# 1 Увод у оперативне системе

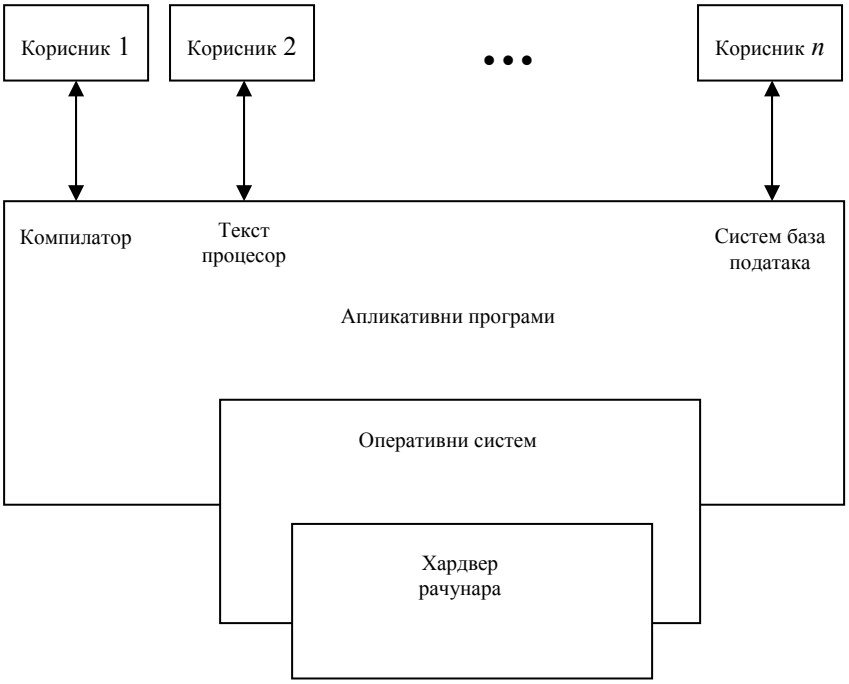
Рачунарски систем састоји се од две основне компоненте: хардвера и софтвера. Као што је познато, хардвер представља саму машину, док софтвер представља програме. Предмет нашег изучавања биће *оперативни системи*, посебна врста софтвера без које употреба саме машине практично не би била могућа.

Оперативни систем је програм који представља интерфејс између хардвера рачунара и корисника рачунарског система. Његова сврха је да обезбеди околину у којој се извршавају кориснички програми. Према томе, основни циљ оперативног система је да учини рачунарски систем погодним за употребу. Други циљ је да се коришћење ресурса рачунарског система одвија на ефикасан начин.

## 1.1 Дефиниција и задаци оперативног система

Рачунарски систем се грубо може поделити на четири компоненте (слика 1.1):

- Хардвер (CPU, меморија, У/И уређаји).
- Оперативни систем.
- Апликативни софтвер (компилатори, системи за управљање базама података, текст процесори, графички програми, итд.).
- Корисници (људи, други рачунари или друге машине).




Сл. 1.1. Апстрактни поглед на компоненте рачунарског система.

Хардвер рачунара пружа основне ресурсе. Апликативни програми дефинишу начине на које се ти ресурси користе за решавање неког од проблема корисника. У систему може постојати више различитих корисника који покушавају да реше различите проблеме. Стога могу да постоје различити апликативни

програми. Оперативни систем управља доделом хардверских ресурса и врши координацију употребе хардвера између различитих апликација различитих корисника.

Из ових разлога можемо посматрати оперативни систем као алокатор ресурса. Рачунарски систем садржи различите ресурсе који могу бити хардверски или софтверски као што су: CPU време, меморијски простор, простор за смештај датотека, У/И уређаји итд. Оперативни систем делује као менаџер ових ресурса додељујући их различитим програмима и корисницима на основу њихових потреба током извршавања њихових задатака. Како је сасвим могуће да се јави више захтава за ресурсима који често могу бити међусобно супростављени, оперативни систем мора да одлучује о томе како ће додељивати ресурсе како би се рад рачунарског система одвијао на непристрасан и ефикасан начин.

Уколико се усресредимо на управљачку функцију оперативног система, можемо рећи да је оперативни систем управљачки програм који настоји да извршење корисничких програма тече без грешака. Ово се нарочито односи на управљање У/И уређајима.

Генерално, не постоји потпуно адекватна дефиниција оперативног система. Разлог за постојање оперативног система је у томе што су оперативни системи разуман начин да се креира употребљив рачунарски систем. Можда је лакше дефинисати оперативни систем дефинишући шта ради један оперативни систем него шта је он у ствари.

Већ смо рекли да су два основна циља оперативног система да рачунарски систем учини погодним за употребу и да обезбеди ефикасан рад рачунарског система. Ова два циља су понекад у контрадикцији. У прошлости се ефикасност сматрала важнијом од погодне употребе, тако да се теорија оперативних система сконцентрисала углавном на оптималну употребу ресурса.

## 1.2 Зачеци оперативних система

На почетку, постојао је само хардвер рачунара. Први рачунари су били физички огромне машине којима се управљало са конзоле. Програмер би написао програм и затим управљао радом програма са конзоле. Најпре би се програм ручно унео у меморију, било преко плоче се прекидачима, папирне траке или бушених картица. Затим би се притиском на одређене тастере унела почетна адреса програма и започело његово извршавање. Програмер који је уједно био и оператер, пратио би извршавање програма преко нарочитих дисплеја на конзоли. У случају грешака, програмер би заустављао програм, испитивао садржај регистара и меморије и отклањао грешку директно са конзоле. Излаз би се штампао или бушио на траку или картицу ради каснијег штампања.

Многи системи користили су *шему резервације*. Онај ко би желео да користи рачунар би се уписивао у распоред тражећи себи слободно време и резервишући га. Међутим, некада је предвиђено време било прекратко (у случају када се јаве проблеми) а некада не би било искоришћено у потпуности.

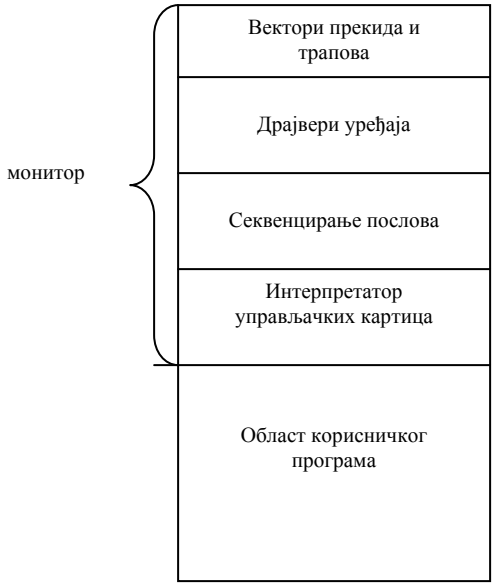
Током времена почело се са коришћењем читача картица, штампача и магнетних трака, Развијени су асемблери, лоудери и линкери како би олакшали програмирање. Креиране су библиотеке функција. Јављају се драјвери уређаја као специјални програми, написани за сваки уређај, а који су узимали у обзир посебности сваког уређаја.

Нешто касније, са појавом виших програмских језика, посао програмирања је олакшан, али је зато отежан посао оператера. Да би се, на пример, извршио неки програм написан у Fortran-у, било је потребно најпре напунити Fortran преводилац у рачунар, тј. монтирати одговарајућу магнетну траку. Програм би се читао са бушених картица и уписивао такође на магнетну траку. Преводилац би генерисао асемблерски код који је морао да се асемблира сличним поступком а затим да се изврши линковање. После овога би се програм извршавао и по потреби пречишћавао. При овоме је *време постављања* било огромно.

Решење је потражено на два начина. Унајмљени су професионални оператери, тако да програмер и оператер више нису били једна особа. Није више било нужно да програмери резервишу време на рачунару. Други начин је подразумевао увођење *пакетне обраде*. Послови сличне природе груписани су у пакете и извршавали један за другим.

Ова решења су мало побољшала искоришћеност рачунара. Остајао је проблем у томе што је, приликом прекида рада програма, оператер утврђивао да ли је прекид био нормалан или абнормалан, по потреби вршио дамп меморије и потом пунио читач картица новим послом, остављајући процесор неискоришћеним за то време.

Да би се и овај проблем превазишао, уведено је *аутоматско секвенцирање послова* и први рудиментарни оперативни системи. На тај начин вршен је аутоматски пренос управљања са једног посла на други. Креиран је мали програм назван *резидентни монитор* (слика 1.2). На почетку рада рачунара управљање се предаје резидентном монитору који је даље преносио управљање програмима. Када се неки програм заврши контрола се поновао враћа резидентном монитору који опет иде даље на други програм.




Слика 1.2. Изглед меморије за резидентни монитор.

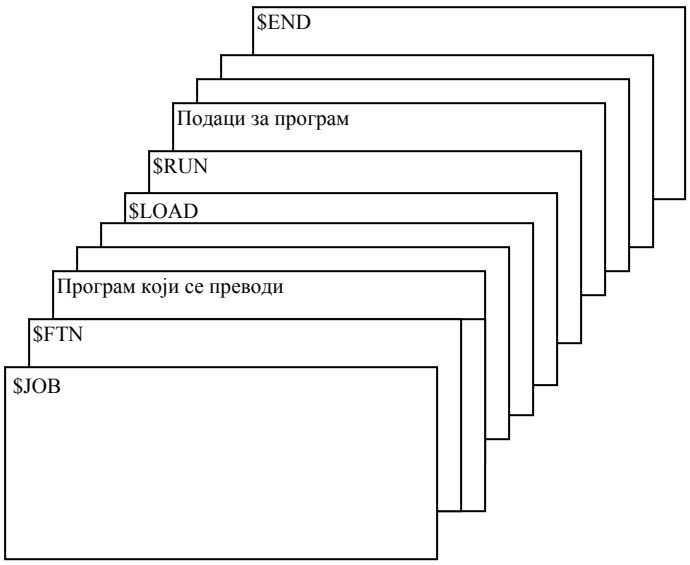
Да би резидентни монитор знао који програм треба да се извршава уведене су управљачке картице које су указивале монитору који програм треба да се изврши. На пример,

$FTN – изврши Fortran преводилац
$ASM – изврши асемблер
$RUN – изврши кориснички програм

За одређивање граница послова користиле се се још две картице:

$JOB – Прва картица посла
$END – Последња картица посла

Знак $ одвајао је управљачке картице од картица које су представљале програме или податке. Неки произвођачи су користили и друге знаке (нпр. //). На слици 1.3 видимо пример скупа картица за извршавање једног пакета послова.




Сл. 1.3. Скуп картица потребних за један пакет послова.

## 1.3 Побољшавање перформанси првобитних оперативних система

Рачунари су, поготову велики, били веома скупе машине, тако да се захтевало да извршавају што је могуће више послова. Прелазак на пакетне системе са аутоматским секвенцирањем послова био је извршен управо из разлога побољшања перформанси. Људи су, у поређењу са рачунаром, веома спори па је пожељно да се што више људских операција замени софтвером оперативног система. Међутим и код овог приступа је CPU остајао неупослен. Механички У/И уређаји су много спорији од електронских уређаја. То значи да CPU често чека на У/И.

### *1.3.1 Рад off-line*

Једно решење проблема састојало се да се улаз са спорих, механичких уређаја ископира најпре на магнетну траку. Када се трака напуни онда она служи као улаз рачунарског система. Приликом излаза се на сличан начин пунила магнетна трака, а штампање је вршено касније. Такав рад читача картица и штампача назива се *off-line*.

Постојала су два приступа овом проблему. Уређаји специјалне намене, као што су читачи картица, линијски штампачи и сл., прављени су тако да је њихов излаз, тј. улаз магнетна трака. Други приступ је подрзумевао употребу малог рачунара који би био посвећен копирању са или на траку. Тај мали рачунар је представљао неку врсту сателита већем рачунару. *Сателитска обрада* је један од првих случаја система са више рачунара који раде заједно у циљу побољшања перформанси.

Главна предност *офф-лине* рада је у томе што главни рачунар није више био ограничен брзином спорих У/И уређаја, већ само брзином много бржих јединица магнетних трака. Осим овога, програми су на тај начин могли да се извршавају независно од типа У/И уређаја (*device independence*). Програми су писани да користе *логичке* У/И уређаје, док се упарављачким картицама или другим командама указивало како се логички уређај пресликава на физички уређај.

### *1.3.2 Баферовање*

Још једно решење проблема спорости У/И уређаја је *баферовање*. Тежња код баферовања је да се и CPU и У/И уређају упосле све време. Пошто се подаци учитају и CPU почиње да ради са њима, улазни уређај одмах почиње са наредним улазом. Уз мало среће, док CPU обради претходне податке, нови улаз биће учитан. Сличан процес може да се одвија и приликом излаза.

Бафери су обично садржавали блокове података који су се сатојали од неколико логичких записа. Проблем који се код имплементације баферовања јављао састојао се у откривању када је У/И уређај завршио свој рад што је пре то могуће. Овај проблем се решавао увођењем *система прекида*. Такође се уводи и *директни приступ меморији* (DMA).

Баферовање није увек довољно да се повећа ефикасност рада рачунара, нарочито код програма који су екстремно посвећени У/И активностима или CPU активностима.

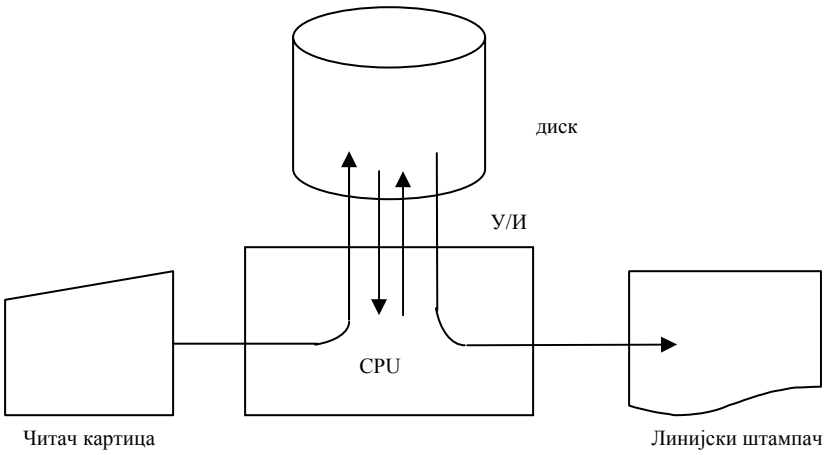
### *1.3.3 Spooling*

Припрема poslova za *off-line* рад се наставила неко време, али је убрзо била замењена у многим системима. Системи дискова су ушли у широку употребу. Проблем са магнетним тракама је био што, рецимо читач картица није могао да уписује на траку на једном крају а CPU да чита податке са другог краја, јер је магнетна трака полуадресиви медијум са уграђеном секвенцијалном природом приступа.

Начин обраде који се одвијао помоћу дискова називао се *Spooling* (*Simultaneous Peripheral Operation On-Line*). Овде се диск користи као велики бафер (слика 1.4).

Баферовање преклапа У/И активности неког посла са његовим CPU активностима. Предност *spooling*-а над баферовањем је што је овде У/И активност једног посла преклопљена са CPU активностима других послова. Екстремни облик spooling-а је да се читав садржај магнетне траке ископира на диск, тако де се све операције обављају са копијом на диску и то многом већом брзином (овај шема се назива *staging a tape*).

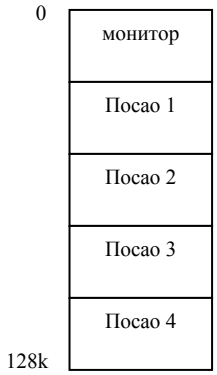
*Spooling* има директан утицај на побољшање перформанси. По цени извесног простора на диску и неколико табела које прате садржај на диску, омогућено је преклапање CPU активности једног посла са У/И активношћу другог. Осим овога, *spooling* обезбеђује важну структуру података: *pool* послова. *Pool* послова омогућује да оперативни систем може да међу спремним пословима из поола одабере који ће се следећи извршити и на тај начин да повећа ефикасност. Док су послови долазили директно са читача картица па и са магнетне траке, они су морали да се извршавају у секвенцијалном FIFO поретку. Међутим, када имамо неколико послова на брзом уређају са директним приступом, као што је диск, омогућен је *планирање послова* (*job scheduling*).




Сл. 1.4. *Spooling*.

### *1.3.4 Мултипрограмирање*

Најважнији аспект планирања послова је способност мултипрограмирања. Идеја мултипрограмирања састоји се у следећем. Оперативни систем бира један од послова из поола послова и почиње да га извршава. У одређеном тренутку посао ће чекати на монтирање траке, на команду са тастатуре, завршетак У/И операције или слично. У том случају се оперативни систем једноставно пребаци на други посао и почиње њега да извршава. Када се слична ствар догоди и са тим послом прича се понавља. Пре или касније ће први посао завршити своје чекање и добиће поново CPU како би се наставио. Докле год постоји неки посао који треба извршавати CPU није беспослен.




Сл. 1.5. Приказ меморије за систем са мултипрограмирањем.

Оперативни системи са мултипрограмирањем су прилично сложени. Да би могли да имамо неколико послова спремних за извршавање, они морају да се истовремено налазе у меморији (слика 1.5). То повлачи да морају да постоје средства за управљање меморијом. Такође морамо да имамо и могућност краткорочног планирања послова (*cpu scheduling*), планирања У/И уређаја, управљања застојима, управљања конкуренцијом, заштите и др.

### *1.3.5 Дељење процесорског времена (Time Sharing)*

Системи са дељењем процесорског времена су резултат покушаја да се обезбеди интерактивна употреба рачунарског система по разумној цени. Овакав оперативни систем користи планирање ЦПУ-а и мултипрограмирање дозвољавајући сваком кориснику да користи процесор током малог, унапред дефинисаног временског интервала. Сада није проблем што корисницима треба релативно пуно времена да изврше унос са тастатуре или прочитају са екрана монитора резултате рада програма, јер се управљање процесором преноси на друге послове током сваког временског интервала.

### *1.3.6 Системи за рад у реалном времену (Real-Time Systems)*

Још један облик оперативних система јесу системи за рад у реалном времену. Овакви системи се често користе као контролери у специјалним апликацијама. Различити сензори снабдевају рачунар подацима. Рачунар анализира те податке и по потреби подешава управљање како би променио улаз у сензоре. Обрада код оваквих система мора бити завршена у оквиру унапред заданог временског периода.

## 1.4 Проблеми заштите

Док су први рачунари били једнокориснички системи са једним програмером који је уједно био и оператер, каснији оперативни системи су омогућили дељење системских ресурса између више програма истовремено. То је, поред повећене ефикасности, наметало проблем заштите, како У/И операција, тако и меморије и процесора од непожељних ефеката.

Да би се спречило да кориснички програми, услед неке грешке, приликом, на пример, читања картица почињу да читају картице других програма и тако доведу до нежељених понашања система, уведена су два одвојена начина рада; *кориснички* и *мониторски* (*супервизорски* или *системски*). Тада се све У/И инструкције дефинишу да буду *привилеговане* а хардвер дозвољава извршавање привилегованих инструкција само у мониторском начину рада. На тај начин се обављање У/И операција поверава оперативном систему, а не корисничким програмима. Када кориснички програм жели да обави неку У/И активност он обавља тзв *системски* или *мониторски* позив. Системски позив ће обавити У/И операцију на ваљан начин и резултат проследити корисничком програму.

Како више послова постоји у меморији, то је неопходно спречити корисничке програме да задиру у део меморије додељен другом послу или самом монитору. Уводе се специјални регистри који садрже граничне адресе за област меморије додељене једном послу. Свака адреса генерисана у корисничком начину рада проверава се у односу на границе у граничним регистрима и на тај начин спречавају меморијски конфликти.

Када је уведно дељење процесорског времена морала су се обезбедити средства која ће омогућити да се то дељење обавља на коректан начин. Из тих разлога уводе се тајмери који се ресетују од стране монитора када се процесор додели неком послу. По истеку дефинисаног периода генерише се прекид после кога се управљање враћа монитору који може да одлучи да ли да послу додели додатно време или да процесор додели другом корисничком програму.

# 2 Сервиси оперативног система

Оперативни систем обезбеђује окружење у оквиру којег се извршавају програми. Већ смо видели да се У/И активности корисничких програма одвијају искључиво посредством оперативног система. Упознаћемо се са сервисима које пружа оперативни систем.

## 2.1 Типови сервиса

Оперативни систем пружа одређене услуге програмима и корисницима тих програма. Специфични сервиси се разликују од једног до другог оперативног система, али постоје неке заједничке класе сервиса које се могу идентификовати. Те функције оперативног система су на располагању ради олакшавања посла програмеру.

- **Извршавање програма**. Корисници желе да изврше програме. Систем мора да буде у стању да напуни програме у меморију и да их покрене. Програм мора да буде у стању да оконча своје извршење било нормално било абнормално.
- **У/И операције**. Програми који се извршавају могу да захтевају улаз или излаз. Овај У/И може да ангажује датотеке или У/И уређаје. За специфичне уређаје могу се захтевати посебне функције. Пошто корисник не може да извршава У/И операције директно, оперативни систем мора да обезбеди начин да се то уради.
- **Руковање фајл системом**. Датотечни систем је од нарочитог значаја, јер је очигледно да програми имају потребу да читају из датотека или да у њих уписују. Такође се јавља потреба да се датотеке креирају и бришу.
- **Детекција грешака**. Оперативни систем мора непрестано да буде у стању да открива могуће грешке. Грешке се могу јавити у хардверу CPU-а или меморије, код У/И уређаја или у корисничким програмима. За сваки тип грешке оперативни систем мора да предузме одговарајућу акцију да би обезбедио коректно и конзистентно израчунавање.

Осим ових, постоји још један скуп функција које постоје не ради корисника, већ ради ефикасности самог система. Вишекориснички системи побољшавају своју ефикасност дељењем рачунарски ресурса.

- **Додела ресурса.** Када постоји више корисника или се више послова извршава у исто време ресурси се морају додељивати сваком од њих. Оперативни систем управља различитим типовима ресурса.
- **Вођење налога.** Пожељно је пратити који корисници, у којој мери и коју врсту рачунарских ресурса користе. Ово може бити пожељно из разлога плаћања коришћења система или ради вођења статистике.
- **Заштита.** Власници информација запамћених у вишекорисничком систему желе да управљају њиховим коришћењем. Када се више различитих послова извршава истовремено ради боље искоришћености система, не сме бити омогућено да послови утичу једни на друге. Осим тога, могући су конфликтни захтеви за различитим ресурсима који се морују непристрасно и разумно разрешити.

## 2.2 Кориснички поглед

Сервиси оперативног система се обезбеђују на различите начине. Два основна метода су *системски позиви* и *системски програми*. Сваки од ових метода има своје предности.

### *2.2.1 Системски позиви*

Најосновнијим нивоом услуга управља се коришћењем системских позива. Системски позиви обезбеђују спрегу између програма који се извршавају и оперативног система. Ови позиви су у општем случају расположиви као инструкције асемблерског језика.

Системски позиви се грубо могу поделити у три велике категорије: управљање процесима или пословима, руковање уређајима и датотекама и одржавање информација.

**Управљање процесима и пословима.**

Програм који се извршава мора да буде у стању да прекине своје извршење било нормално (**end**) било абнормално (**abort**). Ако програм открије грешку у његовом улазу и жели да се заврши абнормално, он ће такође желети и да дефинише ниво грешке. На озбиљније грешке указује већа вредност нивоа грешке. Могуће је посматрати нормални завршетак програма као абнормални са нивоом грешке једнаким 0.

Процес или посао који извршава један програм може да захтева пуњење (**load**) и извршење (**execute**) другог програма. Ово допушта интерпретатору управљачких картица да изврши програм као што то налажу управљачке картице корисничког посла. Интересантно је питање где се враћа управљање када се напуњени програм заврши. Ово питање је у вези са тим да ли је постојећи програм изгубљен, запамћен или му је допуштено да настави своје извршење конкурентно са новим програмом.

Ако се управљање враћа постојећем програму када се нови програм заврши, морамо да сачувамо слику меморије постојећег програма као и да имамо изграђен механизам позива једног програма од стране другог програма.

Ако се оба програма настављају конкурентно креирали смо нови процес који треба да се мултипрограмира. Често постоји специфичан системски позив са том улогом (**create process** ili **submit job**). Ако креирамо нови посао или процес, или чак скуп послова или процеса, морамо да будемо у стању да управљамо њиховим извршењем. То подразумева могућност да се одреде и ресетују атрибути посла или процеса као што су приоритет, максимално дозвољено време извршавања итд. (**get process attributes** i **set process attributes**). Такође ћемо хтети да завршимо процес или посао који смо креирали (**terminate process**) ако утврдимо да се не обавља коректно или да више није потребан.

Када смо креирали нове послове или процесе, можда ћемо желети да сачекамо да се они изврше. Можемо да чекамо извесно време (**wait time**) или на одређени догађај (**wait event**). Послови или процеси треба да јаве када се догађај јавио (**signal event**).

Системски позиви као што су **dump** и **trace** помажу у пречишћавању програма.

**Руковање датотекама**

Морамо да будемо у стању да креирамо (**create**) и бришемо датотеке (**delete**). Пре употребе датотака мора да се отвори (**open**), а може се читати (**read**), уписивати (**write**) или јој се може променити текућа позиција (**reposition**). Најзад, датотека се затвара (**close**). Уколико имамо директоријумску структуру, биће нам потребни слични системски позиви и за директоријуме.

Осим овога, потребно је да постоји и могућност очитавања и постављања атрибура датотекама (**Get File Attributes** и **Set File Attributes**).

**Управљање уређајима**

Датотеке се могу сматрати виртуелним уређајима. Тако су многи системски позиви за датотеке потребни и за уређаје. Ако у систему постоји више корисника, морамо најпрве захтевати (**request**) уређај. По употреби уређај се мора ослободити (**release**).

**Одржавање информација**

Многи системски позиви постоје само ради преноса информација између корисничких програма и оперативног система. На пример, **time** и **date** враћају системско време и датум. Могу постојати и други системки позиви који враћају информације о систему као што су број текућих корисника, верзија оперативног система, количина слободне меморије или простора на диску и др.

Осим овога оперативни систем чува информације о свим пословима и процесима. Одговарајући системски позиви су **get process attributes** и **set proces attributes**.

### *2.2.2 Системски програми*

Следећи аспект модерних система су колекције системских програма. Уз сам мониторски оперативни систем многи системи пружају велику колекцију системских програма који решавају различите проблеме и пружају погоднију околину за развој и извршење програма.

Системски програми се могу поделити у неколико категорија:

- **Управљање датотекама.** Ови програми креирају, бришу, копирају, преименују, штампају, приказују, листају и уопште манипулишу датотекама и директоријумима.

- **Статусне информација.** Неки програми једноставно захтевају од оперативног система датум, време, количину слободне меморије и друге информације.
- **Модификација датотека.** У систему могу да постоје разни текст едитори који омогућавају креирање датотека и модификацију њиховог садржаја.
- **Подршка програмским језицима.** Оперативни системи често обезбеђују компилаторе, асемблере и интерпретаторе за различите програмске језике.
- **Пуњење и извршавање програма.** Када се програм преведе мора да се напуни у меморију ради извршења. Систем мора да обезбеди апсолутне и релокатибилне пуниоце, едиторе веза, пуниоце оверлеја, као и програме за пречишћавање (*debugging*).
- **Апликативни програми.** Оперативни системи садрже и програме за решавање разних проблема као што су компилатори компилатора, текст процесори, пакети за цртање, системи база података, пакети за статистичку анализу, итд.

Можда најважнији системски програм једног оперативног система је његов интерпретатор команди. Команде могу да се оперативном систему дају управљачким картицама (у ствари, команде су у интерактивном систему оно што су управљачке картице у пакетној обради). Алтернативни приступ је да се команде имплементирају посебним системским програмима.

## 2.3 Поглед оперативног система

Поглед на оперативни систем корисника дефинисан је углавном системским програмима, нарочитом интерпретатором команди. Са друге стране, програмер оперативног система има значајно другачији поглед. Тамо где корисник види средства која пружа оперативни систем, систем програмер види једино физичке ресурсе и уређаје и мора да их претвори у логичка средства која се нуде кориснику.

Оперативни систем је програм заснован на догађајима. Ако нема послова који се извршавају, У/И уређаја који се опслужују и корисника који на то одговарају, оперативни систем чека да се нешто деси. Догађаји се обично сигнализирају појавом прекида или трапова. Према томе, оперативни систем је заснован на прекидима.

Та природа оперативног система заснована на прекидима дефинише његову општу структуру. Када се прекид или трап јави, хардвер преноси управљање оперативном систему. Најпре оперативни систем памти стање CPU-а, памћењем његових регистара и програмског бројача. Затим одређује који се тип прекида јавио. Неколико различитих типова прекида се може јавити:

- Системски позив.
- Прекид У/И уређаја.
- Грешка у програму.

### *2.3.1 Системски позив*

Системски позиви оперативном систему се даље класификују према типу позива. Сваки позив има сопствени сегмент кôда којим се имплементира жељена акција. Опште узев можемо разматрати следеће типове:

- **Нормални завршетак**. Ако је системски позив начињен да би се програм који је се тренутно извршава завршио нормално, оперативни систем мора да пренесе управљање интерпретатору команди. Командни интерпретатор прихвата следећу команду и одређује која ће се следећа акција предузети.
- **Абнормални завршетак**. У случају откривања грешке током извршења или грешке у подацима, програм може одлучити да се заврши абнормално и у том случају може да захтева дамп меморије и поруку о грешци. После тога се позива интерпретатор команди. У интерактивним системима интерпретатор команди једноставно наставља са следећом командом; ту се подразумева да ће корисник издати одговарајућу команду као одговор на грешку. У пакетним системима интерпретатор команди обично прекида цео посао и наставља са следећим послом.
- **Сатусни захтеви**. Једна класа системских позива једноставно тражи од оперативног система одређене информације. По одређивању тражених информација управљање се враћа програму који се извршава.
- **Захтев за ресурсима**. Програм током извршења може да захтева додатне ресурсе како би могао да се настави. Ти додатни ресурси могу бити додатна количина меморије, јединице магнетних трака, приступ датотекама и др. Ако је ресурс расположив, он се може одобрити и управљање се враћа корисничком програму; у супротном, програм чека док тражени ресурс не постане расположив.

- **У/И захтев**. Највећи број захтева односи се на У/И, јер највећи број програма врши читање или упис уређаја или датотека током свог извршења.

### *2.3.2 Прекиди У/И уређаја*

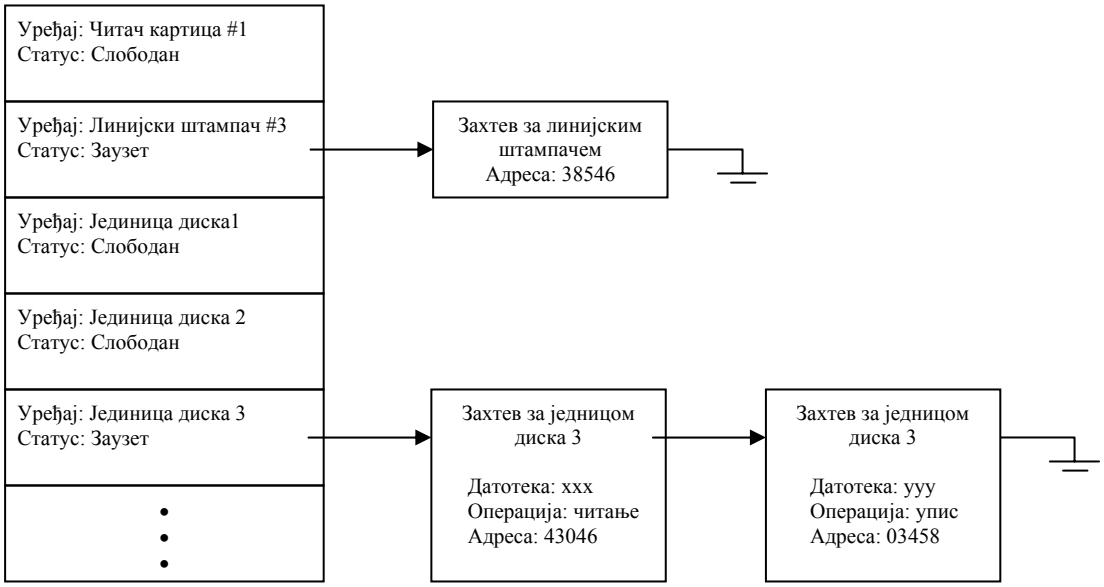
Велика класа догађаја које оперативни систем мора да обради су прекиди У/И уређаја. У/И уређај ће захтевати прекид када заврши неки У/И захтев. Ова ситуација се у општем случају јавља као резултат системског позива корисника којим се захтева нека У/И активност. Када су У/И активност започне, могуће су две врсте акција. У најједноставнијем случају започне се са захтеваним У/И и чека се док се он не комплетира па се потом враћа управљање корисничком програму. Друга могућност је да се управљање врати корисничком програму без чекања на завршетак У/И активности.

Чекање на комплетирање У/И активности може се постићи на један од два начина. Неки рачунари имају специјалну **wait** инструкцију која пасивизира CPU до следећег прекида. Машине које немају такву инструкцију морају да садрже следећу петљу

Petlja: **jmp** Petlja

Предност чекања на довршетак У/И активности је та што се највише један прекид прихвата истовремено, тако да се, када се јави У/И прекид, тачно зна који уређај је изазвао прекид.

Алтернатива је да се започне са У/И активношћу и да се одмах врати управљање корисничком програму. Нови системски позив је потребан да би се дозволило корисничком програму да чека на завршење У/И активности. Према томе, и даље нам је потребан системски позив за чекање. Такође морамо да пратимо више У/И захтева истовремено. У ту сврху оперативни систем користи табелу која садржи по једну ставку за сваку У/И уређај и која се назива *табела стања уређаја* (*device status table*). Свака ставка табеле указује на тип уређаја, његову адресу и његово стање (слика 2.1). Ако је уређај заузет претходним захтевом, тип захтева и други параметри памте се у ставци табеле за тај уређај. Како је могуће да се изда више захтева истом уређају, ови захтеви се уређују у виду листе.




2.1. Пример табеле стања уређаја.

