Кратка история за Linux

Една от първите стабилни операционни системи е UNIX, която излиза през 1969г. През годините е имала различни промени. За момента се ползва като операционна система за големи корпорации и правителства. Съществуват и нейни разновидности като HP-UX.

Персоналните компютри навлизат към края на 70-те и особено в началото на 80-те години. Най-привлекателната част на копютрите по онова време е GUI (графичен потребителски интерфейс), което не е било особено надеждно. За персоналните компютри с GUI се отличават: Xerox (1981), а след това и Apple(1983), но нямат успех тъй като са много скъпи (20-30 хил. $). Най-голям успех е постигнат през 1984 с илизането на Mac OS X за Macintosh.

Поначало буквата X идва от Unix и с нея се означава графиката (т.е. GUI). В последствие се използва за означаване на графиката и при всички UNIX базирани системи. Важно е да се знае, че при повечето UNIX системи графиката може да се заобикаля, например като се премине в терминален режим.

През средата на 80-те години не само компютрите са били сравнително скъпи, но и софтуера за тях. По същото време възниква идеята за безплатен софтер и за безплатна OS. Тази идея е въведена от Ричард Столман, който създава и проекта GNU през 1983г. Абревиатурата GNU произлиза от „GNU is not UNIX“ и се използва за open source операционна система. Обикновенно повечето абревиатури за open source нещата са „рекурсивни“. Та за Ричард Столман – той не успява да създаде такава операционна система. Той написва първия open source компилатор за C++, който се нарича gcc и се ползва и до днес (естествено с новите си версии).

През 1989г. Столман създава и лицензa GPL (General Public License) за програмите който ще се използват за GNU.

По същото време в университета в Хелзинки се е ползвала операционната система MINIX – която е била орязана версия на UNIX, но с отворен код с цел изучаване от студентите. Тогва студентът Линус Торвалдс започва личен проект за операционна система. Към проекта се присъединяват и други студенти. Така през 1991 година възниква първата версия на ядрото Linux (Linus + Unix). Заедно с някои от инструментите и програмите създадени към проекта GNU се сформира операционната система GNU Linux, която е изцяло под лицензa GPL. Голяма роля за създаването на Linux по това време изиграва и Stolman. Към проекта се присъединяват лавинообразно и много други програмисти.

Създават се и различни общности (communities) поддържащи различни възгледи за обвивката на ядрото. Важно е да се знае, че под Linux трябва да се разбира само ядрото на операционната система. Всички останали програми и инструменти за работа (терминал, графична система и др.) с него се наричат обвивка. Възникават и различни дистрибуции:

Wiki -> Linux distribution

В началото (1991-1992) ядрото като обвивка е имало само терминала. По същото време на пазара се появява Windows 3.11 и съответно Linux с нищо не може да се конкурира. Затова се налага създаването на графична среда. За целта участват всички общности поддържащи Linux ядрото.

Първата графична среда е KDE [keidi: i:] която излиза 1996 година. Написана е чрез платформено независимата библиотека Qt. Чрез Qt са написани програми като Skype, Google Maps и др. Като абривиатура KDE означава *K Desktop Environment.* Но Qt има проблем с лиценза и съответно всички програми, включително и KDE са засегнати от това.

Поради тази причина общностите са принудени и написват open source библиотека GTK+. Чрез нея е написана графичната система GNOME [gnoum], която излиза през 1998, официално 1999г.

Малко след това и Qt е обявена под лиценза GPL и съответно KDE става напълно беплатна.

Затова след 2000-та година повечето дистрибуции вървят с два варианта за избор на графична система KDE или GNOME. За момента има разработени и много други (>10) графични системи.

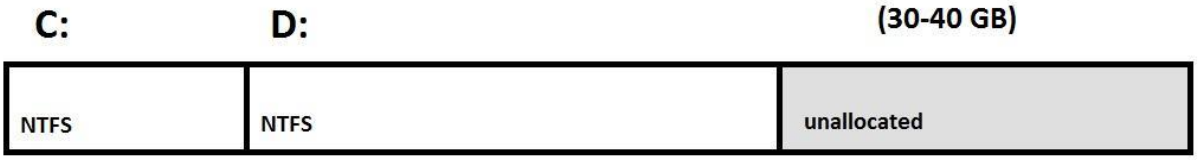
Важно е да се знае, че за всяка графична система има и различен набор от програми. Например за KDE програмите често започват или съдържат буквата K – Kwrite, KAmarok, Ark и др.

Някои дистрибуции плътно се придържат само към една от тези системи. Например Debian базираните (Ubuntu, Linux Mint и др.) са с GNOME с цел никога в историята си да не са имали нещо платено по тях.

Други дистрибуции като Red Hat и Suse са предимно с KDE. Тези две дистрибуции стават и първите комерсиални (retail) (red hat е първа): Red Hat Enterprice и Suse Enterprice. Но тъй като ползват open source ядро са „задължени“ да имат и open source варианти. Те са съответно: Fedora Core и openSuse.

<u>Инсталация на Linux</u>:

Когато се инасталира Linux на отделен хард диск съответнно е най-безпроблемно. Когато на диска има друга ОС – например Windows трябва да се съобразят някои особености. Първо трябва да бяде инсталирана Windows системата, така Linux OS ще я разпознае, ще може да индексира за да може след това да създаде меню чрез което да бъде стартирана и тя. Програмата която зарежда Линукс се нарича GRUB (**GNU GRand Unified Bootloader**), и тя ще конструира меню. Освен това е препоръчително (за да няма проблем с форматиране) да бъде отделен дял от хард диска без форматиране (unformatted или unallocated). Разделянето може да стане и през Windows чрез инструмента Disk Managment.



По този начин инсталатора ще разпознае този диск и ще продължи инсталацията върху празния диск. За файлова система в Linux се ползва Ext3FS (стара е) и Ext4FS.

Обикновено Линукс отрива отделните носители с информация (хардискове, флаш памети и др) като устройства с имена sda, adb, sdc и т.н. След името на устройството стои номера на дяла. За MBR номерата от 1 до 4 са запазени за главни дялове, а от 5 нагоре за логически. При GPT номерацията е от 1 нагоре.

Инсталацията може да се извърши от DVD или bootable Flash Drive.

След инсталация Линукс ще е създал допълнителни дялове:

/ - главна или коренова (root dir) директория. Основна директория за всички останали.

/swap - това е виртуалната памет. Отделена е на отделен дял без форматиране.

/home – домашна директория за всички локални потребители

/root – домашна директория за потребителя root (суперпотребителя)

Основните потребители в Linux са:

root - това е запазено име за потребителя с пълни права за всяка линукс система. Нарича се още

суперпотребител (superuser). В терминала root е с промпт #.

Всички останали потребители са локални и са с ограничени права. В терминала са с промт $. За да имат тези потребители определени права могат да се добавят към групата на администраторите sudo. По време на инсталацията, на една от нейните стъпки се създава и един локален потребител потребител на който се задва име и парола. Този потребител е и към групата sudo.

При Ubuntu потребителя root съществува, но в началото няма парола. Такава може да бъде добавена от създадения локален потребител чрез следната команда в терминала:

```
sudo ␣ passwd
```

!Но няма да я добавяме!

Командата sudo се използва за изпълнение на администраторски комнади от локален потребител, стига той да е в групата sudo (да има sudo привилегии). Тази команда е силно препоръчителна и чрез нея се избягва преминаване като потребител root. Обикновено в Linux логване като root се прави изключително рядко и се избягва. Поради това в Ubuntu няма парола за root, а се използват привилегиите на sudo. Командата passwd служи за промяна на парола на даден потребител. Когато не е посочен такъв потребител като аргумент се подразбира root. За да може обикновенния потребител да влезне като root може да се използва командата su (substitute user):

```
sudo ␣ su␣-
```

Тирето зарежда променливите на средата на root, включително и преминаване в неговата домашна директория (т.е. /root). За да преминем към локалния потребител използваме само :

```
su␣-l␣localUser
```

Опцията -l е за да променим и домашната директория на новия user.

Друг вариант за илизане от сесия е чрез Ctrl+D (logout).

След инсталация може да остане Линукс като система по подразбиране за зареждане. Това се променя от системата за начално зареждане GRUB.

-> Google search: How do I change the GRUB boot order?

Обикновенно се променя конфигурационния файл на grub, който се намира в /etc.

==========

1. Обвивката (shell) в Linux

В самото сърце на операционната система Linux е разположена серия от машинни инструкции, която се нарича ядро (kernel) – това е техническа програма, която не е много подходяща за директно използване от потребителя, тъй като е създадена да комуникира предимно на системно ниво. За тази цел служи така наречената обвивка на ядрото (shell). Чрез нея потребителите могат да комуникират в разбираема форма с ядрото на операционната, основно чрез инструкции. Обвивката случи да превежда инструкциите от командния ред до машинен код. Т.е. още се нарича команден интерпретатор. На фигурата по долу е показана взаимовръзката между ядрото на Linux, хардуера и програмите.

След като ядрото се зареди, всички елементи от централния кръг се управляват от него; достъпът до всички хардуерни устройства и файлове се осъществява чрез ядрото. Областта, дефинирана от средния кръг (където се разполага ядрото), често се нарича пространство на ядрото. Външният кръг е пространството на потребителя; обвивката, всички програми, стартирани от нея, както и демоните, оперират в пространството на потребителя. Тези програми осъществяват достъп до хардуера и другите файлове (ресурсите) само чрез ядрото – предоставянето или отказването на достъпа се определя на базата на позовленията.

Обвивката не е нищо повече от програма, която се изпълнява в пространството на потребителя и ви предоставя интерактивен интерфейс към системата (команден промт) за въвеждане на данни или за задаване на програми, които да се стартират. Подразбиращата се обвивка за повечето Linux дистрибуции е bash (Bourne Again SHell).

BASH е актуализирана версия на първоначалната обвивка Bourne в операционната система Unix.
Обвивката разпознава голям брой команди, които се съдържат в директориите /bin и /sbin

Работа с терминала:

В Linux освен стандартната подразбираща се обвивка BASH има и 6 виртуални терминала.

Те могат да се отворят с комбинацията

Ctrl+Alt + (от F1 до F6) съответно за терминалите от tty1 до tty6.

Ctrl+Alt+F7 се отваря графичната система.

През виртуалните терминали графичната система може да биде спирана и след това стартирана, т.е. в Линукс тя може да бъде заобикаляна.

Терминала в Ubuntu може да се отвори с комбинацията Ctrl+Alt+T.

Стандартният формат на конзолните команди в Unix е:

команда *[-опции] [аргумент]*

Разделителя е интервала (␣), като командите се пишат с малка буква. Linux е чуствителен към регистъра и прави разлика между малки и големи букви. Опциите се задават с тире (-) преди тях.

Примери:

`whoami` - показва потребителското име на текущия потребител

`pwd` – извежда името на текущата директория (print working directory)

`ls` - извежда списък (съдържанието) на текущата директория // list

`ls -a` - ls е стартирана с една опция a (all) ще изведе всички, дори и скритите файлове и директории

`ls -al` - ls е стартиран с две опции , l (long) ще изведе подробна информация за файла или директорията

`ls -al /bin` - ще изведе всички файлове и директории в подробен формат от дир. /bin

Тъй като командите са твърде много не е нужно да се помнят всичките им опции. Ще покажем как да разглеждаме опциите за всички команди. Когато напишем само началото на команда, например pw и когато натиснем два пъти табулация ще ни изведе всички команди които започват с pw. Ако не сме въвели нищо и натиснем два пъти Tab, разбира се ще изведе всички команди от обвивката като списък. Тъй като са много ще ни попита дали да ги изведе. За всяка команда важи извеждането на помощна информация чрез дописването на - -help след нея.

`ls --help` - извежда кратка информация за опциите на дадена команда

Можем да отворим и по подробна информация за всяка команда чрез така наречените man (manual - ръководство) страници или чрез info страниците. Отваря се в целия терминал, като прелистването става чрез Page Up и Page Down . Излиза се чрез Q от клавиатурата:

`man ls` - man страницата за командата ls

`man man` -man страница за самата команда man

`info ls` - info страницата за команда ls

Изпълнението на команда може да се прекрати чрез комбинацията Ctr+C или Ctrl+Z.

Други команди :

`clear` - изчиства екрана

`date` - извежда датата

`cal` - извежда календар

uptime – колко време е работила системата след последния рестарт

За смяна на потребителя се използва командата substitute user:

```
su ␣ otherUser
su ␣ -l ␣otherUser     -- сменя потребителя + домашната му директория
su ␣              -- когато е без аргумент искаме да сменим с root
```

==========

2. Инсталиране на програми:

За Debian дистрибуциите се използват .dpkg пакети и често се използват командите apt-get:

Например за инсталация на програмата Gparted:

sudo␣ apt-get␣ install␣ gparted

а за деинсталиране:

sudo␣ apt-get␣ remove␣ gparted

често се използва и вариант с purge за деинсталиране , чрез който се премахват и конфигурационните файлове:

sudo␣ apt-get␣ purge␣ <package_name>

за обновяване с нови пакети:

sudo␣apt-get␣update

за upgrade на системата

sudo␣apt-get␣upgrade

За други дистрибуции се използват други средства. Например за Red Hat и Suse се използват .rpm пакети заедно с прилежащите им среди за инсталация:

yum - за red hat

yast - за suse

==========

3. Файлова система на Linux

Google search: Linux Filesystem Hierarchy Standard (FHS)

Когато стартираме терминала той се отваря като текущата директория ще бъде домашната директория на потребителя с който сме влезнали.

/root - домашната директория за суперадминистратора

/home/local - домашната директория на потребителя local

В терминала можем да сменим текущия потребител с командата su (substitute user). Ако я стартираме без опции тя ще запази работната директория от предходния потребител. Затова е препоръчително да е ползва с опция –l , чрез която променяме и работната директория.

`su␣ -l` - влизаме като администратор, като отиваме и в домашната му директория /root, ще ни иска да въведем администраторската парола. Паролите не се изписват в терминала, когато я въвеждаме, дори без *.

`su␣ local` - променяме в потребителя local като работната директория остава същата като на предходния потребител. Администратора може свободно да преминава като друг потребител.

За Ubuntu влизането като суперпотребител става чрез:

sudo␣su␣-

`Излизане става с Ctrl+D`

Обхождане по директориите:

`cd␣ [път]` – променя работната директория change directory

`cd` - ако е без параметри ще отиде в домашната директория

За директориите имаме служебни символи:

. – указва текущата директория

.. – указва предходната директория (родителска)

/ - главна директория

~ -домашна директория

- -предишната работна директория

Например чрез cd .. ще отидем в предходната директория, а с cd ~ ще отидем в домашната директория. Разделителя интервал е задължителен. Когато се обхождат директориите имаме два начина за описание на пътя до даден файл или директория:

- абсолютен; той започва винаги от главната директория /

- относителен; започва от текущата директория в която се намираме

Пример: ако текущата директория е /home и искаме да отидем до поддиректорията local в нея, ще стане чрез два начина :

- абсолютен : `cd␣ /home/local`

-относителен `cd␣ local`

Вижда се, че с относителния път в този случай е по кратко, но ако искаме чрез относителен път да отидем до /etc ако в момента сме в /home/local то това ще стане така:

```
cd ../../etc
```

а чрез абсолютния:

```
cd /etc
```

стартираме листа :

`ls –l` за да видим пълната информация за всеки един фаил

Обикновено терминала ги оцветява, като директориите са в синьо, изпълнимите файлове в зелено, връзките (link) в светло зелено и т.н. Подробната информация за файла има следния вид:

`-|9 символа|бр.файлове|собственик|група|размер|дата на последна мод.|име`

Първият символ означава вида на файла:

- – обикновен файл
- d директория
- l символна връзка – symbolic link (shortcut)

Ако първия символ е друг файлът е служебен (S сокет, b блоково устройство и др.)

9-те символа са за правата на достъп.

Пример за файла /etc/passwd:

-rw-r--r-- 1 root root 695 Dec 7 12:48 passwd

Създаване на директории става както е посочено по долу. Директориите също се разглеждат като файлове, само че те съдържат информация за други файлове. Препоръчително е имената на

директориите да не съдържат интервал, тъй като когато я изписваме като аргумент на команда интервала ще се чете като разделител. Ако все пак държим да има интервал, той трябва да се скрива(escape) с наклонена черта : my\ dir . Също така не е желателно да се създават файлове и директории в главната / директория.

`mkdir mydir` - създава директория с името mydir

`rmdir mydir` - изтрива директорията mydir (само ако е празна)

създаваме пак директорията mydir и влизаме в нея :

`cd mydir`

3.1 Файлове – създаване:

Всички файлове които започват с точка (**.**) са скрити. Имената на файловете е препоръчително да не съдържат интервали.

`touch f1` - създава празен файл с името f1

Друг начин за създаване на файл е като се използва знака за насочване на поток (>) и в комбинация с някоя команда, която извежда резултат:

`date >f2` - ще отпечата датата като създаде файл с името f2

`cal >f3` - същото като date, само че за календара

echo ␣ “helloworld” ␣ > ␣ f4

За прочитането на файл най-често се използва командата cat (concatenation), която конкатенира низа прочетен от файла с конзолата

`cat f3` - прочита файла f3 и го отпечатва в конзолата

Чрез нея също може да се създаде файл, като се използва отново знака (>) и този път ни се дава възможност да пишем във файла. Спирането става след последния Enter изпълним комбинацията Ctrl+C.

`cat >file` - създаваме файл с името file като пишем в него.

`cat >. hiddenfile` - създаваме скрит файл. Той ще бъде видим чрез командата ls -l

---

3.2. Копиране на файл

`cp <файл> <директория>` - копира файл в посочената директорията

`cp file1 file2` - копира един файл в друга директория с друго име

`mv <файл> <директория>` - премества файл в посочена директория (cut)

`mv file1 newfile1` - ще преименува file1 в newfile1

Метазнаци:

За определени цели се използват служебни символи които могат да заместват други:

* - замества произволен брой знаци

? – отговаря на единствен знак

Пример:

b* - всички думи които започват с b

b??? – всички четирибуквени думи които започват с b

зад1. Създайте в домашната директория, поддиректория с името newdir, копирайте в нея всички файлове от /bin, чийто имена започват с p и покажете резултата:

```
cd ~
mkdir newdir
cd newdir
cp /bin/p*  .
ls –l
```

в случая (.) означава текущата директория в която се намираме.

3.3 Връзки към файловете:

Създават се чрез командата ln и биват два вида: твърди и меки (символни). Твърдите връзки съдържат системния адрес на който се намира даден файл върху твърдия диск. Могат да сочат към файл от същата файлова система. Работата с твърдата връзка е еквивалентна с работа със самия файл.

`ln file1 hlink1` - създава твърда връзка с името hlink1 към файла file1

`cat hlink1` - изчита връзката; еквивалентно на отваряне на файла

`ls –li` - опцията `–i` показва твърдите връзки. Чрез нея се извеждат номерата на i-възела на всеки файл

Символните връзки съхранява пътя до файла във вид на URL. Могат да сочат към локални или отдалечени файлове. Символните връзки започват с l , когато стартираме ls –l. Символните връзки са оцветени в светло зелено.

`ln –s file2 slink2` - създава символна връзка с името slink2 към файла file2

`readlink slink2` - показва пътя до файла

`cat slink2` - отваряме файла чрез символна връзка slink2

3.4. Изтриване на файлове и директории

`rmdir <директория>` - изтрива празна директория

`rm <файл>` - изтрива фаил

`rm –r <директория>` - изтрива директория рекурсивно, т.е. с всички поддриректории и файлове

`rm –ri <директория>` - изтрива с питане за всеки файл от директорията

`rm –rf <директория>` - изтрива пълна директория без питане !!

Ако искаме да изтрием фаил с името –f може да ни създаде затруднение, т.к. rm –f ще се интерпретира като нормална команда и –f ще бъде флаг на нея. Решението е да се използва по-пълното име на файла (**./-f**) или да се използва аргумента -- на командата rm, който и казва, че всичко след него са имена на файлове, а не опции (**`rm -- -f`**)

`touch␣\␣'-f'`

`rm ␣ --␣\␣'-f'`

3.5. Работа със текст в обвивката:

`cat /etc/passwd`

//Файлът passwd е кунфигурационен файл и съдържа всички потребители в системата (повечето служебни). Използва се следния запис, където всеки ред е потребител и параметрите са разделени с двуеточие:

username : x : UID : GID : full name and description : home dir : shell

x- това поле не се използва, по рано са се съхранявали хешираните пароли (сега са във файла shadows)

UID - User ID

GID - Group ID

shell - подразбиращата се обвивка, най-често за реалните потребители е /bin/bash

//

`head -5 /etc/passwd`

`tail -5 /etc/passwd`

`nl /etc/passwd` - номерира редовете (number line)

`grep root /etc/passwd – търси по шаблон (например думата root)`

`grep [M] /etc/passwd -връща всички редове които съдържат главно F`

`за по сложни шаблони се използват регулярни изрази (RegEx)`

`wc –w /etc/passwd -- преброява колко думи (разделителя е интервал) има в дадения файл`

`wc –c /etc/passwd Символите`

`wc –l /etc/passwd Редовете`

```
find  <име на дир> -type f –name <име на файл> -print
```

търси дали даден файл(f) или директория(d) се срещат като се почне от начална <име на дир>

```
find /var –type f –name messages -print
```

Но ще ни трябват права за да претърсваме в някои директории. Затова може да пробваме:

```
find /etc –type f –name passwd -print
```

3.6. Комбиниране на команди

```
pwd;ls;whoami     -- групираме
```

Функции OR-IF и AND- IF

or –if:

```
ls || pwd   -- ще се изпълни само първата
```

```
lsssss||pwd --
```

and –if:

```
ls && pwd  -- и двете
```

```
lssssss&&pwd -- нито една
```

Използване на конвейр |

При конвейра (pipe, pipeling) резултата от една команда се подава като вход на следваща команда.

```
ls | wc –w
```
– Резултата от командата ls е списък с имена на файлове и директории, който се подава на wc -w, която от своя страна ще преброи тези имена и това ще е резултата

```
ls | sort
```

```
ls | sort –r
```

```
cat /var/log/syslog | less
```
- ако е много дълъг файла ще го изведе на няколко екрана

Командата less разбива резултата на няколко екрана. Изхода от нея е с Q от клавиатурата.

// Файлът syslog съдържа всички системни логове за предупреждения и грешки.

ако нямаме позволения влизаме като администратор или изпълняваме с:

```
sudo cat /var/log/syslog | less
```

==========

Позволения за достъп до файловете (File Permissions):

За достъп се разпознават следните потребители:

- собственик - този който е създал файла
- група на собственика (потребителите който принадлежат на групата на собственика)
- всички останали

Потребителя **root** има пълни права.

Правата са за четене, запис и изпълнение:

Означение	**Право**
r	read
w	write
x	execute
-	липсва

Правата към даден файл или директория могат да се проверят чрез командата `ls␣-l`

От листинга правата се отнасят към първата колона. В таблицата са дадени примерни стойности за един файл и една директория.

Тип на файла	собственик	група	всички останали
-	rwx	rw-	r--
d	rw-	r-x	---

```
//Типа за обикновените файлове се означава с '-' . Други типове за файловете са
l-символна връзка, b-блоково устройство , s-символно устройство и др.
```

Промяната на правата може да се извърши от собственика чрез командата change mod:

```
chmod <value> filename
```

Където value е режима и може да се зададе с цифри или със символи.

I. Задаване чрез цифрови стойности

Ако искаме да зададем на filename следните права:

- за собственика: rw-
- за групата r-x
- за всички останали: r--

използваме съответствието, че където има право е 1, а където няма е 0 и преобразуваме до 10-тично число:

r w -	r - x	r - -
1 1 0	1 0 1	1 0 0
110=> **6**	101=> **5**	100=> **4**

Изпълняваме :

```
chmod ␣ 654 ␣ filename
```

Проверяваме чрез:

```
ls ␣ -l
```

Ясно е че :

`chmod ␣ 777 ␣ filename` – ще зададе пълни права за всички потребители

`chmod ␣ 000 ␣ filename` – ще махне всички права

II . Задаване на правата чрез символи:

- символи за потребители: u (user), g(group), o (other), a (all)
- режим : + (задаване), -(премахване), = (приравняване)
- достъп: r,w,x

Примери:

u+wx - задаване права на собственика за запис и за изпълнение

go-x - премахване на групата и на всички останали правото за изпълнение

g+x,o+w - задаване на групата право да изпълнява и на всички останали да могат да четат (интервал след запетаята не трябва да има)

a= премахване на всички права за всички

a=x задаване на всички да имат право **само** да четат

a=rwx - задаване на пълни права за всички

Горните примери се изпълняват по следния начин:

```
chmod u+wx  filename
```

Освен правата съществуват и някои допълнителни флагове като: sticky bit, setuid, setgid.

Правата могат да се задават и чрез графичната система или някои програми.

==========

4. Работа с текстови редактори и файлови мениджъри:

4.1. Работа с редактора vim

```
sudo apt-get install vim
```

`vim` - отваряне на текстовия редактор

Редактора има два режима : команден и за въвеждане.

За да зададем режим за въвеждане натискаме клавиша "`i`" - за insert или "`a`"-за append от клавиатурата.

-> пишем няколко реда и за запазим написаното като файл минаваме в команден режим с ESC. Тогава може да създадем файла:

```
:w  myvimfile
```

-> може да напишем още някой ред и да съхраним:

```
:w
```

Ако искаме да излезем:

`:q` - изход, но преди това трябва да сме записали

`:q!` - изход без да запишем последните промени

`:wq` - запис и изход

Ако искаме да отворим файла за редакция с vim:

```
vim   myvimfile
```

--

```
vim  –R myvimfile
```
- отваря файла в режим Read Only

```
view  myvimfile
```

4.2. Други текстови редактори:

nano,pico, view, joe,ex

4.3. Файлови мениджъри:

```
mc
```
- отваряне на файловия мениджър Midnight commander

```
sudo apt-get install mc
```

други:

sudo apt-get install gnome-commander

sudo apt-get install vifm

sudo apt-get install lfm

==========

5. Архивиране на файлове

Създаваме няколко файла:

```
touch  file1  file2
```

5.1 .TAR (type archive) - служи само за пакетиране на няколко файла в един, но не за компресиране

за да архивираме (compress) с tar формат

```
tar  –cf  archive.tar   file1  file2
ls
```

нека да разархивираме (extract)

```
rm    file1    file2
tar   –xf   archive.tar
ls
```

5.2. За архивираме с GNU Zip - един файл

```
gzip file1
ls
```

разархивиране:

```
gunzip file1.gz
```

5.3 За архивиране с BZ2 - само за един файл

```
bzip2  file2
bunzip2   file.bz2
```

Обикновено архивите първо се пакетират с tar и след това се компресират с gz или с bz2

`gzip  archive.tar` – ще създаде архива `archive.tar.gz`

5.4 Архивиране с RAR

Необходимо е да се инсталират програмите rar и unrar

```
sudo apt-get install rar
sudo  apt-get install unrar
```

За да добавим файловете file1 и file2 към архива arc.rar изпълняваме:

```
rar a arc.rar   file1  file2
```

Разархивирането става с командата unrar:

```
unrar e arc.rar
```

==========

6. Администриране на процеси

За да предгледаме всички (`-e`) стартирани процеси може да използваме следните команди:

```
ps
ps -eF
ps -ely
ps  -ef
```

В последния листинг има :

UID - user ID,

PID - process ID

PPID - parent process ID

TTY - терминал (tty7 е графичната среда)

CMD - името на процеса (тези в правоъгълни скоби са нишки на ядрото)

Може да се използва и BSD синтаксис:

```
ps  ax
ps aux
```

За да се изпечатат процестие в дървовиден вид (родителски - дъщерни):

```
ps -ejH
ps  axjf
pstree
```

Ако от командите резултата в терминала е много като текст, може да го разделим на няколко екрана чрез командата `less` и използване на конвейр (|)

```
ps  -ef  | less
```

6.1. Статус на процес (от страницата: man ps):

PROCESS STATE CODES

Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a process:

D - uninterruptible sleep (usually IO)
R - running or runnable (on run queue)
S - interruptible sleep (waiting for an event to complete)
T - stopped, either by a job control signal or because it is being traced.
W - paging (not valid since the 2.6.xx kernel)
X - dead (should never be seen)
Z - defunct ("zombie") process, terminated but not reaped by its parent.

For BSD formats and when the stat keyword is used, additional characters may be displayed:

< - high-priority (not nice to other users)
N - low-priority (nice to other users)
L - has pages locked into memory (for real-time and custom IO)
s - is a session leader
l - is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
+ - is in the foreground process group.

Нека да стартираме процес от друг терминал, например tty1 - > Ctrl+Alt+F1. Влизаме първо с локалния потребител и после като root (`sudo su –`) и стартираме програмата Midnight Commander с командата `mc`.

После се връщаме в терминал от графичната среда Ctrl+Alt+F7

```
ps –ef | grep mc
```

```
ps –ef |grep –v root
```

Спираме програмата Midnight Commander от първия виртуален терминал и излизаме от сесия (Ctrl+D).

6.2. Манипулиране с процеси.

За управление на процесите най-често се използва изпращане на сигнали до тях. Сигналите се изпращат до даден процес чрез командата `kill`, като се знае номера на процеса (PID).

Ако искаме да прегледаме списък с възможните сигнали използваме:

```
kill –l
```

или

```
man  7 signal
```

За прекратяване на процес се използва SIGNKILL -9

Например нека да стартираме командата cat във фонов режим

```
cat&
```

Чрез командата ps може да проверим PID на cat и съответно да я прекратим с:

```
kill  –9   catPID
```

Когато трябва да се унищожат всички инстанции на даден процес се използва `killall` и неговото име.

Сигнали към даден процес могат да се изпращат и от клавиатурата, където (Ctrl + C) - > 2)SIGINT , (Ctrl + D) - > 3)SIGQUIT.

Друг често срещан вариант за наблюдение на процесите е чрез командата

```
top
```

Представя се списък с изпълняваните процеси, който се обновява през 3 sec. Колоната PR указва приоритета на процеса, като "-20" е най-високия, а "20" е най-ниския.

За Ubuntu процесите могат да се разглеждат и управляват в графичната среда чрез System Monitor.

==========

6.3. Процеси във фонов режим - daemons

Това са процеси които се изпълняват във фонов режим и обикновено изпълняват определена услуга.

Значението на думата daemon означава дух покровител и е неправилно превеждано като demon.

Най-популярния от този тип процеси е cron, чрез който се изпълняват програми по определен график. Конфигурационните файлове на cron се наричат crontab (от cron table) и обикновено се намират в /etc/crontab или за отделните потребители в /var/spool/cron.

