделат повеќе алтернативни имиња. Цврстата врска е едно од тие имиња. И референцата и оригиналното име имаат ист индексен јазол (i-јазол) така што мора да се наоѓаат на ист систем на датотеки. Датотеката со цврсти врски не може да се избрише сè додека не се избришат и сите цврсти врски кои наведуваат на таа датотека.

Симболичката референца е врска кон објектот во системот на датотеки, односно посебен објект на системот на датотеки кој користи еден индексен јазол и еден блок на податоци во кој е запишана локацијата на оригиналниот објект. Симболичка врска дава поголема флексибилност затоа што овозможува реферецирање на директориумите, непостоечки датотеки и датотеки кои се наоѓаат и на други датотечни системи.

9.4. Реализација на системи на датотеки

При работата на компјутерот, датотеките кои го сочинуваат самиот оперативен систем, корисничките програми и работните верзии на документите се чуваат на дискови. Датотеките кои не се користат често и претставуваат резервна копија на податоците (*backup*) се чуваат на терцијарните мемориски медиуми како што се CD, DVD или магнетни ленти.

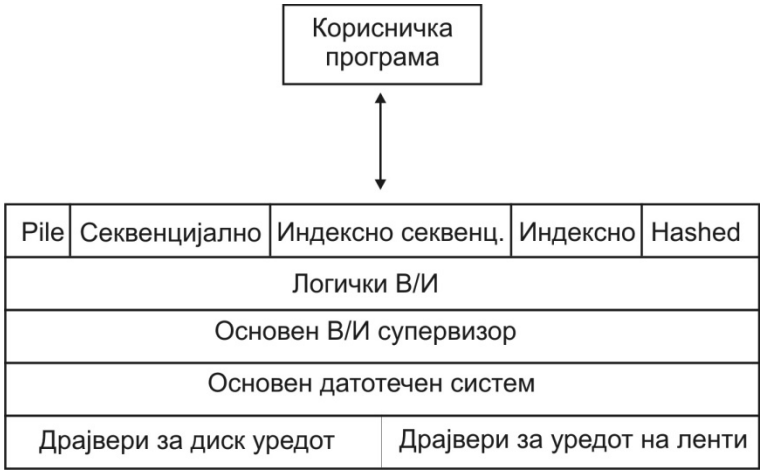
Повеќето дискови може да се поделат на повеќе делови, односно на партиции (*англ. partitions*), секоја со сопствен независен систем на датотеки. Секторот 0 на еден диск се нарекува MBR (*Master Boot Record*) и се користи за стартување на компјутерот. На крајот од MBR се наоѓа табелата на партициите каде што се дадени почетокот и крајот на секоја од партициите. Една од партициите е одбележена како активна. Кога ќе се стартува компјутерот, BIOS-от го чита и извршува MBR-от. Со тоа се наоѓа активната партиција и се чита првиот нејзин блок наречен *boot* блок и го извршува. Со тоа започнува вчитувањето на оперативниот систем кој се содржи на таа партиција.

Кај некои оперативните системи, покрај *boot* блокот на партицијата, датотечниот систем може да има и други податоци, како што е „суперблок“. Тој ги содржи сите клучни параметри за датотечниот систем и се вчитува во меморијата при вклучување на системот. Суперблокот содржи информации како што се магичен број со кој се идентификува видот на системот на датотеки, бројот на блоковите и други административни информации. Потоа може да дојдат и податокот за слободните блокови во датотечниот систем во форма на бит-мапа или поврзана листа, па следуваат и i-јазлите, кои носат податоци за датотеките. Потоа може да следува коренскиот директориум и остатокот од дискот е состои од сите други директориуми и датотеки.

Значи системот на датотеки е збир од правила и структури на податоци кои оперативниот систем ги користи за чување на датотеките. Системот на датотеки го сочинуваат: заглавје и метаподатоци како што се набројани претходно т.е. сите податочни структури до појава на коренскиот директориум. И втората компонента се самите податоци односно датотеките и директориумите. Заглавјето и метаподатоците го сочинуваат *overhead*-от на системот на датотеките неопходен за неговото функционирање. При реализација на системот на датотеки се дефинираат:

- Логичка структура на системи на датотеки, е начин на претставување на системот на датотеките спрема корисникот. Под ова се подразбира дефиницијата на датотеки, директориуми, атрибутите и дозволени операции.
- Физичка структура на системи на датотеки, ја сочинуваат структури на податоци на дискот кои служат за складирање на податоците. На пример, блокови (UNIX, Linux) и кластери (FAT).
- Пресликување на логичката структура на датотеките во физичката, претставува воспоставување на врска помеѓу содржината на конкретна датотека или директориум и структурата на дискот. На пример, пресликувањето одредува дека содржината на една датотека се наоѓа во тие и тие блокови на дискот.

Системот на датотеки се реализира во повеќе слоеви. При работата со објектите во системот на датотеките, на пример, со датотеките потребно е да се измина пат од името на датотеката до соодветниот физички блок на дискот. При тоа се поминува низ следниве слоеви:



Слика 9.14: Слоеви на системот на датотеки

- Логички систем на датотеки. На ова ниво се управува со метаподатоците, односно со сите структури на податоци поврзани со датотеката освен со нејзината содржина. Во метаподатоците спаѓаат контролниот блок на датотеката (*FCB - File Control Block*), положбата на датотеката на дискот и структурата на директориумите. За секој систем се дефинира специфично множество на метаподатоци. На пример, во системот на датотеки FAT метаподатоците се состојат од заглавје, две FAT табели и коренски директориум, додека кај UNIX/Linux системите метаподатоците се реализирани преку табела на индексни јазли и заглавјето (суперблок).

- Организација на датотеки. Модул кој води сметка за адресите на логичките блокови на датотеките и врз основа на барањето за пристап кон одреден дел од датотеката се одредува логичката адреса на која се наоѓа тој дел.
- Физички систем на датотеки. Овој слој испраќа основни команди за управување со дискот кон драјверот. Тука логичките адреси се претвораат во физички тродимензионални адреси.
- Управување со уредите. На ова ниво се сместени драјверите за уредите (дискот или магнетна лента) и рутините за обработка на прекини. Тука за врши управување со самиот уред, односно се врши префрлање на податоците од дискот кон меморијата и обратно. Адресирањето е физичко, адресите се тродимензионални и ја дефинираат местоположбата на податоците на дискот.

9.4.1. Структури на податоци за реализација на системот на датотеки

За да се реализира систем на датотеки, потребни се одредени податочни структури на дискот:

1. BCB (*Boot Control Block*). Содржи информации кои се потребни за започнување на процесот на подигање на еден оперативен систем. Тоа е обично првиот блок на дискот.
2. Контролен блок на партицијата (PCB). Ги содржи информациите за системот на датотеки како што е големината и бројот на блоковите, листа и покажувач на листата на слободни блокови и т.н.
3. Контролни структури за доделување на датотеките. Се одредува конкретната содржина на датотеката, логички блокови за содржина на датотеката. Тука спаѓаат табелата на индексни јазли (*UNIX*), FAT табелата (*FAT*) и MFT (*NTFS*).
4. Директориумски структури кои ги содржат контролните блокови на датотеките.
5. Контролни блокови на датотеките (*FCB*). Ги содржи атрибутите на датотеките и опис на просторниот распоред, односно покажувачите на блоковите во состав на датотеката. Кај UNIX системите, контролниот блок се состои од два дела: FCB на структурата на директориумите, која го содржи името на датотеката и индексниот јазол во кој се наоѓаат сите атрибути и просторниот распоред на датотеката на дискот. Во системите NTFS, секоја датотека го чува својот контролен блок во MFT табелата.

На Слика 9.15 е прикажан еден FCB.

Дозволи на датотеката
Датуми на датотеката (креирање, пристап, запишување)
Сопственик на датотеката, група, ACL
Големина на датотеката
Податочни блокови на датотеката

Слика 9.15: Блок за контрола на датотека FCB

Во работната меморија се дефинираат структури кои ја забрзуваат работата на системот на датотеки:

- Табела на отворени датотеки на системско ниво која ги содржи контролните блокови за секоја датотека.
- Табела на отворени датотеки по процес, која ги содржи покажувачите на главната табела и други информации како што е тековниот покажувач на датотеката.

При отворањето на датотеката, нејзиниот контролен блок се запишува од директориумот во главната мемориска табела. При секој пристап кон датотеката од табелата на процеси преку главната табела на отворени датотеки се наоѓаат блоковите на дискот кон кои се пристапува. Битно е блоковите да се пребаруваат во меморијата со што се подобруваат перформансите на системот.

9.4.2. Доделување на простор на датотеките

Најбитниот дел при имплементација на склад на датотеки е да се следи зафатеноста и доделувањето на блоковите од дискот кон блоковите на датотеката. Се користат повеќе различни методи доделување во различни оперативни системи:

- Доделување на континуиран простор.
- Поврзување на блоковите
- Мапа на датотеките
- Метода на индексни блокови

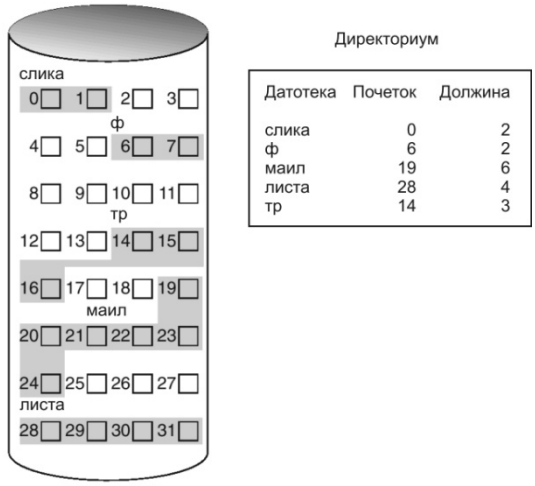
При изборот на методата, се земаат в предвид и следниве претпоставки, дека датотеките се не-структуирани низи од бајтови, треба да се оствари ефикасен пристап кон дискот и за секвенцијалниот и случајниот пристап, односно минимизирање на бавните пребарувања. И на крајот да се овозможи и

ефикасна искористеност на просторот односно да се минимизира надворешната и внатрешната фрагментација

Континуирано доделување

Тоа е наједноставен начин каде што секоја датотека се сместува во континуирана низа од блоковите во дискот. Така ако дискот има блокови со големина од 1 KB, датотека со големина од 50 KB ќе зафати 50 последователни блокови од дискот. На Слика 9.16 е даден пример за доделување на блоковите на повеќе датотеки, потребно е познавање на локацијата на почетниот блок и должината на датотеката во блокови за да се изврши доделувањето. Ако големината на датотеката е помала од цел број на блокови на дискот, последниот блок нема да биде исполнет до крај и ефикасно искористен.

Континуирано доделување на просторот на дискот има две битни предности. Прво, лесен е за изведба, затоа што е доволно познавање на почетниот блок и должината, наместо следењето на позиција на сите блокови во состав на датотеката. Друго, брзината на читањето на датотеката е голема затоа што тоа може да се изведе со една диск операција. Потребно е пронаоѓање само на првиот блок, нема ротациско доцнење и губење на време на пронаоѓање на секој блок одделно туку се читаат последователно со полна преносна брзина. Значи континуираното доделување е едноставно за изведба и има висока перформанса.



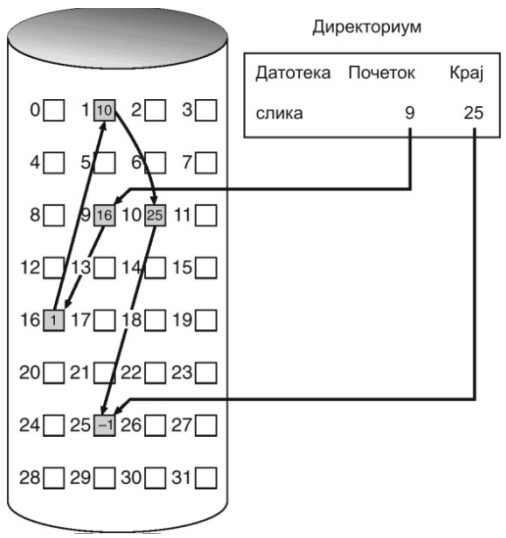
Слика 9.16: Континуирано доделување на простор на датотеки

Меѓутоа, континуираното доделување има голем недостаток, со тек на време дискот станува фрагментиран. На сликата може да се забележат низи од слободни блокови кои настанале по бришењето на датотеките, при што дискот

ќе се состои наизменично од датотеки и од низи на слободни блокови – дупки. Фрагментацијата и може да не биде проблем, затоа што секоја нова датотека може да се запише на крајот, но во еден момент ќе биде потребно да се изведе операција на збивање што е неефикасно. За повторно користење на празните блокови потребно е да се води листа на слободниот простор, при што за сместување на нова датотека треба иницијално да се познава нејзината конечна големина за да се избере вистинската локација. Континуираното доделување широко се користи кај датотечните системи на оптичките уреди. Тука сите големини на датотеките однапред се познати и се непроменливи во текот на користењето на оптичките медиуми.

Сврзана алокација

Блоковите кои се доделени на датотеката може да бидат произволно разместени на дискот. Секоја датотека претставува поврзана листа од блокови на дискот каде што неколку бајти од секој блок (од 32 до 64 бита) се одвојува за сместување на покажувачот на следниот блок од датотеката. Останатиот дел од блокот се користи за сместување на податоци. Последниот блок од датотеката содржи празен покажувач. Во контролниот блок на датотеката доволно е познавање на почетокот и крајот на датотеката односно бројот на првиот и последниот блок.



Слика 9.17: Сврзана алокација

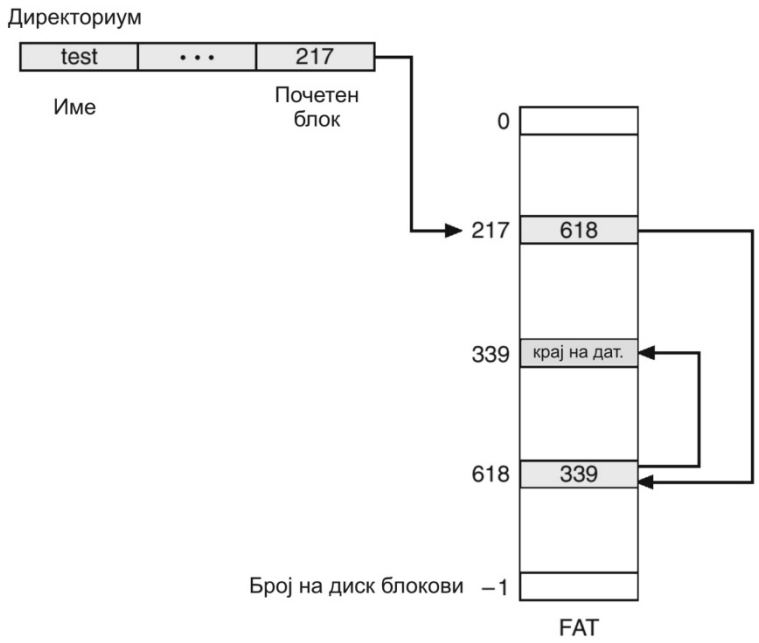
За разлика на континуираното доделување, секој блок на дискот може да биде искористен. На тој начин не се предизвикува губење на простор на дискот заради фрагментација, освен на интерната фрагментација на последниот блок. Иако секвенционален пристап е ефикасен, случајниот пристап е многу бавен.

За да оперативниот систем го најде блокот n, треба да се започне читањето од првиот блок да се прочитаат сите *n-1* и тогаш да се пристапи кон саканиот блок. Освен тоа, податочната содржина во блокот веќе не е со големина степен на два, туку намалена за големината на покажувачот.

Сврзана алокација со користење на табела

Двата недостатоци на доделување со сврзана листа може да се елиминира со одземање на покажувачкиот збор од секој блок на дискот и да се постави во табела во меморијата. На Слика 9.18 е прикажан изгледот на табелата.

Датотеката test ги користи блоковите од дискот 217, 618 и 339 со дадениот редослед. Со користењето на табелата може да се започне со почетниот блок 217 и последователно да се добијат и ознаките на другите блокови. На крај, сите датотеки се терминирани со специјална ознака (на пример -1) кој не претставува валиден број на блок. Таква табела сместена во меморијата се нарекува табела на доделување на датотеките (*FAT - File Allocation Table*). На овој начин, целиот блок е достапен за податоците и случајниот пристап е многу поедноставен затоа што целата табела се наоѓа во меморијата.



Слика 9.18: Сврзана алокација со користење на FAT табела

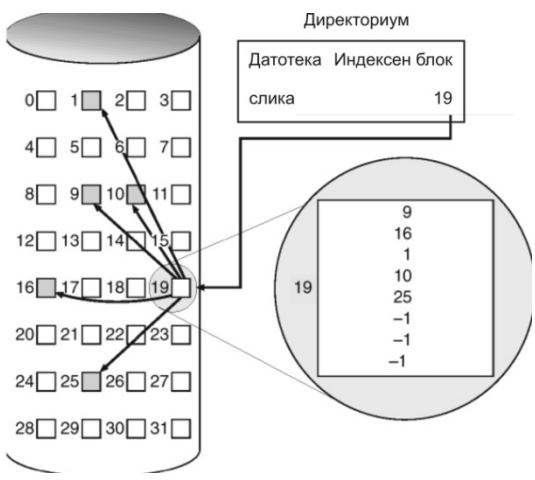
FAT во суштина претставува комбинација на двете методи на мапа на датотеките и континуирана алокација. Системот на датотеките при неговото

формирање се дели на кластери (*англ. clusters*). Во рамки на еден кластер на датотеката се доделува континуиран простор. На една датотека може да и се доделат повеќе кластери но еден кластер не може да се дели помеѓу две датотеки. Неискористениот дел од кластерот е загубен простор кој се нарекува *slack*. Сите кластери на една датотека преставуваат поврзана листа и секој запис во FAT табелата претставува еден кластер.

Главниот недостаток на овој метод е што целата табела треба да биде сместена во меморијата за цело време на работата на системот. Со диск со големина до 20 GB и големина на блок од 1 KB табелата треба да содржи 20 милиони записи со по најмалку 3 бајти должина, што доведува до табела со големина од 60 MB. Покрај ова потенцијални грешки во табелата може да создадат голем број на грешки во системот на датотеките. Во системот на датотеки FAT се формира резервна копија на табелата која редовно се обновува и на тој начин се зголемува доверливоста на системот.

I-јазли

За да се следи доделувањето на блоковите кон датотеките може да се користи податочна структура наречена i-јазол односно индексен јазол. За даден i-јазол може да се најдат сите блокови за една датотека. Голема предност на овој начин пред сврзаното доделување со користење на табела во меморијата е тоа што е доволно во меморијата да биде сместена само индексниот јазол на датотеката која е отворена. Ако секој *i*-јазол зазема *n* бајти и максимум *k* датотеки може да се отворат тогаш меморијата која е потребна за сместување на јазлите е еднаква на *k* пати по *n*. Овој простор е значително помал од оној кој ги зафаќа FAT табела каде што потребниот простор се зголемува со зголемувањето на капацитетот на дискот.



Слика 9.19: Индексни јазли

Проблемот со индексните јазли е фиксната големина на јазелот односно ограничување на бројот на адресите кои тој може да ги содржи. Овој проблем може да се реши на два начини:

1. Поврзување на индексни јазли, каде што за опис на поголема датотека се користат повеќе индексни јазли и последната адреса од јазолот не покажува на број на блок туку на адреса на блок кој содржи уште адреси на блокови.
2. Адресирање на повеќе нивоа. Контролниот блок на датотеката покажува на еден индексен јазол од прво ниво, кој пак покажува на *n* индексни јазли од друго ниво. Индексните јазли од второ ниво ги адресираат блоковите на датотеките. Бројот на нивоата може по потреба да се зголеми. Кај UNIX се користи адресирање на блоковите на податоците на 3 нивоа.

Во однос на перформансите, методата на индексни јазли ги надминува претходно наведените методи но бара најголем простор за мета структурата на дискот. Исто така бројот на индексните јазли е ограничен, со што е ограничен и бројот на датотеките кои може да се отворат. Ова ограничување не постои кај датотечни системи кои користат табела на доделување на блокови.

9.4.3. Имплементација на директориуми

Пред да може датотеката да се чита треба претходно да се отвори. Кога ќе се отвори датотеката, оперативниот систем го користи името на патеката дадено од корисникот за да го најде записот во директориумот. Записот во директориумот ја дава информацијата потребна за да се најдат блоковите на дискот. Зависно од системот, информацијата може да биде: адресите од дискот за целата датотека (континуирано доделување), бројот на првиот блок (поврзана листа) или бројот од индексниот јазол. Во сите случаи, главната функција на директориумот е да се изврши пресликување од името на патеката во информацијата која е потребна за наоѓање на податоците на дискот. Покрај тоа што директориумите ги содржат контролните блокови тие се одговорни и за пребарувањето со што директно имаат влијание врз перформансите на системот на датотеки. Постојат две основни шеми за реализација на директориумите:

- Линеарна листа, која се карактеризира со сместување на нови контролни блокови по редоследот идентичен на редоследот на создавање на датотеките. Пребарувањето може долго да трае доколку во директориумот има голем број на датотеки и поддиректориуми. Пребарувањето се забрзува со воведување на разни техники како што се балансирани стебла.
- Линеарна листа со хеш табела, се карактеризира со организација на линеарна листа каде што управувањето со директориумот се користи

хеш табела со што се забрзува пребарувањето. Главен проблем претставува фиксната големина на хеш табелата со што се поставува ограничување на бројот на записите по директориум.

9.4.4. Раководење со слободниот простор

Слободниот простор на дискот се доделува кон датотеките при нивното креирање и проширувањето односно порастот на датотеките. Управувањето со слободниот простор значително влијае врз перформансите на системот на датотеките.

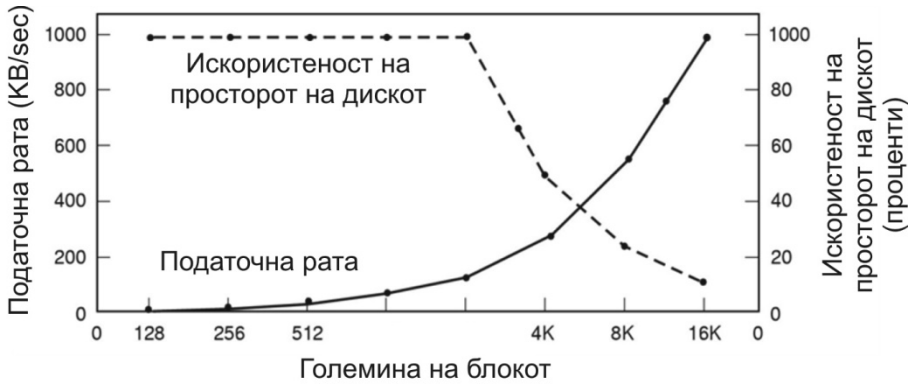
Големина на блоковите

Главен фактор при организација на просторот на дискот е прашањето на големината на блоковите. Според начинот како се организирани дисковите, главни кандидати би биле сектор, патека или цилиндар, а ако се користи и систем за страничење на главната меморија, може покрај овие да биде и големината на страницата.

Избор на големи блокови може да доведе до неефикасно искористување на просторот (на пример, на датотека од 1 KB се доделува блок со големина од 32 KB), а од друга страна пак користење на мали блокови доведува до ситуација една датотека да се состои од премногу блокови и читањето може да биде многу забавено. На пример, ако се земе еден диск со 131.072 бајти по патека, со време на ротација од 8.33 милисекунди и просечно време на пребарување (*seek*) од 10 милисекунди. Тогаш времето за читање на блок од k бајти е сума на пристапот, ротациското доцнење и времето на пренос.

$$10+4.165+(k/131072) \times 8.33$$

На графиконот (Слика 9.20) е прикажана брзината на пренос на податоците за дискот даден во примерот во функција од големината на блокот. Се претпоставува средна големина на датотека од 2 KB. Времето на пристап кон одреден блок се состои најповеќе од времето на ротација и барање (*seek*) и затоа е потребно да се пренесе наеднаш што поголемо количество на податоци. Со зголемувањето на големината на блокот расне и ефикасноста на пренос односно брзината на пренос (KB/sec).



Слика 9.20: Брзина на пренос на податоци и големина на блокот

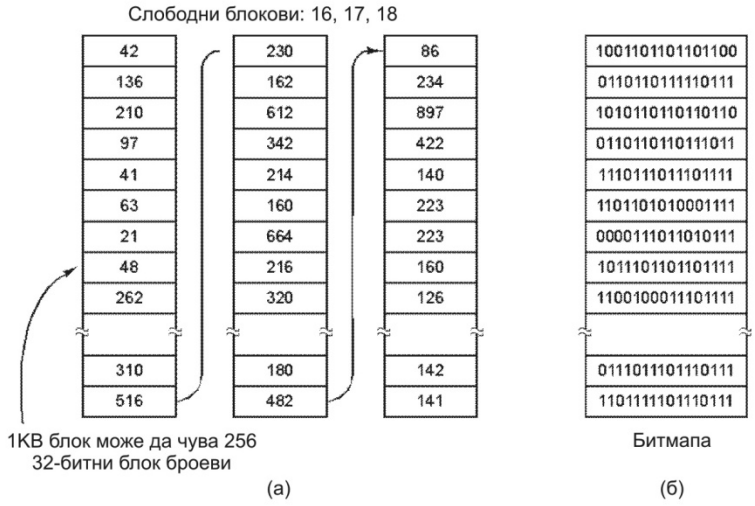
Со мали блокови со големина од степен на 2 и датотеки со претпоставена големина од 2 KB не постои изгубен простор на дискот. Со поголеми блокови се губи простор на дискот и тоа обично во последниот блок од датотеката. Практично ретко се случува датотеката да има големина со точна вредност од степен на двојка и секогаш постои загуба на просторот на дискот. Големината на блокот и перформансите на дискот се во спротивност и е потребен компромис во изборот на големината на блокот. За UNIX обично се користи 1 KB, за MS-DOS блокот може да биде степен на 2 во опсег од 512 бајти до 32 KB, но е одреден од големината на дискот, односно максималниот број на блоковите на диск партицијата е 2^{16} со што при големи дискови се добиваат и големи блокови.

Следење на зафатеноста на блоковите

Постојат два начини за следење на слободните блокови. Првиот се состои од користење на поврзана листа на блокови каде што се содржани броевите на слободните блокови. Со еден блок до 1 KB и 32-битен број на блок на дискот, секој блок може да содржи броеви на 255 слободни блокови и еден број за покажувач на нареден блок во листата. Еден 16 GB диск има потреба од листа со максимум 16,794 блокови за чување на сите 2^{24} броеви на блокови на дискот.

Другиот начин за следење на слободните блокови е со користење на битмапа. Диск со n блокови треба да има битмапа со n битови, алокацијата на блоковите на дискот се претставува како низа од битови каде што секој бит означува дали соодветниот блок до дискот е слободен или не. Бит за секој блок на дискот: 1 – неалоциран блок, 0 – алоциран блок. Овој начин се карактеризира со едноставно и ефикасно наоѓање на слободни блокови. Споредено со претходниот пример, 16 GB диск бара 2^{24} бита (1 KB блок) за мапата=2048 блока, што е очигледно многу помалку во однос на листа на поврзани блокови,

единствено само кога дискот е скоро полн (т.е. има мал број на слободни блокови) тогаш листата користи помал број на блокови во однос на битмапата.



Слика 9.21: Управување со слободните блокови со а) поврзана листа, б) битмапа

9.5. Сигурност и заштита на системи на датотеки

Заради комплексноста на структурите на податоците на системите на датотеки, врз доверливоста на системот влијаат многу фактори. Како на пример доверливоста на самата површина на дискот и начинот како оперативниот систем се справува со неконзистентност по претходен испад на системот.

Уништувањето на системот на датотеките обично предизвикува поголема штета од уништувањето на компјутерот. Ако системот на датотеки се уништи, враќањето на податоците може да биде многу тешко па дури и невозможно, а последиците може да бидат несогледливи. Системот на датотеки не може да овозможи заштита од хардверски испади, но може да имплементира заштита на информациите кои се чуваат на дискот.

Оштетувањата на системот на датотеки може да се поделат во две категории:

- Физички оштетувања, се однесуваат на површината на дискот и може да предизвикаат неповратно губење на податоците. Еден вид на прелиминарна можност за избегнување на испади е да се пријават оштетените блокови при формирањето на системот на датотеките, иако самата појава има тенденција на зголемување. Затоа дискот со физички оштетувања треба да се замени. Причините за физичките оштетувања може да бидат најразлични, на пример, елементарни непогоди (пожар,

вода, земјотрес, глувци и т.н.), хардверска или софтверска грешка или натрапници и други.

- Логички оштетувања, се однесуваат на недоследности во метаструктурите на системот на датотеки, како последица на неизвршување на празнење на кешот (*англ. write behind cache*), грешка во комуникација, багови во програми, човечка грешка и други.

Дел од споменатите причини може да се избегнат со користење на дополнителни уреди, како што се уреди за непрекинато напојување (*Uninterruptable Power Source - UPS*), техники на водење на трансакции во дневник или пак користење на огледални дискови (RAID системи, ниво 1).

9.5.1. Архивирање и резервни копии (Backup)

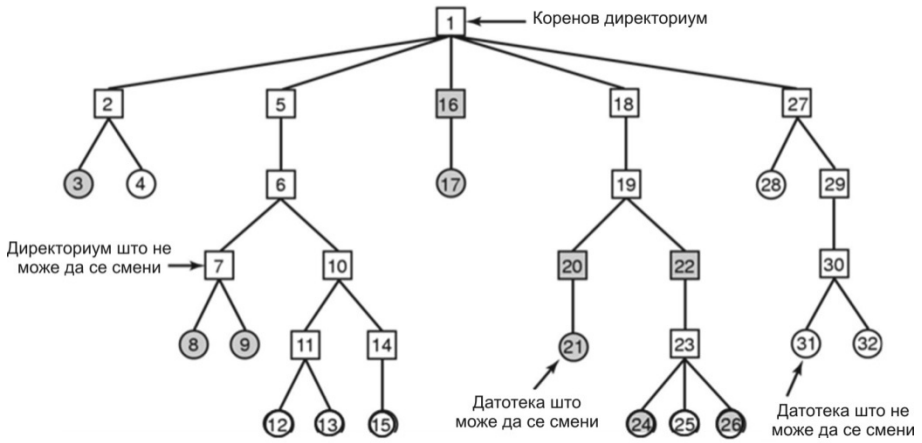
Наједноставен начин за обезбедување на интегритетот на податоците е нивно архивирање, односно создавање на резервни копии. Значењето на оваа операција обично е занемарено од страна на обичните корисници, сè до појава на испад на системот, додека кај компаниите се посветува многу поголемо внимание за архивирање на податоците. Архивирањето извршува едноставна функција – враќање на изгубените податоци и претставува превентивна мерка каде што се архивираат позначајните податоци кои потоа лесно се реконструираат при испад на системот.

Единствената сличност помеѓу системот на датотеките и архивата е во нивната содржина и двете содржат датотеки и директориуми. Архивата е малку поедноставна структура на податоци над која може да се извршат само две операции, додавање на датотеки и нивно враќање назад во состав на системот на датотеки. Пристапот кон содржината на архивата е секвенцијален, при што новите датотеки секогаш се додаваат на крај од архивата, а постојните не може да се избришат.

Операцијата за архивирање на податоците бара многу време и простор и е значајно таа да се изведува ефикасно. Се поставуваат неколку проблеми, дали да се архивира целиот систем на датотеки или само одредени делови и кои од нив, за да се овозможи зачувување на интегритетот. Кај повеќето системи, бинарните односно извршните датотеки се чуваат во одреден дел од стеблото на системот на датотеки. Тие не треба да се архивираат, затоа што при испад на системот, може лесно да се реконструираат со нивно повторно инсталирање. Исто така, директориумите кои содржат привремени податоци или специјалните датотеки не треба па дури и не е пожелно да се архивираат заради нивните карактеристики. Треба да се архивираат само значајни кориснички податоци или податоци кои се неопходни за реконструкција на системот наместо да се архивира целиот систем.

Друго прашање кое се појавува е дали да се архивираат датотеките кои не се измениле при последното архивирање, што доведува до техника на

инкрементално архивирање (*англ. incremental*). Наједноставна форма на оваа техника е да се врши архивирање на целиот систем периодично (неделно или месечно) и да се врши дневно архивирање на податоците кои се измениле од последното целосно архивирање. На овој начин се минимизира времето за архивирање, но реконструкцијата треба да се изврши од последното целосно архивирање и сите инкрементални архиви во обратен редослед. Трето, затоа што при архивирањето се складира огромно количество на податоци, се врши и нивна компресија, при што може да се доведе во прашање доверливоста на архивата во зависност од квалитетот на медиумот за складирање.



Слика 9.22: Систем на датотеки кој треба да се архивира

Четврто, тешко може да се изведе архивирање на активен датотечен систем каде што се можни додавање, бришење и измена на датотеките и директориумите при процесот на архивирање што доведува до неконзистентна копија. Од друга страна, не смее да се дозволи системот да биде долго време исклучен додека се врши архивирањето. И на крај изведувањето на архивирањето предизвикува појава и на не-технички проблеми во компанијата, како на пример обезбедување на соодветна локација за чување на резервните копии. Може да се користат две стратегии за архивирање на содржината на дискот, односно системот на датотеки на секундарен медиум како што е магнетна лента или оптички уред.

Едната е физичко архивирање (*англ. physical dump)* со која што се врши копирање на секој блок од дискот, се карактеризира со брзина која е еднаква на читање на блоковите од дискот. Недостатоците се што не може да се реконструираат поединечни директориуми и датотеки. Заради овие причини обично се користи логичко архивирање (*англ. logical dump*). Логичкото архивирање започнува од еден или повеќе специфицирани директориуми и рекурзивно ги архивира сите датотеки и директориуми кои биле изменето во однос на одредена временска ознака, на пример последно целосно или

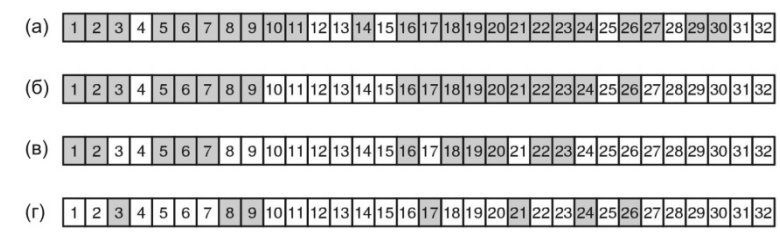
инкрементално архивирање. На магнетната лента се чуваат низа на внимателно означени датотеки и директориуми со што се овозможува реставрација на одредена датотека по потреба. Даден е пример на алгоритам за логичко архивирање кој се користи кај повеќето UNIX системи. На Слика 9.22 е дадено стебло на директориуми (квадрати) и датотеки (кружници), каде што со сенчење се означени оние кои се изменети од последната операција за архивирање и има потреба од нивното повторно архивирање.

Овој алгоритам ги архивира сите директориуми, дури и оние кои се неизменети за да се овозможи пренесување на структурата на системот на датотеки и на други системи и дискови. Како и да се овозможи инкрементално архивирање.

Алгоритамот за архивирање одржува битмапа која е индексирана со бројот на i-јазолот со по неколку битови по јазол. Битовите се поставуваат и бришат при извршувањето на алгоритамот. Алгоритамот за извршува во 4 фази. Првата фаза започнува со почетниот директориум (во примерот тоа е коренскиот директориум) и ги испитува сите кои се содржани во него, при што соодветно се менуваат битовите во битмапата ако испитаната датотека е изменета. За секој директориум се врши одбележување без разлика на неговиот статус. По завршувањето на првата фаза сите изменети датотеки и сите директориуми за одбележени (како што е прикажано на сликата). Втората фаза уште еднаш го изминува стеблото и ги демаркира директориумите кои не содржат изменети директориум и датотеки. Директориумите 10, 11, 14, 27, 29 и 30 сега не се означени. Двете фази може да се комбинираат во една заради поголема ефикасност.

Третата фаза се состои од скенирање на i-јазлите по нумерички редослед и архивирање на сите директориуми кои се означени и на крајот во четвртата фаза се архивираат и сите датотеки заедно со нивните атрибути со што се завршува процесот на архивирање.

Реставрацијата е едноставна и се одвива по следниов редослед, прво се креира празен систем на датотеки, потоа се врши реставрација од целосната архива, директориумите први се реконструираат со што се добива скелетот на системот, а потоа и датотеките. Истиот процес се извршува и за наредните инкрементални архиви.



Слика 9.23: Битмапите кои се користат за логичко архивирање

9.5.2. Конзистентност на системот

Поголемиот дел од датотечните системи вршат читање на блокови, нивна измена и ниво подоцнежно запишување. Ако системот испадне пред да се запишат сите модифицирани блокови, датотечниот систем може да се најде во состојба на неконзистентност. Проблемот е уште поголем ако блоковите се дел од i-јазлите, директориумите или блокови кои ја содржат листата на слободни блокови.

За да се справат со овој проблем оперативните системи поседуваат алатки за проверка на конзистентност на системот, како на пример *fsck* кај UNIX или *scandisk* кај Windows.

Постојат два вида на проверка за конзистентност: во однос на блокови и датотеки. При проверката на конзистентност, програмата создава две табели, една која содржи бројач за секој блок, иницијално поставена на 0. Овие бројачи го следат бројот колку пати секој блок е присутен во една датотека, а бројачите во втората табела колку пати еден блок е присутен во листата на слободни блокови (или битмапата на слободните блокови).

Потоа се читаат сите i-јазли, со што се креира листа на сите броеви на блокови кои се искористени во соодветната датотека. Секој пат кога ќе се прочита блок се зголемува бројачот во првата таблица. Програмата потоа ја испитува листата на слободни блокови за да се најдат блоковите кои не се искористени. Ако системот на датотеки е конзистентен, секој блок ќе има 1 или во првата или втората табела, т.е. блокот или е зафатен или е слободен. Меѓутоа може да се случи, како резултат на испад на системот да нема запис ниту во една табела, тоа се означува како „блок што недостасува“, иако не претставуваат закана овие блокови го намалуваат капацитетот на дискот. Проблемот се решава со користење на некоја претходно наведена програма за проверка на дискот, при што овие блокови се додаваат во листата на слободни блокови (или битмапа). Друг случај кој може да се појави е кога некој блок се појавува два пати во листата на слободни (ова не е можно кога се користи битмапа за управување со слободниот простор) и решението е повторно да се изгради листата. Најлошиот случај кој може да се појави е дека еден блок е присутен во две или повеќе датотеки. Ако некоја од овие датотеки се избрише, блокот ќе се додаде во листата на слободни иако истовремено припаѓа и кај други датотеки. Ако се избрише и некоја од тие, блокот ќе се појави два пати во листата на слободни. Програмата за проверка на системот треба да изврши алокација на слободен блок, да изврши копирање на содржината на блокот и да ја вметне копијата во датотеките. Со тоа се спасуваат податоци од една датотека, а другите ќе бидат оштетени, но датотечниот систем останува во конзистентна состојба.