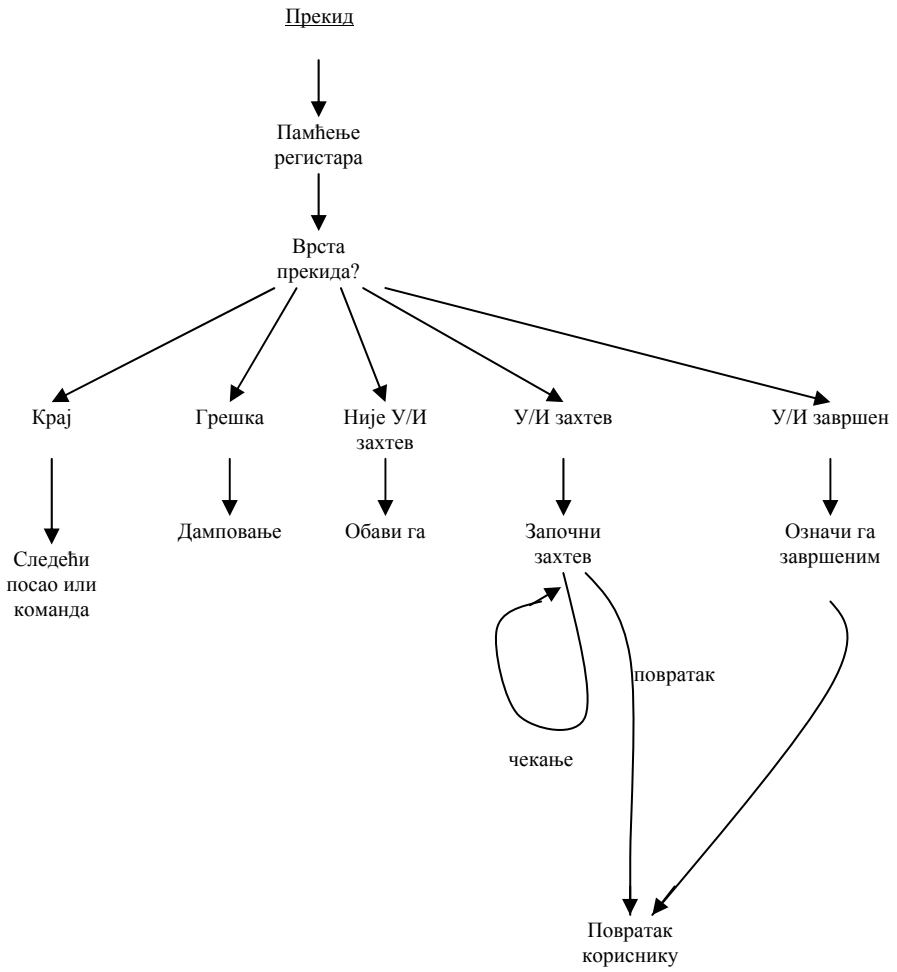
У/И уређај изазива прекид када му је потребно неко опслуживање. Када се прекид јави, најпре се одређује који је У/И уређај изазвао прекид. Број уређаја се, као индекс, користи да се приступи табели стања уређаја како би се одредило његово стање и модификовало стање уређаја као последица појаве прекида. За већину уређаја прекидом се јавља да је У/И активност комплетирана. Ако има додатних захтева који чекају на уређај они се тада опслужују. На крају се управљање враћа програму који је захтевао У/И уређај.

### *2.3.3 Грешке у програму*

Трећа врста прекида односи се на грешке у програму. Неки типови грешака у програму изазивају хардверске трапове. Трап преноси управљање преко вектора прекида на оперативни систем на сличан начин као и код прекида. Када се таква грешка јави оперативни систем мора да абнормално оконча програм. Ова ситуација се решава истим кôдом као и у случају корисничког захтева за абнормалним окончањем програма. Издаје се одговарајућа порука и може се извршити дамповање меморије.

### *2.3.4 Општи дијаграм тока оперативног система*

На слици 2.2 видимо општи дијаграм тока оперативног система. Наравно да стварни оперативни системи не морају да имају тачно такву структуру, али је већина њих слична ономе што имамо на слици. Можемо да идентификујемо неке заједничке делове оперативних система, као што су драјвери уређаја, руковаоци прекидима, скупови рутина системских позива и интерпретатор команди. Датотечни систем је још један од главних делова оперативног система.




Сл. 2.2. Општи дијаграм тока оперативног система.

# 3 Управљање процесима

Управљање процесима је основа оперативних система су мултипрограмирањем. Пребацивањем CPU-а са процеса на процес се повећава продуктивност рачунарског система. У овој глави ћемо се упознати са основним принципима планирања процеса и алгоритмима планирања процеса.

## 3.1 Концепт мултипрограмирања

Најважнији концепт у модерним оперативним системима је *мултипорграмирање*. Када у меморији постоји више програма истовремено, тада они могу да деле CPU. Овим се повећава општа ефикасност система и обавља више послова за краће време. Идеја мултипрограмирања је релативно проста. Један посао се извршава све до момента када му затреба нека У/И активност када чека да се његов У/И захтев задовољи. Уместо да CPU буде неупослен током испуњења У/И захтева, оперативни систем преноси управљање над CPU-ом на други посао.

Користи од мултипрограмирања одсликавају се у повећаној искоришћености CPU-а и већој укупној *пропусној моћи* послова. Пропусна моћ је износ извршеног посла у датом временском интервалу.

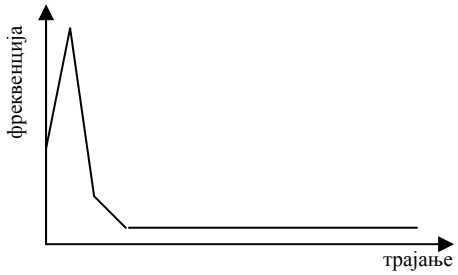
## 3.2 Концепт планирања послова

Планирање је основна функција оперативног система, јер се скоро сви рачунарски ресурси планирају пре употребе. CPU је један од основних рачунарских ресурса, па је његово планирање централно у пројектовању оперативног система.

### *3.2.1 Основни концепти*

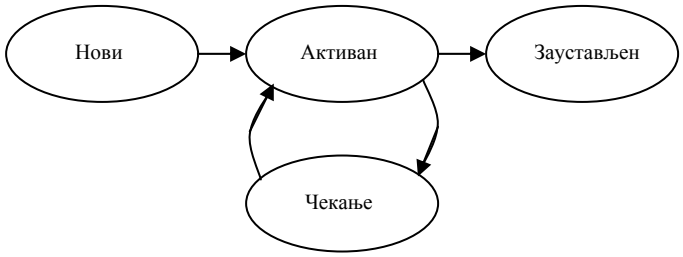
CPU извршава велики број послова или задатака. Док је његова главна активност извршавање корисничких програма, CPU се такође користи и за друге системске активности. Свака акција рачунарског система се иницира од стране CPU -а. CPU мора да одговори на трапове, програмске захтеве и У/И прекиде. Стални проблем код оперативних система је како називати све активности CPU -а. Пакетни системи извршвају *послове* док системи са дељењем процесорског времена имају *корисничке програме*. Чак и код једнокорисничких система корисник може бити у стању да извршава више програма истовремено: један интерактивни и више пакетних програма. Кад се извршава само један програм, оперативни систем може да обавља сопствене интерно програмиране активности. По много чему су све ове активности сличне па их зовемо једним заједничким именом: *процеси*. Процес је програм који се извршава.

Успешност планирања послова заснива се на особини процеса да њихово извршење представља циклус CPU активности и У/И активности. Процес непрестано прелази из једног у друго од ових стања (*cpu burst* и *I/O burst*). Трајања CPU активности су мерена, и мада се прилично разликују од процеса до процеса и од рачунара до рачунара, она показују хиперекспоненцијалну криву фреквенције (слика 3.1). Постоји велики број веома кратких CPU активности и мали број веома дугих.



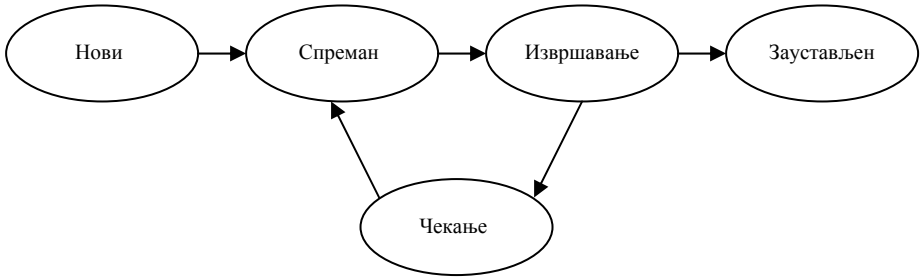

Сл. 3.1. Хистограм трајања CPU активности.

Током свог извршења програм мења стање. Стање процеса дефинисано је тренутном активношћу. Извршење процеса је алтернативна секвенца CPU и У/И активности, која почиње и завршава се CPU активношћу. Сваки од процеса може бити у једном од следећих стања: нови, активан, чека или заустављен (слика 3.2).




Сл. 3.2. Дијаграм стања процеса.

Како више процеса може да дели CPU, то активни процес може да чека да му се додели CPU или да се на њему извршава. Стање када процес чека да му се додели CPU назива се *спреман*, а када му се додели CPU онда је он у стању *извршавања*.




Сл. 3.3. Детаљнији дијаграм стања процеса.

Сваки процес је у оперативном систему представљен његовим *управљачким блоком процеса* (понекад се зове и управљачки блок задатка или посла). PCB је садржи:

- Стање процеса.
- Садржај програмског бројача.
- Садржај регистара CPU-а.
- Информације о управљању меморијом (базни и гранични регистри меморијске области или странична табела).
- Информације о налозима (износ CPU времена и стварног времена, временска ограничења, број налога, број процеса, итд.).
- Информације о стању У/И (У/И захтеви, У/И уређаји додељени процесу, листа отворених датотека, и др.)
- Информације о планирању CPU-а (приоритет процеса, поинтери на редове процеса и др.)

| | |
|---|---|
| поинтер | Стање процеса |
| Број процеса | |
| Програмски бројач | |
| Регистри | |
| Меморијске границе | |
| Листа отворених фајлова | |
| • • • | |

3.4. Управљачки блок процеса

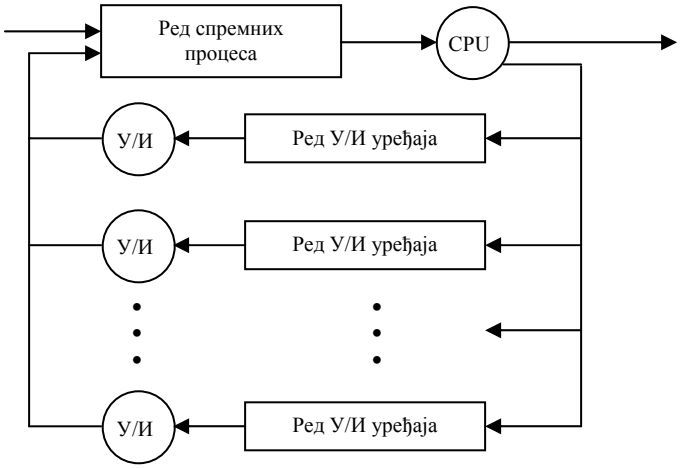
PCB се мора сместити у мониторску меморију. Овом меморијом се може управљати на неколико начина. Најједноставнији начин је да се унапред одреди максимални број процеса и статички додели довољно простора за сваки PCB. Динамичка додела меморије је обично боља, што се тиче флексибилности и бољег искоришћења меморије, али захтева сложеније алгоритме носећи и опасности од могућих грешака од погрешног додељивања меморије.

### 3.2.2 Редови процеса

Циљ мултипрограмирања је да се у сваком тренутку неки процес извршава како би се максимизирала искоришћеност CPU-а. Код једнопроцесорских машина највише један процес се у једном тренутку може извршавати. Ако у систему постоји више процеса, сви остали ће морати да чекају док се процесор не ослободи и додели неком од њих. Спремни процеси држе се у *редовима спремних процеса*. У општем случају, то је повезана листа чији су елементи PCB-ови процеса у реду. Редови спремних процеса не морају бити нужно FIFO структуре.

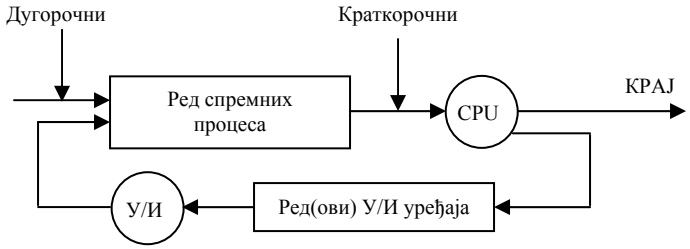
Осим редова спремних процеса, у систему постоје и редови уређаја у којима процеси чекају да им се додели неки У/И уређај.

Општа шема планирања послова је у облику дијаграма редова као на слици 3.5.




Сл. 3.5. Планирање CPU-а приказано помоћу дијаграма редова.

Пошто нас интересује планирање послова, тј. CPU-а, овај дијаграм можемо приказати као на слици 3.6, апстракујући вишеструке У/И редове за различите У/И уређаје.




Сл. 3.6. Поједностављени дијаграм редова.

### 3.2.3 Планери

Оперативни систем има више планера (*schedulers*). Два су основна: *дугорочни* и *краткорочни*.

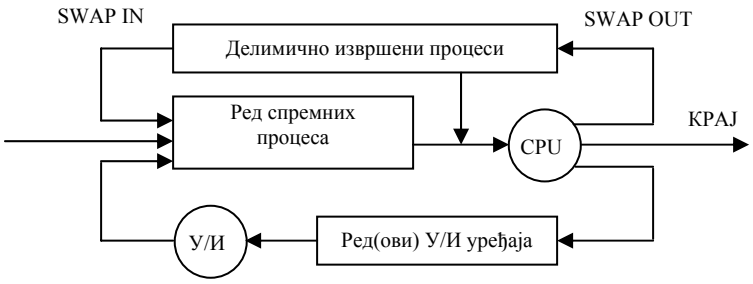
Дугорочни планер (или планер послова) одређује који ће се послови пропустити у систем ради обраде. У пакетним системима има више послова који су у систему него оних који могу да буду извршени. Послови који чекају копирају се на диск. Дугорочни планер бира послове из пула послова и пуни их у меморију ради извршења. Краткорочни планер (планер CPU-а) бира један од тих послова који су у меморији и спремни су за извршење и додељује му процесор.

Основна разлика ова два планера је у фреквенцији њиховог извршења. Краткорочни планер врло често бира процес који ће се извршавати и зато мора да буде веома брз. Дугорочни планер се извршава знатно ређе. Он управља *степеном мултипрограмирања* (бројем процеса у меморији). Уколико је тај степен стабилан, средња брзина доласка послова у систем је једнака средњој брзини изласка из система. Према томе, дугорочни планер се активира само када неки посао напушта систем. Због дугог интервала између два извршења, дугорочни планер има на располагању дуже време за избор посла. Такође је битно да дугорочни

планер тај избор изврши пажљиво и да направи добру мешавину послова оријентисаних ка CPU-у и оних оријентисаних ка У/И активностима.

У неким системима дугорочни планер може и да не постоји или може да буде минималан. На пример, системи са дељењем процесорског времена често немају дугорочни планер, већ се сваки нови процес шаље у меморију краткорочном планеру. Стабилност ових система зависи или од физичких ограничења или од разумног понашања корисника.

Неки системи, нарочито они са виртуелном меморијом или са дељењем процесорског времена, могу да имају још један, средњи ниво планирања. Тај *средњорочни планер* је приказан на слици 3.7




Сл. 3.7. Дијаграм редова са средњерочним планером.

Кључна идеја је да је понекад корисније уклонити процес из меморије како би се смањио степен мултипрограмирања. После извесног времена процеси се могу поново вратити у меморију. Ова шема назива се *swapping* (пребацивање), а изводи се са циљем да се побољша мешавина послова или да би се ослободила меморија у случају загушења.

Још једна компонента је укључена у функцију планирања и назива се *диспечер*. Диспечер је модул који у ствари даје управљење над CPU-ом процесу који је изабран од стране краткорочног планера. Ова функција обухвата пуњење регистара процеса, пребацивање у кориснички режим рада и скок на одговарајућу локацију у корисничком програму како би се он наставио. Ова компонента мора да буде што је могуће бржа.

## 3.3 Алгоритми планирања

Планери CPU-а одлучују о томе којем од спремних процеса треба доделити CPU. Постоји више различитих критеријума планирања.

### *3.3.1 Критеријуми перформанси*

Различити алгоритми имају различите особине и могу да фаворизују једну врсту процеса у односу на друге. Приликом избора одређеног алгоритма за одређену ситуацију морају се имати у виду особине алгоритама. Постоји више критеријума на основу којих се могу поредити алгоритми планирања. Ти критеријуми су следећи:

- **Искоришћеност CPU-а**. Како је CPU веома скуп уређај, нужно је држати га стално упосленим. Искоришћеност CPU-а може да варира од 0% до 100%. У реалним системима она се обично креће од 40% до 90%.
- **Пропусна моћ**. Ако је CPU заузет, онда он обавља неки посао. Пропусна моћ је мера која представља број послова који се могу завршити у јединици времена. За дуге послове она може да износи један посао по сату, док за кратке послове пропусна моћ може да буде десетак послова у секунди.
- **Време обиласка**. Ово време је време које протекне од момента пријављивања посла до његовог завршетка и представља суму времена које посао проведе у чекању да уђе у меморију, чекању у реду спремних процеса, времена током којег се извршавао на CPU-у и времена током којег је обављао У/И операције.
- **Време чекања**. Ово је укупно време које процес проведе у реду чекања спремних процеса.
- **Време одзива**. У интерактивним системима време обиласка није баш најбољи критеријум, јер процес често на почетку произведе неки излаз и наставља своје извршење израчунавајући нове резултате док се кориснику приказују претходни резултати. Друга мера, која се назива време одзива,

представља време које протекне од пријављивања посла док се не произведе први излаз програма, не рачунајући време потребно да са тај излаз заиста и прикаже на неком излазном уређају.

Када изаберемо критеријум на основу којег вршимо поређење алгоритама, обично желимо да добијемо оптималне перформансе, максимизирајући искоришћеност CPU-а и пропусну моћ а минимизирајући времена обиласка, чекања и одзива. У највећем броју случајева оптимизирају се средње вредости, али ће некада бити потребно да се оптимизирају минималне, односно максималне вредности ових мера.

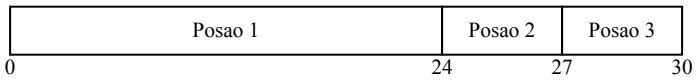
### *3.3.2 FCFS алгоритам*

Најједноставнији алгоритам планирања послова је FCFS (*Furst-Come-Furst-Served*). Процесор се најпре додељује оном процесу који га је први захтевао. Имплементација овог алгоритма се лако врши коришћењем FIFO редова. Када се процес убацује у ред спремних процеса његов PCB се ставља на крај реда. Када се процесор ослободи он се додељује првом процесу из реда.

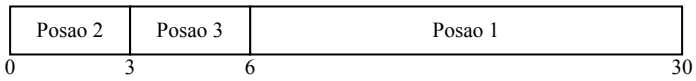
Међутим, перформансе овог алгоритма су веома слабе. Посматрајмо следећа три посла приказана у табели.

| Посао | Време извршавања |
|---|---|
| 1 | 24 |
| 2 | 3 |
| 3 | 3 |

Ако су ови послови пристигли у редоследу навођења и опслужују се на основу FCFS алгоритма, резултат овога видимо у следећој Гантовој карти.




Време обиласка за први посао је 24, за други 27 а за трећи 30. Средње време обиласка је 27. Ако претпоставимо да су послови пристигли у редоследу 2, 3, 1, добићемо следећу Гантову карту.




Сада је средње време обиласка једнако 13, што представља значајно смањење. Средње време обиласка за овај алгоритам у општем случају није минимално могуће, а може значајно да варира у зависности од конкретног случаја.

Поред овога, посматрајмо перформансе FCFS алгоритма у динамичкој ситуацији. Претпоставимо да имамо један посао оријентисан CPU активностима и више послова оријентисаних У/И активностима. Могућ је следећи сценарио. Посао оријентисан CPU активностима добиће CPU и држаће га доке се не изврши. За то време, сви остали послови ће завршити њихове У/И активности и биће у реду спремних процеса. Док чекају у реду, У/И уређају су беспослени. Када се посао који је држао CPU изврши и пређе на У/И активност, остали послови који имају кратке периоде CPU активности а дуге периоде У/И активности, ће се брзо извршити и вратити У/И активностима. Сада је CPU беспослен. Посао оријентисан CPU активностима се враћа у ред спремних процеса и додељује му се процесор и прича се понавља. Ова ситуација, где сви процеси чекају да један дугачак процес ослободи CPU, назива се *ефекат конвоја*. Резултат тог ефекта је слаба искоришћеност и CPU-а и У/И уређаја.

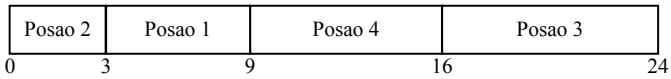
### *3.3.3 SJF алгоритам*

Различит приступ планирању послова присутан је код SJF (*Shortest-Job-Furst*) алгоритма. Код овог алгоритма се сваком послу придружује и податак о дужини његовог следећег CPU циклуса. Када је процесор слободан он се додељује оном послу код кога је та дужина најмања. Уколико постоје два или више послова исте дужине, као секундарни критеријум користи се FCFS.

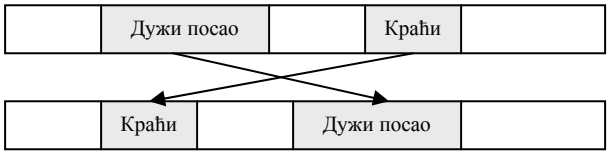
Размотримо следећи скуп послова приказан у табели.

| Посао | Време извршавања |
|---|---|
| 1 | 6 |
| 2 | 3 |
| 3 | 8 |
| 4 | 7 |

Употреба SJF алгоритма даје следећу Гантову карту.




Средње време обиласка које се овде добија је 13. Овај алгоритам је доказано оптималан, јер даје минимално средње време чекања за неки скуп послова. У доказу се показује да премештање краћег посла пре дужег смањује време чекања краћег посла у већој мери него што повећава време чекања дужег посла (види слику 3.8). Према томе, смањује се средње време чекања.




Сл. 3.8. Доказ да је SJF алгоритам оптималан.

Проблем код овог алгоритма је управо у одређивању дужине следећег захтева за CPU-ом. Код дугорочног планирања послова у пакетним системима можемо да користимо границу времена резервисаног за посао. Тако је корисник мотивисан да тачно процени трајање посла, јер што је мања граница коју зада то ће се његов посао брже извршити. Овај алгоритам се најчешће користи код дугорочног планирања. На жалост, и поред оптималности, не може се применити на краткорочно планирање CPU-а. Нема начина да се одреди трајање следеће CPU активности. Један приступ је да се покуша са апроксимацијом овог алгоритма. Иако не можемо да знамо трајање следеће CPU активности, можемо да предвидимо њену дужину.

Обично се за ово предвиђање користи *експоненцијална средина* претходних активности. Ако је $t_n$ трајање $n$-те CPU активности а $\tau_{n+1}$ предвиђено трајање наредне, онда је

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n, \quad 0 \le \alpha \le 1.$$

Вредност $t_n$ садржи најскорију познату информацију; $\tau_n$ садржи давну прошлост у себи. Параметар $\alpha$ управља релативним тежинама недавне и давне прошлости у предвиђању. Ако је $\alpha = 0$, tada је $\tau_{n+1} = \tau_n$, и недавна прошлост нема ефекта. За вредност параметра $\alpha = 1$, $\tau_{n+1} = t_n$, па је само најскорија CPU активност од важности. Најчешће је $\alpha = 1/2$, тако да и недавна и даља прошлост имају подједнаку тежину.

Да би разумели експоненцијалну средину развићемо претходну формулу замењујући вредности за $\tau_n$.

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \ldots + (1-\alpha)^j t_{n-j} + \ldots$$

Како су и $\alpha$ и (1-$\alpha$) мањи од 1, то је сваки следећи сабирак мање тежине од претходног.

### *3.3.4 Алгоритам са приоритетима*

SJF алгоритам је специјални случај општијег алгоритма са приоритетима. Приоритет се придружује сваком послу, а CPU се додељује послу са највишим приоритетом. Послови са истим приоритетом се опслужују по FCFS алгоритму. SJF је у ствари алгоритам са приоритетима где је приоритет $p$ једнак реципрочној вредности предвиђеног трајања $\tau$ активности CPU-а, тј. $p = 1/\tau$.

Приоритети се могу дефинисати интерно или екстерно. Интерно дефинисани приоритети користе неку мерљиву величину за израчунавање приоритета процеса. На пример, временске границе, меморијски захтеви, број отворених фајлова, као и однос средњег трајања У/И активности и средњег трајања CPU активности су неке од величина које се могу користити за израчунавање приоритета. Екстерни приоритети постављају се на основу критеријума који су спољашњи у односу на оперативни систем, као што су врста и износ плаћене накнаде за коришћење рачунара, важност наручиоца посла, као и слични фактори, углавном политичке природе.

Главни проблем са овим алгоритмом је појава *умирања од глади*. Може се десити да неки процес са ниским приоритетом, због честог пристизања послова вишег приоритета у систем, никада не добије прилику да се извршава. Могуће је да се такви процеси изврше тек када се смањи оптерећеност система (на пример, током викенда), или да им се не додели CPU све до престанка рада система. Постоје гласине да је 1973. године на МИТ-у, када је рачунар IBM 7094 требао да буде замењен, пронађен посао ниског приоритета који је пријављен на систем 1967. године и још увек се није извршио.

Друго решење овог проблема је *старење.* Техника старења предвиђа да се постепено повећава приоритет оних послова који дуго чекају у систему.

### *3.3.5 Алгоритми са прекидањем*

До сада описани алгоритми су *алгоритми без прекидања*. Када се код таквих алгоритама једном додели процесор неком процесу, онда га он задржава све док се не изврши до краја или док не обави У/И захтев. FCFS алгоритам је по својој природи алгоритам без прекидања, али се друга два могу модификовати да буду *алгоритми са прекидањем*.

SJF алгоритам може бити реализован и без прекидања и са прекидањем. Дилема настаје када нови посао стиже у ред спремних процеса док се претходни посао извршава. Нови посао може да има краћи CPU циклус него што је потребно за довршетак посла коме је већ додељен процесор. SJF са прекидањем ће прекинути посао који се тренутно извршава, док SJF без прекидања дозвољава да се доврши текући посао. Варијанта SJF алгоритма са прекидањем назива се SRTF (*Shortest-Remainig-Time-First*).

Планирање процеса на основу приоритета такође може бити са и без прекидања. Када посао пристиже у ред спремних процеса његов приоритет се пореди са приоритетом посла који се извршава. Ако је приоритет новог посла виши, текући посао ће се прекинути и процесор доделити новом послу. У варијанти без прекидања нови процес остаје у реду спремних процеса до завршетка текућег посла.

Алгоритми без прекидања (нарочито FCFS) нису погодни за системе са дељењем процесорског времена, где је неопходно да сваки корисник добије део процресорског времена у правилним интервалима.

### *3.3.6 Кружни алгоритам*

Кружни алгоритам (*round-robin*) је алгоритам пројектован специјално за системе са дељењем процесорског времена. Дефинише се мала једница времена (*квантум времена*) која се креће између 10 и 100 милисекунди. Ред спремних процеса реализован је као кружни ред. Планер процеса сваком процесу из реда додељује CPU на коришћење током интервала који је мањи или једнак квантуму времена.

Када се овај алгоритма имплементира, ред спремних процеса се реализује као FIFO ред процеса. Нови процес се додаје на крај реда. Планер процеса бира први процес из реда, поставља тајмер на време од једног квантума и додељује CPU процесу. Могуће је да се деси једна од две ствари. Процес може да има CPU циклус краћи од временског квантума. У том случају он сам ослобађа CPU издајући У/И захтев или терминирањем. Процесор се затим додељује наредном послу из реда спремних процеса.

У другом случају, тајмер ће, по истеку квантума времена, да изазове прекид. Регистри прекинутог процеса се памте у његовом PCB-у, и процес ставља на крај реда. Иза тога, планер процеса бира наредни процес из реда спремних процеса.

Перформансе овог алгоритма веома зависе од квантума времена. Ако тај квантум времена тежи бесконачности, понашање овог алгоритма тежи понашању FCFS алгоритма. Уколико је временски квантум веома мали (на пример 1 микросекунда), роунд-робин се назива *дељење процесора* и теоретски изгледа да сваки од *n* процеса у реду има сопствени процесор који је *n* пута спорији од стварног процесора. Овакво решење примењено је хардверски код рачунара CDC 6600 за имплементацију десет периферних процесора са једном ALU и управљачком јединицом и десет скупова регистара. Хардвер извршава једну инструкцију за један скуп регистара, затим иде на следећи.

Међутим, са становишта софтвера постоје други проблеми. На крају сваког квантума времена тајмер генерише прекид. Пребацивање процесора са једног на други процес захтева памћење садржаја свих регистара старог процеса и пуњење регистара за нови процес. Овај задатак се назива *пребацивање контекста*. Пребацивање контекста је са становишта извршавања процеса потпуно некористан посао. Трајање пребацивања контекста зависи од брзине меморије, броја регистара и постојања специјалних инструкција. Пожељно је да је квантум времена знатно дужи од времена потребног за пребацивање контекста.

Време обиласка такође зависи од квантума времена. Оно је краће ако већина послова своју CPU активност завршава за један квантум времена. Све нас ово наводи да је боље имати дужи квантум времена, али то онда конвергира ка слабом FCFC алгоритму. У пракси се најчешће тежи да квантум времена буде у 80% случајева буде дужи од трајања CPU активности.

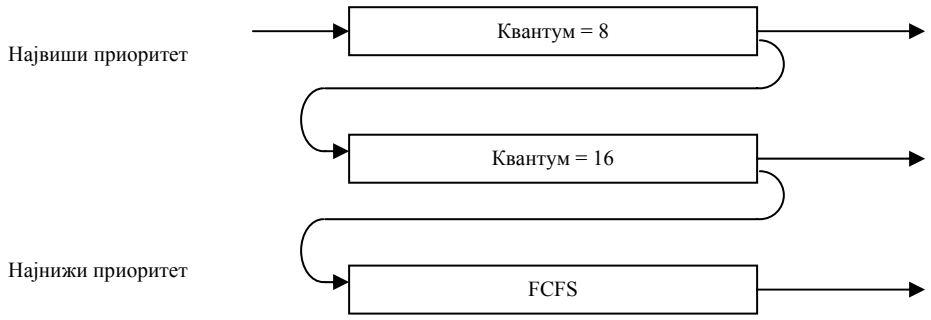
### *3.3.7 Вишенивовски редови*

Ова класа алгоритама планирања креирана је за ситуације када се послови могу лако класификовати у различите групе. На пример, уобичајена је подела на послове у првом плану (интерактивни) и послове у позадини (пакетни). Ова два типа послова имају сасвим различите захтеве у погледу времена одзива и могу да имају различите алгоритме планирања, а уз то интерактивни послови могу имати екстерно додељен виши приоритет од послова у позадини.

Код овог алгоритма је ред спремних процеса подељен у више редова. Послови који пристижу сврставају се у одговарајући ред. Сваки ред има сопствени алгоритам планирања, а између самих редова постоји фиксиран приоритет. На тај начин посао који је први у свом реду може добити процесор на коришћење искључиво ако је ред вишег приоритета празан.

### *3.3.8 Вишенивовски редови са повратном спрегом*

Нормално се код вишенивовских редова чекања послови додељују редовима по њиховом уласку у систем. Послови не могу прелазити из реда у ред. Код вишенивовских редова са повратном спрегом дозвољено је кретање послова између редова. Идеја је да се издвоје послови са различитим карактеристикама CPU активности. Ако посао користи превише CPU времена премешта се у ред са нижим приоритетом. Овај алгоритам оставља послове оријентисане У/И активностима и интерактивне послове у редовима вишег приоритета (види слику 3.9).




Сл. 3.9. Вишенивовски ред са повратном спрегом.

## 3.4 Процеси и нити

Концепт процеса подразумева две карактеристике:

- То је јединица која има власништво над ресурсима.
- То је ентитет над којим оперативни систем врши планирање и дистрибуцију.

У већини оперативних система ове две карактеристике јесу суштина онога што се назива процес. Међутим, код неких оперативних система ове две карактеристике се могу третирати одвојено од стране оперативног система. У многим оперативним системима, нарочито скорије развијеним, јединица дистрибуције (доделе процесору ради извршења) се означава као *нит* или процес мале тежине, док се јединица власништва над ресурсима назива *процес* или задатак.

### *3.4.1 Вишеструке нити у оквиру једног процеса*

Најзначајнија употреба концепта нити је она где више нити може постојати у једном процесу. Приближан приступ коришћен је код оперативног система MVS, а најексплицитније је употребљен код оперативних система Windows NT, OS/2, једној од Sun-ових верзија Unix-а, као и код оперативног система Mach.

Пример апликације која може да користи нити је фајл сервер у некој локалној мрежи. Како нови захтеви за датотекама пристижу, програм за управљање датотека може да креира нове нити. Како сервер опслужује многе захтеве, нити се могу креирати и укидати током кратког периода. Ако је сервер вишепроцесорска машина, нити у оквиру истог задатка могу се извршавати симултано на различитим процесорима. И код једнопроцесорских машина концепт нити је користан због поједностављења структуре програма који логички обавља више функција.

Нити омогућују већу ефикасност и код комуникације различитих програма у извршењу. У већини оперативних система комуникација између независних процеса захтева интервенцију кернала због заштите и комуникационих механизама. Како нити у оквиру истог задатке деле меморију и датотеке, оне могу да комуницирају међусобно без посредовања кернела.

## 3.5 Windows NT

У наредном одељку позабавићемо се једним актуелним примером оперативног система, Ради се о Microsoft-овом Windows NT оперативном систему. Иако је сада актуелнија линија Windows XP, ради се о томе да је Microsoft у Windows XP систему објединио линије Windows 9x и Windows NT.

### *3.5.1 Историјски развој*

Далеки предак оперативног система Windows NT је оперативни систем који је Microsoft развио за први IBM-ов персонални рачунар и који је назван MS-DOS или PC-DOS. Прва верзија DOS 1.0 појавила се у августу 1981. године. Састојала се од 4000 линија изворног кода на асемблерском језику и радила је са 8kB меморије на микропроцесору Intel 8086.

Када је IBM развио персонални рачунар заснован на хард диску, тј. PC XT, Microsoft је развио DOS 2.0 и објавио га 1983. године. Тај оперативни систем је садржао подршку за хард диск и омогућио хијерархију директоријума. До тада је диск могао да садржи само један директоријум и подржавао је максималан број од 64 датотеке. Тако нешто је било довољно у доба *floppy* дискова, али је представљало превелико ограничење за хард диск. Нова верзија је омогућила да директоријуми садрже и поддиректоријуме и датотеке. Такође је обезбеђен богатији скуп интерних команди оперативног система које су обављале неке од функција екстерних програма. Међу могућностима које су додате било је и неколико ствари у стилу Unix-а, као што је редирекција У/И и штампање у позадини. Резидентни део програма нарастао је на 24kB.