Нека редактираме crontab файла за даден потребител (local):

```
crontab –u local –e
```

Може да изберем с кой текстов редактор да редактираме файла. Примерно `vim`.

Във файла всеки един ред задава график за изпълнение на една програма (или комнада). Графика се задава от първите пет символа, където:

```
┌──────────── min (0 - 59)
│ ┌────────── hour (0 - 23)
│ │ ┌──────── day of month (1 - 31)
│ │ │ ┌────── month (1 - 12)
│ │ │ │ ┌──── day of week (0 - 6) (Sunday to Saturday;
│ │ │ │ │                 7 is also Sunday)
│ │ │ │ │
│ │ │ │ │
* * * * * command to execute
```

Всички редове започващи с # са коментари за конфигурационните файлове.

Графика може да се зададе и чрез макроси:

Entry Description Equivalent to	Entry Description Equivalent to	Entry Description Equivalent to
@yearly (or @annually)	Run once a year at midnight of 1 January	0 0 1 1 *
@monthly	Run once a month at midnight of the first day of the month	0 0 1 * *
@weekly	Run once a week at midnight on Sunday morning	0 0 * * 0
@daily	Run once a day at midnight	0 0 * * *

@hourly	Run once an hour at the beginning of the hour	0 * * * *
@reboot	Run at startup	N/A

* означава всички. Но не е добре да се поставя звезда в първото поле освен ако не искаме да се изпълнява всяка минута.

`45 10 * * 1-5` означава ”10:45 от понеделник до петък”

при полетата за деня от седмицата и деня трябва да се внимава, например:

`0,30 * 13 * 5` означава ”всеки половин час в петък и всеки половин час на 13-то число от месеца ”

Полето за команда е всяка валидна команда от обвивката. Може да сложим списък от команди, напр. (`команда1; команда2;команда3`)

Пример:

да напишем в crontab файла:

```
*  *  *  *  *   env DISPLAY=:0   gnome-calculator
```

и след това излизаме от vim с ESC като пишем :wq

Това ще стартира на всяка минута калкулатора което не е особено смислено, но показва че е възможно да се стартират и графични програми .

спирането на cron за дадения потребител (local) става с:

```
crontab  -u  local  -r
```

==========

6.4 Нива на стартиране (runlevels)

Използват се за указване в какъв режим да се стартира системата : еднопотребителски , многопотребителски, без графична среда и т.н. Обикновено не се ползва вече. Използваха командите init и telinit за превключване на различните нива, а проверката за ниво е чрез who -r.

За Ubuntu нивото на старитране по подразбиране е зададено във конфигурационния файл :

```
/etc/init/rc-sysinit.conf
```

Най-често се налага спирането на графичната среда с цел отстраняване на някои проблеми. Това може да стане и без да се минава в друго ниво на стартиране.

Пример за спиране и пускане на X Server за Ubuntu. При Ubuntu се използва LightDM (display manager):

1. Ctrl+Alt+F1

2. Login -> local user
3. `sudo service lightdm stop`

(check with Ctrl+Alt+F7)

4. правим необходимите настройки

5. `sudo service lightdm start`

Ниво 0 е за спиране на системата, но по най-грубия начин. По приемлив вариант е чрез:

`shutdown –h +1` - спиране на системата след 1 минута

==========

7. Администриране на потребители

Оникновено администрирането на потребители се прави само от root или в частност от потребителите в групата sudo.

Отново разглеждаме файлът `/etc/passwd`, който представлява списък с потребители, които системата разпознава. Тя го чете, когато някой потребител влиза в сесия – от него взима UID идентификатора на потребителя и домашната му директория. Всеки ред представлява запис за потребител, като всяко поле е разделено с двоеточие

Добавянето на потребители може да става чрез командата `useradd` или чрез `adduser`. Разликата е че `useradd` е компилирана програма, докато `adduser` е perl скрипт за по-лесно и разбираемо използване. В себе си `adduser` използва `useradd`. В повечето дистрибуции се препоръчва използването на adduser.

//По-долните команди е необходимо да се изпълняват от root или чрез sudo.

`useradd –m newuser` - добавяме нов потребител с домашна директория

`passwd newuser` - добавяме парола за потребителя (нека да е същата като на локалния потребител)

`cat /etc/passwd` - проверяваме

`ls /home/newuser`

`ls –l newuser`

// vipw - редактира файлът passwd

`groups newuser` -проверяваме коя е текущата група на потребителя

`groupadd new` - добавяме нова група с името new

// gpasswd new добавяме парола за новата група

отваряме Midnight Comander

`sudo mc`

отиваме в директорията /etc и отваряме файлът `group` с F4 за редакция

задаваме новия потребител да бъде към групата

….

`new: x: GID: newuser` F2 – SAVE, F10 - exit

….

проверяваме

```
groups newuser
```

```
sudo mc
```

отиваме във файла с хешираните пароли:

```
/etc/shadow
```

Когато се създава парола на потребител тя се хешира с криптогравска хеш функция до определен стринг. Всеки път когато влиза потребител написаната от него парола е хешира отново и се проверява с хеша във файла. Ако съвпада се отваря сесия за потребителя. Има няколко начина за хеширане на пароли:

`MD5` започва с 1 - Изключително слаба хеш функция.

`NIS`

`Blowfish` започва с $2a$

`SHA512` започва с 6

За потребителя root при Ubuntu има символа `!`, т.е. няма създадена парола! За локалния потребител и за newuser хешовете са различни, въпреки че използвахме еднакви пароли. Това е така, тъй като към паролата се добавя допълнителен стринг (най-често username) преди хеширане.

Ако искаме да забраним новия потребител:

- слагаме звезда (*) пред хеша на дадения потребител, съхраняваме и излизаме

--> пробваме за да се убедим, че няма да успеем да се логнем.

За изтриване на потребител може да използваме командата `userdel` или `deluser` (аналогична на `adduser` и отново е perl скрипт).

```
userdel newuser
```

```
rm –r /home/newuser
```

```
groupdel new
```

Вариант с `adduser` и `deluser`:

```
sudo adduser newuser
```

`sudo usermod -aG sudo newuser` - добавяме `newuser` към групата `sudo`

`sudo deluser --remove-home newuser`

==========

8. файлови системи

В Linux най-често са използвани файловите системи ext3FS и ext4FS.

За да разгледаме монтираните устройства използваме:

`df -hT`

или

`sudo lsblk -f`

Дисковете които са sata се откриват по следния начин:

`/dev/sda1` - sata устройство (хард диск) 'a' , 1-ви дял

Ако имаме няколко хард диска те ще се откриват с имената: sda, sdb, sdc, sdd и т.н.

Дяловете на един хард диск са номерирани от 1. По номера се определя и дали дадения дял е главен (primary) или логически (logical). Главните дялове са с номера от 1 до 4. Логическите започват от 5 и всички се съдържат в един главен дял наречен Extended.

`sudo parted /dev/sda 'print'` - Отпечатване информация за всички дялове на диска sda. (Дяловете са с номера +1)

За да бъде възможно четенето на информация от даден носител той трябва да бъде монтиран. Монтирането е термин характерен повече за Unix системите и означава, че даден носител може да бъде използван ако се укаже неговата файлова система. Монтирането става в определена директория от където да се чете. Стандартната точка за монтиране е в директорията /mnt, а /media се използва за преносимите устройства (usb hdd и usb flash drive).

Например ако на хард диска има друга операционна система (Windows) и знаем, че дяла /dev/sda1 е с файлова система ntfs и там се намира C:, можем да го монтираме в /mnt

`sudo mkdir /mnt/C` -създаваме точка за монтиране

`sudo mount -t ntfs-3g /dev/sda1 /mnt/C`

NTFS-3g е open source драйвер за разбота с NTFS дялове.

След приключване на работа устройтвото трябва да се демонтира:

`sudo umount /dev/sda1`

При преносимите устройства (флаш-памети и др.) се използва автоматично монтиране, като по подразбирене точката на монтиране е директорията /media. Също така, когато направим инсталация на Linux при вече съществуваща инсталация на Windows, ще бъдат автоматично монтирани ntfs дяловете от Windows.

Ако имаме инсталация на Linux върху виртуална машина, то за нея се отделя виртуален диск. Принципно не е възможно да се монтират дялове от хоста и затова горния пример с монтиране и демонтиране на C няма да е възможен. Нека направим пример за монтиране и демонтиране на директория когато Linux е инсталиран на виртуална машина (Virtual Box), а ОС на хоста е Windows. Създаваме директория в Windows с името HostDir и в нея слагаме един тестов файл test.txt.

От VBox -> Devices -> Shared Folders Settings -> Add Share -< посочваме пътя до HostDir>- ok

За да използваме файловата система (vboxsf) за VBox трябва да инсталираме допълнително:

VBox -> Devices -> Insert Guest Additions CD Image -> Install (Run) -> Restart

След това отиваме в терминала на Linux :

```
cd  ~
```

```
mkdir   FromHostDir
```
- създаваме точка за монтиране (директория от която са четем)

Монтираме с файлова система vboxsf :

```
sudo mount –t vboxsf HostDir  FromHostDir
```

Можем да променим нещо във текстовия файл test.txt за да проверим дали се отразява на хоста.

```
cd FromHostDir
```

```
cat >test.txt
```

```
sudo umount  FromHostDir
```
- демонтираме

Ако искаме може да изтрием точката на монтиране:

```
rm – r  FromHostDir
```

За проверка и възстановяване на файлови системи може да се използва командата fsck.

```
sudo fsck   /dev/sda6
```

За да изпълним fsck за дяла /dev/sda6 той трябва да бъде демонтиран. Командата fsck връща стойности по които се определя текущото състояние.

```
ls  /sbin/fsck*
```
- за да прегледаме кои файлови системи може да проверяваме

8.1. Системи за начално зареждане (boot loaders)

Стандартната система за начално зареждане в Linux е GRUB (GNU GRand Unified Bootloader). Чрез него има възможност да се зареждат и други операционни системи, като Windows например. Когато ОС са повече от 1, GRUB ще конструира Boot Menu.

Главните конфигурационни файлове за GRUB се намират в директорията /boot/grub

Един от тези файлове е grub.cfg, но обикновено не се редактира директно.

За промяна на настройките се използва файла /etc/default/grub

Най-често това са редовете:

```
GRUB_DEFAULT=4
GRUB_TIMEOUT=6
```

чрез които се указват номера на ОС по подразбиране и времето което да се изчака преди автоматичното зареждане.

Номерата на ОС са номерирани от 1, вместо от 0.

След промяна по файла се изпълнява:

```
sudo update-grub
```

---

9. Мрежови настройки:

```
host   www.google.com
```

Една от основните команди е

```
ifconfig
```
- interface configuration

Чрез нея могат да се разглеждат конфигурираните мрежови устройства и интерфейси, а също и да се настройват, чрез съответните аргументи и опции.

eth0 : Ethernet мрежова карта (в случая е първата - 0)

lo : localhost

enp0s3 : Ethernet network peripheral # serial #

ppp0 : ppp устройство

ippp0 : ISDN ppp устройство

wlan0 : wifi

ra0 : безжична мрежа

Понякога се използва задаването на виртуални мрежови интерфейси – за да се хостват повече уеб сайтове

eth0:0 , eth0:1 и т.н.

Командата ifconfig има разновидност и за wireless карти

`iwconfig`

Портове:

Портът е абстракция на физическия адрес през който се осъществява комуникацията. Портовете са достъпни чрез своя номер от 0 до 65535. Портовете които са с номер по-малък от 1024 обикновено са запазени за системни нужди и не се използват за други приложения.

Сокетът е абстракция на мрежов софтуер, осигуряващ входно изходна комуникация за дадено приложение. Сокътет се създава при валиден IP- адрес и номер на порт. Към един порт могат да се създадат много сокети, позволяващи на много клиенти да използват една услуга. Комуникацията през сокети става чрез потоци.

резервираните портове , могат да се видят във файла:

`cat /etc/services | less`

21 FTP

23 telnet

22 ssh

80 http

25 mail SMTP

Портовете от 0 до 1023 са запазени

`ping <хост>` ICMP заявка за exo (Internet Control Message Protocol)

Ctrl+C за край

`traceroute –n <хост>` - показва пътя на пакетите

`–n` използва числови стойности

`netstat –avn` - показва таблица на връзките с другите компютри

`sudo ifconfig eth0 down` -изключваме мрежовата карта

`sudo ifconfig eth0 up` -включваме я

Проверяваме рутиращата таблица

`route –n` Ако на мястото на gateway имаме * т.е. няма зададена

Чрез командата route може да се добави default gateway

```
route add default gw 192.168.3.1
```

Други приложения:

ssh - Secure Shell Protocol for Remote Login

VNC

Пример за SSH връзка до отдалечен компютър:

Ако имаме два компютъра за примера ще използваме следните означения: Comp1 и Comp2. Съответно локалните потребители ще означим с local1 и local2. А ip адресите: 192.168.1.1 (Comp1) и 192.168.1.2 (за Comp2).

За двата Comp1 и Comp2 инсталираме ssh чрез:

```
sudo apt-get install openssh-server
```

Настройките за SSH се намират във файла `/etc/ssh/sshd_config`. От там можем да променим номера на порта и др. характеристики.

Възможно е да направим връзка директно от Comp1 чрез:

```
ssh   local2@192.168.1.2
```

но по-добре да използваме криптиран канал чрез публичен и частен ключ. За целта най-често се използва криптографския алгоритъм RSA.

Затова от Comp1 създаваме двойка ключове като чрез:

```
ssh-keygen  -t rsa
```

//ще изиска да въведем името на файла където да се съхранят, но може да използваме тези които ни предлага. Ще трябва да въведем и passphrase като парола за идентификация на ключа.

Ключовете ще се съхранят съдържат в /home/local1/.ssh, като:

id_rsa - съдържа частния ключ, който си остава при Comp1

id_rsa.pub -съдържа публичния ключ, който и ще пратим на Comp2

Изпращането на публичния ключ от Comp1 на Comp2 става чрез:

```
ssh-copy-id   local2@192.168.1.2
```

//ще ни поиска паролата за local2

Вече може да се логнем на Comp2 със local2 от Comp1:

```
ssh   local2@192.168.1.2
```

и да променим правата за достъп до файла с публичните ключове (authorized_keys) на local2

```
cd  /home/local2/.ssh
```

```
chmod 600  authorized_keys
```

Изхода става с

```
exit
```

Когато трябва да достъпим Linux система от Windows със SSH се използва програмата PuTTY.

Bash скриптове

1. Въведение

Всички UNIX и Linux системи позволяват писането на програми, които да се изпълняват (интерпретират) от терминала. Тези програми могат да са на различни езици, включително и на езици от по-високо ниво като Perl, Ruby, Rexx, C, Java и др. Най-често използваните обвивки са следните:

- Bash (Bourne Again Shell)
- sh (Bourne Shell – това е най-старата от всички)
- csh (C Shell)
- tcsh (Tenex C shell)
- ksh (Korn Shell)
- zsh (Z shell)
- ash (Almquist shell)
- rsh (Remote shell)

Нека изпълним от терминала следните редове един след друг:

```
a=vision
b=”tele$a”
echo␣$b
```

Това ще изведе стринга `television`. Чрез знака `$` се извлича стойността на дадена променлива. За удобство със символа ␣ ще означим интервала, тъй като неговото произволно използване може да бъде интерпретирано погрешно. Например:

x=10

ще означава, че на променливата x се присвоява целочислената стойност 10, докато

x= ␣10

се разбира, че bash трябва да интерпретира командата „x=“ с аргумент 10, а в някои случаи, че x присвоява стринга „␣10“, което няма да е коректно при използването му в някой брояч.

Нека се върнем към предния пример. Вместо да пишем програмата в терминала е по удобно да се изпълни от файл. Може да ги запишем във файл с името script, чрез подходящ текстов редактор:

```
#!/bin/bash
a=vision
b=”tele$a”
echo␣$b
```

Всички редове които започват със знака `#` означават коментари, докато редовете с `#!` означават директива. За този случай директивата показва, че следващите редове от файла ще се изпълняват от програмата `bash`. За да направим файла изпълним е необходимо да дигнем правото за изпълнение:

```
chmod␣u+x␣script
```

Този текстов файл се нарича скрипт за обвивката и може да се изпълни директно от терминала, стига да се намираме в същата директория, чрез:

```
./script
```

Файла може да се изпълни и чрез:

```
sh␣script
```

Ако искаме програмата Hello world ще използваме следните редове **:**

```
#!/bin/bash
echo␣"Hello, World"

================================================
```

2. Променливи

Променливите в shell скриптовете не са строго типизирани. Типа се определя при инициализация или изрично при деклариране. Имената на променливите е прието да се пишат без използването на служебни символи.

За присвояване на стойност на променлива се използва оператора за присвояване (знака за равенство (=)). Пример:

```
x=10
```

Отново да обърнем внимание, че не трябва да се използват интервали около оператора за присвояване. Ако сложим интервал след него `(x=␣10)`, bash ще се опита да изпълни команда `"x="` с аргумент `10`. Извличането на стойност на променлива става чрез знака за долар `($)`. Стойностите на променливите могат да бъдат произволни символи, но ако се използват символи, които се използват от обвивката, възникват проблеми. Тези символи са: интервал, точка (.), знак за долар ($), по-голямо (>) и по-малко (<), (|), (&), (*), ({) и (}). Тези знаци могат да се използват като стойност на променлива ако са поставени в двойни или единични кавички или с обратно наклонени черти (escape). С двойни кавички може да се използват всички запазени знаци без знака за долар ($). За да се включи в стойността на променливата може да се използва обратно наклонена черта преди него (\$) или да оградите стойността на променливата с единични кавички. Следващия пример илюстрира всичко описано до сега.

```
#!/bin/bash
num1=10
num2=20
msg1="$num1 < $num2 & $num2 > $num1" # Тук ще се покажат всички #знаци без знака за долар,
# който използваме за да се обърнем към стойността на
# променливата.
echo $msg1

msg2="\$100 > \$10" # Тук използваме обратно наклонена черта за да #покажем знака за долар
echo $msg2

msg3='Here we can use all of these symbols: ".", ">", "<", "|", "$", etc'
# по този начин всички знаци ще се покажат на екрана, единственият #символ, който не може да се използва е единична кавичка (')

echo $msg3

# При всички примери по-горе не е нужно стрингът да се слага в #променлива.

echo ""\$msg1" is not needed to print '$num1 < $num2 & $num2 > $num1'"
```

Изпълняваме скрипта от файла с
`./script`

Ако искаме да присвоим резултата от дадена команда на променлива, трябва да се огради командата в обратно (ляво) наклонени апострофи.

Примерно:

```
s=’ls␣/home/s503’
echo␣$s
```

ще изведе просто стринга заграден в единичните кавички, но ако сложим стринга в обратно наклонени апострофи:

```
s=`ls␣/home/s503`
echo␣$s
```

тогава резултата от командата ls ще се присвои на променливата. В случая единичните кавички могат да се използват като друго име на командата:

```
s=’ls␣/home/s503’
$s
```

Тук липсва `echo` и извикването само на `s` ще бъде резултата от командата `ls␣/home/s503`.

Ако искаме да използваме резултата от дадена команда в стринг е необходимо да се загради командата в скоби и да се постави знака за долар преди тях `($(ls))`.

```
#!/bin/bash
echo "The date is $(date)"
```

Ако се изпълни друг скрипт от текущо изпълняващият се, текущият скрипт спира изпълнението си и предава контрола на другия. След изпълнението му се продължава първия скрипт. При този случай всички променливи, дефинирани в първия скрипт не могат да се използват във втория. Но ако те се експортират, променливите от първия те ще могат да се използват във втория. Това става с командата `export`.

```
#!/bin/bash
# Това е първият файл.
var=100
export var
./script-a
# EOF
```

```
#!/bin/bash
# Вторият файл.
echo "The value of var is $var"
# EOF
```

Изпълнението на скрипта (обърнете внимание, че вторият файл трябва да е `script-a`, ако използвате друго име, сменете името на файла в първия скрипт).
Стартираме, чрез:

```
./script
```

Друг начин за деклариране на променливи е командата `declare`. Синтаксисът на тази команда е:

```
declare -тип име-на-променливата
```

Типовете променливи са:

```
-r   - readonly
-i   - integer (цяло число)
-a   - array (масив)
-x   - export
```

Пример:

```
#!/bin/bash
declare -i var # декларираме променлива от тип integer
```

```
var=100
echo $var
declare -r var2=123.456 # декларираме readonly променлива
echo $var2
var2=121.343 # опитваме се да променим стойността на var2
echo $var2   # стойността на var2 ще е все още 123.456
```

Изпълнението на скрипта:

```
./script
```

Повече информация можете да се прочете от bash manual pages (man bash).

==========

3. Shell позиционни променливи

Когато се изпълнява дадена команда от терминала, тя може да включва аргументи, които се записват след нея, разделени с интервал. Тъй като командата е програма, тези аргументи се достъпват от нея. Наричат се позиционни променливи или аргументи на скрипта. Стойностите са от 1 до 9 и могат да се използват чрез `$1, $2, … ,$9`.

Например за командата:

```
command␣arg1␣arg2␣arg3
```

позиционните променливи ще имат съответните стойности:

`$0` – е името на командата или command

`$1 – arg1`

`$2 – arg2`

`$3 – arg3`

Освен позиционните променливи има и специални променливи, които се използват за аргументите на скрипта за обвивката:

`$*` - всички аргументи от командния ред

`$@` - всички аргументи от командния ред поотделно

`$#` - броят на аргументите от командния ред

Разликата между `$*` и `$@` се състои в това, че при `S*` аргументите се вземат заедно като един стринг, докато при `$@` те са поотделно и могат да се ползват като масив.

Нека разгледаме следния пример, чрез който да илюстрираме използването на позиционните променливи:

```
#!/bin/bash
echo "The first argument is $1, the second $2"
echo "All arguments you entered: $*"
echo "There are $# arguments"
```

И да изпълним скрипта чрез следните аргументи от терминала:

./script␣arg1␣arg2␣arg3␣arg4

Да разгледаме още един пример. Да се напише скрипт който преброява файловете в дадена директория, посочена като аргумент.

```
#!/bin/bash
ls $1 | wc –w
```

съответно скрипта трябва да стартираме по следния начин:

```
./script /etc
```

Този пример не пълен, тъй като е коректно да се провери дали подадения аргумент е директория. Ако не сме подали директория, командата ls ще върне съобщение за грешка и след това (по конвейр), командата `wc –w` ще преброи думите в него.

В началото на тази точка споменахме, че позиционните променливи с които можем да работим са от 0 до 9. Какво се получава когато техният брой е по-голям. Стойности на позиционните променливи можем да зададем чрез командата set. Например

```
set `ls  /etc`
```

ще бъдат зададени толкова позиционни променливи колкото резултата върне командата `ls`, и съответно за директорията `/etc` техния брой ще е много повече от 9. В този случай се използва преместване на ляво чрез командата shift за да можем да изчетем всички променливи. Да обърнем внимание, че тук се използват отново обратно наклонени апострофи.

```
#!/bin/bash
set `ls /etc`  #задаваме стойности на позиционните променливи
#в случая техния брой се определя от броя на файловете и #директориите
в /etc
echo $1 $2 $3  #извеждаме първите 3
echo $#        #извеждаме техния брой
echo $*  #извеждаме всички
echo “SHIFT”
shift 2  #изместваме на ляво с две позиции
#отново извеждаме
echo $1 $2 $3
echo $#
echo $*
```

Изпълняваме с

```
./script
```

Ще забележим, че след изместването стойността на `$1` е предишната на `$3`, `$2` на `$4` и т.н. Старите стойности за `$1` и `$2` няма да се пазят и броя на всички `($#)` ще е с две по-малък.

4. Аритметични операции

За въвеждане от клавиатурата се използва командата `read`.

```
#!/bin/bash
echo -n "Enter a string: "
read str
echo "String you entered: $str"
```

За изчисление на аритметични операции се използват командите `let` и `expr`.

Чрез `let` могат да се сравняват две стойности на променливи, да се извършват аритметични операции, както и да се използват за управляващи конструкции на цикли. Синтаксисът на командата е следния:

```
let value1 operator value2
```

Възможните оператори, които могат да се използват са:

```
+     - събиране
-     - изваждане
*     - умножение
/     - деление
%     - остатък при деление
>     - по-голямо
<     - по-малко
>=     - по-голямо или равно
<=     - по-малко или равно
==     - равно
!=     - различно
&     - логическо И (AND)
|     - логическо ИЛИ (OR)
!     - логическо НЕ (NOT)
```

Тази команда има и втори запис, като се използват двойни скоби:

```
$ (( value1 operator value2 ))
```

Когато променливите са от един и същи тип, аритметичните операции могат да се използват директно без командата `let` (или без скобите).

Пример:

```
#!/bin/bash
echo -n "Enter the first number: "
read var1
echo -n "Enter the second: "
read var2
declare -i var3

echo ----------
echo "$var1 + $var2 = $(( $var1+$var2 ))" # за да се изчисли израза #в
двойните скоби, трябва да се сложи знака $
let res=$var1*var2
echo "$var1 * $var2 = $res"
var3=100
var3=$var3+10  # var3 е декларирана като integer и не е нужно да се
#използва let
echo "$var3"
```

Командата `expr` има подобно действие като `let`. Нейния синтаксис е следния:

```
expr value1 ‘operator’ value2
```

Допустими са следните операции: `+`(събиране), `-`(изваждане), `*`(умножение), `/`(деление), `%`(модул), като не е задължително да се слагат в единични кавички. Оператора за умножение трябва да се използва с наклонена черта `\*`. По същия начин се използват и скобите за по-сложни изрази `\(` и `\)`. Възможно е да се съставят и логически изрази, като се използват знаците: `=`, `!=`, `<`, `>`, `<=` и `>=`.
Примери:

```
expr␣5␣+␣2
expr␣5␣\*␣3
expr␣8␣/␣2
expr␣5␣\*␣\(␣2␣+␣3␣\)
expr␣7␣\!=␣5
expr␣3␣\<␣5
```

Командата `expr` може да се използва с някои функции, например за стрингове:

```
expr␣length “Some string”
expr␣index␣”Some string”␣”m”
```

Характерното за командата `expr` е, че тя извежда резултата в терминала, затова ако искаме да го присвоим на променлива трябва да се използват обратно наклонените апострофи:

```
a=5
b=3
i=`expr␣$a␣+␣$b`
echo␣$i
```

5. Условен оператор

Първото нещо което трябва да се разгледа това е какво представлява едно условие. Ако използваме оператора if (който го има в повечето езици) и го комбинираме с друга команда като условие, не е ясно как ще сработи. Например:

if ls

винаги ще е вярно, тъй като дори да не съществува директория, командата ls ще върне стринг за грешка, но exit кода (по късно ще бъде разгледан) ще е 0, което означава, че тя е завършила правилно. За тази цел, не само за оператора if, но и за други оператори като цикли, за условие се използва командата test. Тя има специфичен синтаксис и е команда която изследва. Командата test проверява различни условия и връща тяхната логическа стойност: 0(true), 1(false), >1(error) като exit код. Командата има вида:

```
test израз
```

или втория вариант:

```
[ израз ]
```

където *израз* може да включва променливи и константи, разделени със знаци за операции.

Има три класа знаци: знаци за отношения, знаци за логически операции и знаци за файлове.

Примерът

```
test -f fname
```

проверява дали посоченото име `fname` е име на файл, с `-d` за директория и други. Ако използваме съкратения синтаксис ще изглежда така:

```
[ -f fname ]
```

Командата test за отношение между две променливи се използва по следния начин:

```
test value1 -option value2
```

```
test string operator string
```

или чрез съкратения запис:

```
[ value1 -option value2 ]
```

```
[ string operator string ]
```

Обръщаме внимание, че при сравнение на стрингове се използва оператор, а не опция. Когато се сравняват променливи трябва да се взима тяхната стойност (чрез използването на знака $):

```
[ $a -gt 5 ]
```

Ще изброим само част от използваните операции за test:

=	еднакви низове
!=	различни низове
-z	нулева дължина на низ
-n	ненулева дължина на низ
-eq	равно (за цели числа)
-ne	неравно (за цели числа)
-gt	> (за цели числа)
-lt	< (за цели числа)
-ge	>= (за цели числа)
-le	<= (за цели числа)
-a	логическо И
-o	логическо ИЛИ
!	логическо НЕ
-f	обикновен файл
-d	директория
-w	файл с разширение за запис
-r	файл с разширение за четене
-x	файл с разширение за изпълнение

Останалите опции могат да се проверят чрез командата man в терминала:

```
man test
```

В началото на този точка споменахме за exit кода. При изпълнение програмите често могат да настъпят грешки. Всеки път, когато една програма завърши, се връща код за завършване наречен exit код. По подразбиране стойността 0 означава коректно завършване, а при грешка кода е различен от 0. Този код може да се провери, чрез стойността на специална променлива $?, която съдържа кода на завършване на последната команда. Тази променлива също може да се използва за проверка на резултата от командата test.

```
a=10
b="asdfg"
[ $a -eq 10 ]
echo $?   # exit code = 0

[ $b = "aaaaaa" ]
echo $? # exit code = 1
```

Условения оператор if има следния синтаксис:

```
if  условие
    then
        оператор1
    else
        оператор2
fi
```

Оператора if проверява exit кода на дадена команда, но най-често се използва командата test за условие. В секциите then и else може да се съдържат повече от един оператори.

Когато в else има следващ оператор if се използва ключовата дума elif:

```
if  условие
    then
        оператор1
    elif условие
        then
            оператор2
        else
            оператор3
fi
```

Може да запишем няколко оператора на един ред като използваме знака (;)

```
if  условие; then
    оператор1
    else
        оператор2
fi
```

Примери:

==========

Да се напише скрипт който проверява дали подаденото му име е име на файл или на директория:

```
#!/bin/bash
if␣[␣-f␣$1␣]
    then
        echo "$1 is a file"
    elif␣[␣-d␣$1␣]
        then
            echo "$1 is a dir"
        else
            echo "$1 is not a file or dir."
fi
```

Стартираме скрипта с подходящ параметър:

```
./script  /etc

./script  /etc/passwd
```

==========

```
#!/bin/bash
```

```
echo -n "Enter a string: "
read str1
echo -n "Enter a string: "
read str2
echo -n "Enter a number: "
read num1

if [ $str1 == "asdfg" ]; then
     echo "str1 = asdfg "
elif [ $str1 == "asdfg" ] && [ $str2 == "asdfg" ]; then
     echo "str1 and str2 = asdfg"
     else
         echo "str1 and str2 != asdfg"
fi

if [ -f "/etc/passwd" ]; then
    cat /etc/passwd
fi

if [ $num1 -eq 10 ]; then
     echo "num1 = 10"
elif [ $num1 -gt 100 ]; then
     echo "num1 > 100"
else
     echo "?!?"
fi
```

======

Тук се проверява резултата от дадена команда (в случая whoami) дали е равен на определен стринг. Обръщаме внимание, че се използват обратно наклонени апострофи (`).

```
if  [ `whoami` = “root” ]
     then  echo  “Hello, admin”
     elif [ `whoami` = “s503” ]
          then echo  “Hello, s503”
     else   echo “Hello, guest in  `pwd` ”
fi
```

====

Освен създаването на условия, чрез командата test е възможно използването на условни или аритметични изрази. Те могат да заместят условията за операторите if, while и until. Условните изрази се затварят в двойни квадратни скоби, като преди и след тях има празни интервали, например (за оператора if):

```
if [[␣условен израз␣]]
```

Условните изрази използват синтаксиса на условия чрез test, но те включват и логическите операции:

`&&` - И

`||` -ИЛИ

`!` -Отрицание

Пример:

```
if [[  $x  -gt  3   &&   $x  -lt   42  ]]
```

ще има стойност true , ако стойността на променливата x е между 3 и 42.