Програмата за проверка, покрај датотеките ја проверува и конзистентноста на директориумите, каде што се користат табели не по блокови туку по датотеки и тука се можни неколку потенцијални проблеми. Можно е да се појават дупли

датотеки заради повеќекратно референцирање во повеќе директориуми на една датотека (тврдо поврзување) каде што i-јазолот кажува дека само еден директориум е поврзан со датотеката, ако се избрише датотеката i-јазолот станува 0 и се ослободуваат блоковите на директориумот иако постојат датотеки во него. Во принцип, проблемите настануваат кога бројот на датотеките на кореспондира со бројот во i-јазолот, ако бројот е поголем и при бришење на сите датотеки тој нема да биде ослободен, обратно како што е кажано, пак можно е пребришување на податоци заради предвремено ослободување на јазолот.

9.5.3. Системи на датотеки базирани на дневник

Идејата кај овој вид на системи на датотеки произлегува од фактот дека процесорите во компјутерските системи стануваат сè побрзи и RAM мемориите се поголеми, како и капацитетот на кеш меморијата на дисковите. Следствено станува можно да се задоволат сите барање за читање директно од кеш меморијата на системот на датотеките. Оттука следува дека постои тренд сè повеќе диск операции да бидат главно операции на запишување и тоа на мали парчиња што е крајно неефикасно и предизвикува опаѓање на искористеноста на трансферот на податоците.

Заради тоа, е создаден концептот на датотечни системи базирани на дневник – LFS (*Log-Structured File Systems*) со што се постигнува полн трансфер на дискот иако постојат голем број на операции за запишување на мало количество на податоци. Целиот диск е структуриран како дневник, периодично како што се појавува потреба сите операции за запишување кои се наоѓаат во баферот се собираат во еден сегмент и се запишуваат на дискот како континуиран сегмент на крајот од дневникот. Сегментот може да содржи и i-јазли, блокови од датотеки и директориуми. На почетокот од секој сегмент е запишано што се содржи во него. Кога се отвора датотека потребно е да се користи мапа за да се пронајдат првин i-јазлите, од каде пак се наоѓаат адресите на блоковите, кои се наоѓаат по сегментите некаде во дневникот. Бидејќи дисковите имаат конечен капацитет, дневникот со текот на времето ќе го заземе целиот диск и нема да биде можно да се врши запишување на нови сегменти во дневникот.

Многу сегменти содржат блокови кои не се повеќе потребни, на пример, ако се пребрише една датотека, нејзиниот i-јазол ќе покажува на нови блокови додека старите се уште ќе зафаќаат простор во претходно запишаните сегменти. LFS поседува нишка за чистење (*англ. cleaner thread*) која врши прегледување на дневникот и извршува збивање. Се чита содржината на сегментот, за да се види кои директориуми и датотеки се наоѓаат. Потоа се проверува тековната мапа на i-јазелот дали јазлите се сé уште тековни и дали блоковите од датотеката се користат. Јазлите и блоковите кои се користат се пренесуваат во меморијата за да се запишан во следниот сегмент. Оригиналниот сегмент се

означува како ослободен и дневникот може да го користи за запишување на податоци. На овој начин, нишката-чистач врши изминување од еден до друг крај, ослободува сегменти, а тековните податоци од тие сегменти се презапишуваат во следниот сегмент. Целиот диск претставува циркуларен бафер, каде што нишката –запишувач додава нови сегменти на почетокот и нишката – чистач ги отстранува старите од крајот.

Следењето е на блоковите не е едноставно затоа што кога еден блок од датотека ќе се запише назад во нов сегмент, i-јазолот од датотеката (некаде во дневникот) треба да се пронајде, обнови и да се постави во меморијата за да се запише во следниот сегмент. Мапата на јазлите исто така треба да се обнови за да покаже на новата копија на јазолот.

Системот овозможува едноставно опоравување од испади, затоа што се користи трансакциска семантика. Имено постои покажувач (курсор), кој се поместува на следен сегмент кога изменетите блокови ќе бидат целосно запишани на дискот (стврднати – „*hardened*“). При опоравување по појава на испад при запишување на сегментот, се врши повторување на операции започнувајќи од последната позиција на курсорот. Во случај системот да падне процедурата е следнава, при стартот се чита дневникот, се обезбедува датотечниот систем и податоците да се конзистентни со дневникот, ако е пронајдена неускладеност, се ажурираат датотеките со помош на дневникот, ако последната операција не е извршена, може да биде откажана.

Системот на датотеки структуриран како дневник има неколку предности во однос на другите видови, податоците се запишуваат асинхроно, со што се обезбедуваат поголеми перформанси при операцијата на запишување. И предноста е и можноста за брзо оправување од испад на системот кое зависи од големината на дневникот и на системот на датотеки. Од другата страна, постојат и недостатоци, како што е дополнителни операции за запишување на метаподатоците како и просторот на дискот кој е заземен од дневникот.

Водењето дневник ги подобрува перформансите преку агрегација на запишувањата, на пример, доцнењето на запишување во датотека или директориум е на пример 1 (или 2 со дневникот) запишување, наместо можни 5 на помали делови. Групното запишување овозможува групирање на операциите во едно “големо” запишување и бидејќи на дневникот постојано му се пристапува, главите на дискот се поголем дел од времето позиционирани “над” дневникот, односно тековниот сегмент со што се редуцира движењето на главите на дискот (односно „seek“ времето),

Зголемувањето на бројот на запишувања на дискот поради водењето дневник не влијае негативно на перформансите. Реалните имплементации се хибридни каде што еден дел од дискот користи систем структуриран како дневник, а остатокот се користи за континуирани запишувања.

9.6. Перформанси на системи на датотеки

По прашањето на ефикасност на системот на датотеки порано се придавало големо значење заради малиот капацитет на дисковите. Дисковите кои се произведуваат во денешно време се доволно големи, така што проектантите може да го занемарат проблемот на ефикасноста на искористувањето на просторот на дискот и да се посветат на зголемувањето на перформансите на системот на датотеки. Перформансите на системот на датотеки може да се влијае на повеќе начини. Една од техниките која има големо влијание на перформансите на системот е кеширање (*англ. cache*).

9.6.1. Кеширање

Кеш претставува меѓумеморија со која се премостува јазот помеѓу бавната електромеханиката на дискот и брзата работна меморија. Тој постои како хардверска компонента на процесорите и поновите модели на дискови. Оперативниот систем за да ги зголеми перформансите имплементира и софтверско кеширање со користење на кеш на мемориски страници (*англ. page cache*) и кеш на датотеки (*англ. file cache*). Кеширањето на датотеките се одвива во циклусите на читање и запишување. Со кеширањето на циклусот на запишување, сите датотеки кон кои често се пристапува се пренесуваат во кеш во главната меморија, следното читање на датотеката ќе биде многу побрзо и овој начин не е деструктивен за датотеката. Кеширањето на операцијата на запишување претставува проблем и ефикасно единствено при групирање на запишувањата во сегменти, но ја намалува доверливоста затоа што можат да се појават логички оштетувања на системот на датотеките и губење на податоците. Затоа оваа техника се користи во комбинација со дневник на трансакции.

Постојат 4 правци на развој на кеширање на дискот:

1. Поделба на кеш меморијата на повеќе логички целини по различни критериуми и тоа на најмалку три функционални делови:
    a. Податочен кеш (*англ. data cache*), ги содржи скоро користените делови на датотеките, битни параметри се големината и алгоритамот за доделување на блоковите. Големината на софтверскиот кеш расне кога ќе се намали бројот на процесите и се намалува со зголемувањето на оптоварувањето на системот.
    b. Кеш на метатподатоци, се врши кеширање на метаподатоците кои се мали но значајни. Подрачјето на метаподатоците е многу различно по структура од самите податоци, па кешот се раздвојува на кеш за податоци и за метаподатоците.

c. Кеш на директориумски блокови, за да се пристапи кон одредена датотека потребно е да се пристапи кон директориумски блок кој содржи FCB за таа датотека, а еден директориумски блок содржи FCB за повеќе датотеки. Се покажало дека е многу корисно да се воведе нова компонента на кешот кој ќе ги содржи до скоро користените директориумски блокови.

2. Методи на предиктивно читање (*read-ahead*), покрај бараните податоци, во кешот се запишуваат и предиктивни податоци. Предикцијата се изведува на ниво на дел или цела патека од дискот и се комбинира со софтверскиот кеш при што податоците не се дуплираат а може да се користат и од двата кеша.

3. Методи за размена на податоците во кешот (*англ. cache replacement*). Размена на податоците во кешот се применува на различни начини. Се користат варијанти на LRU алгоритамот, дополнителен помошен кеш, каде што се им се дава уште една шанса на податоците кои треба да се отфрлат од кешот. Покрај тоа може да се користат и следниве услови за отфрлање на блоковите од кешот, а тоа е фреквенцијата на користење на блоковите во кешот. Еден од најдобрите алгоритами е LRFU (*Least Recently - Frequently Used*) со кој се комбинираат два параметри – скоро користење и честотата на користењето.

4. Методи за одложено запишување (*англ. write-back*), најчесто се користи техника за накнадно запишување, при што се користи техниката за водење на дневникот на трансакции (*англ. journaling).* Секое запишување на дискот се прифаќа во баферот на дискот, а подоцна се запишува на дискот. Покрај зголемени перформанси носи и ризик од губење на податоците, затоа се користи трајна RAM меморија или некој друг вид на меморија каде податоците може да го преживеат недостатокот на напојување.

5. Техника на празнење на баферот, типична за UNIX каде кешот функционира како обичен *write-back* кеш, а на секои 30 секунди се извршува процес за синхронизација со кој се запишува на дискот сите блокови од кешот кои се измениле.

6. Запишување по затворање (*англ. write on close*), запишувањето на дискот од кешот се врши кога процесот ќе ја затвори датотеката.

7. LFS (*Log Structured File System*), целиот диск се води како дневник, се дели на големи сегменти каде што се запишуваат датотеките и метаподатоците.

8. *Soft Updates*, оваа техника гарантира исправен редослед на операциите во подрачјето на метаподатоците, така што во случај на оштетувањето настануваат малку грешки и лесно се исправаат.

9. Техника на водење на дневник на трансакции (*англ. journaling*), овозможува управување со баферот за запишување, сите барања за запишувањето се чуваат во баферот на дневникот. Содржината на баферот може да се запише на дискот (во дневникот) кога тој ќе се наполни. Водењето на дневникот на трансакции овозможува брзо и квалитетно опоравување на системот на датотеките, со минимални оштетувања во случајот на хаварија.

9.7. Примери за системи на датотеки

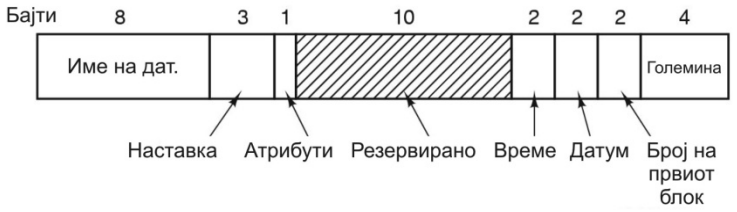
9.7.1. MS-DOS систем на датотеки

Системот на датотеки MS-DOS употребува имиња на датотеки по шема 8 + 3 знаци со ограничување сите тие да бидат со големи букви. Првата верзија (MS-DOS 1.0) беше ограничена дури и на еден директориум, меѓутоа, започнувајќи со MS-DOS 2.0, се внесоа големи подобрувања преку примена на хиерархиски систем на датотеки со кој директориумите може неограничено да се вгнездуваат. Системот формира стебло започнувајќи од коренскиот директориум (*root*). Заедничко за различни апликациски програми е да започнат со креирање на поддиректориум во главниот "root" директориум и таму да ги сместат сите датотеки (или во поддиректориумите што произлегуваат од него) со што се избегнуваат конфликти. Бидејќи самите директориуми се зачувани како датотеки, не постојат ограничувања на бројот на директориуми или датотеки кои што можат да бидат креирани. Во MS-DOS не постои концепт за различни корисници и тековниот корисник има неограничен пристап до сите датотеки. При процесот за читање на датотека, прво се создава системски повик со цел да добие контрола врз неа. Системски повик специфицира патека, која може да биде апсолутна или релативна во однос на моменталниот активен директориум. Патеката се изминува компонента по компонента се додека не се пронајде крајниот директориум и се прочита во меморијата, по што следи барањето за отворање на датотеката.

Иако MS-DOS директориумите имаат променлива големина, тие користат фиксен датотечен запис со должина од 32 бајти. Форматот на MS-DOS директориумски запис е прикажан на Слика 9.24. Тој го содржи името на датотеката, атрибутите, датата и времето на креирање, почетниот блок и вкупната големина на датотеката. Имињата на датотеката пократки од 8 + 3 знака се порамнети лево и пополнети со празнини од десно, во секое поле одделно.

Полето на атрибути (*англ. attributes*) содржи битови со кои се одредува дали датотеката е може само да се чита (*read-only*), дали треба да биде архивирана, дали е скриена или е системска датотека. Датотеките со поставен „read-only" бит не можат да бидат користени за запишување на податоци. Битот за

архивирање нема функција во самиот оперативен систем, туку им се овозможува на кориснички програми за правење на резервни копии (*англ. backup*) да ја демаркираат датотеката по нејзиното архивирање или да ја означат за архивирање по нејзината измена. Битот за сокривање на датотеката може да биде поставен за да се оневозможи нејзиното појавување при операцијата на листање на директориумот. Директориумскиот запис исто така ја содржи датата и времето кога е креирана датотеката или последен пат изменета. Претставувањето на времето е со прецизност од ±2 секунди затоа што е зачувано во 16-битно поле кое што може да зачуваат вкупно 65,536 вредности (а еден денот содржи 86,400 уникатни секунди). Временското поле е поделено во секунди (5 бита), минути (6 бита) и часови (5 бита). Датата се брои во денови користејќи три подполиња: ден (5 бита), месец (4 бита) и година - 1980 (7 бита). Со 7-битен број за годината и времето почнувајќи од 1980, најголемата вредност која може да биде доделена е 2107.



Слика 9.24: MS-DOS директориумски запис

MS-DOS за големината на датотеката користи 32-битен број и теоретски, датотеките можат да бидат со максимална големина и до 4 GB, меѓутоа од други причини се ограничува до 2 GB. MS-DOS води сметка за блоковите на датотеките преку табела за доделување на датотеки во главната меморија. Директориумскиот запис го содржи бројот на првиот блок на датотеката. Овој број се користи како индекс во табела на доделување на датотеките со 64K записи (FAT) која се наоѓа во главната меморија. Следејќи го поврзувањето (како што беше претходно објаснето) може да се пронајдат сите блокови во состав на една датотека.

FAT системот на датотеки се појавува во три верзии за MS-DOS: FAT-12, FAT-16 и FAT-32, во зависност од бројот на битовите со кои е опишана една адреса од дискот. За сите тие варијанти големината на блокот на дискот може да биде поставена на повеќе од 512 бајти, со користење на системот од дозволени големини на блокови (*англ. cluster sizes*). Првата верзија на MS-DOS користеше FAT-12 со 512-бајтни блокови, давајќи максимална големина на партиција од 2^{12} x 512 бајти, каде максималната големина на партиција на дискот изнесува околу 2 MB, а големината на FAT табелата во главната меморија изнесува 4096 записи со по 2 бајти. Со користењето на 12-битна табела запишувањето е многу бавно и овој систем бил соодветен во времето кога се користеле само флопи дискети, но по појавата на хард дисковите стана проблематично користење на мали блокови. Проблемот се реши со тоа што

биле дозволени и дополнителни големини на блокови од 1 KB, 2 KB и 4 KB, со што е запазена структурата и големината на FAT-12 табелата, а големината на диск партициите да достигне и до 16 MB. Бидејќи MS-DOS поддржува четири партиции по диск, новиот FAT-12 систем на датотеки работи со дискови со големина со 64 MB.

Подоцна беше претставен и системот на датотеки кој користел FAT-16 со 16-битни диск покажувачи и големини на блокови од 8, 16 и 32 KB. FAT-16 табелата во главната меморија зафаќа 128 KB. Најголемата диск партиција поддржана од FAT-16 е 2 GB (16 записи, секој со големина од 32 KB) и најголемиот диск од 8 GB т.е. четири партиции секоја од по 2 GB.

Големина на блокот	FAT-12	FAT-16	FAT-32
0.5KB	2MB		
1KB	4MB		
2KB	8MB	128MB	
4KB	16MB	256MB	1TB
8KB		512MB	2TB
16KB		1024MB	2TB
32KB		2048MB	2TB

Слика 9.25: Максимална големина на партиција за различни големини на блокови, празните секции претставуваат недозволени комбинации.

Во втората дистрибуција на Windows 95, беше претставен FAT-32 системот на датотеки, кој користел 28-битни диск адреси, а верзијата на MS-DOS поставена под Windows 95 беше прилагодена да го поддржува FAT-32. Во овој систем, партициите теоретски можат да бидат со големина од 2^{28} x 2^{15} бајти, но се всушност ограничени до 2 TB (2048 GB) затоа што системот води сметка за големините на партициите во 512-бајтни сектори користејќи 32-битен број а 2^{9} x 2^{32} изнесува 2 TB. Максималната големина на партицијата за различни големини на блокови и сите три FAT типа се прикажани на Слика 9.25.

FAT-32 систем на датотеки има уште две други предности пред FAT-16. Прво, диск од 8 GB што користи FAT-32 може да биде единична партиција. Користејќи FAT-16, мора да биде поделена на четири партиции. Друга предност е тоа што за дадена големина на партиција, може да се користи помала големина на блок. На пример, за партиција од 2 GB, FAT-16 мора да користи 32 KB - ни блокови. Во спротивно, со само 64 K достапни адреси на диск, не може да ја покрие опфати целата партиција. Тоа не е случај со FAT-32 кој што може, на пример, да користи 4 KB - ни блокови за партиција од 2 GB. MS-DOS го користи FAT за да се врши управување со слободните блокови на дискот. Блокот е слободен е обележан (маркиран) со посебна ознака. Кога MS-DOS има потреба од нов блок на дискот, врши пребарување на FAT табелата за запис кој ја содржи ознаката и според тоа не се потребни битмапи или слободни листи.

9.7.2. Windows 98 систем на датотеки

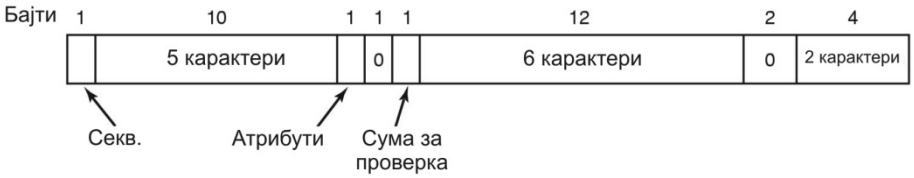
Вистинската дистрибуција на Windows 95 го користеше MS-DOS системот на датотеки вклучувајќи ја и употребата на 8 + 3 знаковни датотечни имиња и FAT-12 и FAT-16. Во втората дистрибуција на Windows 95, беше дозволена употреба на имиња на датотеки со должина поголема од 8 + 3 карактери. И долгите имиња на датотеките и FAT-32 се користени во Windows 98 во истата форма како и кај втората дистрибуција на Windows 95. Еден начин на претставување на долги имиња да датотеки би бил преку креирање на нова директориумска структура, но со тоа би се изгубила компатибилноста со постарите системи. Датотеките кои се креирани под Windows 98 оперативниот систем треба да бидат достапни и од постарите верзии како што е на пример Windows 3.11. Затоа директориумската структура на Windows 98 требда да биде компатибилна со директориумската структура на MS-DOS, која претставува листа од 32-бајтни записи како што е прикажано на Слика 9.26.

Слика 9.26: Проширениот MS-DOS директориумски запис користен во Windows 98

Додадени се пет нови полиња каде што се користат 10 - те неискористени бајти. NT полето обезбедува компатибилност со Windows NT со цел прикажување на датотечните имиња во правилен редослед (во MS-DOS сите датотечни имиња се со големи букви). „*Sec"* полето го решава проблемот за чување на времето од денот во 16-битно поле. Тој обезбедува дополнителни битови така што се постигнува приказ на времето со точност до 10 msec. Полето "*Last access*" ја чува датата (но не и времето) кога последен пат е пристапено до датотеката. FAT-32 системот на датотеки 32 битни броеви на блокови, па е потребно дополнително 16-битно поле за чување на горните 16 бита од почетниот број на блокот.

Решението за поддршка на долгите имиња се состои во тоа што се назначуваат две имиња за секоја датотека: едно долго име за датотека (во *Unicode* за компатибилност со Windows NT) и име со 8 + 3 карактери за компатибилност со MS-DOS. Кон датотеките може да се пристапи преку било кое од двете имиња. Кога е креирана некоја датотека чие име не го задоволува правилото за именување кај MS-DOS, Windows 98 за него создава MS-DOS име според одреден алгоритам. Се земат првите шест карактери од името се конвертираат во големи букви, ако е потребно и се додава ~1 за да се креира основното име. Ако ова име веќе постои, тогаш се користи додатокот ~2 и т.н. Во тој контекст, празните места и дополнителните периоди се избришани, а одредени посебни

карактери се конвертирани во знакот „~“. Секоја датотека има MS-DOS форма на име на датотеката користејќи го директориумскиот формат прикажан на Слика 9.27. Ако некоја датотека исто така има долго име, тоа име е зачувано во еден или повеќе директориумски записи кои претходат на MS-DOS датотечното име. Секој “долго-именски” запис содржи максимум 13 (*Unicode*) карактери. Записите се зачувани во обратен редослед со почетокот на името на датотеката пред записот на MS-DOS Слика 9.27.



Слика 9.27: Запис (дел од) за долго датотечно име во Windows 98.

На Слика 9.28 е даден пример за користење на долги имиња, постои датотека со име “*The quick brown fox jumps over the lazy dog*” со должина од 42 карактери. MS-DOS името е THEQUI~1 и е зачувано во последниот запис.

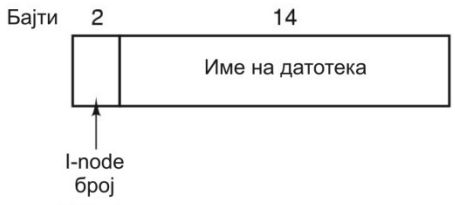
Слика 9.28: Пример за зачувување на долго име во Windows 98

Имплементацијата на FAT-32 датотечниот систем концептуално е слична со FAT-16 датотечниот систем. Сепак, наместо низа од 65,536 записи, постојат толку записи колку што се потребни за да се покрие делот од дискот со податоци. Ако првиот милион блокови се искористени, табелата концептуално има 1 милион записи, каде што само дел од записите се чуваат во главната меморија, а другите на дискот.

9.7.3. UNIX/Linux систем на датотеки

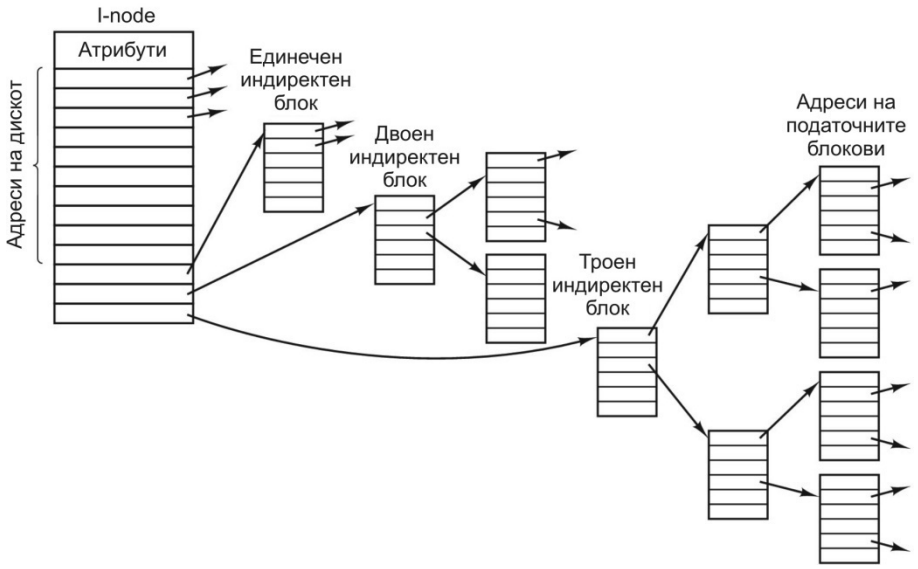
Дури и првичните верзии на UNIX поседуваа софистициран повеќе-кориснички систем на датотеки. Даден е пример за системот датотеки V7 за PDP - 11 кој што го прослави UNIX. Системот на датотеки има форма на

стебло кое започнува од коренскиот (*root*) директориум и со поставување на линкови и врски, формира насочен ацикличен граф. Имињата на датотеките можат да содржат 14 знаци (карактери), но исто така можат да содржат и било каков ASCII карактер освен знакот „/“ (врши одделување помеѓу компонентите на името на патеката) и NUL (затоа што се користи за дополнување на имињата пократки од 14 карактери) и има нумеричка вредност 0. UNIX директориумски запис содржи по еден запис за секоја датотека во тој директориум. Секој запис ја користи “i-node” шемата. Директориумски запис содржи само две полиња: името на датотеката (14 бајти) и бројот на “i-node” - от за таа датотека (2 бајти), како што е прикажано на Слика 9.29. Овие параметри го ограничуваат бројот на датотеките во системот на датотеки на 64 K. UNIX - овите “i-node” - ви содржат одредени атрибути. Тие се состојат од големина на датотеката, три временски податоци (креирање, последен пристап и последна модификација), сопственик, група, информација за заштита и пресметка на бројот на директориумски записи кои што посочуваат на “i-node” - от. Полето “*latter*” е потребно за време на поврзувањето со датотеките. Секој пат кога е креирана нова врска до “i-node” - от, за зголемува бројач во “i-node” - от. Кога некоја врска е отстранета, пресметката се намалува.



Слика 9.29: UNIX V7 директориумски запис

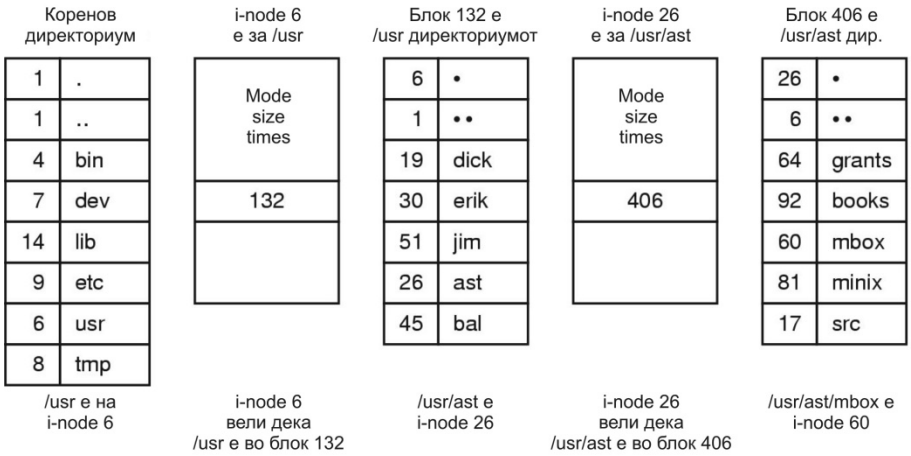
Кога бројачот ќе стигне до 0, “i-node” -от се ослободува и блоковите на дискот се враќаат во листата на слободни блокови. Управувањето со слободни блоковите на дискот се врши на тој начин кој овозможува управување со големи датотеки. Имено првите 10 адреси на дискот се чуваат во самиот “i-node”, така што за малите датотеки сите потребни информации се наоѓаат точно во јазелот кој што е повикан од дискот во главната меморија кога се врши отворање на датотеката. За поголемите датотеки, една од адресите во “i-node” - от е адресата на блок во дискот наречен единичен индиректен блок (*англ. single indirect block*). Овој блок содржи дополнителни адреси на блокови на дискот. Ако и оваа не е доволно, друга адреса во “i-node” - от, наречена двоен индиректен блок (*англ. double indirect block*), ги содржи адресите на блокот кој пак што содржи листа на единични индиректни блокови. Секој од овие единични индиректни блокови покажува на стотина податочни блокови. Дури и кога сето тоа не би било доволно, може да се користи и троен индиректен блок (*англ. triple indirect block*), Слика 9.30. Кога е отворена некоја датотека, системот на датотеки го превзема името на датотеката и ги лоцира неговите блокови на дискот.



Слика 9.30: UNIX - ов “*i*-node”

Пример за процесот е даден за датотека со име */usr/ast/mbox*. Прво, оперативниот систем го лоцира коренскиот директориум. Во UNIX ,“i-node” - от е сместен на фиксирано место на дискот. Од овој “i-node” се наоѓа коренскиот директориум кој што може да биде било каде на дискот иако во овој случај тоа е блокот 1. Потоа, се чита *root* директориумот и се бара првата компонента на патеката, *usr*. Лоцирањето на “i-node” - от преку неговиот број е следен чекор, затоа што секој од нив има фиксирана локација на дискот. Од овој “i-node”, системот го лоцира директориумот за */usr* и ја бара следната компонента, *ast*, во него. Кога ќе го најде записот за *ast*, тој го има “i-node” - от за директориумот */usr/ast*. Продолжувајќи од овој “i-node” тој може да го најде самиот директориум и да ја бара *mbox* датотеката. “i-node” - от за оваа датотека се вчитува во меморијата и се чува таму се додека датотеката не се затвори, Слика 9.31.

Релативните патеки се пребаруваат на ист начин како и апсолутните, со таа разлика што пребарувањето започнува од моменталниот директориум наместо од примарниот директориум. Секој директориум има записи за . и за .. кои што се ставени таму при креирањето на директориумот. Записот . го има бројот на “i-node” - от за моменталниот директориум, додека записот за .. го има бројот на “i-node” - от за родителскиот директориум. Според тоа, процедурата за барање на ../disk/prog.c едноставно го бара .. во работниот (моментален) директориум, го пронаоѓа бројот на “i-node” - от за родителот директориум и го пребарува тој директориум за disk. Не се потребни посебни механизми за управување на овие имиња. Што се однесува до директориумскиот систем, тие се обични ASCII стрингови исто како и сите останати имиња.



Слика 9.31: Чекорите на пребарување на /usr/ast/mbox

10. Дискови и секундарни меморији

За разлика од главната меморија, содржината на секундарната меморија не се губи по исклучувањето на компјутерскиот систем. Во секундарната меморија се сместени оперативниот систем, програмите и податоците кои се обработуваат. При инсталацијата и надградба на оперативниот систем, потребно е да се изврши администрирање на дисковите и системот на датотеки. Форматирањето на дисковите на најниско ниво, поделбата на партиции и форматирањето на системот на датотеки се изведува пред инсталацијата на оперативниот систем иако ако е потребно процедурата може да се повтори при додавање на нови дискови или ако се менуваат параметрите на системот на датотеки. Дискот се дели на партиции ако е неопходно на еден систем да се постават повеќе оперативни системи или со цел да се раздвојат системските податоци од корисничките со што се олеснува нивното архивирање и чување. Во оваа поглавје се опишани структурата, начинот на функционирање на хардверот на дисковите, како и алгоритамите за распоредување на барањата за работа со дисковите, со кои се намалува загубата на времето при работата со дисковите. Дадено е и објаснување на RAID структурите кои може да се искористат за формирање на стабилни системи за складирање.

10.1. Структура на секундарната меморија

Структурата на секундарната меморија, генерално гледано се состои од бројот на дисковите кои се директно прикачени на системот. Како и дисковите кои се делени преку брза локална компјутерска мрежа кои можат да бидат дел од други компјутерски системи или самостојни уреди во така наречена мрежа на складови (*SAN - Storage Area Network*). Дисковите во зависност од нивната сложеност да бидат поделени на едноставни и паметни (*англ. smart*) дискови. Овие последните се карактеризираат со способност за проценка на интегритетот на дискот преку следење на одреден број на параметри на дискот. Дали е надминат одреден праг на толеранција да се определи статусот на дискот како „исправен“ или „неисправен“, како и сокривање на интерната структура на дискот.

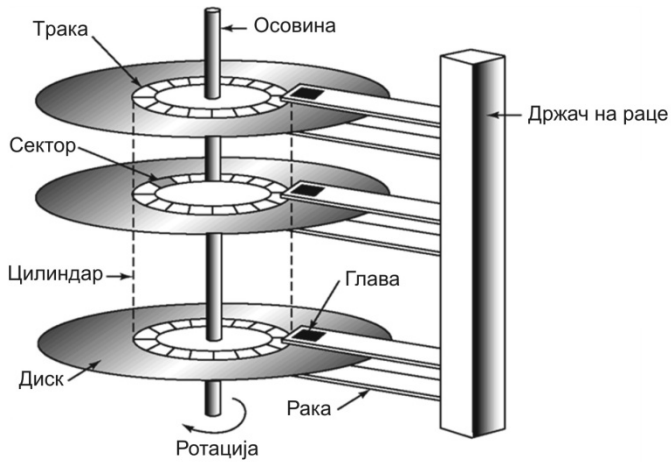
10.1.1. Интерна структура на дискот

Дисковите се среќаваат во повеќе различни видови. Најмногу се застапени магнетните дискови (тврд диск и „флопи“ диск). Тие се карактеризираат со фактот што читањето и запишувањето им е еднакво брзо, што ги прави идеални за изведба на секундарна меморија.

Диск уред се состои од кружни плочи кои се вртат околу заедничка оска. Површината на плочите се премачкани со магнетен материјал. Секоја површина си има своја глава за читање и пишување на податоците на и од магнетната плоча. Главите за читање и запишување се движат линеарно со помош на серво мотори со што со ротирањето на плочине на главите им е овозможен пристап кон сите делови на магнетната плоча.

Процесорот на компјутерот и дискот комуницираат преку контролерот на дискот (*англ. disk controller*). Контролери на различни дискови имаат ист интерфејс кон останатите делови од компјутерскиот систем, со што се обезбедува униформност во комуникацијата со компонентите и не е потребно директно познавање на механиката на дискот.

Површината на дискот е поделена на концентрични кругови – патеки (*англ. tracks*), а секоја патека е поделена на сектори (*англ. sectors*). На постарите видови дискови, сите патеки имаат ист број на сектори, додека кај новите, надворешните патеки заради должината на патеката може да имаат и повеќе сектори со што се обезбедува еднаква магнетна површина за сите сектори на дискот. Еден сектор може да содржи најмалку 512 бајти на податоци. Сите плочи од еден диск се подеднакво поделени на патеки и сектори, така што главите за читање и запишување на сите плочи во еден момент се наоѓаат над исти патеки, истите патеки од различни плочи формираат цилиндер (*англ. cylinder*). Датотеките кои не се наоѓаат во состав на еден цилиндер се фрагментирани, за нивното читање се внесува дополнително задоцнување заради движењето на главите.



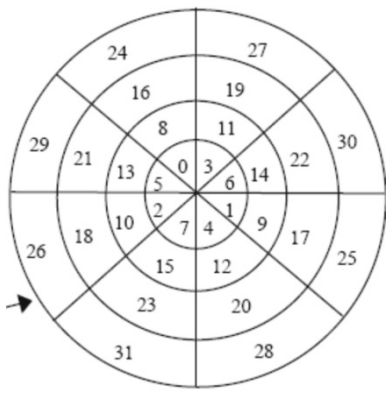
Слика 10.1: Внатрешен изглед на цврст диск

Карактеристиките на дискот произлегуваат од неговата геометрија, како што е капацитетот кој е определен од бројот на плочите, број на патеки/плоча, број на сектори/патека, големината на секторот. Овие податоци се запишани во

CMOS RAM и се читаат секој пат при вклучувањето на системот. Потоа адресирањето се врши со повикување на тродимензионалната адреса (глава, цилиндер, сектор).

Друга многу значајна карактеристика на дисковите е брзината на пристап, која се одредува со времето кое е потребно за да се постави главата на бараниот сектор за читање или запишување. Тоа време ги опфаќа поместувањето на главите (*seek*) до бараната патека, (изнесува 1-20 msec), ротација до бараниот сектор (брзината на ротацијата на дисковите се изразува во ротации по минута, како на пример 3600, 5400, 7200, 9600 RPM). Со која брзина ќе биде запишан или прочитан податокот зависи и од брзината на трансферот на податоците кој се изразува во MB/sec).

Надежност (*англ. reliability*) на дисковите е карактеристика со која се опишува временски период помеѓу два испади на дискот. Се означува како средно време на испад (*MTTF - Mean Time to Failure*) и се изразува во 1000-ци часови и обично вредноста одговара на период од повеќе години. Надежноста на секундарната меморија може значително да се подобри со користење на некој од RAID нивоата на комбинирање на повеќе дискови. Како и со користење на кодови за детекција и корекција на грешки - ECC код (*Error Correction Code*) кој се одредува за секој сектор и се споредува при операцијата на читање на податоците од секторот. Ако се појави случај каде што дискот може има лоши сектори, системот треба да води сметка за нив, или интерфејсот да ги маскира (SCSI). Читањето и запишување на дискот се прави по сектори кои се минимална податочна единица според нивната адреса (#surface, #track, #sector), додека *Smart disk* драјвовите ја сокриваат интерната структура на дискот и кај нив кон секторите се пристапува според еднозначна ознака (#sector). Од дискот не може да се читаат или пишуваат поединечни бајтови. Пристапот е случаен при што позиционирањето на главите (seek) бара одредено време, а ако секторите се читаат секвенцијално пристапот може да биде и неколку стотина пати побрз од случајниот пристап.



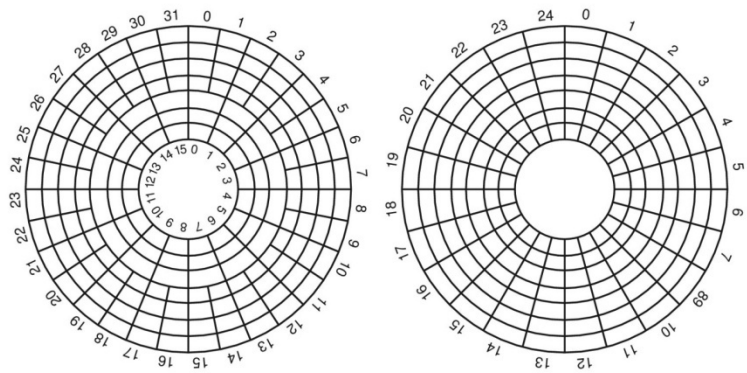
Слика 10.2: Логичка поставеност на секторите

Притоа контролерот на дискот управува со преносот на податоците при што физички соседните сектори не можат да се читаат одеднаш (преполнување на трансферот). Затоа логички соседните сектори обично се физички оддалечени (секој втор, трет и т.н.) за да по завршување на трансферот од страна на контролерот главата пристапува кон следниот сектор. Различните видови на оптички дискови (CD-ROM, CD-R/RW, DVD и други) се исто така важни за складирање на програми, податоци и сл. Во продолжението е даден опис на хардверот на овие видови на магнетни и оптички дискови. Во табелата 10.1 се претставени параметри на оригинален IBM 360 KB флопи диск и параметри на цврст диск за да се покаже развојот на магнетните дискови во периодот од две децении.