Када је IBM објавио PC AT 1984. године, Microsoft је увео DOS 3.0. AT је садржао микропроцесор Intel 80286 који је омогућавао проширено адресирање и функцију заштите меморије, а што није био случај у верзији DOS 2.0. Да би остао компатибилан са претходним верзијама, оперативни систем је користио 80286 као "брзи 8086".

Оперативни систем је обезбедио подршку за нову тастатуру и периферале за хард дискове. Меморијски захтеви су нарасли на 36kB. Било је неколико значајних побољшања верзије 3.0. DOS 3.1 објављен 1984. године је садржао подршку за умрежавање PC-а. Величина резидентног дела није се променила што је постигнуто увећавањем дела оперативног система који је могао да се пребацује. DOS 3.3 је објављен 1987. године и садржао је подршку за нову линију IBM машина PS/2. Ова верзија  није искористила могућности процесора PS/2 које су обезбеђене чиповима 80286 и 80386. Резидентни део је нарастао на минималних 46kB уз додатне захтеве ако су нека од опционих проширења изабрана.

До тог тренутка DOS је користио околину која је далеко премашивала његове могућности. Појављивањем процесора 80486 а потом и Intel Pentium омогућило је снагу и карактеристике који једноставни DOS никако није могао да искористи. У међувремену, од раних 80-тих Microsoft је започео развој графичког корисничког интерфејса (GUI – Graphic User Interface) који би био постављен између корисника и DOS-а.

Након неуспешног покушаја Microsoft-а да заједно са IBM-ом развије оперативни систем следеће генерације који би искористио моћ нових микропроцесора и који би задржао једноставност коришћења Windows-а, Microsoft је наставио да самостално развија Windows NT.

Windows NT са корисничке тачке гледишта изгледа слично као Windows 3.1, али је заснован на радикално другачијем концепту. Windows NT користи снагу савремених микропроцесора и обезбеђује потпуни *multitasking* (обављање више задатака истовремено) у једнокорисничком окружењу.

### *3.5.2 Multitasking у једнокорисничком окружењу*

Windows NT је можда најважнији пример онога што је нови талас у оперативним системима за персоналне рачунаре (други примери су OS/2 и Macintosh System 7). Он је заснован на потреби да се искористи огромна моћ савремених 32-битних микропроцесора који конкуришу *mainframe* и минирачунарима од пре само неколико година и што се тиче брзине и сложености хардвера а такође и меморијског капацитета.

Једна од најзначајнијих карактеристика ових нових оперативних система је та да, иако су намењени да подрже једног интерактивног корисника, то су у ствари *multitasking* оперативни системи. Две главне новине су покренуле потребу за *multitasking*-ом. Прво, са повећаном брзином и меморијским капацитетом

микропроцесора, уз подршку виртуелној меморији, апликације су постале сложеније и међузависне. Корисник може да покрене процесор текста, програм за цртање и апликацију за унакрсне табеле истовремено како би произвео неки документ. Без *multitasking*-а, уколико корисник жели да креира цртеж и да га затим убаци у неки документ, он би морао да направи више следећих корака:

1. Отварање програма за цртање.
2. Креирање цртежа и памћење цртежа у датотеку или *clipboard*.
3. Затварање програма за цртање.
4. Отварање програма за обраду текста.
5. Убацивање цртежа на одговарајућу локацију.

За сваку измену би сличан низ корака морао да буде извршен. То је веома брзо постало заморно. Како су услуге и могућности расположиве кориснику постале значајније и разноликије, околина која је могла да извршава само један задатак је постајала све незгоднија за употребу. У *multitasking* околини корисник отвара сваку од потребних апликација и оставља их отвореним током рада. Подаци се могу лако преносити између апликација. За сваку апликацију постоји један или више отворених прозора а графички интерфејс, уз уређај као што је миш, дозвољава једноставно кретање по тој околини.

Други разлог за *multitasking* је пораст броја клијент/сервер апликација. Овде се PC као радна станица (клијент) и хост (сервер) користе заједно да би се извршила одређена апликација. Оба су повезана а сваком од њих додељен је онај део посла који одговара његовим могућностима. Клијент/сервер се може постићи у LAN-у PC-а и сервера или помоћу везе између корисниковог система и великог хоста као што је *mainframe* рачунар. Апликација може да обухвати један или више персоналниг рачунара и један или више сервера. Да би се обезбедила захтевана способност одзива, оперативни систем мора да подржава софостицирани хардвер за комуникацију у реалном времену и одговарајуће протоколе.

### *3.5.3 Опис оперативног система*

Много елемената је утицало на дизајн оперативног система Windows NT. Обезбеђен је исти тип GUI као и код претходних верзија Windows-а. Интерна структура инспирисана је оперативним системом Mach који је, са друге стране, инспирисан Unix-ом.

На слици 3.10 илустрована је општа структура оперативног система Windows NT.

Високо модуларна структура овог оперативног система омогућује му импресивну флексибилност. NT може да се извршава на различитим хардверским платформама и да подржава апликације које су писане за више других оперативних система.

Као и готово сви други системи, NT раздваја апликацијама оријентисан софтвер од софтвера оперативног система који ради у привилегованом или кернел начину рада. Овај софтвер има приступ системским подацима и хардверу. Остали програми се извршавају у корисничком начину рада и имају ограничен приступ системским подацима. Софтвер који ради у кернел начину рада назива се и NT егзекутива.

Без обзира да ли се извршава на једнопроцесорској или вишепроцесорској машини, CISC или RISC систему, већи део оперативног система има исти поглед на одговарајући хардвер. Да би се постигла ова независност оперативни систем се састоји од четири нивоа:

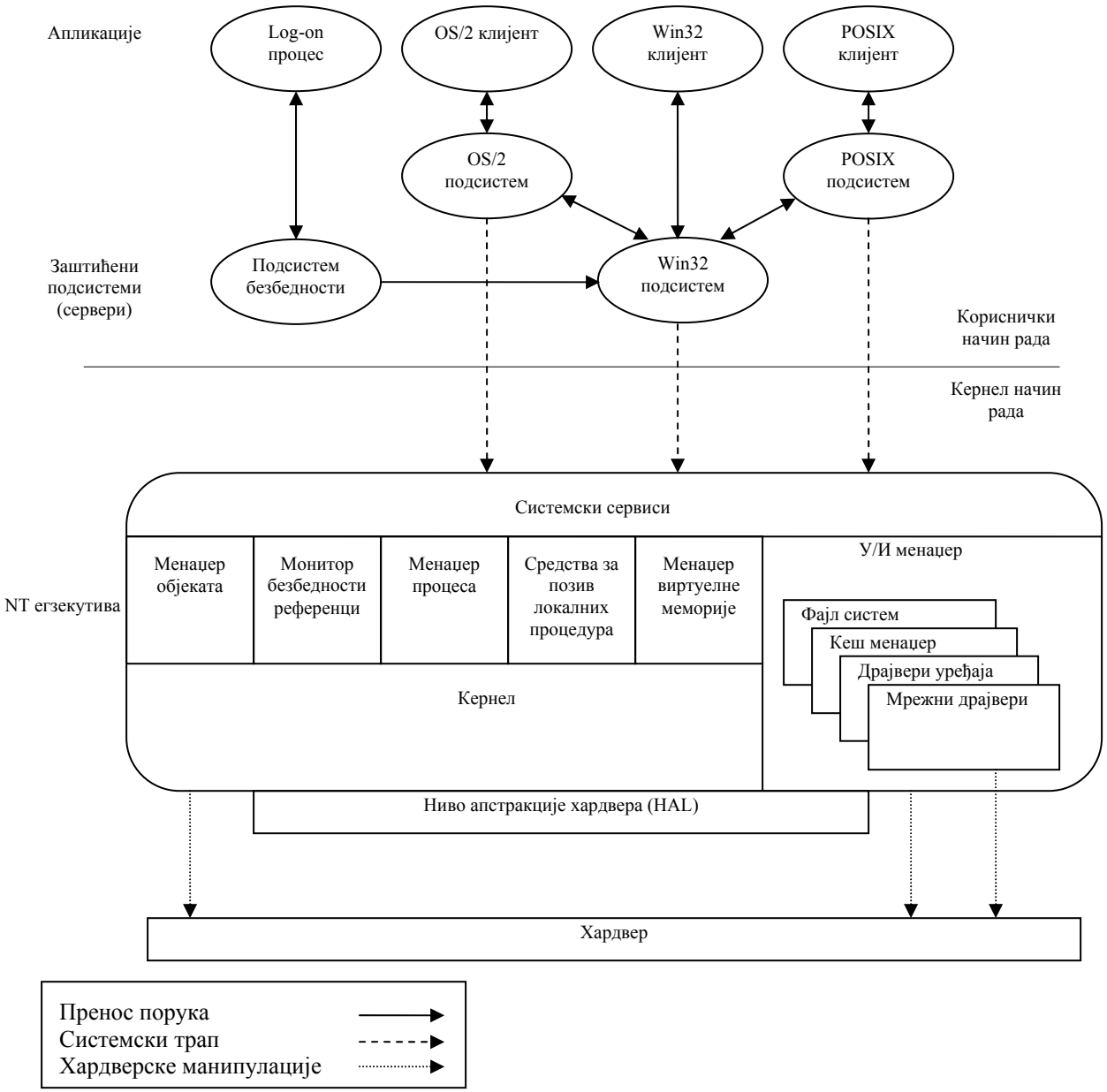
- Ниво апстракције хардвера (HAL): Врши пресликавање хардверских команди и одзива који су јединствени за одређену платформу, као што је Intel 486 или Pentium, Motorola PowerPC или DEC Alpha процесор. HAL чини да интерна магистрала, DMA контролер, контролер прекида, системски тајмер и меморијски модули сваке машине изгледају кернелу на исти начин. Осим тога обезбеђује и подршку неопходну за симетрично мултипроцесирање.
- Кернел: Састоји се од најкоришћенијих и најосновнијих компоненти оперативног система. Кернел управља планирањем процеса, пребацивањем контекста, руковањем изузецима и прекидима и синхронизацијом код вишепроцесорских машина.
- Подсистеми: Обухватају различите модуле за специфичне функције које користе основне услуге кернела.
- Системски сервиси: Пружају интерфејс са софтвером у корисничком начину рада.

Један од подсистема, У/И менаџер, прескаче HAL и директно је повезан на хардвер како би се постигла ефикасност и пропусна моћ која се захтева од У/И операција.

Моћ Windowsa NT потиче од његове способности да подржи апликације писане за друге системе. Начин на који се то постиже је преко заштићених подсистема. Заштићени подсистеми су они делови NT-а који интерагују са крајњим корисником. Заштићени подсистем пружа графички или командни кориснички интерфејс кој дефинише изглед и осећај који корисник има при коришћењу оперативног система. Осим тога, сваки заштићени подсистем пружа интерфејс за програмирање апликација (API – Application Programming

Interface) за одређену радну околину која се може неизмењена извршавати на NT-у, јер је интерфејс оперативног система који оне виде исти као и онај за који су писане.

Начин на који су егзекутива, заштићени подсистеми и апликације структуирани за NT је коришћењем клијент/сервер модела. Клијент/сервер архитектура је уобичајени модел за дистрибуирану обраду, али се може усвојити за интерну употребу у једном систему као што је NT.




Сл. 3.10. Структура Windowsa NT.

Сваки сервер се имплементира као један или више процеса. Сваки процес чека захтев од клијента за једном или више услуга. Клијент, који може бити апликациони програм или неки други модул оперативног система, захтева услугу слањем поруке. Порука се преко егзекутиве усмерава ка одговарајућем серверу. Сервер обавља тражену операцију и враћа резултате или статусне информације помоћу друге поруке, која се, опет преко егзекутиве, враћа назад ка клијенту.

Предности клијент/сервер архитектуре су:

- Поједностављени основни оперативни систем, тј. NT егзекутива. Егзекутива не обезбеђује API тако да је могуће конструисати више API-ја без конфликата или дуплицирања у егзекутиви. Нови API се лако додају.
- Побољшана је поузданост. Сваки сервер се извршава на посебном процесу, са сопственом меморијском партицијом, заштићеном од других сервера. Сервери не могу директно да приступају хардверу или да модификују меморију у којој је смештена егзекутива. Један сервер може да откаже а да то не изазове или грешке у остатку оперативног система.
- Пружена је природна основа за дистрибуирану обраду. Локални сервери могу да шаљу податке удаљеним серверима у име локалних клијентских апликација. Клијенти не морају да знају да ли се захтеви опслужују локално или удаљено.

### *3.5.4 Нити*

Важан аспект NT-а је подршка нитима. Нити садрже неке од функција које су традиционално придружене процесима, али се могу уочити разлике.

*Нит* је јединица посла која се може делегирати. Она се извршава секвенцијално и може се прекинути да би се процесор окренуо другој нити. Са тачке гледишта планирања, овај концепт је еквивалентан појму процеса код већине других оперативних система.

*Процес* је колекција једне или више нити и придружених системских ресурса. Одговара блиско концепту програма у извршењу. Разбијањем једне апликације у више нити програмер има велику контролу над модуларношћу апликација и синхронизацијом догађаја повезаних са апликацијом. Са тачке гледишта посла или апликације, овај концепт је еквивалентан процесу код већине других оперативних система.

### *3.5.5 Симетрично мултипроцесирање*

Код симетричног мултипроцесирања сваки процес или нит се може доделити било ком од процесора. То обухвата и процесе и нити оперативног система. Кључни захтев у савременим оперативним системима је способност да се искористи симетрично мултипроцесирање.

Карактеристике NT-а које подржавају употребу SMP су:

- Рутине оперативног система могу се извршавати на било ком расположивом процесору и различите рутине могу се извршавати на различитим процесорима.
- NT подржава коришћење вишеструких нити код извршавања једног процеса и оне могу да се симултано извршавају на различитим процесорима.
- Процеси сервери могу да користе вишеструке нити ради обраде захтева више корисника истовремено.
- NT нуди одговарајуће механизме дељења података и ресурса између процеса и могућност флексибилне комуникације међу процесима.

### *3.5.6 Windows NT објекти*

Windows NT је повукао многе карактеристике објектно-оријентисаног пројектовања. Овај приступ олакшава дељење ресурса и података међу процесима и заштиту ресурса од неовлашћеног приступа. Кључни објектно-оријентисани концепти су:

- *Енкапсулација*: Објекат се састоји од једног или више података који се називају *атрибутима* или *својствима* и од једне или више процедура које се извршавају над тим подацима и које се називају *сервисима*. Једини начин да се приступи подацима неког објекта је да се позове неки од сервиса објеката. На тај начин се објекти штите од неовлашћеног приступа и некоректне употребе.
- *Класе и примерци објеката*: Класа објеката је шаблон који набраја атрибуте и сервисе неког објекта и дефинише одређене карактеристике објекта. Оперативни систем може да креира поједине примерке класе објеката када је то потребно.

Нису сви ентитети у Windows-у NT објекти. Објекти се користе у случајевима када су подаци отворени за приступ у корисничком начину рада или када је приступ подацима дељив или ограничен. Међу ентитетима који се представљају објектима су датотеке, процеси, нити, семафори, тајмери и прозори. NT креира и управља свим типовима објеката на униформан начин, преко тела кôда у кернелу које се назива менаџер објеката.

Један објекат (нпр. процес) може да референцира други објекат (нпр. датотеку) отварањем хендла за њега (*handle* – ручица). У суштини, хендл је поинтер на референцирани објекат. Код NT-а објекти могу бити именовани или неименовани. Када процес креира неименовани објекат тада менаџер објеката враћа хендл тај

објекат и то је једини начин да се приступи том објекту. Именовани објекти имају своје име које други процеси могу да користе за прављење хендла на неки од њих.

Важна разлика између именованих и неименованих објеката је у томе што именовани објекти увек имају придружену информацију о безбедности дату у форми токена приступа. Та информација се може користити за ограничење приступа том објекту.

Windows NT није у потпуности објектно оријентисан оперативни систем и није написан на неком од објектно оријентисаних језика. Структуре података које се у потпуности налазе у некој од компоненета егзекутиве нису представљене као објекти. NT не подржава неке од уобичајених објектно-оријентисаних могућности, као што су наслеђивање или полиморфизам. Ипак, NT илуструје моћ објектно-оријентисане технологије и представља растући тренд ка употреби ове технологије у оперативним системима.

## 3.6 Опис процеса и управљање процесима код Windows-а NT

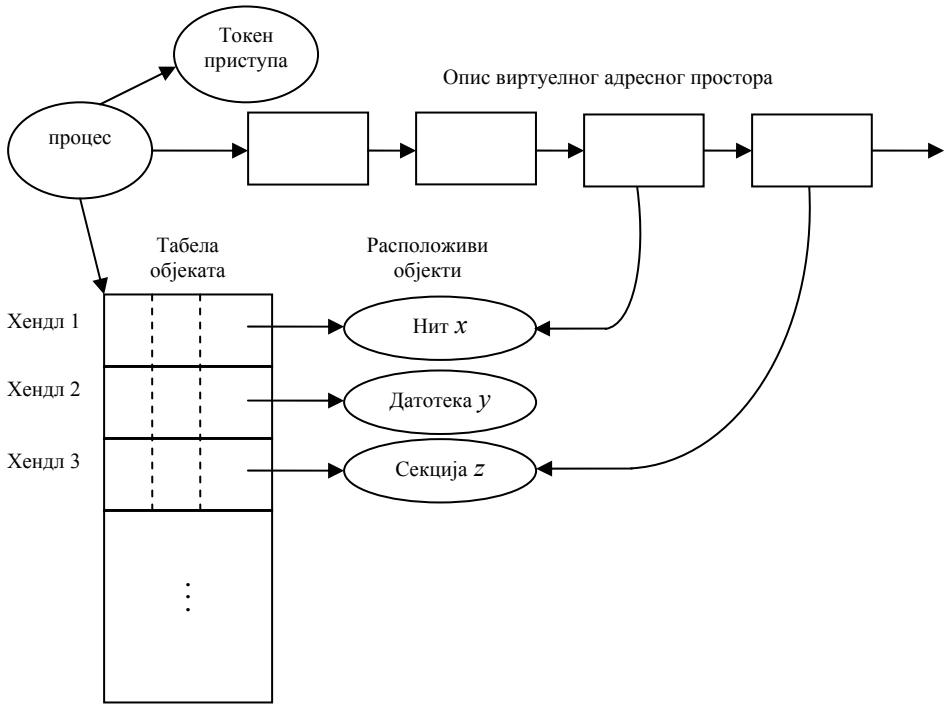
Дизајн оперативног система Windows NT заснован је на потреби да пружи подршку различитим системским околинама. Процеси који су подржани од различитих оперативних система разликују се по више карактеристика, међу којима су:

- Начин именовања процеса.
- Постојање нити унутар процеса.
- Начин представљања процеса.
- Начин заштите ресурса додељених процесу.
- Механизам комуникације и синхронизације процеса.
- Начин на који се процеси односе једни према другима.

Из ових разлога су основна структура процеса и услуге које пружа кернел релативно једноставни и опште намене тако да је омогућено сваком од подсистема да емулира одређену структуру и функцију процеса. Важне карактеристике NT процеса су:

- NT процеси се имплементирају као објекти.
- Извршни процес може да садржи једну или више нити.
- И објекти процеси и објекти нити имају уграђене синхронизационе могућности.
- NT кернел не одржава било какве релације између процеса које је креирао па чак ни релацију родитељ-дете.

Слика 3.11 илуструје начин на који је процес повезан са ресурсима којима управља, тј. које користи.




Сл. 3.11 NT процеси и њихови ресурси.

Токен приступа одређује када процес може да промени своје атрибуте. У овом случају процес нема отворен хендл за свој токен приступа. Ако процес покуша да отвори такав хендл, систем безбедности одлучује да ли је такав покушај дозвољен, тј. да ли процес може да измени своје атрибуте.

Са процесом је повезан и низ блокова који дефинише виртуелни адресни простор тренутно додељен том процесу. Процес не може директно да модификује ове структуре већ мора у томе да се ослони на менаџера виртуелне меморије који пружа услуге додле меморије процесу.

Најзад, процес садржи табелу објеката са хендловима на друге објекте познате том процесу. За сваку нит садржану у објекту постоји по један хендл. На слици 3.11 је приказана само једна нит. Осим тога процес има приступ објекту датотеке и објекту секције који дефинише део дељиве меморије.

### *3.6.1 Објекти процеса и објекти нити*

Објектно-оријентисана структура NT-а олакшава развој процеса опште намене. NT користи две врсте објеката који се односе на процес: процесе и нити. Процес је ентитет који одговара корисничком послу или апликацији која поседује ресурсе као што су меморија и отворене датотеке.

Нит је јединица посла која се може доделити процесору на извршење (*dispatch*) и која се извршава секвенцијално, а могуће је прекинути њено извршење како би процесор прешао на неку другу нит.

Сваки NT процес се представља објектом чија је општа структура приказана на слици 3.12.

Сл. 3.12. Објекат процеса.

Сваки процес се дефинише извесним бројем атрибута и енкапсулира известан број акција или сервиса које може да изврши. Процес извршава одређени сервис по пријему одговарајуће поруке. Једини начин да се отпочне са извршавањем сервиса је помоћу порука упућених објекту процеса који пружа тражену услугу.

Када NT креира нови процес он користи класу објеката (другим речима – тип), дефинисану за NT процес, као шаблон на основу кога се генерише нови примерак објекта. У време креирања својствима (атрибутима) се додељују вредности. Табела 3.1 даје кратку дефиницију сваког од атрибута објекта за неки објекат процеса.

Један NT процес мора да садржи најмање једну нит коју извршава. Та нит може да креира друге нити. У вишепроцесорском систему више нити једног процеса могу се извршавати симултано.

Слика 3.13. описује структуру објекта за један објекат нити а табела 3.2 описује атрибуте објекта нити. Запазимо да неки од атрибута нити подсећају на атрибуте процеса. У тим случајевима се вредност атрибута нити добија на основу вредности атрибута процеса. На пример, *афинитет нити према процесорима*

представља скуп процесора у вишепроцесорском систему који могу да изврше ту нит; овај скуп се поклапа са скупом *афинитета процеса према процесорима* или је његов подскуп.

Табела 3.1. Атрибути објекта процеса код Windows NT.

| | |
|---|---|
| Идентификација процеса | Јединствена вредност која идентификује процес у оперативном систему. |
| Токен приступа | Извршни објекат који садржи безбедносне информације о кориснику прикљученом на систем који је представљен дотичним процесом. |
| Основни приоритет | Основни приоритет извршења за нити процеса. |
| Подразумевани афинитет према процесорима | Подразумевани скуп процесора на којим се нити процеса могу извршавати. |
| Границе квота | Максимални износ страничени и нестраничене системске меморије, страничени простор датотека и процесорско време које кориснички процес може да користи. |
| Време извршавања | Укупан износ времена током кога су се нити процеса извршиле. |
| У/И бројачи | Променљиве које бележе број и врсту У/И операција које су нити процеса извршавале. |
| Бројачи рада виртуелне меморије | Променљиве које бележе број и врсту операција виртуелне меморије које су нити процеса извршавале. |
| Портови за обраду изузетака и пречишћавање | Канали међупроцесорске комуникације којима менаџер процеса шаље поруку када нека од нити процеса изазове изузетак. |
| Стање на излазу | Разлог за терминирање процеса. |

Сл. 3.13. Објекат нити.

Запазимо да је један од атрибута објекта нити њен контекст. Ова информација омогућава нитима да буду суспендоване и обновљене. Даље, могуће је мењати понашање нити мењањем њеног контекста када је суспендован.

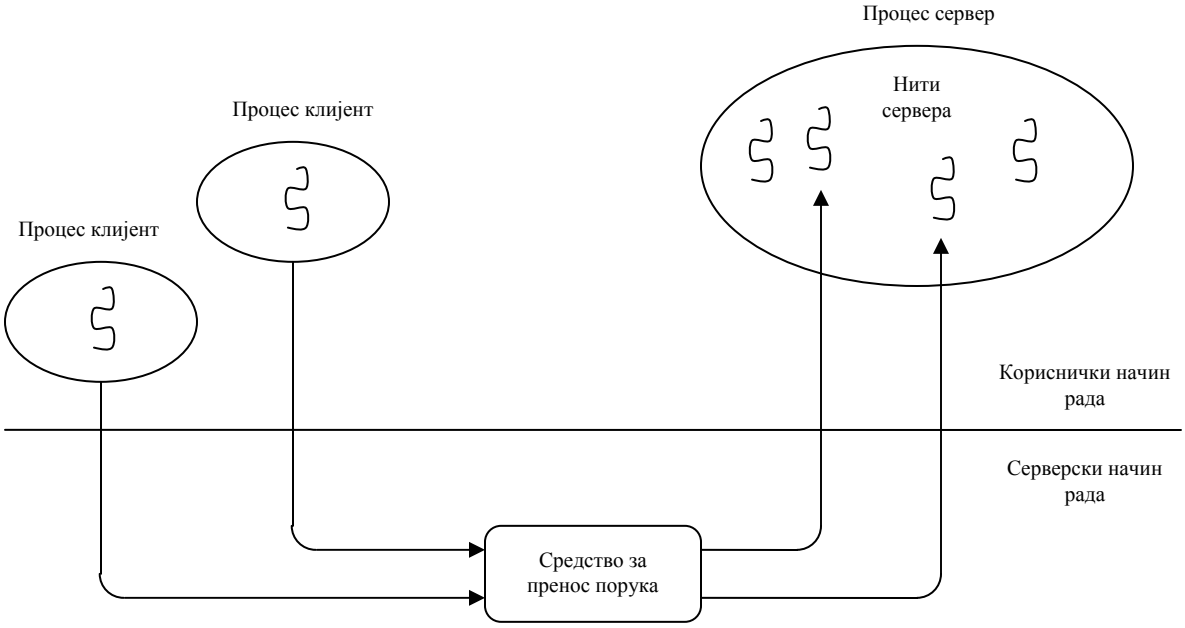
Табела 3.2. Атрибути објекта нити код Windows NT.

| | |
|---|---|
| Идентификација клијента | Јединствена вредност која идентификује нит када се обраћа серверу. |
| Контекст нити | Скуп регистарских вредности и других непостојаних података који дефинише статус извршења нити. |
| Динамички приоритет | Приоритет извршења нити у произвољном моменту. |
| Основни приоритет | Доња граница динамичког приоритета нити. |
| Афинитет нити према процесорима | Скуп процесора на којиме се нит може извршавати и представља подскуп или све процесоре из скупа афинитета процеса. |
| Време извршавања нити | Укупан износ времена током кога су се нити процеса извршиле како у корисничком тако и у кернел начину рада. |
| Стање упозорења | Маркер који указује да ли нит треба да изврши асинхрони позив процедуре. |
| Број суспензија | Број суспензија извршења нити без обнављања. |
| Токен имитирања | Привремени токен приступа који омогућује нити да обавља операцију у име другог процеса (користи га подсистем). |
| Порт за терминирање | Канал међупроцесорске комуникације коме менаџер процеса шаље поруку када се нит терминира (користи га подсистем). |
| Стање нити на излазу | Разлог за терминирање нити. |

### *3.6.2 Multithreding*

NT подржава конкуренцију између процеса, јер се нити у различитим процесима могу извршавати конкурентно. Штавише, више нити у оквиру истог процеса могу бити додељене различитим процесорима и на тај начин извршене конкурентно. Процес са више нити постиже конкуренцију и без, у овом случају непотребног, коришћења више процеса. Нити унутар истог процеса могу да измењују информације преко дељиве меморије и имају приступ дељивим ресурсима процеса.

Објектно-оријентисан процес са више нити представаља ефикасан начин да се омогуће серверске апликације. Слика 3.14 описује општи концепт. Један серверски процес може да опслужи више клијената. Сваки захтев клијента изазива креирање нове нити унутар сервера.




Сл. 3.14. Windows NT сервер са више нити.

### *3.6.3 Подршка подсистемима оперативног система*

Процеси опште намене и нити морају да подрже структуру процеса и нити различитих клијената оперативног система.

Одговорност је сваког подсистема да искористи могућности NT процеса и нити да емулирају процесе и нити одговарајућег система.

Слику о томе како је обезбеђена подршка процесима и нитима можемо да добијемо ако извршимо увид у креирање процеса. Креирање процеса почиње захтевом за новим процесом који долази од стране апликације. Захтев за креирањем процеса апликације издају одговарајућем заштићеном подсистему. Подсистем, са своје стране, издаје захтев за новим процесом NT егзекутиви. Када NT креира процес он не креира аутоматски и нит. У случају Win32 или OS/2, нови процес се увек креира заједно са нити. Према томе, за те оперативне системе, подсистем позива NT менаџер процеса наново да би креирао нит за нови процес, примајући назад хендл од NT-а. Одговарајуће информације о нити и процесу се враћају апликацији. У случају 16-битног Windows-а или POSIX-а нити нису подржане. Према томе, за те оперативне системе подсистем обезбеђује нит за нови процес захтевајући га од NT-а, али се апликацији враћа само информација о процесу.

Када се нови процес креира код Win32 или OS/2, тај нови процес наслеђује многе од атрибута од процеса који га је креирао. Међутим, у NT окружењу ово креирање процеса врши се индиректно. Процес који је клијент апликације издаје захтев за креирањем процеса подсистему оперативног система. Тада процес у подсистему са своје стране издаје захтев NT егзекутиви. Пошто је жељени ефекат да нови процес наследи карактеристике процеса клијента а не процеса сервера, NT омогућава да подсистем означи родитеља новог процеса. Нови процес наслеђује родитељев токен приступа, границе квота, основни приоритет и подразумевану склоност ка процесорима.

Код NT-а не унапред дефинисаних релација између процеса. Међутим, и POSIX и OS/2 намећу хијерархијске односе. Сваки процес, изузев иницијалног, креиран је од стране другог процеса и сматра се потчињеним процесу родитељу. Коришћењем хендлова објеката подсистем оперативног система може да продржи ове релације међу процесима.

# 4 Конкурентни процеси

У овој глави ћемо се бавити општим проблемом конкуренције, и то како унутар самог процеса, тако и између више процеса.

## 4.1 Графови првенства

Размотримо следећи програмски сегмент:

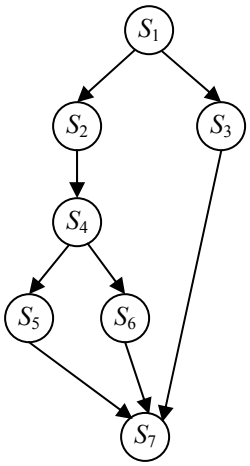
$a := x + y;$
$b := z + 1;$
$c := a - b;$
$w := c + 1;$

Претпоставимо да неке од ових исказа желимо да извршавамо конкурентно. У нашем процесору можемо имати више функционалних јединица или можемо располагати са више процесора. Јасно је да се исказ $c := a - b$ не може извршити пре претходна два, као што се ни исказ $w := c + 1$ не може извршити пре него се изврши њему претходни исказ. С друге стране, искази $a := x + y$ и $b := z + 1$ могу да се изврше конкурентно, јер ни један од њих не зависи од другог.

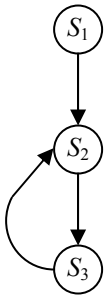
Из овога видимо да у једном програму могу да постоје *ограничења првенства* између различитих исказа.

### *4.1.1 Дефиниција графова првенства*

*Граф првенства* је усмерени ациклични граф чији су чворови појединни искази. Потег од чвора $S_i$ до чвора $S_j$ значи да се исказ $S_j$ може извршити тек пошто се изврши исказ $S_i$. На слици 4.1 видимо пример једног графа првенства. Граф првенства мора да буде ацикличан. Граф на слици 4.2. представља исказе који очигледно не могу да се изврше.




Сл. 4.1. Граф првенства.




Сл. 4.2. Граф првенства са циклусом.

### *4.1.2 Услови конкуренције*

Поставља се питање када два исказа у програму могу да се изврше конкурентно а да то дâ исти резултат као и када се они извршавају секвенцијално. Другим речима, када треба да постоји потег између чворова $S_1$ и $S_2$ у графу првенства који одговара неком програму.

Пре него што одговоримо на ово питање увешћемо следеће нотације:

- $\mathbf{R}(S_i) = \{a_1, a_2, \ldots, a_m\}$ *скуп читања* за $S_i$ је скуп свих променљивих чије се вредности користе током извршења исказа $S_i$.
- $\mathbf{W}(S_i) = \{b_1, b_2, \ldots, b_n\}$ *скуп уписа* за $S_i$ је скуп свих променљивих чије се вредности мењају током извршења исказа $S_i$.

Да би илустровали ове нотације посматрајмо исказ $c := a - b$. Вредности променљивих $a$ и $b$ се користе за израчунавање нове вредности променљиве $c$. Према томе, $a$ и $b$ припадају скупу читања. Стара вредност за $c$ се не користи у исказу већ се дефинише нова вредност као резултат извршења исказа, па је променљива $c$ у скупу уписа, али не и у скупу читања.

$\mathbf{R}(c := a - b) = \{a, b\}$
$\mathbf{W}(c := a - b) = \{c\}$

За исказ $w := c + 1$ скупови читања и уписа су:

$\mathbf{R}(w := c + 1) = \{c\}$
$\mathbf{W}(w := c + 1) = \{w\}$

Пресек ових скупова не мора да буде празан скуп. На пример, за исказ $x := x + 2$ је

$$\mathbf{R}(x := x + 2) = \mathbf{W}(x := x + 2) = \{x\}$$

Још један пример се тиче исказа $read(a)$. Променљива $a$ се учитава, дакле њена вредност се мења, па су скупови читања и уписа

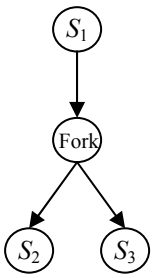
$\mathbf{R}(read(a)) = \{\}$
$\mathbf{W}(read(a)) = \{a\}$

Следећа три услова морају да буду задовољена за два узастопна исказа $S_1$ и $S_2$ да би они могли да се извршавају конкурентно и да при том дају исти резултат као и код секвенцијалног извршења. Ови услови су познати као *Берштајнови услови*.

1. $\mathbf{R}(S_1) \cap \mathbf{W}(S_2) = \{\}$
2. $\mathbf{W}(S_1) \cap \mathbf{R}(S_2) = \{\}$
3. $\mathbf{W}(S_1) \cap \mathbf{W}(S_2) = \{\}$

## 4.2 Спецификација конкуренције

Иако су графови првенства корисно средство за дефинисање ограничења првенства делова неког израчунавања, њих је тешко користити у програмским језицима, обзиром да се ради о дводимензионалним објектима. Стога се користе друга средства која омогућују програмеру да специфицира релације првенства између различитих исказа у програму.