За да използваме аритметични изрази като условия, трябва да ги заградим с двойни кръгли скоби:

```
if ((i>10))
```

6. Оператори за цикли

Използват се циклите: **while**, **until**, for (**for-in**). Операторите while и until проверяват exit кода на дадена команда, но както и при оператора if се използва резултата от test като условие. Цикъла for-in се използва за обхождане на списък от стойности (или масив).

Цикъл while – изпълни докато условието е 0 (true):

```
while условие
    do
        оператори (или команди)
    done
```

Пример – въвеждаме числа, докато са по-големи от 5:

```
#!/bin/bash
i=10
while [␣$i␣-gt␣5␣]
     do
          read i
          echo "You enter:  " $i
     done
```

Пример – извежда числата от 1 до 9

```
#!/bin/bash
i=1
while [␣$i␣-lt␣10␣]
     do
          echo –n "… $i"
```

```
            i=`expr $i + 1`
      done
echo " "
```

Цикъл until – изпълни докато условието е 1 (false):

```
until условие
    do
        оператори (или команди)
    done
```

Пример – въвеждаме стринг, докато не въведем стринга "no":

```
#!/bin/bash
answer=yes
until [ $answer == "no" ]; do
      echo -n "Enter a string: "
      read str1
      echo "You entered: $str1"
      echo -n "Do you want to continue? "
      read answer
done
```

Цикъл for-in

```
for управляваща-променлива  in  списък-от-стойности
      do
            оператори (или команди)
      done
```

Примери:

```
#!/bin/bash
for  k  in 1 2 3 4 5 6 7 8 9
      do
            if  [  $k  -eq   3  ];  then continue;  fi
            if  [  $k  -eq   7  ]; then break;  fi
      echo -n  ". . . $k"
      done;  echo " "
```

======

```
#!/bin/bash
for archive  in   ~/*.tar
     do
          echo $archive
     done
```

==========

Може да използваме цикъла for да обходим позиционните променливи:

```
#!/bin/bash
for  x in  $1 $2 $3 $4
     do
          echo $x
     done
```

Но това може да се замести със служебната променлива $@ която взима всички позиционни променливи поотделно:

```
#!/bin/bash
for x in $@
     do
          echo $x
     done
```

Примера може да се стартира:

```
./script  arg1 arg2 arg3 arg4
```

==========

Да разгледаме още един пример, при който се извеждат всички файлове от дадена директория:

```
#!/bin/bash
set  `ls $1`
for a in $@
     do
          echo $@
     done
```

При този пример се използва командата set, чрез която се задават стойности за позиционните променливи. За да стартираме примера е необходимо да посочим аргумента $1, който е някоя директория:

```
./script   /home/s503
```

==========

Ако е необходимо да запишем резултата от изпълнението на даден цикъл във файл, може да използваме операторите > и >>.

> - изтрива предишното съдържание

>> - долепя в края

```
#!/bin/bash
nums="1 2 3 4 5 6"
for num in $nums
do
 echo $num
done > fnums
```

Или в комбинация с конвеир. В случая след приключване на цикъла се сортира резултата и се записва във файл:

```
#!/bin/bash
strings="asd  dse rdft  wert gty"
for str in $ strings
     do
          echo $str
     done | sort > fwords
```

Файловете можем да прочетем с:

```
cat fnums
cat fwords
```

7. Генериране на случайни числа

За целта се използва променливата на обвивката $RANDOM. Всеки път когато се използва се връща псевдослучайно число между 0 и 32767. Можем да променим интервала чрез аритметичните операции: %, +, -, *, /.
За да гарантираме, че променливата $RANDOM ще генерира различни последователности всеки път, трябва да я инициализираме с различни стойности. Например може да използваме стойността на date с опции +%s, която ще върне текущия брой секунди, изминали от началото на UNIX епохата (започваща в 00:00:00 ч. Гринуичко време на 1 януари 1970г.) .
Пример за скрипт който връща случайно число между 1 и 20:

```
RANDOM=`date  +%s`
let NUM=($RANDOM  %  20  +  1)
echo $NUM
```

8. Масиви

Масивите съхраняват всяка стойност в уникален, отделен, индексиран компонент, наречен елемент. Елементите се означават с индекси автоматично, като се започне от 0. Масивите в bash

могат да се създават по три начина: чрез инициализиране, чрез оператора declare и чрез задаване на стойности в скоби.
Обръщението към конкретен елемент на масив се извършва чрез името на променливата и индексния номер на елемента заграден в квадратни скоби „[]“. За да се видят самите стойности, съхранени в елементите на даден масив е необходимо да се деадресират. За целта се използват фигурни скоби “{}“.

```
#!/bin/bash
array[1]=a1
array[5]=abc
array[10]=123
echo "array[1] = ${array[1]}"
echo "array[5] = ${array[5]}"
echo "array[10] = ${array[10]}"
```

Създаване на масив, чрез присвояване на стойности зададени в скоби:

```
array=(␣a␣4␣wer␣35␣t7␣)
```

В този случай
array[0] = a
array[1] = 4
array[2] = wer
array[3] = 35
array[4] = t7

Може да създадем масив и чрез командата `declare` и опция `-a`:

```
declare -a array
echo -n "Enter some numbers separated by space: "
read -a array # с опция –a въвеждаме елементите на масив, като те са
#разделени с интервал
elements=${#array[@]} #elements ще съдържа броя елементи на масива
#  ${array[@]} съдържа елементите на масива поотделно. Може да
се  #използва за цикъл for-in  например:

for i in ${array[@]}; do
   echo $i
done
#извеждане с цикъл while
i=0
while [ $i -lt $elements ]; do
 echo ${array[$i]}
 let "i = $i + 1"
done
echo " "
```

Примери за използване на масиви в изрази:

```
array=(2 3 4 5)
let array[2]=7
((array[3]=9))
((array[4]=array[2]+3))
array[5]=`expr ${array[0]} + ${array[1]}`
```

9. Оператор case за условни разклонения

Пример – избор на извеждане:

```
#!/bin/bash
echo -n "Enter an option (l, s or al): "
read opt
case $opt in
l)
     ls -l;;
s)
     ls -s;;
al)
     ls -al;;
*) # other cases
     ls;;
esac
```

Пример – избор на цвят:

```
#!/bin/bash
clear
echo “Change terminal font color”
echo “--------------”
echo –en “[1]Red\n  [2]Green\n [3]Brown\n”
echo –en “[4]Blue\n [5]Purple\n [6]Cyan\n”
echo “[x] Exit”
echo –n “Enter: ”
read   char
case  $char in
1)  echo  -e  “\e[033;31m”;;
2)  echo  -e  “\e[033;32m”;;
3)  echo  -e  “\e[033;33m”;;
4)  echo  -e  “\e[033;34m”;;
5)  echo  -e  “\e[033;35m”;;
6)  echo  -e  “\e[033;36m”;;
7)  echo  -e  “\e[033;37m”;;
esac
```

10. Прихващане на сигнали с trap

Понякога е необходимо да се прихващат сигнали към даден скрипт. Това може да стане, чрез trap. Командата trap има следния синтаксис:

trap команда сигнал

и указва коя команда (или функция) да се изпълни при дадения сигнал. Сигнала се задава чрез име или номер.

Пример:

```
#!/bin/bash
trap sorry INT
sorry()
{
    echo "I am sorry. I cannot do that."
    sleep 3
}
for i  in 10  9  8  7  6  5  4  3  2  1  0;  do
    echo $i  seconds remains
    sleep  1
done
```

При стартиране на примера на всяка секунда се отброяват последователно числата от 10 до 0. Ако се опитаме да прекъснем изпълнението чрез комбинацията CTRL + C , чрез trap се извиква функцията sorry. Вместо с името на сигнала може да се използва неговия номер. За сигнала INT (Interrupt) номера е 2 и в горния пример можехме да използваме реда :

```
trap sorry  2
```

Сигнала с номер 9 е kill и той не може да бъде прихващан от trap.

За информация останалите номера на сигнали можем да проверим с:

```
man 7 signal
```

11. Функции

```
#!/bin/bash

func() {
 echo "Hello"
}

info() {
 echo "host uptime: $( uptime )"
 echo "date: $( date )"
}
func   # извикване
info   # извикване
```

Сортировка sleep sort:

```
#!/bin/bash
f() {
    sleep “$1”
    echo “$1”
}
while   [ -n “$1” ]
    do
        f  “$1” &
        shift
    done
wait
```

Стартираме с параметри - числа които искаме да се сортират. Тази сортировка разбира се не претендира за ефективност. При нея се използва оператора sleep, който изчаква толкова секунди колкото е стойността, и затова не е удачно да въвеждаме големи стойности:

```
./sleepsort   3 1 4 1 5 9
```

Някои основни понятия в ОС

Потребителски и защитен режим

Всеки процесор има някои инструкции, достъпни за изпълнение само от “привилегировани” програми. Това са инструкции за работа с паметта, установяване на таймера, определяне на вектори на прекъсване, достъп до рестриктирана памет, спиране на процесора.

Потребителските процеси не би следвало да могат да изпълняват тези инструкции.

Повечето процесори имат флаг в статус статус-регистър, указващ дали се изпълнява режима на работа.

Процесорът стартира в защитен режим и зареждането на boot се изпълнява в защитен режим. Зареждането на ОС и работата на ядрото се изпълнява в защитен режим.

Превключване между режимите

Ако процесорът работи в защитен режим, може да се превключи в потребителски посредством промяна на флага в статус-регистъра - изпълнявайко защитена инструкция. В потребителски режим тази инструкция е недостъпна.

Преминаването от потребителски в защитен режим се извършва чрез TRAP-инструкция. Като всяка инструкция тя се извлича, декодира и изпълнява.

TRAP-инструкцията предава управлението на сервисна процедура (TRAP service routine) чрез зареждане в програмния брояч (PC) на стартовия адрес на съответната процедура.

Ядро (kernel)

Ядрото централният компонент на ОС: главната програма, която управлява достъпа до ресурсите и планирането на процесите.

Примери за функциите, които ядрото на типична ОС изпълнява:

- Управление изпълнението на процесите: създаване, завършване и междупроцесна комуникация;
- Предоставя памет на изпълняващите се процеси; ако системата (КС) остане без достатъчно свободна памет, ядрото освобождава области от нея, записвайки съдържанието им във вторичната памет (swapping iили paging);
- Предоставя вторична памет за съхраняване и извличане на данни, осигурявайки различни права на достъп и взаимно изключване;
- Разрешава управляван от процесите достъп до устройствата (периферията) при различни права на достъп и взаимно изключване.

Системни примитиви (системни извиквания, system calls)

Интерфейса между приложенията и ОС се формира от съвкупността на системните примитиви (open, close, read, fork, execve, …). Системните примитиви използват механозма на TRAP-инструкциите за активиране на точно определена част от ядрото, която се изпълнява в защитен режим.

Изпълнението на системните примитиви включва: съхраняване на параметрите в системния стек, съхраняване на номера на системния примитив, и превключване чрез TRAP към защитен режим.

Пример: ***getpid*** в LINUX/Intel записва номер 20 в регистър ***eax*** и изпълнява INT 0x80, с което генерира TRAP. Ядрото съхранява състоянието на процеса и го възстановява при връщане от сървисната процедура.

Тези детайли са скрити от програмиста посредством библиотечни процедури, всяка от които съответства на системен примитив. Тези процедури съхраняват патраметри, инициират TRAP и пр., така че системните примитиви изглеждат като стандартно извикване на процедури.

Лекция 1.

Цели, функции и място на операционната система. Класове ОС.

1.1. Място на ОС

Съвременните компютърни системи включват един или повече процесори, основна памет, външна памет, таймери, терминали, принтери, мрежови интерфейси и прочие входно/изходни устройства. Управлението на тази част или hardware, се извършва от програмната част на компютърната система.

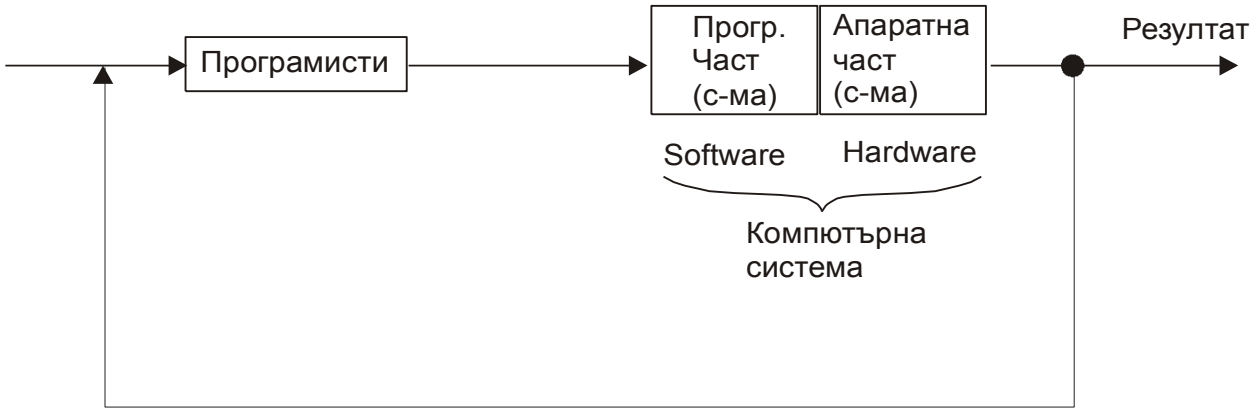


Рис. 1.1. Система за обработка на данни с общо назначение

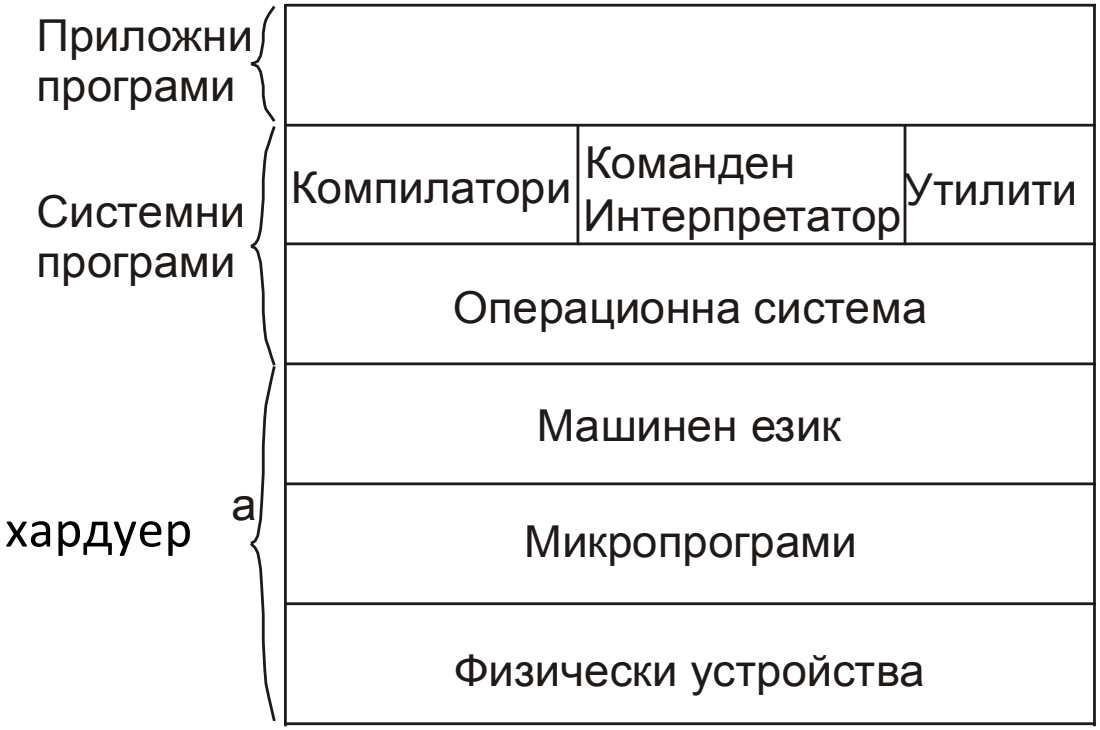

Приложни програми
Системни програми
хардуер
Компилатори
Команден Интерпретатор
Утилити
Операционна система
Машинен език
Микропрограми
Физически устройства

Рис.1.2.

СЪСТАВ И ФУНКЦИИ НА ПРОГРАМНАТА СИСТЕМА

I. СИСТЕМЕН СОФТУЕР

II. ПРИЛОЖЕН СОФТУЕР

Системен софтуер:

1. **Операционна система**. ОС изпълнява две основни, не особено взаимосвързани функции [Tanenbaum] и в зависимост от разглежданата функция се определя и понятието ОС. ОС е тази част от ПС, която изпълнява **управляващи функции**. Тя е защитена от потребителя.

А1) ОС като разширена машина. ОС изгражда абстрактна (виртуална) машина над хардуера, като предоставя нови функции, реализирани програмно. ОС премахва неприятното и трудоемко задължение да програмираме на машинен език, да следим и управляваме входно/изходните операции на ниско ниво, включващи операции по преместване на главите на диска, форматиране, инициализиране, ре-инициализиране, ре-калибриране на контролера и много други операции по непосредствено управление на компютърната система на ниво архитектура: машинни инструкции, организация на паметта, вход/изход и структура на шината.

2) ОС като разпределител на ресурси.

В основата на тази концепция е схващането, че основна функция на ОС е да осигури съвместно ефективно използване на ресурсите на компютърната система от всички потребители независимо от начина на използване – последователно или паралелно. Под потребители ще разбираме не само програмисти, но и задачи, всеки обект, който желае да ползва ресурс. С други думи, ОС трябва да следи на кого какъв ресурс е предоставен, да удовлетворява заявките за ресурси, да решава конфликтни ситуации и прочие.

2. Система за програмиране.

Включва инструментални средства за разработка на програми. Главно това са системни програми, свързани с решаването на езикови проблеми. Тук влизат:

транслатори: главната им цел е преобразуването на алгоритъма на програмата от един език в друг. При това същността на алгоритъма се запазва, а се изменя формата на представянето

библиотеки: предварителна транслирани програми, които потребителят указва по време на транслация и които стават достъпни за програмата му по време на изпълнението ù. Пример за това са математически програми.

3. Утилити (обслужващи програми).

Назначението на тази част от системния софтуер е преобразуването на програми и данни при смяна на носителя (въвеждане от клавиатурата, печат от диска и прочие.), прекодиране на данни (изменение начина на представянето им). Тази функция често е крайна цел на използване на компютъра. Реализира се от **редактори**, извършващи редактиране на програми и други текстове и от **програми за форматиране** текст, които управляват разместването на текста по определен начин. Към тях принадлежат и **програмите за прекодиране**, които обслужват нуждите на приложни програми, използващи различни знакови таблици и за кодиране, и декодиране на секретна информация. Към утилитите принадлежат и т.нар. **програми за сортиране**, извършващи реорганизация на данните. В техните функции могат да влизат или не входно/изходни операции, но винаги се включва явно или неявно изменение на адресите на разположение на данните.

Операционната система е съвкупност от програми, които осигуряват връзката на потребителя с хардуера на компютърната система.

Цели на ОС:

1. Да направи КС удобна за използване;
2. Да използва ефективно компютърния хардуер.

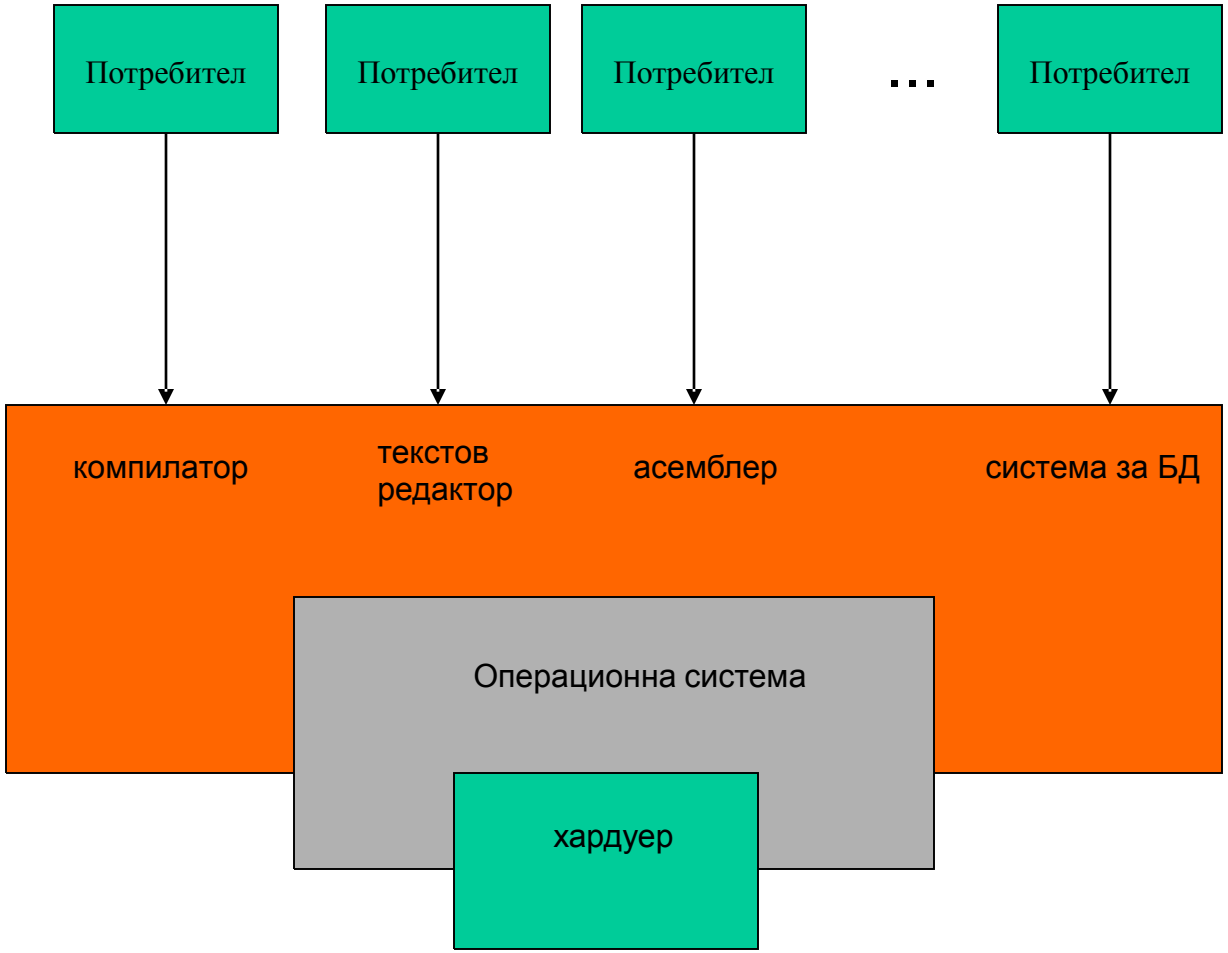
За разбиране същността на ОС е необходимо да разгледаме еволюцията на компонентите на ОС. Тази еволюция е резултат на естествени решения на проблемите в ранните ОС. Разбирането на причините за такова развитие дава възможност да преценим за какво служи операционната система и как изпълнява това.

Определение за операционна система
(истината е, че няма универсално определение за ОС)

ОС е важна част от почти всяка КС. Една КС може (грубо) да се определи като съставена от 4 части:

- хардуер – процесори, памети, вх/изх устройства – основните ресурси на КС;
- операционна система;
- приложен софтуер;
- потребители.

Може да има много и различни потребители, опитващи се да решат различни задачи, следователно, може да има множество различни приложни програми. Операционната система е тази част от системния софтуер, която **контролира и координира използването на хардуера** от страна на софтуера на различните потребители.



КС като съвкупност от четири компонента

Функции на операционната система

Друг гледна точка може да представи КС като съвкупност от хардуер, софтуер и данни. Операционната система предоставя средства за правилното използване на тези ресурси при функционирането на КС, т.е. предоставя среда, в която други програми могат ефективно да вършат полезна работа. Това означава че:

1) А: Операционната система е разпределител на ресурси. ОС трябва да реши как да разпределя ресурсите на КС между множество потребители (да предоставя и отнема), да решава конфликтни ситуации;

   В: Операционната система има управляваща функция: управлява различните устройства в КС.

2) Предоставя удобен (приятелски ☺) потребителски интерфейс.

Други възможни определения за ОС

Операционните системи съществуват, защото са разумен начин за създаване на използуема компютърна система. Фундаменталната цел на ОС е изпълняване на потребителските програми и улесняване решаването на задачите на потребителите.

Едно възможно определение за ОС: всичко, което производителят е решил да достави като ОС;

Друго: този системен софтуер, който е защитен от потребителя (не може да бъде пренаписан, модифициран от потребителя).

Трето: тази програма, която се изпълнява на компютъра постоянно (често се нарича ядро), заедно с приложенията.

Всъщност е важно да се определи кои софтуерни компоненти са части на операционната система.

По-лесно е да се определи операционната система чрез това, което изпълнява, отколкото чрез това, което представлява.

Двете главни цели на ОС – удобство и ефективност, са донякъде противоречиви. В миналото ефективността е била по-важна от удобството, затова голяма част от теорията на ОС се занимава с оптималното използване на компютърните ресурси.

Операционните системи и компютърните архитектури са в много тясна взимовръзка. ОС са разработени именно с цел улесняване използването на хардуера. Решаването на определени проблеми на ОС води до промяна в компютърната архитектура и обратно.

Класификация на ОС (по хронология)

1. Пакетни (batch) системи;
2. Системи с времеделене (time-sharing);
3. ОС за персонални компютри;
4. ОС за паралелни компютърни системи;
5. Системи в реално време (real-time systems);
6. Мрежови и разпределени системи.

1. Пакетни (batch) системи

Характерни за ранните КС. Потребителят не взаимодейства с КС, а подготвя т.нар. задание (job), което се състои от програма, данни и управляваща информация. Обикновено е във вид на тесте перфокарти, което потребителят предава на оператор и след определено време (минути, часове, дни ☺), се появява и резултата.

ОС за тези КС са елементарни. Главната им задача е да предават управлението от едно задание на друго. С цел ускоряване на работата операторите групират заданията в пакети по показател приблизително еднакви характеристики и потребности.

ОС е винаги в паметта. Едно задание не се стартира, докато не завърши предишното. Това създава проблем с ефективното използване на ресурсите: докато се извършва вх/изх, процесорът не работи. Процесорите са на един, два и повече порядъка по-бързи от периферията.

1. Пакетни (batch) системи

1 решение: кои от тях да бъдат “заредени” в паметта – планиране от високо ниво (job scheduling).

Наличието на няколко програми в паметта налага създаване на някакви правила за управление на паметта, т.е. ОС взема

2 решение: как да разпределя паметта между тези програми и да осигури защита на паметта.

Обикновено процесорните устройства са по-малко от заданията. Операционната система трябва да вземе

3 решение: на коя от програмите, намиращи се вече в паметта, да предостави процесора – планиране от ниско ниво (CPU scheduling)

2. Системи с времеделене (съвместно използване във времето) – Time Sharing Systems

Многопрограмните пакетни системи предоставят среда за ефективно използване на компютърните ресурси, но не дават възможност на потребителя да взаимодейства с КС.

Интерактивна (hands-on) компютърна система предоставя възможност за директна комуникация на потребителя със системата. Потребителят дава инструкции на ОС от входно устройство и очаква междинни резултати (или краен) на изходно устройство. Времето за реакция на системата е кратко.

CPU превключва между множество програми, но с такава честота, че потребителят може да вазимодейства с всяка от тях за времето на изпълнението и, а впечатлението е, че всяка програма разполага монополно с компютърните ресурси.

2. Системи с времеделене (съвместно използване във времето) – Time Sharing Systems

TSS са по-сложни от мултипрограмните ОС. Основни характеристики:

1. Множество програми в паметта налагат създаване на механизми за управление и защита на паметта;
2. Суопинг – преместване на програми от паметта на диска и обратно, с цел постигане на по-добро време на реакция от системата;
3. Виртуална памет – възможност за изпълняване на програми, които не са изцяло в паметта;
4. Дискова система: следствие на 2 и 3, и механизми за управлението на дисковата памет;
5. Механизми за паралелно управление на процеси: комуникация и синхронизация между процесите;
6. Евентуално, механизми за диагностициране и предотвратяване на deadlock (безкрайно очакване).

Важно!

Многопрограмност и многозадачност са различни понятия!

Multiprogramming <> multitasking!

Мултипрограмирането се отнася до пакетните системи; многозадачността – до системите с времеделене.

Процес: изпълняваща се програма заедно с всички предоставени за изпълнението и ресурси.

3. ОС за персонални компютри

Началото на 70 г. на миналия век.

През първите 10 г. На съществуването си ОС за тези компютри не разполагат със средства за защита на ОС от потребителските програми. Тогава не са нито многопотребителски, нито многозадачни.

Целите на тези ОС и досега остават максимално удобство и реакция на системата, вместо максимална ефективност на процесора и периферията.

ОС за микрокомпютри са наследили много добри характеристики от ОС за големи и средни компютри. Но тъй като утилизацията на процесора вече не е толкова важна, някои от решенията в ОС се оказват ненужни усложнения и не са подходящи за малки компютри.

4. Паралелни системи

Голяма част от КС имат повече от един процесор. Ако тези процесори си поделят системната шина, памет (не винаги), таймер и други устройства, такава система се нарича силносвързана (tightly coupled).