Табела 10.1. Параметри за дисковите IBM 360 KB и WD 18300

ПАРАМЕТРИ	IBM 360-KB ФЛОПИ ДИСК	WD 18300 ЦВРСТ ДИСК
Број на цилиндари	40	10601
Патеки на еден цилиндар	2	12
Сектори на една патека	9	281 (avg)
Сектори на диск	720	35742000
Бајти на еден сектор	5.12	512
Капацитет на дискот	360 KB	18.3 GB
Време на барање (соседни цилиндари)	6 msec	0.8 msec
Време на барање (среден случај)	77 msec	6.9 msec
Време на ротација	200msec	8.33 sec
Стопирање на моторот /почетно време	250msec	20 msec
Време на пренос на еден сектор	22 msec	17 msec

Како што беше спомнато кај постарите дискови, бројот на сектори на една патека е иста за сите цилиндари. Модерните дискови се поделени на зони со повеќе сектори во надворешните отколку во внатрешните зони.



Слика 10.3: а) Физичка геометрија на диск со две зони,
б) можна виртуелна геометрија на овој диск.

На Слика 10.3 а) е илустриран мал диск со две зони. Надворешната зона има 32 сектори на една патека: внатрешната има 16 сектори на една патека. Вистинскиот диск како што е WD 18300, најчесто има 16 зони, со број на сектори со пораст од околу 4% во секоја зона од внатрешната во надворешната зона. За да се сокријат деталите за тоа колку сектори има секоја патека, модерните дискови имаат виртуелна геометрија претставена во оперативниот систем. Можна виртуелна геометрија за физичкиот диск од Слика 10.3 а) е прикажана на Слика 10.3 б). И во двата случаи дискот има 192 сектори, единствено јавното уредување е различно од оној на вистинскиот.

10.1.2. Современи диск уреди

Развојот на механичките делови на современите диск уреди е насочен кон подобрување на перформансите. Во однос на механиката на дискот, може да се извршат следниве подобрувања како што се скратување на времето за позиционирање и зголемувањето на брзината на ротација, со што се намалува ротационото доцнење а со тоа се зголемува и брзината на читање/запишување.

Во однос на обликот на податоците и густината на магнетниот медиум може да се подобрат перформансите преку зголемувањето на густината со што се зголемува и капацитетот на дискот. Може да се користи и зонска техника (*англ. Bit Zone Recording*), каде што цилиндарите се групираат во зони со иста густина (*BZR*). Со тоа се зголемува и капацитетот на дискот и брзината но се усложнува обликот на податоците на дискот а со тоа и контролерот. Алтернативни сектори за управување со дефектите се користат за замена на оштетените сектори на дискот.

Во однос на управувачката електрониката на дисковите постојано има подобрување на перформансите и на функционалноста со употреба на помоќни процесори (ранг на i386) на самата плоча на контролерот како и сопствена RAM меморија.

Кеширање на диск уредот служи за усогласување на брзините на пренос на податоците од дискот или на дискот и интерфејсот на дискот. Кеширањето може да се изведува и на ниво на дискот. Кеш меморијата може да се подели на сегменти чија големина одговара на големината на патеката или нејзин дел. Работата на кешот на дискот се базира врз техника позната како предиктивно читање или „*read ahead*“ читање. По читањето на бараните податоци, дискот продолжува да чита и податоци кои не се барани експлицитно од него. При секој пристап кон дискот првин се проверува кеш меморијата, при операцијата на читање може да се случи полн или делумен погодок или тотално промашување. Во случај на запишувањето, доколку дојде до погодок во кешот се врши обновување на податоците или им се поништува валидноста. Кога кешот е пополнет, потребно е да се изврши замена на страницата и притоа се користи некој од LRU алгоритамите. Со користењето на кеширањето на ниво

на дискот значително се подобруваат перформансите преку заобиколување на бавниот процес на позиционирање на главите.

Пресликување на логички адреси во физички врши придружување на логичкиот број на блокот во физичка адреса на секторот и се врши со доделување на ознака 0 на првиот сектори на првиот диск и се оди до крај додека не се пополни капацитетот на дискот (*LBN - >PBN*). Мапирањето на почетокот се врши фабрички земајќи ги предвид преплетувањето и изместувањето (*англ. skew*) на секторите на патеката, но при експлоатацијата на дискот можно е да се појави потреба од повторно пресликување заради замена на дефектните сектори со резервни. Начинот на распоредување на командите во редицата за чекање, има големо влијание врз оптималната работа на дискот, затоа што интерниот распоредувач на барањата има најдобар преглед на состојбата на механичките и баферските компоненти.

10.1.3. ATA и SCSI дискови

ATA и SCSI дисковите (вклучително и SATA) се најкористените класи на дискови на персоналните компјутери. ATA (*Advanced Technology Attachment*) е познат и како IDE (*Integrated Drive Electronics*). Овие уреди се карактеризираат со капацитети до 160 GB. Брзина од 5400 до 7600 RPM, како и брзина на трансфер во распон од 33-133 Mbps. Секој ATA контролер има два канали еден примарен и еден секундарен каде што на секој канал има по два уреди (*master* и *slave*), од кои само еден е активен во еден момент. Уредите на различни канали можат истовремено да испраќаат и примат податоци. Секој ATA уреди има прекинувачи (*англ. jumper*) со кои се одредува во која положба ќе работи (*master* или *slave*) при приклучувањето на системот.

SCSI (*Small Computer System Interface*) претставува професионален интерфејс за поврзување на повеќе типови на уреди (CD, DVD, скенери, маг. ленти). Контролерот не е интегриран со матичната плоча и одвоено се вградува во системот. Електрониката на SCSI контролерот е посложена и постигнува поголеми брзини од ATA (320 Mbps, ultra SCSI 2560 Mbps) и на контролерот можат да се прикачат 7-15 уреди. Тие се поврзуваат според приоритети зададени преку идентификационен број. Уредот со ID број 0 има највисок приоритет и треба да му се додели на системскиот диск, бројот 7 е резервиран за самиот контролер, а уредот со број 15 има најмал приоритет.

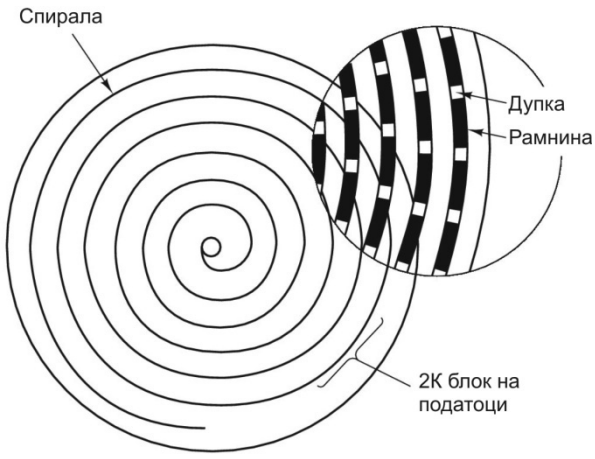
SATA претставува контролер кој е наследник на ATA и се карактеризира и со поголема брзина од 1200-2400 Mbps (*eSATA*), со потенки кабли со што се овозможува подобро ладење во компјутерскиот систем како и можност за замена на дисковите без исклучување на напојувањето на системот (*англ. hot swapping*). SCSI од друга страна има поголема цена, но и можност за истовремено приклучување на повеќе уреди за разлика од SATA каде што треба да се користи дополнителен уред.

10.2. Оптички дискови

10.2.1. CD-ROM

Оптичките дискови станале сè попопуларни за складирање и пренос на податоците во однос на магнетните дискови заради поголемата густина на запис во однос на магнетните (флопи) дискови стануваат. Овие дискови се познати како CD (*Compact Disc*). Првата генерација оптички дискови е претставена во 1980 од страна на Philips заедно со компанијата Sony. Според стандардот сите дискови имаат дијаметар од 120 mm, дебелина од 1.2 mm и отвор на средината со дијаметар од 15 mm. Аудио оптичките дискови биле првите оптички медиуми за складирање кои постигнале комерцијален успех.

Оптичките дискови се составени од алуминиумска плоча на која податоците се запишуваат така што со ласерски зрак се прават вдлабнатини со 0.8 микрони дијаметар со различна длабочина. Диск единицата изведува читање на податоци со ласер при што се користи 0.78 бранова должина на ласерот. Кога зракот ќе наиде на дупката се добива отклонување на зракот од фото сензорот и се детектира нула или единица. Спиралата е долга 5,6 км и за правилно читање потребно е да се одржува линеарна брзина од 530 до 200 rpm. Спиралата е илустрирана на Слика 10.4.



Слика 10.4: Структура на снимање на CD

Во 1984 Philips и Sony го препознале потенцијалот на употребата на CD-то за складирање на дигитални податоци и предлагаат стандард за CD-ROM (*Compact Disc – Read Only Memory*).

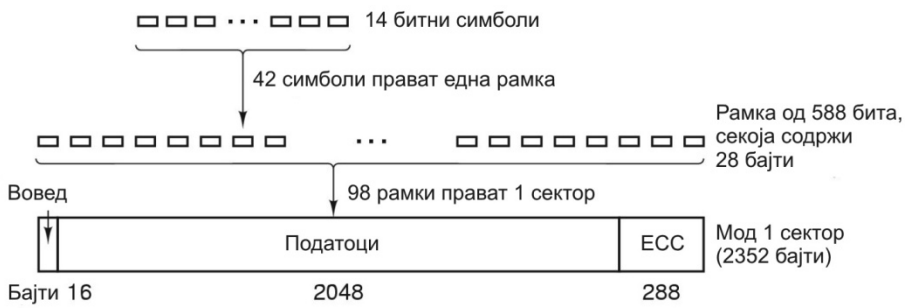
Во стандардот е опишан и начинот на форматирањето на компјутерските податоци. Основниот формат на CD-ROM се состои од кодирање на секој бајт

во 14-битен симбол. Што е доволно за Хамингово кодирање на 8-битен и остаток од 2 бита. Всушност е искористен многу пософистициран систем за кодирање а читањето од 14 во 8 бита се изведува хардверски со "look up" табела. На повисоко ниво, група од 42 последователни симболи формираат 588 битна рамка. Секоја рамка има 192 бита на податоци (24 бајти), а останатите 396 бита се користат за корекција на грешка и контрола. Оваа шема е идентична и за аудио CD и CD-ROM дисковите. Уште повеќе, се врши групирање на 98 рамки во сектори од CD-ROM-от, на начин како што е прикажано на Слика 10.5.

Секој CD-ROM сектор започнува со 16 бајтна преамбула, од кои првите 12 се 00FFFFFFFFFFFFFFFFFFFF00 (хексадекадно), за да му се овозможи на читачот да го препознае почетокот на CD-ROM секторот. Следните 3 бајти го содржат бројот на секторот, кој е неопходен бидејќи пребарувањето на CD-ROM-от (во спиралата) е многу потешко отколку кај магнетен диск со еднакви концентрични патеки. Последниот бајт од преамбулата е начинот (модот).

Стандардот дефинира два начини. Првиот начин прикажан на Слика 10.5 има 16 бајтна преамбула, 2048 бајти податоци и 288 бајти за код за корекција на грешка. Вториот мод ги комбинира податоците и ECC полињата во 2336 бајтни податочни полиња за оние апликации за кои е потребна корекција на грешка, како што се аудио и видео. За да се постигне поголема доверливост, се користат три одделни шеми за корекција на грешки: во опсег на симбол, во опсег на рамка и во опсег на CD-ROM сектор.

Единичните бит грешки се исправаат на најниско ниво, кратките „burst" грешки се исправаат ниво на рамка, а другите грешки се откриваат и исправаат на ниво на сектор.



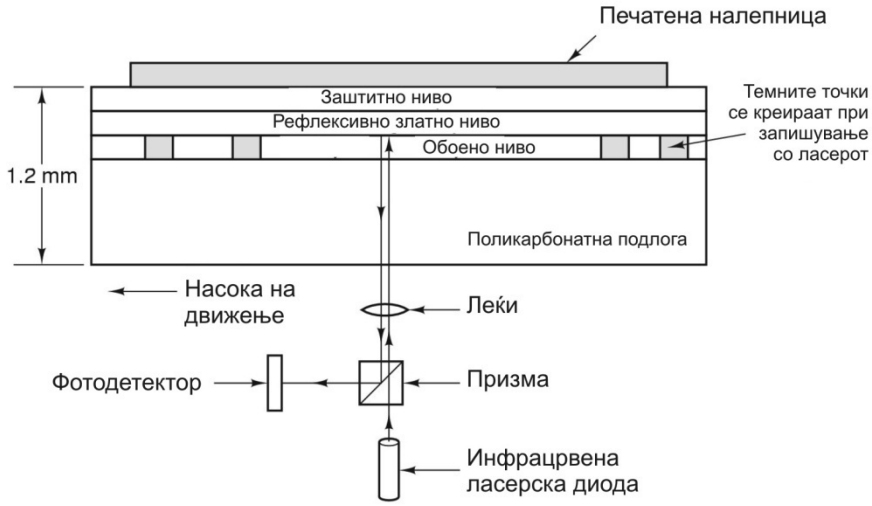
Слика 10.5: Логичка организација на податоците на CD-ROM

Едно-брзинскиот CD-ROM читач работи со брзина од 75 сектори/сек што дава вредност на податоци од 153,600 бајти/сек во мод 1 и 175,200 бајти/сек. во мод 2. Дво-брзинските читачи се два пати побрзи. Според тоа 40х читачот може да дава податоци со 40 х 153,600 бајти/сек. Стандардно аудио CD има место за

складирање на 74 минути за музика, што ако се искористи во мод 1 податоците даваат капацитет од 681.984.000 бајти или 650MB (1 МБ е 1,048,575 бајти).

10.2.2. CD-R

CD-R се оптички дискови кај кои може само еднаш да се изврши запишување на податоци, кои подоцна не можат да се избришат. Физички, CD-R дисковите се составени од поликарбонатни празни дискови како што се и кај CD-ROM-овите освен што кај CD-R постои 0.6 mm широка вдлабнатина за насочување ласерот за пишување. Вдлабнатината има синусоидно отстапување од 0.3 mm со фреквенција од 22.05 kHz, со што се постигнува следење на брзината на ротацијата на дискот и прилагодување по потреба. CD-R дисковите изгледаат како обични CD-ROM-ови, освен што се златно обоени наместо сребрено. Кај CD-R дисковите разликата во рефлексивноста се постигнува преку додавање на ниво на премачкување помеѓу поликарбонатот и рефлективното златно ниво, како што е претставено на Слика 10.6. Искористени се два вида на премази: „*cyanine*“ која е зелена, „*pthalocyanine*“ која е жолтопортокалова.



Слика 10.6: Напречен пресек на CD-R

За пишување, CD-R ласерот користи поголема моќност (8-16 mW). Кога при запишувањето ласерот ќе погоди точка од бојата, се создава темна дамка заради промената на молекуларната структура. За читањето на повторно читање на податоците се користи помала моќност на ласерот (на 0.5 mW), тогаш фотодетекторот може да ја препознае разликата помеѓу црните дамки каде што премазот бил погоден од ласерот и транспарентните области кои останеле непроменети.

10.2.3. CD-RW

CD-RW дисковите имаат можност за запишување, но и бришење на запишаните податоци. CD-RW (*CD-ReWritable*) има исти димензии како како и CD-R. Сепак, наместо премаз помеѓу рефлективното ниво и носачот од поликарбонати од *„cyanine“* или *„pthalocuanine“*, се користи легура од сребро, индиум, антимонид и телуриум. Оваа легура има две стабилни состојби: кристална и аморфна, кои се карактеризираат со различен степен на рефлексивност.

CD-RW читачот користи ласери со три различни степени на моќност и тоа со висока, средна и слаба. Со висока моќност, ласерот ја топи легурата претворајќи ја од високо рефлексивна кристална состојба во слабо рефлексивна аморфна состојба што претставува вдлабнатина. Со средна моќност, легурата се топи и се претвора во негова природна кристална состојба за да стане повторно поле. Со слаба моќност, состојбата на материјата се детектира за читање, но не се изведува фаза на премин од една во друга состојба.

10.2.4. DVD

DVD (*Digital Versatile Disk*) ја користи истата генерална структура како кај CD оптичките медиуми. Она по што се разликуваат е користењето на:

1. Помали отвори (0.4 микрони наспроти 0.8 микрони за CD).
2. Потесна спирала (0.74 микрони помеѓу патеките наспроти 1.6 микрони за CD).
3. Црвен ласер (0.65 микрони наспроти 0.78 микрони за CD).

Заедно, овие подобрувања го зголемуваат капацитетот на вредност од 4.7 GB. DVD читач со брзина 1 x работи со брзина од 1.4 MB/sec (наспроти 150 KB/sec за CDs).

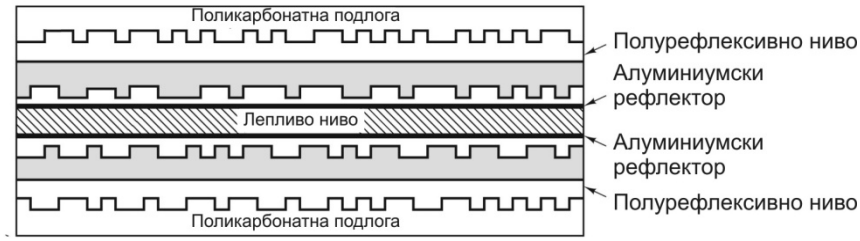
Еден 4.7 GB DVD диск може да содржи 133 минути квалитетно видео со резолуција (720 x 480), како и звук до осум јазици и повеќе од 32 преводи. 99% од сите филмови во Холивуд се прават под 133 минути. Покрај овој основен формат се дефинирани уште три други формати:

1. Едно-страно, едно-нивовско (4.7 GB).
2. Едно-страно, дво-нивовско (8.5 GB).
3. Дво-страно, едно-нивовско (9.4 GB).
4. Дво-страно, дво-нивовско (17 GB).

Дво-нивовската технологија има рефлексивно ниво на дното, а на врвот полурефлексивно ниво. Во зависност од тоа каде е фокусиран ласерот, се очитува едното или другото ниво. На пониското ниво се потребни

незначително поголеми вдлабнатини и полиња за посигурно читање, па затоа неговиот капацитет е незначително помал отколку на повисокото ниво.

Дво-страните дисковите се направени со комбинирање на два 0.6-mm едно-страни дискови. Структурата на дво-страно и дво-нивовско DVD е илустрирано на Слика 10.7.



Слика 10.7: Дво-страно, дво-нивовско DVD

10.3. Подготовка на дискот

При производство на цврстите дискови, плочите се премачкуваат со метален оксид кој може да се намагнетизира, притоа дискот не содржи никакви податоци ниту форматирање. Пред да може да се користи за запишување и читање врз дискот треба да се изведе форматирање на ниско ниво (*англ. Low level formatting*) или да се направи физичко форматирање (*англ. Physical formatting*). Тој процес се изведува од страна на производителот и притоа дискот се дели на патеки и сектори. Потоа може да се изврши партиционирање од страна на корисникот каде што секоја партиција се однесува како засебен диск. И на крајот се изведува и логичко форматирање (*англ. Logical formatting*) со што се создава системот на датотеките, FAT, I-node, NTFS, мапа на слободен и алоциран простор, почетен празен директориум. За да на крајот се инсталира и самиот оперативен систем на дискот треба да се постави на фиксна локација така наречен полнач (*англ. loader*) кој се повикува од страна на полнач кој се наоѓа во ROM. На тој начин се определува кој диск е системски и од кој се врши boot-ирање на системот. Форматот на секторот е прикажан на Слика 10.8.

Вовед	Податоци	ECC

Слика 10.8: Формат на диск сектор

Преамбулата започнува со одредена низ на битови со која се овозможува на контролерот да го препознае почетокот на секторот. Исто така во преамбулата се содржи и бројот на цилиндарот и бројот на секторот како и други информации. Големината на делот за овие податоци е одредена со форматирањето на ниско ниво. Повеќето дискови користат 512 бајтни сектори.

Полето ECC содржи дополнителна информација која може да се користи за детекција и корекција на грешки при читање. Големината и содржината на ова поле се разликува од производител до производител во зависност во која мерка треба да се постигне доверливост на читањето. Обично за должината на ECC полето се зема вредност од 16 бајти. Покрај тоа сите тврди дискови имаат одреден број на резервни сектори кои се користат за замена на сектори кои се со дефект. Позицијата на сектор 0 на секоја патека е поместен од претходната патека. Оваа поместување се нарекува и извртување на цилиндарот и се прави за да ги подобри перформансите. Основната идеја е да се овозможи на дискот во една продолжена операција на читање да се изминат повеќе патеки без да се изгубат податоците. Видот на проблемот може да се види од Слика 10.9.

Да се претпостави дека за некое барање потребни се 18 сектори почнувајќи од сектор 0 на внатрешна патека. За читањето на првите 16 сектори е потребна една ротација на дискот, но за да се задоволи барањето е потребно поместување на главата една патека кон надвор за да се стигне до 17-от сектор. Но додека главата се поместила за една патека, секторот 0 е изминат и се наоѓа зад главата и потребна е една цела ротација за да се стигне до него. Овој проблем е елиминиран со поместувањето на секторите како што е прикажано на Слика 10.9.

Слика 10.9: Илустрација на извртување на цилиндарите

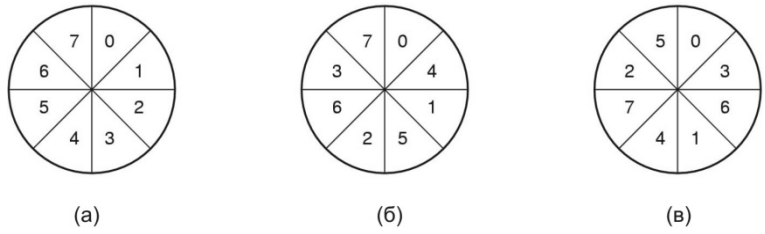
Обемот на извртувањето на цилиндарите зависи од геометријата на уредот. На пример: уред со 10.000 RPM (вртежи во минута) ротира за 6 msec. Ако патеката содржи 300 сектори, новиот сектор преминува покрај главата секои 20 μsec. Ако времето на барање од патека до патека е 800 μsec, 40 сектори ќе

поминат во текот на барањето, па извртеноста на цилиндарот би била 40 сектори, наместо трите сектори прикажани на Слика 10.9. Треба да се спомене и дека измената на главите исто така одзема определено време, па затоа постои и извртување на главите (*англ. head skew*) исто како и кај цилиндарите, но во помал обем.

Како резултат на форматирањето на ниско ниво, капацитетот на дискот е намален, во зависност од големината на преамбулата, празниот простор во меѓу секторите и ECC, како и бројот на зафатените резервни сектори. Често форматираниот капацитет е 20% помал од не форматираниот капацитет. Тоа може да предизвика недоразбирање бидејќи некои производувачи објавуваат капацитети на не форматирани дискови за да ги направат нивните уреди да изгледаат поголеми отколку што навистина се.

Форматирањето влијае и врз перформансите. Ако диск со 10.000 RPM има 300 сектори на една патека со по 512 бајти, потребни се 6 msec да се прочитаат 153.600 бајти од патека со брзина на пренос на податоците од 24.4 MB/sec. Оваа брзина не може да се надмина без разлика каков интерфејс се користи, дури и ако е SCSI интерфејс со 80 MB/sec или 160 MB/sec.

За да се изврши непрекинато читање со оваа брзина потребен е голем бафер на контролерот. Ако се преполни баферот може да се промаши следниот сектор за читање па да е потребно да се чека цела ротација. Меѓутоа проблемот може да се надмине со нумерирање на секторите на преплетен начин (*англ. interleaving*). На Слика 10.10 а) е претставен обичниот начин на нумерирање. На Слика 10.10 б) е претставено единчено преплетување, кое му дава на контролерот простор „за дишење“ помеѓу последователни сектори со цел за тоа време да се пренесе содржината баферот во главната меморија. Ако процесот на копирање е многу бавен, тогаш потребено е двојно преплетување, претставено на Слика 10.10 в).



Слика 10.10: Нумерирање на секторите

Откако е направено форматирање на ниско ниво, дискот се дели на партиции. Логички, секоја партиција претставува посебен диск. Сектор 0 го содржи „*master boot*“ записот, кој содржи код за подигнување како и табелата на партициите. Табелата на партиции го дава стартниот сектор и големината на секоја партиција. Кај Pentium, табелата за партиции може да содржи четири партиции. Ако сите партиции се за Windows, тие ќе имаат ознаки C:, D:, E: и

F:. Ако три од нив се за Windows а една за UNIX, тогаш од Windows партициите ќе се викаат C:, D: и E: а првиот CD-ROM ќе биде F:. Од овие партиции една се означува како активна и од неа се подига системот.

Последниот чекор за подготвување на дискот е форматирање на секоја партиција на високо ниво. Со тоа се поставува *boot* блокот, листата на слободни блокови или битмапата, коренскиот директориум и празен систем на датотеки. Исто така се поставува и код во табелата на партиции со кој се покажува кој систем на датотеки се користи на која партиција. Од тој момент системот може да се подигне, при вклучувањето, првин се извршува BIOS и го чита главниот *boot* запис (*Master Boot Record – MBR*) и рутината скока на него. *Boot* програмата проверува која од партициите е активна (каде е поставен оперативниот систем) и го чита *boot* секторот на таа партиција и го извршува. *Boot* секторот содржи мала програма која го пребарува коренскиот директориум за одредени програми (или оперативниот систем или поголема „*bootstrap*“ програма) која се полни во меморијата и се извршува.

Слика 10.11: Организацијата на дискот
и системот на датотеки кај MS-DOS оперативниот систем

10.4. Справување со грешки

При производство на дисковите може да се случи појава на лоши сектори. Тоа се сектори од кои не може правилно да се прочита вредноста која претходно била запишана. Ако оштетувањето е многу мало, на пример, неколку бита, возможно е да се искористи лошиот сектор и да се дозволи на ECC да ја поправи грешката во секое време. Ако оштетувањето е поголемо, грешката не може да се поправи.

Постојат два основни пристапи за справување со лоши блокови: со помош на контролерот или да се остави оперативниот систем да се справи со нив. Пред

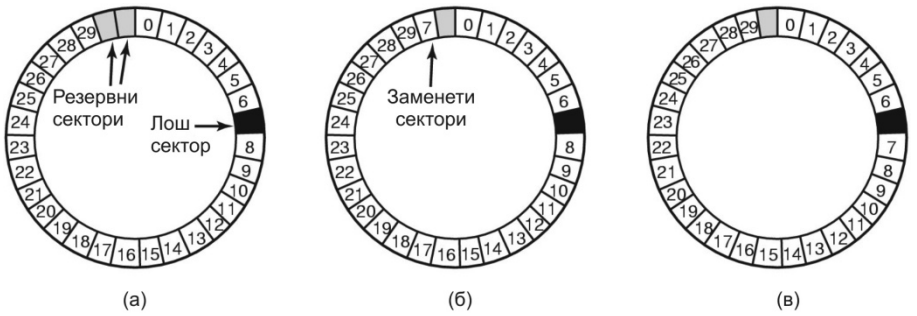
дискот да се транспортира од фабриките, тој се тестира и листата од лошите сектори е испишана во самиот диск.

За секој лош сектор, постои замена со еден од резервните и постојат два начини да се изврши таа замена. На Слика 10.12 а) е прикажана една патека на диск со 30 податочни сектори и два резервни сектори. Она што може контролерот да направи е да премести еден од резервните како сектор 7, како што е прикажано на Слика 10.12 б).

Друг начин е да се преместат на сите сектори за еден погоре, како на Слика 10.12 в). Во двата случаи потребно е контролерот да знае кој сектор е кој. Контролерот може да чува информација во своите интерни табели или може да ги препише информациите во преамбулите на секторите. Ако се препишуваат преамбулите, методот на Слика 10.12 в) побарува повеќе работа (потребно е да се препишат 23 преамбули), но дава подобри перформанси бидејќи на внатрешната патека може се уште да се чита во една ротација.

Исто така можат да се појават грешки и при нормална работа откако уредот ќе биде инсталиран. Првото нешто што може да се направи за да се избегне појава на грешка со која ECC не може да се справи, е обид за повторно читање. Некои грешки во читање се привремени, тие се случуваат заради честичките од прашина под главата кои ќе се отстранат при наредно читање.

Ако контролерот забележи дека грешката се повторува на определен сектор, може да го замени со резервен сектор пред тој целосно да се уништи. На овој начин, ниеден податок не е изгубен и оперативниот систем и корисникот нема да го забележат проблемот. Најчесто, методот на Слика 10.12 б) треба да се користи ако другите сектори можат да содржат податоци. Ако се користи методот на Слика 10.12 в) потребно е не само препишување на преамбулата, туку и копирање на сите податоци.



Слика 10.12: Диск со лоши сектори

Лошите сектори не се единствените извори на грешки. При операцијата за пристап кон патека можни се грешки предизвикани и од механички проблеми. Контролерот ја чува позицијата на носачот на главите. За извршување на барање (*англ. seek*) се испраќа серија од импулси на сервомоторот на раката,

еден импулс за еден цилиндар, да се помести раката на друг цилиндар. Кога раката ќе стигне до дестинацијата, контролерот го чита тековниот број на цилиндарот од преамбулата на следниот сектор. Ако раката е на погрешно место, тогаш се појавува грешка.

Многу контролери ги исправаат овие грешки автоматски, но многу флопи контролери поставуваат бит за грешка оставајќи го другото на уредот. Уредот со оваа грешка се справува со користење на команда за поместување на раката. Најчесто ова го решава проблемот, во спротивно уредот треба да се поправи или замени.

10.5. Алгоритами за движење на раката на дискот

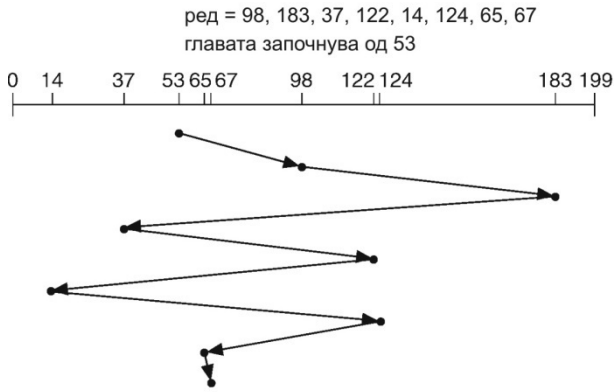
Оперативниот систем треба да обезбеди ефикасно искористување на хардверот, а В/И уредите преставуваат тесно грло во однос на перформансите. На пример, да се разгледа колку време му е потребно на дискот да прочита или запише блок. Времето се одредува со следните три фактори:

1. Време на барање (време за движење на раката до точниот цилиндар).
2. Ротационо задоцнување (време да точниот сектор ротира под главата).
3. Време на трансфер на податоците.

Брзината на пренесувањето на податоците од и код дискот (*англ. bandwidth*) е количник на вкупното количество на пренесени бајтови и вкупното време кое ги опфаќа овие три компоненти. Во повеќе процесно окружување постојат повеќе барања за работа со дискот. Со правилното распоредување на овие барања (*англ. disk scheduling*), може да се скрати времето на позиционирање или ротационото задоцнување. Постојат повеќе алгоритами за распоредување на барањата за работа со дискот.

10.5.1. First Come First Served (FCFS)

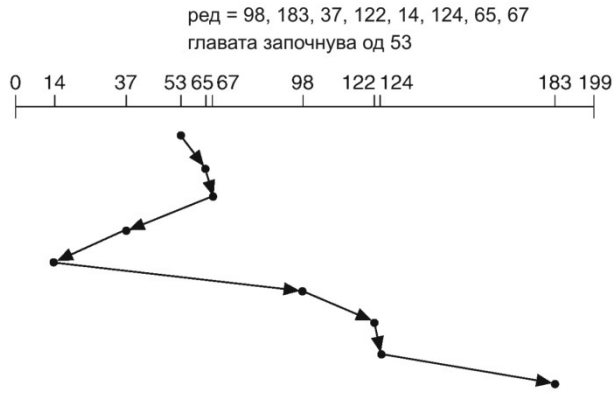
Ако дискот ги прифаќа барањата и ги реализира по истиот редослед како што ги примил, тогаш станува збор за алгоритамот наречен „Прв-дојден, прв-услужен“ (*First-Come, First-Served – FCFS*). Овој алгоритам се однесува фер спрема сите барања но не е многу ефикасен и потребно е да се усоврши. На сликата е даден пример за алгоритамот, каде што главата на читање тековно се наоѓа на позицијата 53, а барањата за пристап стигнуваат по следниот редослед 98, 183, 37, 122, 14, 124, 65, 67 при што се добива вкупно поместување на „главата“ од 640 цилиндари.



Слика 10.13: Пример за FCFS

10.5.2. Shortest Seek First (SSF)

Нека се земе истиот пример од претходната слика. Доаѓа барање за читање на блок од цилиндар 98. Додека извршувањето на барањето за цилиндарот 98 е во тек, доаѓа ново барање за цилиндарите 183, 37, 122, 14, 124, 65, 67 во тој редослед. Барањата се прикажани на Слика 10.14.



Слика 10.14: SSF алгоритам

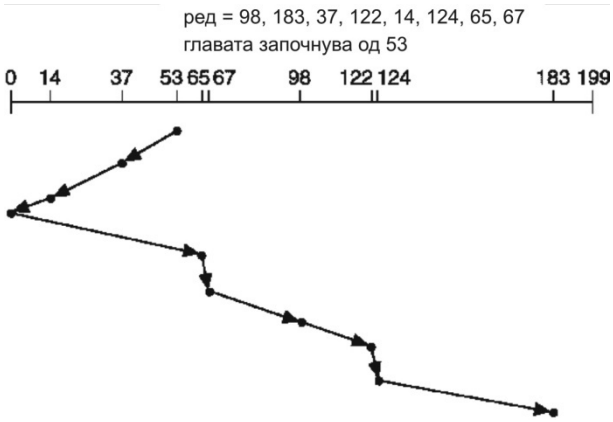
Откако барањето за читање на цилиндар 98 е завршено, диск уредот има можност да избира на кое барање ќе продолжи. Со користење на FCFS, дискот би одел до цилиндарот 183, па 37 и т.н. За да се намали времето за пребарување може да се изврши она барање за цилиндарот кој е најблиску до позицијата на главата. Овој алгоритам е наречен Најкраткото барање прво (*Shortest Seek First – SSF*). И во овој случај е постигнато движењето на главата од 236 цилиндри.

И овој алгоритам има одредени проблеми, ако се претпостави дека барањата продолжуваат да доаѓаат додека барањата од Слика 10.14 сè уште се во тек. На

пример, ако после поминувањето на цилиндар 124, ново барање за цилиндар 67 е пристигнато, кој има поголем приоритет од цилиндар 183. Ако дојде барање за цилиндар 37, протокот ќе оди на 37, наместо на 183. Овој алгоритам е оптимален според времето на позиционирање, но предизвикува „изгладнување“, главата може премногу време да остане околу една позиција опслужувајќи ги барањата со мали поместувања на главата.

10.5.3. SCAN - Лифт (Elevator)

Проблемот на „изгладнување“ на барањата за пристап кон дискот е решен со користење на логиката на лифт. Па така главите продолжуваат да се движат во иста насока сè додека не стигнат до крајот на дискот, потоа ја менуваат насоката и ги опслужуваат барањата во таа насока. На примерот е прикажано вкупно движење од 208 цилиндари. Во овој случај алгоритамот на лифтот е малку подобар отколку SSF.



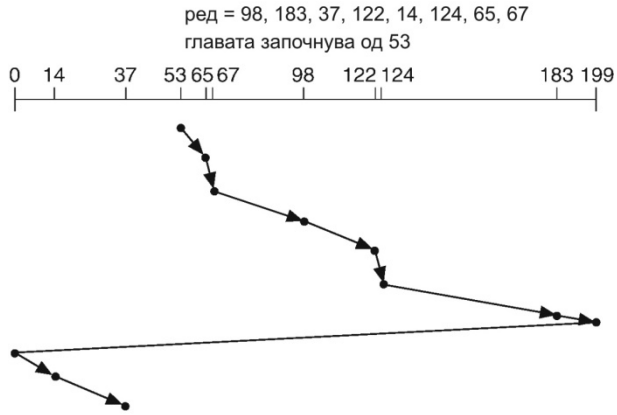
Слика 10.15: Алгоритам на лифт

Според перформансите SSF, може да биде подобар од алгоритамот SCAN, што тоа не е случај на прикажаниот пример (SSF со вкупно поместување од 236, наспроти SCAN со 208 цилиндари поместување). При обработката на барањата, SCAN дава предност на внатрешните цилиндари во однос на периферните.

10.5.4. C-SCAN

Алгоритамот C-SCAN (циркуларен SCAN) е варијанта на алгоритамот на претходно прикажаниот алгоритам кој обезбедува поуниформно време на чекање отколку SCAN и го решава проблемот на фаворизирање на внатрешните цилиндри. Главата се движи од еден крај на дискот кон другиот, опслужувајќи ги притоа барањата. Кога ќе се достигне крајот, веднаш се враќа на почетокот, без опслужување на барањата при враќањето назад. Ги гледа

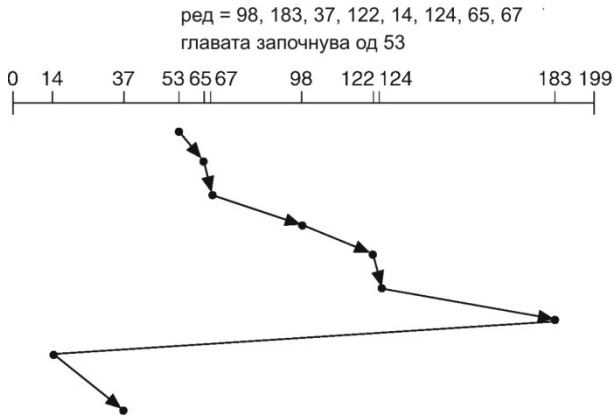
цилиндарите како циркуларна листа која се протега од последниот цилиндар до првиот.



Слика 10.16: C-SCAN алгоритам

10.5.5. C-LOOK

Овој алгоритам е варијанта на C-SCAN, каде што главите не се поместуваат до крај или почетокот на дискот туку само до последното барање во една насока, а потоа ја менува насоката веднаш без придвижување кон крајот на дискот. Постои сличен алгоритам кој е варијанта на SCAN и се нарекува LOOK, кој врши опслужување на барањата во двете насоки но не оди до крајот на дискот туку само до крајните барања во редицата на чекање.



Слика 10.17: C-LOOK алгоритамот

Ако дискот има такво својство времето на пристапот да е многу побрзо од ротационото задоцнување, тогаш би била користена друга стратегија за оптимизација. Важно е да се сфати дека горенаведените алгоритами за запишување на диск, претпоставуваат дека вистинската диск геометрија е иста

како и виртуелната геометрија. Ако не, тогаш барањата за запишување на дискот се без смисла бидејќи оперативниот систем не може да каже дали цилиндар 40 или цилиндар 200 е поблиску до цилиндар 39. Од друга страна, ако диск контролерот може да прифати повеќе надворешни барања, може да користи и внатрешен алгоритам.