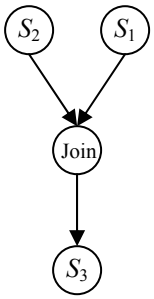

Сл. 4.3. Граф првенства за конструкцију **fork**.

### *4.2.1 Конструкције* Fork *и* Join

Инструкције **fork** и **join** увели су Конвеј (Conway) 1963 и Денис (Dennis) и Ван Хорн (Van Horn) 1966. Оне су биле међу првим језичким нотацијама са специфицирање конкуренције.

Инструкција **fork** $L$ производи два конкурентна извршења у програму. Једно извршење почиње од исказа означеног са $L$, док друго почиње са наредним исказом иза **fork** инструкције. На пример, посматрајмо следећи програмски сегмент

```
  S1;
  fork L;
  S2;
  .
  .
  .
L: S3
```




Сл. 4.4. Граф првенства за конструкцију **join**.

Део графа првенства који одговара овом програму представљен је на слици 4.3. Инструкција **join** пружа начин да се два конкурентна израчунавања удруже у једно. Како се та два израчунавања могу одвијати различитим брзинама могуће је да једно од њих изврши **join** инструкцију пре другог. У том случају се израчунавање које је прво извршило ову инструкцију привремено прекида док се оном другом допушта да се

настави. Потребно је да знамо број израчунавања које ће се удружити. **Join** инструкција има параметар који специфицира тај број, тако да **join** инструкција са параметром *count* има следећи ефекат:

*count* := *count* – 1;
**if** *count* <> 0 **then** *quit*;

при чему је *count* ненегативна целобројна променљива, а *quit* је инструкција која терминира израчунавање које изврши ову конструкцију. Променљива *count* треба да буде иницијализована на вредност која одговара броју израчунавања које се удружују.

**Join** инструкција мора да се изврши *атомски*, што значи да је конкурентно извршење два joин исказа еквивалентно секвенцијалном извршењу у недефинисаном редоследу.

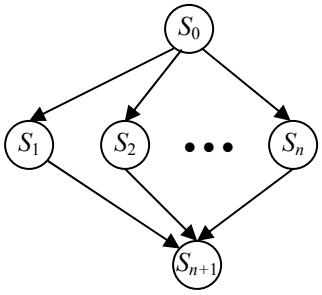
Недостатак конструкција **fork** и **join** је у томе што се због неопходне употребе **гото** исказа добија изразито неструктурни, а тиме и нејасан, кôд.

### *4.2.2 Конкурентни искази*

Конструкција за специфицирање конкуренције у вишим програмским језицима је исказ **parbegin**/**parend** који је увео Дијкстра (Dijkstra) 1965, и који је следећег облика:

**parbegin** $S_1$; $S_2$; …; $S_n$ **parend**;

Свако $S_i$ представља један исказ. Сви искази унутар заграда **parbegin** и **parend** се извршавају конкурентно. Граф првенства који одговара горњем исказу дат је на слици 4.5, где су $S_0$ и $S_{n+1}$ искази који се јављају непосредно пре и после конструкције **parbegin**/**parend**, респективно.




Сл. 4.5. Граф првенства за конкурентни исказ.

### *4.2.3 Поређење конструкција* fork *и* join *и конкурентних израза*

На жалост, конкурентни изрази не могу да опишу све могуће графове првенства. У случајевима када то није могуће користе се и други синхронизациони механизми, као што су *семафори*. С друге стране, већ смо видели да конструкције **fork** и **join**, у општем случају, производе неструктурни и тешко читљив кôд. Када говоримо о конкурентним изразима, треба имати у виду да у пракси није потребно изразити све могуће графове првенства, већ само оне који се заиста јављају у реалним проблемима. Не може се са сигурношћу тврдити да конкурентни изрази не могу да опишу све проблеме из реалног света.

## 4.3 Проблем критичне секције

Посматрајмо систем који се састоји од $n$ конкурентних процеса $\{P_1, P_2, \ldots, P_n\}$. Сваки процес има део кôда који се назива *критична секција* у којој може да приступа неким глобалним променљивама. Важна особина система је да само један процес може да извршава своју критичну секцију, тј. извршење критичних секција процеса је узајамно искључиво у времену. Проблем критичне секције састоји се у проналажењу протокола на основу којег процеси могу да се конкурентно извршавају. Сваки процес мора да захтева дозволу за улазак у своју критичну секцију. Тај део кôда се назива *улазна секција*. Иза кртитичне секције налази се *излазна секција*, а остатак кôда представља *секцију остатка*.

Решење проблема критичне секције мора да задовољи следећа три захтева:

a) **Узајамно искључивање.** Ако се процес $P_i$ извршава у својој критичној секцији тада ниједан други процес не може да извршава своју критичну секцију.
b) **Напредовање.** Ако нема процеса у критичној секцији и постоје неки процеси који желе да уђу у своје критичне секције, тада само ти процеси могу да одлучују о томе који ће процес следећи ући у критичну секцију, а тај избор се не може бесконачно одлагати.
c) **Ограничено чекање.** Мора да постоји граница у односу на број пута када се другим процесима дозвољава да уђу у критичну секцију пошто је неки процес издао захтев за улазак у критичну секцију, а пре него што му је тај улазак одобрен.

Подразумева се да се сваки процес извршава брзином која је већа од нуле, али се никаква претпоставка о релативним брзинама процеса не може направити.

### *4.3.1 Софтверско решење за два процеса*

У овом одељку ћемо се ограничити на алгоритме који су примењиви само када у систему постоје два процеса. Процеси су означени са $P_0$ и $P_1$ а општа структура проблема је

```
begin
        Deklaracije globalnih promenljivih;
        parbegin
                P0;
                P1;
        parend;
end.
```

Како су решења за поједине процесе симетрична приказиваћемо, осим глобалних променљивих, алгоритам за процес $P_i$, претпостављајући да је $P_j$ други процес, тј. $j = 1 - i$.

Општа структура процеса $P_i$ је следећа:

```
repeat
    ulazna sekcija;
        kritična sekcija;
    izlazna sekcija;
        sekcija ostatka;
until false;
```

**<u>Алгоритам 1</u>**

Први приступ решавању проблема састојаће се у омогућавању да процеси деле глобалну променљиву *redjena* која се иницијализује било на 0 било на 1. Ако је *redjena* = *i*, процесу $P_i$ се дозвољава да уђе у критичну секцију.

```
repeat
    while redjena <> i do skip;
        kritična sekcija;
    redjena:=j;
        sekcija ostatka;
until false;
```

Ово решење обезбеђује да само један процес истовремено може бити у критичној секцији. Међутим, оно не задовољава захтев напредовања, јер дозвољава искључиво да се процеси смењују у критичној секцији. Један процес не може два или више пута заредом да уђе у критичну секцију, без обзира што други процес све време извршава своју секцију остатка.

**<u>Алгоритам 2</u>**

Проблем са Алгоритмом 1 је што он не памти стање сваког процеса, већ само ком је процесу допуштено да уђе у критичну секцију. Да би превазишли овај проблем заменићемо променљиву *redjena* следећим пољем.

```
var flag: array[0..1] of boolean;
```

Сви елементи поља иницијализују се на *false*. Ако је *flag*[*i*] једнако *true*, процес $P_i$ може да уђе у критичну секцију. Општа структура процеса $P_i$ је следећа.

```
repeat
    while flag[j] do skip;
    flag[i] := true;
        kritična sekcija;
    flag[i] :=false;
        sekcija ostatka;
until false;
```

Овде се најпре проверава да ли је други процес у критичној секцији и по потреби чека све док он не изађе из ње. Тада процес поставља свој маркер *flag*[*i*] на *true* и улази у критичну секцију. Када изађе из своје

критичне секције поново поставља свој маркер на *false*, дозвољавајући да други процес уђе у критичну секцију уколико је чекао на то. Међутим, овај алгоритам не обезбеђује узајамну искључивост, тј. могуће је да оба процеса уђу у критичну секцију.

## Алгоритам 3

Проблем са Алгоритмом 2 је да је процес $P_i$ направио одлуку о уласку у критичну секцију проверавајући стање процеса $P_j$ пре него што је овај стигао да промени *flag*[*j*] на *true*. У овом алгоритму ћемо покушати да исправимо тај проблем. Као и у Алгоритму 2, задржаћемо поље *flag*, али ће нам сада *flag*[*i*] = *true* указивати једино да $P_i$ жели да уђе у критичну секцију.

```
repeat
    flag[i] := true;
    while flag[j] do skip;
        критична секција;
    flag[i] := false;
        секција остатка;
until false;
```

Овде је задовољен услов узајамног искључивања, али не и захтев напредовања. Уколико оба процеса поставе своје маркере на *true* пре него што било који од њих изврши **while** исказ, обојица ће препуштати оном другом да уђе у критичну секцију и тако ниједан од њих то никада неће учинити.

## Алгоритам 4

Најзад, представићемо коректно решење проблема критичне секције које је дао Питерсон (Peterson) 1981. У основи, то решење се заснива на Алгоритму 3 и малој модификацији Алгоритма 1. Процеси деле две глобалне променљиве

```
var flag: array[0..1] of boolean;
    redjena: 0..1;
```

Иницијално је *flag*[0] = *flag*[1] = *false*, док вредност променљиве *redjena* може бити било 0 било 1. Структура процеса $P_i$ је следећа.

```
repeat
    flag[i] := true;
    redjena := j;
    while (flag[j] and redjena = j) do skip;
        критична секција;
    flag[i] := false;
        секција остатка;
until false;
```

Када процес жели да уђе у критичну секцију, он најпре поставља *flag*[*i*] на *true*, препуштајући да је ред на други процес да уђе у критичну секцију уколико он то жели (*redjena* := *j*). Ако оба процеса желе да у исто време уђу у критичну секцију, сигурно ће вредност променљиве *redjena* бити постављена на индекс тачно једног од њих, чиме се том процесу допушта да заиста уђе у критичну секцију.

Показаћемо да је Питерсоново решење коректно. Да би то урадили треба да покажемо да се задовољена сва три захтева.

Да би показали да је задовољен захтев за узајамним искључивањем приметимо да $P_i$ улази у критичну секцију само ако је или *flag*[*j*] = *false* или је *redjena* = *i*. Такође приметимо да ако би оба процеса била у критичној секцији истовремено тада би било *flag*[0] = *flag*[1] = *true*. Ова два запажања имплицирају да оба процеса не могу успешно да изврше своје **while** исказе у исто време јер *redjena* може бити или 0 или 1, а никако не може имати истовремено обе вредности. Према томе, само један процес, рецимо $P_j$, успешно пролази **while** исказ, док процес $P_i$ то не може да уради због тога што је *redjena* = *j*. Тиме је очувана узајамна искључивост. Да би показали да су испуњени захтеви за напредовање и ограничено чекање запазимо да се процес $P_i$ може спречити да уђе у критичну секцију једино ако се врти у **while** петљи зато што је испуњен услов *flag*[*j*] **and** *redjena* = *j*. Ако процес $P_j$ није заинтересован да уђе у критичну секцију *flag*[*j*] = *false*, и процес $P_i$ може да уђе у критичну секцију. Ако је $P_j$ поставио *flag*[*j*] = *true*, *redjena* може бити или *i* или *j*. Ако је *redjena* = *i*, $P_i$ улази у критичну секцију. Ако је *redjena* = *j*, тада $P_j$ улази у критичну секцију али ће, пошто је изврши, да постави *flag*[*j*] = *false* допуштајући процесу $P_i$ да уђе у критичну секцију. Осим тога, $P_j$ је морао

да постави и *redjena* = *i*. Како $P_i$ не може да промени *redjena* док је у **while** петљи, $P_i$ улази у критичну секцију (напредовање) после највише једног уласка процеса $P_j$ у критичну секцију (ограничено чекање).

### *4.3.2 Софтверска решења за N процеса*

Овде ћемо приказати два решења овог проблема за *n* процеса у систему. Прво решење дали су Ајзенберг (Eisemberg) и Мекгвајер (McGuire) 1972., а друго Лампорт (Lamport) 1974.

**<u>Алгоритам 5</u>**

Заједничке структуре података су

```
var flag: array[0..n-1] of (neradi, zeliu, uks);
    redjena: 0..n-1;
```

Сви елементи поља *flag* иницијално су *neradi*, док почетна вредност за *redjena* није од значаја. Структура процеса $P_i$ је следећа.

```
var j: 0..n;
repeat
    repeat
        flag[i] := zeliu;
        j : = redjena;
        while j <> i do
                if flag[j] <> neradi then
                        j := redjena
                else
                        j := j+1 mod n;
        flag[i] := uks;
        j := 0;
        while (j < n) and (j = i or flag[j] <> uks) do j := j+1;
    until (j >= n) and (redjena = i or flag[redjena] = neradi);
    redjena := i;
        kritična sekcija;
    j := redjena + 1 mod n;
    while (flag[j] = neradi) do j := j + 1 mod n;
    redjena := j;
    flag[i] := neradi;
        sekcija ostatka;
until false;
```

Да би доказали да овај алгоритам поседује особину узајамне искључивости, запазимо да $P_i$ улази у критичну секцију само ако је *flag*[*j*] <> *uks* за свако *j* <> *i*. Како само$P_i$ може да постави *flag*[*i*] = *uks* и како $P_i$ испитује *flag*[*j*] само док је *flag*[*i*] = *uks*, то увек највише један процес може бити у критичној секцији.

За показивање својства напредовања запазимо да се вредност променљиве *redjena* може модификовати само када процес улази у своју критичну секцију и када је напушта. Према томе, ако нема процеса у критичној секцији или процеса који напушта критичну секцију, вредност за *redjena* остаје константна. У критичну секцију ће ући први од процеса кои су надмећу за улазак у њу према цикличном редоследу (*i*, *i*+1, …, *n*-1, 0, …, *i*-1).

Да би показали ограничено чекање приметимо да када процес напушта критичну секцију он мора да означи као јединственог наследника први међу процесима који се надмећу за улазак у критичну секцију из наведеног редоследа. Тиме је обезбеђено да процес који чека на улазак у критичну секцију то може да уради у највише *n*-1 циклуса.

**<u>Алгоритам 6</u>**

Различит приступ је предложио Лампорт (1974) у свом *пекаровом алгоритму*. Овај алгоритам је развијен за дистрибуисане околине, а ми ћемо се сконцентрисати само на оне аспекте који важе и за централизоване околине.

Пекарев алгоритам је базиран на алгоритму планирања који се обично (у писменом свету) користи у пекарама, посластичарама, месарама и у сличним ситуацијама. Када муштерија уђе у радњу он добије број.

Муштерија са најнижим бројем се прва услужује. На жалост, овај алгоритма не гарантује да два процеса не могу да приме исти број. У таквом случају процес са мањим индексом се опслужујуе први. Како су имена процеса јединствена и потпуно уређена, овај алгоритам је потпуно детерминистички.

Заједничке структуре података су

```
var izbor: array[0..n-1] of boolean;
    broj: array[0..n-1] of integer;
```

Иницијално су ове структуре постављене на *false* и на 0, респективно. Дефинишимо следеће нотације

- $(a,b) < (c,d)$ ако је $a < c$ или ако је $a = c$ и $b < d$
- $max(a_0, \ldots, a_{n-1})$ је број $k$ за који важи да је $k \geq a_i$, за $i = 0, \ldots, n - 1$.

Структура процеса $P_i$ је следећа.

```
repeat
    izbor[i] := true;
    broj[i] : = max[broj[0], …, broj[n-1])+1;
    izbor[i] := false;
    for j := 0 to n – 1 do
        begin
            while izbor[j] do skip;
            while broj[j] <> 0 and (broj[j], j) < (broj[i], i) do skip;
        end
    kritična sekcija;
    broj[i] : = 0;
    sekcija ostatka;
until false;
```

Да би показали коректност пекаровог алгоритма треба да покажемо да ако је $P_i$ у критичној секцији и $P_k$ ($k <> i$) је већ изабрао свој $broj[k] <> 0$, тада је $(broj[i], i) < (broj[k], k)$. Иза тога је лако показати узајамно искључивање. Заиста, ако је $P_i$ у критичној секцији и $P_k$ извршава други **while** исказ за $j = i$ он налази да је

- $broj[i] <> 0$, и
- $(broj[i], i) < (broj[k], k)$.
-

Према томе, он наставља да се врти у петљи све док $P_i$ не заврши своју критичну секцију.

Да би се показале особине напредовања и ограниченог чекања, као и да алгоритам обезбеђује правичност, довољно је уочити да процеси улазе у критичну секцију на FCFS оснви.

### *4.3.3 Хардверска решења*

Многе машине имају специјалне хардверске инструкције које дозвољавају да се тестира и модификује садржај неке меморијске локације, или да се замени садржај две меморијске локације у једном меморијском циклусу. Ове специјлане инструкције се могу употребити за решење проблема критичне секције. Размотримо најпре основне концепте ових инструкција дефинишући *Test-and-Set* инструкцију на следећи начин:

```
Function Test-and-Set(var target: boolean): boolean;
    Begin
      Test-and-Set:=target;
      Target:=true;
    End;
```

а *Swap* инструкцију као:

```
Procedure Swap(var a, b: boolean): boolean;
    Var temp: boolean;
    Begin
      temp:=a;
      a:=b;
      b:=temp;
    End;
```

Важна карактеристика ових инструкција је да се оне извршавају атомски, тј. у једном меморијском циклусу. Због тога, уколико се две такве инструкције извршавају симултано, њихов ефекат је као да се извршавају секвенцијално у произвољном редоследу.

Ако машина подржава *Test-and-Set* инструкцију, тада се узајамно искључивање може имплементирати декларисањем логичке променљиве *lock* иницијализиране на *false*.

```
repeat
    while Test-and-Set(lock) do skip;
        критична секција;
    lock := false;
        секција остатка;
until false;
```

Ако машина подржава *Swap* инструкцију, тада се узајамно искључивање може имплементирати на сличан начин. Глобална логичка променљива *lock* иницијализује се на *false*. Осим овога, сваки процес има локалну логичку променљиву *key*.

```
repeat
    key:=true;
    repeat
        Swap(lock, key);
    until key = false;
        критична секција;
    lock := false;
        секција остатка;
until false;
```

Горња два алгоритма не задовољавају захтев за ограниченим чекањем, тако да треба користити додатне променљиве. Барнсов алгоритам (1978) користи *Test-and-Set* инструкцију и задовољава све захтеве проблема критичне секције.

Заједничке структуре података су

```
var cekanjer: array[0..n-1] of boolean;
    lock: boolean;
```

Иницијално су ове структуре постављене на *false*. Структура процеса $P_i$ је следећа.

```
var j: 0..n-1;
    key: boolean;
repeat
    cekanje[i] := true;
    key : = true;
    while cekanje[i] and key do key := Test-and-Set(lock);
    cekanje[i] := false;
    критична секција;
    j : = i + 1 mod n;
    while (j <> i) and (not cekanje[j]) do j := j + 1 mod n;
    if j = i then
        lock := false
    else
        cekanje[j] := false;
    секција остатка;
until false;
```

Да би доказали да је задовољено узајамно искључивање запазимо да $P_i$ улази у критичну секцију једино ако је *cekanje*[*i*] = *false* или *key* = *false*. *Key* може постати *false* јединп извршавањем *Test-and-Set*. Први процес који изврши *Test-and-Set* наћи ће *key* = *false*, док сви остали морају да чекају. *Cekanje*[*i*] може постати *false* једино ако неки други процес напусти своју критичну секцију; само једно с*ekanje*[*i*] се поставља на *false*, чиме се одржава узајамно искључивање.

За доказивање захтева напредовања запазимо да се исти аргументи могу и овде применити пошто процес који напушта критичну секцију или поставља *lock* на *false* или *cekanje*[*j*] на *false.* Оба од ова два дозвољавају просецу који чека да уђе у критичну секцију.

Што се ограниченог чекања тиче, уочимо да кад процес напушта своју критичну секцију он проверава поље *cekanje* у цикличном редоследу ($i$+1, $i$+2, …, $n$-1, 0, …, $i$-1). Он, затим, означава први процес у овом поретку који је у улазној секцији (тј. код кога је *cekanje*[*j*] = *true*) као следећи који ће ући у критичну секцију. То значи да ће сваки процес који чека на улазак у критичну секцију то и учинити у оквиру $n$-1 циклуса.

## 4.4 Семафори

Решења проблема критичне секције представљена до сада не могу се генерализоивати на сложеније проблеме. Због тога је уведена нова синхронизациона примитива названа *семафор* (Дијкстра 1965). Семафор је целобројна променљива којој се може приступати једино преко атомских операција *P* и *V*. Класична дефиниција тих операција је:

*P*(*S*): **while** *S* <= 0 **do** *skip*;
*S* := *S* - 1;
*V*(*S*): *S* := *S* + 1;

### *4.4.1 Коришћење семафора*

Семафори се могу користити за решавање проблема критичне секције са $n$ процеса. Процеси деле глобалну семафорску променљиву *mutex* која се иницијализује на 1. Сваки процес $P_i$ организован је на следећи начин:

**repeat**
*P*(*mutex*);
kritična sekcija;
*V*(*mutex*);
sekcija ostatka;
**until** *false*;

Семафори могу да се користе за решавање различитих синхронизационих проблема. На пример, посматрајмо два конкурентна процеса: $P_1$ са исказом $S_1$ и $P_2$ са исказом $S_2$. Претпоставимо да се $S_2$ може извршити само после извршења $S_1$. Ово се лако може имплементирати ако процеси деле семафорску променљиву *synch* иницијализовану на 0 ако у процес $P_1$ убацимо

$S_1$;
*V*(*synch*);

а у процес $P_2$ убацимо

*P*(*synch*);
$S_2$;

Семафори се могу користити и у сложенијим проблемима. На пример, конкурентни изрази, за разлику од конструкција **fork** и **join**, не могу да опишу све графове првенства. Овај недостатак се надомешћује тако што се коришћење конкурентних израза комбинује са употребом семафора.

### *4.4.2 Имплементација семафора*

Главни недостатак код решења проблема критичне секције и код дефиниције семафора које су до сада представљене је у томе што свако од њих захтева *заузетост чекањем* (*busy-waiting*). Заузетост чекањем изазива да неки процес троши циклусе CPU-а иако не ради ништа корисно, док би за то време други процеси могли да обављају своје CPU активности. Да би се превазишао овај проблем треба модификовати семафорске операције *P* и *V*. Када процес установи да вредност семафорске променљиве није позитивна, уместо чекања он би могао да блокира сам себе. Операција блокирања смешта процес у стање чекања а управљање се преноси на планер процеса који бира други процес, из реда спремних процеса, да се извршава на CPU-у. Процес који је блокиран због чекања на семафору *S* треба да се обнови извршењем операције *V* од стране неког другог процеса. Процеси се обнављају *wakeup* операцијом које мења стање процеса из блокиран у спреман и смешта га у ред спремних процеса.

Да би имплементирали семафоре на основу ове дефиниције, дефинисаћемо семафор као запис.

```
type    semaphore = record
                    value: integer;
                    L: list of process;
                end;
```

Семафорске операције се сада могу дефинисати на следећи начин:

```
P(S):   S.value := S.value – 1;
        if S.value < 0 then
            begin
                dodaj ovaj proces u S.L;
                block;
            end;

V(S):   S.value := S.value + 1;
        if S.value <= 0 then
            begin
                izbaci proces P iz S.L;
                wakeup(P);
            end;
```

Листа процеса се лако може имплементирати као повезана листа PCB-ова тих процеса. Сваки семафор садржи целобројну вредност и указатељ на листу PCB-ова. Најчешће се ова листа организује као ред са различитим алгоритмима за избацивање процеса из реда (FIFO, LIFO, приоритети и др.). Критични аспект семафора је тај да се извршење семафорских операција мора обављати атомски. Мора се гарантовати да се истовремено над истим семафором не могу обављати операције *P* и *V*. Ова ситуација је у ствари проблем критичне секције. Она се може решити на два начина. Код једнопроцесорских машина се једноставно забрањују прекиди током извршења ових операција. Код вишепроцесорских машина ово није довољно. Уколико хардвер не пружа одговарајуће специјалне инструкције, може се употребити било које коректно софтверско решење проблема критичне секције, при чему се критичне секције састоје од семафорских операција.

## 4.5 Класични проблеми координације процеса

Представићемо неколико различитих синхронизационих проблема који се важни као примери за велике класе проблема управљања конкуренцијом.

### *4.5.1 Проблем ограниченог бафера*

Овде се ради о апстракцији веома честог проблема у оперативним системима. У основи ради се о проблему произвођача и потрошача. *Произвођач* производи неке податке који се користе од стране процеса *потрошача*. Да би се омогућило да произвођач и потрошач раде конкурентно, креира се бафер који се пуни од стране процеса произвођача и празни од стране процеса потрошача. Мора се обезбедити синхронизација ова два процеса тако да потошач не покушава да узме податке из празног бафера.

У варијанти проблема са *неограниченим бафером* имамо да нема ограничења у броју ћелија у баферу. Потрошач мора понекада да чека да се произведе нови податак, док произвођач увек може да смести произведени податак. Варијанта са ограниченим бафером претпоставља да постоји фиксиран број *n* ћелија у баферу, тако да се може десити да и произвођач чека у ситуацији када су све ћелије у баферу попуњене.

У решењу које следи *in* и *out* су два индекса у баферу који указују на следећу празну ћелију и следећу пуну ћелију, респективно. Семафор *mutex* обезбеђује узајамно искључивању у приступу баферу, док семафори *empty* и *full* броје празне и пуне ћелије, респективно.

```
type item = T;
var bafer: array[0..n-1] of item;
    in, out: 0..n-1;
    nextp, nextc: item;
    full, empty, mutex: semaphore;
begin
```

```
in := 0; out := 0; full := 0; empty := n; mutex := 1;
parbegin
  proizvodjac: begin
                 repeat
                   proizvodi(nextp);
                   P(empty);
                   P(mutex);
                   bafer[in]:=nextp;
                   in:=(in+1) mod n;
                   V(mutex);
                   V(full);
                 until false;
               end;
  potrosac: begin
                 repeat
                   P(full);
                   P(mutex);
                   nextc:=bafer[out];
                   out:=(out+1) mod n;
                   V(mutex);
                   V(empty);
                   koristi(nextp);
                 until false;
               end;
parend;
end.
```

### *4.5.2 Проблем читалаца и писаца*

Подаци (обично датотеке или записи) се често деле између више конкурентних процеса. Неки од тих процеса имају потребу једино да прочитају садржај тих података, док други могу и да их ажурирају. Прву врсту процеса назваћемо *читаоцима*, а другу *писцима*. Очигледно је да два читаоца могу истовремено да приступају подацима. Међутим, ако писац и било који други процес могу да приступе истовремено истом податку долази до нарушавања Бернштајнових услова.

Да се ови проблеми не би јављали захтева се да писци имају ексклузивно право приступа дељивим подацима. Овај синхронизацион проблем, познат као проблем читалаца и писаца, дефинисао је и решио Куртоа (Courtois) 1971. Овај проблем има више варијанти. Најједноставнија, названа *први проблем читалаца и писаца* захтева да ниједан читалац не сме да чека осим ако је неки писац већ добио дозволу да користи дељиви објекат. Другим речима ниједан читалац не чека на друге читаоце да заврше јер писац чека. *Други проблем читалаца и писаца* захтева да када је писац спреман, он обавља писање што је пре могуће. Другим речима, ако писац чека, ниједан нови читалац не може да започне читање.

Приметимо да решења оба проблема могу довести до умирања од глади. У првом случају, писци могу да умиру од глади, а у другом читаоци.

Овде ћемо приказати решење првог проблема читалаца и писаца. Дељиви подаци су следећи:

```
var mutex, pisanje: semaphore;
    brcitalaca: integer;
```

Семафори *mutex* и *pisanje* се иницијализују на 1, док је *brcitalaca* иницијализован на 0. Семафор *pisanje* је заједнички и за читаоце и за писце. Семафор *mutex* обезбеђује узајамну искључивост током ажурирања промењиве *brcitalaca*. *Brcitalaca* броји колико процеса тренутно чита податке. Семафор *pisanje* је обезбеђује узајамну искључивост за писце, а такође се користи од стране првог (последњег) читаоца који улази (излази) у (из) критичну секцију.

Општа структура процеса читаоца је

```
P(mutex);
    brcitalaca := brcitalaca + 1;
    if brcitalaca = 1 then P(pisanje);
V(mutex);
    …
    citanje
```

```
    …
P(mutex);
    brcitalaca := brcitalaca - 1;
    if brcitalaca = 0 then V(pisanje);
V(mutex);
```

док је општа структура процеса писца следећа.

```
P(pisanje);
    ...
    pisanje
    …
V(pisanje);
```

### *4.5.3 Проблем филозофа који вечерају*

Проблем филозофа који вечерају поставио је и решио Дијкстра (1965). Проблем се сатоји у следећем. Пет филозофа проводи свој живот размишљајући и једући. Филозофи деле округли сто за којим је пет столица, за сваког филозофа по једна. На столу се налази чинија пиринча и пет штапића. Док филозоф размишља он не интерреагује са својим колегама. Повремено филозофи огладне и покушавају да узму два штапића која су најближа месту на којем седе. Филозоф може да узме само један штапић истовремено. Наравно не може да узме штапић који је неки од његових суседа већ узео, Када гладан филозоф узме оба штапића, тада може да једе. Када заврши са јелом спушта оба штапића на сто.

Једноставно решење које ћемо приказати представља штапиће семафорима. Заједнички подаци су:

**var** *stapic*: **array**[0..4] **of** *semaphore*;

где се сви елементи поља *стапиц* иницијализују на1. Структура филозофа *i* је следећа:

```
repeat
    P(stapic[i]);
    P(stapic[(i+1) mod 5]);
    jede;
    V(stapic[i]);
    V(stapic[(i+1) mod 5]);
    misli;
until false;
```

Мада ово решење обезбеђује да два суседа не могу истовремено да једу, оно се ипак мора одбацити због могућности застоја. Постоји више могућности да се избегне појава застоја, а она се заснивају на некој од следећих стратегија:

- Дозволити да највише четири филозофа могу да седну за сто.
- Дозволити да филозоф узима штапић једино ако су оба штапића око њега слободна.
- Користити асиметрично решење. Непарни филозофи узимају најпре леви штапић, док парни филозофи узимају десни штапић.

Решење које би било задовољавајуће мора да поведе рачуна и о проблему умирања од глади.

# 5 Конкурентно програмирање

Оперативни системи се данас углавном пишу на вишим програмским језицима. На тај начин је омогућена њихова боља имплементација и одржавање као и преносивост. У овој глави ћемо проучити еволуцију конструкција виших програмских језика за конкурентно програмирање.

## 5.1 Мотивација за коришћење виших програмских језика

Оперативни систем се састоји од великог броја програма који раде асинхроно и који сарађују у испуњавању задатака оперативног система. Традиционално су ови програми писани у асемблерском језику из следећих разлога:

- Виши програмски језици нису пружали механизме за писање машински зависног кôда, као што су драјвери уређаја.
- Виши програмски језици нису пружали одговарајуће алате за писање конкурентних програма.
- Виши програмски језици за конкурентно програмирање нису били ефикасни.

Међутим, током времена су се развијали нови језици који су подржавали конкуренцију и који су били релативно ефикасни. Шта више, многи аргументи против употребе виших програмских језика засновани су на претпоставци да се дотични језик користи у писању централних функција оперативног система, као што су управљање меморијом и управљање процесима. Но, већина кôда оперативног система се не односи на извршавање тог нивоа функција оперативног система. Према томе, уз погодан виши програмски језик, тај део кôда се не мора писати у асемблерском језику.

Постоје и предности у употреби виших програмских језика. Програми на вишим програмским језицима су лакши за тестирање, модификацију и преношење са машине на машину, име се скраћује време потребно за развој и тестирање програма. Главни недостатак виших програмских језика је проблем генерисања неефикасног кôда. Овај проблем се може решити користећи различите технике са оптимизирајућим компилаторима. Данас оптимизирајући компилатори генеришу кôд који је у најмању руку добар као и ручуно писани асемблерски програми. Степен оптимизације који се може постићи зависи од изабраног језика. На пројектанту језика је да обезбеди да се предвиђене језичке конструкције могу ефикасно имплементирати.

Ми ћемо се углавном бавити вишим програмским језицима који се користе за писање конкурентних програма. Такви језици морају да обезбеде средства за модуларизацију и синхронизацију.

## 5.2 Модуларизација

Модуларизација је појам који се користи за дељење једног великог програма на више мањих модула. Извршићемо преглед типова модула које можемо да очекујемо у оперативном систему писаном на неком вишем програмском језику.

### *5.2.1 Процеси*

Процеси су основни градивно блок оперативних система. Оперативни систем је састављен од одређеног броја асинхроних процеса који морају да комуницирају и синхронизују своје акције. У најблажем случају процеси деле све променљиве. Тада се заштита против временски зависних грешака мора уградити у саме алгоритме за синхронизацију, а компилатор не може да помогне програмеру у овом послу. Комуникација и синхронизација се могу постићи једино путем слања порука, односно преносном параметара између појединих

Сл. 5.1. Шематски приказ процеса.

процедура.

Процес се састоји од извесног броја локалних података и секвенцијалног програма који ради са тим подацима (слика 5.1). Локалним подацима се може приступати једино од стране секвенцијалног програма који је инкапсулиран у том процесу. Према томе, процес не може директно да приступа локалним подацима другог процеса. Ако процеси могу да деле глобалне податке, ти подаци морају да буду дефинисани у заједничкој области или инкапсулирани у процедуре или апстрактне типове података.

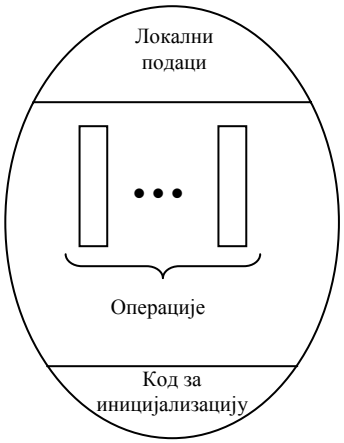
### *5.2.2 Процедуре*

Следећа јединица модуларизације јесу процедуре или функције. Ако процедуре могу да имају сопствене постојане променљиве (које не нестају између два позива порцедуре), тада се неки глобални подаци могу инкапсулирати унутар процедуре. Тада процес не може да приступа тим подацима директно, већ мора да позове одговарајућу процедуру.