Причини за създаването на такива системи:

- Увеличаване на производителността;
- Икономия на средства;
- Повишаване на надеждността.

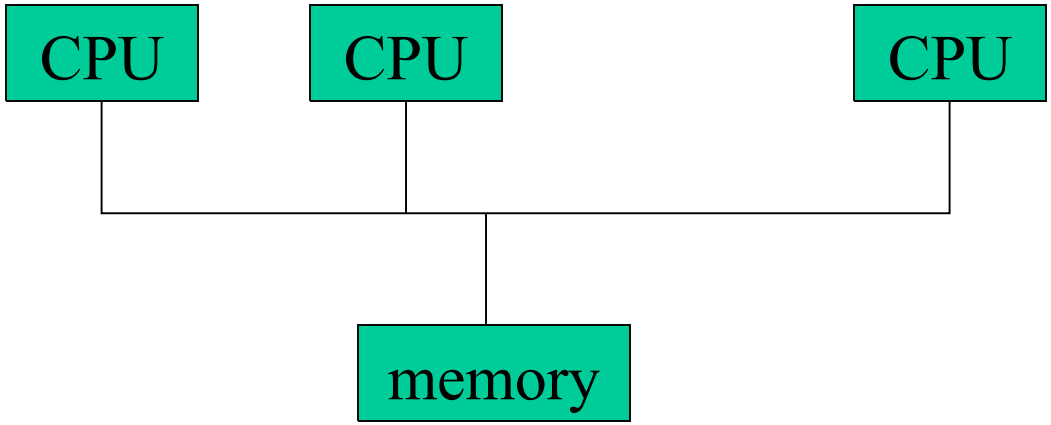
4. Паралелни системи

Симетрично мултипроцесиране (SMP): това е най-често разпространената схема на работа в мултипроцесорните системи. Всеки процесор изпълнява идентично копие на ОС. Между тези копия при разработването на системата са заложени необходимите средства за комуникация.

При асиметричното мултипроцесиране (AMP) всеки процесор има определена задача: главният управлява цялата система, останалите очакват инструкции от него или изпълняват предефинирани задачи. Тази схема определя отношението “master – slave”. Водещият процесор планира и определя работата за останалите процесори.

4. Паралелни системи

Ако в схемата SMP, при която всички процесори имат равни права и ако те са независими, е възможно достигане на неравномерно натоварване на процесорните компоненти. За решаване на този проблем процесите динамично се разпределят между процесорите, а и ресурсите също се разпределят динамично (динамично споделяне на ресурси).



5. Системи в реално време (Real-time systems)

Това е пример за специализирани ОС. Използват се когато се налагат времеви ограничения върху изпълнението на операциите или скоростта на потока от данни. Често се използват като управляващи устройства в системи за

- научни експерименти;
- производствени процеси;
- медицински системи за изображения;
- в автомобилни инжекционни системи;
- оръжейни системи;
- контролери на домакински уреди
- ....

Една RTS работи правилно само ако дава верния отговор за произволно времево ограничение.

5. Системи в реално време (Real-time systems)

Определят се два класа RTS:

1. Hard RTS : със строго рестриктивни изисквания за завършване на операциите. При тях вторична памет липсва или е съвсем ограничена. Данните и програмите се намират в бърза RAM или даже в ROM – поради енергонезависимост. Много от развитите характеристики на съвременните ОС липсват, тъй като те отделят потребителя от хардуера, а това отдалечаване резултира в неопределеност на времето, необходимо за завършване на операциите. Например, HRTS никога не се правят с виртуална памет. HRTS са в конфликт с операциите, които се очакват например от една TSS, затова двата класа не могат да се комбинират. Нито една от съществуващите универсални ОС не посреща изискванията на тези системи. Това са тясноспециализирани ОС.

5. Системи в реално време (Real-time systems)

2. Soft RTS. Не всички задачи е необходимо да бъдат решени в критичен интервал от време. Тези, които го изискват, са с по-висок приоритет от останалите. Както и при HRTS, забавяне на обслужването от страна на ядрото е в ограничен времеви интервал.

SRTS имат такива цели, които позволяват комбинирането им с други класове ОС. Използването им обаче е по-ограничено – в роботиката и за управление на производствени процеси е твърде рисковано. Сфери на приложение:

- мултимедия;
- виртуална реалност;
- научни изследвания;
- ....

6. Разпределени системи

Една тенденция в съвременните системи е разпределянето на изчисленията между множество процесори. За разлика от мултипроцесорните системи тук процесорите не споделят памет и таймер. Всеки процесор работи със собствена памет и общува с другите посредством различни комуникационни устройства. Те са известни като слабосвързани (loosely coupled), или разпределени системи.

Мрежовите ОС осигуряват мрежова връзка между компютрите в системата, дават възможност за споделяне на файлове и предоставят правила за комуникация, позволяващи различните процеси, изпълняващи се на различните компютри, да обменят съобщения.

6. Разпределени системи

Компютър, изпълняващ мрежова ОС, работи автономно и независимо от останалите в мрежата, но “знае” за нейното съществуване и може да общува с останалите машини в нея. Разпределената ОС, за разлика от мрежовата, е по-малко автономна среда. Различните ОС в разпределена система имат много по-тясна схема на общуване, което създава представата за единствена ОС, управляваща разпределената система.

Въпроси и задачи

1. Кои са главните цели и функции на операционната система?
2. Кои са главните предимства на мултипрограмирането?
3. Определете основните характеристики за следните класове ОС:

- пакетни;
- интерактивни;
- с времеделене;
- в реално време;
- мрежови;
- разпределени.

4. Опишете разликите между симетричното и асиметрично мултипроцесиране.
5. Какви са предимствата и недостатъците на мултипроцесорните системи?

Лекция 2. Структури на ОС. Структури в КС.

2.1. Структури на ОС.

ОС предоставя среда за изпълнение на потребителските програми. Операционните системи могат много да се различават по своите цели и възможности. Затова при разработването на нова ОС е необходимо целите и да са ясно дефинирани пред проектантите. Типът на желаемата система е базата, според която се избира между различните алгоритми и стратегии.

ОС могат да се определят от различни гледни точки:

- чрез дисасемблиране на системата на компоненти и определяне на взаимовръзките между тях;
- чрез услугите, които предоставят;
- чрез интерфейса;

Тези три гледни точки определят: разработващите ОС, програмистите и потребителите .

2.1.1. Компоненти на системата

Една ОС, която е обемен и сложен системен софтуер, може да бъде разработена само чрез дисасемблирането ѝ на по-малки компоненти. Всеки компонент трябва да представлява ясно определена част от системата, с внимателно дефиниран вход, изход и функции.

Не всички системи имат еднаква структура, но много от съвременните ОС поддържат разгледаните в тази лекция компоненти.

2.1.1.1. Управление на процесите

Една програма става активен обект само ако инструкциите и се изпълняват от процесора. Този активен обект се нарича процес.

Накратко, една програма, намираща се в главната памет и имаща всички необходими за изпълнението си ресурси, е **процес**.

Изпълнението на процес е последователно: процесорът изпълнява инструкция след инструкция до завършване.

Процесът е единица за работа в системата. В системата има съвкупност от процеси, системни и потребителски. Всички те могат да се изпълняват паралелно, с мултиплексиране на процесора между тях.

ОС трябва да предостави следните възможности (дейности) за управление на процесите:

- Създаване и унищожаване на потребителски и системни процеси;
- Спиране и възобновяване на процеси;
- Предостявяне на механизми за синхронизация;
- Предоставяне на механизми за комуникация;
- Предоставяне на механизми за управление на безкрайното очакване (deadlock)

2.1.1.2. Управление на паметта

Главната памет е масив от думи или байтове, всяка (всеки) с уникален адрес. За да се изпълни една програма, тя трябва да бъде “проектирана” върху абсолютните адреси на главната памет. При прекъсване на изпълнението тези адреси могат да бъдат освободени, а кодът и данните – копирани на диска. При многозадачност ОС трябва да управлява множество програми в паметта. Това става посредством различни механизми за управление на паметта, а ефективността на различните алгоритми зависи от конкретния случай. Изборът на такъв механизъм особено силно е зависим от хардуерната реализация на съответния компютър.

ОС трябва да изпълнява следните дейности по управление на паметта:

- Да поддържа информация кои части от паметта са предоставени за използване и на кои процеси;
- Да решава кои процеси да бъдат “заредени” в паметта при освобождаване на памет;
- Да предоставя и отнема памет при необходимост.

2.1.1.3. Управление на файловете

Файловата подсистема е най-видимият компонент на ОС. Компютрите могат да съхраняват информация на най-различни видове физически носители и всеки от тях има собствени характеристики и начини на организация на информацията. Всяка среда се контролира от устройство, което също има собствени характеристики. Те включват скорост (време) за достъп, обем, скорост на предаване на данни, метод за достъп (пряк или последователен).

За удобство ОС предоставя унифициран подход към съхраняваната информация. Той предвижда абстрахиране от физическите особености на носителите и устройствата и определя логическата единица **файл**.

Файлът е именувана съвкупност от логически обвързана информация, намираща се на външен носител. ОС реализира абстрактната концепция за файл като управлява средите за съхранение, устройствата и контролерите.

За улесняване на работатата с файлове, те се организират в някаква структура, най-често дърво. Освен това, когато множество потребители имат достъп до файловете, е необходимо да се контролира вида на достъпа – за четене, модифициране, изпълнение и пр.

ОС трябва да изпълнява следните дейности за управление на файловете:

- Създаване и унищожаване на файлове;
- Създаване и унищожаване на директории;
- Поддържане на функции за манипулиране с файлове и директории;
- Разполагане на файловете във вторичната памет;
- Съхраняване на файловете в стабилна (енергонезависима) среда.

2.1.1.4. Управление на вторичната памет

Главната цел на компютърната система е да изпълнява програми. За изпълнението те, заедно с данните, трябва да са в **главната памет**. Тъй като тя е с недостатъчен обем за всички програми и данни, и е енергозависима, КС използват **вторична памет** за съхраняване на програмите и данните.

Повечето съвременни КС използват дискове в качеството на голяма по обем среда за онлайн съхраняване на програми и данни. Повечето системен софтуер също се намира на диска, докато не бъде стартиран. Тогава диска може да се разглежда и като източник, и като приемник на резултата при изпълнението на програмите.

Затова правилното управление на дисковата памет също е от централно значение за компютърната система.

ОС трябва да изпълнява следните дейности за управление на вторичната памет:

- Да управлява свободното дисково пространство;
- Да предоставя памет на диска;
- Да планира диска.

Тъй като вторичната памет се използва много често, общата скорост на работата на компютърната система силно зависи от скоростта на дисковата подсистема и алгоритмите, които са избрани за управлението и.

2.1.1.5. Управление на входа/изхода

Една от целите на ОС е да скрие от потребителя особеностите на хардуера, и съответно на устройствата, от потребителя.

ОС трябва да изпълнява следните дейности за управление на устройствата:

- Да изпълнява буфериране, кеширане и спулинг (компонент, управляващ памет);
- Да предостави универсален интерфейс с драйверите на устройствата;
- Да предостави драйвери за всички конкретни устройства. Само драйверите на устройства “знаят” специфичните особености на устройствата, които обслужват.

2.1.1.6. Мрежова среда

Разпределената система е съвкупност от процесори, които не споделят обща памет, устройства или таймер. Процесорите могат много да се различават по размер и функции – от малки микропроцесори, през работни станции, миникомпютри, до големи компютри.

Процесорите в системата се свързват чрез **комуникационна мрежа**, която може да бъде конфигурирана по различни начини. Може да бъде напълно или частично свързана.

Комуникационната мрежа трябва да обслужва изпращането на съобщения, с подходяща маршрутизация, да решава проблеми със състезания, сигурност и защита.

Разпределената система обединява физически различни, възможно хетерогенни системи, в една обща съгласувана система, предоставяйки на потребителя достъп до множество разнообразни ресурси, които тази система поддържа.

Достъпът до споделяните ресурси позволява да се увеличи скоростта на изчисленията, фунционалността, достъпа до данните и надеждността.

Операционните системи обикновено обобщават мрежовия достъп като достъп до файлове, като детайлите на този достъп са съсредоточени в драйверите на устройствата на мрежовия интерфейс.

Използването на WWW предизвиква създаването на нови методи за достъп до споделяна информация. Определен е протокола ***http*** за комуникация между web-сървър и web-браузър.

В една разпределена система е необходимо да се предоставят механизми за:

- синхронизация на процеси;
- за управление на междупроцесната комуникация;
- за решаване на проблема с безкрайното очакване;
- за манипулиране на различни сривове и грешки, които не са характерни за централизирана система.

2.1.1.7. Команден интерпретатор

Един от най-важните софтуерни компоненти за ОС е **командният интерпретатор (shell),** който е интерфейсът между потребителя и ОС. Някои ОС включват команден интерпретатор в ядрото си, при други (MS-DOS, UNIX), е специална програма, която работи при стартирането на задание или при регистриране на потребител, при TSS.

Инструкциите се задават на ОС посредством управляващи оператори. Функцията на командния интерпретатор е да избере следващия управляващ оператор и да го изпълни.

ОС понякога се диференцират според командния интерпретатор:

- ОС с графичен интерфейс;
- ОС с команден интерфейс.

2.1.1.8. Защита

Ако КС има множество потребители и позволява паралелно изпълнение на множество процеси, необходимо е процесите да бъдат защитени един от друг, т.е. да има механизми, осигуряващи използването на файлове, сегменти памет, процесор и други ресурси само от тези процеси, оторизирани от ОС.

(напр., регистрите на контролерите са недостъпни за потребителя)

Защита е всеки механизъм за контрол на достъпа на процеси или потребители до ресурсите на компютърната система.

Защитата трябва да:

- предоставя средства за спецификация на този контрол и средства за прилагането му;
- различава оторизирано и неоторизирано използване на ресурсите, и да го ограничава (забранява).

Защитата може да увеличи надеждността на системата чрез откриване на латентни грешки в интерфейса между компонентите ѝ.

1.2. Услуги на ОС

ОС предлага определени услуги на програмите и потребителите, като по този начин предоставя среда за изпълнението на програмите. Те правят програмирането по-лесно и удобно.

За различните ОС тези услуги се различават, но могат да бъдат обобщени в няколко групи:

2.1. *Изпълнение на програмите:* системата трябва да може да зареди програмата в главната памет и да я стартира, да завърши изпълнението и *по нормален или неправилен (с грешка),* начин.

2.2. *Вх/изх операции:* изпълнявяща се програма може да инициира вх/изх, с необходимост от файл или устройство. За различните устройства са необходими различни, специфични действия за вх/изх. Поради съображения за ефективност и защита, потребителите не могат да управляват вх/изх, затова ОС трябва да осигури средства за изпълнение на вх/изх.

System calls (системни примитиви)

System calls осигуряват интерфейса между процесите и ОС. Обикновено са достъпни като асемблер-инструкции и са изброени в наръчниците за асемблер-програмистите.

Някои езици от високо ниво, като C, C++, Perl, са създадени с цел да изместят асемблера от системното програмиране. Те позволяват директното използване на системни примитиви.

Java не позволява директно използване на системни примитиви, тъй като едно такова извикване е специфично за дадената ОС и резултира в разбираем само за съответната платформа код. Но, ако Java приложение се нуждае от специфично системно обслужване, приложението може да извика метод, написан на друг език – C или C++. Такива методи са известни като “native”.

Системните примитиви могат да бъдат групирани в следните пет главни категории:

3.1. *Управление на процеси:*

- end, abort;
- load, execute;
- create process, terminate process;
- get process attributes, set process attributes;
- wait for time;
- wait event, signal event;
- allocate free memory;

3.2. *Управление на vtorichnata памет:*

- create file, delete file;
- open, close;
- read, write, reposition;
- get file attributes, set file attributes.

3.3. Управление на устройствата

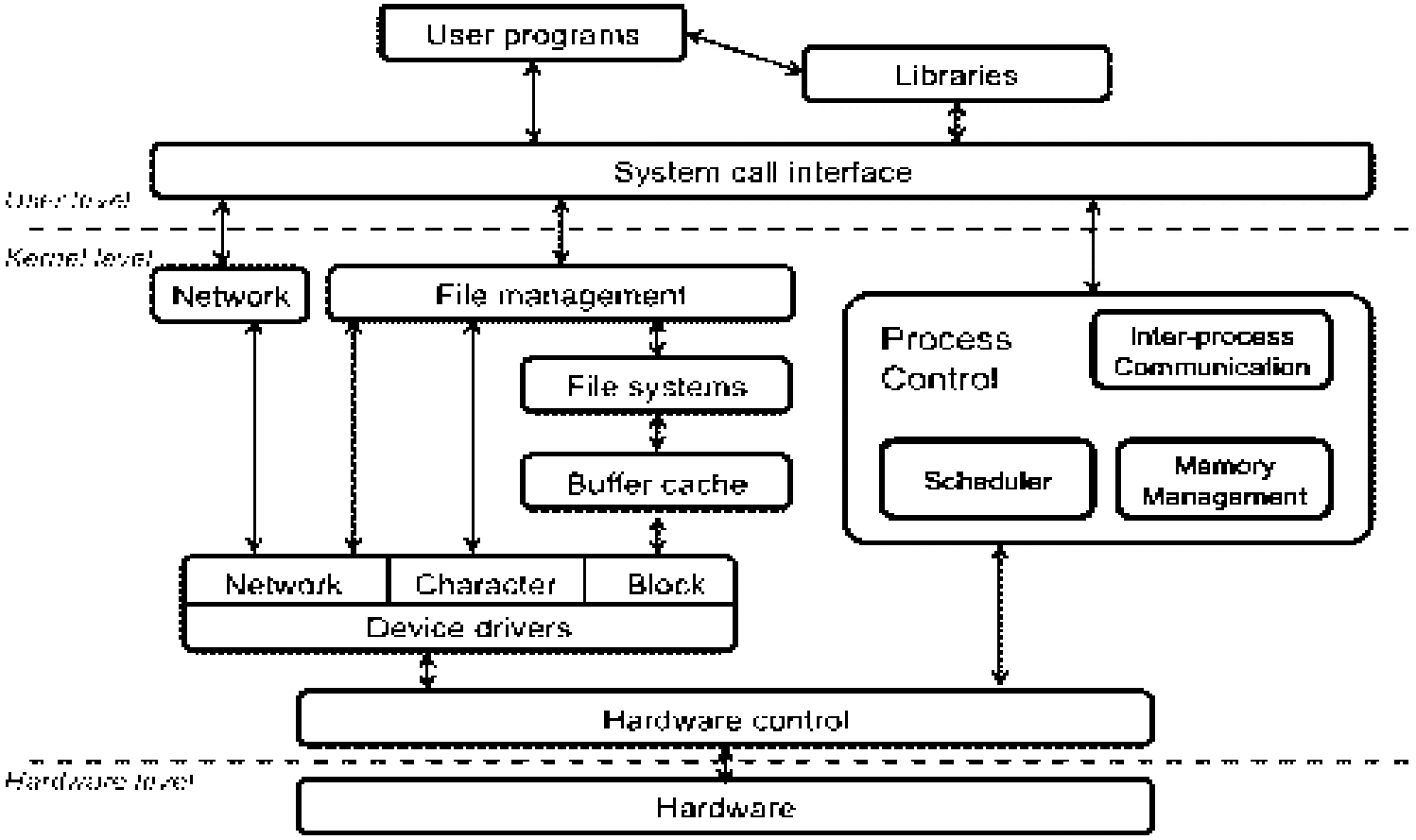
- request device, release devise;
- read, write, reposition;
- get device attributes, set device attributes;
- logically attach or detach devices;

3.4. *Поддържане на информация*

- get time or date, set time or date;
- get system data, set system data;
- get process, file, or service attributes;
- set process, file, or device attributes;

3.5. *Комуникации*

- create, delete communication connection;
- send, receive messages;
- transfer status information;
- attach or detach remote devices.



Структура на типична ОС

Въпроси и задачи към част 1 „Структури на ОС“

1. Кои са петте главни дейности на ОС по отношение управлението на процесите?

2. Какво представлява командният интерпретатор?

3. Кои са основните компоненти на универсална ОС?

4. Какво е файл?

5. Какви са групите услуги, които предоставя ОС?

Лекция 3 . Структури в компютърните системи.

1. Операции в КС

Съвременните универсални компютърни системи се състоят от един или повече процесори и няколко контролера, свързани с обща шина, осигуряваща достъп до споделяна памет. Всеки контролер обслужва специфичен тип устройства (дискове, видео дисплеи, аудио устройства...). Процесорът и контролерите могат да работят паралелно, конкурирайки се за достъп до паметта. Контролерът на паметта синхронизира достъпа до паметта от страна на устройствата и процесора.

За стартиране на компютъра – при включване на захранването или при презареждане е необходимо наличието на начална програма. Тя се нарича “начална зареждаща програма” - bootstraping program. Тя инициализира (дава начални значения) на всички компоненти на системата; локализира мястото на ядрото на ОС и го зарежда в паметта.

След това ОС стартира първия процес (напр. init) и очаква настъпването на някакво събитие. Настъпването на събитие се обозначава чрез **прекъсване** от хардуера или от софтуера. Хардуерът изпраща сигнал за прекъсване по системната шина, който постъпва по определен вход в процесора. Софтуерът генерира прекъсване посредством специални операции, наречени “системни директиви”, “системни извиквания” : **system calls.**

Прекъсване се генерира по различни причини: завършване на вх/изх операция, опит за делене на нула, невалиден достъп до паметта, при заявка от процес към ОС за изпълнение на обслужване. За всяко прекъсване съществува обслужваща (сервисна) процедура - **interrupt handler**.След приключване на такава процедура процесорът възобновява прекъснатите изчисления.

Прекъсванията са важна част от КС. Всеки модел компютър има собствен механизъм на прекъсванията, но принципите на тези механизми са общи.

Механизмът на прекъсването предава управлението на сервисната процедура.

Броят и видът на прекъсванията в една КС са предварително определени.

Началните адреси на сервисните процедури се намират в таблица, наречена **вектор на прекъсванията.** Обикновено тази таблица се съхранява в ниските адреси на паметта.

Векторът на прекъсванията се адресира от уникално число, което постъпва със самия сигнал за прекъсване от устройството. Чрез достигнатия указател се извлича адреса на първата инструкция от обработващата прекъсването процедура. При това е необходимо да се съхрани адреса на инструкцията, с която трябва да продължи изпълнението на прекъснатата програма.

След приключване на обслужването на прекъсването (изчистване на прекъсването), адреса на тази инструкция се извлича в програмния брояч и изпълнението на програмата продължава.

Съвременните ОС се “движат” от прекъсванията.

Настъпващите в една КС събития почти винаги се отбелязват с прекъсване или **trap-инструкция** (изключение). Това е софтуерно-генерирано прекъсване, предизвикано от грешка или от заявка от страна на потребителски процес за специфично обслужване от ОС.

Тази така определена от прекъсванията същност на ОС определя структурата ѝ като цяло. За всяко специфично прекъсване съществува отделен сегмент от код, който определя предприеманите в случая действия.

2. Структура на входа/изхода

В една КС има множество устройства. Всеки контролер обслужва определен тип устройства, като може да е свързан с повече от едно устройство.

Контролерът поддържа неголяма локална буферна памет и съвкупност от регистри със специално предназначение.

Контролерът управлява прехвърлянето на данни между устройството, което управлява, и локалната буферна памет.

Размерът на буфера може да е различен за различните контролери в зависимост от спецификата на устройствата.

(например размерът на буфера за дисков контролер е същият или кратен на размера на най-малката адресуема порция на диска (сектор), която обикновено е 512 байта.)

Буфер: област от памет, в която се намира единствен екземпляр от данни.

2.1. Входно/изходни прекъсвания

За стартиране на вх/изх операция CPU записва определени стойности в регистрите на съответния контролер. Контролерът проверява регистрите, определя какви операции трябва да изпълни. Ако, например, определи заявка за вход, стартира трансфер на данни към локалния си буфер. При приключванена трансфера контролерът съобщава за това на процесора чрез прекъсване.

След стартиране на вх/изхода са възможни два варианта:

- В по-простия случай, след края на трансфера на данни, управлението се връща към потребителския процес – **синхронен вход/изход**

- Във втория случай управлението се връща към потребителския процес без да се чака края на трансфера. Нарича се **асинхронен вход/изход.** Трансферът на данни се осъществява паралелно с други системни операции.

Главното предимство на асинхронния вх/изх е неговата ефективност, т.е. докато се изпълнява въвеждане или извеждане на данни, процесорът, вместо да стои, може да процесира или да стартира други вх/изх операции. Тъй като вх/изх може да бъде много по-бавен от процесора, по този начин работата в системата значително се ускорява.

Възможно е инициативата за вх/изх операция да не е от изпълняващ се процес, а от устройства: много интерактивни системи позволяват на потребителите да въвеждат данни от клавиатурата, преди още тези данни да са станали необходими на активен процес. Тогава се генерира прекъсване, сигнализиращо за постъпване на символи от клавиатурата, а в същото време ОС знае, че не е постъпвала заявка от изпълняваща се програма за вх/изх операция. Набраните символи се съхраняват в буфер, докато някой процес не си ги “потърси”.

2.2. Структура на прекия достъп до паметта (DMA)

При DMA (direct memory access) контролерът на устройство изпраща или получава цял блок от данни между локалния си буфер и паметта **без участието на процесора.** За един блок данни се генерира само едно прекъсване, вместо по едно за всеки байт или дума.

Докато DMA-контролерът изпълнява трансфер на данни, CPU е свободно за други операции. За това време обаче шината на паметта е заета и недостъпна за процесора.

3. Структура на паметта

Компютърните програми трябва да бъдат в главната памет, наречена RAM (Random Access Memory), за да бъдат изпълнени. Оперативната памет е единственото голямо пространство за съхранение, до което процесорът има директен достъп. Тя е изпълнена чрез полупроводникова технология наречена DRAM (Dynamic Random-Access Memory), която формира поредица от машинни думи, всяка от които има собствен адрес. Взаимодействието се постига, чрез поредица от **load** или **store-**инструкции за определени адреси. Една LOAD-инструкция премества дума от паметта в регистър на процесора, а STORE инструкция премества съдържанието от регистър в паметта.

Отделно процесорът избира инструкции за изпълнение от паметта.

Стандартно изпълнението на 1 инструкция протича по следния начин – системата изтегля инструкцията от паметта, запаметява я в регистър на инструкцията (instruction register), декодира я, изтегля операнди от паметта и ги запаметява в регистри за операндите, изпълнява инструкцията и записва разултата, но не задължително в паметта .

Паметта разпознава единствено потока от адреси. Тя не знае как се генерират (индексиране, адреси на символи и т.н.) или за какво са (инструкции или данни). Следователно няма нужда да знаем как програмата генерира адреси, а само да се интересуваме от последователността на генерирането.

Теоретично, искаме програмите и данните да се запазват в оперативната памет постоянно, но това не е възможно заради следните 2 причини:

1)Оперативната памет е малка, за да съхрани всичко за постоянно.

2) Оперативната памет е енергозависима..

Затова повечето КС предоставят **вторична памет** като разширение на главната. Главното изискване към вторичната памет е да може да съхранява голям обем данни постоянно.

Главната памет и регистрите, вградени в процесора, са единствената памет, към която процесорът има пряк достъп: ако разгледаме която и да е машинна инструкция от който и да е машинен език, ще видим, че в качеството на аргументи може да стоят адреси в паметта, но никога дискови адреси.

Към структурата на паметта, която описахме дотук, се включват още кеш–памет, дискови устройства, магнитни ленти и др. Разликата в различните видове памет идва от скоростта, цената, размера и устойчивостта.

Йерархия на паметта

Може да се определят четири главни нива на паметта:

- *Вътрешна* – регистри на процесора и кеш;
- *Главна* – системната RAM и контролерните карти;
- *On-line масова памет* – вторичната (secondary) памет;
- *Off-line bulk памет* – “третична” tertiary и off-line памет.

Такова структуриране в йерархията е най-общо.

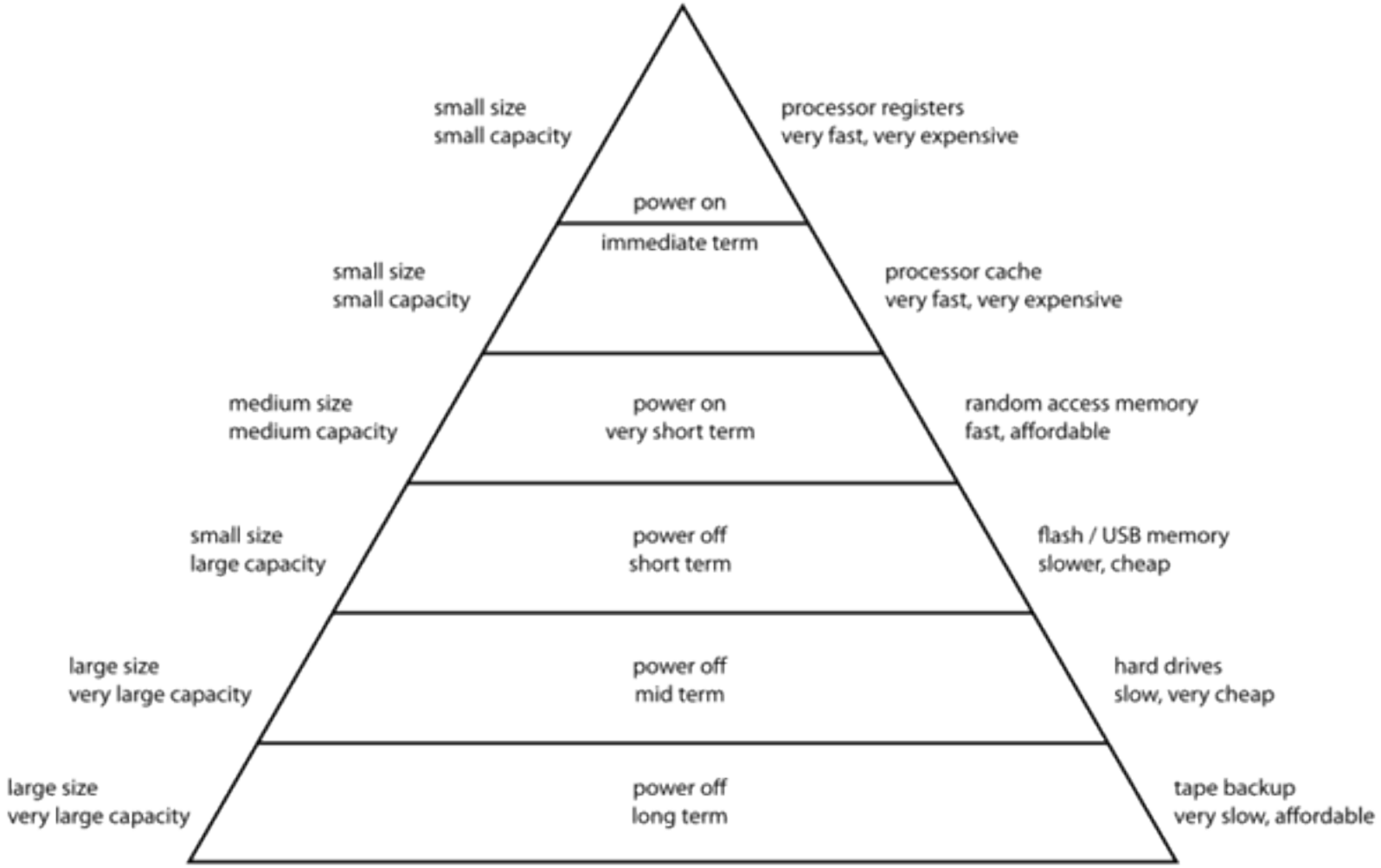
Йерархия на паметта

Разнообразието на паметта в една система може да бъде представено йерархично в зависимост от скоростта и цената. Скоростта намалява с нарастване на обема и обратното.