Може да се каже дека SSF е најкористен и најсоодветен, додека SCAN и C-SCAN имаат подобри перформанси кај системите кои предизвикуваат поголемо оптоварување на дискот. Перформансите зависат од бројот и видот на барањата, дури можат да имаат влијание и од начинот на алокација на датотеките (*англ. file-allocation method*). Алгоритамите за распоредување на дискот, можат да бидат напишани како одвоени модули од оперативниот систем, притоа овозможувајќи да бидат заменети со друг по потреба, а алгоритамите SSTF или LOOK се соодветен избор за *„default“* алгоритам.

10.6. Стабилно складирање

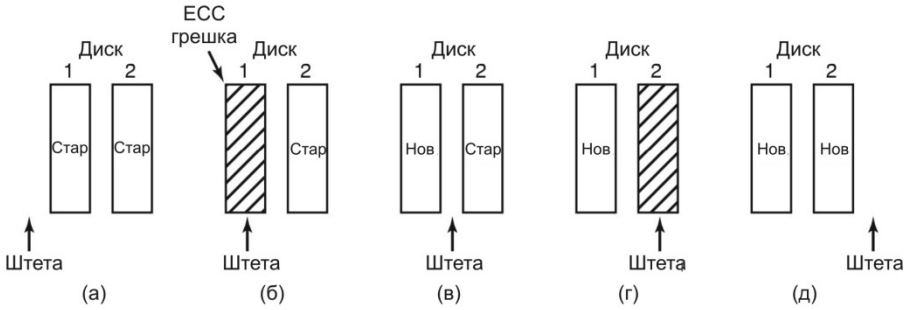
Дисковите понекогаш прават грешки. Добри сектори можат одеднаш да станат лоши сектори. Па дури и диск уредот неочекувано може да престане да работи. За некои апликации многу е важно податоците никогаш да не се изгубат или да бидат оштетени, па дури и да има грешки на дискот или грешки предизвикани од процесорот. Она што може да се постигне е диск подсистем кој при запишување на податоците или ги запишува правилно или ништо не прави, системот е наречен стабилно складирање.

Тој користи пар идентични дискови со блокови кои се совпаѓаат и кои работат заедно да формираат еден блок без можност на грешка. Во отсуство на грешка и двата блока ќе бидат идентични па било кој да се прочита ќе се добие истиот резултат. За постигнување на оваа цел, претставени се следните три операции:

1. *Стабилно запишување***.** Стабилното запишување се состои од: запишување на блокот од дискот 1, потоа читање наназад да се провери дали се е точно запишано. Ако не е точно запишано, пишувањето и читањето се прави n пати додека не проработи. Потоа, блокот е заменет со резервен и операцијата сѐ повторува додека не успее, без разлика колку резервни блокови треба да се искористат. Откако запишувањето на дискот 1 ќе биде успешно, совпаднатите блокови на дискот 2 се запишани и препрочитани, повторувајќи се до конечниот успех.
2. *Стабилно читање***.** Стабилното читање прво го чита блокот од дискот 1. Ако ова предизвика грешен ECC, читањето се повторува сѐ до n пати. Ако сето ова даде лош ECC, совпаднатиот блок се чита од дискот 2. Имајќи го фактот дека успешни стабилни запишувања остваат две добри копии од блокот и веројатноста на истиот блок спонтано да се влошува на двата уреди во определен временски интервал не е возможна, стабилното читање секогаш успева.

3. Оправување од испад**.** По испадот, програмот за опоравување ги скенира двата диска и ги споредува блоковите. Ако парот блокови е добар и е ист, ништо не с прави. Ако еден од нив има ECC грешка, лошиот блок е препишан со вредноста од соодветниот исправен блок. Ако парот од блокови се добри но не се исти, блокот од дискот 1 се запишува во дискот 2.

На Слика 10.18 се прикажани анализи на влијанијата од штети на стабилни запишувања.



Слика 10.18: Анализа на влијанието од штети при стабилни запишувања

На Слика 10.18 а) ЦПЕ штетата е направена пред другата копија на блокот да биде прочитана па ништо нема да биде направено. На Слика 10.18 б), ЦПЕ оштетување се случило во текот на запишување на дискот 1, уништувајќи ги содржините на блокот. Како и да е, програмот за покривање ја наоѓа оваа грешка и го обновува блокот во дискот 1 од дискот 2. На Слика 10.18 в), ЦПЕ штетата е направена после запишувањето на дискот 1, но пред читањето на дискот 2. Па така програмот за покривање на штета го копира блокот од дискот 1 во дискот 2. Слика 10.18 г) е ист како Слика 10.18 б): правејќи покривање, добриот блок го препишува лошиот блок. Повторно, крајното достигнување е дека двата блока се исти. Слика 10.18 д) програмот за обновување забележува дека двата блока се исти, па ниеден не е сменет па пишувањето и тука завршува успешно. Кон оваа шема се возможни повеќе изменувања и надополнувања со цел поуспешно запишување и читање од диск уредите.

10.7. Редундантна низа од ефтини дискови (RAID)

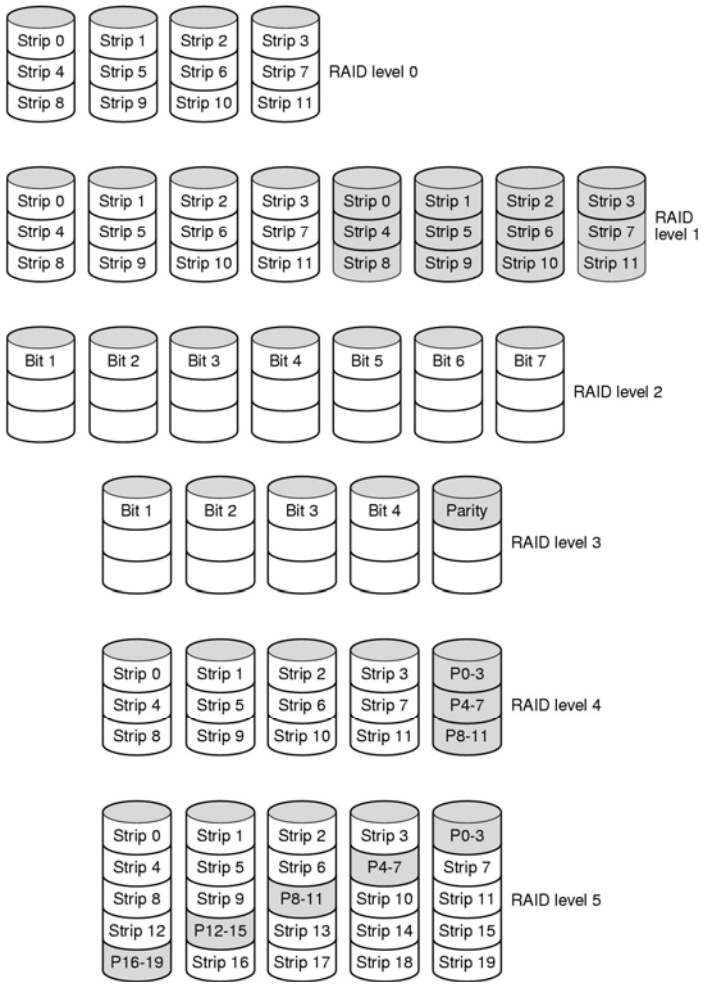
Запишувањето на дискот може да се заврши на три начини, успешно запишување, делумно успешно кога се појавува отказ на средината на пренос, некои од секторите се добро запишани некои не и потполн отказ кога не е запишано ништо. Системот може релативно лесно да се опорави од отказ ако се користи техниката на водење на дневник или кеширање, како и користењето на RAID структури.

RAID (*Redundant Array of Inexpensive Disks*) покрај можност за оправување од испади овозможуваат и подобрување на перформансите. Со RAID техниката повеќе дискови се комбинираат во еден логички диск со подобри перформанси и доверливост. RAID структурите се карактеризираат со:

1. Зголемување на перформансите со паралелизам на операциите на дискот за големи побарувања со што се делат податоците помеѓу различни дискови. Минималната единица на податоците кои може да се смести се нарекува лента (*англ. stripe*). Во случај на мали ленти, еден поголем логички блок се сместува на повеќе дискови. Со поголеми ленти, малите барања се опслужуваат од еден диск, што доведува до извршување на паралелни независни операции со дискот истовремено (конкурентност).
2. Зголемување на доверливоста (*англ. reliability*), секој диск се карактеризира со своето средно време на испад (MTTF). Веројатноста за испад на RAID структурата се намалува за N пати ако се додадат N дискови. Со тоа се обезбедува доверливост преку воведување на редундантност. Со чување на дополнителни информации се овозможува брзо и правилно оправување на податоците во случај на испад на еден диск. Постојат две основни шеми на RAID структури: огледало (*англ. mirroring*) каде што за запишување и читање на користат два диска. Истовремено се запишува на двата диска, така што при отказ на едниот, се врши замена на дискот и податоците се копираат од исправниот, основниот проблем е што на овој начин се искористува само 50% од расположливиот капацитет. Другата техника која се користи за зголемување на доверливоста е користење на техниката на паритет (*англ. parity information*).

10.7.1. RAID нивоа

Делењето на ленти ги подобрува перформансите но ја намалува доверливоста. Техниката на огледала ја зголемува доверливоста, но ги намалува перформансите и искористувањето на капацитетот. Затоа како компромис се воведува паритетот. RAID се организира во 6 различни нивоа кои меѓусебно се разликуваат по доверливоста, цената и перформансите. RAID нивоите се прикажани на Слика 10.19.



Слика 10.19: RAID нивоа

10.7.2. RAID 0

RAID 0 претставува конфигурација каде што лентата е на ниво на еден или повеќе блокови на податоци. Најмала цена за било која RAID организација, каде што не постои никаква редундантност, но има најдобра перформанса за запишување и читање. Една причина за тоа е што нема потреба од обновување на редундантната информација. Основен проблем е не постоењето на редундантност, испад на еден диск доведува до целосно губење на сите податоци. Затоа се користи во околини каде што перформансите и капацитетот (наместо доверливоста) се на прво место.

Оптимална „stripe“ единица - RAID-0

```
RAID-0
Stripe единица= Sqrt[(P x X x (L-1) x Z)/N]

Каде што
       P е просечно време за позиционирање,
       X просечна брзина на пренесување,
       L е конкурентност,
       Z е големина на побарувањето
       N е број на дисковите во низата

MTTF(RAID-0) = MMTF(disk) / N
```

10.7.3. RAID 1

Во оваа конфигурација, секој диск си има свое огледало. Се користат 2 пати повеќе дискови отколку не-редундантна низа и има најлоши перформанси за читање и запишување. При запишувањето се трошат 2 циклуси на запишување, додека при читањето на податоците, тие може да се преземат од дискот со помало чекање, пронаоѓање и ротационо доцнење. Ако се случи испад на дискот може да се користи копијата за работа. Огледалната шема (*англ. mirroring*) често се користи кај податочни бази, каде што достапноста и обемот на трансакциите се позначајни од ефикасноста на запишување.

10.7.4. RAID 2

Ова конфигурација е позната како RAID со мемориски стил на корекција. Мемориските системи обезбедуваат опоравување од испаднатите компоненти со користење на Hamming-овите кодови по многу помала цена (ECC). Мемориите имаат ECC алгоритам каде што се секој бајт има 3 дополнителни бита за корекција на еднобитни грешки. Кај RAID 2 во една варијанта од оваа шема, за 4 податочни дискови треба да се користат 3 редундатни дискови, значи еден помалку од RAID 1. Ефикасноста на складирањето се зголемува пропорционално на логаритамот од вкупниот број на дисковите во системот. Ако се појави испад на една компонента, може да се процени која е и може да се изврши обновување на информацијата преку читање на другите компоненти во подмножеството (податоци и паритет). Повеќе редундантни дискови се потребни за идентификација на испаднатиот диск, но само еден е потребен за обновување на изгубената информација.

10.7.5. RAID 3

Ова е конфигурација која се користи делење на податоците на ниво на бит или бајт, а по прашањето на паритетот прв пат се воведува паритет на дискови (*англ. bit interleaved parity*). За разлика од мемориски ECC, кај низи на дискови важи дека може лесно да се идентификува испаднатиот диск. Затоа, може да се

користи диск со единечен паритет за обновување на изгубената информација, затоа што точно се знае на која битска позиција настапила грешката. За сите дискови во низата доволен е еден диск за исправка на парноста. Во овој случај „*stripe*“ единица е помала од 512 бајти и секое барање за читање пристапува кон сите дискови, а секое барање за запишување пристапува кон сите дискови и дискот за паритет**.** Според тоа само едно барање може да се опслужи во еден момент. Затоа што дискот за паритет содржи само паритети, но не и податоци тој не може да учествува во операцијата за читање, што доведува до помали перформанси во однос на другите шеми. „*Bit-interleaved*“, паритетни дискови се користат во апликации каде што е потребен поголем пропусен опсег.

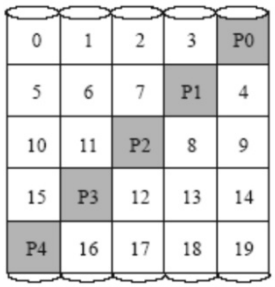
10.7.6. RAID 4

Конфигурацијата има организација на поделбата на податоците на ниво на блок, а за редундантност користи парност за дискови на ниво на блок (*Block-Interleaved Parity*), што значи дека „*striping*“ единица се состои од *N* диск блокови за кои е доволен еден блок за парност кој е еднаков на големината на лентата. Барањата за читање се помали во однос кога се пристапува само кон еден диск. Барањата за запишување треба да ги обноват податочните блокови и исто така да го обноват и блокот за паритет. За големи запишувања, кои ги опфаќаат блоковите на сите дискови, паритетот може лесно да се премета со XOR на новите податоците на секој диск. За мали запишувања кои обновуваат само еден блок на еден диск, паритетот се пресметува од колку новите податоци се разликуваат од старите и со примена на таа разлика на блокот за паритет. Малите запишувања бараат 4 В/И циклуси, еден да се запише новиот податок, два да се прочита стариот податок и паритетот за да се пресмета новиот паритет и еден за запишување на новиот паритет. Ова се означува како „*read-modify-write*“ процедура.

Постои само еден диск за паритет, кој се обновува на сите операции за запишување и може да стане тесно грло. Заради ова ограничување почесто се користи дистрибуиран паритет на преплетени блокови.

10.7.7. RAID 5

Со дистрибуиран паритет се отстранува тесното грло на пристап кон еден диск за паритет, конфигурацијата е многу слична на RAID 4, но во овој случај сите дискови се користат и за податоците и за паритет. Уште се нарекува и дистрибуиран паритет на преплетени блокови (*англ. Block-Interleaved Distributed-Parity*). Паритетот се запишува по лев симетричен распоред.



Слика 10.20: P0 го пресметува паритетот преку единиците 0,1,2 и 3, P1 преку единиците 4,5,6 и 7, секој квадрат соодветствува на stripe единица, секоја колона претставува диск

RAID 5 е најдобар за мали читања, големи читања и постигнува најдобра перформанса за големи запишување од сите други и сите дискови се симетрично оптоварени. Малите запишувања се неефикасни во споредба со на пр. „*mirroring*“ заради изведување на операциите за запишување за обнова на паритетноста на сите дискови. Тоа е и најголемата слабост на RAID level 5 низите на дискови и е предмет на интензивни истражувања.

```
stripe_unit= 0.5K+1/4 * средно време на позиционирање * брзина на
податочен трансфер * (конкурентност -1))

stripe_unit=  0.5 * средно време на позиционирање * брзина на
податочен трансфер
```

MTTF(RAID-5) = MMTF^2(disk) / MMTR(disk)xN(G-1)

N е број на дисковите
G група на паритет

MTTF(RAID-1) = MMTF^2(disk) / 2xMMTR(disk)

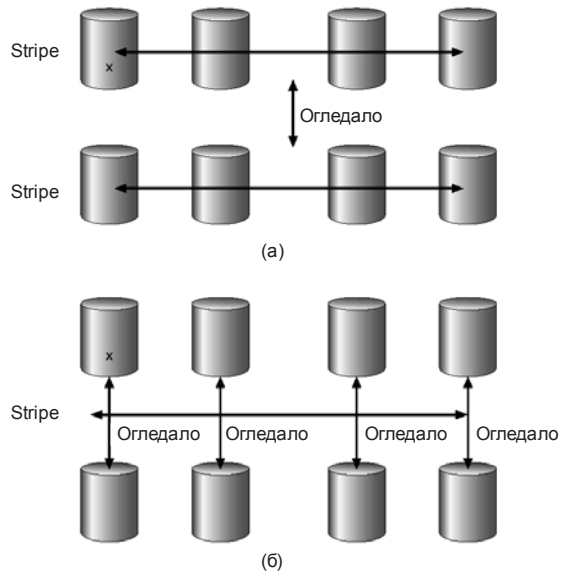
N = 2
G = 2

10.7.8. RAID 6

За поголеми дискови и повеќекратни испади не е доволно користењето на Хаминговиот код, за исправање на една грешка. Затоа се користи *P+Q редундантноста*, која користи *Reed-Solomon* код за заштита од две грешки со користење на најмалку два диска за паритет. *P+Q редундантните* низи се слични на блоковските низи со дистрибуиран паритет и функционираат на сличен начин. Во суштина, P+Q редундантните низи исто изведуваат мали операции на запишувања со користење на „*read-modify-write*“ процедурата, но наместо 4 В/И операции, P+Q бара 6 операции за обнова на ‘P’ и ‘Q’ информации.

10.7.9. RAID (0 + 1) и (1 + 0)

Ова се две квалитетни комбинации на техниките 0 и 1, кои се прикажани на сликата. RAID 0 обезбедува високи перформанси, а RAID 1 висока доверливост. Меѓутоа комбинацијата е скапа затоа што се трошат двојно повеќе дискови. Во комбинацијата 0+1, множество на дискови дели податоци, а потоа нивната копија се запишува на своето огледало. Втората комбинација 1+0 значи дека секој диск има сопствено огледало, а податоците се разместуваат помеѓу огледалата. Теориски 1+0 е подобар од 0+1, кога во комбинацијата 0+1 еден диск откаже, целата група на ленти не е достапна на две места, додека таа во огледалото е достапна. Кај, 1+0, испад на еден диск влијае врз доверливоста на само таа лента која има само една расположлива копија.



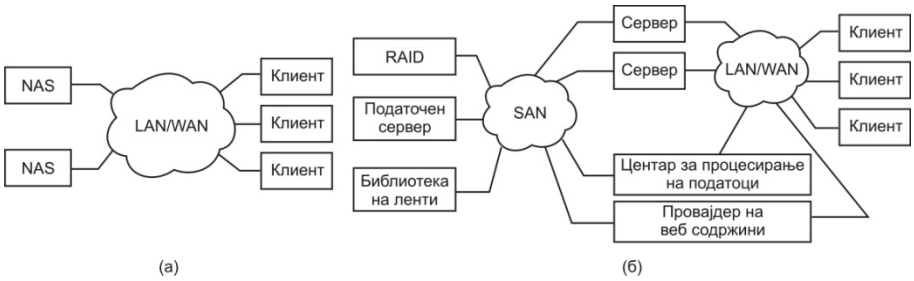
Слика 10.21: а) RAID 0+1 б) RAID 1+0

10.8. Прикачување на дискови

Дисковите на компјутерските системи може да се прикачат на два различни начини: на самиот хост преку В/И порта, кој се користи на помалите компјутерски системи и друг начин е преку мрежна конекција која се користи во големите дистрибуирани системи.

Во последниов начин, клиентите пристапуваат кон специјален систем на дискови со користење на компјутерска мрежа. Притоа основен недостаток е преносот на големо количество на податоци преку мрежата со што се нарушуваат нејзините перформанси.

Поврзувањето на дисковите преку В/И конектор се изведува со различни технологии, што веќе се претходно опишани, ATA, SATA, SCSI како и FC (*англ. fibre channel*). FC претставува сериска архитектура со високи перформанси базирана на оптички кабел.



Слика 10.21: а) Мрежно поврзани дискови б) SAN поврзување

Постојат две варијанти на FC технологијата:

- SAN (*Storage Area Network*) која овозможува приклучување на голем број на компјутери и дискови. SAN има 24-битен адресен простор.
- Арбитрарна јамка (*англ. arbitrated loop FC-AL*) на која може да се приклучат 12 уреди.

SAN се реализира како приватна мрежа помеѓу серверите и дисковите кои се приклучени на мрежата. Во овој концепт, клиентите не пристапуваат директно кон дисковите туку преку обичните LAN/WAN мрежи пристапуваат кон серверот на SAN системот.

11. UNIX

Во претходните подглавја беа објаснети повеќе концепти на оперативните системи. Во овие две поглавја ќе бидат разгледани практичните имплементации на тие концепти. Прв оперативен систем кој ќе биде разгледан е UNIX од причина што тој работи на повеќе видови компјутерски системи. Исто така тој заедно со неговите варијации е доминантен оперативен систем кој се среќава кај серверите, работните станици, но сè почесто се среќава и кај персоналните компјутери. Тој внимателно е дизајниран и изработен, па иако е стар се уште е модерен и елегантен.

11.1. Историја на UNIX

UNIX има долга и интересна историја. Започнал да се развива како мал проект на млад истражувач за да прерасне во милионска индустрија која вклучува универзитети, меѓународни компании, влади и меѓународни тела за стандардизација.

11.1.1. UNICS

Во 40-тите и 50-тите, сите компјутери биле големи и само една личност (програмерот) можел да ги користи во одредено време. Во 60-тите програмерот ги поставувал задачите за извршување на дупчени картички во посебна соба. За извршување на задача се потребни повеќе часови за да биде вратен резултатот назад. Под овие околности, отстранувањето на грешки е многу тежок и комплексен процес, бидејќи само една испуштена запирка, може да предизвика неколку часа изгубено време.

Потоа се воведува временска поделба на ресурсите. Главни реализатори на оваа идеја се *Dartmouth College* и M.I.T. По одредено време, истражувачите на M.I.T., Bell Labs и General Electric започнале да ја дизајнираат втората генерација на оперативни системи, MULTICS (*MULTiplexed Information and Computing Service*).

Покасно Bell Labs се повлекуваат од проектот, меѓутоа еден од нивните истражувачи *Ken Thompson*, започнал да програмира негова верзија на MULTICS на тогашниот миникомпјутер PDP-7. Овој систем од *Brian Kernighan* на шега бил наречен UNICS (*UNiplexed Information and Computing Service*) за да подоцна биде пишуван како UNIX.

11.1.2. PDP-11 UNIX

Thompson-овата работа била толку импресивна, така што повеќе негови колеги од Bell Labs заедно со *Dennis Ritchie* се придружиле во развојот. Како прво, UNIX бил префрлен од PDP-7 на помодерните PDP-11/20, PDP-11/45 и PDP-11/70. Последните два миникомпјутери доминирале во нивната област во 70-тите.

Вториот дел од развојот се однесувал на јазикот во кој UNIX е напишан. До тогаш за секој компјутер целиот код требало да се препишува, што и не е баш лесно и едноставно. Така *Thompson* одлучил да го препише UNIX во јазик од повисоко ниво, кој го дизајнирал и го нарекол *B*. Поради многуте слабости на *B*, овој обид не бил баш успешен. *Ritchie* тогаш го напишал наследникот на *B*, програмскиот јазик *C* и направил одличен компајлер за него. Заедно, *Thompson* и *Ritchie* го препишале UNIX во *C*. Од тогаш *C* станува доминантен програмски јазик. Во 1974 година тие издаваат публикација за UNIX, која е прифатена на повеќе универзитети. Универзитетите побарале копија од UNIX бидејќи оперативниот систем што доаѓал со PDP-11 не бил многу корисен. Така UNIX бргу ја пополнува празнината.

Многу научни конференции се организирале за UNIX и многу луѓе се вклучиле во пронаоѓањето и отстранувањето на грешките. Австрискиот професор *John Lions* напишал коментари на UNIX кодот. Кодот на верзијата 6 содржел 8200 линии *C* код и 900 линии асемблерски код. Како резултат на оваа активност дошло до појавување на нови идеи и усовршувања.

До средината на 80-тите, UNIX бил најкористен кај миникомпјутерите и работните станици. Многу компании од изворниот код правеле и свои верзии на UNIX. Меѓу нив била и компанијата Microsoft која ја продавала верзијата 7 под име XENIX.

11.1.3. Berkeley UNIX

Еден од повеќето универзитети кои ја имаат набавено верзијата 6 на UNIX е Универзитетот од Калифорнија на Berkeley. Бидејќи целиот изворен код им бил достапен, истражувачите од Berkeley имале можност да прават измени. Со донации на ARPA, тие направиле и издале нова верзија за PDP-11 која била наречена 1BSD (*First Berkeley Software Distribution*). Набргу следувале и 2BSD, 3BSD и 4BSD верзиите. Верзијата 4 содржела многу подобрувања. Помеѓу нив позначајни биле и користењето на виртуелната меморија и страничењето, со што било дозволено програмите да се поголеми од достапната физичка меморија. Друго подобрување било и тоа што имињата на датотеките можеле да бидат поголеми од 14 карактери. Бил сменет и системот на датотеките. Она што е важно е воведувањето на мрежа, каде за стандардни протоколи кои ќе бидат користени во UNIX се избрани TCP/IP протоколите. Покрај сите подобрувања, Berkeley имаат додадено и многу кориснички програми.

11.1.4. Стандарден UNIX

До доцните 80-ти, биле користени две различни и често и некомпатибилни верзии на UNIX: 4BSD и System V Release 3. Па често се случувало и секој дистрибутер да додаде нешто не стандардно. Со ова UNIX светот бил разделен, не постоеле стандарди па така дистрибутерите не можеле да напишат програма која ќе работи на било кој UNIX систем (нешто што не било карактеристично за MS-DOS).

Многу обиди за стандардизација пропаднале. Првиот посериозен обид бил од страна на IEEE одборот за стандардизација. Стотици луѓе од индустријата, образованието и владините тела земале учество во оваа активност. Името на овој проект е POSIX (*Portable Operating System*).

После долга работа бил произведен стандардот со ознака 1003.1. Тој дефинирал множество на библиотечни процедури кои секој UNIX систем треба да ги поддржува. Целта била да било кој дистрибутер кој пишува програма според POSIX стандардот ќе може да ја извршува на било кој UNIX.

Иако 1003.1 стандардот се однесува за системските повици, дополнителни документи ги стандардизираат нишките, алатките, мрежното работење и други особини. Исто така и C јазикот е стандардизиран од страна на ANSI и ISO.

Група на дистрибутери предводени од IBM, DEC, HP, направиле конзорциум скратено наречен OSF (*Open Software Foundation*) за да произведат систем кој освен IEEE стандардите ќе користи и други како X11 графички кориснички интерфејс, дистрибуирано пресметување и др.

11.1.5. MINIX

Заедничко својство за сите овие системи е тоа што се големи и комплицирани, за разлика од оригиналната идеја зад која е развиван UNIX-от. Па дури и изворниот код да биде даден (што не е секогаш случај) тешко дека една личност може да го разбере. Ова води кон идеја група на автори да напишат помал систем, кој ќе личи на UNIX и кој ќе биде лесен за разбирање. Тој систем се состои од 11.800 линии на код во C и 800 линии асемблерски код. Издаден е во 1987 година.

MINIX е еден од првите системи слични на UNIX, базиран на микројадро. Идејата зад микројадрото е да обезбеди минимална функционалност на јадрото а да биде ефикасен. Микројадрото е подобро од монолитските јадра затоа што лесно се разбира. Исто така префрлањето од јадрен во кориснички мод го прави многу посигурен поради испади на корисничките процеси и се прави помала штета. MINIX е бесплатен и може да се симне од Интернет и пред сè се користи за истражувачки и научни цели.

11.1.6. Linux

За време на првите години на MINIX и дискусиите на Интернет, многу корисници побарувале подобри особини за кои авторите на MINIX постојано го одбивале со цел да се одржи системот доволно мал за да студентите во еден семестар го разберат функционирањето на оперативниот систем. После неколку години, Финскиот студент, *Linus Torvalds*, одлучил да креира друг клон на UNIX, под името Linux. Првата верзија се појавила во 1991 година, се состои од 9.300 линии код и целиот изворен код е отворен.

Низ годините многу програмери го додавале својот код кон Linux, па така неговата големина бргу пораснала а кон него се додадени многу функционалности.

Следното поголемо издание од 1994 година содржело 165.000 линии код. Тој бил компатибилен со UNIX па така многу UNIX програми можеле да се стартуваат под Linux што го правело многу поразличен и покорисен.

Изданието 2.0 од 1996 година содржело 470.000 линии код. Во него била вклучена и поддршка за 64 битна архитектура, симетрично мултипрограмирање, нови мрежни протоколи, голем број на драјвери за уреди и други особини.

Многу програми од различни автори се додавани кон Linux, а направени се и различни графички интерфејси (GNOME и KDE).

Овој оперативен систем може да се најде во бесплатна верзија. Она што е карактеристично за овој оперативен систем е тоа што тој се дистрибуира по GPL (*GNU Public License*) лиценцата. Тоа значи дека корисниците може да го користат, копираат, менуваат и редистрибуираат кодот бесплатно. Но главното ограничување е тоа што не може да се продава или редистрибуира само во бинарна форма, туку заедно со тоа треба да оди и кодот.

Трендот на овој оперативен систем оди нагоре. Иако пред сè се користи на серверите или работните станици, сè почесто е користењето и на персоналните компјутери и претставува главен конкурент на Microsoft-овиот Windows.

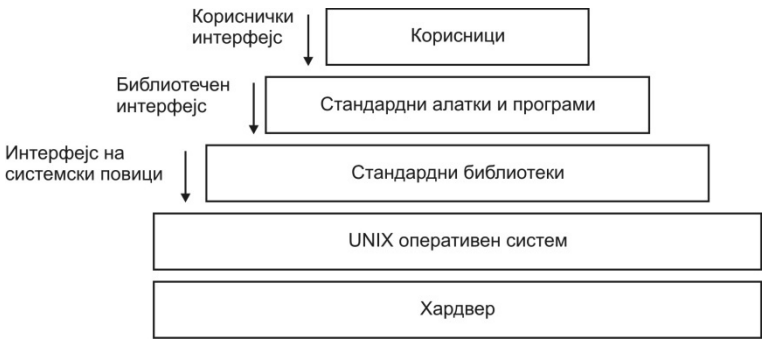
11.2. Општ преглед на UNIX

UNIX е интерактивен систем дизајниран да се справи со повеќе процеси и повеќе корисници во исто време. Тој е дизајниран од програмери за програмери, со цел да користат околина во која повеќето од корисниците не се почетници. Во многу случаи програмерите активно соработуваат на ист проект а притоа да делат податоци и информации на контролиран начин. Ова добра особина го прави UNIX различен уште од самото негово појавување.

Она што програмерите го сакаат е едноставен, елегантен и складен систем. Исто така сакаат голема моќ и флексибилност. Ова значи дека системот има мал број на основни елементи кои може да се комбинираат на бесконечно

начини за да се креираат добри апликации. Па така во UNIX секоја програма работи една работа но ја работи најдобро (ова се однесува за наредбите кои се извршуваат во конзолата). Исто така програмерите сакаат скратени работи па така наместо *copy* се користи *cp*.

UNIX системот може да биде поделен на повеќе нивоа како што е прикажано на Слика 11.1. Најниското ниво е хардверот. Над него е самиот оперативен систем кој го контролира хардверот и обезбедува системски повици до сите програми. Овие повици овозможуваат корисниците да креираат и управуваат со процесите, датотеките и другите ресурси.



Слика 11.1: Нивоа кај UNIX оперативниот систем

Програмите прават системски повици па така ако C програма треба да го повика *read* системскиот повик, *C* програмата може да ја повика *read* библиотечната процедура. Така се повикува библиотечен интерфејс наместо интерфејсот на системскиот повик, што е специфициран во POSIX. Со други зборови POSIX кажува која библиотечна процедура на кој системски повик одговара.

Сите верзии на UNIX подржуваат голем број на стандардни програми, некои од нив специфицирани во POSIX стандардот а некои се разликуваат кај различните верзии на UNIX.

Според ова може да се зборува за три различни интерфејси во UNIX: интерфејс на системски повици, библиотечниот интерфејс и интерфејсот формиран од стандардните алатки.

Иако повеќето UNIX системи имаат графички интерфејс, голем дел од програмерите преферираат работа со конзола. Тоа е така бидејќи е лесна и брза за користење, помоќна и корисниците не мора да го користат глувчето. Постојат повеќе конзоли меѓу кои најпозната е *Bourne shell (sh)*. Постои и можност за пишување на скрипти кои содржат повеќе наредби кои се извршуваат една по друга.

11.2.1. Структура на јадрото

Прикажувањето на структурата на јадрото на UNIX е малку проблематично поради тоа што кај различните верзии на UNIX има разлики во јадрото. На Слика 11.2 е прикажано јадрото на 4.4BSD и главно одговара со другите јадра.

Слика 11.2: Јадрото на UNIX 4.4BSD

Најниското ниво на јадрото се состои од драјвери на уредите и рспоредувач на процесите. Сите UNIX драјвери се класифицирани.

Над долното ниво, кодот е различен во секоја од четирите колони на Слика 11.2. Од лево се уредите со знаци. Мрежниот софтвер е модуларен со различни уреди и протоколи кои ги подржува. Нивото над мрежните драјвери има рутирачка функција со што се осигурува дека одреден пакет ќе отиде на правото место. Над рутирачкиот код е стекот од протоколи кој ги вклучува IP и TCP како и некои додатни протоколи. Над диск драјверите се наоѓа бафер на системот на датотеки и кеш за страничење. Потоа доаѓа системот на датотеки. Повеќето UNIX системи подржуваат различни системи на датотеки. Сите од нив го делат истиот бафер. Над него доаѓа именувањето на датотеките, управувањето со директориумите, тврдите и симболичките врски и други својства на системи на датотеки кои се заеднички за сите. Над кешот за страничење е системот за виртуелна меморија во кој се алгоритамите за замена на страници и кодот за *„page fault“* грешки кои може да се појават.

Последната колона се справува со процесите. Над доделувачот е распоредувачот на процеси кој одлучува кој следен процес ќе биде извршуван. Тука е и делот за управување со нишки, процесирање на сигнали и нивно испраќање до соодветната дестинација, за креирање и уништување на процеси. Највисокото ниво е интерфејсот на системот. На лево е интерфејсот за системските повици додека на десно е влезот за замки и прекини, вклучувајќи сигнали, грешки на страничење, исклучоци и сите видови на влезно/излезни прекини.

11.3. Процеси во UNIX

Единствените активни ентитети во UNIX се процесите. Секој процес преставува извршување на некоја програма, додека пак на секој процес му е дозволено да креира и сопствени нишки. На UNIX може да се извршуваат повеќе процеси во исто време па така секој корисник може да има повеќе активни процеси. Дури и ако корисникот не работи во моментот со компјутерот а е најавен, во позадина се извршуваат повеќе процеси кои му припаѓаат на тој корисник (овие процеси се познати како *демони*).

Процесите се креираат со системскиот повик *„fork“* кој креира копија на оригиналниот процес. Процесот кој креира процес со *„fork“* се нарекува родителски процес, додека новиот процес е процес дете. И двата процеси имаат своја меморија па така ако родителот направи промена на променливите тоа нема да се одрази врз променливите на процесот дете и обратно.

fork повикот враќа 0 кај детето а има друга вредност кај родителот. Овој идентификатор се нарекува PID (*Process IDentifier*). И двата процеси ја проверуваат вредноста и знаат кој процес е родител а кој дете. Еве дел од код каде се проверува кој процес е кој:

```
pid=fork();
if (pid<0){
       handle_error();     //не е креиран процесот
       }
else if (pid>0)
       {
                                  //код за родителот
       }
else
       {
                                  //код за детето
       }
```

Процесите во UNIX може да комуницираат еден со друг со испраќање на пораки преку канал помеѓу процесите. Овие канали се нарекуваат цевки. Еве пример за користење на цевки во конзола:

```
$ cat file.txt | head
```

Во овој пример излезот од процесот *cat* се пренесува на процесот *head*. Процесите може да комуницираат и преку софтверски прекини т.е. да пратат сигнал до друг процес. Постојат повеќе видови на сигнали кои се објаснети во делот за задачи.

11.3.1. Управување со системските повици

На Слика 11.3 се прикажани поважните системски повици.

Системски повик	Опис
pid = fork ()	Создава процес – дете идентичен на родителот
pid = waitpid (pid, &statloc, opts)	Чека на завршување на процесот - дете
s = execve (name, argv, envp)	Замена на содржината на процесот
exit (status)	Завршува со извршување на процесот и враќа статус
s = kill (pid, sig)	Испраќа сигнал на процесот
residual = alarm (seconds)	Поставување на часовникот на алармот
s = pause ()	Суспендирање на повикувачот сè до појава на следен сигнал

Слика 11.3: Поважни системски повици во UNIX

Нивното повикување е едноставно и во продолжение ќе бидат разгледани некои од нив.

Единствен начин за креирање на процес е *fork*. Со него се креира дупликат на оригиналниот процес и се враќа PID вредност. Доколку родителскиот процес ја повика наредбата:

```
n = waitpid(-1, &status, 0);
```

процесот ќе биде стопиран се додека не заврши процесот дете. Доколку процесот излезе со 4, тогаш *n* ќе ја има вредноста на PID на детето а во *status* ќе биде излезот од програмата, а тоа е 4.

За да се справи со сигналите процесот може да ги користи системските повици кои се наменети за тоа. На пример, кај *sigaction* првиот параметар е сигналот кој треба да се фати, вториот е покажувач кон структурата која дава покажувач кон процедурата за справување со сигналот и некои дополнителни битови и знаменца. Додека третиот покажува кон структура каде системот враќа информација за справувањето со сигналот моментално, во случај да биде вратен подоцна.

11.3.2. Имплементација на процеси во UNIX

Секој процес има кориснички дел кој ја извршува корисничката програма. Меѓутоа, кога една негова нишка ќе направи системски повик, тој влегува во мод на јадрото каде има пристап до сите ресурси. Иако станува збор за истата нишка, сега има повеќе моќ и друг стек и програмски бројач.

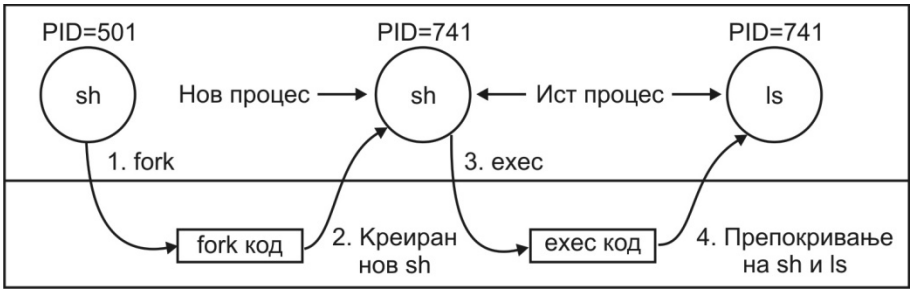
Јадрото чува две податочни структури кои се однесуваат на процесите, табела на процесот и корисничка структура. Табелата е постојана и ги содржи информациите за сите процеси па и за оние кои не се во моментот присутни во меморијата. Корисничката структура се страничи кога процесот не е во меморијата со цел да не се троши меморија на информации кои не се потребни.