Инкапсулација штити кориснике процедура од имплементационих детаља, допуштајући им да се сконцентришу на начин употребе процедуре а не на стварну имплементацију процедуре.

### *5.2.3 Апстратктни типови података*

Процедуре пружају ограничени механизам за скривање информација. Да би у потпуности сакрили метод дефинисања података, морамо да имамо нешто компликованији механизам. Потребно нам је средство које омогућава програмеру да креира класу апстрактних објеката који нису експлицитно предвиђени приликом пројектовања програмског језика. Ова врста механизма данас постоји у многим програмским језицима пода називом *апстрактни типови података*. Један апстрактни тип података се карактерише скупом операција дефинисаних од стране програмера. Репрезентација апстрактног типа података састоји се од декларација променљивих чије вредности дефинишу стање једног примерка типа, као и тела процедура или функција које имплементирају операције над подацима тог типа (слика 5.2).




Сл. 5.2. Шематски приказ апстрактног типа података.

Класичан пример апстрактног типа података је магацин. Већина језика не предвиђа магацин као уграђени тип. Тако је магацин тип дефинисан од стране програмера. Магацин има мали скуп операција (push, pop, empty, top и др.). Магацину се може приступати једино преко тих операција.

Програмски модул који представља репрезентацију апстрактног типа података називаћемо *класом*. Овај појам је уведен у програмски језик *Simula 67* и коришћен је у *Конкурентном паскалу*, а касније и у многим другим језицима. Синтакса класе је:

```
type ime-klase = class
        deklaracije promenljivih

      procedure P1(…);
   begin … end;
      procedure P2(…);
   begin … end;
              .
              .
              .
      procedure Pn(…);
   begin … end;
begin
      kod za inicijalizaciju
end.
```

Јасно је да се репрезентација апстрактног типа података не сме користити од стране других компоненти програма осим преко оператора тог типа. Процедура дефинисана унутар класе може једино да приступа променљивама локално декларисаним унутар класе и својим формалним параметрима.

Механизам којим се у језику намеће заштита за типове дефинисане модулима је у ствари преформулисано правило досега у блок-структурним језицима. Модул извози имена и типове својих оператора, али не и репрезентацију типа који дефинише. Корисник апстрактног типа података може једино да позове операције које су извезене. Класа не мора да извози све своје процедуре. Да би разликовали процедура које су извезене од оних које то нису, користимо следећу нотацију:

**procedure entry** *P*(…);

Само процедуре **ентру** типа могу бити позване изван класе.

## 5.3 Синхронизација

Језик мора да обезбеди начин заштите од временски зависних грешака. Такве грешке се могу јавити ако више конкурентних процеса комуницира међусобно путем заједничких променљивих или путем размене порука.

### *5.3.1 Критичне области*

Семафори се могу ефикасно користити за решавање проблема критичне секције. Међутим, мале грешке у програму које се огледају у погрешном редоследу семафорских операција или у њиховом погрешном навођењу могу изазвати озбиљне проблеме. Из тих разлога уведена ја нова језичка конструкција названа *критична област*. Променљива *v* типа *T*, коју дели више процеса, може се декларисати као:

**var** *v:* **shared** *T;*

Променљивој *v* може се приступати само унутар исказа *region* следећег облика:

**region** *v* **do** *S;*

Ова конструкција значи да док се исказ *S* извршава ниједан други процес не може да приступи променљивој *v*. Критичне области могу да заштите од једноставних грешака које чини сам програмер, али у ствари не елиминишу све временски зависне грешке, већ само смањују њихов број.

### *5.3.2 Условне критичне области*

Критичне области се могу успешно користити за решавање проблема критичне секције, али се не могу применити за решавање неких општих синхронизационих проблема. Из тих разлога су уведене *условне критичне секције* (Хоар (Hoare) 1972.). То су конструкције облика:

**region** *v* **when** *B* **do** *S;*

где је *B* логички израз. Када процес улази у критичну област израчунава се вредност тог израза; и ако је она *true* извршава се исказ *S*. Ако је вредност израза *B* једнака *false*, процес ослобађа критичну секцију и одлаже се док *B* не постане *true* и ниједан други процес није у области придруженој променљивој *v*.

Илуструјмо овај концепт на проблему ограниченог бафера. Сам бафер и његови поинтери су инкапсулирани у следећој дељивој структури података:

```
var bafer: shared record
                niz: array[0..n-1] of item;
                broj, in, out: integer;
            end;
```

Произвођач додаје нови елемент *nextp* у бафер извршавајући

```
region bafer when broj < n do
  begin
    niz[in]:=nextp;
    in:=(in+1) mod n;
    broj:=broj+1;
  end;
```

Потрошач узима један елемент из дељивог бафера на следећи начин

```
region bafer when broj > 0 do
  begin
    nextc:=niz[out];
    out:=(out+1) mod n;
    broj:=broj-1;
  end;
```

Приметимо да се израчунавање услова код ове конструкције врши само на почетку критичне области. Постоје околности које налажу да се синхронизација врши било где унутар критичне области. Из тих разлога је Бринч Хансен (Brinch Hansen) 1972. увео следећу конструкцију:

```
region v do
  begin
    S1;
    await(B);
    S2;
  end;
```

Када процес уђе у критичну област он извршава исказ $S1$ (који може бити и празан). Потом се израчунава вредност израза $B$. Ако је она *true* извршава се $S2$. Ако је $B$ једнако *false* процес напушта узајамно искључивање и одлаже се док $B$ не постане *true* а ниједан други процес није у критичној области придруженој променљивој $v$.

Илустроваћемо овај концепт на другом проблему читалаца и писаца. Код овог проблема писац, када је спреман, може да пише чим је то могуће. Према томе, читаоци улазе у критичну секцију једино ако нема писаца који чекају или су у критичној секцији.

```
type citaoci-pisci = class
    var v: shared record
                    brcitalaca, brpisaca: integer;
                    zauzet: boolean;
                end;
    procedure entry pocni-citanje;
        region v do
          begin
              await(brpisaca = 0);
              brcitalaca:=brcitalaca+1;
          end;

    procedure entry zavrsi-citanje;
        region v do
          begin
              brcitalaca:=brcitalaca - 1;
          end;

    procedure entry pocni-pisanje;
        region v do
          begin
              brpisaca:=brpisaca+1;
              await((not zauzet) and (brcitalaca = 0));
              zauzet:= true;
          end;

    procedure entry zavrsi-pisanje;
        region v do
          begin
              brpisaca:=brpisaca - 1;
              zauzet:= false;
          end;

begin
    zauzet:=false;
    brcitalaca:=0;
    brpisaca:=0;
end.
```

Процес читаоц може да врши читање у следећој секвенци

*rw.pocni-citanje*;
…
*citaj podatke*
…
*rw.zavrsi-citanje;*

где је *rw* примерак класе *citaoci-pisci*. Слично, код уписа података извршава следећу секвенцу:

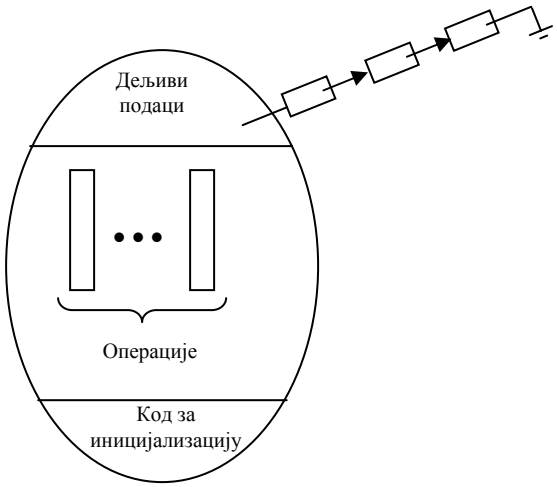
*rw.pocni-pisanje*;
…
*pisi podatke*
…
*rw.zavrsi-pisanje;*

Сам концепт класе не гарантује да ће се таква секвенца заиста и наћи у програму. Да би смо осигурали да ће процеси читаоци и писци заиста извршавати коректне секвенце, можемо да саме податке укључимо у класу. Тада процедуре из претходног примера не би биле **entry** типа. За узврат, увели би две нове процедуре, сада **entry** типа, које би садржавале претходне секвенце.

### *5.3.3 Монитори*

Са жељом да се одговарајућа синхронизација аутоматски обезбеди уведена је нова језичка конструкција названа *монитор* (Бринч Хансен 1973 и Хоар 1974). Монитор је механизам који допушта безбедно и ефикасно дељење апстрактних типова података између више процеса. Синтакса монитора је идентична синтакси класе осим што је кључна реч **class** замењена кључном речи **monitor**. Основна семантичка разлика између монитора и класе састоји се у томе што монитор обезбеђује узајамну искључивост, тј. само један процес у једном тренутку може бити активан у оквиру монитора. Ова особина гарантована је самим монитором, па програмер не мора да експлицитно програмира ово синхронизационо ограничење.

Монитор је по много чему сличан критичној секцији. Међутим, с обзиром на слабост конструкције **region** у односу на синхронизацију, потребни су додатни синхронизациони механизми. Ово се обезбеђује конструкцијом *condition*. За писање сопствених синхронизационих шема може се дефинисати једна или више променљивих типа *condition*. Једине две операције које се могу извршити над променљивама овог типа су *wait* и *signal*. Операција *x.wait* значи да се процес који изврши ову операцију суспендује све док неки други процес не изврши операцију *x.signal*. Ова операција обнавља тачно један суспендовани процес. Ако нема суспендованих процеса она нема ефекта.




Сл. 5.3. Шематски приказ монитора.

Претпоставимо да је, у тренутку када је *x.signal* извршено од стране процеса P, постојао процес Q придружен услову *x*. Јасно је да, ако се суспендованом процесу Q дозвољава да обнови своје извршење, процес P мора да чека. Иначе би и P и Q били истовремено активни у монитору. Овде постоје две могућности:

a) P чека док Q не напусти монитор, или чека још један услов.
b) Q чека док P не напусти монитор, или чека још један услов.

Постоје аргументи за прихватање алтернативе (а) и одбацивање алтернативе (б) и обрнуто. Пошто се P већ извршава у монитору (б) изгледа логичније. Међутим, ако допустимо да се P настави услов за настављање процеса Q не мора да важи када се дозволи да он настави своје извршење. Хоар је прихватио принцип (а), док је Бринч Хансен изабрао компромис. Када процес P изврши операцију *signal* он одмах напушта монитор, па према томе Q одмах иза тога наставља своје извршење. Ова шема је слабија од Хоарове пошто процес не може да сигнализира више од једном током позива процедуре.

Илуструјмо ове концепте на примеру монитора који симулира бинарни семафор.

```
type semaphore = monitor
var zauzet: boolean;
    slobodan: condition;
procedure entry P;
  begin
    if zauzet then slobodan.wait;
    zauzet:=true;
  end;
procedure entry V;
  begin
    zauzet:=false;
    slobodan.signal;
  end;
begin
  zauzet:=false;
end.
```

Запазимо да ако монитор M1 позива други монитор M2, узајамна искључивост у M1 се не ослобађа приликом преноса управљања на M2. Ова чињеница има две последице:

- Произвољан процес који позива M1 биће блокиран ван M1 за време извршавања M2.
- Ако процес уђе у ред услова у M2 може се јавити застој.

Овај проблем се зове *проблем угњежђених позива монитора*.

Када имамо више процеса суспендованих на основу услова *x* и изврши се *x.signal*, често FCFS алгоритам не задовољава (тј. да се обнови први процес из реда). Из тих разлога је Хоар (1974) увео конструкцију *conditional wait*

$$x.wait(c)$$

где је *c* целобројни израз који се израчунава када се *wait* операција извршава. Вредност променљиве *c*, која се назива *број приоритета*, са памти заједно са именом суспендованог процеса. Када се изврши операција *x.signal* процес са најмањим придруженим бројем приоритета се обнавља.

## 5.4 Синхронизација код Windows NT-а

Windows NT обезбеђује механизам синхронизације нити као део објектне архитектуре. Механизам који користи NT егзекутива за имплементацију средстава за синхронизацију чини фамилија синхронизационих објеката која садржи следеће типове објеката:

- Процес
- Нит
- Датотека
- Догађај
- Пар догађаја
- Семафор
- Тајмер
- Мутант

Прва три типа објеката имају у основи другачију намену али се такође могу користити и за синхронизацију. Остали типови објеката су специјално пројектовани да подрже синхронизацију.

Сваки примерак синхронизационог објекта може бити ули у *сигнализираном* или у *несигнализираном* стању. Нит може бити суспендована на објекту у несигнализираном стању, а ослобађа се када дотични објекат дође у сигнализирано стање. Механизам је очигледан: нит издаје захтев за чекањем NT егзекутиви путем хендла синхронизационог објекта. Када објекат пређе у сигнализирано стање NT егзекутива ослобађа све објекте нити који чекају на тај синхронизациони објекат. У табели 5.1 приказани су догађаји који изазивају прелазак сваког од типова објеката у сигнализирано стање и ефекти које тај прелаз има на нити које чекају.

*Пар догађаја* је објекат придружен клијент/сервер апликацији и могу му приступати само нити клијента и сервера. Било клијент било сервер могу да поставе објекат у сигнализирано стање што изазива да друга нит (нит сервера или нит клијента) буде сигнализирана. За све остале синхронизационе објекте више нити може да чека на једном објекту.

Таб. 5.1. Windows NT синхронизациони објекти.

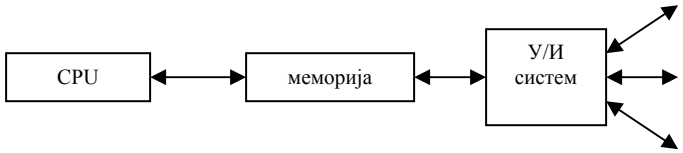
| Тип објекта | Дефиниција | Поставља се у сигнализирано стање када (се) | Ефекат на нити које чекају |
|---|---|---|---|
| Процес | Позивање програма обухватајући адресни простор и ресурсе | терминира последња нит | Све се ослобађају |
| Нит | Ентитет који се може извршити у оквиру процеса | нит терминира | Све се ослобађају |
| Датотека | Примерак отворене датотеке или У/И уређаја | У/И операција заврши | Све се ослобађају |
| Догађај | Објава да се јавио системски догађај | нит постави догађај | Све се ослобађају |
| Пар догађаја | Назнака да је посвећена клијентска нит копирала поруку Win32 северу или обрнуто | посвећена клијентска или серверска нит постави догађај | Друга посвећена нит се ослобађа |
| Семафор | Бројач који регулише број нити које могу користити ресурс | вредност семафора постане 0 | Све се ослобађају |
| Тајмер | Бројач који прати проток времена | Пристигло време које треба поставити или истекао интервал времена | Све се ослобађају |
| Мутант | Механизам који обезбеђује узајамну искључивост за Win32 или OS/2 окружење | Нит власник или друга нит ослободи мутанта | Једна нит се ослобађа |

Објекат *мутант* се користи да наметне узајмно искључив приступ неком ресурсу при чему се дозвољава да само један објекат нити може да добије приступ. Према томе, ова врста синронизационог објекта делује као бинарни семафор. Када објекат мутант уђе у сигнализирано стање само једна нит од оних које чекају се ослобађа. За остале синхронизационе објекте који подржавају чекање више нити све нити које чекају се ослобађају када објекат уђе у сигнализирано стање.

# 6 Управљање меморијом

Код система који могу да имају више процеса у меморији, ти процеси деле меморију као један од најважнијих ресурса рачунара. У овој глави ћемо размотрити различите начине за управљање меморијом. Алгоритми за управљање меморијом варирају од примитивног приступа "голе" машине до стратегија страничења и сегментације. Сваки од ових приступа има своје предности и мане.

Меморија је један од најбитнијих ресурса за рад модерних рачунарских система. Као што се може видети са слике 6.1, и CPU и У/И систем комуницирају са меморијом.




Сл. 6.1. Централни положај меморије у рачунарском систему.

Меморија је дугачак низ речи или бајтова који имају своје адресе. Интеракција са меморијом се одвија кроз низ читања или уписа у локације са одређеном адресом. CPU прибавља инструкције и податке из меморије и смешта податке у меморију.

У већини случајева кориснички програм пролази кроз неколико корака пре свог извршења. Адресе се могу представљати на различите начине у току тих корака. Адресе у изворном програму су у општем случају симболичке (називи променљивих). Компилатор обично повезује те симболичке адресе са релокатибилним адресама. Едитор веза или пунилац повезује те релокатибилне адресе са апсолутним адресама. Свако повезивање је пресликавање из једног адресног простора у други.

Програм се на крају преслика у апсолутне адресе и напуни у меморију ради извршења. Током извршења програма приступа се његовим инструкцијама и подацима генерисањем њихових апсолутних адреса. Када се програм заврши њему додељен меморијски простор се прогласи слободним и наредни програм може да се напуни и изврши.

Типичан циклус извршења инструкције ће, на пример, најпре прибавити инструкцију из меморије. Инструкција се декодира и по потреби прибављају операнди из меморије. После извршења инструкције над операндима резултати се смештају назад у меморију. Запазимо да јединица меморије види само низ меморијских адреса незнајући како су оне генерисане (који адресни начин рада је употребљен) и шта представљају (податке или инструкције). Према томе, можемо да занемаримо како програм генерише меморијске адресе, већ нас интересује сам низ меморијских адреса које генерише програм који се извршава.

## 6.1 Гола машина

До сада најједноставнији алгоритам за управљање меморијом је *непостојећи*. Корисник има на располагању голу машину и има потпуну контролу над читавим меморијским простором (слика 6.2). Овај приступ заиста има неке предности. Он обезбеђује максималну флексибилност кориснику и максимално је једноставан уз минималну цену. Не постоји потреба за посебним хардвером нити за системским софтвером.

Сл. 6.2. Гола машина.

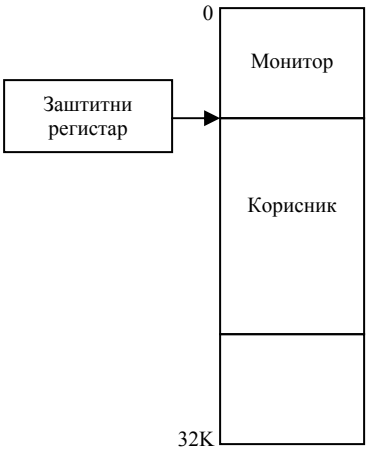
Међутим, овај систем има и озбиљне недостатке, јер не пружа никакве услуге. Корисник има потпуно управљање над рачунаром, али оперативни систем нема управљање над прекидима, нема резидентног монитора да обрађује системске позиве или грешке, и нема простора додељеног секвенцирању управљачких картица или послова.

Због својих ограничења се овај приступ користи само код специјалних

система где корисници захтевају флексибилност и и једноставност и где ће корисници желети да програмирају сопствене рутине за подрушку.

## 6.2 Резидентни монитор

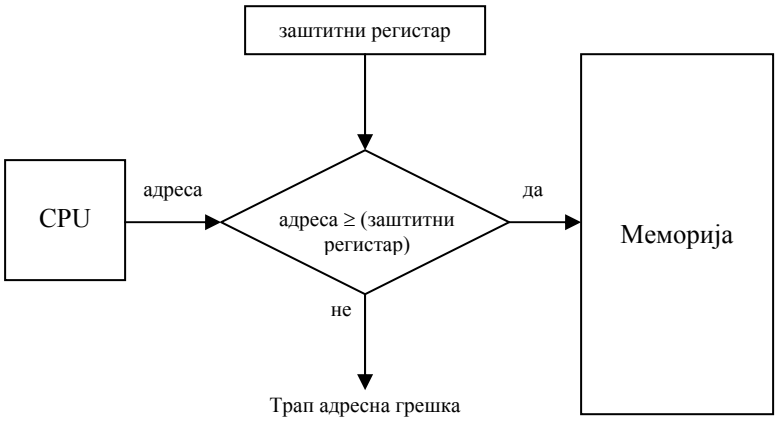
Следећа једноставна шема је да се меморија подели на два дела, један за корисника и један за резидентни монитор оперативног система (слика 6.3). Постоји могућност да се резидентни монитор смести или на почетак или на крај меморије. Главни фактор који утиче на одлуку о томе је у општем случају локација вектора прекида. Како је вектор прекида често на почетку меморије, обично се и резидентни монитор смешта на почетак меморије.




Сл. 6.3. Резидентни монитор.

### *6.2.1 Заштитни хардвер*

Ако се монитор налази у нижој меморијској области а кориснички програм који се извршава у вишој, морамо да заштитимо кôд и податке монитора од случајних или намерних измена од стране корисничког програма. Ову заштиту пружа хардвер и може се имплементирати на неколико начина. Општи приступ је приказан на слици 6.4. Свака адреса коју генерише кориснички програм се пореди са садржајем заштитног регистра. Уколико је она мања од садржаја заштитног регистра генерише се трап (адресна грешка) и оперативни систем предузима одговарајуће акције (обично се терминира кориснички програм уз издавање одговарајуће поруке о грешци).




Сл. 6.4. Хардверска адресна заштита за резидентни монитор.

Свака меморијска референца коју генерише кориснички програм мора да се провери што успорава приступ меморији. Оперативни систем који се извршава у мониторком начину рада има неограничени приступ и мониторској и корисничкој меморији.

Начини на које се имплементира заштитни регистар се разликују. Један приступ је да се у хардвер угради константна вредност заштитне адресе. Ово решење није флексибилно па се често користи посебан заштитни регистар који садржи заштитну адресу. Вредност у заштитном регистру може се мењати привилегованим инструкцијама у мониторском начину рада.

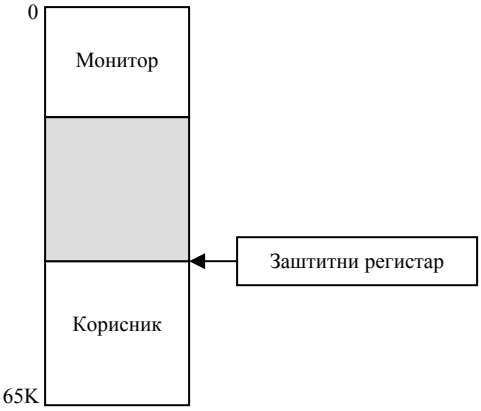
### *6.2.2 Релокација*

Следећи проблем који треба размотрити је пуњење корисничког програма. Иако адресни простор почиње од адресе 0, прва адреса у корисничком програму није 0, већ прва адреса после вредности у заштитном регистру. Ова промена може да утиче на адресе коришћене у корисничком програму. Повезивање инструкција и података са меморијским адресама може се извршити или у време компилације или у време пуњења. Ако је вредност заштитне адресе позната у време компилације може се генерисати *апсолутни кôд*.

Овај кôд ће почети од заштитне адресе и простираће се нагоре. Међутим, ако се вредност заштитне адресе промени потребно је поновно превођење програма. Алтернатива је да компилатор генерише *релокатибилни кôд*. Тада се повезивање одлаже до момента пуњења. У том случају, код промене заштитне адресе, потребно је само поновно пуњење програма.

У оба случаја, вредност заштитне адресе мора да буде *статичка* током извршења програма. Али, има случајева када је пожељно да се мења величина монитора током извршења програма, а тиме и заштитна адреса. Такав кôд се понекад назива *транзиентни* мониторски кôд.

Постоје два начина да се модификује основна шема да би се омогућила динамичка промена величине monitora. Operativni sitem računara PDP-11 користио је приступ приказан на слици 6.5. Кориснички програм се пунио у вишу меморију према граничној адреси. Неискоришћени простор у средини омогућавао је ширење било монитора било корисничког програма.

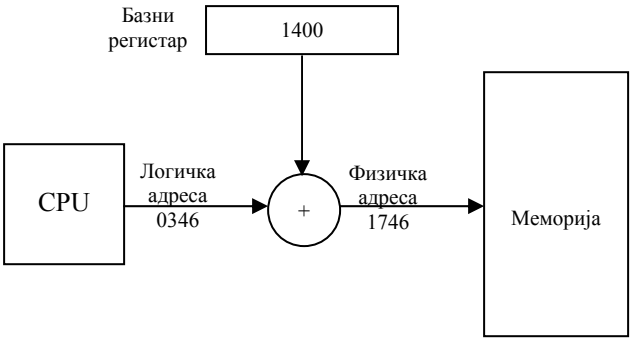



Сл. 6.5. Пуњење корисничког програма у вишу меморију.

Општији приступ, коришћен код рачунара CDC 6600, је да се одложи повезивање адреса до времена извршења. Ова шема динамичке реалокације захтева нешто другачију хардверску подршку, као што је приказано на слици 6.6. Заштитни регистар се сада зове *релокациони* или *базни* регистар. Вредност у базном регистру се додаје свакој адреси коју генерише кориснички процес у моменту када се пошаље у меморију. Уочимо да корисник никада не види стварне физичке адресе. Кориснички програм ради са *логичким* адресама, док хардвер за меморијско пресликавање претвара логичке адресе у физичке.

Код овог хардвера промена у вредности граничне адресе захтева једино измену базног регистра и премештање корисничке меморије на коректне локације у односу на нову вредност граничне адресе. Са једне стране, троши се значајно време за копирање меморије, али је могућа измена граничне адресе у сваком тренутку.

Приметимо да овде имамо два различита типа адреса: логичке (у опсегу од 0 до *max*) и физичке (у опсегу од *R*+0 до *R*+*max* ако је *R* вредност граничне адресе). Корисник генерише једино логичке адресе и мисли да програм ради на локацијама од 0 до *max*. Оперативни систем може да приступа физичкој меморији директно у мониторском режиму рада. Све информације које се преносе од корисничког програма оперативном систему морају се експлицитно релоцирати од стране оперативног система пре него што се употребе.




Сл. 6.6. Динамичка релокација која користи релокациони регистар.

Концепт логичког адресног простора који се пресликава у физички адресни простор је кључан за правилно управљање меморијом.

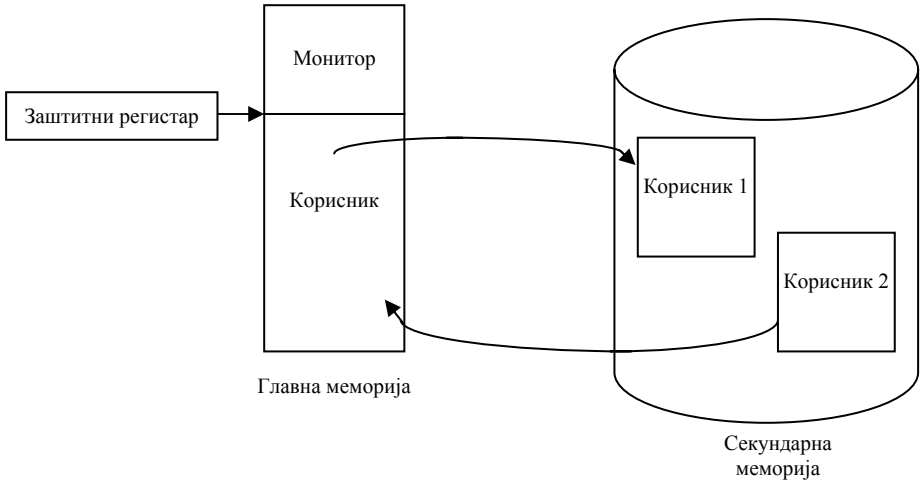
## 6.3 Пребацивање (swapping)

Управљање меморијом са резидентним монитором није много употребљиво обзиром да је по својој природи једнокорисничка шема. Ипак, то је била основна шема и у прва два тајм-шеринг система: CTSS и Q-32. Ови системи су користили резидентни монитор при чеми је остатак меморије био расположив кориснику који се тренутно извршавао. Када се прелазило на следећег корисника текући садржај корисничке меморије се уписивао на секундарни меморијски медијум (диск или добош) а меморија следећег корисника се учитавала. Ова шема се назива пребацивање или *swapping* (слика 6.7).

Пребацивање захтева брзу јединицу секундарне меморије, која је довољно велика да прихвати меморијске слке за све кориснике, а да је при том могућ директан приступ тим меморијским сликама. Ред спремних процеса се састоји од свих процеса чије су меморијске слике на секундарном медијуму и који су спремни за извршење. Посебна системска променљива указује на то који је процес тренутно у меморији. Када планер процеса одлучи да изврши неки процес он позива диспечера. Диспечер проверава да ли је тај

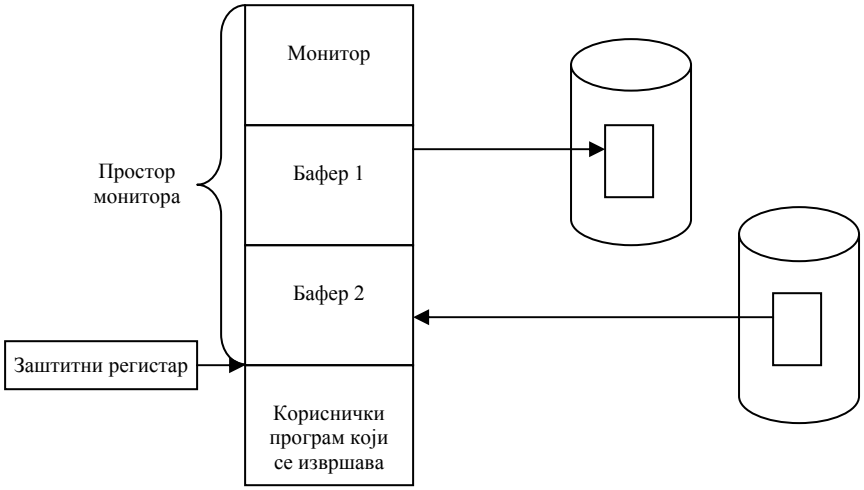
процес у меморији; ако није, пребацује процес који је тренутно у меморији на диск а учитава жељени процес. Потом се врши пуњење регистара као и обично и управљање преноси изабраном процесу.

Очигледно је да је време потребно за пребацивање контекста прилично велико. Да би искоришћеност CPU-а била боља пожељно је да времена трајања CPU активности процеса буду знатно дужа од времена пребацивања. Главна компонента времена потребног за пребацивање контекста је време преноса, а оно је директно пропорционално количини меморије која се пребацује. Стога се тежи да се пребацује само онај део меморије који је заиста потребан (дужина корисничког програма може бити мања него што је износ меморије намењен корисничким програмима). Осим овога, ефикасност пребацивања се побољшава са побољшањем перформанси јединица секундарних медијума.




Сл. 6.7. Пребацивање два корисника на секундарни меморијски медијум.

Ефекат пребацивања у односу на време пребацивања контекста се може даље побољшати преклапањем пребацивања и извршења програма. Посматрајмо слику 6.8. Циљ је да се преклопи пребацивање једног процеса са извршењем другог. Тако CPU није беспослен док траје пребацивање једног процеса.




Сл. 6.8. Преклопљено пребацивање.

Приметимо да пошто текући кориснички програм ослободи CPU, морамо да следећи кориснички програм померимо из бафера 2 у корисничку област меморије. Програм који се извршавао такође мора да се помери у један од бафера ради пребацивања.

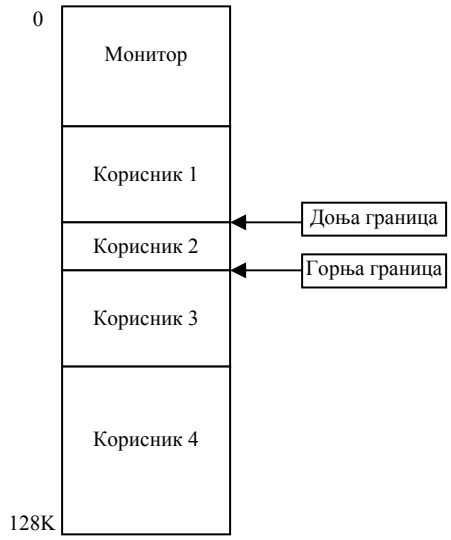
Ако се тако не поступи, можемо да извршавамо програм из бафера 2 померањем заштитне адресе. Претходни кориснички програм (у корисничкој области) је тада изложен неправилној модификацији од стране програма у баферу 2. Према томе, морамо да вршимо пребацивање унутар меморије.

## 6.4 Вишеструке партиције

Конфигурација меморије која је произишла из преклопљеног пребацивања је ефективно иста као и конфигурација меморије код мултипрограмирања, јер имамо више од једног програма у меморији у исто време. Код мултипрограмирања се CPU пребацује са једног на други од тих програма. Проблем који се тиче управљања меморијом је у додели меморије тим програмима када они треба да се изврше.

Меморија се дели на известан број области ил *партиција*. Свака област може да садржи један програм који се извршава. На тај начин степен мултипрограмирања је ограничен бројем области. Када је нека област слободна бира се програм из реда послова и пуни у слободну област. Када се програм заврши меморијска област постаје расположива за неки други програм.

Могуће су две основне шеме за управљање меморијом. Сваки од ова два приступа дели меморију на известан број области или партиција. Основна разлика између ова два приступа је у томе да ли су те области статичке или динамичке. Ове шеме су *додела меморије у непрекидним партицијама фиксне дужине* (MFT) и *додела меморије у непрекидним партицијама променљиве дужине* (MVT).




Сл. 6.9. Два гранична регистра који дефинишу логички адресни простор.

Као и раније, потребно је обезбедити заштиту кôда и података из једне области од програма из друге области. Како сада имамо више корисничких програма у меморији то је потребно да имамо могућност да заштитимо меморијски простор пре и после програма који се извршава. Ова заштита се може обавити коришћењем два регистра, као на слици 6.9. Та два регистра садрже горњу и доњу границу адреса које кориснички програм може легално генеристати, и могу се дефинисати на два начина:

- **Гранични регистри**. Вредности најмање и највеће физичке адресе. Легалне адресе налазе се унутар тих граница.
- **Базни и гранични регистар**. Вредност најмање физичке адресе и граница опсега логичких адреса. Легалне корисничке адресе су у опсегу од 0 до границе опсега које се динамички релоцирају на физичке адресе у опсегу од базне адресе до збира базне адресе и границе опсега логичких адреса.

### *6.4.1 Партиције фиксне дужине (MFT)*

Послови који пристижу у систем смештају се у ред спремних послова. Планер послова води рачуна о меморијским потребама сваког посла и расположивим меморијским областима како би одредио којим пословима да додели меморију. Када се послу додели меморијски простор, он се пуни у одговарајућу партицију. Он се тада може надметати за добијање CPU-а. Када се посао заврши, он ослобађа претходно додељену меморијску област која се онда може напунити другим послом из реда послова.

Код доделе меморије могуће су различите стратегије. Код једне од њих се сви послови приликом уласка у систем класификују према меморијским захтевима. Ове захтеве може да специфицира сам корисник или се могу одређивати аутоматски. Свака партиција има сопствени ред послова у коме су послови чији меморијски захтеви одговарају величини дотичне партиције. Саме партиције могу бити међусобно различите по величини.