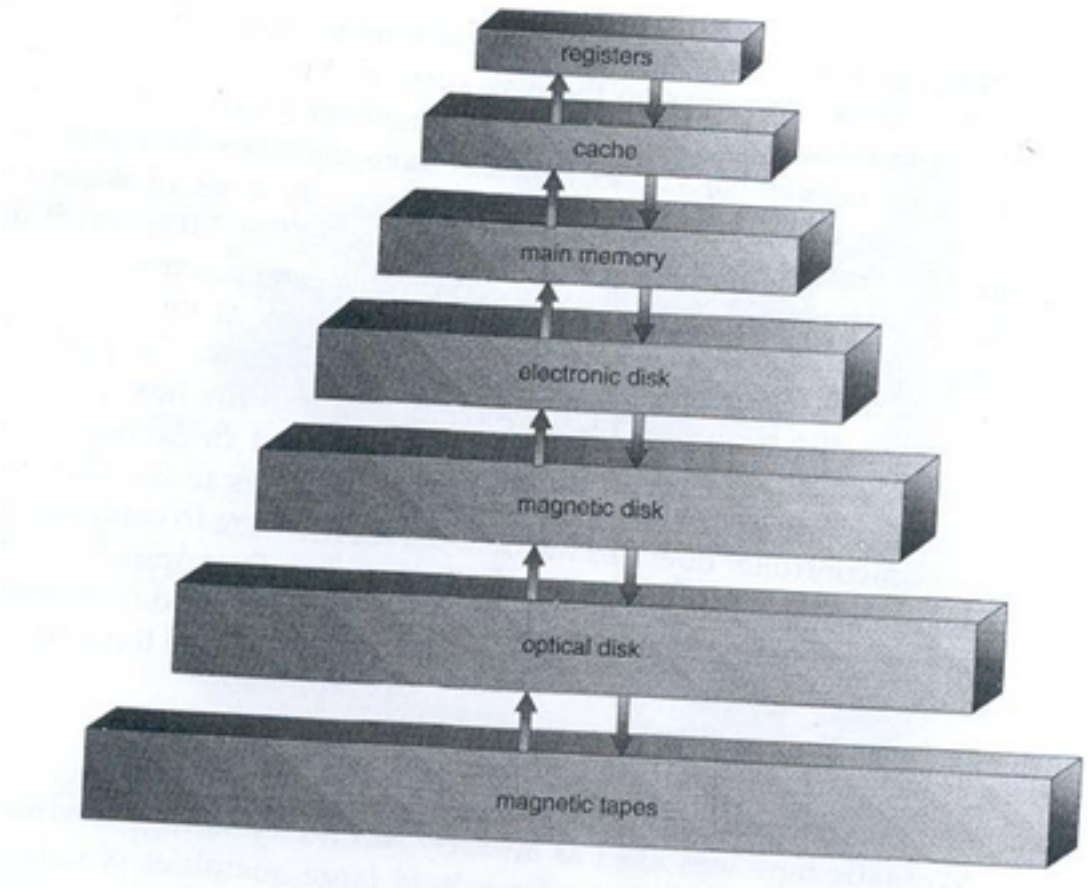
В допълнение паметта се дели на енергозависима – регистри, оперативна памет и кеш и енергонезависима – електронни, магнитни и оптични дискове, дискети и др.

Системата трябва да е балансирана в зависимост от 4 показателя – скорост, обем, енергозависимост и цена.

Computer Memory Hierarchy



Йерархия на паметта



Регистри

Позволяват директен достъп на процесора до тях. Затова, всички изпълняващи се инструкции и всички данни, които те използват трябва да се намират в тях. Ако не са там, те първо трябва да бъдат преместени преди работа на процесора с тях.

В случай на I/O операция, всеки I/O контролер включва регистри за задържане на предаваните инструкции и данни. Обикновено, специални I/O инструкции позволяват обмен на данни между тези регистри и паметта.

За да се осъществи по-удобен достъп до вх/изх устройства, много компютърни архитектури предоставят memory-mapped I/O.

Memory-mapped I/O

В този случай определен диапазон от адреси в паметта е резервиран. В този диапазон всеки адрес съответства на устройство, така че запис по този адрес означава извеждане на данни към устройството (“пишем” в устройството), а четене от този адрес означава въвеждане от устройството.

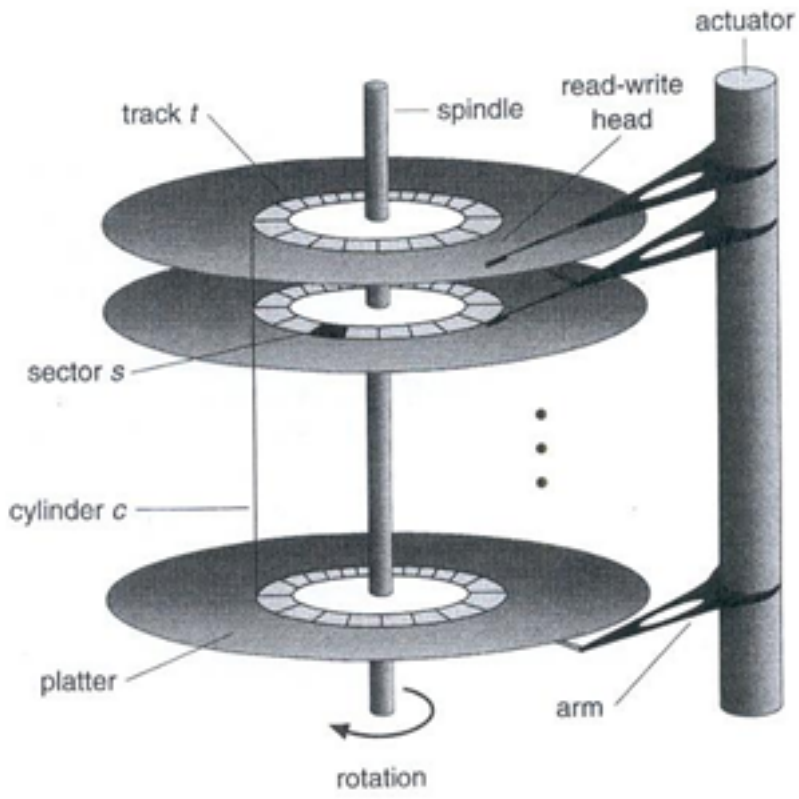
Този метод е подходящ за устройства, които работят бързо (например видео контролер).

Такъв I/O е подходящ и за други устройства – например серийни и паралелни портове за свързване на модем или принтер.

Достъпът до паметта може да отнеме няколко цикъла. В този случай процесорът обикновено трябва да спре, тъй като няма нужните данни за изпълнение на инструкцията.

Тази ситуация е нежелателна заради честия достъп до паметта. Решението е добавянето на бързодействаща памет между процесора и оперативната памет наречена **кеш–памет.**

Магнитни дискове



Магнитни дискове

Предоставят основната част от вторичната памет. Всяка плоча от диска има плоска и кръгла форма. Стандартния размер е от 1.8 до 5.25 инча.

Двете страни на плочите са покрити с магнитна материя, като записването става чрез намагнетизиране. Над всяка плоча има глава, която чете и записва.

Главите са закачени към носач, който ги задвижва. Повърхността на главата е разделена на логически пътеки, които са разделени на сектори.

Пътеките, които са на еднаква позиция спрямо носача формират цилиндър. Капацитета на дисковете се измерва в гигабайта.

Магнитни дискове

При използване на диска той се завърта с голяма скорост – над 100 завъртания в секунда. Скоростта на диска се определя от скоростта на трансфера между него и системата (MB/s) и от времето за позициониране на главата на мястото за четене/запис и завъртане на диска до достигане на даден сектор (милисекунди).

Тъй като главите преминават на микрони от плочите има опасност от контакт и въпреки, че плочите са покрити със защитен слой, понякога главата може да повреди повърхността. Ако това се случи, диска не може да се поправи и трябва да бъде заменен.

Магнитни дискове

Друг вид дискове са тези, които могат да се добавят и премахват в режим на работа на системата. Обикновено се състоят от една плоча. Пример са дискетите.Те са бавни, защото главата им стои върху повърхността и завъртането става бавно.

Дисковете се прикачват към компютъра чрез мрежа наречена I/O Bus – EIDE, SCSI. Трансфера на данни се извършва от контролери (електронни процесори). Хост-контролерът се свързва с този на диска, а за да се осъществи операция компютърът записва командата в своя контролер (обикновено се използват memory-mapped I/O портове) и след това я изпраща чрез съобщение на контролера на диска.

Кеш–памет

Използва се за по-бърз трансфер на информация. Когато е необходима дадена информация, първо се проверява дали тя се намира в кеша. Ако не е там, се използва оперативната памет, като се оставя копие в кеша предполагайки, че скоро ще потрябва отново.

Вътрешните регистри (индексните регистри например) предоставят високоскоростен кеш за оперативната памет. Програмистът изпълнява алгоритмите за регистровото разпределяне и възстановяване и решава коя информация да запази в регистрите и коя в оперативната памет.

Кеш–памет

Повечето системи имат кеш за инструкции, който задържа следващата инструкция. Без него процесорът ще трябва да чака няколко цикъла докато се вземе инструкцията от оперативната памет. Повечето системи имат по няколко кеша.

Движението на информация между нивата на паметта е раабота на различни нива на йерархия в КС – трансферът от кеша към процесора и регистрите е хардуерна задача, а трансферът от диска към паметта е работа на ОС.

Съгласуваност и последователност

(кохерентност на данните)

В йерархична структура едни и същи данни могат да се появят в различно ниво на паметта. Например, ако променливата А, разположена във файл Б, трябва да се увеличи с единица (файлът се намира на диска), за да се увеличи А, файлът (или част от него), се копира в оперативната памет и след това в кеша и вътрешните регистри. По този начин А се появява на няколко места и когато А се увеличи с единица във вътрешните регистри стойността е различна от тази в другите памети.

Стойността на А става еднаква след като новата и стойност се запише обратно на диска.

Съгласуваност и последователност

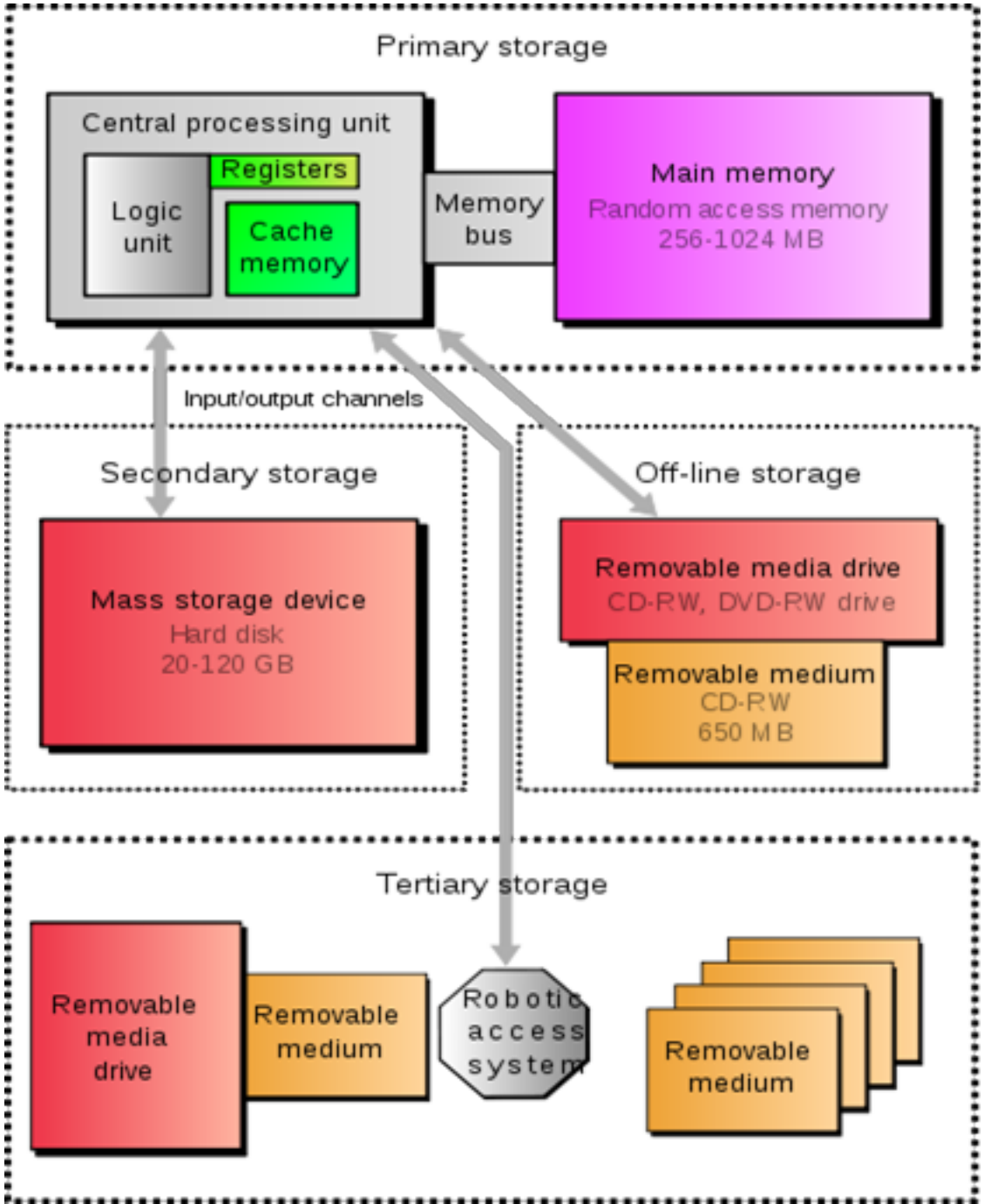
В ситуация на единичен процес за дадено време, това не създава проблеми тъй като достъпът до А е винаги на най - високото ниво. В многозадачна среда където процесорът извършва множество операции е важно той да работи със най-обновената стойност на А.

Ситуацията става още по-сложна при многопроцесорна среда, където в допълнение към поддържането на вътрешните регистри, процесорът съдържа също и локален кеш. В такава среда копието на А може да съществува едновременно в няколко кеша и след като отделните процесори могат да изпълняват едновременно, трябва да сме сигурни, че обновяване на А в единия кеш ще изключи възможността за обновяване в другите.

Съгласуваност и последователност

Гореописаната ситуация се нарича кохерентност на кеша и обикновено е хардуерен проблем.

Ситуацията става още по-сложна в среда с много компютри където има няколко копия (дубликати). Тъй като дубликатите могат да бъдат обновени едновременно, трябва да сме сигурни, че когато дубликата се обнови на едно място, ще се обнови и на другите възможно най-бързо.


Primary storage
Central processing unit
Logic unit
Registers
Cache memory
Memory bus
Main memory
Random access memory
256-1024 MB
Input/output channels
Secondary storage
Mass storage device
Hard disk
20-120 GB
Off-line storage
Removable media drive
CD-RW, DVD-RW drive
Removable medium
CD-RW
650 MB
Tertiary storage
Removable media drive
Removable medium
Robotic access system
Removable medium

Въпроси и задачи:

1. Каква е разликата между trap и хардуерно прекъсване?
2. Защо се използва кеш?
3. Каква е разликата между кеш и буфер?
4. Какви са предимствата на асинхронния вход/изход?
5. С кои видове памет може да работи централният процесор (процесори)?
6. Кой от долуизброените видове памет е енергозависим?

   а) оптичен диск;

   б) RAM;

   в) ROM;

   г) USB-памет:

Лекция 4. Управление на процесора

Процеси . Концепция за процес

Всички изпълними програми в една компютърна система, включително и ОС, могат да се представят като съвкупност от последователни процеси, или просто процеси. Процесът е изпълняваща се програма заедно със значенията на програмния брояч (PC), регистрите на процесора и значения на променливите на тази програма. Така както една програма не може да се изпълнява, ако не се намира в изпълнимата памет, така тя не може да се изпълнява и ако не й е достъпен процесорът. Процесорът се разглежда като ресурс, използван за изпълнението на програма. Процесът се разглежда като обект, на който може да бъде предоставен процесорът.

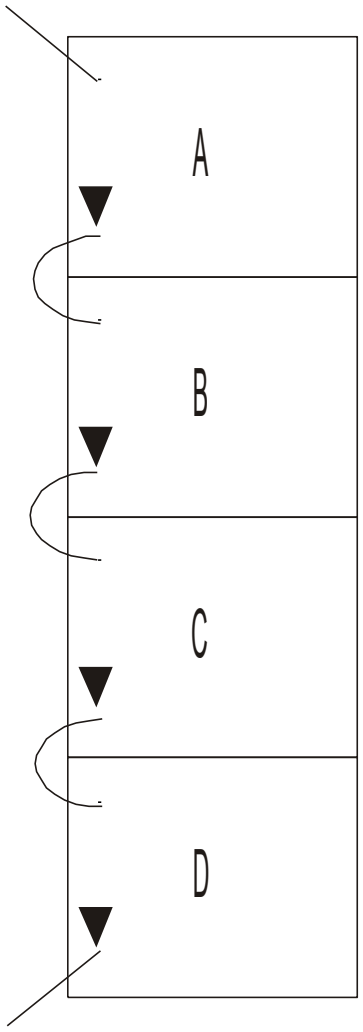
В съвременните системи процесът се разглежда като единица за работа.

Системата може да се разглежда като съвкупност от процеси: системни и потребителски.

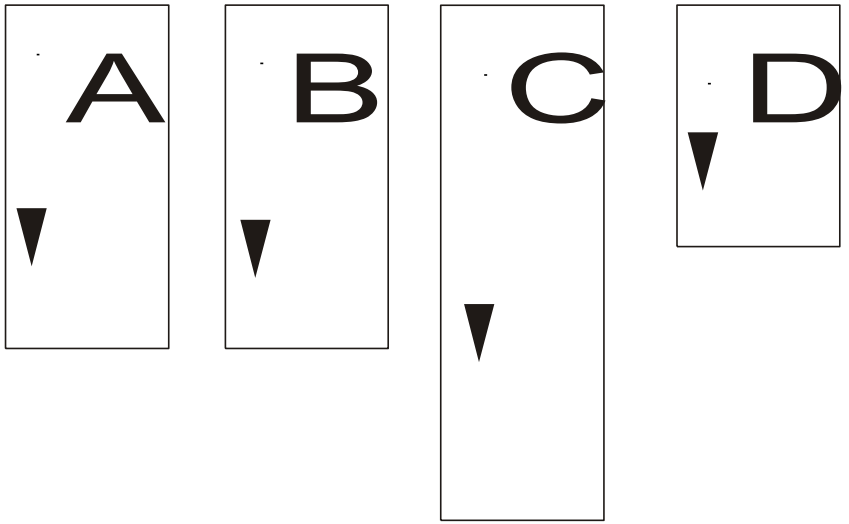
Всички процеси в системата могат да бъдат изпълнявани паралелно, посредством мултиплексиране на процесора между тях.

В многозадачните и мултипрограмни системи използването на един фактически процесор създава илюзията, че всеки процес използва процесор независимо от останалите, т.е. с всеки процес се свързва негов собствен виртуален процесор

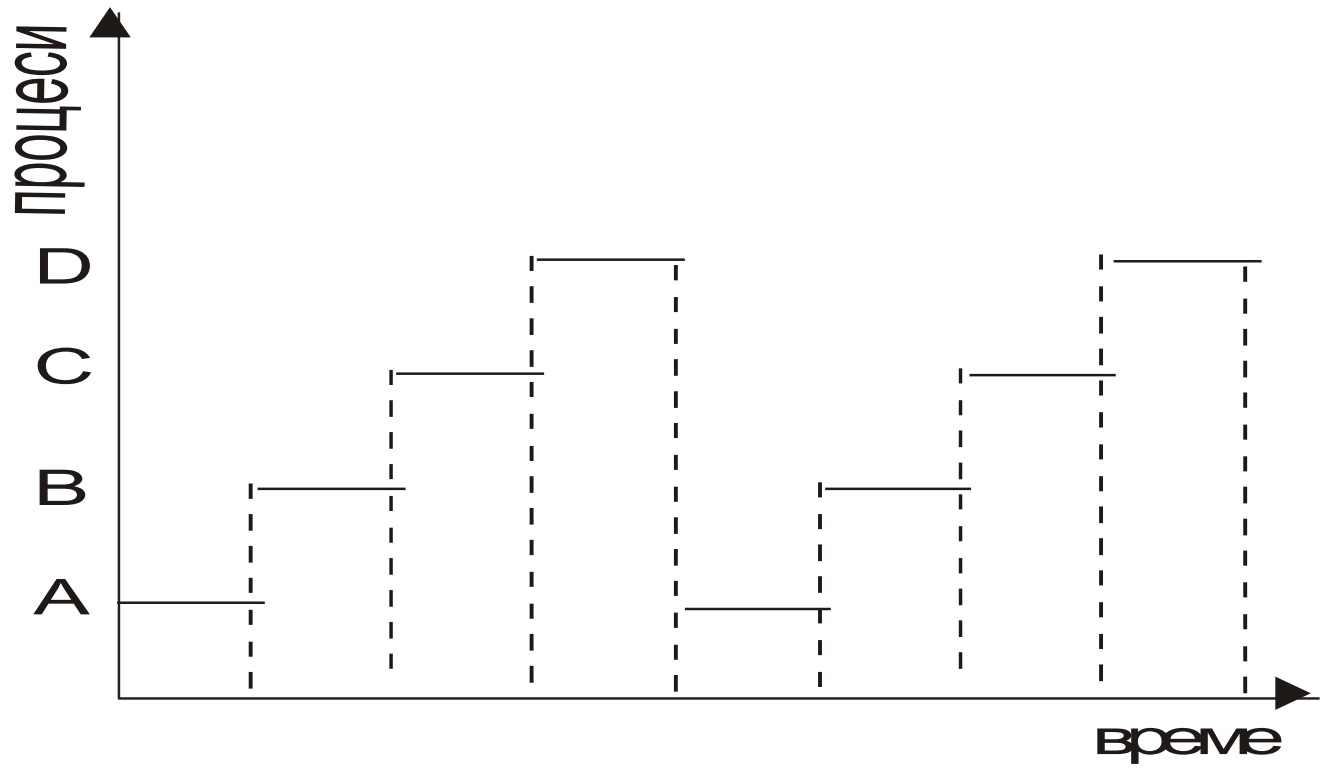
Както съвместното използване на паметта, така и съвместното използване на процесора се характеризира със снижаване скоростта на изпълнение на програмата. Главна характеристика за всеки процесор е неговата скорост (брой операции за единица време), а многозадачността (мултипрограмирането) снижава възможностите на виртуалните процесори, достъпни на потребителите: скоростта на всеки виртуален процесор е по-малка от скоростта на реалния процесор, чрез който е реализиран.



А) Един програмен брояч



Б) Четири програмни брояча



Многозадачност с четири процеса

Състояние на процесите.

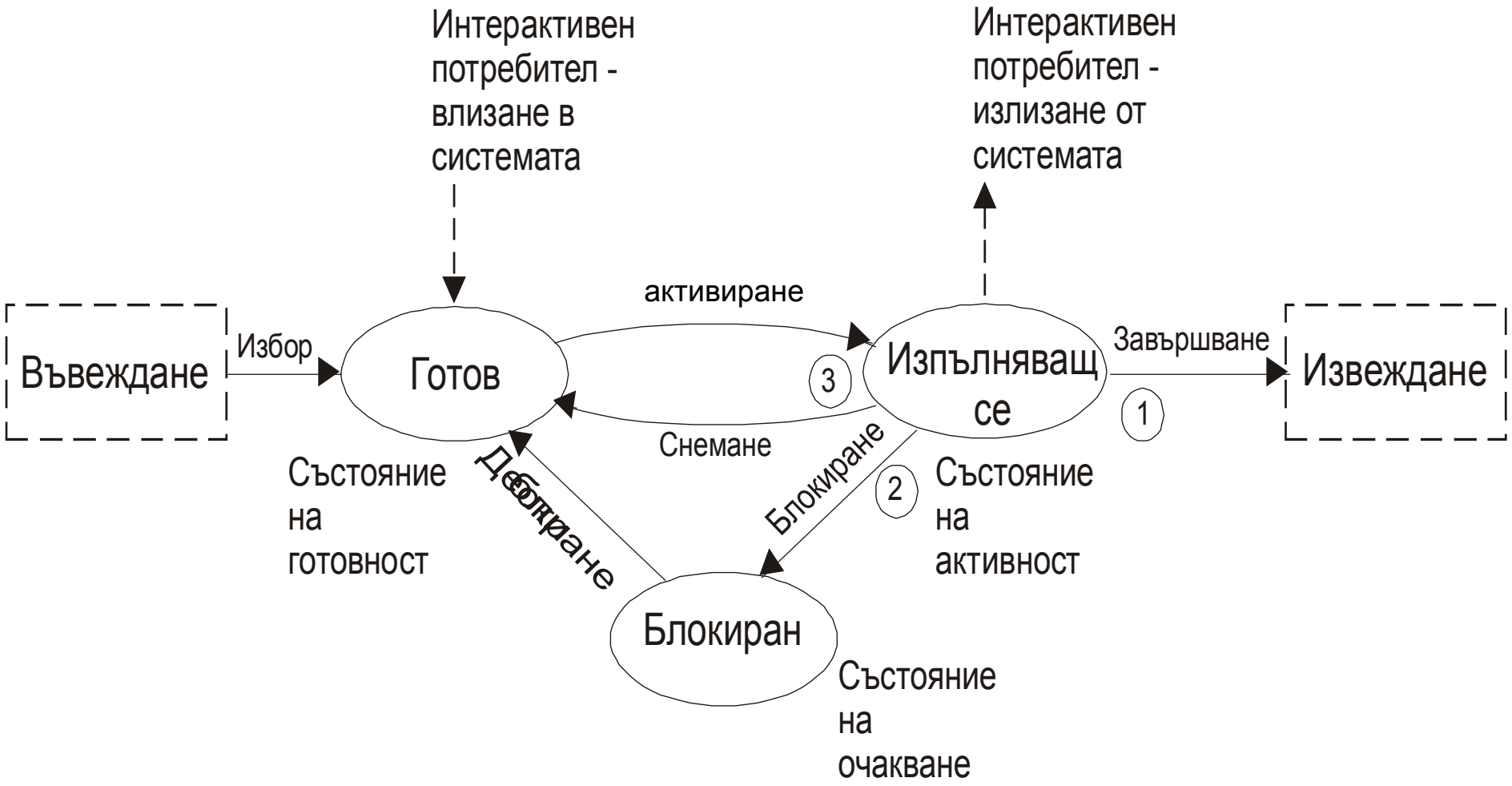
Може да се счита, че в проста многозадачна система всеки процес се намира в едно от следните три състояния:

състояние на готовност (готов);

състояние на блокиране (блокиран);

състояние на изпълнение (изпълняващ се, активен);

Диаграма на състоянията на процесите



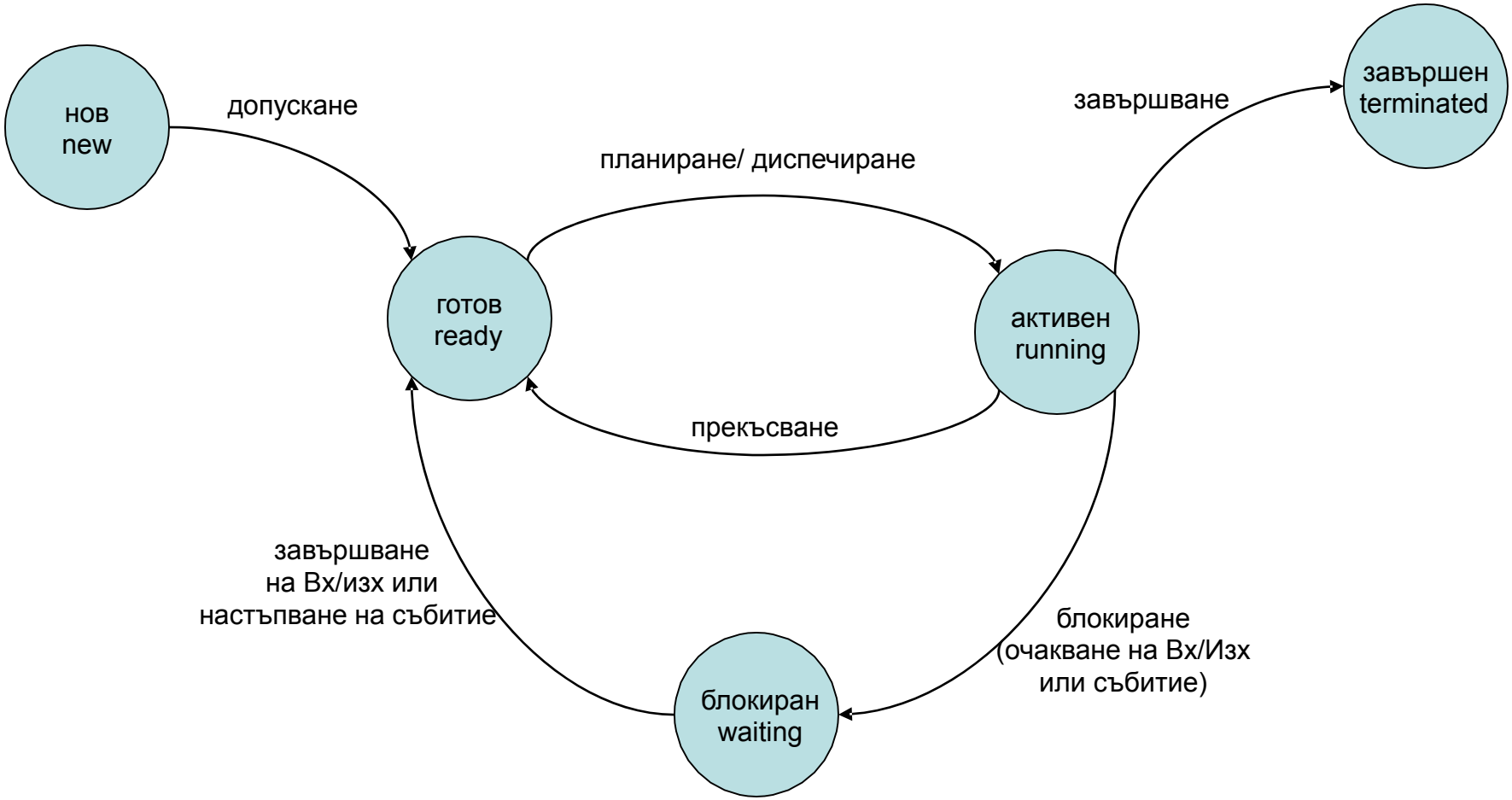
- При наличието на един процесор поне един процес може да се изпълнява.
- Готов процес – вече му е предоставена памет и всички други ресурси, необходими за изпълнение. Процесът очаква само един ресурс – процесора.
- Ако процесът не може да бъде продължен поради отсъствието на някакво събитие, той бива блокиран. Такова събитие е: заявка за допълнителен ресурс; отсъствие на сегмент или страница; вх/изх операции, породени от самия процес и др. За да не заема памет, блокираният процес може да бъде снет от ОП във ВП.