Податоците во табелата на процесите може да се поделат:

1. *Параметри за распределба*. Приоритет, процесорско време кое се користи, време колку процесот спие. Заедно, овие параметри се користат да се одлучи кој процес ќе биде следен.
2. *Слика на меморијата*. Покажувачи кон текст, податоци, стек сегменти и табели за страничење.
3. *Сигнали.* Маски кои кажуваат кои сигнали да бидат игнорирани, кои да бидат фатени, кои да бидат привремено блокирани итн.
4. *Останато.* Моментални состојби на процесите, PID, PID на родителите и идентификации за корисниците и групите.

Корисничката структура содржи информации кои не се потребни кога процесите не се во меморијата и не се извршуваат. Затоа информациите за сигналите се во табелата на процеси, бидејќи може да се појави и сигнал кога процесот не е во меморијата. Информациите може да се поделат на следниве категории:

1. Машински регистри;
2. Состојба на системските повици;
3. Дескриптори за датотеки;
4. Пресметки за процесорското време;
5. Стек на јадрото.



Слика 11.4: Чекори при извршувањето на наредбата *ls*

Кога системскиот повик *fork* се извршува, јадрото бара слободно место во табелата на процесите. Ако има место се копираат сите информации од родителот на детето. Корисничката структура се копира заедно со стекот. Потоа детето е спремно да се извршува.

Кога корисникот ќе ја напише ls командата во конзолата, конзолата креира нов процес копија на нејзе. Новата конзола го повикува *exec* за да ја дополни нејзината меморија со содржината на извршната датотека *ls*. Чекорите се прикажани на Слика 11.4.

11.3.3. Нишки во UNIX

Имплементацијата на нишките зависи дали тие воопшто ги подржува верзијата на UNIX. 4BSD не ги подржува нишките, додека System V и Solaris подржуваат нишки.

Да се претпостави дека процес со повеќе нишки прави *fork* повик. Дали сите нишки ќе бидат креирани во новиот процес? Нека се претпостави дека е така. Да се претпостави дека една нишка го има блокирано читањето од тастатурата. Дали и новата нишка во новиот процес исто така го има блокирано читањето од тастатурата? Тука е можна појава на проблем. Во процесите со една нишка, не се појавува проблем бидејќи една нишка не може да блокира кога го повикува *fork*.

Да се претпостави дека другите нишки не се креирани во процесот дете. Ако се претпостави дека некоја не - креирана нишка држи *mutex* тогаш може да се појави проблем. Проблеми може да се појават и при работа со датотеки како и при работа со сигнали. Основата на имплементацијата на нишките во Linux се заснова на системскиот повик *clone* кој се повикува на следниот начин:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

Знаменце	Значење кога е поставен	Значење кога е избришан
CLONE_VM	Создава нова нишка	Создава нов процес
CLONE_FS	Делење на umask, root и работни директориуми	Нема делење на директориумите
CLONE_FILES	Делење на дескрипторите на датотеките	Копирање на дескрипторите
CLONE_SIGHAND	Делење на табелата со ракувачи на сигналите	Копирање на табелата
CLONE_PID	Нова нишка добива стар PID	Нова нишка добива сопствен PID

Слика 11.5: Битови за *sharing_flags*

Со ова се креира нова нишка, во истиот или во нов процес во зависност од *sharing_flags*. На Слика 11.5 се прикажани битовите за *sharing_flags*. Ако нишката е во истиот процес, таа ги дели адресното подрачје со другите нишки, а во спротивно ова подрачје не е исто. Во двата случаи новата нишка започнува со извршувањето на *function* со аргументи *arg*. И во двата случаи новата нишка има свој приватен стек со стек покажувач *stack_ptr*.

11.3.4. Распоредување во UNIX

Бидејќи уште од самиот почеток UNIX е систем кој подржува извршување на повеќе процеси, неговиот алгоритам за распоредување е дизајниран да обезбеди добро однесување кон интерактивните процеси. Алгоритамот се состои од две нивоа. Алгоритамот од ниското ниво одбира процес кој ќе се извршува следен додека алгоритамот од високо ниво ги преместува процесите помеѓу меморијата и дискот, така сите процеси добиваат шанса да се извршат.

Различните верзии на UNIX имаат различни алгоритами на ниско ниво но главно се слични. Алгоритамот користи повеќе редови за чекање. Процесите кои се во кориснички мод имаат позитивни вредности додека процесите кои се во мод на јадро, негативни вредности. Негативните вредности имаат висок приоритет додека позитивните низок.

Алгоритамот ги пребарува редовите со највисок приоритет и го избира редот кој е зафатен. Првиот процес во тој ред започнува да се извршува одреден временски квантум (околу 100 msec) или претходно да биде блокиран. Ако процесот го премине квантумот, се сместува на крајот на редот и алгоритамот се стартува повторно. Така процесите со ист приоритет го користат *Round-Robin* алгоритамот.

Во некои алгоритами се пресметува новиот приоритет на процесот според формулата:

```
priority = CPU_usage + nice + base
```

Па процесот се става на крајот на редот во кој припаѓа.

За разлика од UNIX, во Linux распределбата се базира на нишки наместо на процеси и во тој разликува три класи на нишки кога станува збор за распределувањето (*Real-time FIFO, Real-time round-robin, Timesharing*)

11.3.5. Подигнување на UNIX

Покрај другите работи и начинот на подигнување зависи од верзијата. Овде ќе биде разгледано подигнувањето на 4.4BSD. Најпрво се стартува мала 512 бајтна програма од записот MBR (*Master Boot Record)*. Таа ја внесува *boot* програмата од *boot* уредот кој најчесто е IDE диск. Оваа програма се копира во меморијата на фиксни адреси.

Потоа се внесува јадрото и *boot* програмата престанува со работа. Јадрото започнува со код напишан во асемблер кој е зависен од компјутерот. Се поставува стекот на јадрото, се идентификува видот на процесорот, големината на RAM меморијата, се овозможува MMU и се повикува *main* процедурата напишана во *C* за да се стартува главниот дел од оперативниот систем. Во овој дел се спремаат пораките за грешки и отстранување од грешки доколку се појават при подигањето и се внесуваат податочните структури за јадрото (повеќето од нив се со фиксна големина).

Системот започнува автоматска конфигурација, читајќи ги конфигурациските датотеки за влезно/излезните уреди. Доколку уредот одговори тој се додава во табелата на уреди и се лоцираат неговите драјвери (тие треба да се статички поврзани затоа што јадрото не може да ги поврзе динамички).

Откако хардверот ќе биде конфигуриран, следен на ред е процесот 0 кој продолжува со иницијализацијата, монтирање на системот на датотеки и креирање на *init* (процес 1) и демонот за страничење (процес 2).

11.4. Управување со меморијата

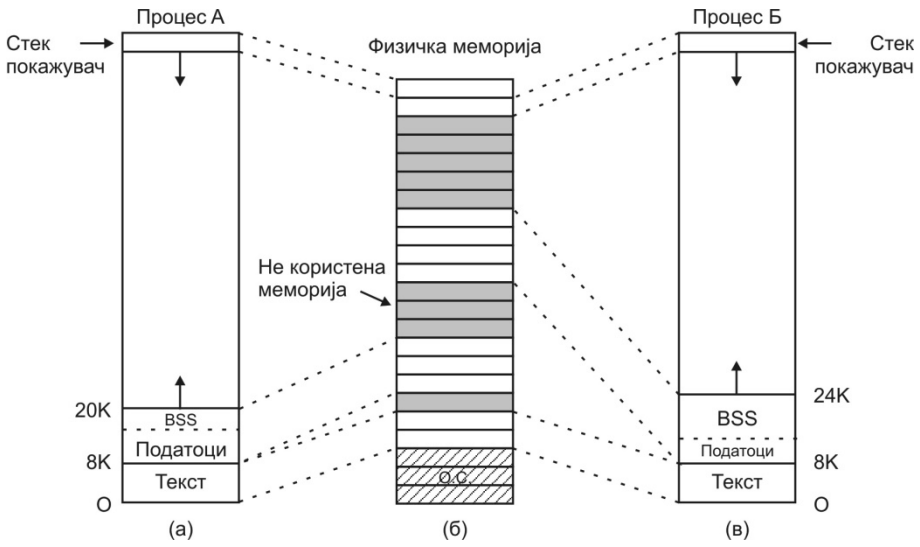
Меморискиот модел на UNIX е доста јасен со цел да може да се имплементира UNIX на компјутери со различни мемориски единици. Секој процес во UNIX има адресно подрачје кое се состои од три сегменти: текст, податоци и стек. Пример за адресното подрачје на еден процес е прикажан на Слика 11.6.

Текст сегментот се состои од машински инструкции кои го сочинуваат кодот кој се извршува. Овој сегмент е само за читање и во него не се прават измени.

Податочниот сегмент претставува склад за променливите, стринговите, низите и другите податоци кои се потребни на процесот. Има два дела, иницијализирани податоци и неиницијализирани податоци (наречени BSS). На Слика 11.6 а), програмскиот текст е 8 KB а и иницијализираните податоци се

8 KB. Неиницијализираните податоци (BSS) се 4 KB. Извршната датотека е само 16 KB (текст + иницијализирани податоци), плус кратко заглавје кое кажува на системот да додели дополнителни 4 KB.

Податочниот сегмент може да се менува, а често и динамички расте или се намалува (функцијата *malloc* во C служи за доделување на меморија).



Слика 11.6: а) Виртуелно адресно подрачје на процес А,
б) физичка меморија, в) виртуелно адресно подрачје на процес Б

Третиот сегмент, стек сегментот, започнува од врвот на виртуелното адресно подрачје. Кога започнува програмата, стекот не е празен. Тој ги содржи сите системски променливи. Повеќето UNIX системи подржуваат и делена меморија на текст сегментот. На Слика 11.6 се забележува дека и двата процеси *А* и *Б* го имаат истиот текст сегмент.

11.4.1. Системски повици за управување со меморијата во UNIX

POSIX не специфицира системски повици за управување со меморијата, поради фактот што оваа тема во голема мера зависи од структурата на компјутерите. Па така проблемот е сведен на тоа што на програмите на кои им е потребен динамичко управување со меморијата може да ја користат *malloc* процедурата (дефинирано со ANSI C стандардот). Како *malloc* е имплементирана е надвор од POSIX стандардот.

Во пракса повеќето UNIX системи имаа системски повици за оваа цел. Најзначајните се прикажани на Слика 11.7. *brk* ја специфицира големината на податочниот сегмент, давајќи ја адресата на првиот бајт после него. Ако

новата вредност е поголема од старата, тогаш податочниот сегмент станува поголем, во спротивно се намалува.

Системски повик	Опис
S = brk(addr)	Промена на големината на податочниот сегмент
A=mmap(addr, len prot, flags, fd, offset)	Пресликување на датотека
S=unmap(addr, len)	Прекин на пресликување на датотеката

Слика 11.7: Системски повици за управување со меморијата

Системските повици *mmap* и *unmap* ги контролираат мемориски пресликаните датотеки. Првиот аргумент на *mmap* ја одредува адресата каде датотеката е мапирана, а вториот аргумент кажува колкава е должината т.е. колку бајтови да мапира. Третиот аргумент ја одредува заштитата (читање, запишување, извршување). *flags* аргументот служи да каже дали датотеката е приватна или делена, петтиот аргумент е дескриптор на датотеката која ќе се мапира додека последниот аргумент кажува кој дел од датотеката да биде мапиран. Другиот повик *unmap* служи да отстранување на мапирана датотека.

11.4.2. Имплементација на управување со меморијата во UNIX

Повеќето UNIX системи се базирани на замена (*англ. swapping*), па така ако повеќе процеси се активни отколку што има меморија, некои од нив се заменуваат на дискот. За таа цел постои заменувач (*англ. swapper*) кој се активира кога нема повеќе слободна меморија:

1. На *fork* повикот му е потребна меморија за процесот дете;
2. *brk* процесот има потреба за проширување на податочниот сегмент;
3. Стекот постанал голем.

Исто така кога треба да се премести процес на тврдиот диск, тој може да биде голем па ако често се преместува ќе биде потребно многу време.

За да избере жртва, заменувачот ги гледа процесите кои се блокирани или чекаат некој влез. Ако се пронајдени повеќе, тогаш се избира оној со најголемо време на престој. Ако нема блокиран процес, тогаш се одредува некој врз база на повеќе критериуми.

На неколку секунди, заменувачот ја изминува листата на процеси кои се заменети за да види дали некој од нив се спремни за извршување. Ако се повеќе тогаш се избира тој што е најдолго време на дискот. Потоа се проверува дали процесот може лесно да се замени т.е. има за него меморија или пак прво треба да се замени некој друг процес.

Страничење во UNIX

Почнувајќи од верзијата 3BSD во UNIX е воведено страничење. Идејата на страничењето е едноставна: процесот не мора цел да е во меморија за да се извршува. Сè што е потребно е корисничка структура и табела на страници. Страниците на текстот, податоците и стекот се донесуваат динамички во меморијата, една по една, како што од нив има потреба. Ако корисничката структура и табелата на странци не се во меморијата, тогаш процесот не може да се изврши.

Страничењето е имплементирано од јадрото и од процесот *„page daemon“* (демон за страничење - процес 2). Како сите демони и овој се стартува периодично и тогаш ја набљудува ситуацијата и превзема дејства.

Главната меморија во 4BSD се состои од три дела. Првите два дела (јадрото и „*core map“*) никогаш не излегуваат од меморијата. Остатокот од меморијата е поделен на рамки за страници и секоја може да содржи текст, податоци, страна на стек, страна на табела на страници или да биде на слободната листа.

Кога процесот е стартуван, може да се јави грешка на страничење бидејќи една или повеќе страни не се присутни во меморијата. Ако се појави ваква грешка, тогаш се зема првата рамка од слободната листа, се трга од листата и се вчитува страната во неа. Ако листата е празна, процесот е суспендиран додека демонот на страничење не ослободи рамка на страница.

Алгоритам за замена на страници

Алгоритамот за замена на страници се извршува од демонот за страничење. На секои 250 msec се буди за да го провери бројот на слободни рамки за страници дали е помал или еднаков на системскиот параметар *„lotsfree“* (вообичаено е ¼ од меморијата). Ако има недостаток од слободни рамки тогаш започнува да трансферира страни од меморијата на дискот. Ако има повеќе слободно место, тогаш демонот се враќа да спие.

Демонот користи модифицирана верзија на алгоритамот на часовник. Тој е глобален алгоритам и кога се отстранува страна не се води грижа чија страна се отстранува. Па така бројот на страници за секој процес варира.

Оригиналниот Berkeley UNIX го користи основниот алгоритам на часовник, но подоцна е откриено дека со поголеми мемории не работи добро (трае премногу долго). Потоа алгоритамот е модифициран во дво-рачен алгоритам на часовник. Во овој алгоритам демонот користи два покажувачи. Кога се извршува, прво го брише битот за користење на почетокот а потоа го проверува битот за користење на крајот, после што ги поместува двете стрелки. Ако двете стрелки се блиску, тогаш само многу тешко користените страни имаат голема шанса да бидат пристапени помеѓу времето да предната рака помине а потоа и задната. Ако двете раце се 359 степени одвоени (задната рака само што ја поминала предната рака), се враќа на оригиналниот

алгоритам на часовник. Секогаш кога демонот се извршува, рацете ротираат помалку од полна револуција, во зависност колку треба да одат за да се постигне параметарот *„lotsfree“*.

Во 4BSD алгоритамот за замена е следен. Прво, заменувачот бара процес кој е неактивен повеќе од 20 sec. Ако постои, тогаш тој со најголемо неактивно време ќе биде заменет. Ако не постои, четирите најголеми процеси се изминуваат и тој што е најмногу во меморијата се заменува. И така алгоритамот се повторува додека не се ослободи доволно простор.

Страничењето во System V е слично со тоа во 4BSD, но постојат две разлики. System V го користи оригиналниот алгоритам на часовник со една рака. Наместо со ставање на некористена страна на слободната листа на второто поминување, страната се става само ако не е користена во n последователни изминувања. Со овие измени алгоритамот не ослободува страници брзо како Berkeley алгоритамот, но ја зголемува можноста да страница која еднаш е ослободена нема да биде потребна во блиска иднина. Овде, наместо една променлива *lotsfree*, System V има две променливи *min* и *max* во зависност од кои се активира демонот.

12. Windows

Windows претставува модерен оперативен систем кој работи на модерни десктоп компјутери или сервери. Неговиот развој поминал низ многу фази и во него денес се имплементирани многу напредни софтверски решенија. Поради многуте предности кои ги има овој оперативен систем, денеска тој се користи на најголем број од компјутерите во светот.

12.1. Историја на оперативните системи Windows

Оперативните системи на Microsoft за десктоп и лаптоп компјутери можат да се поделат на три групи: и тоа MS DOS, Windows (Windows 95/98/Me) за обични корисници и Windows NT за сервери.

12.1.1. MS-DOS

Во 1981 година најголемата и најмоќната компјутерска компанија IBM го произведе компјутерот базиран на 8088. Компјутерот беше опремен со 16 битен командно линиски ориентиран оперативен систем кој работи во реален мод и подржува еден корисник, наречен MS-DOS 1.0. Оперативниот систем беше направен од Microsoft, познат во тоа време по неговиот интерпретер на BASIC користен на 8088 и Z-80 системите. Овој оперативен систем се состоеше од 8 KB резидентен мемориски код, мал оперативен систем за 8 битните 8080 и Z-80 процесорски единици. Две години подоцна беше објавен далеку помоќниот 24 KB оперативен систем, MS-DOS 2.0.

Кога INTEL го произведе 286 чипот, IBM произведе нов компјутер со него, PC-AT (*AT - Advanced Technology*) во 1986. Технологијата на 286 работеше во тоа време на импресивните 8 MHz и можеше да адресира, иако со големи тешкотии, 16 MB RAM. PC-AT беше опремен со DOS 3.0 кој беше 36 KB. Со години MS-DOS продолжи да се развива но беше и остана командно линиски ориентиран оперативен систем.

12.1.2. Windows 1.0/3.11/95/98/Me

Инспирирани од корисничкиот интерфејс на Apple Lista, оперативниот систем на Apple Macintosh, Microsoft решава да му даде на MS-DOS графички кориснички интерфејс кој го нарекува Windows. Windows 1.0 објавен во 1985 не беше ништо особено. Дури Windows 3.0 кој беше направен за системите 386 (објавен 1990), а особено неговите наследници 3.1 и 3.11, беа прифатени и доживеа огромен комерцијален успех. Ниедна од овие рани верзии на Windows не беа вистински оперативни системи, туку повеќе беа графички интерфејс на MS-DOS, кој сè уште ја контролираше машината.

Објавата на Windows 95 во август 1995 година сè уште целосно не го елиминира MS-DOS, иако ги префрли скоро сите делови од MS-DOS делот на Windows делот.

Како додаток на старите 16 битни асемблерски кодови во јадрото, Windows 98 има и други два сериозни проблеми. Како прво, ако некој процес манипулира со податочната структура од јадрото и ќе му помине доделеното време и некој друг процес ќе почнеше со работа, постои опасност новиот да ја најде податочната структура во изменета состојба. За спречување на проблем од овој вид, по влегувањето во јадрото, повеќето процеси прво требаа да обезбедат огромен *mutex* кој го покрива целиот систем пред да можат било што да работат. Овој пристап го елиминира претходниот проблем но во исто време ги елиминира и повеќето од придобивките на мултипрограмирањето. Вториот проблем што Windows 98 го има е тоа што процесот има 4 GB виртуелен адресен простор. Од овие 2 GB беа комплетно на располагање на процесот. Но затоа, следниот 1 GB беше поделен помеѓу сите останати процеси во системот. Последните 1 MB се делени помеѓу сите процеси за да им се дозволи пристап до MS-DOS векторите за прекини. Ова делење е често користено од Windows 98 апликациите. Случајна грешка во некој од процесите може да ги избрише клучните податочни структури и да доведе до паѓање на сите процеси.

12.1.3. Windows NT

Во доцните осумдесетти Microsoft согледа дека градењето на модерен 32 битен оперативен систем врз 16 битна подлога најверојатно не е најдобриот избор. Најмен е *David Cutler* да работи на создавање на нов 32 битен оперативен систем од почеток до крај. Овој нов систем кој подоцна е наречен Windows NT (*Windows New Technology*) беше наменет за критични бизнис апликации исто како и за домашни корисници. Во тоа време изгледало визионерски дека некој ќе користи персонален компјутер за бизнис но историјата покажа дека биле во право. Сигурноста и високата доверливост, кои изостанувале на верзиите на Windows базирани на MS-DOS, биле на прво место во креирањето на NT.

Проектот беше успешен и првата верзија наречена Windows NT 3.1 е објавена во 1993 година. Бројот во името на овој систем намерно е одберено како и 16 битниот Windows во тоа време затоа што во Microsoft очекувале дека NT ќе го замени Windows 3.1 заради својата техничка супериорност.

Изненадувачки било што повеќето корисници останале на старите 16 битни системи, отколку да преминат на непознат 32 битен систем за кој не знаеле како ќе се покаже. Понатаму NT барал многу повеќе меморија од Windows 3.1 и исто така немало 32 битни програми во тоа време. Пропаста на Windows NT 3.1 беше причината Microsoft да се реши да направи 32 битна верзија на Windows 3.1 наречена Windows 95. Продолжената одбојност спрема NT го присили Microsoft да го издаде Windows 98 и на крај Windows Me.

Првата поголема надградба на NT дојде со NT 4.0 во 1996. Системот имаше моќ, сигурност, доверливост на новиот оперативен систем, а исто така го поддржуваше истиот кориснички интерфејс од Windows 95. Оваа компатибилност овозможи полесна миграција од Windows 95 на NT.

12.1.4. Windows 2000

Изданието на NT кое требаше да излезе по NT 4.0 прво требаше да се вика NT 5.0 . Microsoft го промени името во Windows 2000, во главно за да проба да им се понуди неутрално име на корисниците на Windows 98 и NT и за да во него видат логичен чекор напред.

Бидејќи Windows 2000 всушност е NT 5.0 тој наследил доста од својствата на неговиот NT претходник. Тој е вистински 32 битен систем со мултипрограмирање со индивидуално заштитени процеси. Секој процес има приватен 32 битен виртуелен адресен простор. Оперативниот систем работи во мод на јадро додека корисничките процеси работат во кориснички мод. Процесите можат да имаат една или повеќе нишки кои се видливи и се распоредуваат од оперативниот систем. Исто така има потполна поддршка за работа со повеќе симетрични мултипроцесори до 32 процесорски единици.

Една работа што Windows 2000 ја нема е MS-DOS. Постои командно линиски интерфејс, но ова е сосема нова 32 битна програма кој задржува некои од функционалности на MS-DOS, но исто така има и свои.

12.2. Структура на оперативниот систем

Windows 2000 се состои од два главни дела: самиот оперативен систем кој работи во мод на јадро и околинските подсистеми кои работат во кориснички мод. Јадрото е традиционално јадро во смисла дека раководи со процесите, меморијата, системот на датотеки и друго.

Едно од многуте подобрувања во однос на Windows 3.1 е неговата модуларна структура. Тој се состоел од големо јадро кое работело во мод на јадро плус некои серверски процеси кои работеле во кориснички мод. Корисничките процеси комуницирале со серверските процеси користејќи клиент-сервер модел. Ова овозможило Windows 2000 да се имплементира освен на INTEL и на други машини.

Сепак постои некаква структура и во Windows 2000. Поделен е на неколку слоеви од кои секој ги користи сервисите од слојот под него. Тоа е прикажано на Слика 12.1.

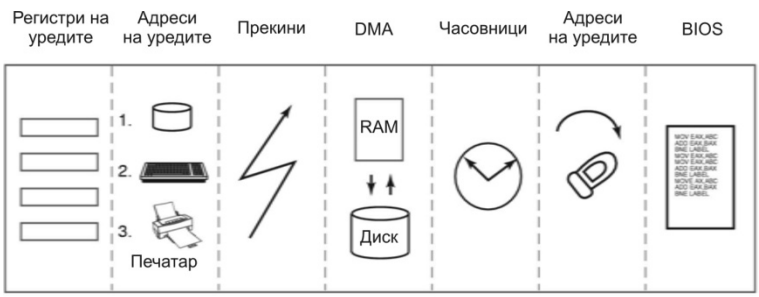
Најниските два слоја се HAL и јадро и се напишани во C и во асемблер и се делумно зависни од машината. Погорните слоеви скоро сите се напишани во C и се независни од машината.

<table>
<tr><td>Сервиси</td><td>POSIX програми</td><td>Win32 програми</td><td>OS/2 програми</td><td rowspan="3">Кориснички мод</td></tr>
<tr><td></td><td>POSIX подсистем</td><td>Win32 подсистем</td><td>OS/2 подсистем</td></tr>
<tr><td colspan="4">Системски интерфејс (NT, DLL)</td></tr>
<tr><td colspan="4">Системски сервиси</td><td rowspan="5">Мод на јадро</td></tr>
<tr><td colspan="4">Менаџери (В/И, објектен, процесен, мемориски, напонски...),
Win32 GDI, видео драјвери, датотечен систем</td></tr>
<tr><td colspan="4">Јадро</td></tr>
<tr><td colspan="4">HAL</td></tr>
<tr><td colspan="4">Хардвер</td></tr>
</table>

Слика 12.1: Структура на Windows 2000

12.2.1. HAL (Hardware Abstraction Layer)

Една од главните цели на Windows 2000 е да се направи преносливна различни платформи. Погорните слоеви можат да се направат комплетно преносливи (заради тоа што работат со податочни структури), но со пониските слоеви кои работат со регистрите на уредите, прекините, DMA и останатите можности на хардверот не е така заради разликите кои се јавуваат од машина до машина. Потполно свесни за овој проблем во Microsoft направија сериозен обид да сокријат многу од машинските зависности во тенок слој наречен HAL (*Hardware Abstraction Layer*).



Слика 12.2: Некои од хардверските функционалности на HAL

Задачата на HAL е да му прикаже на останатиот дел од оперативниот систем апстрактни хардверски уреди. Сервисите кои се избрани за HAL се тие кои се поврзани со чипот на матичната плоча и кои се разликуваат од машина до машина во разумно предвидливи граници. HAL сервисите вклучуваат пристап до регистрите на уредите, магистрално независното адресирање на уредите,

управување со прекини и рестартирање, DMA трансфери, контрола на тајмерите и часовникот, мултипроцесорска синхронизација, интерфејс со BIOS и неговата CMOS конфигурациска меморија. Некои од хардверските функционалности се прикажани на Слика 12.2.

12.2.2. Јадро

Над слојот HAL се наоѓа слојот кој го содржи тоа што во Microsoft го викаат јадро. Улогата на јадрото е да се направи остатокот од оперативниот систем независен од хардверот и на тој начин преносliв. Тој превзема таму каде што HAL завршува. Тој му пристапува на хардверот преку HAL и се надградува над сервисите на HAL кои се на екстремно ниско ниво, конструирајќи така апстракции од повисоко ниво.

Кодот за управување со нишки исто така се наоѓа во јадрото. Во прилог на понатамошна апстракција на хардверот и преминување од нишка на нишка, јадрото исто така има и друга клучна функција и тоа обезбедување на две класи на објекти: контролни и диспечерски објекти. Контролните објекти се оние кои го контролираат системот, објекти на примитивни процеси, објекти на прекини и два помалку чудни објекти наречени DPC и APC. DPC (*Deferred Procedure Call*) објектот се користи да се подели делот од процедурата на сервисот за прекини за кој времето не е критично од оној за кој времето е критично. APC (*Asynchronous Procedure Call*) е контролен објект на јадрото кој е сличен на DPC со таа разлика што овие се извршуваат во контекст на специфичен процес. Друг вид на објекти во јадрото се диспечерските објекти. Овде спаѓаат семафори, mutex, настани, чекачки тајмери и други објекти на кои нишките можат да чекаат.

12.2.3. Извршен слој

Над јадрото и драјверите на уредите е горниот дел на оперативниот систем наречен извршен слој (*англ. executive*). Извршниот слој е напишан во *C*, е независен од архитектурата и може да се префрлува на други машини со многу малку напор. Се состои од 10 компоненти, од кои некои се само колекции од процедури кои работат заедно за да постигнат некоја цел.

Објект менаџерот ги управува сите објекти кои му се познати на оперативниот систем. Тука се вклучени процесите, нишките, датотеките, директориумите, семафорите, влезно излезните уреди, тајмерите и многу други. Влезно излезниот менаџер обезбедува рамка за менаџирање на влезно излезните уреди и обезбедува генерички влезно излезни сервиси. Процесниот менаџер оперира со процесите и нишките, вклучувајќи го тука и нивното креирање и уништување. Мемориските менаџер ја имплементира Windows 2000 виртуелна мемориска архитектура. Го управува мапирањето на виртуелните страници во физички странични рамки. Безбедносниот менаџер го работи безбедносниот

механизам на Windows 2000. Кеш менаџерот ги чува диск блоковите кои се најскоро користени во меморијата, за да го забрза пристапот до нив во случај да се потребни повторно. *Plug and play* менаџерот ги добива сигналите за сите ново прикачени уреди. Менаџерот за моќ се грижи за напојувањето. Менаџерот на локални процедури обезбедува високо ефикасни интерпроцесни комуникации кои се користат помеѓу процесите и нивниот подсистем. На врвот од овој слој се системските сервиси. Тоа е тенок слој чија функција е да обезбеди интерфејс до извршниот слој.

12.2.4. Драјвери на уредите

Последниот дел од оперативниот систем го сочинуваат драјверите на уредите. Секој драјвер може да контролира еден или повеќе влезно/излезни уреди, додека драјвер на уредот може да работи и работи кои не се директно поврзани со одреден уред како што се енкрипција на податочен тек или обезбедување пристап до структура на податоци во јадрото. Тие не се дел од ntoskrnl.exe бинарната датотека. Предноста на овој пристап е во тоа што драјверот е инсталиран на системот, е додаден на листа во регистарот и се внесува динамички кога системот се подига. На овој начин ntoskrnl.exe е ист за сите додека секој систем се конфигурира прецизно во однос на тоа кои уреди ги содржи.

12.2.5. Имплементација на објекти

Објектите претставуваат најважен концепт во Windows 2000. Тие обезбедуваат еднаков и постојан пристапен интерфејс до сите системски ресурси и податочни структури како што се процеси, нишки, семафори и други. Униформноста се огледа во повеќе аспекти. Прво, сите објекти се именувани и им се пристапува на ист начин со користење на ракувачите за објекти. Второ, сите пристапи до објектите одат преку менаџерот на објекти. На овој начин е возможно сите сигурносни проверки да се стават на едно место и да се обезбеди ниту еден процес да не може да ги заобиколи. Трето, делењето на објектите помеѓу процесите може да се изврши на униформен начин. Четврто, заради тоа што сите објекти се отвораат и затвораат преку објектниот менаџер, возможно е да се следат објектите и да се види кои се сè уште во употреба а кои може безбедно да се избришат. Петто, униформниот модел за менаџирање на објекти прави менаџирањето да биде едноставно.

Секој објект содржи заглавие со некои информации за сите објекти од сите видови. Полињата во заглавието ги содржат името, директориумот во кој тој е формиран, сигурносни информации и листа од процеси со покажувачи до објектот. Секое заглавие на објект исто така користи поле кои ги одредува трошоците за отворање на објектот.

Објектите зафаќаат скапо место во меморискиот простор на јадрото, затоа кога веќе не ни е потребен одреден процес би требало да се отстрани и да се ослободи адресниот простор. Објектите се типизирани што значи дека секој од нив има одредени својства кои се заеднички за сите објекти од неговиот тип.

Извршните компоненти можат динамички да формираат нови типови на објекти. Портите, тајмерите и редовите се поврзани со комуникација и синхронизација. Портите се канали за комуникација помеѓу процесите. Тајмерите овозможуваат начин за блокирање на одреден временски интервал. Редовите се користат да им јават на нишките дека претходно почнатата асинхрона влезно/излезна операција е завршена.

Секциите се објекти користени од меморискиот систем за работа со мемориски мапирани датотеки. Тие запишуваат која датотека или нејзин дел е запишан на која мемориска локација. Клучевите се регистарски клучеви. Симболичките врски овозможуваат име во еден дел на објектниот простор да покажува на име во друг дел од објектниот простор. Корисниците можат да креираат нови објекти со повикување на *CreateSemaphore* или *OpenSemaphore*. Ова се повици до процедури од библиотеките кои резултираат во адекватни системски повици.

12.3. Процеси и управување со процеси

Windows има голем број на концепти за управување на централната процесорска единица и групирање на ресурсите. Во наредното поглавје ќе биде продискутирано и покажано како се имплементирани овие концепти.

12.3.1. Фундаментални концепти

Windows ги подржува традиционалните процеси кои можат да комуницираат помеѓу себе исто како што можат и во UNIX. Секој процес содржи барем по една нишка, која пак содржи барем по едно влакно. Понатаму процесите можат да бидат групирани во задачи заради некои цели за менаџирање на ресурсите. Кратко објаснување на овој концепт е дадено на Слика 12.3.

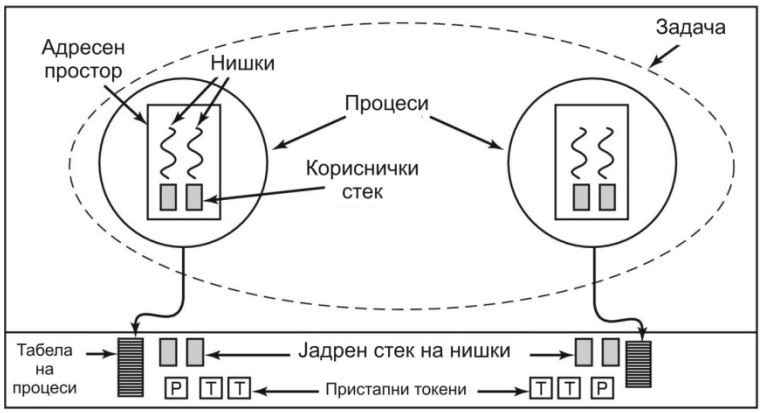
Име	Опис
Задача	Колекција на процеси кои делат квоти и граници
Процес	Контејнер кој држи ресурси
Нишка	Ентитет кој се распределува од јадрото
Влакно	Лесна нишка менаџирана во корисничкиот простор

Слика 12.3: Основни концепти кои се користат за управување на ЦПЕ и ресурсите

Процесите се контејнери за ресурсите. Секој процес може да има 4 GB адресен простор од кои корисникот ги добива долните два, а оперативниот систем

останатото. Процесот содржи идентификација, една или повеќе нишки, листа на „*handles*“ и пристапни токени (*англ. access token*) кој ги содржи безбедносните информации. Секој процес започнува со една нишка, но подоцна може динамички да се формираат и нови. Нишките ја формираат основата на процесорското распоредување затоа што оперативниот систем секогаш бира која нишка ќе работи а не кој процес. Следствено на тоа, секоја нишка се наоѓа во некоја состојба (подготвен, се извршува, блокиран ...), додека процесите пак немаат состојби. Секоја нишка има два стека, еден за корисничкиот еден за нагледниот начин. Битно е да се забележи дека нишките се распоредувачки концепт, а не концепт на поседување на ресурси. Секоја нишка може да пристапи до секој објект што припаѓа на нејзиниот процес.

Освен што постојат нишки кои работат во рамките на кориснички процеси, Windows има бројни демон нишки кои работат исклучиво во просторот на јадрото и не се поврзани со ниту еден кориснички процес. Релацијата помеѓу задачите, процесите, нишките и влакната е покажана на Слика 12.4.



Слика 12.4: Релации помеѓу задачи, процеси и нишки

Нови процеси се креираат со користење на Win32 API повикот *CreateProcess*. Windows не применува ниеден вид на родител-дете хиерархија. Сите формирани процеси се еднакви. Секој процес во Windows е креиран со една нишка, но тој сепак може подоцна да формира нови. Креирањето на нишките е поедноставно.

12.3.2. Интерпроцесна комуникација

Нишките можат да комуницираат на многу начини вклучувајќи тука и цевки, поштенски слотови, сокети, повици кон оддалечени процедури (*RPC- Remote Procedure Call*) и делени датотеки. Цевките имаат два начини на работа: бајтови и пораки, селектирани во времето на креирањето. Поштенските слотови се карактеристика на Windows. Тие се слични со цевките на некој

начин но сепак се разликуваат во целост. На пример, тие се еднонасочни додека цевките се двонасочни.

Далечински повици на процедури се начин процесот А да направи процесот Б да повика процедура во својот адресен простор заради потреба на А и резултатот да го проследи до процесот А.

Како што Windows обезбедува бројни интерпроцесни комуникациски механизми, обезбедува и синхронизациски механизми како што се: семафори, *mutex*-и, критични региони и настани. Сите овие механизми работат со нишки а не со процеси.

12.3.3. Распоредување

Windows нема централна нишка за распоредување. Наместо тоа, кога нишката не може да работи повеќе, таа влегува во мод на јадро и го стартува распоредот да види на која нишка да се префрли. Оваа состојба предизвикува нишката која се извршува да го изврши кодот на распоредот.

1. Нишката блокира семафор, mutex, настан;
2. Сигнализира на објект;
3. Временскиот квантум за работа истекува.

Во првиот случај нишката веќе е во мод на јадро за да изврши операција на диспечер или влезно излезен објект. Не може да продолжи со работа, затоа го снима својот контекст, го извршува кодот на распоредот за да избере наследник и го внесува контекстот на таа нишка за да продолжи со работа.

Во вториот случај, нишката исто така е во мод на јадро. Сепак, по сигнализирање на објект, таа може да продолжи со работа бидејќи сигнализирањето на објект не блокира. Сепак, нишката треба да го изврши распоредот, да види дали како резултат на нејзината акција се појавила нишка со повисок приоритет која е спремна за извршување. Ако е така се појавува смена на нишка.

Во третиот случај, се појавува замка во модот на јадрото, во време во кое нишката го извршува кодот за распоредување за да види која се извршува следна.

Распоредувачот се повикува и во два други случаи:

1. Ако е завршена влезно излезна операција;
2. Поминало времето за чекање.

Ќе биде разгледан алгоритамот за распоредување.

Прво, тука е повикот *SetPriorityClass* кој ја поставува класата на приоритети на сите нишки во повикувачкиот процес. Дозволени вредности се: *realtime, high, above normal, normal, below normal* и *idle*.

Второ, се повикува *SetThreadPriority* кој ги поставува релативните приоритети на некои нишки во споредба со другите нишки во процесот. Дозволени вредности се: *time critical, highest, above normal, normal, below normal, lowest* и *idle*. Со 6 класи за процесите и седум за нишките, нишката може да има било која од 42 комбинации. Оваа е влезот во алгоритамот за распоредување.