Други приступ је да се сви послови смештају у један ред. Планер послова бира следећи посао који ће се извршити и чека док се меморисјка област одговарајуће величине не ослободи. Очигледна варијанта ове шеме је да се спречи да меморијске партиције буду празне. Тако се посао који чека да се ослободи њему одговарајућа партиција прескаче уколико постоји слободна партиција која одговара неком другом послу. Овде се поставља питање да ли посао треба да чека на ослобађање њему најприближније партиције по величини ако постоји већа расположива партиција, или да се тај посао напуни у ту партицију при чему смањујемо чекање али трошимо знатно више меморијског простора него што је потребно. Ова дилема представља избор између стратегија најбољег поклапања (*best-fit-only*) и најбољег расположивог поклапања (*best-available-fit*).

Ако посао захтева *m* меморијских речи он се може извршити у партицији величине *n* речи ($n \geq m$). Разлика *n* - *m* представља *интерну фрагментацију*. *Екстерна фрагментација* се јавља када је нека партиција слободна али је превише мала за све послове који чекају, па тако увек остаје неискоришћена. Оба типа фрагментације се јављају код MFT стратегије.

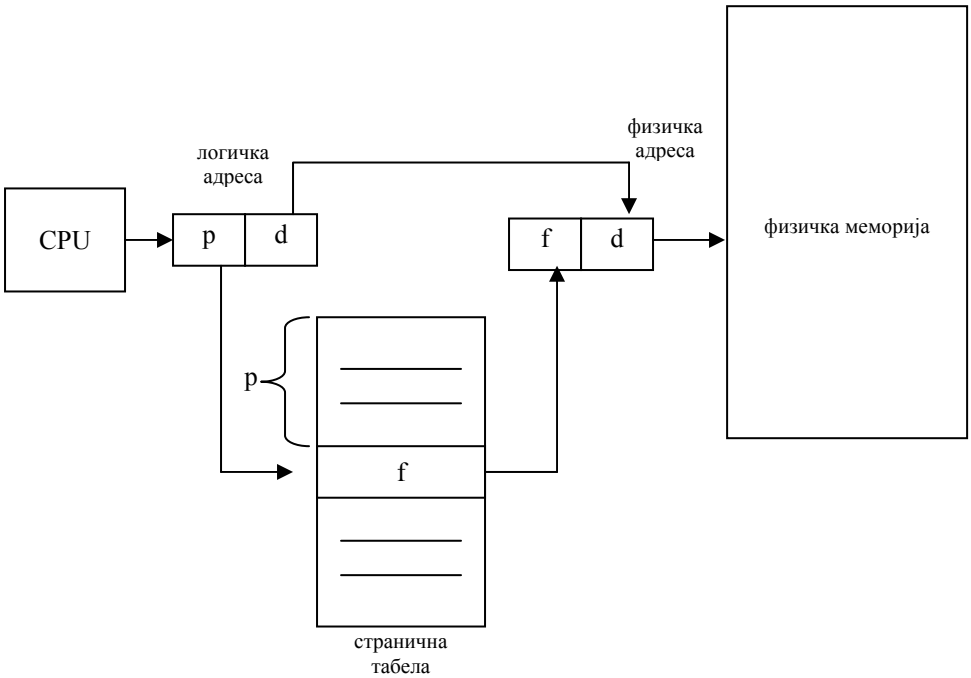
### *6.4.2 Партиције променљиве дужине (MVT)*

Основни проблем код MFT је одређивање најбоље величине партиција како би се минимизирале интерна и екстерна фрагментација. На жалост, са динамичким скупом послова који треба да се изврше највероватније не постоји оптимална подела меморије.

Решење овог проблема је да се дозволи динамичка промена величине меморијских партиција. Овај приступ се назива додела меморије у вишеструким непрекидним партицијама променљиве дужине. Управљање меморијом на основу MVT стратегије је прилично једноставно. Оперативни систем креира табелу која указује који делови меморије су слободни а који заузети. На почетку је читава меморија расположива за корисничке порграме и посматра се као један велики блок расположиве меморије који се назива *рупа*. Када наиђе посао коме треба меморија тражи се довољно велика рупа за тај посао. У случају да се она пронађе, послу се додељује само онолико меморије колико му је потребно, док се остатак рупе оставља расположивим за будуће захтеве. Када се посао заврши он ослобађа меморију, која тада постаје рупа. Ако је та нова рупа суседна другој рупи оне се уједињују у једну већу рупу. Најчешћи алгоритми за доделу меморије су *прво поклапање* (*first-fit*) и *најбоље поклапање* (*best-fit*). Приметимо да је разлика између MFT и MVT у софтверу, док је хардвер идентичан.

Још један проблем се јавља код MVT. Код јако малих рупа је потребно више меморије за одговарајућу ставку у табели него што је сама величина рупе. У том случају се, код доделе меморије која оставља мали део неке рупе слободним, уместо проглашавања малих остатака меморије за рупе, оне додељују том захтеву правећи мали износ фрагментације.

Искоришћеност меморије је у општем случају боља код MVT него код MFT. Међутим, постоји проблем екстерне фрагментације који може бити веома изражен. Једно решење овог проблема је *компакција*. Циљ је да меморија преуреди тако да се све слободне партиције сместе заједно у један велики блок. Приметимо да компакција није увек могућа. Ако се релокација врши статички комапакција није могућа. Она је могућа само код динамичке релокације, у време извршења, коришћењем базног и граничног регистра.




Сл. 6.10. Хардвер потребан код страничења.

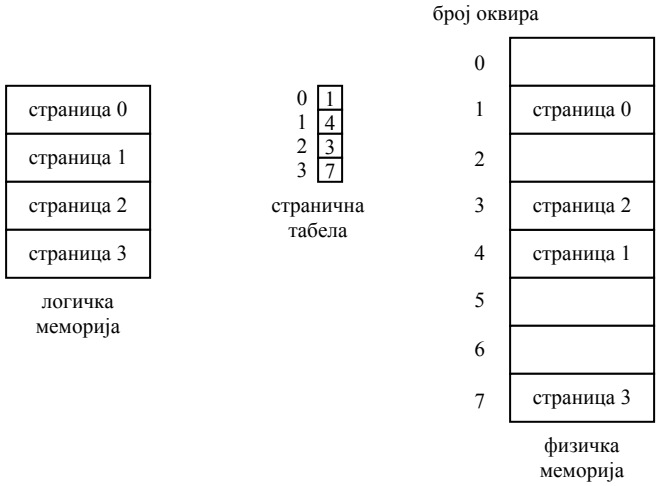
## 6.5 Страничење

MVT шема показује проблем екстерне фрагментације. Генерално, ова ситуација се јавља када расположива меморија није континуална већ је подељена на више расејаних блокова. Како меморија која се додељује неком послу мора да буде континуална, такав меморијски простор се не може искористити. Компакција је један од начина да се превазиђе овај проблем. Међутим, *страничење* дозвољава да меморија једног програма буде некотинуална, чиме се омогућује да се програму додељује физичка меморија где год је она расположива.

Хардверска подршка страничењу је илустрована на слици 6.10.

Свака адреса коју генерише CPU дели се на два дела: *број странице* (*p*) и *странични офсет* (*d*). Број странице се користи као индекс у *страничној табели*. Странична табела садржи базне адресе сваке странице у физичкој меморији. Ова базна адреса се комбинује са страничним офсетом да би се дефинисала физичка меморијска адреса која се шаље јединици меморије.

Странични модел меморије је приказан на слици 6.11. Физичка меморија је подељена у блокове фиксиране величине који се називају *оквири*. Логичка меморија се такође подељена на блокове исте величине који се називају *странице*. Када се програм извршава његове странице се пуне у било који расположиви оквир, а странична табела врши транслацију корисникових страница у меморијске оквире.




Сл. 6.11. Странични модел логичке и физичке меморије.

Величина странице, као и оквира, дефинисана је хардверски и обично је неки степен броја два. Ако је величина странице $P$, тада логичка адреса $U$ даје број странице $p$ и офсет $d$ на следећи начин:

$$p = U \textbf{ div } P$$
$$d = U \textbf{ mod } P$$

Избор степена двојке за величину странице чини транслацију логичке адресе у број странице и офсет веома једноставним. Ако је величина странице $2^n$, тада нижих $n$ битова представља офсет а преостали виши битови број стране.

Када посао пристигне у систем, планер послова испитује његову величину. Величина посла изражена је у страницама. Планер послова испитује расположиве меморијске оквире. Ако посао захтева $n$ страница, онда $n$ страничних оквира мора да буде слободно. Прва страница посла пуни се у један од додељених оквира, а број оквира се смешта у страничну табелу за тај посао. Следећа страна се пуни у следећи оквир итд.

Код страничења нема екстерне фрагментације. Сваки слободни оквир може да буде додељен. Међутим, постоји извесна интерна фрагментација, јер обично последњи оквир додељен послу не мора да буде пун.

Размотримо хардверску имплементацију страничне табеле. У најједноставнијем случају она се имплементира као скуп специјализованих регистара. Диспечер CPU-а пуни те регистре баш као што пуни и остале регистре. Инструкције које пуне или модификују страничну табелу су привилеговане и само оперативни систем може да их изврши.

Употреба регистара за имплементацију страничне табеле је задовољавајућа ако је странична табела релативно мала. Код машина код којих то није случај се странична табела држи у меморији, а *базни регистар страничне табеле* (*Page Table Base Register – PTBR*) указује на страничну табелу. Измена страничне табеле захтева једино измену тог регистра чиме се смањује време пребацивања контекста. Проблем код овог приступа је време потребно за приступање локацијама корисничке меморије. Ако желимо

да приступимо локацији *i*, морамо најпре да приступимо страничној табели користећи PTBR и индекс који одговара страници за *i*. Ово значи да морамо да приступимо меморији. Када из страничне табеле добијемо број оквира који се комбинује са офсетом стране можемо да приступимо жељеној локацији у меморији. Према томе за приступ некој меморијској локацији потребно је обавити два приступа меморији. Тиме се меморија практично успорава два пута.

Стандардно решење овог проблема је да се користи специјална хардверска меморија малог капацитета, која се обично назива *асоцијативна меморија*. Скуп асоцијативних регистара сачињава меморију велике брзине која има могућност истовременог претраживања садржаја свих регистара. Оваква меморија јесте веома брза, али је и веома скупа па мора бити релативно малог капацитета. Асоцијативна меморија садржи свега неколико ставки из страничне табеле и то оне најактуелније. Када CPU генерише логичку адресу, одговарајућа страна се тражи у асоцијативној меморији. Ако се она налази у њој, одговарајући број меморијског оквира је одмах расположив и може се приступити жељеној локацији у меморији. У случају промашаја морамо да ипак потражимо жељену ставку у делу страничне табеле која је у меморији.

Проценат успешног проналажења жељене ставке у асоцијативној меморији назива се фактор поготка (*hit ratio*). Ако је фактор поготка 80%, а време претраживања асоцијативне меморије 50ns и време приступа меморији 750ns, тада можемо израчунати *ефективно време приступа меморији* као

$$t_{ef} = 0.80*(50+750) + 0.20*(50+750+750).$$

Још једна предност страничења је могућност дељења заједничког кôда. Уколико је кôд тзв. *reentrant* типа, тј. не врши самомодификацију, онда је могуће да више корисника користи исту копију тог кôда у меморији. Такви програми су едитори, компилатори, асемблери, системи база података итд.

Заштита меморије код страничења врши се заштитним битовима који су придружени свакој страници. Ти битови се чувају у страничној табели. Једним битом се може дефинисати да ли се дотична страница може и читати и модификовати или само читати. Како свака меморијска референца иде преко страничне табеле ради одређивања броја меморијског оквира, том приликом се и проверава да ли се тражи упис у страницу која је смочитајућа. Још један бит се обично придружује свакој ставци у страничној табели. Овај бит служи да се изазове трап уколико је генерисана илегална адреса (valid/invalid bit). Оперативни систем поставља вредности за све странице да би дозволио или забранио приступ појединим страницама.

## 6.6 Сегментација

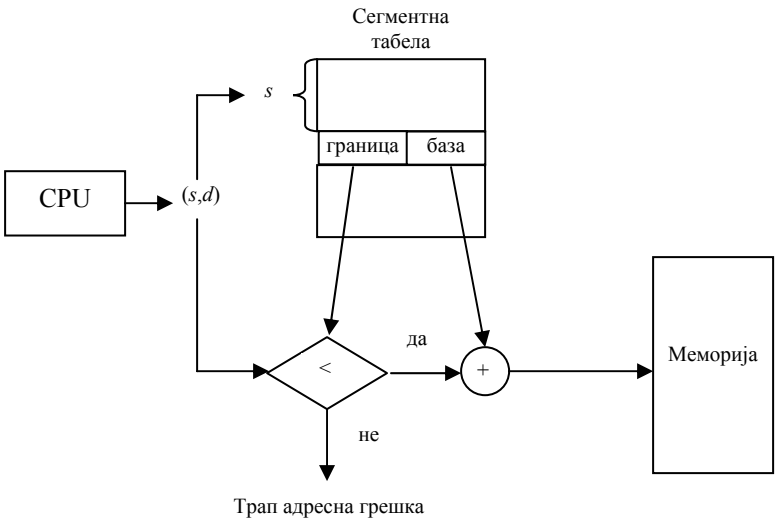
Код страничења је неизбежно да се раздваја кориснички поглед на меморију и стварна физичка меморија. Корисников поглед на меморију није исти као стварна физичка меморија, већ се само пресликава на њу. То пресликавање омогућава разлику између логичке и физичке меморије.

Опште је мишљење да корисници не виде меморију као линеаран низ речи, већ као скуп сегмената различите дужине. *Сегментација* је шема за управљање меморијом која подржава такав кориснички поглед на меморију. Логички адресни простор је колекција сегмената. Сваки сегмент има име и дужину. Адресе специфицирају како име сегмента, тако и офсет унутар сегмента. Према томе, корисници специфицирају сваку адресу помоћу две величине: имена сегмента и офсета (код страничења су корисници специфицирали само једну адресу која се хардверски делила на број стране и офсет , што је било невидљиво за програмера). Ради једноставности, сегменти се референцирају бројем сегмента а не именом сегмента.

Иако сада корисник указује на објекте у програму помоћу дводимензионалне адресе, физичка меморија је још увек једнодимензионални низ речи. Због тога се мора дефинисати имплементација пресликавања дводимензионалне корисничке адресе у једнодимензионалну физичку адресу. Ово пресликавање је реализовано преко *сегментне табеле*.

Употреба сегментне табеле је илустрована на слици 6.12. Логичка адреса се састоји од два дела: броја сегмента *s* и офсета унутар сегмента *d*. Број сегмента се користи као индекс у сегментној табели. Свака ставка у сегментној табели има базну адресу сегмента и границу сегмента. Офсет *d* у логичкој адреси мора да има вредност између 0 и границе. Иначе се генерише трап адресна грешка. Уколико је офсет легалан он се сабира са базном адресом сегмента чиме се добија адреса жељене речи у физичкој меморији.

Као и странична табела, и сегментна табела може бити смештена у асоцијативној меморији или у оперативној меморији. Када је број сегмената релативно велики није изводљиво читаву сегментну табелу држати у асоцијативним регистрима, већ се она смешта у меморију. *Базни регистар сегментне табеле* (*Segment Table Base Register -STBR*) ukazuje na segmentnu tabelu. Kako broj segmenata koje koristi program može da varira, koristi se i *registar dužine segmentne tabele* (*Segment Table Length Register – STRL*). За логичку адресу (*s*,*d*) се најпре проверава да ли је број сегмента *s* легалан ($s < STRL$). Потом се број сегмента додаје садржају *STBR*-а чиме се добија меморијска адреса одговарајуће ставке у сегментној табели. Та се ставка чита из меморије и проверава офсет да ли је у оквиру граница сегмента, а затим се израчунава физичка адреса жељене речи као сума базне адресе сегмента и офсета.




Сл. 6.12. Хардвер за подршку сегментацији.

Као и код страничења, ово пресликавање захтева два обраћања меморији по једној логичкој адреси што успорава рад рачунарског система два пута, уколико се нешто не предузме. Обично се користи скуп асоцијативних регистара који садрже најактуелније ставке сегментне табеле. Релативно мали број ових регистара (8 до 16) може смањити време приступа меморији тако да оно не буде дуже више од 10-15% него код приступа меморији без пресликавања.

Нарочита предност сегментације је што је заштита меморије придружена сегментима. Како сегменти представљају семантички дефинисане делове програма, вероватно је да ће се све ставке сегмента користити на исти начин. Према томе, неки сегменти садрже инструкције а неки сегменти садрже податке. У модерним рачунарским архитектурама нема инструкција које су самомодификујуће, тако да се сегменти који садрже инсрукције могу дефинисати као самочитајући или самоизвршавајући. Хардвер за меморијско пресликавање ће проверити заштитни бит придружен свакој ставци сегментне табеле да би се заштитили од илегалних приступа меморији.

Још једна предност сегментације је могућности дељења кôда или података. Сваки посао има своју сегментну табелу придружену његовом PCB-у а коју користи диспечер да дефинише табелу хардверских сегмената када се посао даје CPU-у. Сегменти се деле када ставке у сегментним табелама два различита посла указују на исту физичку локацију. Дељење се јавља на нивоу сегмената, тако да се могу делити и програми који се састоје од више сегмената.

Планер послова треба да нађе и додели меморију за све сегменте корисничког програма. Ова ситуација је слична страничењу осим што су сегменти променљиве дужине. Као и код MVT, додела меморије представља проблем динамичке доделе меморије, и може се решити коришћењем *best-fit* или *first-fit* алгоритама. Сегментација може да изазове екстерну фрагментацију када су сви блокови слободне меморије исувише мали за смештај сегмента. У том случају посао може једноставно да сачека док се довољно меморије не ослободи или се може извршити компакција. Како је сегментација по својој природи алгоритам за динамичку релокацију меморије, компакција се може вршити у сваком тренутку.

И страничење и сегментација имају своје предности и недостатке. Могуће је комбиновати ове две шеме како би се побољшале њихове карактеристике. Тако се добијају шеме *сегментирано страничење* и *странична сегментација*.

# 7 Виртуелна меморија

У последњој глави разматрали смо различите стратегије меморије које се користе у рачунарским системима. Све ове стратегије имају исти циљ: држање више процеса у меморији истовремено ради мултипрограмирања. Међутим, све оне захтевају цео процес у меморији како би се он извршио.

Виртуелна меморија је техника која омогућава да се извршава процес који није у потпуности у меморји. Главна видљива предност ове шеме је да кориснички програми могу да буду већи од физичке меморије. Није лако имплементирати овај концепт, а како то може озбиљно да смањи перформансе, треба га користити опрезно.

## 7.1 Оверлеји

Испитивања реалних програма су показала да у многим случајевима није потребан читав програм у меморији. На пример:

- Програми често имају кôд за руковање неуобичајеним грешкама. Пошто су те грешке ретке, тај део кôда се готово никад не извршава.
- Низовима, листама и табелама се често додељује више меморије него што им је заиста потребно.
- Неке опције и карактеристике програма се могу ретко користити.

Чак и у оним случајевима када је цео програм потребан, не мора да значи да је читав програм потребан у исто време.

Способност да се извршава програм који је само делимично у меморији има многе предности:

- Програм више није ограничен износом физичке меморије која је расположива у рачунару. Корисници су у стању да пишу програме за веома велики *виртуелни* адресни простор, олакшавајући програмирање.
- Како сваки корисник може да заузме мањи износ физичке меморије, више корисника се може извршавати истовремено, што одговара повећању искоришћености CPU-а и пропусне моћи, али не и времена одзива или времена обиласка.
- Мање У/И активности је потребно за пуњење или пребацивање корисника у меморију, тако да ће се сваки корисник извршавати брже.

Видимо да извршавање програма који нису у целости у меморији има својих предности и за систем и за кориснике. Техника позната као *оверлеји* (*overlays*) се понекад користи да би се омогућило да програми буду већи од износа меморије који им се додељује. Идеја оверлеја је да се у меморији држе само оне инструкције и подаци који су потребни у неком задатом времену. Када су друге инструкције потребне оне се пуне у ону меморију у којој су биле неке друге инструкције које сада више нису потребне.

Размотримо као пример двопролазни асемблер. Током првог пролаза асемблер конструише табелу симбола а у другом пролазу генерише кôд на машинском језику. Могли би смо да поделимо такав асемблер у кôд првог пролаза, кôд другог пролаза, табелу симбола и заједничке рутине за подрушку које користе и први и други пролаз. Претпоставимо да су величине ових компонената:

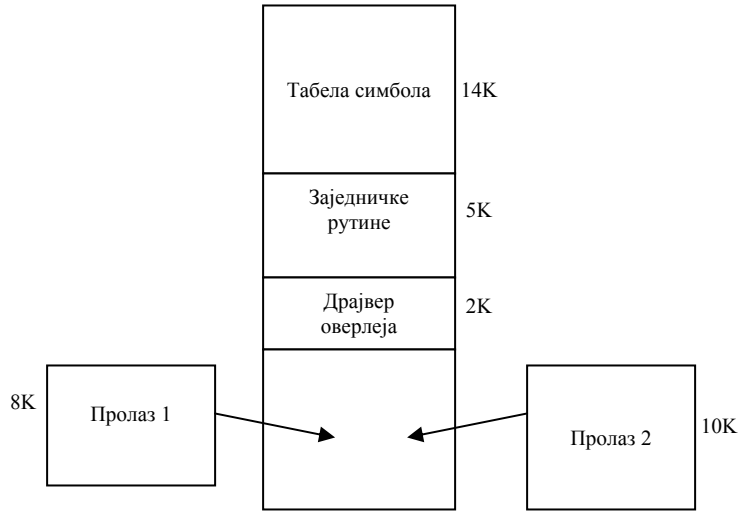
| | |
|---|---|
| Први пролаз | 8К |
| Други пролаз | 10К |
| Табела симбола | 14К |
| Заједничке рутине | 5К |

Да би напунили све то одједном у меморију потребно нам је 37К. Ако је само 32К расположиво не можемо да извршимо наш програм. Међутим, приметимо да пролаз 1 и пролаз 2 не морају да буду у меморији истовремено. Дефинисаћемо два оверлеја: (А) табела симбола, зајединичке рутине и пролаз један, и (В) табела симбола, заједничке рутине и пролаз 2.

Додаћемо и драјвер оверлеја (величне 2К) и почети са оверлејем А у меморији. Када завршимо са првим пролазом скочимо на драјвер оверлеја који учитава оверлеј В у меморију преко оверлеја А и преноси управљање другом пролазу. Оверлеју А треба 29К, а оверлеју В 31К (слика 7.1). Сада можемо да покренемо

наш асемблер у 32К расположиве меморије. Међутим, радиће нешто спорије због додатних У/И операција које се тичу учитавања кôда оверлеја B преко кôда оверлеја A.

Кôд за оверлеје A и B налази се на диску или траци као слика апсолутне меморије а драјвер оверлеја их учитава када су потребни. Потребан нам је посебан алгоритам за релокацију и повезивање.




Сл. 7.1. Оверлеји за двопролазни асемблер.

Појам који је сродан овоме јесте *динамичко пуњење*. Код динамичког пуњења се рутине не пуне у меморију док се не позову. Све рутине се налазе на диску у релокатибилном формату. Главни програм се пуни у меморију и извршава. Када рутина жели да позове неку другу рутину, онда се најпре проверава да ли је позвана рутина напуњена. Ако није, релокатиблни пунилац се позива да би се жељена рутина учитала у меморију и ажурирају одговарајуће табеле да би се одсликало ново стање. Потом се управљање преноси на новонапуњену рутину. Предност динамичког пуњења је у томе да се рутине које нису потребне никада не пуне у меморију. Ова шема је нарочито корисна када је дугачак кôд потребан за руковање случајевима који се ретко јављају.

Ни оверлеји ни динамичко пуњење не траже посебну подршку оперативног система. Обе технике се могу у потпуности имплементирати од стране корисника, једино што ће бити више У/И операција него обично. С друге стране корисник порграмер мора правилно да пројектује и програмира структуру оверлеја. Овај задатак може да представља тежак подухват. Стога су пожељније аутоматске технике за извршавање великих програма на ограниченим капацитетима физичке меморије.

## 7.2 Страничење на захтев

Виртуелна меморија описује скуп техника које дозвољавају да се извршавају програми који нису у потпуности у меморији. *Страничење на захтев* (*demand paging*) је најраспрострањенији систем виртуелне меморије.

Страничење на захтев је слично страничењу са пребацивањем (слика 7.2). Програм се налази на секундарном меморијском медијуму. Када желимо да се изврши он се пребацује у меморију. Уместо да се читав програм смести у меморију, смештају се само оне странице које су неопходне. На овај начин се смањује време пребацивања, смањује неопходни капацитет физичке меморије а повећава степен мултипрограмирања.

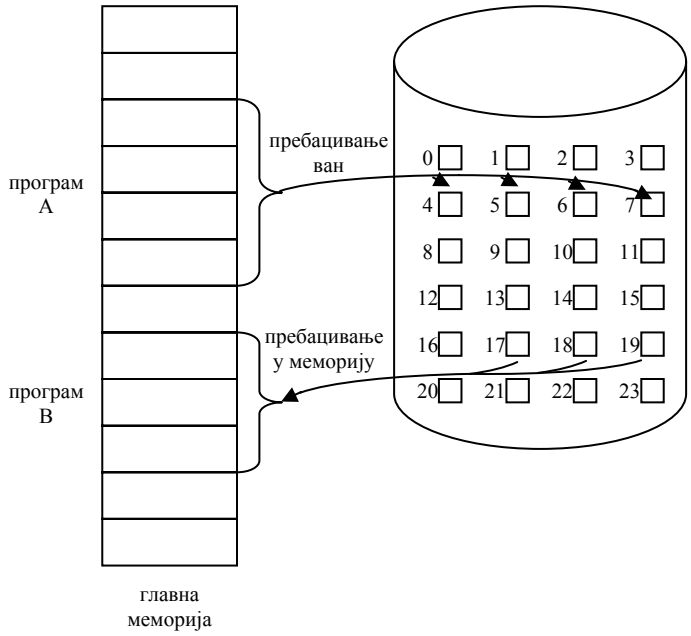
Међутим, шта се дешава када програм покуша да користи страницу која није у меморији. Хардвер ће покушати да преведе корисничку адресу у физичку користећи страничну табелу. Ставке у страничној табели које одговарају страницама које нису у меморији и којима није придружен странични оквир имају постављену вредност валид/инвалид бита којом се спречава приступ (слика 7.3).

Ово инвалидирање неће имати ефекта ако програм никада не покуша да приступи тим страницама. Према томе, ако добро проценимо и у оперативну меморију пребацујемо само оне и све заиста потребне странице, програм ће радити на исти начин као и када су све странице у меморији.

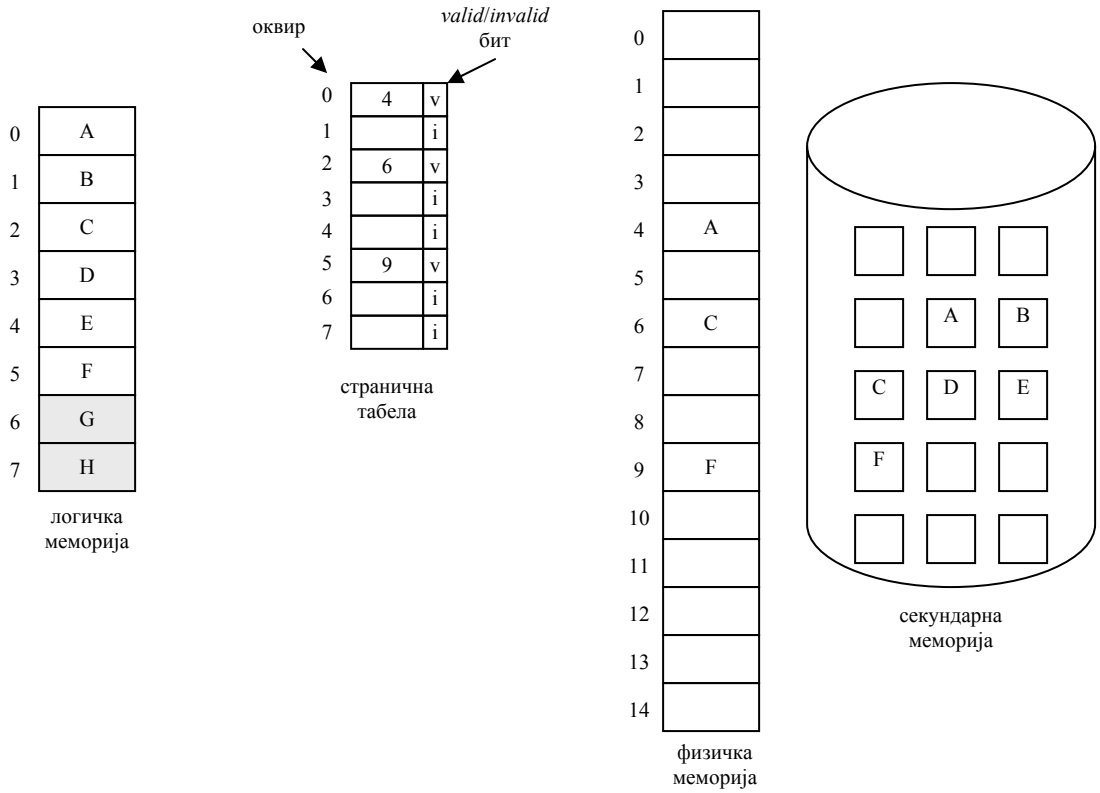
Наравно, ако је та процена лоша и програм покуша да приступи страници која није у меморији, јавља се промашај странице. Хардвер за страничење ће приликом превођења адресе преко страничне табеле

установити да је постављен инвалид бит и изазвати трап типа адресна грешка. Нормално би се због овог типа грешке програм терминирао. У овом случају трап је реултат пропуста оперативног система да допреми валидни део програма у меморију.

Према томе, треба да исправимо овај пропуст. Процедура је сасвим проста (слика 7.4.)

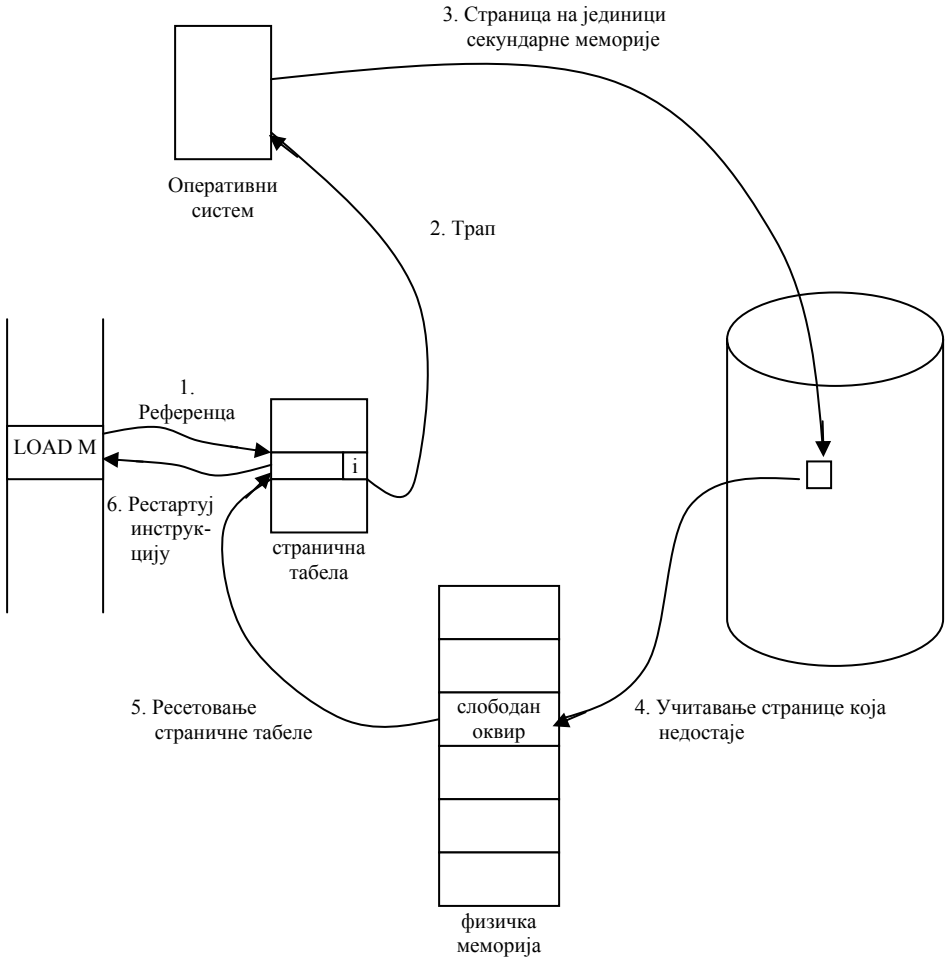



Сл. 7.2. Пребацивање страничне меморије на континуални простор на диску.




Сл. 7.3. Странична табела када неке странице нису у меморији.

1. Најпре се проверава интерна табела која се чува заједно са PCB-ом датог процеса да би се одредило да ли дотична меморијска референца представља валидан приступ меморији. Уколико то није случај, програм се терминира.
2. Ако се ради о валидној референци али та страница није још у меморији, она мора да се учита у меморију.
3. Проналази се слободан меморјиски оквир.
4. Налаже се учитавање жељене странице са диска у новододељени оквир.
5. Када се читање диска заврши, модификује се интерна табела и странична табела да би указивале да је та страница сада у меморији.
6. Рестартује се инструкција која је прекинута појавом трапа. Процес сада може да приступи страници као да је одувек била у меморији.




Сл. 7.4. Кораци приликом опслуживања промашаја странице.

Важно је запазити да, захваљујући памћењу стања прекинутог процеса, тај процес можемо обновити на потпуно истом месту и у истом стању, осим што је сада жељена страница у меморији. Ово нам омогућава да програм извршавамо и када он није у потпуности у меморији.

У екстремном случају можемо започети извршење програма када ниједна страница није у меморији. Процес ће одмах произвести промашај странице већ првом инструкцијом. Када се прва страница допреми у меморију процес ће се наставити, изазивајући промашај стране све док му све потребне стране не буду у меморији. Од тог момента програм се може извршавати без промашаја. Ово се назива *чисто страничење на захтев*: никада се не учитава страница у меморију док не буде потребна.

Хардвер који подржава страничење на захтев је исти као за страничење и пребацивање:

- Странична табела са могућношћу означавања сваке ставке инвалидном помоћу *valid*/*invalid* бита или посебном вредношћу заштитних битова.