Създаване на процес: независимо от това в какво състояние се намира процесът, веднъж той е получил необходимите ресурси с изключение може би на фактическия процесор. По този начин на процеса е бил предоставен виртуален процесор.

Първоначалното предоставяне на виртуален процесор на процес се нарича създаване на процеса.

В интерактивен режим процес се създава всеки път, когато на потребителя се разреши влизане в системата. Този процес веднага се привежда в състояние на готовност.

В режим на пакетна обработка задачите се поместват в опашка, където се намират в състояние на въвеждане. От там задачата се избира по някаква предварително определена дисциплина и й се предоставят необходимите ресурси, при което се създава процес и се счита, че той се намира в състояние на готовност.

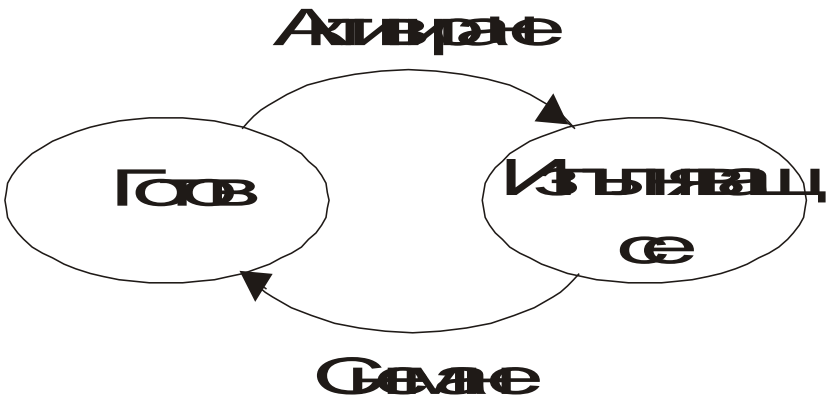


Диаграма на състоянията на процесите

Процесът може да бъде изведен от състояние на активност поради една от следните три причини:

1. Процесът е завършен. Той напуска виртуалния процесор.
2. Процесът е блокиран: процесът спира изпълнение, възможно искайки въвеждане/извеждане, което трябва да бъде изпълнено преди процесът да може да продължи своята работа. Процесът се намира в състояние на очакване до момента, в който бъдат изпълнени очакваните действия;
3. Процесът се снема от управляващата програма, като правило поради завършване на кванта от време, който му е бил предоставен. Привежда се отново в състояние на готовност.

Диаграма на процеси с две състояния



Някои автори отдават предпочитане на диаграма с пет състояния, които освен изброените три включват още състоянията “нов” и “завършен”, като: “нов” е процесът за времето на неговото създаване; “завършен” е процесът, чиято програма е завършила изпълнение. В система с един процесор само един процес може да бъде активен във всеки момент от време. В система с n процесора не повече от n процеса могат да бъдат активни във всеки момент от време

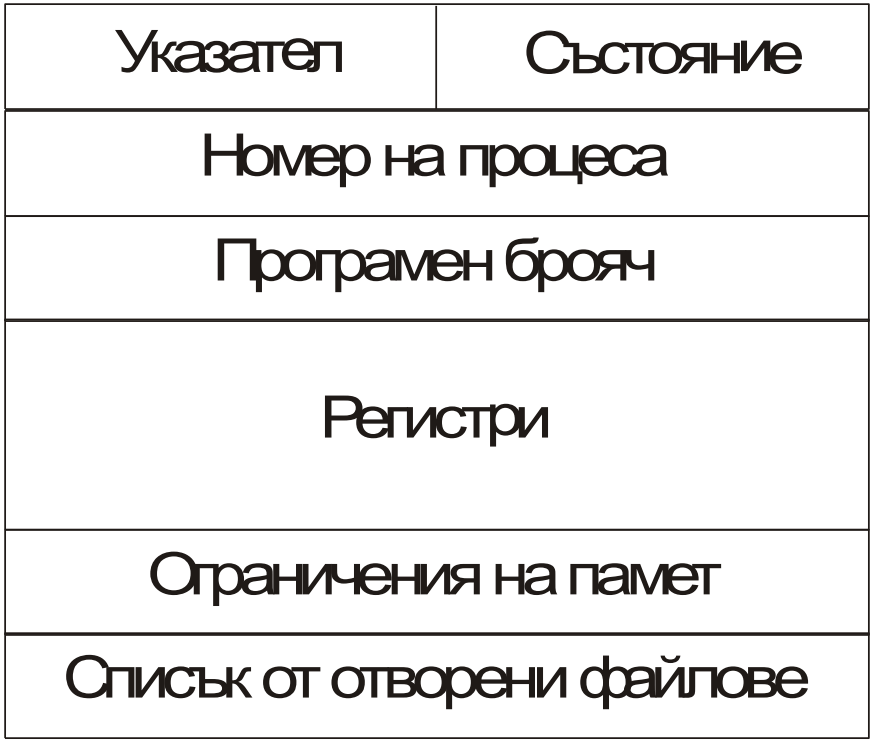
Дескриптори на процесите.

За управлението на процеси с няколко състояния ОС работи с някаква информация за всеки процес. Тази информация се нарича дескриптор на процеса, или блок за управление на процеса (process control block – PCB).

Дескрипторите обикновено се съхраняват в областта памет на управляващите програми. Ако компютърът работи в монополен режим, когато в паметта има само една задача, се използва само един дескриптор. При многозадачен режим се поддържат няколко дескриптора.

Данните, включени в дескриптора, зависят от структурата на хардуерната част на компютъра и от функционалните възможности на ОС.

Дескриптор на процес



В състава на дескриптора може да е включена следната информация:

* състояние: състоянието може да е едно от гореизброените;
* програмен брояч: съдържа адреса на следващата инструкция, която трябва да бъде изпълнена за този процес;
* регистри на процесора: в зависимост от архитектурата регистрите са различни по брой и вид. Те включват акумулаторни, индексни регистри, указатели в стек, регистри с общо назначение (РОН), регистри за състояние. Заедно с програмния брояч съдържанието на тези регистри трябва да се съхрани при настъпване на прекъсване с цел коректно възобновяване в някакъв бъдещ момент на процеса от точката на прекъсване;
* информация за планиране: включва приоритет на процесите, указатели към опашките за планиране (диспечеризация) и други параметри;
* информация за управление на паметта: може да включва значения на базови регистри, адреси на таблица на страници или сегменти в зависимост от приетата организация на паметта;
* статистическа информация: включва използването процесорно време, времеви ограничения, номер на процеса и пр.
* I/O – информация: включва списък от входно/изходни устройства, предоставени на този процес, списък от отворени файлове и пр.

Планиране на процесите

Целта на мултипрограмирането е в системата винаги да има изпълняващи се процеси (активни) за максимално добро използване на процесорните устройства. Съвместното използване във времето е преразпределяне на процесора (процесорите) с висока честота, така че потребителят да може да взаимодейства с всяка стартирана програма.

Опашки

При влизане на задача (програма) в системата, тя се разполага в опашка на заданията (job queue).

Процесите, които са в изпълнимата памет, готови за изпълнение и очакват предоставяне на физически процесор, се поместват в опашка на готовите процеси (ready queue). Обикновено тази опашка се реализира като свързан списък, като се поддържат указатели към първия и последния дескриптор в списъка.

В системата има и други опашки. Когато един процес е активен, той се изпълнява за някакъв интервал от време, след което може да настъпи прекъсване за времето на очакване на дадено събитие, например завършване на входно/изходна операция. В случай на заявка за вход/изход, тя може да касае споделяно устройство. Това устройство може в момента да е заето. Тогава процеса се помества в опашка към устройството (device queue). Всяко устройство има собствена опашка.

Всеки нов процес отначало постъпва в опашката на готовите процеси. Там очаква да бъде избран за изпълнение, т.е. очаква планиране. След като един процес вече е получил процесор и се изпълнява, възможно е настъпване на едно от следните събития:

- процесът да даде I/O заявка и тогава се поставя в опашката към входно/изходното устройство;

- процесът може да създаде друг и да очаква завършването му;

- процеса може принудително да бъде снет от процесора, например в резултат на прекъсване или изтичане на кванта му, след което се помества обратно в опашката готови процеси.

Планировчици

За времето на своето съществуване процесите преминават от едно състояние в друго и съответно от една опашка в друга. Операционната система трябва да избира от тези опашки според някакъв критерий (в този смисъл терминът “опашка” не е съвсем точен, ако изборът не е по дисциплината FIFO). Този избор се извършва от съответния планировчик.

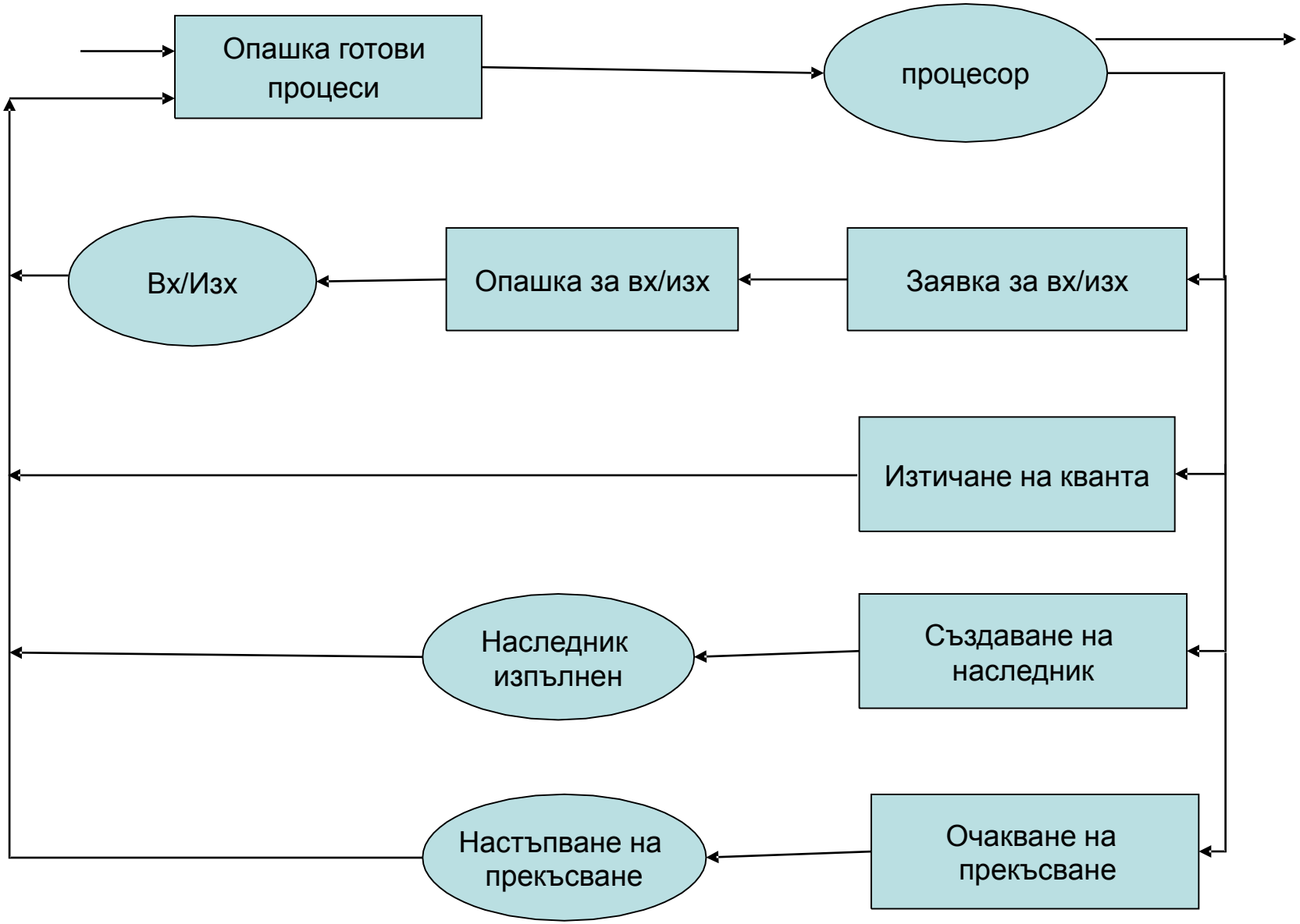
В пакетните системи обикновено съществуват повече програми отколкото могат да бъдат незабавно стартирани. Те се съхраняват във външна памет в опашка на заданията (jobs queue).

Планировчикът от високо ниво (*long-term scheduler, job scheduler*) избира от тази опашка задание, създава процес и зарежда съответната програма в паметта.

Планировчикът от ниско ниво, или още планировчик на процесора (*short-term scheduler, CPU scheduler*) избира един от готовите процеси и му предоставя процесор.

Първата разлика между двата планировчика е в честотата на изпълнение. Планировчикът на процесора работи много по-често и следователно трябва да е много бърз.

Втората разлика се състои в това, че едно задание бива планирано еднократно (т.е. преди то да стане процес), а един процес може да бъде планиран многократно на ниско ниво.

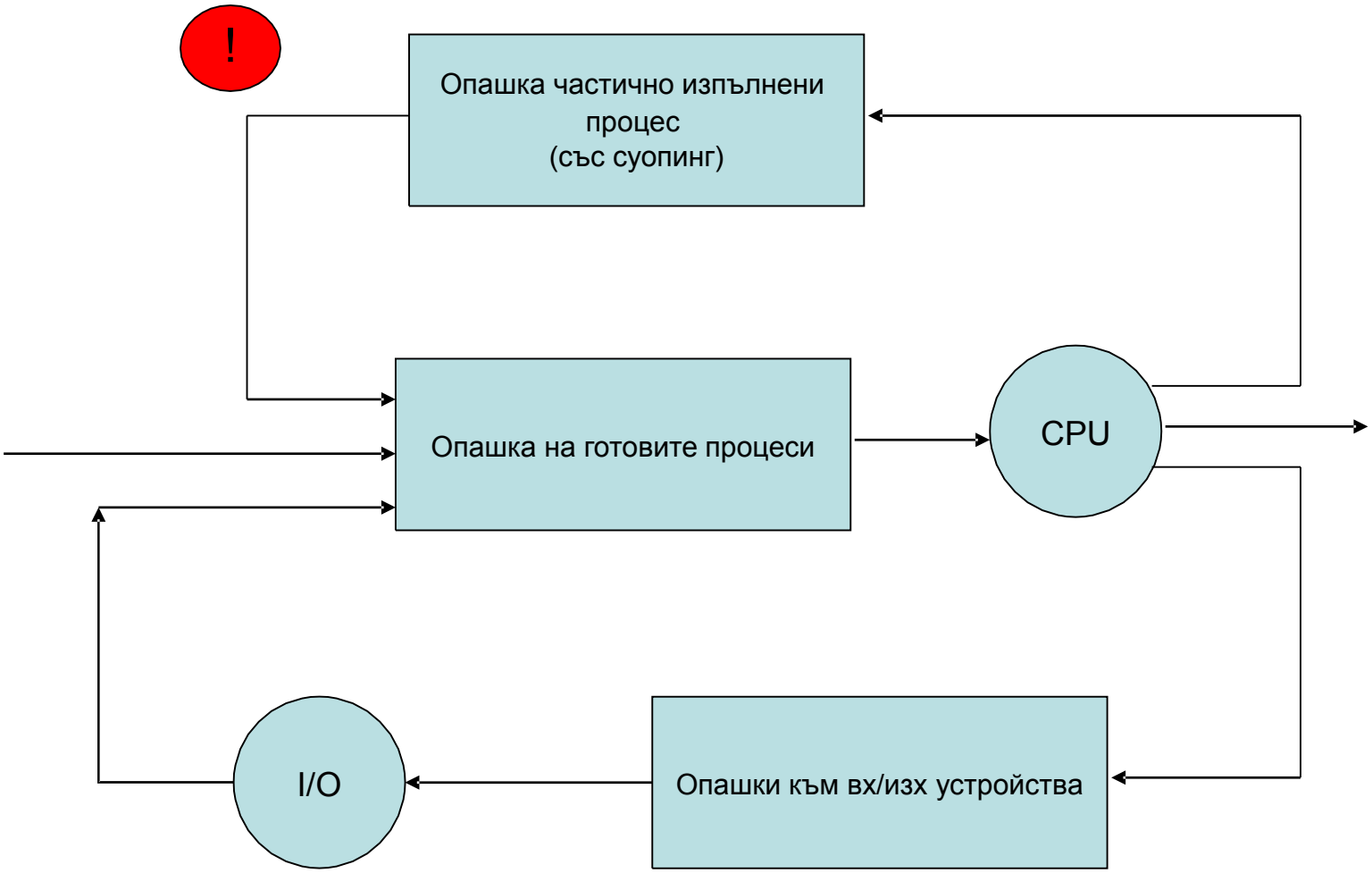


Диаграма на опашките при планиране

В някои източници се използва понятието диспечиране (диспечеризация) вместо планиране на процесора. Тук се придържаме към определените от [Avi Silbershatz] понятия, според които диспечер е този модул на ОС, предоставящ управлението на процесора на този процес, който е избран от планировчика от ниско ниво.

Функциите на диспечера са:

- превключване на контекста;
- превключване в потребителски режим;
- зареждане на инструкцията, от която трябва да се стартира (рестартира) даден процес.



Планиране на средно (междинно) ниво

Контекст на процесите

Спирането и възобновяването на процесите налага съхраняване на състоянията им. Отнемането на процесора от един процес и предоставянето му на друг е известно като превключване на контекста.

Контекстът на процеса представлява съдържимото на всички регистри на процесора, състояние на процеса, информация за управление на паметта – т.е. тези сведения, които са необходими за възобновяване работата му от точката на спиране.

Контекста на процеса се съхранява в дескриптора му. Когато се налага превключване на контексти, ядрото на ОС спира изпълнението на процес, записва контекста на изпълняващия се процес в дескриптора му, а в регистрите зарежда контекста на процеса, избран от планировчика за активиране.

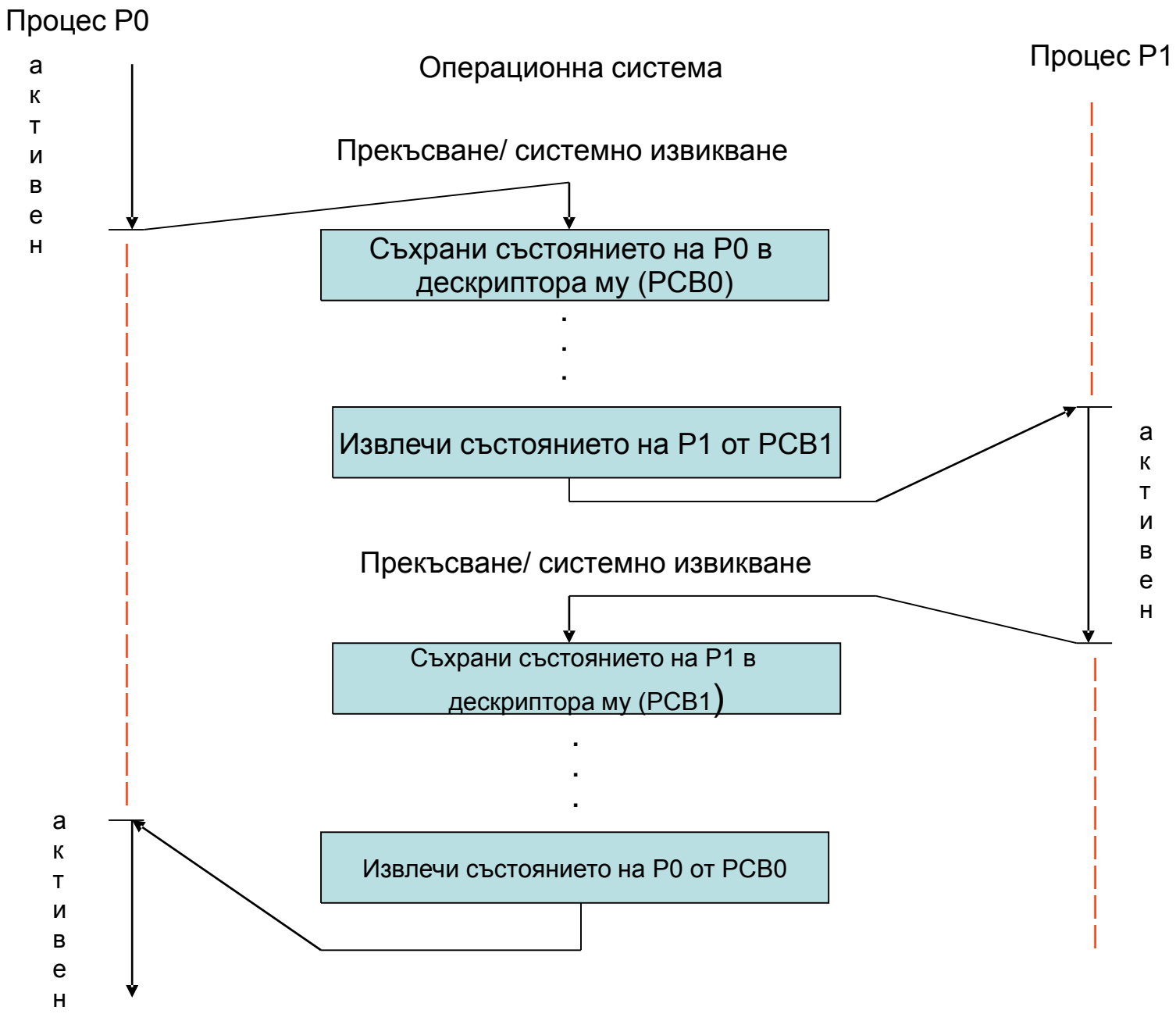
Всъщност превключването не е само между контекстите на двата процеса: ако системата е еднопроцесорна, участва и контекстът на ядрото:

1. превключване от контекст на изпълняващ се процес към контекст ядрото:
2. следва избор на процес за активиране;
3. превключване контекст

Всяко превключване на контексти съдържа двете операции:

а) съхраняване на контекст – на спрения процес;

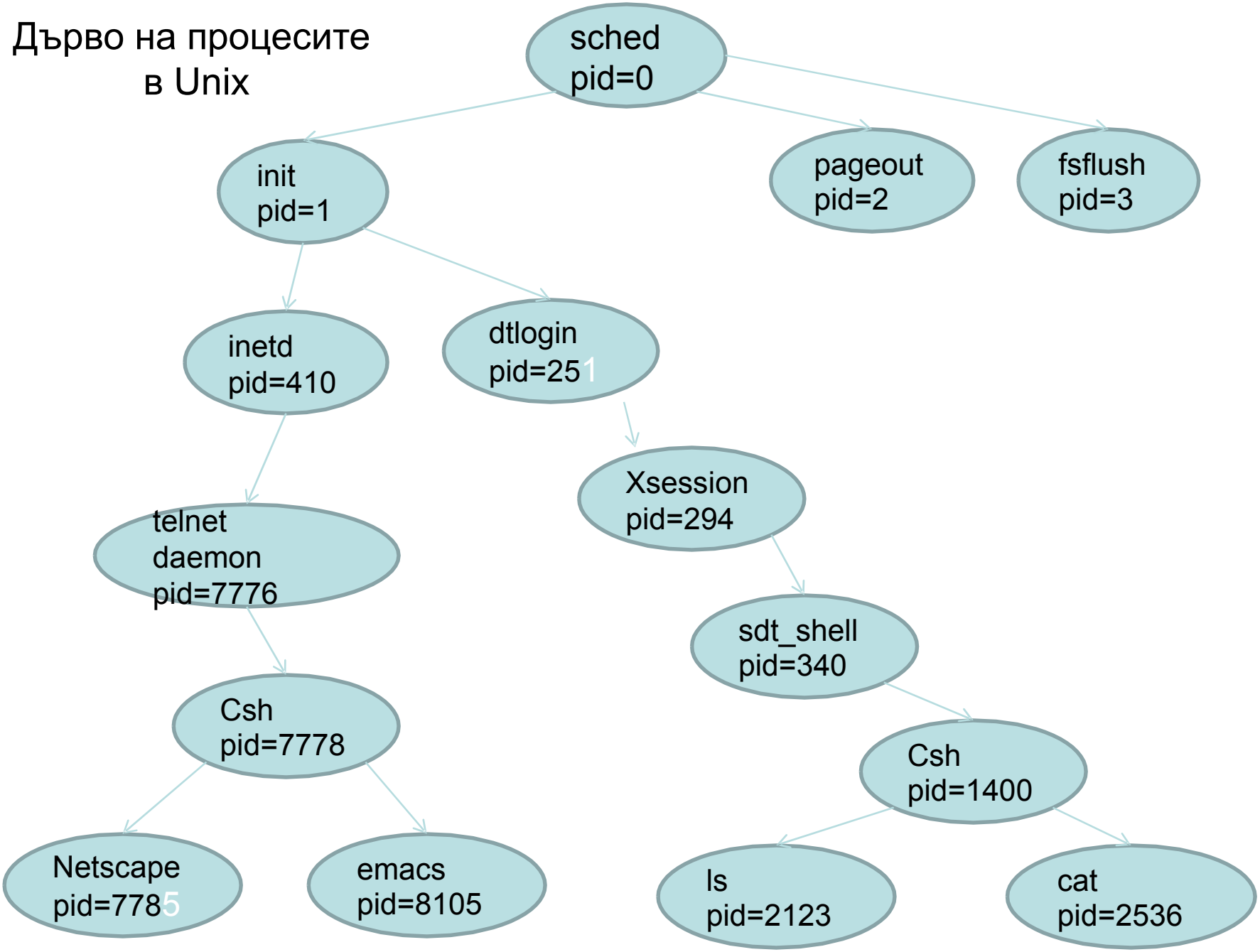
б) възстановяване на контекст – на активирания (възобновен) процес;


Процес P0
активен
Операционна система
Процес P1
Прекъсване/ системно извикване
Съхрани състоянието на P0 в дескриптора му (PCB0)
Извлечи състоянието на P1 от PCB1
активен
Прекъсване/ системно извикване
Съхрани състоянието на P1 в дескриптора му (PCB1)
Извлечи състоянието на P0 от PCB0
активен

Превключването на контексти е пример за период на чиста непродуктивност на системата – не се извършва никаква полезна работа.

Времето, изразходвано за превключване е различно за различните машини и зависи от скоростта на паметта, от броя регистри, които трябва да се копират, от наличието на специални инструкции (например една инструкция за съхраняване и една за извличане на всички регистри). Това време варира от 1 до 1000 микросекунди и много силно зависи от хардуерната поддръжка.

Дърво на процесите в Unix



Кореновият процес е Sched, pid=0

pageout, pid=2 управлява паметта

fsflush, pid=3 управлява файловата система

init, pid=1 е родителски за всичко потребителски процеси

inetd, pid=140 управлява мрежовите услуги (telnet и ftp напр.)

dtlogin, pid=251 управлява екрана за регистрация на потребителя

При регистриране на потребител dtlogin създава X-Windows сесия (Xsession, pid=294), който от своя страна създава std_shell, pid=340. След това std_shell създава потребителския команден интерпретатор C-shell: csh, pid=1400, който предоставя команден интерфейс на потребителя. Чрез командите към него могат да бъдат създадени редица процеси, като ls и cat.

(напр. ls разпечатва всички файлове в дадена директория:

ls [option][name]

ls -l

Тази команда ще разпечата списък на всички файлове в текущата директория, правата за достъп за всеки файл, размер, дата на последната модификация и име на файла или директорията.