Распоредувачот работи вака. Системот има 32 приоритети нумерирани од 0 до 31. 42 комбинации се мапирани во 32 приоритетни класи. Бројот во табелата го определува базниот приоритет на нишката. Како додаток, секоја нишка има моментален приоритет, кој може да биде повисок (не и понизок) од базниот.

За да ги користи овие својства за распоредување, системот одржува низа со 32 влеза, кои кореспондираат со приоритетите од 0 до 31. Основниот алгоритам за распоредување се состои од пребарување на низата од приоритет 31 до 0. Штом ќе се најде слот кој не е празен нишката се избира и работи еден временски квантум. Кога ќе истече квантумот, нишката оди на крајот од редот и нишката пред неа се избира за следна.

Со тек на времето додадени се некои додатоци на базичниот алгоритам за да се подобрат системските перформанси. Под одредени услови, моменталниот приоритет може да биде подигнат над базниот приоритет. Windows 2000 подржува 32 приоритети за нишки. Кај Windows 2000 Professional квантумот за извршување е 20 msec; на еднопроцесорски сервери е 120 msec, додека на мултипроцесорски варира зависно од фреквенцијата на системскиот часовник.

12.3.4. Подигање на Windows

Пред да се стартува, Windows, треба да се подигне. Процесот на подигнување креира иницијални процеси кои го подигаат системот. Во овој дел накратко ќе разгледаме како се одвива процесот на подигнување кај Windows. Хардверскиот процес на подигнување се состои од читање на првиот сектор од првиот диск (*Master Boot Record*) и префрлување на него. Оваа кратка програма пишувана во асемблер ја чита табелата на партициите за да види на која од нив е сместен оперативниот систем. Кога ќе ја пронајде оваа партиција, го чита првиот сектор на таа партиција, наречен *boot* сектор и се префрлува на него. Програмот во овој сектор го наоѓа *root* директориумот на оваа партиција и ја бара датотеката *ntldr* и ако ја пронајде ќе ја внесе во меморијата и ќе ја изврши. Оваа датотека го внесува Windows.

Ntldr сега ја чита датотеката наречен boot.ini, која е единствената конфигурациска датотека која не е сместена во регистарот. Таа ги листа сите верзии на hal.dll и ntoskrnl.exe кои се на располагање за подигнување на оваа партиција. Оваа датотека исто така обезбедува многу параметри како што се:

колку процесори и колку меморија ќе се користат, дали на корисничките процеси да се дадат 2 GB или 3 GB и на која фреквенција да се постави часовникот. Тогаш ги избира и вчитува hall.dll и ntoskrnl.exe, како и bootvid.dll која е видео драјвер која се користи за приказ на екранот при вчитувањето. Потоа Ntldr чита од регистарот за да види кои драјвери се потребни за да се подигне системот. На крај ги чита и ја предава контролата на драјверите на ntoskrnl.exe.

Кога е стартуван, оперативниот систем прави генерална иницијализација и повикува извршни компоненти да извршат нивна иницијализација. Се на се, се извршуваат десетици чекори, за време на кои прогрес барот расте на екранот како што чекорите се извршуваат. Последен чекор е креирањето на првиот вистински кориснички процес, менаџерот на сесијата smss.exe. Кога ќе се подигне овој процес, подигнувањето е завршено. Сесискиот менаџер ја повикува Win32 околината ја стартува датотеката csrss.exe.

Сега оперативниот систем е подигнат и работи. Сега е време за подигање на сервисните процеси и да се дозволи на корисниците да се најават. За таа цел се активира Winlogon.exe. Тој креира менаџер за автентикација (lsass.exe), а потоа се стартува родителскиот процес на сите сервиси (services.exe). Од services.exe потоа се формираат сите сервиси. Winlogon.exe е исто така одговорен за најавувањето на корисниците (msgina.dll), при кое ги чита корисничкиот профил од регистарот. Десктопот е всушност explorer.exe со некои поставени опции.

12.4. Меморија

Windows има екстремно софистициран виртуелен мемориски систем. Има голем број на Win32 функции за негово користење и исто така користи и дел од извршните заедно со 6 јадрени нишки за управување со мемориският систем. Процесите прикажани на Слика 12.5. се стартуваат во фазата на подигнувањето. Тие кои се над линијата се секогаш стартувани, додека оние под линијата се примери за сервиси кои може да бидат стартувани.

12.4.1. Фундаментални концепти

Во Windows, секој кориснички процес има свој виртуелен адресен простор. Виртуелните адреси се долги 32 бита, значи дека секој процес има 4 GB виртуелен адресен простор. Долните 2 GB минус околу 256 MB се на располагање на кодот и податоците на процесот, додека горните 2 GB мапирани од јадрото. Виртуелниот адресен простор е страничен, со фиксна големина на страниците (4 KB кај Pentium).

Долните и горните 64 KB на виртуелниот адресен простор за секој процес не се мапирани. Ова е намерно направено за да се помогне да се фатат програмските грешки.

Процес	Опис
idle	Не е процес, но е основа на idle нишката
system	Ги креира smss.exe и датотеките за страничење, чита од регистарот, отвара DLL
smss.exe	Првиот реален процес, иницијализација, ги креира csrss и winlogon
csrss.exe	Win32 подсистемски процес
winlogon.exe	Демон за логирање
lsass.exe	Менаџер за автентикација
services.exe	Ги стартува сервисите
Printer server	Дозволува одалечени задачи да го користат печатачот
File server	Ги услужува барањата за локалните датотеки
Telnet daemon	Дозволува најавување од далечина
Incoming email hanler	Прифаќа и зачувува дојдовни е-пошти
Incoming fax handler	Прифаќа и печати дојдовини факсови
DNS resolver	Интернет DN сервер
Event logger	Логира различни системски настани
P&P manager	Го надгледува хардверот

Слика 12.5: Процеси кои се стартуваат при покренувањето

Почнувајќи од 64 KB, доаѓа приватниот кориснички код и податоци. Ова се протега до скоро 2 GB. Последниот дел од долните 2 GB содржи некои системски бројачи и тајмери кои се делени од сите корисници само за читање.

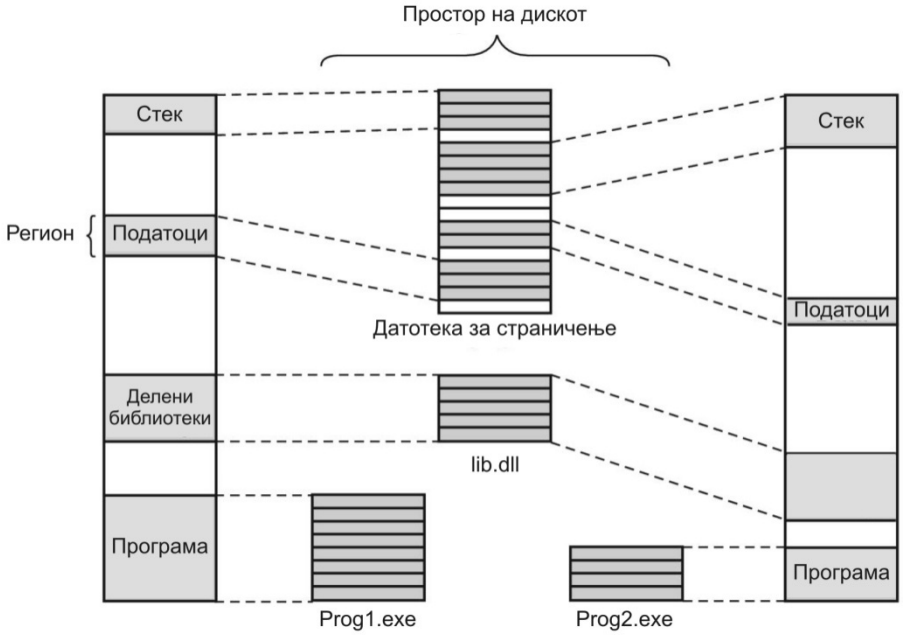
Горните 2 GB го содржат оперативниот систем, вклучувајќи ги тука кодот, податоците и страничените и нестраничените простори. Горните 2 GB се делени помеѓу сите кориснички процеси освен табелите со страници, кои се за секој процес посебно.

Секоја виртуелна страница може да биде во една од следните три состојби: слободна (*free*), резервирана (*reserved*) и посветена (*committed*). Слободна страница моментално не е во употреба и нејзино референцирање доведува до грешка. Кога процесот е стартуван сите страници се во слободна состојба, додека програмата и иницијалните податоци не се мапирани во адресниот простор. Кога кодот или податоците се мапирани во страница тогаш за неа велиме дека е „*committed*“. Референца до „*committed*“ страница е мапирана со користење на хардверот за виртуелна меморија и е успешна ако страницата е во главната меморија. Ако страницата не е во главната меморија доаѓа до грешка. Виртуелната страница уште може да биде во резервирана состојба, што значи дека не може да биде мапирана додека резервацијата не е експлицитно тргната.

Windows дозволува датотеките да бидат мапирани директно во региони на виртуелниот адресен простор. Откако датотеката е мапирана во адресниот простор, може да биде читана или запишувана користејќи обично мемориско референцирање. Мемориски мапираните датотеки се имплементираат на ист

начин како и останатите „*committed*“ страници, само „*shadow*“ страниците се во корисничката датотека наместо во датотеката за страничење.

Windows експлицитно дозволува два или повеќе процеси да се мапираат на истиот дел од истата датотека, најверојатно на друга виртуелна адреса како што е прикажано на Слика 12.6.



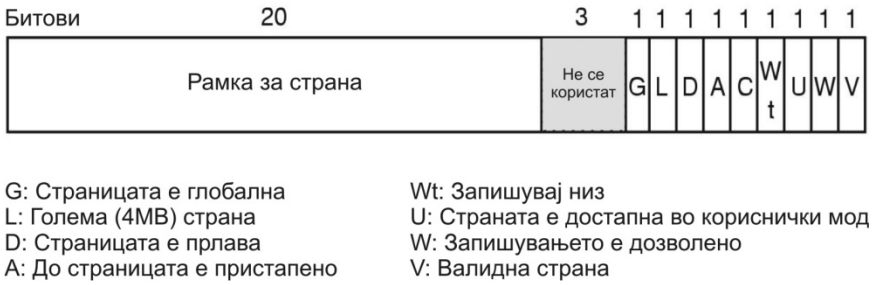
Слика 12.6: Мапирани региони со нивните „shadow“ страници на дискот

Со читање и запишување на мемориски зборови, процесите може да комуницираат едни со други и да си разменуваат податоци со многу голем проток, бидејќи не е потребно копирање. Различните процеси можат да имаат различни дозволи за пристап. Бидејќи сите процеси ја користат делената мапирана датотека, промените направени од еден од нив им се видливи на останатите дури и ако датотеката на дискот сè уште не е променета. Исто така водено е грижа ако друг процес ја отвори датотеката за нормално читање, да го види од RAM меморијата а не од дискот.

Вредно е да се спомене дека има проблем ако два програми делат DLL датотека и ако едниот од нив направи промена во статичките податоци на датотеката. Проблемот е решен со мапирањето на сите страници како дозволени само за читање само што тајно е споменато дека некои се и за запишување. Кога се случува запишување на страница која е мапирана како за читање, а таа е всушност допуштена и за запишување, се прави приватна копија од оваа страница и се мапира.

12.4.2. Разрешување на грешка на страничење

Windows не користи ниту една форма на пред-страничење. Кога некој процес ќе се стартува ниту една од неговите страници не се во меморијата. Сите тие се донесуваат динамички кога ќе се случи грешка на страничење (*англ. page fault*). На секоја грешка, јадрото гради машински независен дескриптор кој кажува што се случило и го предава тоа на извршниот дел на мемориските менаџер. Тогаш мемориските менаџер ја проверува валидноста. Ако страницата припаѓа во *„committed“* или *„reserved“* регионот, ја бара адресата во VAD, ја наоѓа или креира табелата за страници и бара релевантен внос. Записите во табелите за страничење се различни за различни архитектури. За Pentium е прикажан на Слика 12.7.



Слика 12.7: Табела на страничење за мапирани страни на Pentium

Најважни битови во вносот во табелата за страници од аспект на алгоритамот за страничење се A и D битовите. Тие се пополнуваат од хардверот и кажуваат дали страната е референцирана или е запишувано во неа.

Овие грешки се делат на 5 категории:

1. Референцираната страница не е *„committed“*;
2. Дошло до прекршување на заштитата;
3. Запишано е во делена страница;
4. Потребно е стекот да расте;
5. Референцираната страница е „*committed“,* но моментално не е мапирана.

Првиот и вториот случај се фатални грешки од кои нема опоравување за процесот. Третиот случај ги има истите симптоми како и вториот но третманот е различен. Решението е во копирање на страницата во физичка странична рамка и да се мапира таа како дозволена за читање и запишување. За четвртиот случај потребно е алоцирање на нова странична рамка и нејзино мапирање. Петтиот случај е нормална грешка на страничење.

12.4.3. Алгоритам за замена на страниците

Замената на страниците работи на следниот начин. Системот се труди да одржува некој одреден број на слободни страници во меморијата така да кога ќе дојде до грешка, да може да се додели слободна страница, без да треба таа прво да се запише на диск.

Секако, страниците на слободната листа мораат да доаѓаат од некаде, така да вистинскиот алгоритам за замена на страниците е како тие да се одземат од процесите и да се стават на слободната листа.

Во Windows целиот систем за страничење користи концепт на работен сет. Секој процес има работен сет. Овој сет се состои од мапирани страници кои се во меморијата и можат да бидат рефенцирани без грешка на страничење.

Работниот сет на секој процес е опишан со два параметри: минималната и максималната големина. Ова не се цврсти граници па процесот може да има и помалку страници во меморијата од минимумот или во некој случај и повеќе од максимумот. Секој процес стартува со истиот минимум и максимум, но овие граници можат со тек на време да се сменат.

Ако се случи грешка на страничење и работниот сет е помал од минимумот, се додава страница. А ако се случи грешка на страничење а работниот сет е поголем од максимумот, треба да се извади страница за да се направи место за новата. Овој алгоритам значи дека Windows користи логичен алгоритам, за да спречи еден процес да им наштети на другите заземајќи меморија. Сепак, системот може самиот да се преподеси, ако на пример види дека некој процес страничи многу а другите не, да му го зголеми максимумот на работниот сет.

Менаџерот за балансирање на сетовите, проверува дали има доволно слободни страници. Ако нема, тогаш ја стартува нишката на менаџерот за работни сетови, за да ги испита работните сетови и да извлече уште страни. Ги испитува процесите, така што големите процеси кои не работат имаат предност пред малите активни процеси.

Потоа менаџерот за работни сетови ги испитува процесите. Ако работниот сет на процесот има помалку страници од минимумот или направил поголем број грешки на страничење од последното испитување се остава. Во друг случај се одземаат една или повеќе страници. Целниот број на страници кои треба да се одземат е сложена функција од големината на RAM меморијата, колку е тесна меморијата, како моменталните работни сетови се однесуваат во споредба со минимумот и максимумот и некои други параметри.

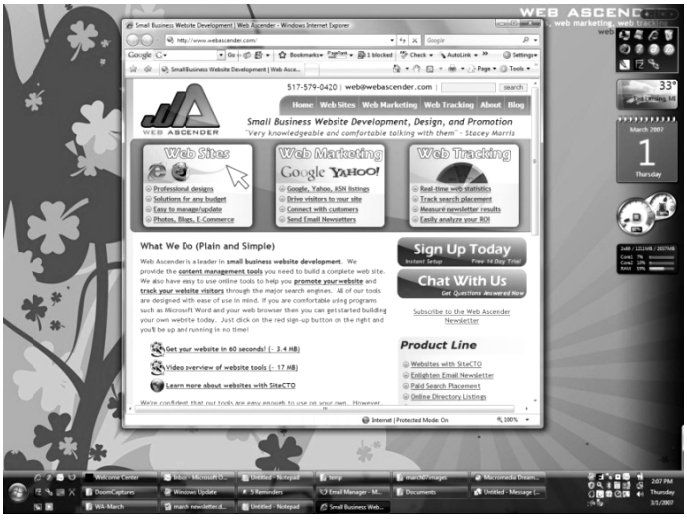
Битно е да се спомене дека за потребите на замена на страници целиот оперативен систем се смета како процес. Поседува страници и работен сет.

12.5. Windows Vista

Со објавувањето на Windows XP, Microsoft се соочи со еден сериозен проблем. Тоа е замената на оперативен систем кој што во себе ја содржи поддршката за сиот можен нов хардвер, одличен кориснички интерфејс и огромна база на корисници кои што навикнале на него. Секако дека одржувањето на вака сложен систем и објавувањето на закрпите за него бараат ангажман на одредени ресурси, но на стара слава не се живее долго.

Ако се навратиме наназад, Windows 95 со себе донесе поддршка за 32-битни апликации, PnP (во пракса познат како *Plag and Pray*), подобрено мултипрограмирање, унапредено управување со датотеки (крај на ограничувањето за името во облик 8+3 карактери). Подоцнежните верзии на Windows донесоа поддршка за USB и AGP како и PnP за надворешни уреди. Од другата страна се развиваше Windows NT со потполно ново јадро и уште од старт наменет за работа во мрежа. Тоа јадро послужи како основа за Windows 2000 кој што беше замислен да ги обедини двата развојни правци на NT и 9X во една целина. Со појавата на XP (предвиден да биде наследник на 2000) добивме NT јадро како и вистинска стабилност на оперативниот систем, на кој што сè работи добро како што и изгледа, така да почна да го снемува „синиот екран“ кој се појавуваше и при помали грешки.

Windows Vista е работен од нула. Значи не го дели заедничкото потекло со ниту еден свој претходник. Градење на нов оперативен систем од нула не е лесна работа и одлуката да започне таков проект не била лесна, но сигурно е дека била неизбежна. Ова станува појасно ако се земе фактот дека Microsoft веќе десетина години најавува дека ќе изврши надградба на некои технологии, посебно на системот на датотеки систем, на начин кој што ќе овозможи комотна и нормална работа и во следните десетина години.



Слика 12.8: Изглед на новиот кориснички интерфејс на Windows Vista

Во Редмонд одлучиле дека најдобро ќе ги привлечат корисниците со уште подотераниот кориснички интерфејс како и со зголемена безбедност во работната околина. Ова е особено значајно, земајќи во предвид дека луѓето во се поголем дел од работниот ден се приклучени на Интернет, со сите предности и сигурносни ризици кои таквата работа ги носи со себе. Тука е и поддршката за 64-битни апликации, која е подеднакво добро имплементирана како и поддршката за 32-битни апликации. На Слика 12.8 е прикажан корисничкиот изглед на Windows Vista и изгледот на новата верзија на Internet Explorer, 7.

Имено, иако тоа не беше толку очигледно, постојат големи надежи за 64-битните процесори кои од серверските платформи полека ќе си го најдат патот до корисничките работни маси, а дистрибуциите на Vista доаѓаат во двете верзии: 32 и 64 битна, со еднаква поддршка во вид на драјвери за севкупниот хардвер. За сите оние кои што купиле 64-битен процесор независно дали е AMD или Intel, Microsoft се потрудил секој да може да ги искористи предностите на оваа технологија веднаш по инсталирањето на новиот ОС. Уште со појавувањето на бета верзијата на Vista, од веб страните на независните производувачи можно е да се симнат најновите драјвери за Vista во 32 и 64 битна верзија.

Зошто е 64-битната технологија толку значајна и кои се нејзините предности? Теоретски, се работи за нешто побрзо извршување на апликациите, но само на оние кои се специјално напишани и прилагодени за 64 бита. Она што е поважно е поддршката за поголема количина на RAM, но и ова ќе зависи од произведувачите на матичните плочи и на чипсетовите, што беше случај и до сега. Теоретското ограничување од 4 GB RAM по конфигурација сега е поместено на 4 GB RAM по апликација (32-битниот ОС дозволува до 2 GB по апликација), што значи дека оваа година најмногу ќе се продава меморија.

Vista е поубава и подобро опремена од било која досегашна верзија на Windows. Во верзијата Ultimate доаѓа Media Centar – апликација за слушање на музика и гледање филмови, слушање радио, прегледување на слики и фотографии. Тука е и DVD Maker – програма за креирање на DVD филмови, мошне едноставен и за најголемиот број на корисници сосема доволен. Треба да се спомнат и подобрените верзии на Media Player и Internet Explorer, Movie Maker, Windows Slidebar со своите „gadgeti“, кои можат да се прилагодат според желбите на корисникот или да се симнат некои други, па дури и да се локализираат според потребите (временска прогноза, состојба на берза и т.н.).

Како најголема новина се спомнува безбедноста на оперативниот систем. Бидејќи е пишуван од нула, би требало да биде „имун“ на најголемиот број на вируси кои ги напаѓаа сите досегашни верзии на Windows.

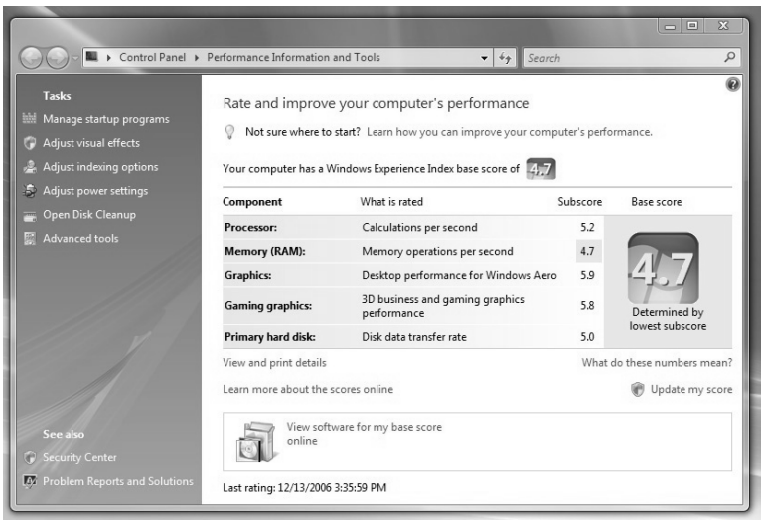
Една од главните карактеристики на Windows Vista е тоа што овој оперативен систем бара многу повеќе ресурси за разлика од неговите претходници Windows XP и Windows 2000. Поради многуте процеси (како и сервиси) кои се

активни, препорачлив е процесор со повеќе јадра, но и поголемо количество на RAM меморија. За да може да се активираат "интересните" графички додатоци потребна е и понова графичка картичка која подржува DirectX 9. Во следнава табела се прикажани минималните и проширените карактеристики на конфигурација за работа на Windows Vista.

Табела 12.2 Минимална и проширена конфигурација за Windows Vista

	Минимални	**Проширени**
Процесор	800 MHz	2 GHz (двојадрен)
RAM	512 MB	2 GB
Графичка картичка	32 MB RAM и DX 9	128 MB RAM, DX 9 и PS 2.0
Диск	20 GB	40 GB

Во Windows Vista постои и можност за оценување на хардверот на компјутерот, па така оценката може да се движи од 1 до 5.9 која граница ќе се поместува со создавањето на побрз и помоќен хардвер. Оценката главно е креирана врз основа на процесорот, графиката, меморијата и дискот. На Слика 12.9 е прикажана оценка 5.2 за компјутер со процесор Intel Core2 2.33GHz, со 3 GB RAM.



Слика 12.9: Оценка добиена од Windows Vista тестот

13. Прилог: вежби

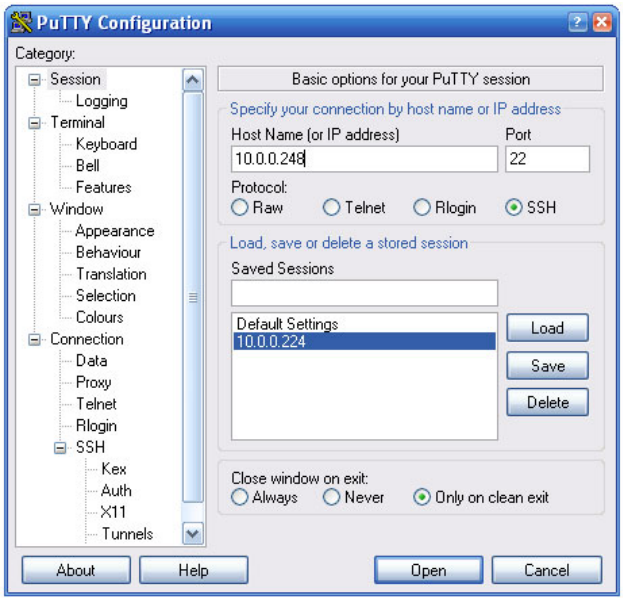
1. UNIX
2. Процеси
3. Распределба на процесорско време
4. Синхронизација на процеси
5. Застој
6. Виртуелна меморија
7. Bourne Again Shell скрипти
8. RAID

13.1. Вежба 1: UNIX

Во првата вежба ќе бидат објаснети основните команди за работа на оперативниот систем UNIX. Бидејќи оперативниот систем Linux е креиран по моделот на UNIX, командите кои ќе се извршуваат се UNIX команди но ќе бидат извршени во Linux.

13.1.1. Најавување

На серверот со локална IP адреса 10.0.0.248 е инсталиран Linux Mandriva 2007 во кој се креирани корисници *studentXX* каде *XX* е бројот на компјутерот од кој се најавува корисникот. Најавувањето ќе се изведе преку Windows XP со програмата PuTTY. Се кликнува *Start* > *Programs* > *PuTTY* и ќе се отвори прозорецот од Слика 13.1.



Слика 13.1: Најавување преку PuTTY

Се внесува IP адресата 10.0.0.248, се избира SSH протоколот преку кој ќе се одвива конекцијата (порта 22) и се избира *Open*. По ова ќе се отвори конзолата каде треба корисникот да се најави со корисничко име *studentXX* и лозинка *studentXX*. Откако се најавил, може да се продолжи со работа под Linux. Одјавувањето од серверот се врши со командата *logout* или *exit*.

Напомена: преку PuTTY не можат да се извршуваат графички (X Window) апликации.

13.1.2. Команди

Оперативниот систем содржи и кориснички посредник преку кој корисникот комуницира со него. Кај UNIX тоа е команден интерпретер, а комуникацијата се одвива преку командна линија. Командите кај UNIX обично претставуваат извршни програми кои интерпретерот ги пронаоѓа и ги извршува. Тие ја имаат следната структура:

```
$ команда опции аргументи
```

Кај UNIX се прави разлика помеѓу мали и големи букви. Затоа, командите мора да се задаваат точно онака како што се опишани. “опции” се од облик *-x*, каде *x* е некој карактер. Помеѓу '-' и '*x*' не смее да постои празно место.

На пример, кај:

```
$ ls –a /home
```

ls претставува команда за листање на содржина на директориум, *-a* е опција со која се бара листање на сите датотеки (*all*), а */home* е аргумент со кој се одредува директориумот кој ќе се листа. Една команда може да има повеќе опции кои го одредуваат начинот на кој ќе се изврши командата. Тие може да се задаваат посебно и да се комбинираат со еден знак '-'. Така:

```
$ ls -l -a /home
```

е еквивалентно со :

```
$ ls -la /home
```

Командата:

```
$ ls -R /home
```

ја листа содржината од секој подиректориум – рекурзивно. Некои опции се состојат од збор или зборови наместо од букви и имаат две црти наместо една, на пример:

```
$ ls -l --full /home
```

Некои опции можат да имаат и вредност:

```
$ ls -l --sort=size
```

13.1.3. Прирачник за команди

Синтаксата на командите е зададена прецизно и за секоја команда таа може да се види преку *man* командата. Оваа команда се користи на следниот начин:

```
$ man команда
```

Со ова се врши прикажување на страници од прирачникот за дадената команда.

```
$ man ls
```

Може да се користат тастерите *Up* и *Down* како и *PgUp* и *PgDn*. Од прирачникот се излегува со *q*. Постојат неколку правила за толкување на синтаксата на командите:

- опциите и аргументите кои се во средни загради [ ] се опциони
- сè што не е во средни загради мора да се внесе
- името на командата и опциите мора да се испишат точно онака како што е дефинирано со синтаксата
- аргументите се заменуваат со она на што се однесува командата, на пример име на датотека или директориум
- '...' значи дека е можен произволен број на повторувања на претходниот аргумент.

Помошна информација за самиот прирачник се добива со:

```
$ man man
```

За да се видат и опционите знаменца се користи:

```
$ man -ls
```

13.1.4. Корисници

За UNIX системите карактеристично е тоа што постои еден главен корисник *root* кој ги има сите привилегии над системот и повеќе корисници. Препорачливо е да не се работи како *root* корисник, бидејќи е можно извршување на несакани дејствија кои би го оштетиле системот, па затоа

администраторот се најавува како *root* само кога има потреба за тоа. Постојат повеќе начини да се види кој е во моментот најавен на системот, па дури и што прави.

```
$ who

gorgik   :0           Sep 18 08:08
gorgik   pts/0        Sep 18 08:08
gorgik   pts/1        Sep 18 08:09
student0 pts/2        Sep 18 12:35 (gebel.home)
```

Командата *su* (*Set User*) служи за промена на активниот корисник. Ако оваа команда е без аргументи се врши најавување како *root* корисник (и ако се знае *root* лозинката). Командата *useradd* додава нов корисник во системот (може да се повика само од *root* корисникот):

```
$ useradd test
```

Се поставува нова лозинка за *test* корисникот:

```
$ passwd test

Changing password for user test
New UNIX password:
BAD PASSWORD: it is too short
Retype new UNIX password:
passwd: all authentication tokens updated successfully
```

Доколку од *root* се врши пренајавување како било кој друг корисник, не е потребна лозинка. Ако обичен корисник се пренајавува во *root* или едноставно се менува корисникот, лозинката е неминовна. Корисникот се пренајавува како *test*.

```
$ su test
```

Се преминува во сопствениот (*home*) директориумот на корисникот, т.е. во */home/test* со користење на командата:

```
$ cd
```

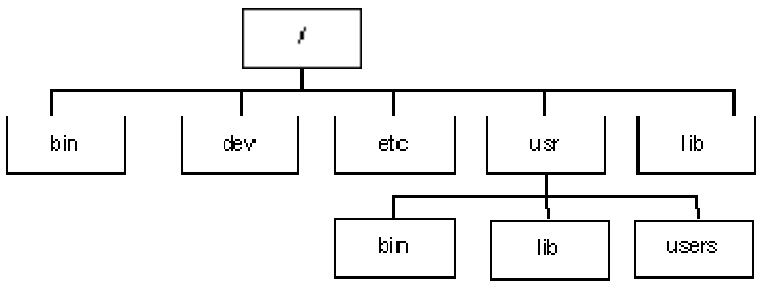
Сега активниот корисник се враќа како *root*.

```
$ exit
```

13.1.5. Управување со директориуми и датотеки

Овде ќе бидат споменати само основните операции за менаџирањето на директоруми и датотеки.

Кај UNIX постои хиерархиска структура од директориуми во која се организирани датотеките. На врвот од структурата на директориумите се наоѓа еден посебен директориум наречен коренски директориум (*root*). Една вообичаена организација е прикажана на Слика 13.2:



Слика 13.2: Структура на UNIX директориуми

Во UNIX сè е организирано во датотека (од датотеки, директориуми сè до звучната картичка и DVD режачот). Директориумот е всушност датотека во која се чува листа од други датотеки. Листата не може да содржи други податоци. Секој корисник на системот има свој директориум (*home directory*) кој се наоѓа во */home*. Така, на корисникот *student01* му припаѓа директориумот */home/student01*. По најавување на системот, тој станува тековен (моментален). Командата со која се прикажува кој е тековниот директориум е:

```
$ pwd
```

Ќе биде прикажано: */home/student01*. Датотеките кои ги содржи тековниот директориум можат да се излистаат со командата:

```
$ ls
```

Ако директориумот е празен, *ls* нема да даде никаков одговор. *ls* може да се зададе со опции кои ќе го променат начинот на кој ќе биде прикажана листата од датотеки. *ls -l* генерира на листинг со детални информации. На пример со:

```
$ ls -l

total 12
-rw-rw-r--  1 student01 student01    3 сеп 18 12:23 pismo.txt
-rw-rw-r--  1 student01 student01    4 сеп 18 12:23 programa.c
drwx------  2 student01 student01 4096 сеп 18 09:48 tmp/
```

се добиваат и информации за датотеките. *-rw-rw-r--* е низа која покажува кој какви права има над датотеката. Доколку започнува со -, станува збор за датотека, а ако започнува со *d* за директориум. Излистани се 2 датотеки и еден директориум. Нивен сопственик е *student01*. Следат податоци за големината на датотеката, датумот и времето на последна измена и на крај нејзиното име.

Прикажана е листа на директориуми кои обично се наоѓаат во Linux и UNIX системите и соодветниот Windows еквивалент:

/

Ова е коренски директориум каде што се наоѓаат сите другите директориуми.

/bin

Означува "*binary*" и ги содржи извршните датотеки - програми. Тука се наоѓаат наредбите како што е "*ls*". Во Windows, фолдерот *c:\windows\commands* содржи некои од командните програми а другите се расфрлани насекаде.

/dev

Означува "*devices*". Содржи одреден број на специјални псевдо - датотеки кои се користат за пристап кон физичкиот хардвер. На пример паралелниот порт е датотека наречена "*lp0*", хард дискот "*hda*", а неговата прва партиција би била "*hda0*". Windows користи сличен метод, уредите имаат имиња LPT1, COM1 или CON.

/etc

Тука се сместени конфигурациски информации во текст датотеки и може да се погледнат.

/home

Тука се наоѓаат директориумите на корисниците означени со нивните имиња. Приближен еквивалент во Windows е *c:\windows\profiles*, каде што можат да се најдат детали за корисниците.

/lib

Во овој директориум се наоѓаат библиотеките – со функции и процедури кои можат да бидат повикувани од програмите. Во Windows библиотеките се чуваат во *c:\windows\system32*.

/mnt или **/media**

Тука се монтираат мемориските уреди, а обично ги содржи поддиректориумите "*cdrom*", "*floppy*" и т.н. каде се наоѓа содржината на соодветниот уред.

/opt

Тука се инсталираат опционалните компоненти на системот. Приближно во Windows би одговарал *c:\Program Files*.

/tmp

Привремен директориум. Сите датотеки се бришат автоматски. Во Windows е *c:\windows\temp*.

/usr

Содржи копија од повеќето директориуми во *root*. На пр. постои "*bin*" директориум кој содржи програми, "*lib*" со библиотеки итн. Обично, "*core*" Linux датотеки се содржани во *root* директориумите, додека "*non-core*" датотеки во "/*usr*" поддиректориуми.

/var

Означува "*various*". Содржи системски лог датотеки, *spool* и други податочни датотеки.

Со командата *cd* се врши промена на директориум. На пример, со:

```
$ cd ..
```

се преминува еден директориум назад, а со:

```
$ cd student01
```

се преминува во директориумот *student01* кој е на исто ниво со тековниот.

Директно во сопствениот директориум се преминува со:

```
$ cd
```

Името на сопствениот директориум може да се замени со употреба на карактерот ~. Така, ~ е сопствениот директориум на корисникот кој е најавен.

Директориум се креира со:

```
$ mkdir директориум
```

На пример, со:

```
$ mkdir primeri
```

се креира нов директориум под име *primeri* во тековниот директориум, а со:

```
$ mkdir ../nov
```

се креира директориум *nov* на исто ниво како и тековниот. За копирање на датотеки и директориуми се користи командата *cp* со општа синтакса:

```
$ cp [-опција] извор цел
```

Ако постои датотека која има исто име како и целната датотека, тогаш таа ќе биде пребришана. За да се избегне тоа, може да се користи *-i* опција која во тој случај го оневозможува копирањето. Датотеки можат да се копираат во ист или различни директориуми при што имињата на датотеките може да се прошируваат со употреба на мета-знаци. Со командата:

```
$ rm датотека
```

се врши бришење на датотека ако корисникот има соодветна привилегија над таа датотека. Еднаш избришана датотека не може да се врати назад, затоа употреба на *-i* опција и во овој случај е препорачлива. Директориум се брише со командата:

```
$ rmdir директориум
```

Со *rmdir* не може да се избрише директориум кој не е празен. Тоа може да се постигне со *rm –r* директориум, но е многу деструктивна операција бидејќи ќе го избрише *директориум* и сите негови датотеки и поддиректориуми. Така на пример, со

```
$ rmdir temp
```

ќе се избрише празниот директориум *temp*, а за да се отстрани директориумот *bak* и сите датотеки и директориуми што ги содржи, треба да се зададе:

```
$ rm -r bak
```

Со командата *mv* може да се изврши преместување на директориуми и датотеки. Општо, таа може да се зададе како:

```
$ mv [option] датотека1 датотека2
```

а може да се употреби и за преименување на датотеки и директориуми. Ако новата локација на директориум или датотека претставува директориум или датотека кои веќе постојат, тогаш ќе бидат пребришани без предупредување. Слично како и *rm* и *mv* може да се зададе со *-i* опција за преместување со потврда. Сепак, таа опција не мора да постои кај сите верзии на UNIX. На пример:

```
$ mv primer.c c_prog/lekcija1.c
```

е пример за преместување на датотека од еден директориум во друг, а со:

```
$ mv spisoci ../stari/listi
```

директориумот *spisoci* се преместува во погорен директориум *stari* и се преименува во *listi*.

13.1.6. Работа со датотеки

За прикажување на датотеки кои се состојат од само неколку линии наједноставно е да се користи командата:

```
$ cat датотека
```

но кога се работи за поголеми текстови, тогаш е попогодно да се користи алатка која има можност за листање на текстот екран по екран. Такви се командите *more* и *less*, при што *less* има повеќе можности од *more*. За прикажување на првите неколку редови од *датотека* се користи командата *head*. Зададена без опции:

```
$ head датотека
```

таа ќе ги прикаже првите 10 реда од *датотека*. Со

```
$ head -n 5 датотека
```

ќе се прикажат првите 5 реда од датотека. Слична е и командата *tail* која ги прикажува редовите од датотеката започнувајќи од нејзиниот крај.

Покрај тоа што датотеки можат да се креираат со други програми или со уредувачи на текст, можно е креирање на датотеки со користење на UNIX команди и пренасочување на стандардниот излез.

Се пишува командата:

```
$ cat > nova
```

а по <*Enter*> се испишува текстот кој корисникот сака да го смести во датотека. По завршувањето, со *Ctrl+D* се прекинува *cat* командата, а текстот ќе биде зачуван во датотеката со име *nova*. Вака може да се случи да се избрише датотеката *nova* ако таа веќе постои.

Командата:

```
$ touch датотека
```

креира празна датотека. На пример, за да се креира *primer.txt* датотека се користи:

```
$ touch primer.txt
```

Со командата:

```
$ file датотека
```

се одредува типот на датотеката според нејзината содржина со голема точност.

Основни команди за споредување на датотеките: командата *cmp* проверува дали две датотеки се исти, а командата *diff* како резултат ја дава разликата помеѓу две датотеки во вид на листа од линии кои треба да се (a) додадат, (d) избришат или (c) променат за двете датотеки да бидат еднакви. Ако командата е зададена како:

```
$ diff датотека1 датотека2
```

тогаш редовите од датотека1 се претставени со <, а од датотека2 со >.

```
$ diff -iw датотека1 датотека2
```

значи споредување на датотека1 и датотека2 третирајќи ги како исти малите и големите букви и сметајќи дека повеќе празни места се исто што и едно.