- Секундарни меморијски медијум за чување страница које нису у меморји. То је обично брзи диск.

Осим ове хардверске, потребна је и значајна софтверска подршка.

## 7.3 Перформансе страничења на захтев

Страничење на захтев може да има значајни утицај на перформансе рачунарског система. Да би видели зашто, изразимо ефективно време приступа код страничења на захтев.

$$t_{a_{ef}} = (1\text{-}p)t_a + pt_f$$

Овде је $p$ фактор промашаја, $t_a$ време приступа меморији а $t_f$ је време које се утроши приликом промашаја странице (учитавање потребне странице и приступ тој страници).

Да би могли да израчунамо ефективно време приступа потребно је да знамо колико се времена утроши за опслуживање промашаја странице. Промашај странице изазива појаву следеће секвенце.

1) Трап оперативног система.
2) Памћење регистара и стања програма.
3) Одређивање да ли је прекид изазван због промашаја странице.
4) Провера да ли је странична референца била легална и одређивање локације странице на диску.
5) Налагање читања са диска у слободан меморијски оквир.
   a) Чекање у реду уређаја док се захтев за читање опслужује.
   b) Чекање током времена постављања и/или времена ротационог кашњења диска.
   c) Отпочињање трансфера странице у слободан меморијски оквир.
6) Док чекамо можемо доделити CPU другом процесу.
7) Прекид изазван од стране јединице диска (У/И завршен).
8) Памћење регистара и стања програма другог корисника
9) Одређивање да ли је прекид изазван од стране диска.
10) Кориговање страничне табеле и других табела да би показивале да је жељена страница сада у меморији.
11) Чекање да се CPU поново додели процесу.
12) Обнављање регистара, стања програма и нове страничне табеле, а онда настављање прекинуте инструкције.

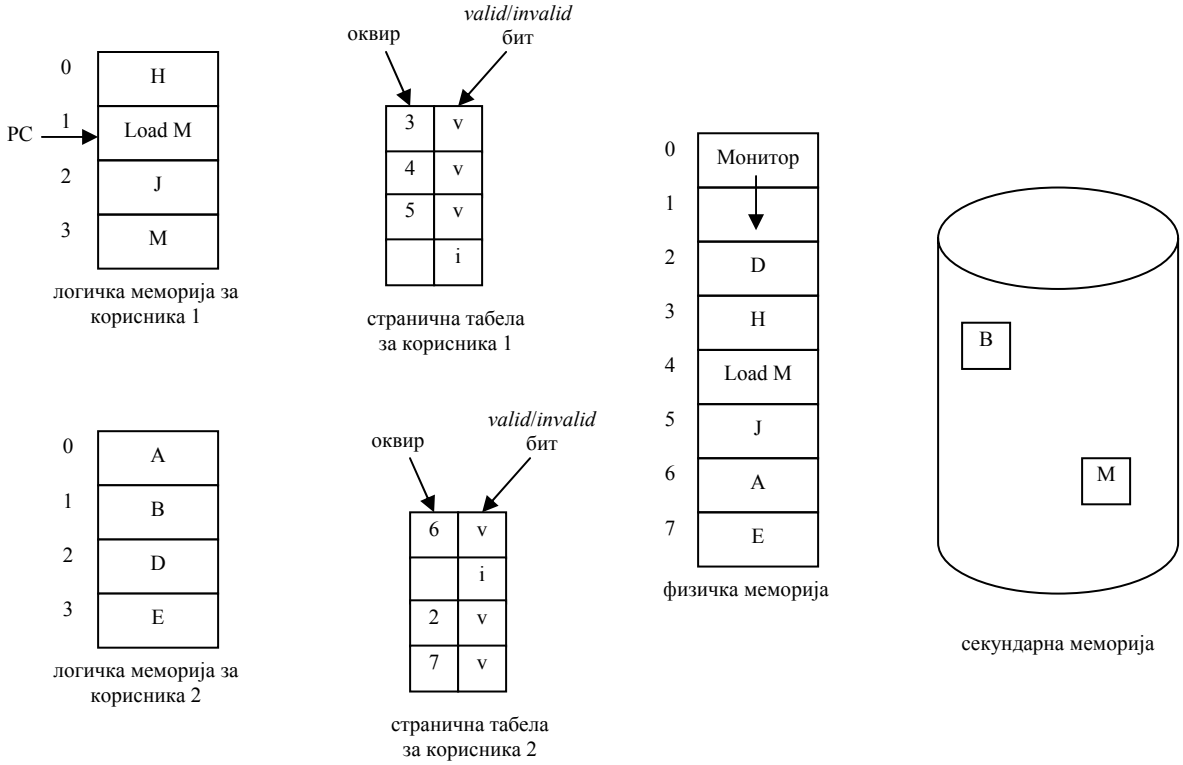
Не морају сви ови кораци да буду неопходни у свим случајевима. У сваком случају суочени смо само са три главне компоненте времена опслуживања промашаја странице.

- Опслуживање прекида изазваног промашајем странице.
- Учитавање странице.
- Обнављање процеса.

## 7.4 Замена странице

У досадашњим разматрањима фактор промашаја није представљао проблем, јер се промашај сваке стране јављао највише једном, када се она први пут референцира. Међутим, ово није сасвим тачно.

Ако желимо да повећамо степен мултипрограмирања онда се врши тзв. *предодељивање* меморије где стално имамо већи број процеса у меморији са свега неколико страница, уместо да дозволимо да се неки процес у потпуности учита у меморију.




Сл. 7.5. Ситуација када је потребна замена странице.

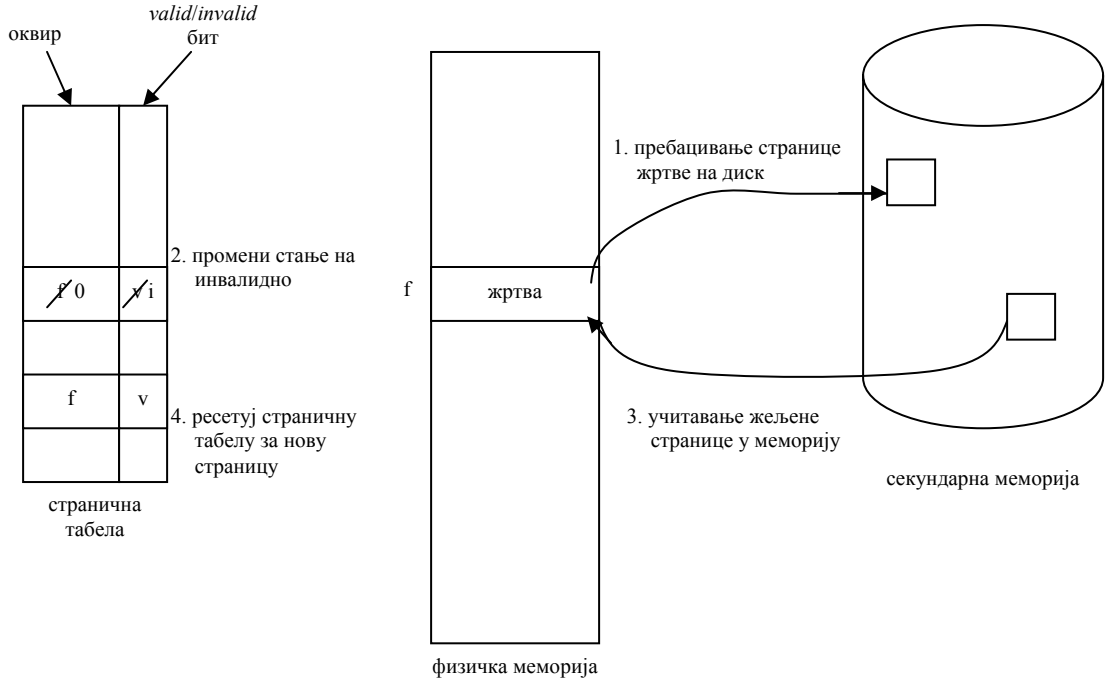
У таквом случају се дешава да у тренутку када је потребно учитати нову страницу нема слободног меморијског оквира (слика 7.5). Оперативни систем тада има неколико могућности. Једна је да терминира кориснички програм. Међутим, страничење на захтев је ствар коју оперативни систем ради да би побољшао искоришћеност и пропусну моћ рачунарског система. Корисници не треба да буду свесни да се њихови програми извршавају у страничном систему. Страничење треба да буде логички транспарентно кориснику, тако да ова могућност није најбољи избор.

Можемо да читав програм пребацимо на диск, ослобађајући све његове оквире и тиме смањимо степен мултипрограмирања. То може да буде добра идеја, али постоји још занимљивија могућност: *замена странице*.

Замена странице представља следећи приступ. Ако нема слободног оквира, налази се онај који се тренутно не користи и ослобађа се. Ослобађање оквира може се вршити уписивањем његовог садржаја на диск и изменом страничне табеле (и других табела) како би указали да се та страница не налази више у меморији (слика 7.6). Ослобођени оквир се сада може користити да прими страницу због које је дошло до промашаја.

Рутина за опслуживање промашаја странице је сада измењена да обухвати замену странице.

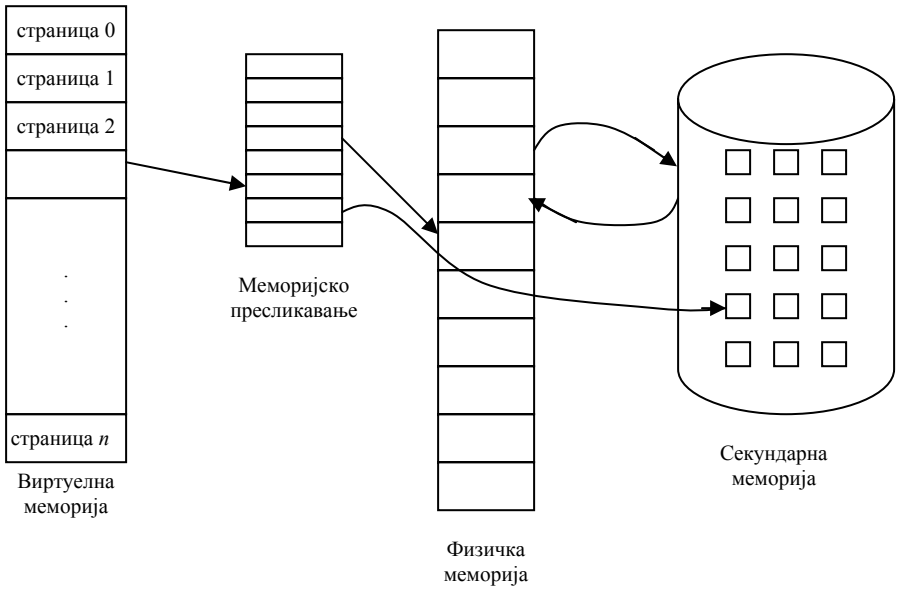
1) Нађи локацију жељене странице на диску.
2) Нађи слободан оквир.
   a) Ако постоји слободан оквир, користи га.
   b) Ако не, употреби алгоритам за замену странице за избор оквира *жртве*.
   c) Упиши жртву на диск и сагласно томе измени табелу страница и оквира.
3) Учитај жељену страницу у новооослобођени оквир и ажурирај табеле страница и оквира.
4) Обнови кориснички процес.




Сл. 7.6. Замена странице.

Приметимо да су потребна два преноса странице ако нема слободног оквира. Оваква ситуација удвостручује време промашаја странице а тиме се повећава ефективно време приступа меморији.

Да би се ово време смањило користи се *dirty* bit. Овај бит означава да ли је било шта из оквира измењено од последњег учитавања странице. У случају да страница жртва није модифкована није потребно њено уписивање на диск.




Сл. 7.7. Виртуелна меморија може бити много већа од физичке.

## 7.5 Концепти виртуелне меморије

Замена странице је суштинска за страничење на захтев. Она комплетира раздвајање логичке од физичке меморије. Код страничења које није на захтев корисничке адресе се пресликавају у физичке, дозвољавајући да та два скупа адреса буду сасвим различита. Код страничења на захтев величина логичког адресног простора није више ограничена физичком меморијом.

Виртуелна меморија представља раздвајање корисничке логичке меморије од физичке и обично се имплеметира страничењем на захтев. На тај начин се програмерима може обезбедити велика виртуелна меморија на системима са мањом физичком меморијом (слика 7.7).

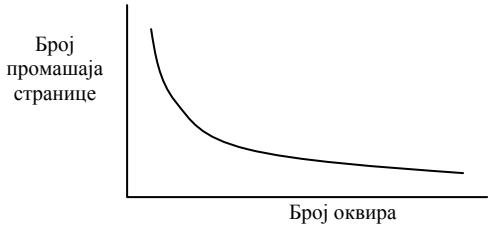
Виртуелна меморија чини задатак програмирања лакшим, јер програмер не мора више да брине о капацитету расположиве физичке меморије, већ се може сконцентрисати на проблеме који се програмирају. Једна од последица виртуелне меморије је виртуелно нестајање оверлеја.

Виртуелна меморија се такође може имплементирати и код система са сегментацијом (сегментација на захтев) али су алгоритми замене страница сложенији због променљиве дужине сегмената.

Два главна проблема се морају решити када се имплементира страничење на захтев: алгоритам за доделу оквира и алгоритам за замену страница. Ако имамо више процеса у меморији морамо одлучити колико оквира да доделимо сваком процесу. Када је потребна замена странице, треба да изаберемо оквир који ће се заменити.

## 7.6 Алгоритми замене страница

Постоји више различитих алгоритама за замену страница. У општем случају, најбољи је онај који даје најмањи фактор промашаја. Евалуација алгоритама се врши њиховим извршавањем на неком низу меморијских референци и рачунањем броја промашаја страница.




Сл. 7.8. Број промашаја странице у зависности од броја оквира.

Да би смањили број потребних података приметићемо да нам је, уз фиксирану величину странице, потребно да знамо број странице који се референцира. Потребно је да знамо и број страничних оквира који је расположив. Како број оквира расте, смањује се број промашаја (слика 7.8).

### *7.6.1 FIFO*

Најједноставнији алгоритам је FIFO (*First-In-First-Out*). Код овог алгоритма се свакој страници придружује податак о времену када је она учитана у меморију. Када је потребна замена странице, бира се најстарија страница. Приметимо да није неопходно баш пратити време, већ се може креирати FIFO ред у коме су странице из меморије.

FIFO алгоритам је једноставна за разумевање и за програмирање. Међутим, његове перформансе нису увек добре. Страница која се замењује може бити неки модул за иницијализацију који је одавно искоришћен и више није потребан. С друге стране, она може да садржи веома често коришћену променљиву која је одавно иницијализирана и налази се у сталној употреби.

Лош избор при замени повећава фактор промашаја и успорава извршење програма, али не изазива некоректно извршење.

Код FIFO алгоритма је могуће да се за исти низ референци са већим бројем оквира добије већи фактор промашаја. Ова појава се назива *Беладијева аномалија*.

### *7.6.2 Оптимална замена*

Један од резултата открића Беладијеве аномалије био је трагање за оптималним алгоритмом који би имао најмањи могући фактор промашаја и који никада не би патио од Беладијеве аномалије. Овакав

алгоритам постоји и може се просто исказати на следећи начин: Замени ону страницу која се неће користити током најдужег периода времена.

На жалост, овај алгоритам је тешко имплементирати, јер захтева познавање будућег низа референци. Стога се обично користи ради поређења.

### *7.6.3 LRU*

Када већ оптимални алгоритам није изводљив можда је могућа његова апроксимација. Кључна разлика између FIFO алгоритма и ОПТ је да FIFO користи време када је страница учитана у меморију, док ОПТ користи време када ће страница бити коришћена. Ако користимо недавну прошлост као апроксимацију блиске будућности, тада можемо да заменимо страницу која није била коришћена током најдужег периода времена. Овај алгоритам се назива LRU (*the Least Recently Used*).

LRU свакој страници придружује протекло време од последње употребе. Код замене странице за жртву се бира она која најдуже није коришћена. Ово је оптимални алгоритам са инверзијом времена.

LRU се често користи као алгоритам замене страница и сматра се прилично добрим. Највећи проблем је како га имплементирати, јер се захтева значајна хардверска подршка. Могућа су два начина имплементације:

- Бројачи. Свакој ставци у страничној табели придружује се регистар са временом коришћења а CPU-у бројач који се инкрементира са сваком меморијском референцом. Кад год се изврши обраћање некој страници, садржај бројача се копира у регистар у одговарајућој ставци страничне табеле. Замењује се она страница са најмањом вредношћу у регистру.
- Магацин. Користи се магацин бројева страница. Када се страница референцира она се уклања из магацина и смешта на врх. Тако су на врху магацина најскорије коришћене странице а на дну најмање недавно коришћене. Како се странице уклањају из магацина, најбоље је магацин имплементирати као двоструко повезану листу са указатељима на почетка и крај.

Ни ОПТ ни LRU не пате од Беладијеве аномалије. Постоји класа алгоритама за замену страница која се назива *магацински алгоритми*, и који не показују Беладијеву аномалију. Магацински алгоритам је алгоритам за који се може показати да је скуп страница у меморији за *n* оквира увек подскуп скупа страница које би биле у меморији за *n*+1 оквир.

### *7.6.4 Апроксимације* LRU *алгоритма*

Мали број система пружа довољну хардверску подршку за прави LRU алгоритам. Неки системи не пружају хардверску подршку уопште па се морају применити други алгоритми, као што је FIFO.

Многи системи пружају извесну помоћ у виду референтног бита. Тај бит се за неку страницу поставља хардверски када се страница референцира. Не можемо знати редослед референцирања али знамо које странице су коришћене а које нису. Ова делимична информација о редоследу води ка више алгоритама замене страница који покушавају да апроксимирају LRU замену страница.

**<u>Додатни референтни битови</u>**

Један бајт се придружује свакој страници у табели. У правилним интервалима се податак о референцирању уписује у најстарији бит тог бајта уз претходно померање удесно садржаја тог бајта

**<u>Замена друге шансе</u>**

У основи ради се о FIFO алгоритму али се консултује референтни бит и замена врши ако је његова вредност 0. Ако ја његова вредност 1, страници се пружа друга шанса а референтни бит се брише.

**<u>LFU (Least Frequently Used)</u>**

Свакој страници се придружује бројач који броји референце. Замењује се страница са најмањом вредношћу бројача.

**<u>MFU (Most Frequently Used)</u>**

Ова варијанта се заснива на аргументу да је страница са најмањим бројачем највероватније тек учитана у меморију и још није коришћена. Ни MFU ни ЛФУ се не користе често, јер је њихова имплементација прилично скупа.

**<u>Класе страница</u>**

Посматрају се референтни бит и *dirty* бит као уређени пар који дели странице у четири класе.

0. (0,0) ни коришћена ни модификована
1. (0,1) није коришћена (скоро) али је модификована

2. (1,0) коришћена али није модификована
3. (1,1) коришћена и модификована

Замењује се произвољна страница из најниже непразне класе. Може се користити FIFO као секундарни алгоритам или се избор врши случајно.

## 7.7 Управљање меморијом код Windows NT-а

Windows NT је пројектован тако да се може имплементирати на различитим процесорима. Једна од најважнијих платформи за Windows NT је Intel 80486 (и следбеници) која је део фамилије процесора који се користе за изградњу IBM и њима компатибилних персоналних рачунара. Windows је усвојио величину странице од 4kB која је коришћена код 486 као основ за механизам виртуелне меморије. Извршићемо преглед средстава за подршку виртуелној меморији код 486.

### *7.7.1 Адресни простори*

Почев од 386 Intel фамилија садржи хардвер који подржава и сегментацију и страничење. Оба механизма могу да се искључе што омогућује кориснику да изабере један од четири различита погледа на меморију:

- *Несегментирана нестраничена меморија*: У овом случају виртуелна адреса је иста као и физичка што може бити корисно у једноставним контролерским апликацијама са високим перформансама.
- *Несегментирана страничена меморија*: Овде се меморија види као линеарни адресни простор издељен на странице. Заштита и управљање меморијом врше се на основу страничења. Овај приступ је фаворизован код неких оперативних система, као што је Unix.
- *Сегментирана нестраничена меморија*: Овде се меморија види као колекција логичких адресних простора. Предност овог приступа у односу на страничење је да омогућује заштиту све до нивоа бајтова, уколико је то неопходно. Осим тога, насупрот страничењу, гарантовано је да ће неопходна сегментна табела бити у самом чипу када је сегмент у меморији. Према тома, сегментирана нестраничена меморија резултира предвидљивим временом приступа.
- *Сегментирана страничена меморија*: Сегментација се користи за дефинисање логичких меморијских партиција које су подложне управљању приступом, док се страничење користи за доделу меморије унутар партиција. Оперативни систем који фаворизује овакав поглед је OS/2.

### *7.7.2 Сегментација*

Када се користи сегментација свака виртуелна адреса (која се назива логичком адресом у документацији фамилије Intel) састоји се од 16-битне референце сегмента и 32-битног офсета. Два бита сегментне референце односе се на механизам заштите док осталих 14 битова специфицира одређени сегмент. Према томе, са несегментираном меморијом корисничка виртуелна меморија је $2^{32}$ = 4GB. Са сегментираном меморијом меморијом укупан виртуелни меморијски простор који корисник види је $2^{46}$ = 64TB. Физички адресни простор захтева 32-битну адресу за максималних 4GB.

Износ виртуелне меморије може у ствари да буде већи од 64TB. Интерпретација виртелне адресе код 80386 зависи од тога који процес је тренутно активан. Половина виртуелног адресног простора (8k сегмената × 4GB) је глобална меморија дељива између свих процеса, док је остатак локалан и различит за сваки процес.

Сваком сегменту су придружена два облика заштите*: ниво привилегије* и *атрибут приступа*. Постоје четири нивоа привилегије од најзаштићенијег (ниво 0) до најмање заштићеног (ниво 3). Ниво привилегије прифружен сегменту података представља његову *класификацију*; ниво привилегије придружен програмском сегменту је његов *клиренс*. Програм који се извршава може да приступи само оним сегментима података за које је његов клиренс мањи (привилегованији) или једнак нивоу привилегије тог сегмента података.

Хардвер не диктира начин коришћења ових нивоа привилегије већ та употреба зависи од дизајна и имплементације оперативног система. Намера је била да ниво привилегије 1 користи већи део оперативног система, док је ниво привилегије 0 намењен мањем делу оперативног система који је намењен управљању меморијом, заштити и контроли приступа, док су преостала два нивоа остављена за апликације. У многим системима апликације остају на нивоу привилегије 3 док ниво 2 остаје неискоришћен. Подсистеми специјализованих апликација које морају да се штите, јер имплементирају сопствене безбедносне механизме, представљају добре кандидате за ниво 2. Неки примери су системи за управљање базама података, системи за аутоматизацију администрације и околине софтверског инжењерства.

Осим што регулише приступ сегментима података, механизам привилегије ограничава употребу неких инструкција. Неке инструкције, као што су оне које раде са регистрима за управљање меморијом, могу се

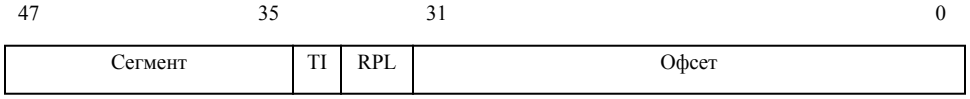
извршавати само на нивоу 0. У/И инструкције се могу извршавати само до одређеног нивоа означеног оперативним системом (обично је то ниво 1).

Атрибут приступа сегмента података специфицира када је приступ типа *Read*/*Write* тј. *Read-Only* дозвољен. За програмски сегмент атрибут приступа специфицира *Read*/*Execute* или *Read*/*Only* приступ.

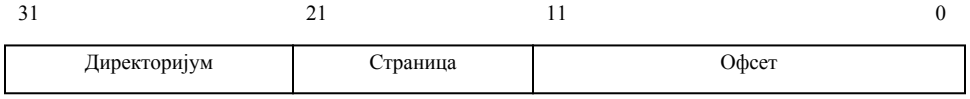
Механизам превођења адреса ради сегментације обухвата пресликавање виртуелне адресе у линеарну адресу (слика 7.10). Формат виртуелне адресе (слика 7.9) обухвата следећа поља:

- *Индикатор табеле* (TI): Указује да ли се глобална или локална сегментна табела користи за превођење адреса.
- *Број сегмента*: Служи као индекс у сегментној табели.
- *Офсет*: Офсет адресираног бајта у оквиру сегмента.
- *Захтевани ниво привилегије* (RPL): Ниво привилегије који се захтева да би се приступило сегменту.

Свака ставка у сегментној табели састоји се од 64 бита (слика 7.11). Поља су дефинисана у табели 7.1.




Сл. 7.9. Формат виртуелне адресе код Intel 486.




Сл. 7.10. Формат линеарне адресе код Intel 486.

Сл. 7.11. Ставка сегментне табеле код Intel 486.

Таб. 7.1. Параметри управљања меморијом код Intel 486.

| | |
|---|---|
| **Ставка сегментне табеле** | |
| Граница | Дефинише величину сегмента. Процесор интерпретира ово поље на два начина, зависно од бита грануларности: у јединицама од једног бајта до границе од 1MB, или у јединицама од 4kB до границе од 4GB. |
| База | Дефинише почетну адресу сегмента унутар 4GB линеарног адресног простора. |
| Бит приступа (A) | Поставља се када се приступи сегменту. Оперативни систем који користи сегментирану нестраничену меморију може да, у сврху управљања меморијом, користи овај бит за праћење фреквенције коришћења сегмента. |
| Тип | Прави разлику између различитих врста сегмената и указује на атрибуте приступа. |
| Дескриптор нивоа привилегије (Dpl) | Специфицира ниво привилегије сегмента на који указује ова ставка сегментне табеле. |
| Бит присутности (P) | Користи се код нестраничених система. Указује да ли је сегмент присутан у главној меморији. За системе са страничењем његова вредност је увек 1. |
| Бит грануларности (G) | Указује да ли се поље границе интерпретира у јединицама од 1 бајт или 4kB. |
| **Ставка директоријума страничне табеле и страничне табеле** | |
| Адреса оквира странице | Физичка адреса странице у емморији ако је бит присутности постављен. Како се странични оквири поравнавају на 4K, нижих 12 битова су 0 док се виших 20 налазе у овом пољу. |
| Адреса страничне табеле | Физичка адреса страничне табеле у меморији ако је бит присутности постављен. |
| Бит присутности (P) | Указује да ли је странична табела или страница у главној меморији. |
| Бит приступа (A) | Процесор поставља овај бит на оба нивоа страничне табеле када се јаве операције читања или уписа у дотичну страницу. |
| *Dirty* бит (D) | Процесор поставља овај бит када се јави операција уписа у дотичну страницу. |
| Корисник/супервизор бит (US) | Указује да ли је страница расположива само за оперативни систем (супервизорски ниво) или је расположива и за оперативни систем и за апликације (кориснички ниво). |
| *Read*/*Write* бит (RW) | За странице на корисничком нивоу указује када је приступ страници *Read-only* односно *Read/Write* за програме на корисничком нивоу. |
| Расположиви битови (AVAIL) | Битови расположиви програмеру система. |

### *7.7.3 Страничење*

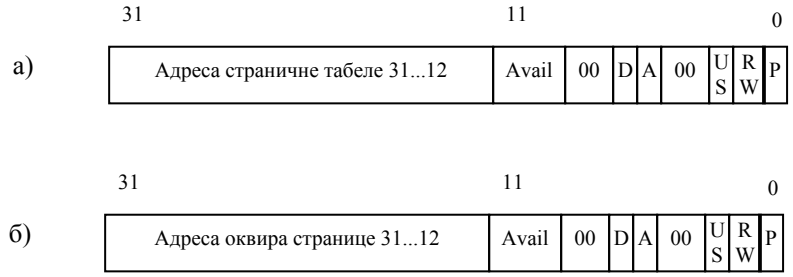
Сегментација је опциона и може се искључити. Када је у употреби, адресе које се користе у програму представљају виртуелне адресе које се преводе у линеарне адресе као што је то раније описано. Када сегментација није у употреби у програму се користе линеарне адресе. У оба случаја следећи корак је превођење линеарне адресе у 32-битну адресу.

Да би се разумела структура линеарне адресе треба да знамо да је механизам страничења код 80386 у ствари операција над двонивовском табелом. Први ниво је *директоријум страница* и има 1024 ставке. Овим се линеарна меморија од 4GB дели у 1024 страничне групе од којих свака има своју страничну табелу и где је свака група величине 4MB. Свака странична табела се састоји од 1024 ставке где свака од њих одговара једној страници величине 4kB. Логика за управљање меморијом има могућност да користи један странични директоријум за све процесе, један директоријум за сваки од процеса или неку комбинацију претходна два приступа. Странични директоријум за текући задатак је увек у главној меморији. Странична табела може бити и у виртуелној меморији.

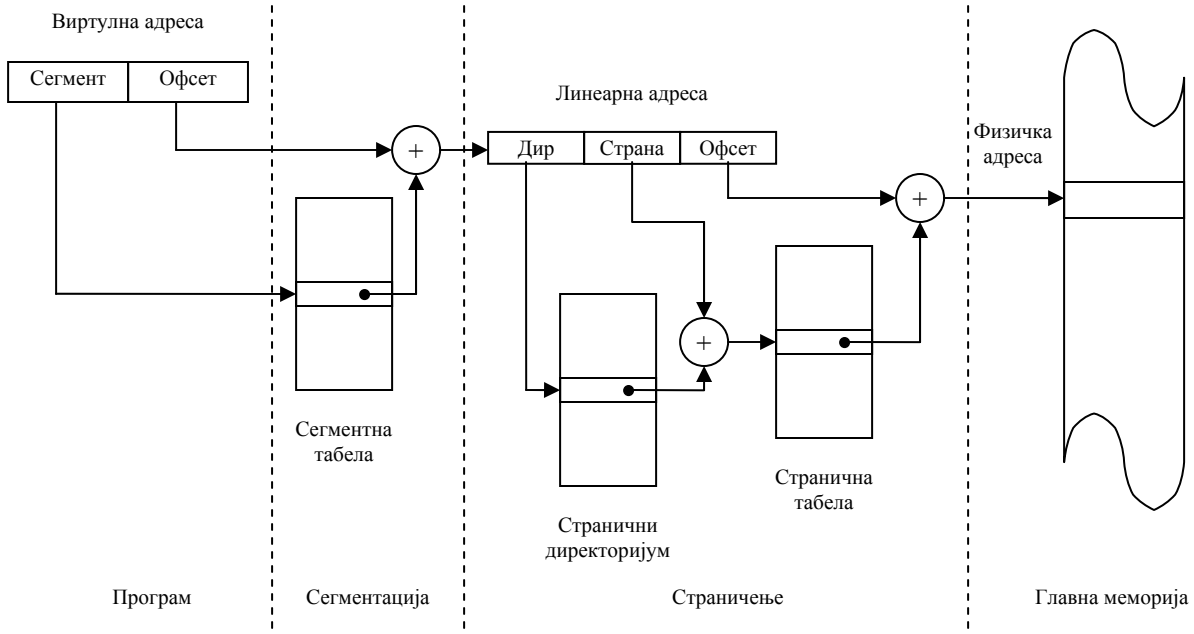
Слика 7.12 приказује формат ставки у страничном директоријуму и страничној табели док су поља дефинисана у табели 7.1. Запазимо да се механизам за управљање приступом може обезбедити над страницама и над групама страница.

80386 користи TLB (*Translation Lookaside Buffer*) који може да садржи 32 ставке страничне табеле. Свакипут када се промени странични директоријум овај бафер се обрише.

Слика 7.13 илуструје комбинацију механизама сегментације и страничења. Ради једноставности TLB и механизам кеш меморије нису приказани.




Сл. 7.12. а) Ставка директоријума страничне табеле. б) Ставка страничне табеле.




Сл. 7.13. Механизам адресне транслације код Intel 486.

# 8 Управљање дисковима

У системима са мултипрограмирањем више процеса може захтевати приступ диску, тако да ред уређаја који одговара диску обично није празан. Када се један захтев опслужи треба изабрати захтев из реда чекања који ће се следећи опслужити. Постоји више алгоритама по којима се то може урадити.

- FCFS (*First-Comed-First-Served*). Лак је за програмирање и по правилу фер, али у просеку не даје најбоље опслуживање.
- SSTF (*Shortest-Seek-Time-First*). У основи представља облик SJF алгоритма и као и он може изазвати умирање од глади.
- SCAN. Глава за упис/читање полази са једног краја диска и, крећући се ка другом крају, опслужује захтеве оним редоследом којим наилази на њих. Када се стигне до краја диска, глава се опет креће ка супротном крају, опслужујући захтеве. Назива се још и *алгоритма лифта*.
- C-SCAN (Circular-SCAN). Варијанта SCAN-а где кад се стигне до краја, глава се одмах враћа на почетак без опслуживања захтева.
- LOOK, C-LOOK. Варијанте SCAN-а и C-SCAN-а код којих се глава не креће до краја диска, већ само до последњег захтева у датоме смеру.

# 9 Застоји

У околини са мултипрограмирањима се више програма надмеће за коначан број ресурса. Процес захтева ресурс и ако он у том тренутку није расположив, процес прелази у стање чекања. Може се десити да процеси који чекају никада више не промене своје стање, јер су тражени ресурси већ додељени другим процесима који чекају. Таква ситуација се назива застој. Ради спречавања застоја или опоравка од њега систем може да предузима релативно екстремне акције.

## 9.1 Проблем застоја

Проблем застоја није само својствен оперативним системима, већ се може јавити и у свакодневној ситуацији. На пример, код прелажења преко реке преко камења на које се може ступити само једном ногом. Камење се може сматрати ресурсима. Ступање на камен представља заузимање ресурса, а склањање ноге са камена одговара ослобађању ресурса. Када се две особе које прелазе реку са супротних обала сусретну на средини, долази до застоја. Ова ситуација се може разрешити ако се нека од особа врати назад, на обалу са које је почела прелазак реке. У оперативним системима ово се назива *rollback*. Уколико више особа, једна за другом, прелазе реку са исте стране, потребно је да се све оне врате назад. Кад год особа започиње прелазак реке без провере да ли неко покушава да пређе реку са супротне обале, застој је увек могућ.

Скуп процеса се налази у стању застоја када сваки од процеса чека на догађај који може изазвати неки други процес из тог скупа. Догађаји о којима је реч јесу заузимање и ослобађање ресурса.

## 9.2 Карактеризација застоја

Јасно је да су застоји непожељни, јер ниједан процес не довршава своје извршење а ресурси система су заузети, спречавајући друге послове чак и да се започну.