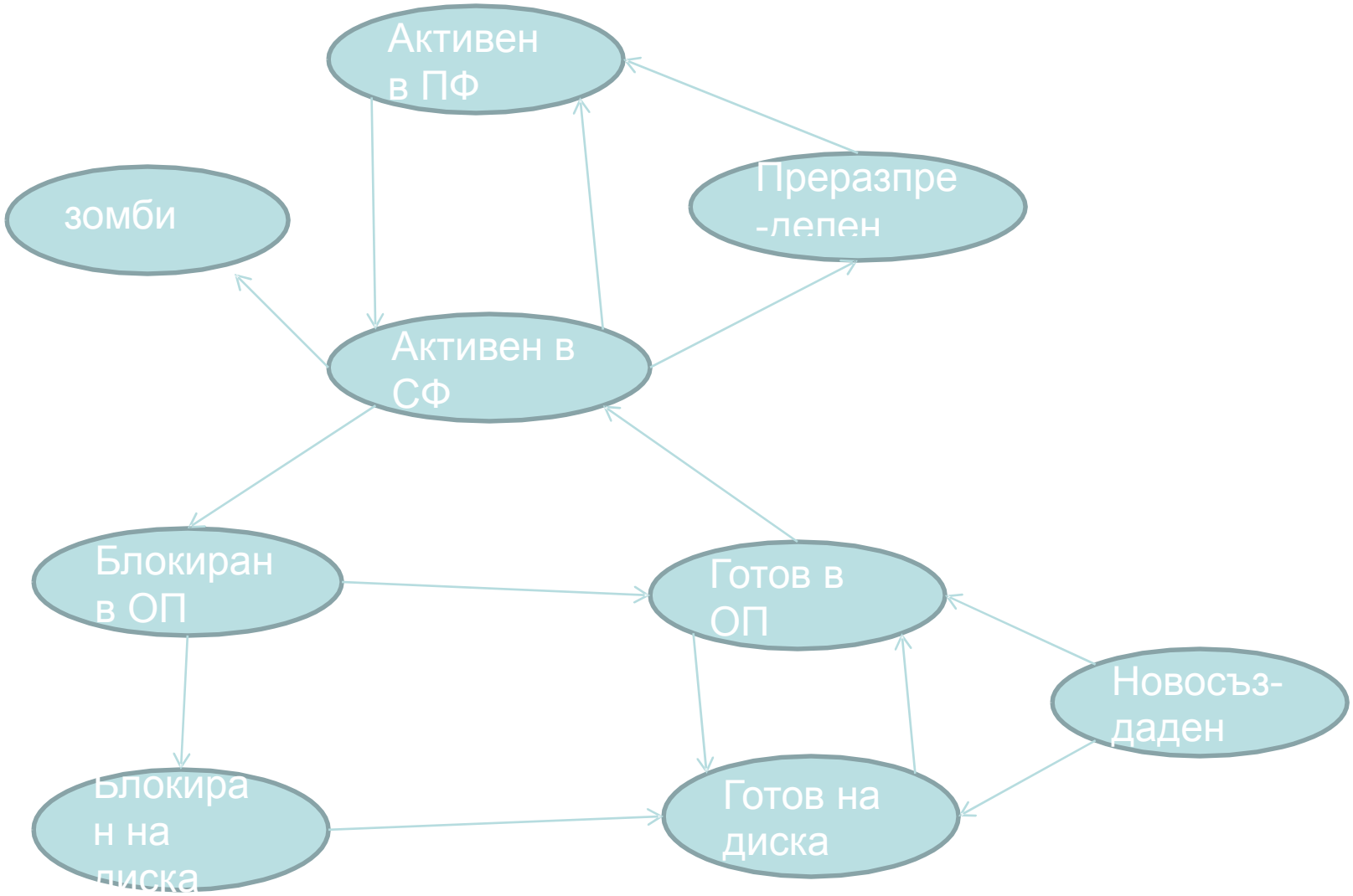
Permissions	***Directories***		***Group***	***Size***	***Date***
Directory or file					
drwx------	*2*	*users*	*4096*	*Nov 2 19:51*	*mail/*
drwxr-s---	*35*	*www*	*32768*	*Jan 20 22:39*	*public_html/*
-rw-------	*1*	*users*	*3*	*Nov 25 02:58*	*test.txt*

Процеси в Unix

Моделът на процесите в Unix SystemV включва девет състояния:

1. Активен в ПФ (потребителска фаза). Изпълнява се в потребителски режим, ЦП изпълнява инструкции от потребителската програма, свързана с процеса.
2. Активен в СФ (системна фаза). Изпълнява се в режим ядро, като ЦП изпълнява инструкции от ядрото, т.е. Процесът е инициирал изпълнението на модули от ядрото.
3. Готов в ОП – процесът е в паметта и чака процесор.
4. Готов на диска – процесът е готов за изпълнение, но планировчикът трябва да го зареди в ОП, за да може да бъде избран за активиране.
5. Блокиран в паметта – намира се в паметта и очаква настъпване на събитие, за да бъде приведен в състояние 3.

6. Блокиран на диска – процесът е блокиран и планировчикът от средно ниво го е изхвърлил на диска (суопинг) в суоп-област, която се разглежда като разширение на паметта.
7. Преразпределен (preempted). Процесът е бил на път да се върне в състояние 1 след завършване на системната фаза, но планировчикът му отнема ЦП с цел да го предостави на друг процес.
8. Новосъздаден – начално състояние, с което процес влиза в системата. Създаден, но не напълно работоспособен. Това е преходно състояние при създаването на всеки процес.
9. Зомби. Процесът е изпълнил системния примитив exit и вече не съществува, но дескрипторът му е още в паметта. За процеса се съхранява информация, която може да бъде предадена на родителския му процес. Всички процеси преминават през това състояние преди за изчезнат от системата.



Диаграма на състоянията на процесите в Unix System V

Контекст на процесите в Unix

Контекстът на процес в Unix има три компонента:

1. Потребителска част – определяща работата на процеса в потребителска фаза и включва образа на процеса;
2. Машинна (регистрова) част – включва съдържанието на машинните регистри: PC, съдържащ адреса на следващата машинн инструкция; PSW, определящ режима на работа на процесора по отношение на процеса, признак за резултата от последната инструкция и др,; други регистри, съдържащи данни на процеса.
3. Системна част – включва структури в пространството на ядрото, описващи процеса.

Образ на процес

Образът на процес съдържа програмния код и данните, използвани от процеса в потребителска фаза. За създаването му служи файл с изпълним код. Състои се от следните части:

1. Код (text). Съдържа машинните инструкции на приложението – т.е. Инструкциите, изпълнявани от процеса в потребителска фаза.
2. Данни (data). Съдържа глобалните данни, с които процесът работи в потребителска фаза. Инициализираните данни се зареждат от изпълнимия файл, а за неинициализираните компилаторът при генерацията на код е записал в изпълнимия файл информация за размера им, и при създаване на образа на процеса в паметта се заделя необходимия обем памет.
3. Стек (stack). Създава се автоматично с размер, определян от ядрото. Необходим е за обръщане към потребителските функции. Във всеки слой (стекова рамка) се съдържат копия на локалните данни на функция, неизпълнила return, както и адрес на връщане към извикалата я програмна част. Този стек се използва при изпълнение на процеса в потребителска фаза. За системна фаза се използва стек на ядрото.

В Unix-системите тази логически единици, на които се дели образа на процеса, се наричат области (региони). Всяка област е последователна част от виртуалното адресно пространство на процеса и се разглежда като независим обект за защита или съвместно ползване. Така няколко процеса могат да използват една и съща област за код, т.е едно копие на кода. Общите области се считат за за част от образа на всеки процес.

Понятието “област” е логическо и не зависи от приетия метод за управление на паметта.

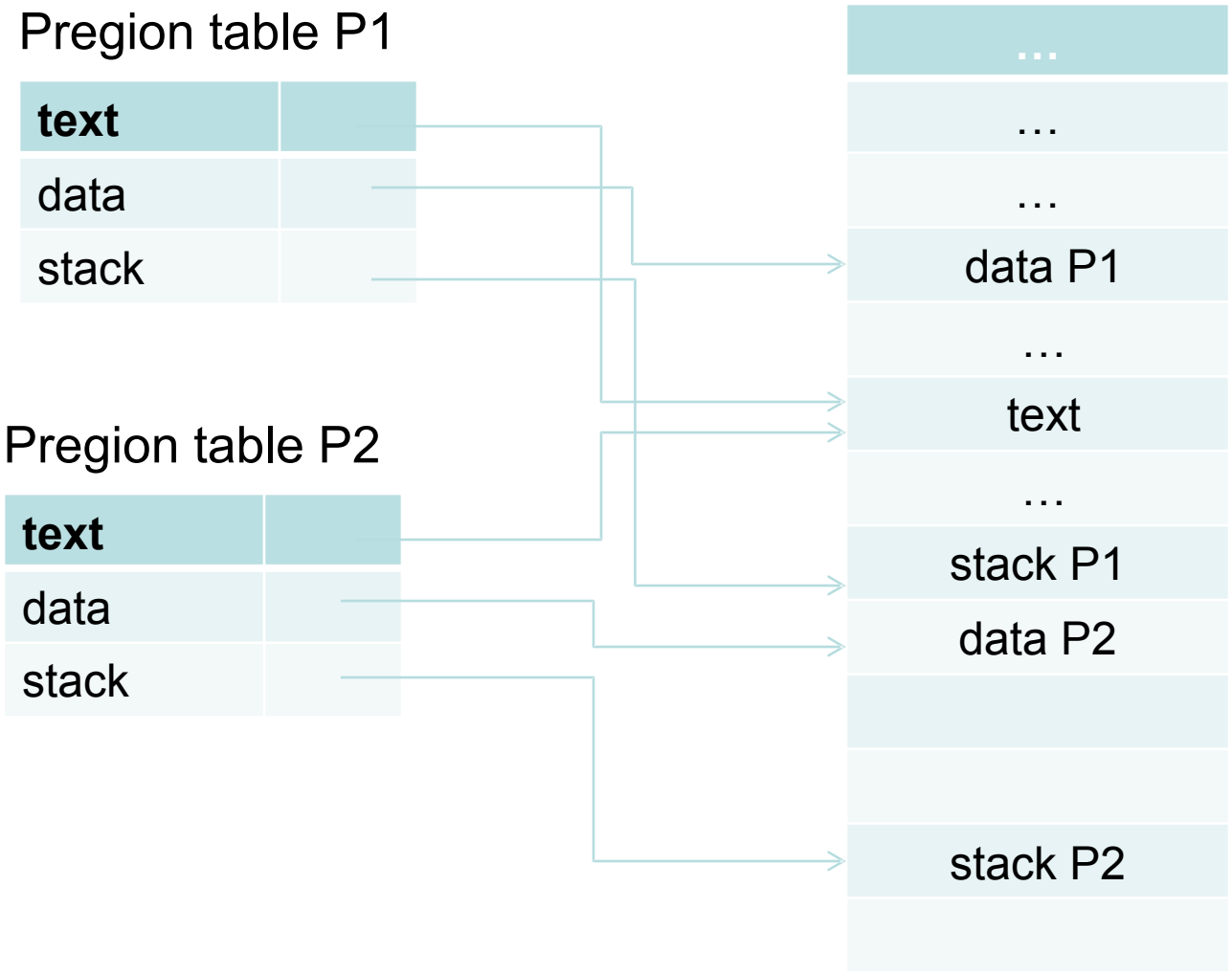
Информацията за всички активни области се съдържа в Таблицата на областите (Region Table). Всеки запис в нея описва една област и съдържа:

- тип – код, данни, стек, ...
- размер;
- адрес в паметта;
- състояние – зарежда се, заключена, ...
- брой процеси, които я споделят.

Всеки процес има Pregion table – таблица на своите области. Всеки запис в тази таблица описва една от областите на процеса и съдържа:

- тип на разрешения на процеса достъп до областта – read-only, read-write, read-execute;
- виртуален адрес на областта в процеса;
- указател към таблицата на областите (Region table);

Таблицата Pregion Table и съответните записи от таблицата Region Table са части от системното ниво на контекста на процеса. Към това ниво принадлежат и Таблицата на страниците (Page Table) или сегментите (Segment Table) на процеса, в зависимост от приетия метод за управление на паметта.


Pregion table P1
text
data
stack
Pregion table P2
text
data
stack
...
…
…
data P1
…
text
…
stack P1
data P2
stack P2

Дескриптори в Unix

Таблицата на дескрипторите (таблицата на процесите) е масив от дескриптори в пространството на ядрото. За Unix/Linux системите информацията, описваща един процес, обикновено е разделена на две системни структури:

- таблица на процесите - съдържа данни за процеса, които са нужни и достъпни за ядрото независимо от състоянието на процеса;

- потребителска област (U-area) – съдържа данни за процеса, които са необходими и достъпни за ядрото само когато е активен.

Някои от полетата в таблицата на процесите са:

- идентификатор – pid;
- идентификатор на родителя – ppid;
- идентификатор на група – gpid (pid на процеса, лидер на група) ;
- идентификатор на сесия - spid (pid на процеса – лидер на сесия);
- състояние;
- събитие, настъпването на което процесът очаква в състояние “блокиран”;
- полета, осигуряващи достъп до образа на процеса и U-area (адрес на Pregion Table);
- приоритет на процеса – за планиране;
- полета, в които се съхраняват изпратените към процеса, но все още необработени от него сигнали;
- полета за времеви данни: използвано процесорно време в системна и потребителска фаза, таймер, зареден с примитива `alarm` и др.
- код на завършване на процеса.

Полета в **U-area**:

- таблица на файловите дескриптори;
- текуща директория – указател към запис от таблицата на индексните описатели;
- сменена коренова директория;
- управляваща конзола – указател към tty на терминала или NULL , ако такъв няма;
- маска, използвана при създаване на файлове и заредена с примитива umask;
- реален потребителски идентификатор (ruid – определя потребителя, създал процеса);
- ефективен потребителски идентификатор (euid – определя правата на процеса);
- указател към записа на таблицата на процесите;
- реален идентификатор на потребителска група;
- ефективен идентификатор на потребителска група;
- полета, определящи реакцията на процеса при получаване на различните сигнали;
- полета за текущия системен примитив и върнати от него стойности.

Въпроси и задачи :

1. Кои са трите състояния на процесите в универсална КС?
2. Кой от преходите между следните състояния не е възможен:
   а) готов-> активен;
   б) готов-> блокиран;
   в) активен-> блокиран;
   г) блокиран-> готов.
3. Какво е “дескриптор на процес” (Process Control Block) ?
4. Какво е “ контекст на процес”?
5. Колко превключвания на контексти се извършват при спиране на потребителски процес и стартиране (или възобновяване) на друг потребителски процес?
6. Какви опашки от процеси има в една стандартна КС?

***Забележка**: Silberschats, A., Galvin, P., Gagne, G. Operating Systems Concepts with JAVA. Int. Student Version, John Wiley & sons, 2011.*

Въпроси и задачи:

Опишете разликата между процес и програма.

7. Суопинг се извършва от:

   а) планировчика за заданията (long-term scheduler);

   б) планировчика на процесора (CPU-scheduler);

   в) диспечера;

   г) планировчика от междинно ниво (medium-term scheduler).

8. В какви фази могат да работят активните процеси?

9. При стартиране Unix- ядрото работи с:

   а) pid=1;

   б) pid=0;

   в) pid=1000;

   г) pid=2;

ЛЕКЦИЯ 6

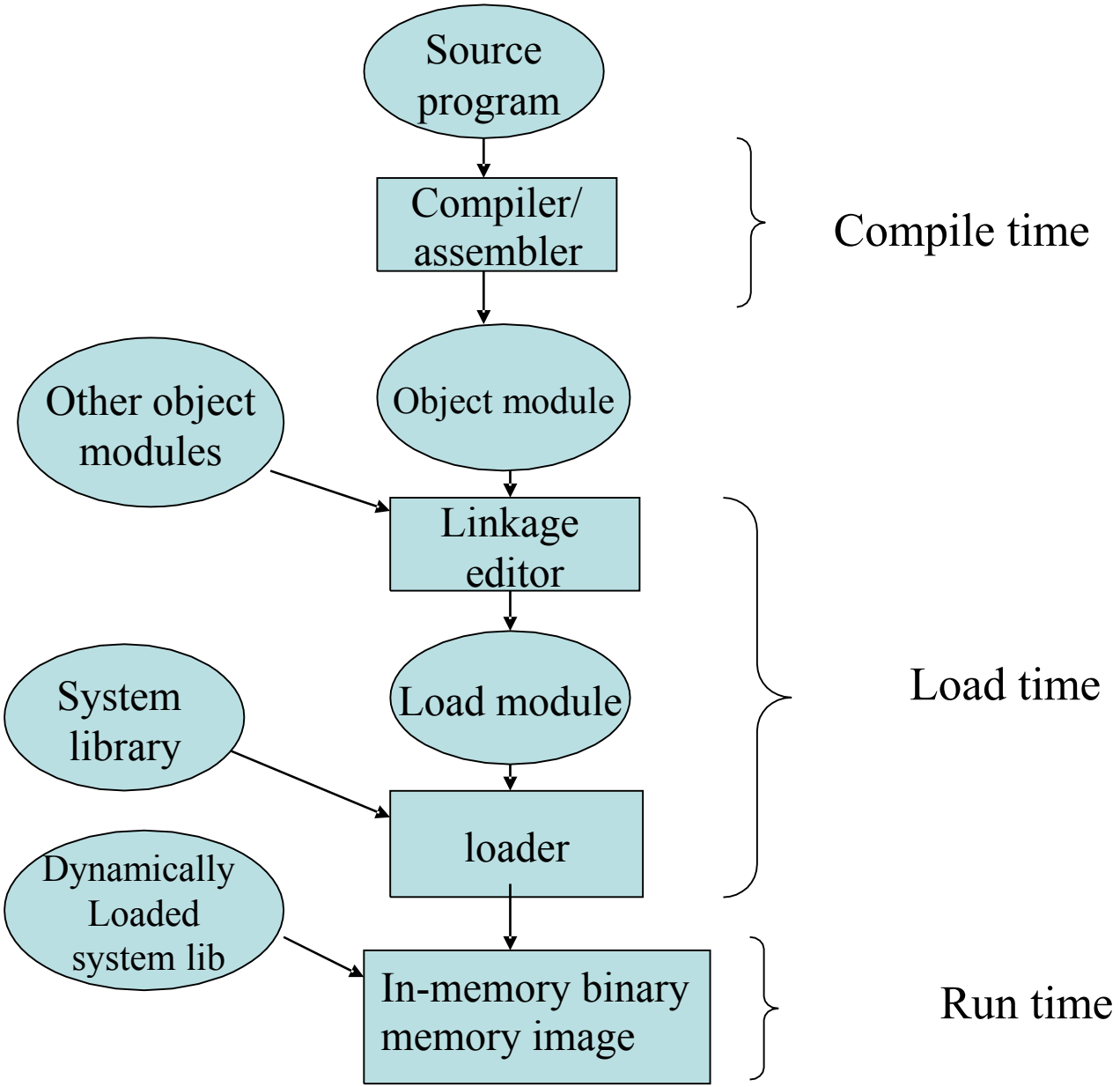
УПРАВЛЕНИЕ НА ПАМЕТТА

За да може да бъде изпълнена, всяка програма трябва да бъде заредена в основната памет. Преди да започне изпълнение, програмата преминава през редица фази (сл.2). Адресите, използвани в програмата, се представят по различен начин за всяка фаза. В изходната програма адресите имат символен вид: напр. *MaxValue*; компилаторът свързва тези адреси с относителни, напр. “ *16 байта от началото на този модул”;* свързващият редактор или зареждащата програма ги превръща в абсолютни, напр. 76016. Всяко такова свързване представлява проектиране на едно адресно пространство върху друго.

Свързването на инструкции и данни с конкретни адреси в паметта може да се изпълни на всеки от следните етапи:

- по време на компилация (compile time);
- по време на зареждане (load time);
- по време на изпълнение (run time).

УПРАВЛЕНИЕ НА ПАМЕТТА



Просто непрекъснато (последователно) разпределяне на паметта

В този случай на потребителя се предоставя цялата памет с изключение на тази фиксирана област, в която се намира ядрото на ОС. Обикновено то се разполага в областите с крайни адреси (начални или последни):

Ядрото на ОС в крайните адреси на паметта

Всички следващи клетки образуват непрекъсната линейна област, която се предоставя на програмата на потребителя. При всяко свое изпълнение тази програма ще заема клетки с едни и същи адреси. Следователно, потребителската програма може да бъде написана така, че при изпълнението ù да се използват фиксирани физически клетки. При такъв подход е необходим по-прост транслатор, отколкото, ако използваните при изпълнение на програмата клетки са неизвестни на етапа на транслация.

Абсолютно зареждане

Най-простото непрекъснато разпределяне се извършва с помощта на така наречената програма за абсолютно зареждане .

Функции :

- четене на текста, съдържащ програмата и данните;
- поместване на данните на определените адреси;
- предаване на управлението на даден адрес.

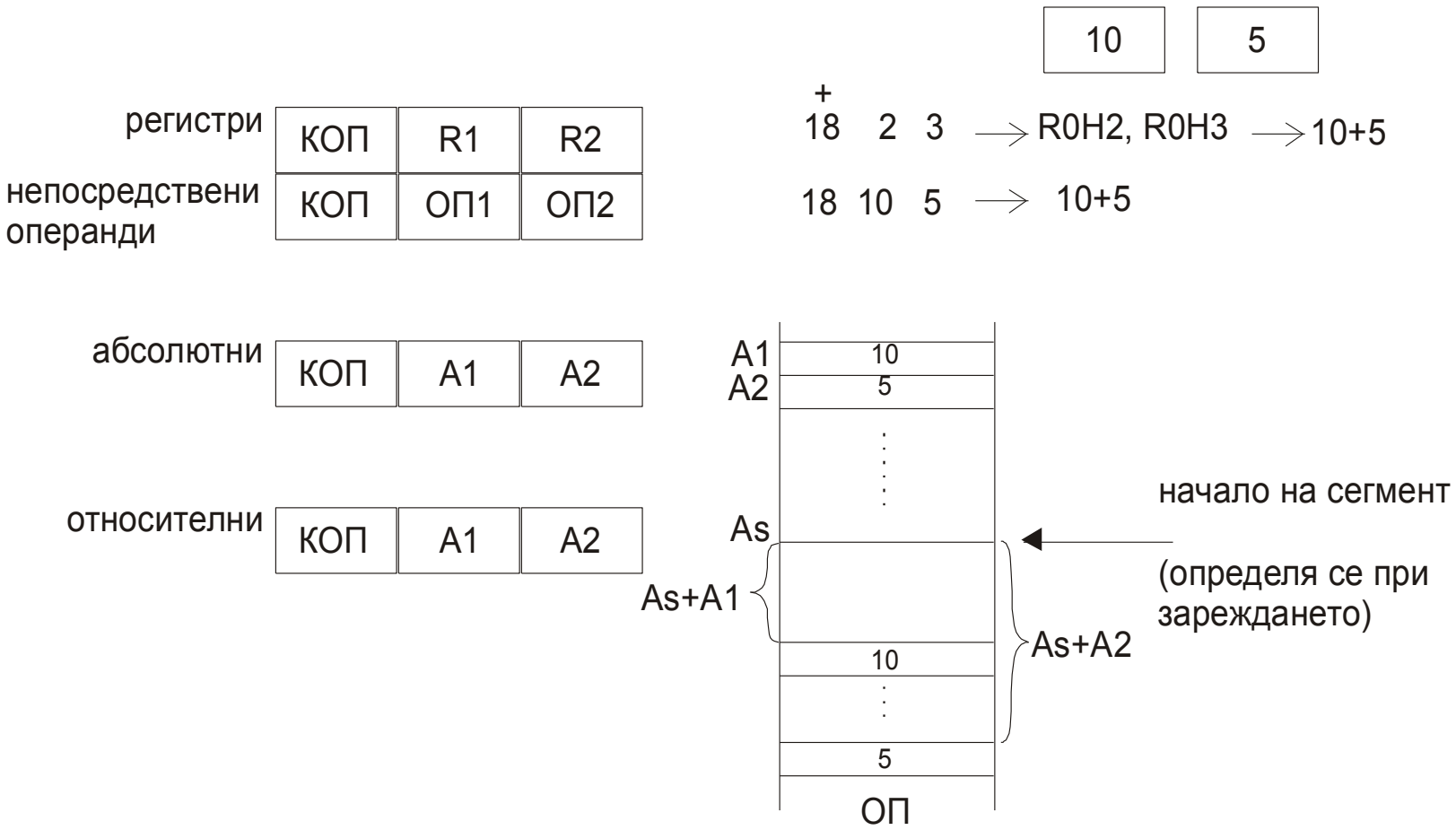
Транслираният текст се зарежда на части, наречени записи, в паметта. Всеки запис съдържа текст, подлежащ на зареждане, и адрес в паметта.

Ако записите са с променлива дължина, тя също се указва.

Настройващо зареждане. Способи на адресация

Настройващото зареждане предвижда подготовка на програмата от транслатора по такъв начин, че поместването ù в паметта да се определя в момента на зареждане. В адресните полета на инструкциите от потребителската програма не се определят физическите адреси памет. Необходимото съответствие се определя по-късно. Подготвената в такава форма програма се нарича премествaема, а управляващата програма, която извършва настройването за определени физически адреси се нарича програма за настройващо зареждане.

Програмата, подлежаща на зареждане и настройка, се получава в резултат на работата на транслатора и се нарича обектна програма или обектен код. Настройваната порция обектен код обикновено се нарича сегмент. Адресните полета на инструкциите от обектната програма трябва да бъдат коригирани от настройващата зареждаща програма с величина, наречена константа на отместване. Полетата за данни и команди, за които е необходима такава корекция, се наричат относителни, тъй като техните значения се задават по отношение на началото на сегмента. Съответно полетата, които не трябва да се коригират, се наричат абсолютни.



Абсолютни и относителни адресни полета

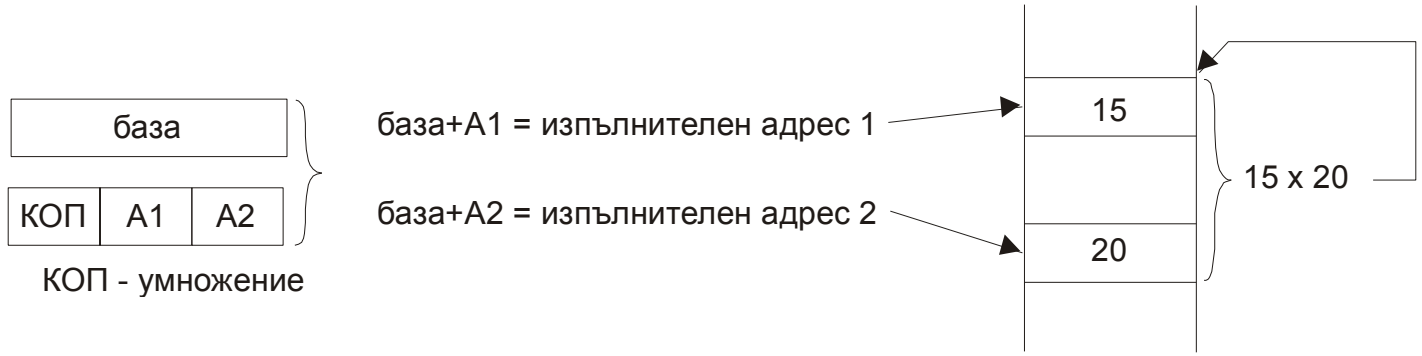
Реализацията на настройката на програмата зависи от използваните в машината методи на адресация, три от които са най-общи:

1. Пряка адресация. Адресното поле е число, непосредствено представляващо адрес на клетка от паметта.

2. Косвена с неявно базиране. Адресното поле съдържа число, задаващо разликата или отместването, между физическата клетка, към която се обръщаме и началото на програмния сегмент

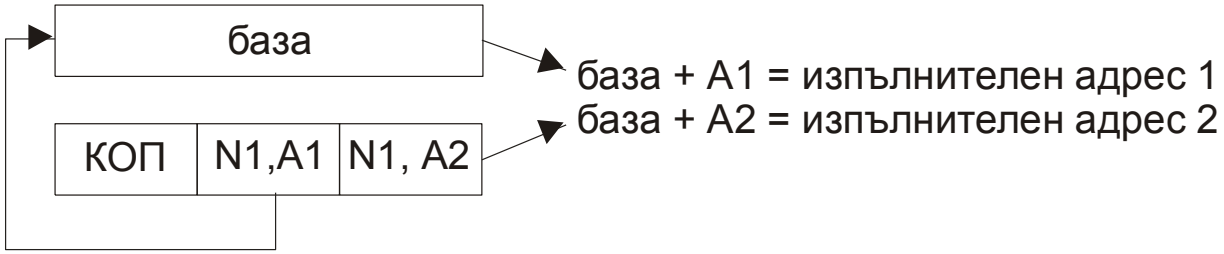
Тъй като в командата базовият регистър явно не се посочва, този метод се нарича косвена адресация с неявно базиране.

Преди да започне изпълнението на програмата ОС записва в така наречения базов регистър адреса на началото на сегмента – базов адрес. При изпълнението на инструкцията отместването автоматично се прибавя към значението на базовия регистър. Полученият по такъв начин изпълнителен адрес определя клетката ОП, с която ще се работи.



Изчисляване на изпълнителен адрес при косвена адресация с неявно базиране

3. Косвена адресация с явно базиране. Адресното поле съдържа две числа, едното от които е номерът на базовия регистър, а второто задава отместването. Зареждането на началото на сегмента в съответния базов регистър се задава от програмата или от транслатора.



Косвена адресация с явно базиране

Динамично зареждане

Ако на някоя задача е необходима памет, размерът на която превишава предоставеното ù пространство, то за изпълнение на програмата са необходими по-развити дисциплини за планиране. Ако програмата и данните могат да бъдат разделени на сегменти по такъв начин, че не всички сегменти едновременно да се намират в ОП, то те ще могат да използват ОП последователно във времето. Сегмент, който в даден момент не е нужен, може да се намира във вторичната памет дотогава, докато не възникне необходимост да бъде зареден в ОП, припокривайки един или повече сегменти, които вече не са нужни или поне няма да са нужни в течение на някакъв интервал от време

Разпределяне с няколко непрекъснати раздела

Многозадачността позволява да се избягнат големите загуби, свързани с простото непрекъснато разпределяне. В режим на мултипрограмиране в ОП едновременно могат да се намират няколко програми, които се изпълняват паралелно. Основна стратегия на многозадачността е максималното натоварване на процесора.

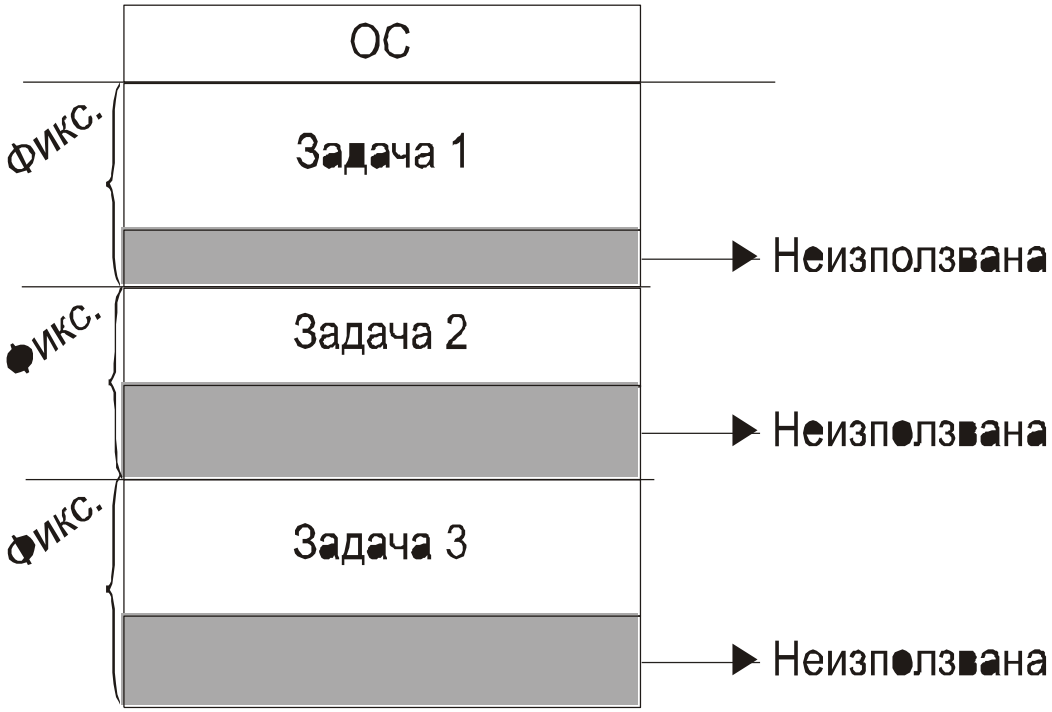
За разполагането на няколко програми паметта се разбива на области. Те са определени от своите граници.