Со командата *find* се врши пронаоѓање на датотеки во еден систем на датотеки. Нејзината општа синтакса е:

```
$ find патека -name датотека -print
```

патека го одредува директориумот од кој е потребно да се започне пребарувањето, при што се пребарува рекурзивно во неговите поддиректориуми. Опцијата *-print* се задава за да се изврши прикажување на резултатот. Ако името на датотеката се задава со мета-знаци, тогаш мора да биде заградено со апострофи. Во случај да не биде најдена датотеката што се бара, не се испишува никаков резултат.

На пример, со:

```
$ find . -name jan92 -print
```

се бара датотеката jan92 почнувајќи од тековниот директориум, а со:

```
$ find ~/ -name '*.c' -print
```

се бараат сите датотеки со наставка .c во сопствениот директориум.

За да се пронајде директориум *gnu* во */usr/local* треба да се зададе:

```
$ find /usr/local -name gnu -type d -print
```

Командата:

```
$ locate датотека (или директориум)
```

ја врши истата работа, со тоа што прво треба да се создаде база на податоци за пребарување со *updatedb* командата:

```
$ updatedb
$ locate finger

/usr/bin/finger
/usr/bin/kfinger
/usr/sbin/safe_finger
```

Таа врши пребарување за датотека или директориум на целиот хард диск и ги прикажува локациите каде се наоѓаат. Базата на податоци може да ја креира само *root* корисникот.

13.1.7. Уредување на датотеки

Стандарден (и најкористен) едитор кој е вклучен во секоја верзија на UNIX и Linux е *vi* едиторот т.е. неговата усовршена верзија *vim* која распознава и форматира голем број на текстуални датотеки и скрипти. Се активира со командата *vi* како на Слика 13.3.

```
$ vi датотека
```

```
<html>
<head>
   <title>NetBeans IDE - README</title>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
   <link rel="stylesheet" type="text/css" href="netbeans.css">
</head>

<body>
<h1>README</h1>
<h2>NetBeans<sup class="tmtext">TM</sup> IDE 5.0</h2>
<p>NetBeans IDE is an integrated development environment (IDE) for writing,
compiling, testing, and debugging desktop applications and web applications for
the Java<sup>TM</sup> platform.
NetBeans IDE includes a full-featured text editor with syntax highlighting and
error checking, visual design tools, Ant support, version control system
support, and many other features.</p>

<h3>To start the IDE (Microsoft Windows) use one of these methods:</h3>
<ul>
  <li>Double-click the NetBeans 5.0 icon on your desktop.</li>
  <li>Choose Start &gt; All Programs &gt; NetBeans 5.0 &gt; NetBeans IDE.</li>
```

Слика 13.3: Vi едитор

Отако ќе се отвори едиторот треба да се премине во режим за уредување. Тоа се прави со *I* од тастатурата а се излегува од овој режим со *Esc*. Додека корисникот е во режимот *insert*, во долниот лев агол пишува -- *INSERT* -- и тогаш се уредува текстот. Се излегува од овој режим (со *Esc*) и се пишува *:wq* за снимање и излегување (*write & quit*) или *:q!* за излегување без снимање на датотеката.

13.1.8. Простор на диск

Командата *du* ја прикажува искористеноста на просторот на дискот на тековниот директориум за секоја датотека и сите поддиректориуми изразено во килобајти и завршува со вкупниот број.

```
$ du
```

-s опцијата служи за приказ на сумираниот резултат,

```
$ du -s
```

додека *-k* ја прикажува големината во килобајти (KB).

```
$ du -s -k
```

13.1.9. Пренасочување на стандардниот влез и стандардниот излез

Повеќето команди се имплементирани така што како влез го користат влезниот поток (стандарден влез), а резултатот од нивната работа го прикажуваат на излезниот поток (стандарден излез). Овие низи од знаци се нарекуваат потоци бидејќи немаат никаква повисока внатрешна структура. Обично, стандардниот влез се презема од тастатурата, а стандардниот излез е мониторот. Може да се врши пренасочување на овие потоци со помош на операторите > и <. На пример, со:

```
$ cat pismo > poraka
```

излезот од командата *cat* ќе се пренасочи од мониторот кон датотеката *poraka*. По извршувањето на оваа команда, ако датотеката *poraka* не постои, ќе се креира и ќе биде копија на *pismo*. Ако *poraka* постои, нејзината претходна содржина ќе се прекрие без предупредување.

Операторите < и > може да се комбинираат заедно во една команда. Кај:

```
$ cat < pismo > poraka
```

pismo ќе биде влезот на командата *cat*, а потоа излезот ќе се пренасочи во *poraka*. Пренасочувањето на стандардниот влез и излез е една од поважните особини на UNIX. Бидејќи датотеките се организирани како низи од бајти, без внатрешна структура, многу од вообичаените задачи со датотеки (сортирање, филтрирање, конкатенирање и т.н.) можат да се извршуваат само со комбинирање на UNIX команди (кај други системи најчесто би било потребно креирање на посебни апликации).

Уште еден оператор за пренасочување на стандардниот излез е >> (не постои и <<). Ако излез од команда се пренасочи кон датотека со >>, тогаш резултатот од извршување на командата ќе се надоврзе на крајот од датотеката. Со помош на >> може да се имплементира конкатенирање на датотеки.

```
$ cat pismo >> poraka
```

ќе ја додаде *pismo* на крајот од *poraka*. Истиот ефект ќе го има и следната низа од команди:

```
$ cat poraka pismo > zaedno
$ cat zaedno > poraka
$ rm zaedno
```

13.1.10. Цевки

Операторот |, кој се нарекува цевка (*pipe*), се користи за поврзувањето на командите. Притоа, излезот од првата команда се користи како влез на втората. Многу честа комбинација е следната:

```
$ ls -al | more
```

Имено, ако во директориумот има многу датотеки, излезот од *ls* “ќе избега”. Командата *more* ќе го преземе и ќе го прикаже страна по страна. Со | може да се поврзуваат произволен број на команди. Таква командна линија се нарекува цевковод (*pipeline*). На пример со:

```
$ ls -al test | sort | cat > lista
```

листата на датотеки во директориумот ќе се сортира и ќе се пренасочи во датотеката *lista*.

13.2. Вежба 2: Процеси во UNIX

UNIX е повеќепроцесен оперативен систем. Тоа значи дека во исто време може да се извршуваат повеќе програми. Секоја програма која се извршува се нарекува процес. Всушност, во секој момент компјутерот може да извршува низа од процеси. Секогаш кога корисник ќе се најави на системот се активира процес сврзан со неговата сесија. Секоја команда што се задава исто така може да иницира нов процес. Покрај нив, за цело време додека работи системот се активни голем број системски процеси.

13.2.1. Добивање на информации за процесите

Командата *ps* (*process status*) прикажува на екранот листа на активни процеси т.е. PID (*procecs identificator*) и командата со која процесот е иницран.

```
$ ps

 PID TTY          TIME CMD
4340 pts/1    00:00:00 bash
4399 pts/1    00:00:00 ps
```

Секој процес добива свој идентификационен број PID кога се активира. Тој број може да се движи од 0 до некоја максимална вредност. Кога таа ќе се достигне, повторно се започнува од 0, со тоа што ако постои процес со некој број, се зема следниот слободен.

Оваа команда има три основни аргументи и тоа: *-e* (*every process*), *-f* (*full listing*) и *–u* (*user*). Командата:

```
$ ps -e
```

ги прикажува сите процеси на системот, командата:

```
$ ps -f
```

прикажува додатни информации за процесите, а

```
$ ps –u UID
```

ги прикажува сите процеси кои ги има иницирано корисник со UID внесено како параметар. Многу користен параметар кој е додаден во Linux системите е *--forest* кој прикажува дрво на процеси.

```
$ ps --forest

 PID TTY          TIME CMD
4340 pts/1    00:00:00 bash
4472 pts/1    00:00:00  \_ ps
```

што значи дека процесот *ps* (командата) е дете процес на конзолата (*bash – Bourne Again SHell*).

```
$ ps –e --forest
```

Командата може да се користи и во комбинација со други наредби:

```
$ ps –e | grep kdesktop
```

за прикажување на сите процеси кои содржат *kdesktop* или

```
$ ps –ef –forest | less
```

за прикажување во *less* на стеблото на процеси. Аргументот *-t* ги прикажува сите процеси кои се во врска со зададен терминал. Аргумент *-l* се задава ако е потребно да се генерира комплетна листа на сите атрибути на процесите. Во следниот пример е дадена листа на сите процеси кои се во врска со терминалот *pts/1*.

```
$ ps –l –t pts/1

F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   500  4241  4240  0  75   0 -  1001 wait   pts/1    00:00:00 bash
0 R   500  4350  4241  0  75   0 -   647 -      pts/1    00:00:00 ps
```

Вредноста во листата која треба да се забележи е бројот PPID. Тоа е PID на процесот - родител. Во горниот пример се гледа дека процесот *bash* е родител на процесот *ps*. На тој начин може да се следи хиерархијата на генерирање на сите активни процеси.

13.2.2. Праќање на команда kill на процесите

Зависно од имплементацијата, во секој UNIX систем дефинирани се околу 40 сигнали кои може да се пратат на процесите. Процесот (програмата) може да има механизам за справување со сигналот кој ќе го прими што е задача на

програмерот. За прикажување на листа на сите сигнали се извршува командата:

```
$ kill –l

 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1
36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4  39) SIGRTMIN+5
40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8  43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1
64) SIGRTMAX
```

Во случај кога некој процес не може да заврши нормално или е во бесконечна јамка, потребно е да се уништи со команда *kill*. Таа го има следниот општ облик:

$ kill [-signal] *PID-идентификатор*

Најсигурно е да се “убие” процес со:

$ kill *PID-идентификатор*

што има ист ефект како и

$ kill -15 *PID-идентификатор*

Ако тоа не успее, треба да се обиде со сигналот 1 (HUP):

$ kill -1 *PID-идентификатор*

со што процесот се прекинува како при одјавување од системот.

$ kill -9 *PID-идентификатор*

ќе го уништи процесот сигурно, но се користи како последно средство бидејќи на тој начин може да останат активни процесите кои тој ги иницирал.

Напомена: не може да се уништат процеси кои им припаѓаат на други корисници.

Позначајни сигнали се:

- TERM (15), уништување на процесот – процесот може да го игнорира овој сигнал,
- KILL (9), неотповикливо уништување на процесот – процесот не може да го игнорира овој сигнал,
- HUP (1), уништување на процесот кој се извршува во позадина доколку корисникот се одјави,
- INT (2), интерактивен сигнал на прекин (*interrupt*), исто со *Ctrl+C,*
- QUIT (3), интерактивно уништување на процесот, исто со *Ctrl+D.*

Од наведените сигнали процесите може да ги игнорираат сите освен KILL сигналот.

Сигнали за управување со процесите се:

- STOP (), стопирање на процесот – процесот не може да го игнорира овој сигнал,
- CONT (), продолжување со извршувањето на процесот – процесот не може да го игнорира овој сигнал

и други сигнали со помала важност.

Постои и команда *killall* која праќа сигнал на сите процеси. Доколку корисникот е најавен како *root* и ја изврши оваа команда компјутерот ќе престане со работа.

13.2.3. Извршување на процес во преден и заднински план

Бидејќи UNIX е повеќепроцесен систем, овозможено е извршување на повеќе процеси во исто време. Комуникацијата помеѓу конзолата и корисникот најчесто се одвива со извршување на еден процес во моментот. На пример ако се изврши командата *sort golemadatoteka*, корисникот треба да чека нејзино извршување за да почне да работи со друг процес. За таа цел, процесот кој нема потреба од комуникација со корисникот може да се изврши во позадина а корисникот да извршува други процеси во исто време.

Ќе се искористи командата *sleep* која чека да поминат одреден број на секунди.

```
$ sleep 20
```

чека да изминат 20 секунди. За тоа време корисникот не може да работи ништо. За преместување на процесот во позадина, на крајот од командата го додаваме знакот &:

```
$ sleep 20 &
[1] 4376
```

По што ќе се врати бројот на процесот кој се извршува во позадина (1) и неговиот PID (4376). Активен процес може да се стопира со *Ctrl+Z* по што процесот ќе отиде во позадина и ќе му се додели број. За да продолжи со работа процесот во позадина се користи:

```
$ bg %1
[1]+ sleep 20 &
$ jobs
[1]+ Running sleep 20 &
```

jobs командата се користи за да се видат процесите кои работат во позадина. За да некој процес се префрли од позадина во преден план се користи командата *fg*:

```
$ fg %1
```

13.2.4. Приоритет на процесите

Сите процеси кои се извршуваат, добиваат одредено време кога располагаат со процесорот кое време е одредено од јадрото. Јадрото користи претходно знаење - евристика (множество на правила) за да одреди колку време на секој процес да алоцира. Многу процеси чекаат на интеракција со корисникот или на некој уред па во ова време тие не го искористуваат процесорот. Во UNIX секој процес има одредена вредност на приоритет од +20 до -20 каде пониската вредност има поголем приоритет во користењето на процесот. Постои можност за промена на оваа вредност со командата *renice*.

```
$ renice +<priority> <pid>
$ renice -<priority> <pid>
```

при што само *root* корисникот може да доделува негативни вредности за приоритетот на процесите. Пример:

```
$ sleep 50 &

[1] 4603

$ renice +17 4603

4603: old priority 0, new priority 17

$ top -p 4603
```

```
top - 13:04:32 up  2:41,  3 users,  load average: 0.00, 0.00, 0.00
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0% us,  0.3% sy,  0.0% ni, 99.7% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:    255748k total,   233700k used,    22048k free,    15856k buffers
Swap:   971892k total,        0k used,   971892k free,   125588k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4603 gorgik    37  17  2936  700  600 S  0.0  0.3   0:00.00 sleep
```

top командата ги сортира процесите според меморијата и процесорското време кое го користат. Аргументот *–p* служи за специфицирање на одреден PID на процес.

13.3. Вежба 3: Распределување на процеси

Бидејќи повеќе процеси може да бидат активни во исто време а само еден да се извршува (во случај на еден процесор), оперативниот систем треба да има механизам за доделување на процесорското време на активните процеси. За распределба на процесите постојат повеќе различни алгоритами кои имаат различни особини и се погодни за одредена класа на процеси во однос на друга. Помеѓу главните критериуми за развивање на ваков алгоритам се:

- максимизирање на искористеноста на процесорот
- максимизирање на пропусната моќ (извршени процеси во единица време)

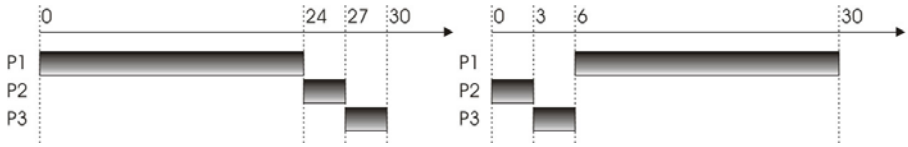
13.3.1. First Come, First Served (FCFS)

FCFS (прв дојден, прв услужен) е најпрост алгоритам за распределба на процесите, процесите го добиваат процесорот по оној редослед по кој пристигнуваат во редот на чекање.

Задача 1: Процесите пристигнуваат по редоследот P1, P2 и P3 а нивното време на извршување е 24, 3 и 3 ms. Да се нацрта Гантовиот дијаграм за алгоритамот FCFS и да се пресмета средното време на чекање.

Задача 2: Процесите од Задача 1 пристигнуваат по следниот редослед P2, P3 и P1. Да се нацрта Гантограмот за алгоритамот FCFS и да се пресмета средното време на чекање.

Решение:



Слика 13.4: Гантови дијаграми за FCFS

$t_w=(0+24+27)/3=17ms$ $t_w=(6+0+3)/3=3ms$

Кај овој алгоритам може да се јави ефект на конвој, така што повеќе мали процеси да чекаат извршување на некој поголем процес. Алгоритамот не е добар за интерактивни процеси кои чекаат внес на корисникот или ослободување на некој ресурс.

13.3.2. Shortest Job First

Целта на овој алгоритам е да го процени времето на извршување на процесите и да го додели процесорот на најкраткиот процес. Проценувањето на процесорското време се сведува на проценка на траење на следниот циклус кој зависи од претходниот со релацијата:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

Каде τ е проценето време, а t е времето на извршување. Коефициентот α е помеѓу 0 и 1.

Задача 3: Да се пресмета τ (од 1 до7) ако $\alpha = 0.5$, $\tau_0=10$ а вредностите на t се дадени во табелата.

	0	1	2	3	4	5	6	7
t	6	4	6	4	13	13	13	
τ	10	**8**	**6**	**6**	**5**	**9**	**11**	**12**

Решение:

$\tau_1 = \alpha t_0 + (1-\alpha)\tau_0$; τ_1=0.5 * 6 + (1 - 0.5) * 10 = 8

$\tau_2 = \alpha t_1 + (1-\alpha)\tau_1$; τ_2=0.5 * 4 + (1 - 0.5) * 8 = 6

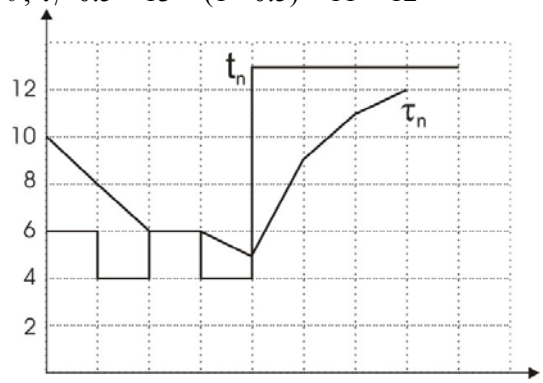
$\tau_3 = \alpha t_2 + (1-\alpha)\tau_2$; τ_3=0.5 * 6 + (1 - 0.5) * 6 = 6

$\tau_4 = \alpha t_3 + (1-\alpha)\tau_3$; τ_4=0.5 * 4 + (1 - 0.5) * 6 = 5

$\tau_5 = \alpha t_4 + (1-\alpha)\tau_4$; τ_5=0.5 * 13 + (1 - 0.5) * 5 = 9

$\tau_6 = \alpha t_5 + (1-\alpha)\tau_5$; τ_6=0.5 * 13 + (1 - 0.5) * 9 = 11

$\tau_7 = \alpha t_6 + (1-\alpha)\tau_6$; τ_7=0.5 * 13 + (1 - 0.5) * 11 = 12



Слика 13.5: Процена на траењето на следниот циклус

Постојат две варијанти на овој алгоритам, со испразнување (*preemptive*) и без испразнување (*non-preemptive*). Тие се однесуваат различно кога нов процес се појави во редот на чекање. Кај варијантата со испразнување, доколку се извршува процес а дојде нов процес кој е пократок од времето потребно претходниот да заврши, се прекинува тековниот и се извршува новиот процес.

Задача 4: Во табела е дадено пристигнување на процесите и време на нивно извршување. Да се распредели процесорското време според *FCFS*, *SJF* со и без испразнување. Да се пресметаат времињата на чекање на секој процес и средното време на чекање.

Процес	Време на активирање	Време на извршување
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Решение:

P1	P2	P3	P4
0 - 7	7 - 11	11 -12	12 - 16

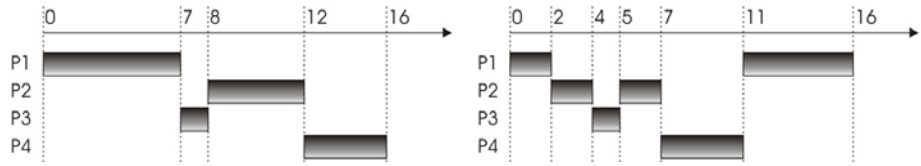
t_{w1}=0; t_{w2}=7-2=5; t_{w3}=11-4=7; t_{w4}=12-5=7; t_w=(0+5+7+7)/4=4.75;

P1	P3	P2	P4
0 - 7	7 - 8	8 -12	12 - 16

tw1=0; tw2=8-2=6; tw3=7-4=3; tw4=12-5=7; tw=(0+6+3+7)/4=4;

P1	P2	P3	P2	P4	P1
0 - 2	2 - 4	4 - 5	5 - 7	7 - 11	11 - 16

tw1=11-2=9; tw2=5-4=1; tw3=0; tw4=7-5=2; tw=(9+1+0+2)/4=3;



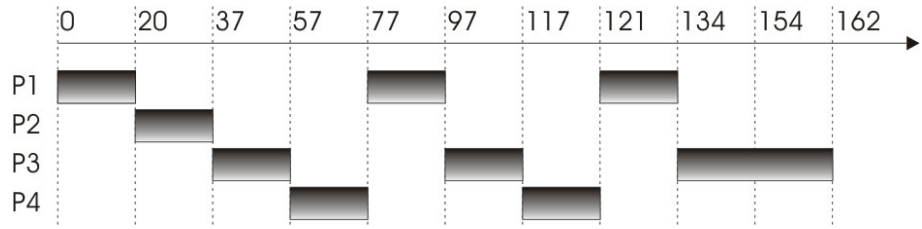
Слика 13.6: Гантови дијаграми са SJF

13.3.3. Round Robin

Овој алгоритам функционира на следниот начин: се дефинира одреден временски квантум (*time slice*) од 10 до 100 ms и кружен ред на чекање на процесорот. Системот по истекот на временскиот квантум го прекинува процесот кој се извршува и го доделува процесорот на следниот процес.

Задача 5: Да се распределат процесите P1, P2, P3 и P4 кои пристигнуваат во исто време со *Round Robin* алгоритамот ако временскиот квантум е 20 временски единици а времето на извршување на процесите е 53, 17, 68 и 24 временски единици.

P1	P2	P3	P4	P1	P3	P4	P1	P3	P3
0-20	20-37	37-57	57-77	77-97	97-117	117-121	121-134	134-154	154-162



Слика 13.7: Гантограм за Round Robin алгоритамот

Задача 6: Во табелата е дадено времето на извршување на процесите. Процесите доаѓаат во редослед P1, P2, P3, P4 и P5, сите приближно во момент t=0.

Процес	P1	P2	P3	P4	P5
Време на извршување	10	1	2	1	5

а) да се распределат процесите според *FCFS, SJF* без испразнување и *Round Robin* со временски квантум q=1.

б) да се одреди времето на комплетирање на процесите (*turnaround time*)

в) да се одреди времето на чекање на секој процес и средното време на чекање

Решение:

FCFS

P1	P2	P3	P4	P5
0-10	10-11	11-13	13-14	14-19

SJF

P2	P4	P3	P5	P1
0-1	1-2	2-4	4-9	9-19

RR

P1	P2	P3	P4	P5	P1	P3	P5	P1	P5	P1	P5	P1	P5	P1
0-1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10	10-11	11-12	12-13	13-14	14-19

време на комплетирање:

	FCFS	SJF	RR
P1	10	19	19
P2	11	1	2
P3	13	4	7
P4	14	2	4
P5	19	9	14

време на чекање:

	FCFS	SJF	RR
P1	0	9	9
P2	10	0	1
P3	11	2	5
P4	13	1	3
P5	14	4	9
средно време	9.6	3.2	5.4

Задача 7: Дадена е табела на 3 процеси со нивно време на пристигнување и време на извршување. Да се распределат процесите и пресметаат средното време на чекање ако распределбата е според:

а) FCFS

б) SJF

Процес	P1	P2	P3
Време на пристигнување	0	0.4	1
Време на извршување	8	4	1

Решение:

FCFS

P1	P2	P3
0-8	8-12	12-13

	в. комплетирање	в. чекање
P1	8	0
P2	10-0.4=11.6	8-0.4=7.6
P3	13-1=12	12-1=11
средно време	10.53	6.2

SJF

P1	P3	P2
0-8	8-9	9-13

	в. комплетирање	в. чекање
P1	8	0
P2	13-0.4=12.6	9-0.4=8.6
P3	9-1=8	8-1=7
средно време	9.53	5.2

13.4. Вежба 4: Синхронизација на процеси

Процесите ги користат ресурсите кои се на располагање на компјутерскиот систем. Поради можноста повеќе процеси да посакаат да го користат истиот ресурс, оперативниот систем мора да обезбеди механизам за синхронизација на процесите.

13.4.1. Проблемот на произведувач-потрошувач

```
item buffer[N];              // definicija na bafer so golemina N
int in = 0;                  // prvo slobodno mesto
int out = 0;          // prvo zafateno mesto
int counter = 0;             // broj na zafateni elementi

// Proizveduvac
item sleden_proizveden;
while(1) {
       // proizveduvacot sozdava proizvod
       while(counter == N);
       // baferot e poln, se ceka potrosuvacot da zeme od baferot
       buffer[in] = sleden_proizveden;
       in = (in + 1) % N;
       counter++;
}

// Potrosuvac
item sleden_potrosen;
while(1) {
       while(counter == 0);
       // baferot e prazen, se ceka proizvoditelot da proizvede
       sleden_potrosen = buffer[out];
       out = (out+1) % N;
       counter--;
}
```

Овој код би функционирал исправно само ако операциите counter++ и counter-- се извршуваат неделиво без прекин. Даден е кодот за овие операции на машински јазик (*asembler*), онака како се извршуваат на процесорот.

counter++	counter--
register1 = counter register1 = register1 + 1 counter = register1	register2 = counter register2 = register2 – 1 counter = register2

Се претпоставува дека вредноста на променливата *counter* е 5 и дека произведувачот и потрошувачот ги извршуваат наредбите *counter*++ и *counter*--. По извршувањето на овие две наредби вредноста на променливата *counter* треба да биде 5. Меѓутоа можно е следното сценарио:

Команда	Извршува	Инструкција	Вредност
T0	произведувач	register1=counter	register1=5
T1	произведувач	register1=register1+1	register1=6
T2	потрошувач	register2=counter	register2=5
T3	потрошувач	register2=register2-1	register2=4
T4	произведувач	counter=register1	counter=6
T5	потрошувач	counter=register2	counter=4

13.4.2. Критична секција

Критична секција е дел од кодот во кој процесот пристапува до заедничките ресурси и ги модификува (мемориски променливи, табели, датотеки...). Притоа треба да е задоволено оперативниот систем да може да дозволи само еден процес да е во својата критична секција (додека еден процес е во критичната секција, никој друг процес не смее да влезе во неговата критична секција).

```
do {
       entry section
              critical section
       exit section
       // kod koj ne e del na kriticnata sekcija
} while (1);
```

13.4.3. Алгоритам на строга алтернација

Се користи заедничка променлива turn која одредува кој процес може да влезе во својата критична секција. Влезната секвенца е реализирана со *while* која чека променливата *turn* да биде еднаква со индексот на процесот.

```
do {
       while(turn != 0);   //vlezna sekcija
              //kriticna sekcija
       turn = 1;           //izlezna sekcija
              //ostatok od kodot
} while(1);


do {
       while(turn != 1);   //vlezna sekcija
              //kriticna sekcija
       turn = 0;           //izlezna sekcija
              //ostatok od kodot
} while(1);
```

За процес *Pi*:

```
do {
       while(turn != i);    //vlezna sekcija
                            //kriticna sekcija
       turn = j;            //izlezna sekcija
                            //ostatok od kodot
} while(1);
```

Во овој случај е задоволен условот да само еден процес може да влезе во својата критична секција, меѓутоа тоа влегување е по строго дефиниран редослед, па така процесот *P1* не може да влезе во својата критична секција ако процесот *P0* не ја поминал својата критична секција. Друг недостаток е тоа што се троши време за проверка на променливата *true*.

13.4.4. Реализација на критичната секција без строга алтернација

Строгата алтернација може да се избегне со воведување на бинарна променлива *flag* за секој процес посебно. Ако *flag[i]=1*, процесот *Pi* е спремен и сака да влезе во својата критична секција, а ако *flag[i]=0*, тогаш процесот *Pi* не сака да влезе во својата критична секција.

```
int flag[2];
flag[0] = flag [1] = 0;
```

За процесот *Pi*:

```
do {
       flag[i] = 1;
       while (flag[j]);    //vlezna sekcija
              //kriticna sekcija
       flag[i] = 0;
              //ostatok od kodot
} while(1);
```

Проблем кај овој алгоритам може да се јави ако *P0* го постави знаменцето на 1 а потоа контролата се префрли на процесот *P1* и тој го постави знаменцето на 1. Во овој случај и двата процеси влегуваат во бесконечна јамка.

13.4.5. Декер-Петерсонов алгоритам

Комбинација од претходните две решенија предложиле Декер и Петерсон кое се заснова на променливите *turn* и *flag*. Променливата *flag* кажува кој процес сака да влезе во својата критична секција, додека променливата *turn* одредува кој процес има предност (пример, 1. процесот *Pj* сака да влезе во својата

критична секција, *flag[j]=1*, 2. на процесот *Pj* му е дадена шанса за влез во критичната секција, *turn=j*). За процесот *Pi*:

```
do {
        flag[i] = 1;
        turn = j;
        while (flag[j] && turn == j);
                //kriticna sekcija
        flag[i] = 0;
                //ostatok od kodot
} while(1);
```

13.4.6. Семафори

За решавање на покомплексни случаи на синхронизација на процеси се користат семафори. Семафорите се целобројна променлива со ненегативна вредност која штити определен ресурс. Ако вредноста на семафорот *val(s)=0*, ресурсот е зафатен. Ако пак вредноста *val(s)>0*, тогаш ресурсот е слободен. Над семафорите се извршуваат операциите *wait(s)* која го декрементира семафорот само ако *s>0* и *signal(s)* која го инкрементира семафорот.

Проблемот на критична секција може да се реши со употреба на бинарен семафор *mutex* со почетна вредност 1.

```
semaphore mutex;           //inicijalno mutex=1
do {
        wait(mutex);
                //kriticna sekcija
        signal(mutex);
                //ostatok od kodot
} while(1);
```

Само еден процес ќе ја донесе вредноста на *mutex* на 0 и ќе влезе во својата критична секција, додека сите други процеси ќе чекаат на семафорот.

Семафорот може да се дефинира како структура чии елементи се целобројна вредност на семафорот (која може да биде и негативна) и листа на покажувачи кон процесите кои чекаат на семафорот.

```
typedef struct {
        int value;
        struct process *L;
};
```

Вредностите на семафорот може да бидат *>0* (ресурсот е слободен), =0 (ресурсот е зафатен, ама семафорскиот ред е празен) и *<0* (ресурсот е зафатен а во семафорскиот ред постојат процеси кои чекаат на тој семафор).

```
wait(S) {
        S.value--;
        if (S.value < 0) {
                //dodaj go procesot vo semaforskiot red
                sleep();
        }
};

signal(S) {
        S.value++;
        if (S.value <= 0) {
                //trgni go procesot P od semaforskiot red
                wakeup(P);
        }
};
```

Ако имаме лоша имплементација на семафорите, може да дојдеме во ситуација на застој.

Процес P0	**Процес P1**
wait(S)	wait(Q)
wait(Q)	wait(S)
...	...
signal(S)	signal(Q)
signal(Q)	signal(S)

Оваа секвенца ќе доведе до застој. Кога процесот *P0* ќе ја изврши операцијата *wait(S),* вредноста на семафорот *S* ќе биде 0 а истото ќе се случи и со семафорот *Q*. Тука процесите ќе застанат, бидејќи првиот процес ќе чека на семафорот *Q* а вториот на *S*.

13.4.7. Проблем на ограничен бафер

Следниот пример илустрира решение на проблемот производувач-потрошувач со употреба на семафори.

```
typedef struct {} item;     //structura na proizvodot
item buffer[N];             //definiranje na baferot
int in = 0;                 //prvo slobodno mesto
```

```
int out = 0;        //prvo zafateno mesto

podatokot_spremen = 0;
slobodno_mesto = N;
bafer_spremen = 1;         //mutex

//proizveduvac
do {
       item sleden_proizveden;
       wait (slobodno_mesto);
       wait (bafer_spremen);
              buffer[in] = sleden_proizveden;
              in = (in+1)%N;
       singal (bafer_spremen);
       signal (podatok_spremen);
} while(1);

//Potrosuvac
do {
       item sleden_potrosen;
       wait (podatok_spremen);
       wait (bafer_spremen);
              sleden_potrosen = buffer[out];
              out = (out+1)%N;
       signal (bafer_spremen);
       signal (slobodno_mesto);
} while(1);
```

13.4.8. Проблем на читач и пишувач

Во проблемот се вклучени читачи кои само ги читаат објектите и пишувачи кои сакаат да ја променат содржината на објектите. Два читача може да читаат објект, меѓутоа пишувач не смее да дозволи друг пишувач да ја менува содржината во исто време. Проблемот се решава со два *mutex* семафори *zapisuvanje*, кој служи за меѓусебно исклучување на процесите читач и пишувач и *mozna_izmena*, кој обезбедува меѓусебно исклучување кон пристапот кон променливата *broj_citaci*.

```
//Pisuvac
wait (zapisuvanje);
       // procesot go modifikuva objektot
signal (zapisuvanje);

//Citac
wait (mozna_izmena);
       broj_citaci++;
       if (broj_citaci == 1) wait (zapisuvanje);
signal (mozna_izmena);
       //procesot ja cita sodrzinata na objektot
wait (mozna_izmena);
       broj_citaci--;
       if (broj_citaci == 0) signal (zapisuvanje);
signal (mozna_izmena);
```

Процесот читач проверува дали има активни читачи и ако е прв чека да заврши пишувачот, доколку таков има, а ако не е прв слободно чита.

13.4.9. Филозофска вечера

Проблемот се решава така што секој филозоф претставува процес а секое стапче претставува семафор *stapce[i]*.

```
semaphore stapce[5];

//filozof [i]
do {
       wait (stapce[i]);
       wait (stapce[(i+1)%5]);
              //filozofot jade
       signal(stapce[i]);
       signal(stapce[(i+1)%5]);
              //filozofot razmisluva
} while (1);
```

Решението не е добро затоа што е можна појава на застој која може да се избегне ако:

- се дозволи да најмногу четири филозофи седат на масата,
- се дозволи филозофот да ги земе стапчињата ако и двете се слободни,
- непарен филозоф прво го земе левото а после десното стапче а парните обратно.

13.5. Вежба 5: Застои

На компјутерските системи бројот на ресурси е ограничен, така да повеќе процеси може да се натпреваруваат да ги користат овие ресурси. Кога процесот побарува некој ресурс, а ресурсот не е слободен, процесот влегува во состојба на чекање (*wait*) и се блокира се додека ресурсот не се ослободи. Процесот може да биде во состојба на чекање засекогаш, доколку тој ресурс е доделен на друг процес кој со текот на време исто така преминал во состојба на чекање на друг нерасположлив ресурс. Оваа ситуација се нарекува застој.

13.5.1. Банкарски алгоритам

Овој алгоритам претставува алгоритам за избегнување на застој а се применува доколку се исполнети следните услови:

- ресурсите имаат повеќе инстанци

- секој процес мора однапред да го најави бројот на најмногу инстанци од секој ресурс кои ќе ги користи
- кога процесот побарува ресурси, системот проценува дали после тоа ќе остане во стабилна состојба. Ако остане, процесот ќе добие ресурс а во спротивно процесот мора да почека некој друг процес да ослободи некој од ресурсите
- процесот кој добива ресурси, мора да ги врати во конечен рок

Нека n е бројот на процеси а m е бројот на ресурси. Се воведуваат следните структури на податоци:

- вектор на расположливите: $raspolozlivo[j]$, $j \in [0,m]$. Ако $raspolozlivo[j]=k$, тогаш k инстанци од ресурсот Rj се расположливи.
- матрица на максимални побарувања: $max[n,m]$. Ако $max[i,j]=k$, тогаш процесот Pi може да бара најмногу k инстанци од ресурсот Rj.
- матрица на алокации: $dodeleno[n,m]$. Ако $dodeleno[i,j]=k$, тогаш процесот Pi има добиено k инстанци од ресурсот Rj.
- матрица на потреби: $potreba[n,m]$. Ако $potreba[i,j]=k$, тогаш процесот Pi може да бара уште k инстанци од ресурсот Rj.

Притоа важи $potreba[i,j]=max[i,j]-dodeleno[i,j]$. Банкарскиот алгоритам одредува дали системот се наоѓа во безбедна состојба. Се состои од 4 чекори и тоа: иницијализација, пронаоѓање на процес кој може да се изврши, додела на ресурси на процесот и проверка за крај.

Задача 1: Во систем има еден ресурс A со 12 инстанци и три процеси со матрици *max* и *dodeleno* дадени во табелата. Да се одреди матрицата *potreba* и да се одреди дали системот е во безбедна состојба.

Процес	dodeleno	max	potreba
P0	5	10	5
P1	2	4	2
P2	2	9	7

Процес	dodeleno	max	potreba	raspolozlivo
P0	5	10	5	1
P1	4	4	0	
P2	2	9	7	

Процес	dodeleno	max	potreba	raspolozlivo
P0	5	10	5	5
P1	завршен			
P2	2	9	7	

Процес	dodeleno	max	potreba	raspolozlivo
P0	10	10	0	0
P1	завршен			
P2	2	9	7	

Процес	dodeleno	max	potreba	raspolozlivo
P0	завршен			10
P1	завршен			
P2	2	9	7	

Процес	dodeleno	max	potreba	raspolozlivo
P0	завршен			3
P1	завршен			
P2	9	9	0	

Процес	dodeleno	max	potreba	raspolozlivo
P0	завршен			12
P1	завршен			
P2	завршен			

Задача 2: Во системот има 5 процеси (*P0* до *P4*) и три ресурси A, B и C кои имаат 10, 5 и 7 инстанци.

а) да се одреди векторот *raspolozlivo* и матрицата *potreba*,

б) да се одреди стабилноста на системот за *baranje1=(1,0,2), baranje41=(3,3,0), baranje01=(0,2,0), baranje02=(1,2,2)* и *baranje42=(1,1,0).*

	dodeleno			max			potreba		
Ресурс	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

raspolozlivo = (3, 3, 2)

Вектор	dodeleno			max			potreba			raspolozlivo		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P1	3	0	2	3	2	2	0	2	0	2	3	0
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P1	3	2	2	3	2	2	0	0	0	2	1	0
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P1	завршен									5	3	2
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P3	2	2	2	2	2	2	0	0	0	5	2	1
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P3	завршен									7	4	3
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P4	4	3	3	4	3	3	0	0	0	3	1	2
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P4	завршен									7	4	5
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	7	5	3	7	5	3	0	0	0	0	0	2
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	завршен									7	5	5
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P2	9	0	2	9	0	2	0	0	0	1	5	5
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P2	завршен									10	5	7

Според Банкарскиот алгоритам постои секвенца на извршување на процесите *P1, P3, P4, P0* и *P2* при *baranje1*. За *baranje41* имаме:

Вектор	dodeleno			max			potreba			raspolozlivo		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P4	3	3	2	4	3	3	1	0	1	0	0	2

Системот влегува во застој, бидејќи после *baranje41* не може ниту еден процес да продолжи со неговото извршување. За *baranje01* имаме:

Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	3	0	7	5	3	7	2	3	3	1	2
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P3	2	2	2	2	2	2	0	0	0	3	0	1
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P3	завршен									5	2	3
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P1	3	2	2	3	2	2	0	0	0	4	0	1
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P1	завршен									7	2	3
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P2	9	0	2	9	0	2	0	0	0	1	2	3
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P2	завршен									10	2	5

Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	7	5	3	7	5	3	0	0	0	3	0	2
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	завршен									10	5	5
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P4	4	3	3	4	3	3	0	0	0	6	2	4
Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P4	завршен									10	5	7

Според Банкарскиот алгоритам постои секвенца на извршување на процесите *P3, P1, P2, P0* и *P4* при *baranje01*.