### *9.2.1 Потребни услови*

Застој се може јавити ако и само ако су у систему испуњена следећа четири услова

- Узајамно искључивање. Најмање један ресурс се налази у недељивом режиму рада, тј. само један процес истовремено може да га користи. Ако други процес захтева тај ресурс, он мора да чека док ресурс не постане слободан.
- Држи и чекај. Мора да постоји процес који држи бар један ресурс и који чека да заузме додатне ресурсе које тренутно држе други процеси.
- Нема прекидања. Ресурси се не могу ослобађати осим ако процес који их заузима није завршио свој задатак.
- Кружно чекање. Мора да постоји скуп процеса $\{p_0, p_1, \ldots, p_n\}$ који чекају, таквих да $p_0$ чека на ресурс који заузима $p_1$, $p_1$ чека на ресурс који заузима процес $p_2$, …, $p_{n-1}$ чека на ресурс који заузима $p_n$ и $p_n$ чека на ресурс који заузима $p_0$.

Напоменимо да сва четири услова морају да се испуне да би се јавио застој. Кружно чекање повлачи држи и чекај услов, тако да ови услови нису потпуно независни.

### *9.2.2 Граф доделе ресурса*

Застоји се могу прецизно представити усмереним графом који се назива *граф доделе системских ресурса* $G = (V, E)$. Нека је $P = \{p_1, p_2, \ldots, p_n\}$ скуп процеса а $R = \{r_1, r_2, \ldots, r_m\}$ скуп типова ресурса. Онда је $V = P \cup R$ скуп чворова графа $G$, а

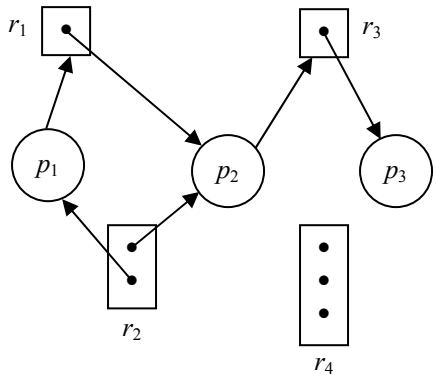
$$E=\{(p_i, r_j) \vee (r_j, p_i) \mid p_i \in P \wedge r_j \in R\}$$

скуп грана графа G. Грана ($p_i$, $r_j$) представља *грану захтева*, а грана ($r_j$, $p_i$) *granu dodele*. Размотримо следећи пример.

Нека је скуп процеса $P = \{p_1, p_2, p_3\}$ и нека постоје следећи типови ресурса $R = \{r_1, r_2, r_3, r_4\}$. Скуп грана је

$$E=\{(p_1, r_1), (p_2, r_3), (r_1, p_2), (r_2, p_2), (r_2, p_1), (r_3, p_3)\}$$

Нека у систему постоји један примерак ресурса типа $r_1$, два примерка ресурса типа $r_2$, један примерак ресурса типа $r_3$ и три примерка ресурса типа $r_4$. На основу изнетог, граф доделе системских ресурса је приказан на слици 9.1.




9.1. Граф доделе ресурса.

Лако се може показати да ако граф не садржи циклусе, тада ниједан процес у систему није у стању застоја. Ако постоји само по један примерак ресурса сваког типа, тада циклус у графу казује да је дошло до застоја. Такође, ако циклус обухвата само неке од ресурса, од којих сваки има по један примерак, јавља се застој и сваки процес обухваћен циклусом се налази у стању застоја.

У овом примеру циклус у графу је потребан и довољан услов за настанак застоја (међутим, у општем случају, не значи да ће до застоја и доћи!).

## 9.3 Превенција застоја

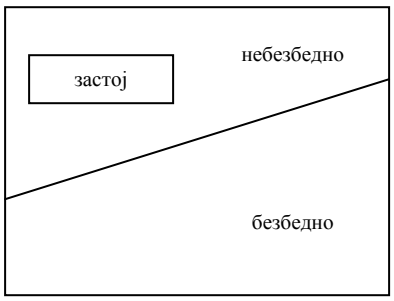
Превенција застоја састоји се у томе да се обезбеди да бар један од четири поменута услова не важи.

## 9.4 Избегавање застоја

Избегавање појаве застоја може се вршити помоћу алгоритама за превенцију, али је могуће да то изазове слабу искоришћеност ресурса и умањену пропусну моћ система.

Алтернативни методи за избегавање застоја састоје се у обезбеђивању додатних информација о начину на који се ресурси захтевају. Постоји више алгоритама који се разликују по износу и типу захтеваних информација.

Најједноставнији и најкориснији модел захтева да сваки процес декларише максимални број ресурса сваког типа који му могу бити потребни. Са датим информацијама за сваки процес, могуће је конструисати алгоритам који обезбеђује да ситем никад не уђе у стање застоја. Овакав алгоритам дефинише приступ избегавања застоја.




Сл. 9.2. Безбедно, небезбедно и стање застоја.

Алгоритам за избегавање застоја динамички испитује стање доделе ресура да би осигурао да се никад не јави услов кружног чекања. Стање доделе ресурса дефинисано је бројем расположивих и додељених ресурса и максималним захтевима процеса. Стање је *безбедно* уколико систем може да изврши доделу ресурса сваком процесу (до њихових максималних потреба) у одређеном редоследу а да се при том избегне појава

застоја.

Систем је у безбедном стању само ако постоји *безбедна секвенца*. Секвенца процеса $<p_1, p_2, \ldots, p_n>$ представља безбедну секвенцу за текуће стање доделе ако се за свако $n_u$ потребе за ресурсима могу задовољити располоивим ресурсима плус оним ресурсима које тренутно држе процеси $p_j$ ($j < i$). У овој ситуацији, ако ресурс који је потребан процесу $p_i$ није тренутно расположив, тада $p_i$ треба да сачека док се сви $p_j$ не заврше. Ако безбедна секвенца не постоји, систем је у небезбедном стању.

Безбедно стање није стање застоја, и обрнуто, стање застоја је небезбедно стање. Међутим, нису сва небезбедна стања у застоју, али таква стања могу да воде ка застоју (слика 9.2).

### *9.4.1 Банкаров алгоритам*

Овај алгоритам за избегавање застоја добио је име по томе што се може користити у банкама да се осигура да банка никад не додели свој расположиви новац на такав начин да после тога не може да задовољава потребе свих осталих клијената.

Нека у систему постоји $n$ процеса и $m$ типова ресурса. Следеће структуре података кодирају стање система расподеле ресурса.

1. Вектор *Available* дужине $m$ који указује на број расположивих ресурса сваког типа.
2. Матрица *Max* димензија $n \times m$ дефинише максималне захтеве сваког процеса.
3. Матрица *Allocation* димензија $n \times m$ дефинише број ресурса сваког типа који је тренутно додељен сваком процесу.
4. Матрица *Need* димензија $n \times m$ указује на преостале потребе сваког процеса за ресурсима.

Користићемо следећу релацију поређења вектора

$$X \le Y \Leftrightarrow X[i] \le Y[i], \quad \forall i \in \{1, 2, \ldots, n\}$$

Такође, врсте у матрицама *Allocation* и *Need* можемо сматрати векторима у ознаци $Allocation_i$ и $Need_i$.

Нека је још $Request_i$ вектор захтева процеса $p_i$. Према овом алгоритму, када се изврши захтев за ресурсима од стране процеса $p_i$, предузимају се следеће акције:

1. Ако је $Request_i \le Need_i$, пређи на корак (2); иначе, јавља се грешка пошто је процес превазишао своје максималне захтеве.
2. Ако је $Request_i \le Available$, пређи на корак (3); иначе, ресурс није расположив па $p_i$ мора да чека.
3. Систем тежи да додели тражене ресурсе процесу $p_i$ модификујући стање на следећи начин:
   $Available := Available - Request_i$;
   $Allocation_i := Allocation_i + Request_i$;
   $Need_i := Need_i - Request_i$;

Ако је резултујуће стање доделе ресурса безбедно, трансакција је завршена а процес $p_i$ је добио своје ресурсе. Међутим, ако стање није безбедно тада $p_i$ мора да чека на ресурсе из $Request_i$ а старо стање доделе ресурса се обнавља.

**<u>Safety алгоритам</u>**

Овај алгоритам одређује да ли је систем у безбедном стању или не.

1. Нека су *Work* и *Finish* вектори дужине $m$ и $n$, респективно. Иницијално је *Work*:=*Available* и *Finish*[$i$]:=*False*, $i$=1, …, $n$.
2. Наћи такво $i$ за које важи *Finish*[$i$]:=*False* и $Need_i \le Work$. Ако таквог $i$ нема, пређи на корак (4).
3. *Work*:=*Work*+$Allocation_i$; *Finish*[$i$]:=*True*. Иди на корак (2).
4. Ако је *Finish*[$i$]=*True*, за свако $i$=1, …, $n$, систем је у безбедном стању.

# 10 Датотечни систем

За већину корисника је фајл систем највидљивији аспект оперативног система. Датотеке садрже податке и програме. Оперативни систем имплементира апстрактне концепте управљања датотекама тако што врши управљање јединицама секундарне меморије.

## 10.1 Концепт датотеке

Датотеке се смештају на различите физичке медијуме као што су магнетне траке, дискови и слично. Сваки од ових уређаја има своје карактеристике и физичку организацију.

Ради погоднијег коришћења рачунарског система оперативни систем пружа униформни логички поглед на секундарну меморију апстракујући физичке особине појединог меморијског уређаја и дефинишући логичку јединицу – датотеку (*file*). Датотеке се пресликавају на физичке уређаје. Датетека је колекција логички сродних података која има своје име као и неке друге атрибуте.

Информације смештене у датотеци дефинише креатор датотеке. Различити типови информација могу бити смештени у датотеци: извршни програм, објектни програм, нумерички подаци, текст, итд. Фајл има одређену дефинисану структуру сагласно његовој употреби.

Једно од основних питања је колико од те структуре треба да буде познато оперативном систему и подржано од стране оперативног система.

У неким оперативним системима софтвер препознаје структуру датотека и подржава правилну употребу датотека одређене структуре. На пример, ако покушамо да извршимо програм чији је изворни кôд измењен, систем је у стању да најпре аутоматски изврши превођење изворног кôда, а тек онда извршење програма. Недостатак овог приступа је повећана величина оперативног система, јер мора да постоји кôд за подршку различитих типова датотека, ма колико да има тих типова. Следећи проблем је ако се појави нови тип датотеке који дотле није био подржан.

Друга крајност је да оперативни систем не намеће и не подржава било какве датотеке. Пример таквог оперативног система је Unix који све датотеке види као низове бајтова. Овакав приступ даје максималну флексибилност али минималну подршку.

Први фајл системи били су засновани на магнетним тракама. Сваки фајл је имплементиран његовим пресликавањем на сопствени котур траке. Предност овог приступа је у једноставности, али је недостатак у неефикасности, јер је већина датотека релативно мала док је капацитет трака релативно велики. Другачији, али сличан проблем се јавља када се појави веома велика датотека за чије смештање је потребно више трака. Да би подржали ову ситуацију многи системи су подржавали датотеке на више волумена.

Да би се решио проблем држања релативно малих датотека на скоро празној траци прешло се на смештање једнаког броја малих датотека на једну траку. Сада је настао проблем да се одреди која датотека је на којој траци. Из тих разлога је свакој траци придодат *директоријум*. Директоријум садржи имена и локације сваке датотеке на траци, као и додатне информације, као што су величина датотеке, датум и време креирања итд. Проналажење датотеке сада захтева проналажење одговарајуће траке и претраживање директоријума. Читање се одвија на исти начин као и када смо имали само једну датотеку на траци, с тим што се сада разликују знаци *end-of-file* и *end-of-tape*. Такође се разликује премотавање на почетак фајла и на почетак траке.

Овде постоје и друге тешкоће. Замислимо програм који чита алтернативно слогове из две датотеке које су на истој траци. Једна од датотека мора да се ископира на посебну траку или ће јединица траке да троши много времена крећући се од једне до друге датотеке и назад. Такође, код модификације датотека, све датотеке са траке се морају копирати и на друге траке и поново ископирати на оригиналну траку.

Многи од проблема са магнетним тракама решени су када се почело са употребом дискова. Диск је подељен на стазе, а свака стаза је даље подељена на секторе. Сектор је најмања јединица информације која се може прочитати са диска или која се може уписати на диск. Већи дискови имају више тањира, при чему сваки тањир има две површине. Да би се приступило сектору, морамо да специфицирамо површину, стазу и сектор. Уписно-читајуће главе се позиционирају на одговарајућу стазу (*seek time*), електронски пребацују на

одговарајућу површину а онда се сачека (*latancy time* - ротационо кашњење) да се ротацијом тражени сектор позиционира испод главе. Цилиндар је скуп стаза које су на истом растојању од осе диска, али се налазе на различитим површинама тањира. Није потребно тражење да се приступи стазама на истом цилиндру. Пренос података између меморије и диска врши се у јединицама које се састоје од једног или више сектора.

Дискови се разликују од трака на два начина. Пре свега, модификација датотека се може вршити на лицу места. Могуће је прочитати блок података, изменити га и уписати на исто место. Осим тога постоји директан приступ сваком блоку података на диску.

Као и траке, и дискови имају директоријум који указује које су датотеке на диску, заједно са подацима који се односе на њихову адресу на диску, величину и друге потребне информације.

## 10.2 Подршка датотекама

Концепт фајлова је имплементиран оперативним системом. Обезбеђени су системски позиви за креирање, упис, читање, премотавање и брисање датотека. Да би смо разумели како је фајл систем подржан, размотрићемо ове операције детаљније као и директоријуме.

### *10.2.1 Операције са датотекама*

Претпоставићемо да је фајл систем заснован на дисковима. Размотримо шта оперативни систем мора да уради за сваку од пет основних операција са датотекама.

- **Креирање датотеке**. Неопходна су два корака код креирања датотеке. Најпре се мора наћи простор у фајл систему за дотичну датотеку. Затим треба направити одговарајућу ставку у директоријуму. Та ставка бележи име датотеке и њену локацију у фајл систему.
- **Упис у датотеку**. Да би се извршио упис у датотеку мора да се начини системски позив где се специфицира и име датотеке и информација која треба да се упише у датотеку. За дато име датотеке систем претражује директоријум да би пронашао локацију датотеке. Ставка у директоријуму треба да садржи поинтер на текући крај датотеке. Коришћењем тог поинтера израчунава се адреса следећег блока и имформација може да се упише. Поинтер уписа мора да се ажурира. На тај начин сукцесивни уписи се могу користити за упис секвенце блокова у датотеку.
- **Читање датотеке**. За читање датотетке системски позив специфицира име датотеке и место у меморији где следећи блок датотеке треба да се смести. И овде се претражује директоријум, а одговарајућа ставка треба да садржи поинтер на блок који требе следећи да се прочита. Када се читање блока обави, тај поинтер се ажурира. У општем случају, датотека се чита или се у њу уписује, па иако је могуће да постоје два поинтера, већина система има само један који представља *текућу позицију датотеке*. И читање и упис користе исти поинтер, штедећи простор у одговарајућој ставци у директоријуму и поједностављујући систем.
- **Премотавање датотеке**. Премотавање датотеке у ствари не представља праву У/И активност, већ се само претражује директоијум и у одговарајућој ставци се поинтер текуће позиције ресетује да указаује на почетак.
- **Брисање датотеке**. За брисање датотеке се у директоријуму тражи специфицирана датотека, а онда се сав простор додељен датотеци ослобађа, а дотична ставка у директоријуму инвалидира.

Примећујемо да је за све описане операције потребно претраживање директоријума. Да би се избегло ово стално претраживање многи системи врше *отварање* датотеке када она почне да се стварно користи. Оперативни систем креира малу табелу са информацијама о свим отвореним датотекама, што олакшава претраживање. Када датотека више није потребна, она се *затвара* и одговарајућа ставка брише из табеле отворених фајлова.

Неки системи имплицитно отварају датотеку када се врши прво обраћање. Датотека се аутоматски завара када се програм који ју је отворио заврши. Већина система ипак захтева да се датотеке експлицитно отварају од стране програмера помоћу ситемског позива (*open*).

### *10.2.2 Директоријум уређаја*

Информације које се за сваку датотеку чувају у директоријуму варирају од једног до другог оперативног система. Следеће информације могу се наћи у директоријуму (не мора да значи да сви системи чувају све наведене информације).

- **Име датотеке**. Симболичко име датотеке.
- **Тип датотеке**. За оне системе који подржавају различите типове датотека.
- **Локација**. Поинтер на уређај и локацију на том уређају где се датотека налази.

- **Величина**. Тренутна величина датотеке (у бајтовима, речима или блоковима) и максимална дозвољена величина.
- **Текућа позиција**. Поинтер на текућу позицију за читање или упис у датотеку.
- **Заштита**. Податак о управљању приступу ради читања, уписа, извршавања и др.
- **Бројач коришћења**. Указује на број процеса који тренутно користе (имају отворену) дотичну датотеку.
- **Време, датум и идентификација процеса**. Ова информација се може чувати за креирање, последњу модификацију и последње коришћење. Ово може бити од користи за заштиту и надгледање употребе.

Да би смо записали ове информације за неку датотеку може нам бити потребно од 16 до 1000 бајтова. У системима са великим бројем датотека, величина самог директоријума може бити и на хиљаде бајтова. Тако се директоријум који се налази на уређају може учитавати у меморију само у мањим деловима, по потреби.

Јасно је да се директоријум може организовати на више начина. Један од њих је је линеарна листа што захтева линеарно претраживање за проналажење одговарајуће ставке. Одговарајући програм је једноставан али троши превише времена.

Друга структура података која се може користити за смештање директоријума је хеш табела. Највећи проблем са хеш табелама јесте њихова генерално фиксирана величина и зависност хеш функције од величине хеш табеле.

## 10.3 Методи приступа датотекама

Постоји више начина да се приступи информацијама запамћеним у датотекама. Неки системи подржавају само један метод приступа, док код многих постоји могућност избора више метода за приступ датотекама.

### *10.3.1 Секвенцијални приступ*

Код секвенцијалног приступа читање једног блока података аутоматски увећава вредност поинтера, а слично се дешава и код уписа. Дакле, могуће је вршити читање података само у редоследу у коме су уписани, односно вршити упис узастопних блокова података. Овакав приступ заснован је на моделу датотека које су уписиване на магнетне траке које су по својој природи секвенцијални уређаји.

### *10.3.2 Директни приступ*

Алтернативна метода приступа је *директни приступ* што је карактеристично за модел датотека уписаних на диск. Овде се датотека види као нумерисана секвенца слогова фиксне дужине. Уз навођење релативног броја блока може се приступити произвољном слогу датотеке.

### *10.3.3 Индексни приступ*

Овај метод користи *индекс* за сваку датотеку који користи указатеље на сваки од блокова датотеке. Да би се пронашла ставка у датотеци, најпре се претражује индекс и онда користи указатељ да би се директно приступило одговарајућем блоку датотеке и тражењу жељене ставке у њој.

## 10.4 Методи доделе простора на диску

Постоје три основне методе за доделу простора на диску. То су *континуална*, *повезана* и *индексна* додела. Свака од ових метода има своје предности и недостатке, па постоје системи који подржавају све три методе. Међутим, чешћи је случај да један систем користи само једну од наведених метода.

### *10.4.1 Управљање слободним простором на диску*

За праћење слободног простора на диску систем креира листу слободног простора. Ова листа садржи све блокове на диску који су слободни, тј. нису додељени ни једној датотеци. Када се датотека креира претражује се листа слободног простора да би се у њој пронашао жељени износ слободног простора и доделио датотеци. Када се датотека брише, простор на диску који је она заузимала се додаје листи слободног простора.

Упркос имену, ова листа се имплементира као *бит мапа* или *бит вектор*. Сваки блок представљен је једним битом, при чему, ако је блок слободан, одговарајући бит има вредност 0, а ако је блок заузет, вредност одговарајућег бита је 1.

Други приступ је да се сви слободни блокови повежу у листу при чему се користи указатељ на почетак листе. Ова шема није нарочито ефикасна, јер за прелазак целе листе треба утрошити доста времена. Модификација овог приступа састоји се у томе да се адресе *n* слободних блокова налазе у првом слободном блоку (тада имамо *n*-1 заиста слободних блокова); последња адреса у блоку је адреса наредног блока који садржи адресе слободних блокова.

Један други приступ користи чињеницу да више континуалних блокова може бити заузето или ослобођено. Тада је боље имати листу која садржи адресу првог блока у секвенци слободних блокова и број блокова у тој секвенци.

### *10.4.2 Метод континуалне доделе*

Код овог метода се захтева да свака датотека заузима скуп континуалних адреса на диску. Тада ставка у директоријуму за сваку датотеку садржи почетну адресу датотеке и број блокова који су јој додељени. Међутим, јавља се проблем када треба пронаћи простор за нову датотеку. Овај проблем се може сматрати специјалним случајем општег проблема динамичке доделе меморије.

Овде простор на диску посматрамо као велико поље блокова. У произвољном тренутку неки блокови су заузети, док су остали слободни. Тако простор на диску можемо сматрати колекцијом заузетих и слободних сегмената где је сваки сегмент континуалан скуп блокова. Слободни сегмент назива се *рупа* или *шупљина*.

Овде се јавља проблем фрагментације који се може решавати компакцијом. Постоје и други проблеми везани за континуалну доделу. Основни проблем је одредити колико простора да се додели датотеци. Када се датотека креира, читав износ простора треба доделити датотеци. Како корисник зна величину датотеке која се креира? У неким случајевима ово питање је једноставно, на пример, када се копира постојећа датотека, али је, у општем случају, ту процену тешко извршити. Ако доделимо мало простора датотеци, ускоро ћемо доћи у ситуацију да она не може више да се проширује, јер простор на оба краја датотеке може бити заузет од стране других датотека. Две могућности стоје на располагању у таквом случају. Прва је да се кориснички програм терминира уз одговарајућу поруку. Друга могућност је да се потражи већа шупљина у којој можемо да сместимо датотеку, ослобађајући претходно заузети простор. Ово се, уз утрошак времена, може понављати све док постоји слободан простор.

Чак и када унапред знамо количину потребног простора за неку датотеку коју креирамо, може се испоставити да није ефикасано да сав простор доделимо датотеци. Датотеке које нарастају споро, током дугог временског периода (током месеци или чак година) у том случају у доброј мери непотребно заузимају простор.

### *10.4.3 Метод повезане доделе*

Код ове методе свака датотека представља повезану листу блокова на диску, при чему се ти блокови могу налазити било где на диску. Директоријум садржи указатељ на први и последњи блок датотеке.

Овде немамо екстерну фрагментацију. Такође, при креирању датотеке није потребно декларисати величину датотеке, а датотека може да нараста докле год има слободних блокова на диску. То значи да није потребно вршити компакцију диска.

Основни проблем код овог начина доделе је што је овај метод ефикасан само код секвенцијалног приступа датотеци. Уз то се троши и простор за указатеље. Такође је смањена поузданост, јер у случају оштећења указатеља настају нежељене последице. Једно решење овог проблема је коришћење двоструко повезаних листа или памћење имена датотеке и релативног броја блока у сваком блоку; све ово захтева додатни утрошак времена и меморије за сваку датотеку.

### *10.4.4 Метод индексне доделе*

Овај метод решава проблем повезане доделе у смислу директног приступа. Овде су сви указатељи заједно смештени у *индексни блок*. Свака датотека има сопствени индексни блок који представља поље адреса блокова на диску. Директоријум садржи адресу индексног блока.

Индексна додела подржава директни приступ а при том нема екстерне фрагментације. Ипак постоји проблем утрошка простора због указатеља у индексном блоку који је већи него код повезане доделе.

Овај проблем је повезан са величином идексног блока па се тежи да он буде релативно мали, док се за веће датотеке уланчава више индексних блокова или се користи индексирање у два нивоа.

Још једна алтернатива је да у адресару чувамо првих неколико указатеља индексног блока, тако да за мале датотеке индексни блок није потребан (Unix).

## 10.5 Системи директоријума

Датотеке представљају ставке у директоријуму уређаја (или садржају волумена, како се негде зове). Директоријум уређаја бележи информације као што су име, локација, величина и тип датотека на том уређају.

Директоријум уређаја може бити довољан за једнокорисничке системе са ограниченим секундарним меморијским простором. Како секундарни меморијски простор и број корисника расту, постаје веома тешко организовати и пратити све те датотеке. Решење овог проблема је да се уведе структура директоријума у фајл систем. Структура директоријума обезбеђује механизме за организовање мноштва фајлова у фајл систему. Она може да се протеже и на више јединица дискова. Тако корисник треба да посматра само логичке директоријуме и структуру датотека, потпуно игноришући проблеме физичке доделе простора датотекама.

У ствари, многи системи имају две одвојене структуре директоријума: директоријум уређаја и директоријуме датотека. Директоријум уређаја је смештен на сваком физичком уређају и описује све датотеке на том уређају, концентришући се на физичке особине сваке датотеке (локација, дужина, начин доделе итд.). Директоријуми датотека представљају логичку организацију датотека на свим уређајима, концентришући се на логичке особине (име, тип, власник, приступни кôд итд.). Директоријум датотеке може једноставно да указује на одговарајућу ставку у директоријуму уређаја да би се прочитале физичке особине датотеке или да садржи копију те информације. Наш главни интерес ће бити структура директоријума датотека.

Различите структуре директоријума датотека су до сада предлагане и коришћене. Директоријум је у суштини табела симбола. Оперативни систем узима симболичко име датотеке и проналази ту датотеку. Основне операције које се извршавају над директоријумом су:

- **Претраживање**. Потребно је да будемо у стању да претражујемо директоријум да би пронашли ставку за одређени фајл. Пошто фајлови имају симболичка имена а слична имена могу да указују на сродне датотеке, треба да будемо у стању да проналазимо све датотеке које одговарају неком шаблону.
- **Креирање датотеке**. Треба да постоји могућност креирања и додавања нових датотека директоријуму.
- **Брисање датотека**. Када датотека није више потребна, врши се њено уклањање из директоријума.
- **Листање директоријума**. Потребно је да будемо у стању да излистамо фајлове који су у директоријуму и садржај ставки директоријума за сваку датотеку у тој листи.
- **Бекап**. Ради повећане поузданости система добро је повремено памтити садржај структуре фајл система. Ове се често врши копирањем свих датотека на магнетну траку. Тако имамо резервну копију у случају отказа система, или ако се датотека више не користи, можемо да ослободимо њен простор на диску.

Најједноставнија структура директоријума је једнонивовски директоријум. Све датотеке се налазе на истом директоријуму, што је лако подржати и разумети. Међутим, овде постоји проблем када број датотека нарасте или се повећа број корисника, јер свака датотека треба да има јединствено име.

Највећи недостатак једнонивовског директоријума је у конфузији која настаје када имамо више корисника. Стандардно решење је да се за сваког корисника креира одвојени директоријум. Кориснички директоријуми представљају логичку, а не физичку организацију.

Двонивовски директоријуми могу се сматрати двонивовским стаблима. Као природно уопштење овог приступа настала је структура директоријума у оквиру стабла. Корисницима се сада омогућава да креирају сопствене поддиректоријуме и на тај начин организују своје датотеке. Стабло директоријума има свој корен директоријум. Свака датотека у систему има јединствено име пута. Име пута је пут од корена стабла, преко свих поддиректоријума, до специфициране датотеке.

# Садржај


**1 УВОД У ОПЕРАТИВНЕ СИСТЕМЕ ..... 1**

1.1 Дефиниција и задаци оперативног система ..... 1
1.2 Зачеци оперативних система ..... 2
1.3 Побољшавање перформанси првобитних оперативних система ..... 4
*1.3.1 Рад off-line ..... 4*
*1.3.2 Баферовање ..... 4*
*1.3.3 Spooling ..... 4*
*1.3.4 Мултипрограмирање ..... 5*
*1.3.5 Дељење процесорског времена (Time Sharing) ..... 5*
*1.3.6 Системи за рад у реалном времену (Real-Time Systems) ..... 5*
1.4 Проблеми заштите ..... 6

**2 СЕРВИСИ ОПЕРАТИВНОГ СИСТЕМА ..... 7**

2.1 Типови сервиса ..... 7
2.2 Кориснички поглед ..... 7
*2.2.1 Системски позиви ..... 7*
*2.2.2 Системски програми ..... 8*
2.3 Поглед оперативног система ..... 9
*2.3.1 Системски позив ..... 9*
*2.3.2 Прекиди У/И уређаја ..... 10*
*2.3.3 Грешке у програму ..... 11*
*2.3.4 Општи дијаграм тока оперативног система ..... 11*

**3 УПРАВЉАЊЕ ПРОЦЕСИМА ..... 12**

3.1 Концепт мултипрограмирања ..... 12
3.2 Концепт планирања послова ..... 12
*3.2.1 Основни концепти ..... 12*
*3.2.2 Редови процеса ..... 14*
*3.2.3 Планери ..... 14*
3.3 Алгоритми планирања ..... 15
*3.3.1 Критеријуми перформанси ..... 15*
*3.3.2 FCFS алгоритам ..... 16*
*3.3.3 SJF алгоритам ..... 16*
*3.3.4 Алгоритам са приоритетима ..... 17*
*3.3.5 Алгоритми са прекидањем ..... 18*
*3.3.6 Кружни алгоритам ..... 18*
*3.3.7 Вишенивовски редови ..... 19*
*3.3.8 Вишенивовски редови са повратном спрегом ..... 19*
3.4 Процеси и нити ..... 19
*3.4.1 Вишеструке нити у оквиру једног процеса ..... 19*
3.5 Windows NT ..... 20
*3.5.1 Историјски развој ..... 20*
*3.5.2 Multitasking у једнокорисничком окружењу ..... 20*
*3.5.3 Опис оперативног система ..... 21*
*3.5.4 Нити ..... 23*
*3.5.5 Симетрично мултипроцесирање ..... 23*
*3.5.6 Windows NT објекти ..... 23*
3.6 Опис процеса и управљање процесима код Windows-а NT ..... 24

*3.6.1 Објекти процеса и објекти нити* ... *25*
*3.6.2 Multithreding* ... *27*
*3.6.3 Подршка подсистемима оперативног система* ... *28*

**4 КОНКУРЕНТНИ ПРОЦЕСИ ... 29**

4.1 Графови првенства ... 29
*4.1.1 Дефиниција графова првенства* ... *29*
*4.1.2 Услови конкуренције* ... *29*
4.2 Спецификација конкуренције ... 30
*4.2.1 Конструкције Fork и Join* ... *30*
*4.2.2 Конкурентни искази* ... *31*
*4.2.3 Поређење конструкција fork и join и конкурентних израза* ... *31*
4.3 Проблем критичне секције ... 31
*4.3.1 Софтверско решење за два процеса* ... *32*
*4.3.2 Софтверска решења за N процеса* ... *34*
*4.3.3 Хардверска решења* ... *35*
4.4 Семафори ... 37
*4.4.1 Коришћење семафора* ... *37*
*4.4.2 Имплементација семафора* ... *37*
4.5 Класични проблеми координације процеса ... 38
*4.5.1 Проблем ограниченог бафера* ... *38*
*4.5.2 Проблем читалаца и писаца* ... *39*
*4.5.3 Проблем филозофа који вечерају* ... *40*

**5 КОНКУРЕНТНО ПРОГРАМИРАЊЕ ... 41**

5.1 Мотивација за коришћење виших програмских језика ... 41
5.2 Модуларизација ... 41
*5.2.1 Процеси* ... *41*
*5.2.2 Процедуре* ... *42*
*5.2.3 Апстратктни типови података* ... *42*
5.3 Синхронизација ... 43
*5.3.1 Критичне области* ... *43*
*5.3.2 Условне критичне области* ... *43*
*5.3.3 Монитори* ... *45*
5.4 Синхронизација код Windows NT-а ... 46

**6 УПРАВЉАЊЕ МЕМОРИЈОМ ... 48**

6.1 Гола машина ... 48
6.2 Резидентни монитор ... 49
*6.2.1 Заштитни хардвер* ... *49*
*6.2.2 Релокација* ... *49*
6.3 Пребацивање (swapping) ... 50
6.4 Вишеструке партиције ... 52
*6.4.1 Партиције фиксне дужине (MFT)* ... *52*
*6.4.2 Партиције променљиве дужине (MVT)* ... *53*
6.5 Страничење ... 54
6.6 Сегментација ... 55

**7 ВИРТУЕЛНА МЕМОРИЈА ... 57**

7.1 Оверлеји ... 57
7.2 Страничење на захтев ... 58
7.3 Перформансе страничења на захтев ... 61
7.4 Замена странице ... 61
7.5 Концепти виртуелне меморије ... 64
7.6 Алгоритми замене страница ... 64
*7.6.1 FIFO* ... *64*
*7.6.2 Оптимална замена* ... *64*
*7.6.3 LRU* ... *65*

*7.6.4 Апроксимације LRU алгоритма* ... *65*
7.7 УПРАВЉАЊЕ МЕМОРИЈОМ КОД WINDOWS NT-А ... 66
*7.7.1 Адресни простори* ... *66*
*7.7.2 Сегментација* ... *66*
*7.7.3 Страничење* ... *68*

**8 УПРАВЉАЊЕ ДИСКОВИМА ... 69**

**9 ЗАСТОЈИ ... 70**

9.1 ПРОБЛЕМ ЗАСТОЈА ... 70
9.2 КАРАКТЕРИЗАЦИЈА ЗАСТОЈА ... 70
*9.2.1 Потребни услови* ... *70*
*9.2.2 Граф доделе ресурса* ... *70*
9.3 ПРЕВЕНЦИЈА ЗАСТОЈА ... 71
9.4 ИЗБЕГАВАЊЕ ЗАСТОЈА ... 71
*9.4.1 Банкаров алгоритам* ... *72*

**10 ДАТОТЕЧНИ СИСТЕМ ... 73**

10.1 КОНЦЕПТ ДАТОТЕКЕ ... 73
10.2 ПОДРШКА ДАТОТЕКАМА ... 74
*10.2.1 Операције са датотекама* ... *74*
*10.2.2 Директоријум уређаја* ... *74*
10.3 МЕТОДИ ПРИСТУПА ДАТОТЕКАМА ... 75
*10.3.1 Секвенцијални приступ* ... *75*
*10.3.2 Директни приступ* ... *75*
*10.3.3 Индексни приступ* ... *75*
10.4 МЕТОДИ ДОДЕЛЕ ПРОСТОРА НА ДИСКУ ... 75
*10.4.1 Управљање слободним простором на диску* ... *75*
*10.4.2 Метод континуалне доделе* ... *76*
*10.4.3 Метод повезане доделе* ... *76*
*10.4.4 Метод индексне доделе* ... *76*
10.5 СИСТЕМИ ДИРЕКТОРИЈУМА ... 77