Вектор	**dodeleno**			**max**			**potreba**			**raspolozlivo**		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	1	3	2	7	5	3	6	2	1	2	1	0

По извршување на *baranje02*, не постои процес кој може да продолжи со неговото извршување, со што се заклучува дека системот е нестабилен. За доделувањето на ресурси според *baranje42=(1,1,0)* системот ќе остане стабилен а процесите ќе се извршат според секвенцата *P3, P4, P1, P2* и *P0*.

Задача 3: Постои систем со пет процеси (од *P0* до *P4*) и четири ресурси со вектор *raspolozlivo=(1,5,2,0)*. Состојбата на системот во момент t0 е дадена во табелата. Да се одреди матрицата *potreba*, дали системот е во стабилна состојба и дали може да биде одобрено барањето *baranje1(0,4,2,0)*.

	dodeleno				max				potreba				raspolozlivo			
Процес	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P0	0	0	1	2	0	0	1	2	0	0	0	0				
P1	1	0	0	0	1	7	5	0	0	7	5	0				
P2	1	3	5	4	2	3	5	6	1	0	0	2	1	5	2	0
P3	0	6	3	2	0	6	5	2	0	0	2	0				
P4	0	0	1	4	0	6	5	6	0	6	4	2				

- *P0* директно враќа (0,0,1,2) ресурси па расположливо ќе биде (1,5,3,2),
- *P3* зема (0,0,2,0), враќа (0,6,5,2) ресурси, а расположливо ќе биде (1,11,6,4),
- *P2* зема (1,0,0,2), враќа (1,7,5,0) ресурси, а расположливо ќе биде (2,14,11,8)
- *P1* зема (0,7,5,0), враќа (1,7,5,0) ресурси, а расположливо ќе биде (3,14,11,8)
- *P4* зема (0,6,4,2) а враќа (0,6,5,6) ресурси.

За *baranje1* се проверува дали важи *baranje1<=raspolozlivo* и бидејќи важи, на процесот *P1* се доделуваат (0,4,2,0) по што следи следната состојба.

	dodeleno				max				potreba				raspolozlivo			
Процес	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P0	0	0	1	2	0	0	1	2	0	0	0	0				
P1	1	0	0	0	1	7	5	0	0	3	3	0				
P2	1	3	5	4	2	3	5	6	1	0	0	2	1	1	0	0
P3	0	6	3	2	0	6	5	2	0	0	2	0				
P4	0	0	1	4	0	6	5	6	0	6	4	2				

- *P0* директно враќа (0,0,1,2) ресурси па расположливо ќе биде (1,1,1,2),

- *P2* зема (1,0,0,2), враќа (2,3,5,6) ресурси, а расположливо ќе биде (2,4,6,6),
- *P3* зема (0,0,2,0), враќа (0,6,5,2) ресурси, а расположливо ќе биде (2,10,9,8)
- *P1* зема (0,3,3,0), враќа (1,7,5,0) ресурси, а расположливо ќе биде (3,14,11,8)
- *P4* зема (0,6,4,2) а враќа (0,6,5,6) ресурси.

Задача 4: Во системот се наоѓаат пет процеси (од *P0* до *P4*) и три ресурси *A* (7 инстанци), *B* (3 инстанци) и *C* (6 инстанци). Состојбата на системот во t0 е дадена во табела. Во истиот момент *t0 raspolozlivo=(0,0,0).* Да се одреди дали системот е во застој ако:

- процесот *P2* не побарува ресурси
- процесот *P2* има *baranje2=(0,0,1)*

	dodeleno			max			potreba		
Ресурс	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	0	1	0	0	0	0
P1	2	0	0	4	0	2	2	0	2
P2	3	0	3	-	-	-	-	-	-
P3	2	1	1	3	1	1	1	0	0
P4	0	0	2	0	0	4	0	0	2

- Во моментот *t0* нема расположливи ресурси, но процесите *P0* и *P2* бидејќи немаат нови побарувања, ги враќаат доделените ресурси па *raspolozlivo=(3,1,3)*,
- *P3* зема (1,0,0), враќа (3,1,1) ресурси па *raspolozlivo=(5,2,4),*
- потоа *P1* зема (2,0,2) а враќа (4,0,2) ресурси а *raspolozlivo=(7,2,4)*
- на крај, *P4* зема (0,0,2) и враќа (0,0,4) ресурси.

Во *t0* нема слободни ресурси, меѓутоа процесот *P0* ослободува (0,1,0) па *raspolozlivo=(0,1,0)* што не е доволно да се изврши *baranje2*.

13.6. Вежба 6: Виртуелна меморија

Виртуелна меморија е техника која дозволува извршување на процеси чии делови може да бидат сместени на секундарните мемории, т.е. тврди дискови. Виртуелната меморија формира апстракција во вид на логичка меморија, која ја сочинуваат работната и секундарната меморија и ја раздвојува логичката од

физичката меморија. Количината на расположливата физичка меморија повеќе не претставува проблем, па програмерите не треба да се грижат за тоа.

Кога на некој процес ќе му затреба страница која не е во меморијата (се наоѓа на секундарните меморији), потребно е одредена страница која припаѓа на друг процес за биде запишана на секундарната меморија а потребната страница од секундарната во работната меморија. За ова намена се користат повеќе алгоритами меѓу кои најзначајни се: *Firs In First Out*, оптимален алгоритам, *Least Recently Used, Least Frequently Used* и *Most Frequently Used*.

13.6.1. Firs In First Out

Овој алгоритам при замената на страниците ја бара најстарата страница како жртва а на нејзино место ја става потребната.

Задача 1: Ако на процесот се доделени 3 рамки, колку PF прекини ќе бидат појавени за следната низа на мемориски референци: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
PF	PF	PF	PF		PF	PF	PF	PF	PF	PF			PF	PF			PF	PF	PF
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1

FIFO: PF=15

Задача 2: Колку PF прекини ќе се појават за следнава низа мемориски референци: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 ако на процесот се доделени:

а) 2 рамки

б) 3 рамки

в) 4 рамки

1	2	3	4	1	2	5	1	2	3	4	5
PF	PF	PF	PF	PF	PF	PF	PF	PF	PF	PF	PF
1	1	3	3	1	1	5	5	2	2	4	4
	2	2	4	4	2	2	1	1	3	3	5

FIFO: PF=12

1	2	3	4	1	2	5	1	2	3	4	5
PF	PF	PF	PF	PF	PF	PF			PF	PF	
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4

FIFO: PF=9

1	2	3	4	1	2	5	1	2	3	4	5
PF	PF	PF	PF			PF	PF	PF	PF	PF	PF
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3

FIFO: PF=10

Во последните два примери се забележува дека со зголемувањето на бројот на рамките, наместо намалување на PF доаѓа до зголемување на PF. Оваа аномалија се нарекува Беладиева (*Belady*) аномалија.

13.6.2. Оптимален алгоритам

Оптимален алгоритам е оној кој предизвикува најмало појавување на PF и никогаш не доведува до Беладиева аномалија. При замената ја жртвува страната која најдолго време нема да се користи, со што максимално се одолговлекува времето за појава на PF.

Задача 3: Да се реши Задача 1 со оптималниот алгоритам.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
PF	PF	PF	PF		PF		PF			PF			PF				PF		
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1

OPT: PF=9

За да се реализира овој алгоритам, мора да е познато на кој процес која страница ќе му биде потребна и кога. Но ова е тешко да се изведе поради интерактивноста на процесите.

13.6.3. Last Recently Used

Овој алгоритам како критериум го користи времето на последната употреба на страницата, што претставува податок од минатото а со тоа неговата имплементација возможна. По овој алгоритам се заменува страницата која најдолго не е користена. Овој алгоритам е доста добар и затоа често се користи.

Задача 4: Да се реши Задача 1 со примена на LRU.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
PF	PF	PF	PF		PF		PF	PF	PF	PF			PF		PF		PF		
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7

LRU: PF=12

Задача 5: Дадена е следната низа од 20 мемориски референци: 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6. Да се скицира состојбата во рамките и да се одреди бројот на PF според LRU, FIFO и OPT за:

а) 2 рамки

б) 3 рамки

в) 4 рамки

Решение:

а) 2 рамки

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
PF	PF	PF	PF	PF	PF	PF	PF	PF	PF		PF	PF	PF	PF	PF	PF		PF	PF
1	1	3	3	2	2	5	5	2	2	2	2	7	7	3	3	1	1	3	3
	2	2	4	4	1	1	6	6	1	1	3	3	6	6	2	2	2	2	6

LRU: PF=18

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
PF	PF	PF	PF	PF	PF	PF	PF	PF	PF		PF	PF	PF	PF	PF	PF		PF	PF
1	1	3	3	2	2	5	5	2	2	2	3	3	6	6	2	2	2	3	3
	2	2	4	4	1	1	6	6	1	1	1	7	7	3	3	1	1	1	6

FIFO: PF=18

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
PF	PF	PF	PF		PF	PF	PF		PF		PF	PF	PF		PF	PF		PF	PF
1	1	3	4	4	1	5	6	6	1	1	3	3	3	3	3	1	1	3	3
	2	2	2	2	2	2	2	2	2	2	2	7	6	6	2	2	2	2	6

OPT: PF=15

б) 3 рамки

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
PF	PF	PF	PF		PF	PF	PF	PF	PF		PF	PF	PF		PF	PF			PF
1	1	1	4	4	4	5	5	5	1	1	1	7	7	7	2	2	2	2	2
	2	2	2	2	2	2	6	6	6	6	3	3	3	3	3	3	3	3	3
		3	3	3	1	1	1	2	2	2	2	2	6	6	6	1	1	1	6

LRU: PF=15

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
PF	PF	PF	PF		PF	PF	PF	PF	PF		PF	PF	PF		PF	PF		PF	PF
1	1	1	4	4	4	4	6	6	6	6	3	3	3	3	2	2	2	2	6
	2	2	2	2	1	1	1	2	2	2	2	7	7	7	7	1	1	1	1
		3	3	3	3	5	5	5	1	1	1	1	6	6	6	6	6	3	3

FIFO: PF=16

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
PF	PF	PF	PF			PF	PF				PF	PF			PF	PF			PF
1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	6
	2	2	2	2	2	2	2	2	2	2	2	7	7	7	2	2	2	2	2
		3	4	4	4	5	6	6	6	6	6	6	6	6	6	1	1	1	1

OPT: PF=11

в) 4 рамки

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
PF	PF	PF	PF			PF	PF				PF	PF	PF			PF			
1	1	1	1	1	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	5	5	5	5	5	3	3	3	3	3	3	3	3	3
			4	4	4	4	6	6	6	6	6	7	7	7	7	1	1	1	1

LRU: PF=10

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
PF	PF	PF	PF			PF	PF	PF	PF		PF	PF	PF		PF	PF		PF	
1	1	1	1	1	1	5	5	5	5	5	3	3	3	3	3	1	1	1	1
	2	2	2	2	2	2	6	6	6	6	6	7	7	7	7	7	7	3	3
		3	3	3	3	3	3	2	2	2	2	2	6	6	6	6	6	6	6
			4	4	4	4	4	4	1	1	1	1	1	1	2	2	2	2	2

FIFO: PF=14

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
PF	PF	PF	PF			PF	PF					PF				PF			
1	1	1	1	1	1	1	1	1	1	1	1	7	7	7	7	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
			4	4	4	5	6	6	6	6	6	6	6	6	6	6	6	6	6

OPT: PF=8

13.7. Вежба 7: BOURN AGAIN SHELL скрипти

Shell скриптите (програмите) ги комбинираат UNIX командите така да се овозможи извршување на покомплексна структура на команди потребна за одредена задача. Примарен *shell* кој се користи во UNIX и Linux е BASH.

13.7.1. Пренасочување на стандардниот влез, излез и грешка

Повеќето команди се имплементирани така што како влез (Слика 13.8) го користат влезниот поток (стандарден влез), а резултатот од нивната работа го прикажуваат на излезниот поток (стандарден излез).

Слика 13.8. Влез и излез на UNIX програма

Подетално за ова во 13.9. Покрај овие два потока постои и поток на стандардна грешка во кој се насочуваат стандардните грешки кои се јавуваат при грешка во извршување на команда.

```
$ cat nova
```

ќе испише порака за грешка ако датотеката nova не постои:

```
cat: cannot open nova.
```

И ако се изврши пренасочување на стандардниот излез:

```
$ cat nova > izlez
cat: cannot open nova.
```

cat повторно ќе ја прикаже грешката на екранот. Пренасочување на стандардната грешка може да се изврши со помош на операторите 2> и 2>>. Првиот ќе ја креира датотеката во која е пренасочен потокот (или ќе ја пребрише ако не постои), а 2>> ќе ја надоврзе грешката. Операторот >& и двата излези ги пренасочува на иста локација:

```
$ cat postoecki nepostoecki >& izvestaj
```

13.7.2. Системски променливи (Environment variables)

Секогаш кога корисник се најавува на еден систем се дефинира работна околина во зависност од содржината на датотеките за иницијализација. Системските променливи се низи од знаци од облик:

```
име=вредност
```

каде што име е произволна низа од знаци која не смее да содржи $, & или празно место, а за вредност нема ограничувања од тој тип. На системските променливи им се доделува вредност со задавање на команда:

```
$ HELLO="Zdravo na site"
```

при што пред и по = не смее да има празно место. Ако веќе постои променлива со такво име, тогаш нејзината вредност ќе биде заменета со нова, а ако не постои, тогаш интерпретерот ќе креира нова променлива. Вредноста на системска променлива се добива со додавање на знак $ пред нејзиното име. За таа цел може да се употреби команда *echo* која ќе ја прикаже вредноста на променливата.

```
$ echo $HELLO
Hello world
```

Системските променливи можат да се дефинираат и така што ќе претставуваат команди. На пример ако се дефинира променлива со вредност команда, потоа со $ и името на променливата ќе се изврши таа команда:

```
$ PECATI="cat pismo"
$ $PECATI
```

Постои множество од системски променливи кои се креираат при секое најавување на системот. Некои од нив се за потребите на командниот интерпретер, а други можат да бидат во врска со одредени апликации. Кои променливи постојат и кои се нивните вредности, може да се види со командата:

```
$ env
```

Обично следните променливи се присутни кај секој кориснички профил:

LOGNAME - корисничко име

PATH - патеки до директориуми

HOME - домашен директориум на моменталниот корисник

PS1 - команден одзивен знак (*prompt*)

PWD – моментален директориум

Доколку е потребно променливата да биде видлива и во други скрипти тогаш треба да се експортира:

```
$ export HELLO
```

А потоа оваа променлива ќе се забележи во листата на променливи:

```
$ env | grep 'HELLO'
```

Со командата *unset* може да се тргне одредена променлива:

```
$ unset HELLO
```

13.7.3. BASH скрипти

BASH скриптите се текстуални датотеки кои се составени од UNIX (Linux) наредби. Првата команда во секоја скрипта е:

```
#!/bin/bash
```

која укажува на тоа која програма ќе ја извршува скриптата. *vi* едиторот го подржува форматирањето на BASH скриптите. # во другите наредби означува коментар. Скриптите вообичаено имаат наставка *sh*. Ќе се напише скрипта *prog.sh* која ќе ја прикаже пораката *Zdravo na site*.

```
#!/bin/bash
# prva nasa skripta
echo zdravo na site
```

Откако е креирана скриптата за да може да се изврши треба да се додели својство *executable* на датотеката. Тоа се постигнува со командата:

```
$ chmod +x prog.sh
```

Скриптата се извршува со:

```
$ ./prog.sh
```

Дадена е скрипта која на променливата *pozdrav* ќе и додели вредност *Zdravo* а потоа вредноста на променливата ќе се прикаже:

```
#!/bin/bash
pozdrav=″Zdravo″
echo $pozdrav
```

Дуплите наводници ја затвораат реченицата како една целина, а празните места се сметаат како карактери:

```
#!/bin/bash
pozdrav=″Zdravo na site″
echo $pozdrav
```

Во наредниот пример дуплите наводници ѕвездата ја претвораат во карактер:

```
#!/bin/bash
beleska=″Moze da gi izlistate programite so ls *.c″
echo $beleska
```

Доколку променливата е во дупли наводници таа се пресметува:

```
#!/bin/bash
pobednik=″Jas″
beleska=″Pobednikot e $pobednik″
echo $beleska
```

Ако променливата е во единични наводници, нејзината вредност не се пресметува:

```
#!/bin/bash
pobednik=″Jas″
beleska=′Pobednikot e $pobednik′
echo $beleska
```

Доколку е потребно да се пресмета променливата и да се прикаже знакот $ ќе употребиме \$ за негово прикажување:

```
#!/bin/bash
pobednik=”Jas”
beleska=”Pobednikot $pobednik osvoi \$100.000”
echo $beleska
```

Она што е означено во коси наводници се пресметува и се доделува на променливата:

```
#!/bin/bash
listc=`ls *.c`
echo $listc
```

ги листа сите датотеки во тековниот директориум со наставка c, додека:

```
#!/bin/bash
lst=`ls`
echo $lst
```

ги листа сите датотеки и поддиректориуми кои се наоѓаат во тековниот директориум. Со командата *read* се врши читање на вредност на променлива од тастатура:

```
#!/bin/bash
echo -n “Vnesete pozdrav”
read pozdrav
echo “Pozdravot koj go vnesovte bese $pozdrav”
```

Корисни оператори при креирањето на скрипти се операторите *let* и *test*. Првиот оператор пресметува математички израз додека вториот тестира одреден услов. Кога се користи командата *let* треба да се внимава помеѓу вредностите и операторот да нема празни места а може да има ако изразот е сместен во наводници:

```
let 2*7
let “2 * 7”
```

Пресметаниот израз може да се смести во променлива:

```
let "rez = 2 * 7"
echo $rez
```

Командата *test* е во следниот формат:

```
test vrednost -opcija vrednost
```

Резултатот од оваа команда е *true* или *false* а вообичаено се употребува кај *if* услови или *while* циклуси. Доколку треба да се споредат две низи од знаци се употребува:

```
test string = string
```

Наместо употреба на зборот *test* се користат [ ]. Пример:

```
[ $pozdrav = "hi"]
[ $num -eq 10 ]
```

За споредување на целобројни вредности се користат *gt* (>), *lt* (<), *ge* (>=), *le* (<=), *eq* (==), *ne* (!=). Во заградите се зададени соодветните оператори за споредување во програмскиот јазик C кои овде не се употребуваат. При споредување на целобројни вредности пред опцијата не треба да се заборави знакот минус (-). Најчесто споредување на стринговите е со *z* (празен стринг), =, !=. Она што е карактеристично и корисно за оваа команда се операторите за тестирање на датотеки и тоа: *f* (постои), *s* (не е празен), *r* (може да се чита), *w* (може да се запише во него), *x* (може да се изврши), *d* (станува збор за директориум), *h* (датотеката е симболички линк).

За да може да се искористат придобивките на UNIX наредбите, постојат повеќе структури и тоа *if, case, while, for* и други. Овде се опишани неколку примери каде се употребени овие структури. Првиот пример претставува скрипта каде во зависност од изборот на корисникот се листа содржината на тековниот директориум.

```
#!/bin/bash
echo -n "Vnesete izbor: "
read izbor
if [ "$izbor" = a ]
       then
              ls -a
       else
              ls -l
fi
```

Следниот пример прави проверка на бројот на аргументи и доколку не е 1 излегува од програмата со излез 1.

```
#!/bin/bash
if [ $# -ne 1 ]
       then
       echo "Pogresen broj na argumenti!"
       exit 1
fi
```

Наредниот пример е сличен како првиот со таа разлика што се користи и *elif* за испитување на нов услов.

```
#!/bin/bash
echo –n "Vnesete izbor: "
read izbor
if [ "$izbor" = a ]
       then
              ls -a
       elif [ "$izbor" = l ]
              then
                     ls -l
              else
                     echo "Pogresen argument"
fi
```

Доколку постојат повеќе услови кои треба да се испитаат може да се користи *case* структурата:

```
#!/bin/bash
echo –n "Vnesete izbor: "
read izbor
case $izbor in
       a)
              ls -a
              ;;
       l)
              ls -l
              ;;
       c)
              ls -c
              ;;
       *)
              echo "Pogresen broj na argumenti"
              ;;
esac
```

while циклусот извршува одреден број на наредби сè додека условот е точен. Скриптата во примерот му овозможува на корисникот да го внесува името сè додека името не е правилно внесено:

```
#!/bin/bash
povtorno=ne
while [ "$povtorno" = ne ]
do
       echo -n "Vnesi ime: "
       read ime
       echo "Imeto e $ime"
       echo -n "Dali sakas da prodolzis?"
       read povtorno
done
```

Скриптите можат да примаат аргументи од командната линија. При активирање на вакви скрипти по правило се користи слична синтакса како кај останатите UNIX команди. За процесирање на аргументите на командата линија постојат оператори и вградени системски променливи преку кои им се пристапува на аргументите. Променливата $# го содржи бројот на аргументи, а $* сите аргументи. Дадена е скрипта која го прикажува бројот на влезни аргументи и ги прикажува сите аргументи:

```
#!/bin/bash
echo $#
for VAR in $*
do
    echo $VAR
done
```

Скриптата се активира на стандардниот начин со таа разлика што после името следуваат аргументите.

```
$ ./skripta.sh prv vtor tret
3
prv
vtor
tret
```

Секој аргумент на командната линија е и индивидуално достапен преку променливите $1, $2 итн. Претходниот пример може да се напише и на следниот начин:

```
#!/bin/bash
echo $#
echo $1
echo $2
echo $3
```

Овој пример ги прикажува само првите три аргументи. Променливата $0 го содржи името на самата команда. Во случаи кога процедурата е стартувана без аргументи, може да се користи командата *set* која ќе ги постави променливите на предефинирани вредности.

```
#!/bin/bash
set prv vtor tret
echo $#
for VAR in $*
do
        echo $VAR
done
```

Скриптата сега се извршува без аргументи:

```
$./skripta.sh
3
prv
vtor
tret
```

Вообичаената синтакса на UNIX командите покрај аргументите содржи и опции. Преку функцијата *getopts* може да се врши парсирање на командната линија и процесирање на наведените опции. Функцијата *getopts* има два аргументи. Првиот е листа на дозволените букви, а после оние за кои мора да има и аргументи се става :. Вториот аргумент е привремена променлива во која се сместува буквата која се обработува во циклусот. Циклусот *while* се извршува по еднаш за секоја опција на командната линија. Да се претпостави скрипта која може да прима три опции *x, y* и *z* меѓутоа опцијата *z* да може да прима и аргумент. Стандардна имплементација на скриптата би била:

```
#!/bin/bash
while getopts yz:x c
do
        case $c in
                x)
                XFLAG=true
                ;;
```

```
                y)
                YFLAG=true
                ;;
                z)
                ZFLAG=true
                ZOPT=$OPTARG
                ;;
                *)
                echo $USAGE
                exit 2
                ;;
        esac
done
```

Скриптата може да се активира на следните начини:

```
$ ./script
$ ./script -x
$ ./script -z hello
$ ./script -y -z hello
```

Ако се зададе:

```
$ ./script -z
```

ќе се иницира прикажување на вредноста на променливата *USAGE* во која функцијата *getops* сместува порака за грешка при погрешна употреба на опциите. Променливата *OPTARG* го содржи аргументот соодветен на таа опција (ова се користи за опцијата *z*).

UNIX подржува и зависно извршување на програмите па така:

```
command1 && command2
```

прво ја извршува *command1*. Ако таа заврши со излезен код 0, тогаш се извршува и *command2*, во спротивно не се извршува *command2*.

Тоа е еквивалентно со:

```
if command1
then
    command2
fi
```

Типично се користи за кратенки од типот:

```
test -f file && rm file
```

со што се брише датотека само ако постои. Секвенцата:

```
command1 || command2
```

ја извршува *command1*. Ако не успее, ја извршува *command2*. Пример:

```
test -f file || (echo datotekata ne postoi && exit 1)
```

Со сместување на команда во загради (. . .) се предизвикува извршување на командата во нов процес. Тоа се користи во случаи кога е потребна измена на околината за некоја команда без да се влијае на други команди. На пример, за одредување на патеката на директориумот во погорно ниво може да се употреби:

```
(cd ..; pwd)
```

Проверка на бројот на аргументи на командна линија се изведува со:

```
#!/bin/bash
if [ $# -ne 3 ]
then
    echo "Usage: $0 arg1 arg2 arg3"
    exit 1
fi
```

или

```
test $# -ne 3 && echo Usage: $0 a1 a2 a3 && exit 1
```

Задача 1: Да се напише скрипта која ќе провери дали постои датотеката /etc/passwd

```
#!/bin/bash
file="/etc/passwd"
if [ -f $file ]
then
       echo "Датотеката $file postoi"
fi
```

Задача 2: Да се креира скрипта која ќе провери дали корисникот се наоѓа во домашниот директориум.

```
#!/bin/bash
dir=`pwd`
```

```
if [ $dir = $HOME ]
then
       echo "Se naogjas vo home direktoriumot"
else
       echo "Ne se naogjas vo home direktoriumot "
fi
```

Задача 3: Да се креира скрипта која ќе го креира /tmp/ss директориумот и ќе го извести корисникот дали директориумот успешно е креиран.

```
#!/bin/bash
mkdir /tmp/ss
ret=$?
if [ $ret -eq 0 ]
then
        echo "Direktoriumot e uspesno kreiran"
else
        echo " Direktoriumot ne e uspesno kreiran"
fi
```

Променливата $? го содржи излезот од последната извршена команда.

Задача 4: Да се креира празна датотека *nova* доколку не постои.

```
#!/bin/bash
if [ ! -f nova ]
then
       touch nova #kreira datoteka so golemina 0B
fi
```

Задача 5: Да се креира скрипта која ќе го прикаже бројот на датотеки со наставка .c кои содржат *if* услов.

```
#!/bin/bash
count=0
for VAR in `ls *.c`
do
echo $VAR
if [ `grep if $VAR|wc -l` -gt 0 ]
then
    let "count=$count + 1"
fi
done
echo "Vakvi datoteki ima: $count"
```

Задача 6: Да се напише скрипта која ќе ги прикаже броевите од 2 до 15 во растечки редослед со употреба на *while* циклус.

```
#!/bin/bash
val=1
while [ $val -lt 15 ]
do
        let "val=$val + 1"
        echo $val
done
```

Задача 7: Да се креира скрипта која ќе прикаже дали корисникот е *root* корисник или не.

```
#!/bin/bash
case $LOGNAME in
   petko)
        echo "obicen smrtnik"
        ;;
   root)
        echo "administrator"
        ;;
esac
```

Задача 8: Да се креира скрипта која ќе ја пресмета големината на датотеките во тековниот директориум.

```
#!/bin/bash
function fifth_arg {
   echo $5
}
sum=0
for file in *
do
   if [ ! -d $file ]
   then
      file_ls=`ls -l $file`
      size=`fifth_arg $file_ls`
      let sum=$sum+$size
   fi
done
echo $sum
```

Задача 9: Да се напише скрипта која ќе ги процесира сите датотеки кои може да се читаат.

```
#!/bin/bash
for file in *
do
    if [ ! -r $file ]
    then
```

```
        echo Ne moze da se procite $file
        continue
    fi
    # procesiranje na datoteka
done
```

Задача 10: Да се напише скрипта која ќе ги избрише сите најавувања и одјавувања на UNIX системот (датотека */var/log/wtmp*) и ќе ги избрише сите системски пораки (датотека */var/log/messages*) освен последните N, каде N е прв аргумент на скриптата а доколку не е внесен N=50.

```
#!/bin/bash
LOG_DIR=/var/log
ROOT_UID=0
LINES=50

if [ "$UID" -ne "$ROOT_UID" ]
then
     echo "Mora da ste najaveni kako root."
     exit 1
fi

if [ -n "$1" ]
then
     lines=$1
else
     lines=$LINES
fi

cd $LOG_DIR
if [ `pwd` != "$LOG_DIR" ]
     then
          echo "Ne moze da se vleze vo $LOG_DIR."
     exit 2
fi

tail -n $lines messages > mesg.temp
mv mesg.temp messages

cat /dev/null > wtmp
echo "Najavuvanjata se izbrisani."
exit 0
```

Задача 11: Да се напише скрипта која ќе чека корисникот за време од 4 секунди да внесе вредност во променливата, во спротивно ќе биде прикажано дека времето за внесување изминало.

```
#!/bin/bash
TIMELIMIT=4

echo -n "Vnesete vrednost za vreme od 4 sec: "
read -t $TIMELIMIT variable
```

```
if [ -z "$variable" ]
then
  echo "Vremeto istece, promenlivata ne e vnesena."
else
  echo "variable = $variable"
fi
```

Задача 12: Да се напише скрипта која на сите датотеки во тековниот директориум ќе им ја смени наставката (првиот аргумент е старата а вториот новата наставка).

```
#!/bin/bash
for filename in *.$1
do
          mv $filename ${filename%$1}$2
done
```

Задача 13: Да се напише скрипта која ќе генерира 10 случајни броеви.

```
#!/bin/bash

MAXCOUNT=10
count=1

echo
echo "$MAXCOUNT slucajni broevi:"
echo "-----------------"
while [ "$count" -le $MAXCOUNT ]
do
  number=$RANDOM
  echo $number
  let "count += 1"
done
echo "-----------------"
```

13.8. Вежба 8: RAID

RAID е ознака од *Redundant Array of Inexpensive Disks* (редундантна низа од евтини дискови). Целта е да повеќе мали дискови се поврзат и да служат како еден голем диск. Со ова се зголемуваат перформансите како паралелизам на операциите на дискот за големи побарувања (неколку диска работат паралелно) и се добива конкурентност во време за помали барања (неколку барања на различни дискови). Постојат две основни шеми и тоа огледало и паритет, додека организацијата може да е во 6 различни нивоа.

Задача 1: Креиран е RAID-0 од 4 дискови со големина 10 GB со *stripe* единица од 4 KB (8 блока). Дисковите се идентични со цена од 50$. Карактеристиките на дисковите се дадени во табела:

QUANTUM ATLAS-V	10 GB
average seek time	6.3 ms
rotational speed	7,200 rpm (8.33msec)
sustain data rate (media access)	24.5 MB/sec
MTTF(disk)	200.000 hours (23 years)

а) Да се скицира RAID-0 во *stripe* единици, колкав е капацитетот на RAID, колку блокови има и колку *stripe* единици?

Диск 1	Диск 2	Диск 3	Диск 4
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
			N-1

Капацитетот на овој RAID е 4 * 10 GB = 40 GB

```
N_of_blocks = N * N_of_blocks_per_disk = 4 * (10.000.000.000 / 512) =
       = 4 * 19.531.250 = 78.125.000 блокови
N_of_stripes = N_of_blocks / Stripe_size = 78.125.000 / 8 = 9.765.625
```

б) Да се одреди на кој диск се наоѓа логичкиот блок LA=153

Прво се одредува на кој stripe припаѓа а потоа тој stripe на кој диск:

```
Stripe = Integer(La / Stripe_size) = Integer(153 / 8) = 19
Disk = (Stripe % N_of_disks) = 19 % 4 = 3
```

Се наоѓа на Диск 4 (трет диск ако се брои од 0).

в) Моменталната состојба на I/O редот во однос на должината на трансферот е (1K, 12K, 4K, 8K, 2K). Колку барања може да опслужи овој RAID истовремено а колку воопшто?

Теориски може да се извршат онолку барања колку што има дискови по услов да е трансферот помал од *stripe* единица. Во овој случај тоа се 3 (за 1K, 2K и 4K) под услов да влегуваат во *stripe* и да се однесуваат на *stripe*-ови на различни дискови. Теоретски овој RAID може да изврши 4 барања истовремено.

г) Да се одреди цента по GB и процентуалната искористеност на просторот

Price_per_GB = Total_price / Total_capacity =

= 4 x 50$ / 4 x 10GB = 5$/GB

Искоритеноста на RAID-0 е 100%.

д) Да се одреди средното време на откажување (mean time to failure) на RAID-0

MTTF(RAID-0) = MTTF(disk) / N = 200.000 / 4 =

= 50.000 часа (≈ 6 години)

ѓ) Да се одреди оптималната stripe единица за степен на конкурентност 4 req/s и просечна големина на трансферот 5K.

Ова се пресметува со користење на формулата:

Stripe_unit = sqr([P * X * (L-1) * Z] / N)

каде

P е просечно време за позиционирање,

X просечна брзина на пренесување,

L е конкурентност,

Z е големина на побарувањето

N е број на дисковите во низата

P = Average_seek_time + Average_rotational_time =

= 6.3 + (8.33 / 2) = 10.47 msec

X = 24.5 MB/s

```
L = 4 req/s
Z = 5K
N = 4
Stripe_unit = 31.02K
```

а се заокружува на 32K, 64 блока.

Задача 2: Креиран е RAID-1 од 4 дискови со големина 10GB со *stripe* единица од 4 KB (8 блока). Дисковите се идентични со цена од 50$. Карактеристиките на дисковите се дадени во табела:

QUANTUM ATLAS-V	10 GB
average seek time	6.3 ms
rotational speed	7,200 rpm (8.33msec)
sustain data rate (media access)	24.5 MB/sec
MTTF(disk)	200.000 hours (23 years)

а) Да се скицира RAID-1 во *stripe* единици, колкав е капацитетот на RAID и колку блокови има?

Диск 1	Диск 2	Диск 3	Диск 4
0	0	N	N
1	1	N+1	N+1
2	2	N+2	N+2
3	3	N+3	N+3
4	4	N+4	N+4
5	5	N+5	N+5
N-1	N-1	2N-1	2N-1

Капацитетот на овој RAID е 2 * 10 GB = 20 GB

```
N_of_blocks = (N / 2) * N_of_blocks_per_disk =
= 2 * (10.000.000.000 / 512) =   2 * 19.531.250 = 39.062.500 блокови
```

б) Да се одреди на кој диск се наоѓа логичкиот блок LA=335

Блоковите од 0 до 19.531.250 се наоѓаат на првиот диск, па така LA=335 се наоѓа на Диск 1 (а копијата на Диск 2).

в) Моменталната состојба на I/O редот во однос на должината на трансферот е (1K, 12K, 4K, 8K, 2K). Колку барања може да опслужи овој RAID истовремено а колку воопшто?

Теориски може да се извршат онолку барања колку што има парови на дискови по услов да се однесуваат на различни дискови (2 барања истовремено). За случајот даден повторно се 2 да се наоѓаат на различни дискови.

г) Да се одреди цента по GB и процентуалната искористеност на просторот

Price_per_GB = Total_price / Total_capacity =

= 4 x 50$ / 2 x 10GB = 10$/GB

Искористеноста на RAID-1 е 50%.

д) Да се одреди средното време на откажување (*mean time to failure*) на RAID-1 ако MTTR=1 час (*mean time to repair*)

MTTF(RAID-1) = MTTF2(disk) / (2 * MTTR) = 200.0002 / (2 * 1) =

= 20.000.000.000 часа = 2.283.105 години

Задача 3: Креиран е RAID-3 од 4 дискови со големина 10GB со *stripe* единица од 1B. Дисковите се идентични со цена од 50$. Карактеристиките на дисковите се дадени во табела:

QUANTUM ATLAS-V	10 GB
average seek time	6.3 ms
rotational speed	7,200 rpm (8.33msec)
sustain data rate (media access)	24.5 MB/sec
MTTF(disk)	200.000 hours (23 years)

а) Да се скицира RAID-3 во *stripe* единици, колкав е капацитетот на RAID и колку блокови има?

Диск 1	Диск 2	Диск 3	Диск 4
0	1	2	P
3	4	5	P
6	7	8	P
9	10	11	P
12	13	14	P
15	16	17	P
N-3	N-2	N-1	P

Капацитетот на овој RAID е 3 * 10 GB = 30 GB

```
N_of_blocks = (N - 1) * N_of_blocks_per_disk =
= 3 * (10.000.000.000 / 512) =   3 * 19.531.250 = 58.593.750 блокови
```

б) Да се одреди на кој диск се наоѓа логичкиот блок LA=21411

Се наоѓа на сите три диска.

в) Моменталната состојба на I/O редот во однос на должината на трансферот е (1K, 12K, 4K, 8K, 2K). Колку барања може да опслужи овој RAID истовремено а колку воопшто?

За RAID-3 со *stripe*=1B може да се изврши едно единствено барање во кое учествуваат сите дискови.

г) Да се одреди цента по GB и процентуалната искористеност на просторот

```
Price_per_GB = Total_price / Total_capacity =
= 4 x 50$ / 3 x 10GB = 6,66 $/GB
```

Искористеноста на RAID-1 е 75%.

Литература

1. Andrew. S. Tanenbaum, *Modern Operating Systems*, Second Edition, Prentice Hall, (2001)
2. Abraham Silberschatz et al., *Operating System Concepts*, 7th edition, Wiley, (2004)
3. William Stallings, *Operating Systems, Internals and Design Principles*, Fifth Edition, Pearson, (2005)
4. Borislav Djordjevic et al., *Operativni sistemi, teorija, praksa i reseni zadaci*, Mikro Knjiga, (2005)
5. Jean Bacon, Tim Harris, *Operating Systems: Concurrent and Distributed Software Design*, Addison Wesley, (2003)
6. Miroslav Hajduković, *Operativni sistemi (Problemi i struktura),* Elektronsko izdanje, verzija 2, Novi Sad, (2004)
7. Mark Minasi, Mastering Windows Server 2003, Sybex, (2003)
8. Norton et al., *Complete Guide to Windows 2000 Professional*, (1999)
9. Andrew. S. Tanenbaum, Albert Woodhull, *Operating Systems: Design and Implementation*, Third Edition, Prentice Hall, (2006)
10. Andrew. S. Tanenbaum, *Structured Computer Organization*, Fifth Edition, Prentice Hall, (2006)
11. Olaf Kirch & Terry Dawson, *Linux Network Administrator's Guide*, 2nd Edition, (2000)
12. Tom Adelstein, Bill Lubanovic, *Linux System Administration*, O'Reilly, (2007)
13. *Linux Bible, Boot Up to Fedora™, KNOPPIX, Debian®, SUSE™, Ubuntu™, and 7 Other Distributions*, Wiley Publishing, (2006)
14. Paul Sheer, *Linux: Rute User's Tutorial and Exposition*, (2001)
15. Peter Jay Salzman, Michael Burian, Ori Pomerantz, *The Linux Kernel Module Programming Guide*, (2001)
16. Arnold Robbins, *UNIX in a Nutshell,* O'Reilly, (2005)
17. Kenneth H. Rosen, *UNIX: The Complete Reference*, McGraw-Hill Professional, (1999)