Възникват следните проблеми:

- Защита: всяка програма да не излиза извън рамките на определените граници
- Определяне броя и размера на разделите. Определено е, че за да бъде зает процесорът 90 % от времето, е необходимо най-често коефициентът на мултипрограмиране (броят на паралелно изпълняваните задачи) да е 4 или 5 и следователно са необходими толкова раздели. При това е важно дали разделите и границите между тях са фиксирани на променливи.

Раздели с фиксирани граници

Най-простият механизъм на многозадачност предвижда фиксирано количество, размер и поместване на разделите. Средния размер на раздела се определя от количеството раздели и наличната памет. Тъй като задача не може да започне изпълнение, ако не се помества даже в най-големия раздел памет, то някои раздели могат да бъдат разширявани за сметка на други. В някои системи броят и размерът на разделите могат да се изменят от програмата.



Външна фрагментация

Прекъснато разпределяне на паметта

Един метод за борба с външната фрагментация е предоставянето на всеки процес на няколко непрекъснати области в паметта. Такова разпределение избавя потребителя от необходимостта да знае нещо за физическите адреси, където се разпределя неговата програма.

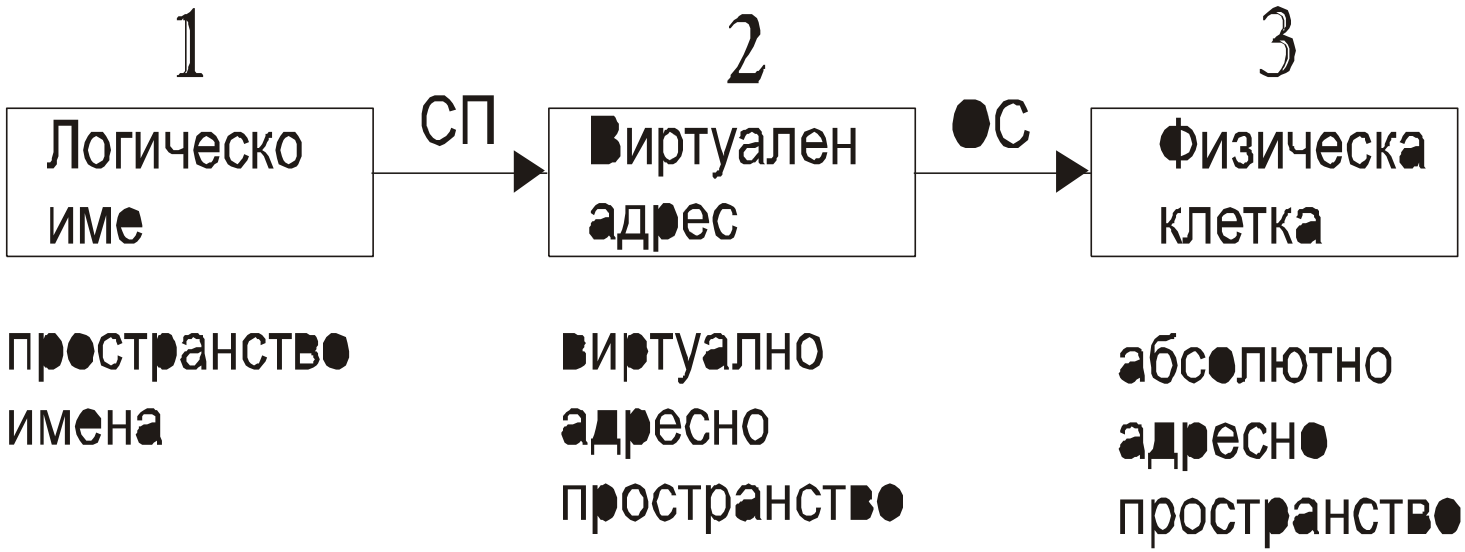
Физическата памет (абсолютна, фактическа, реална) е фиксиран брой подредени клетки, обръщането на които се извършва с помощта на уникално число с фиксирана дължина. Броят използваеми физически клетки е постоянен, предварително определен и представлява това, което ще наричаме физическа памет на компютъра. Друг термин е “абсолютно адресно пространство”.

С изключение на програмирането на машинен език, програмистът се обръща към паметта посредством множество логически имена, които обикновено са символни (не числови) и за които не съществува отношението подреждане. Символните имена, използвани от един програмист, са уникални. Но няма гаранции, че потребителите в една система няма да използват същите имена за други обекти. Следователно, логическите имена не са уникални за множеството потребители на една компютърна система. Освен това множеството логически имена не е предварително определено и се изменя във времето.

За множеството логически имена, използвани от един потребител, ще се използва термина “пространство имена”.

Програмната система трябва да свърже всяко символно име с физическа клетка, тоест за съвкупността потребителски програми системата трябва да осигури проекция (съответствие) от пространството имена върху физическата памет. В общия случай това се извършва на два етапа:

1) от системата за програмиране;
2) от операционната система.



Трансформиране на логическо име в абсолютен адрес

Частен случай 1: тъждественост на виртуалното адресно пространство и абсолютното адресно пространство. Тогава съответствието се осигурява от системата за програмиране, генерирайки абсолютна програма. Операционната система само осигурява непрекъснатото разпределяне на паметта.

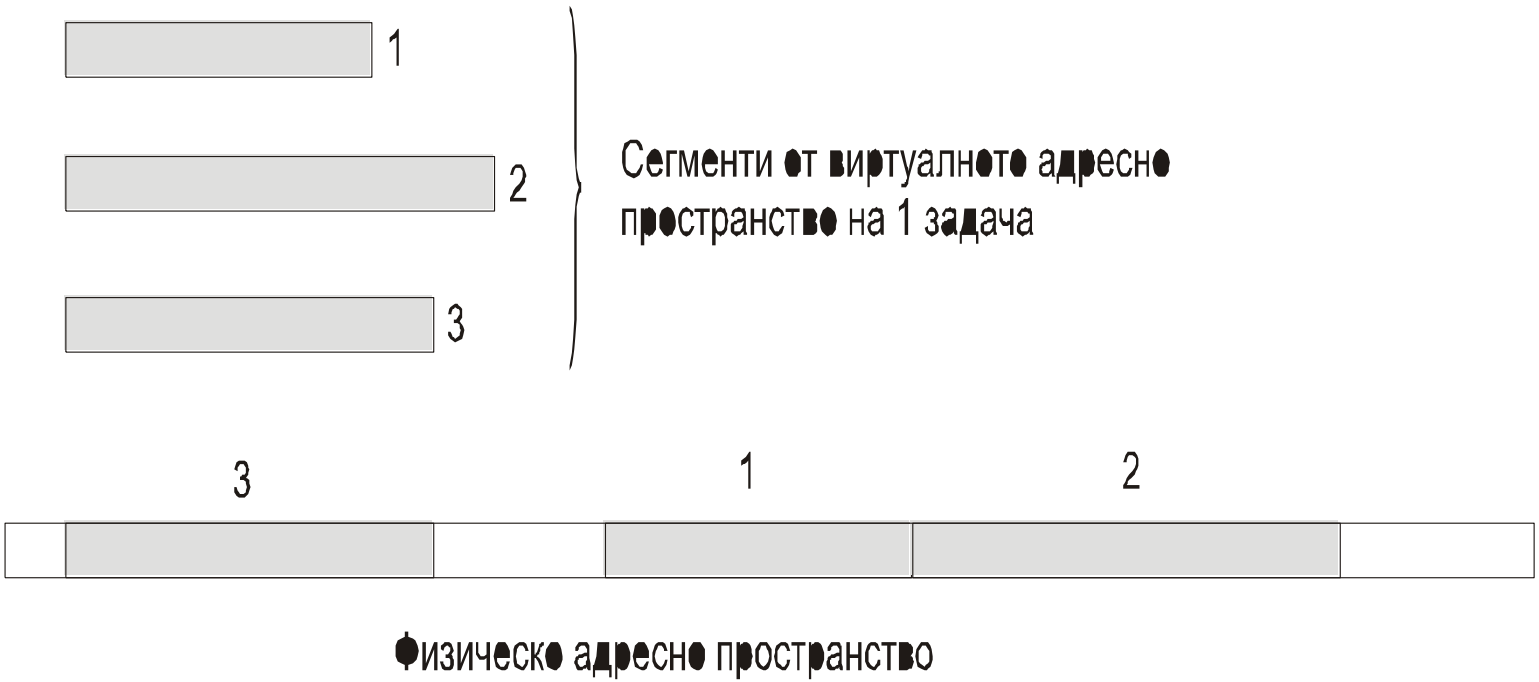
Частен случай 2: тъждественост на пространството имена и виртуалното адресно пространство. Тук съответствието се осигурява от операционната система. Използва се рядко, тъй като се изпълнява за всяко име и доста време изразходва за анализ и класификация на имената. По-често се среща при интерпретаторите, за които стадиите на транслация и изпълнение са неразличими.

Терминът “виртуална памет” се отнася фактически за тези системи, които съхраняват виртуалния адрес по време на изпълнение. Уточняваме, че “виртуално адресно пространство” и “виртуална памет” са **различни** термини. На този етап ще работим само с първия – виртуално адресно пространство, виртуални адреси, а прекъснатото разпределяне на паметта отначало ще разгледаме без концепцията за виртуална памет.

Сегментна организация на паметта

Адресното пространство на един процес може да бъде разглеждано като съвкупност от логически части или сегменти. Потребителят се обръща към всяка клетка от сегмента по следния начин: име на сегмента, което по време на транслация се преобразува в число, и отместване по отношение началото на сегмента. Размерът на сегментите може динамично да се изменя, така че сегментите са удобни за достъп до данни, чиито размери се изменят по време на изпълнение. Адресацията вътре в сегмента е непрекъсната. Сегментите на една задача могат да не представляват непрекъсната област памет и още повече – да не се намират всички едновременно в паметта.

В основата на сегментирането лежи следният принцип: вторият етап от проекцията на пространството имена върху абсолютното адресно пространство, а именно преходът от виртуалните адреси към физическите, да се извършва **при всяко обръщане към паметта**.



Сегментна организация на паметта

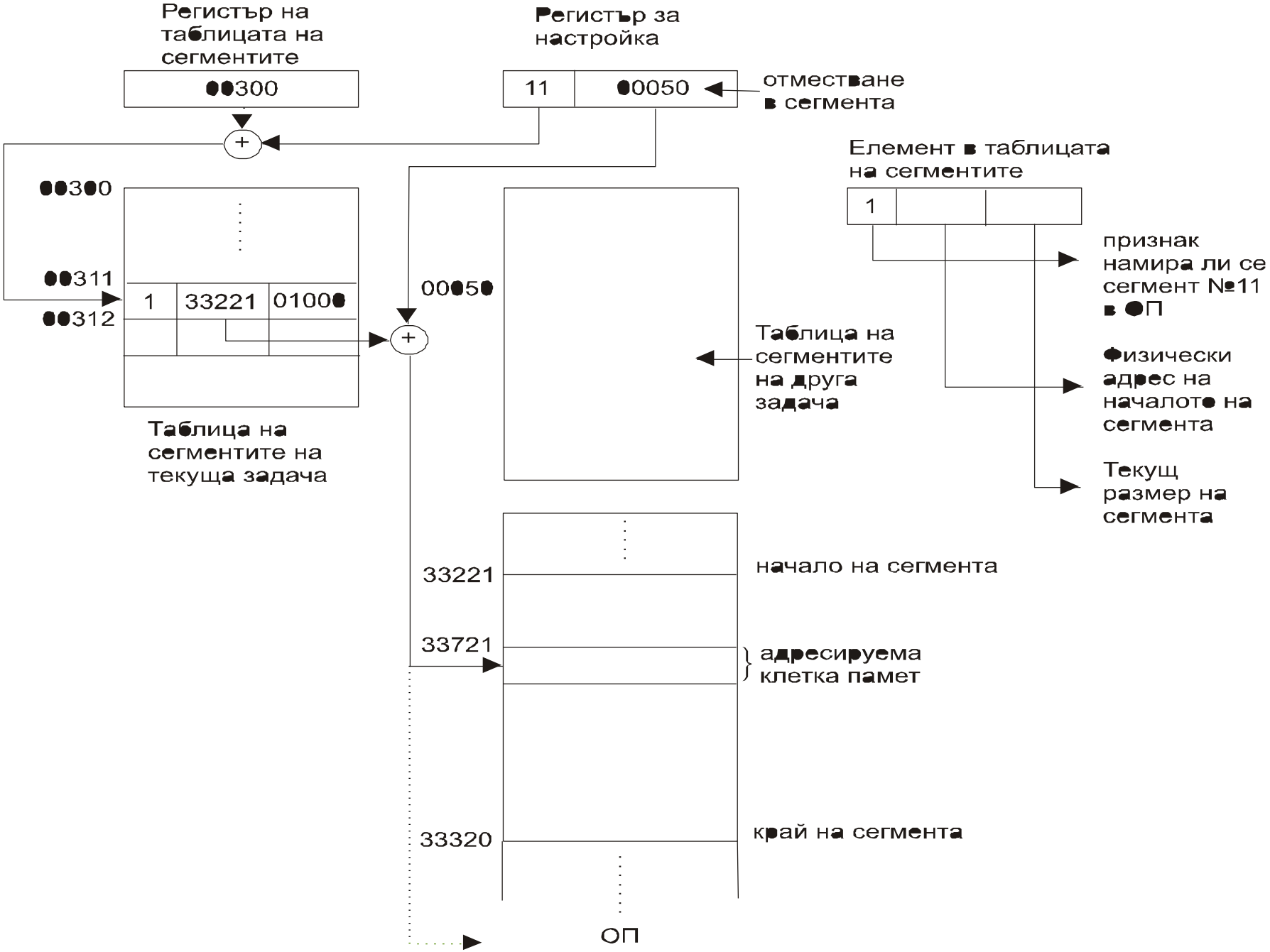
Реализацията на сегментирането предполага, че за всеки процес се формира таблица на сегментите.

Регистър на таблица на сегментите указва началото на таблицата за текущия процес.

Виртуалният адрес има вид (s, d), където:

s е номер на сегмента

d е отместване по отношение началото на сегмента.



Формиране на изпълнителния адрес при сегментиране

Ако сегментът липсва в паметта, се генерира прекъсване, при което се извиква програмата за зареждане на сегмент. Това е управляваща програма, която има следните функции:

1. Зареждане на сегмента от външната в изпълнимата памет и обратно – освобождаване на памет;
2. Превключване на процесора от един процес към друг.

Странична организация

Друг начин на разпределяне на паметта е странична организация, която изключва външната фрагментация посредством разбиването на паметта на единици с еднакъв размер и разпределя тези единици. Виртуално адресно пространство се дели на страници с фиксиран размер. Обръщането към паметта има вид

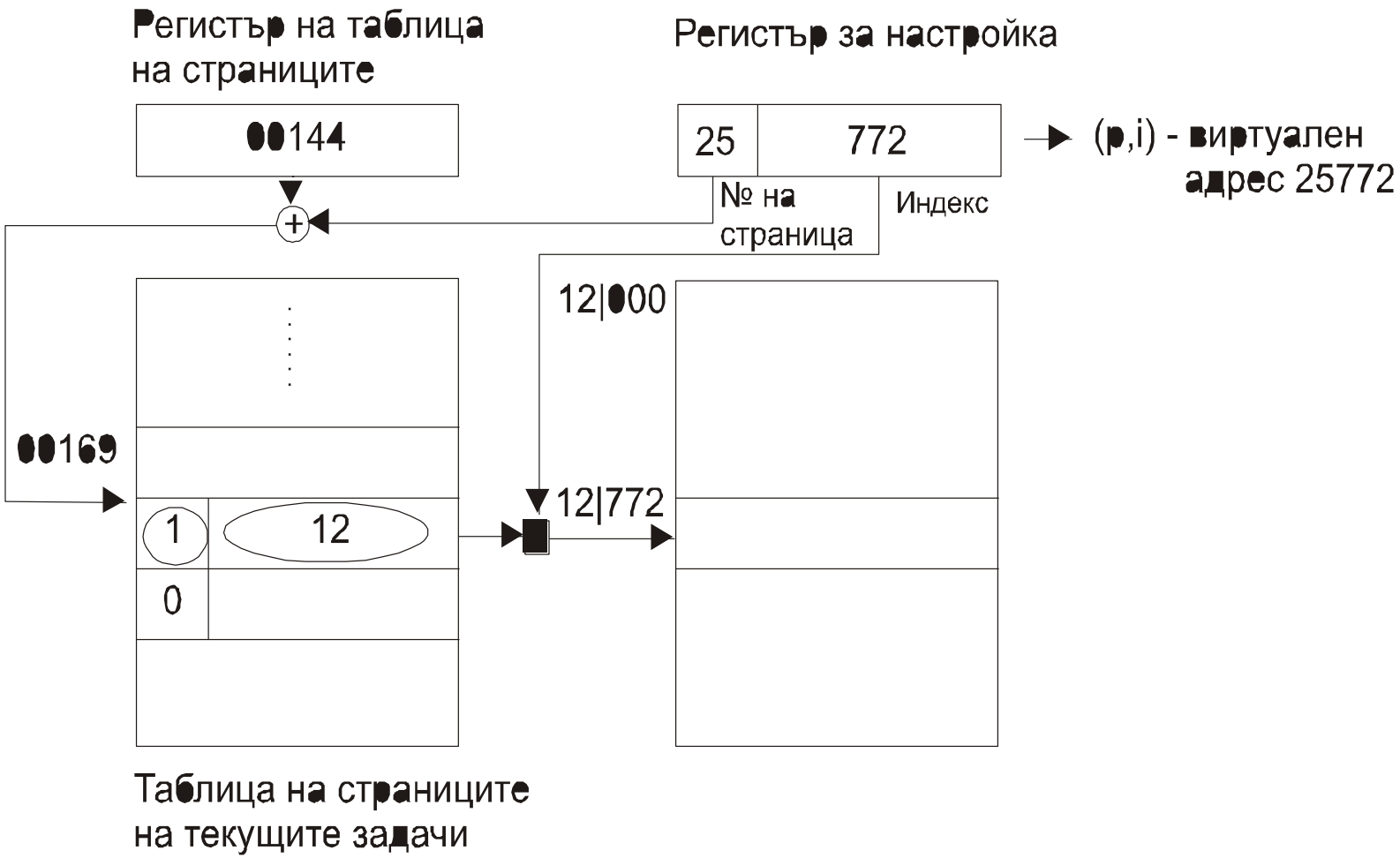
(p, i), където

p – номер на страницата;

i – отместване относно началото на страницата (индекс).

Физическата памет се разбива на единици с еднакъв размер, които се наричат кадри. Броят на физическите страници (кадри) зависи от размера на паметта и избора на размер за самата страница. Размерът на виртуалните страници и размерът на кадрите е еднакъв за дадена компютърна система – всъщност, размерът на виртуалните страници на процесите се определя от размера на кадрите. Броят на виртуалните страници на процесите в една система може и обикновено превишава броя на физическите.

Съответствието между виртуалните и физически страници на процес се определя по неговата таблица на страниците. За всяка страница има елемент, състоящ се поне от две полета: номер на кадъра и индикация намира ли се страницата в паметта.

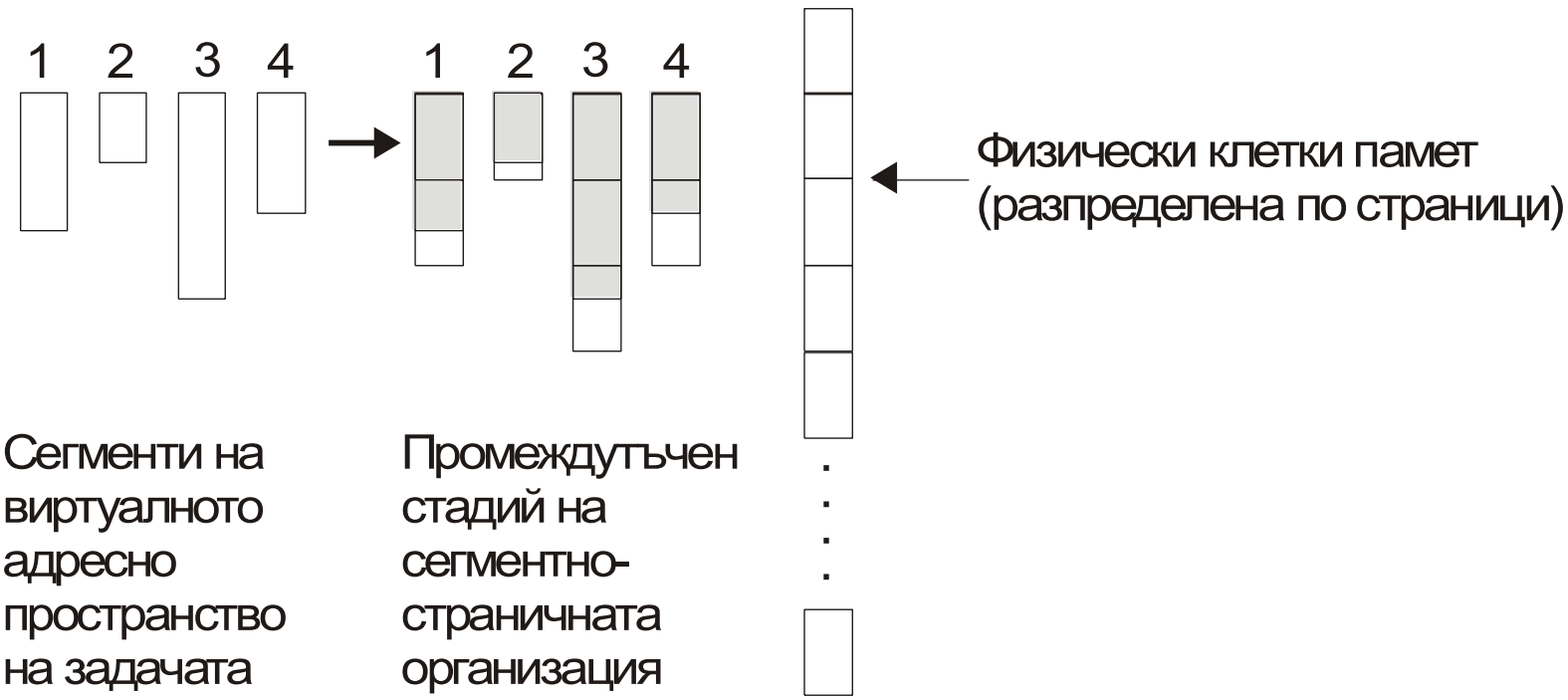


Формиране на изпълнителния адрес при страниране

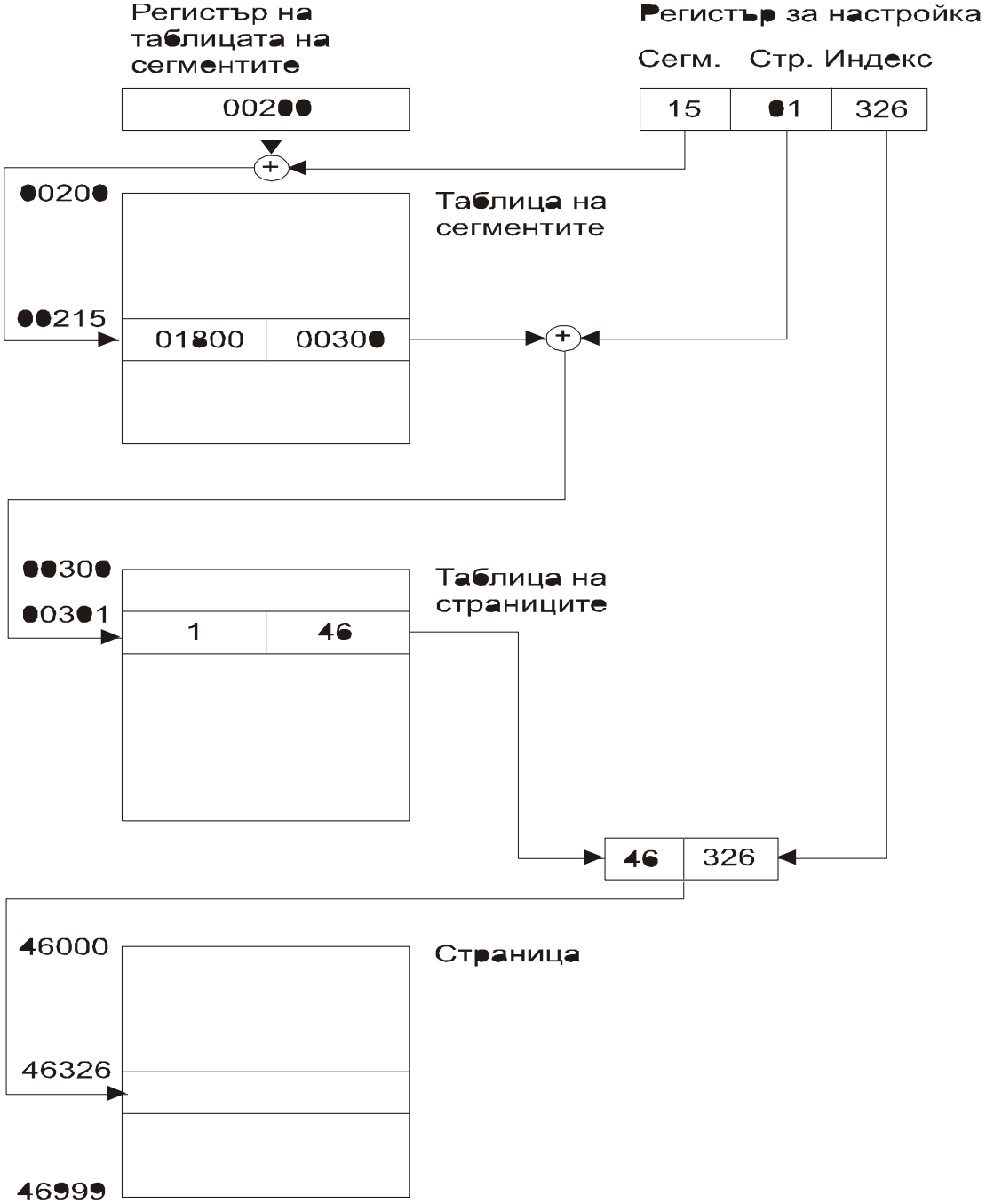
Сегментно-странична организация

За намаляване на фрагментацията може да се реализира сегментиране с използването на странична организация.

Във виртуалния адрес (s, d) отместването d се разглежда като двойка (p, i) – страница и индекс. Тогава виртуалният адрес (s, d) е (s, p, i).



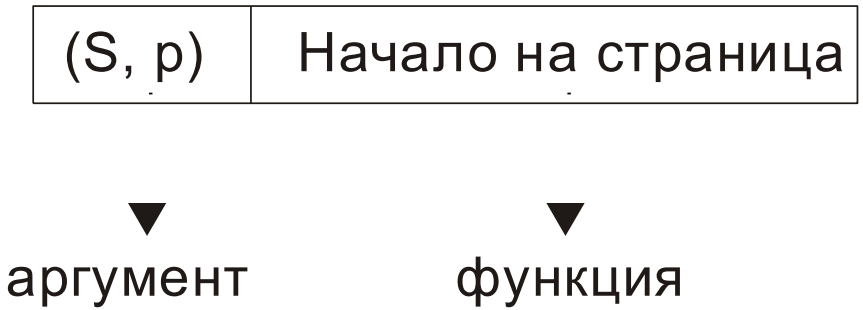
Странирани сегменти



Получаване на изпълнителния адрес при странирани сегменти

Асоциативна памет

Стандартно решение за ускоряване изчисляването на изпълнителния адрес е използването на асоциативна памет. Използва се малък високоскоростен кеш, наричан асоциативни регистри. Комплект асоциативни регистри се реализира хардуерно, от високоскоростна памет. Всеки такъв регистър се състои от две части: значение на аргумент и значение на функция, всяко от които е с определена дължина. На асоциативната памет се подава значение, наречено ключ. Паралелно се сравняват значенията на всички аргументи със значението на ключа. При съответствие с даден аргумент като резултат от търсенето се подава значението на функцията. Търсенето е изключително бързо, но такава памет е много скъпа. Броят асоциативни регистри може да е от 8 до 2048.



Асоциативен регистър

Използването на асоциативните регистри става по следния начин: в тях се съдържа само малка част от елементите на таблицата на страниците. Когато процесорът генерира логически адрес, номерът на страницата, а при странирани сегменти двойката (s,p), се подава като ключ към асоциативните регистри. При открито съответствие значението на функцията в асоциативния регистър е номера на кадъра във физическата памет и веднага се използва за достъп до паметта. При всяко избиране на нов процес (нова таблица на страниците) всички асоциативни регистри трябва да бъдат “изчистени”, за да няма грешно използване на страници от другия процес.

Важен показател при използването на асоциативна памет е процентът на попаденията. 80% попадения означава, че от 100 търсения в 80 случая търсеното съответствие на ключ и аргумент е било налице. Ако времето за достъп до асоциативните регистри и получаване адреса на кадъра е 10 ns, а 50 ns е достъпа до паметта, времето за достъп до искания адрес е 60 ns. Ако не е налице съвпадение на ключ и аргумент (отнема 10 ns), необходим е достъп до таблицата страниците и номер на кадъра (50 ns), след което достъп до търсения адрес (50 ns), ако системата е с чисто страниране. Общото време за достъп е 110 ns

При този процент на попаденията средното ефективно време за достъп с отчитане темпото на възможните случаи е:

ефективно време за достъп $= 0,80*60+0,20*110 =$ ***70 ns***

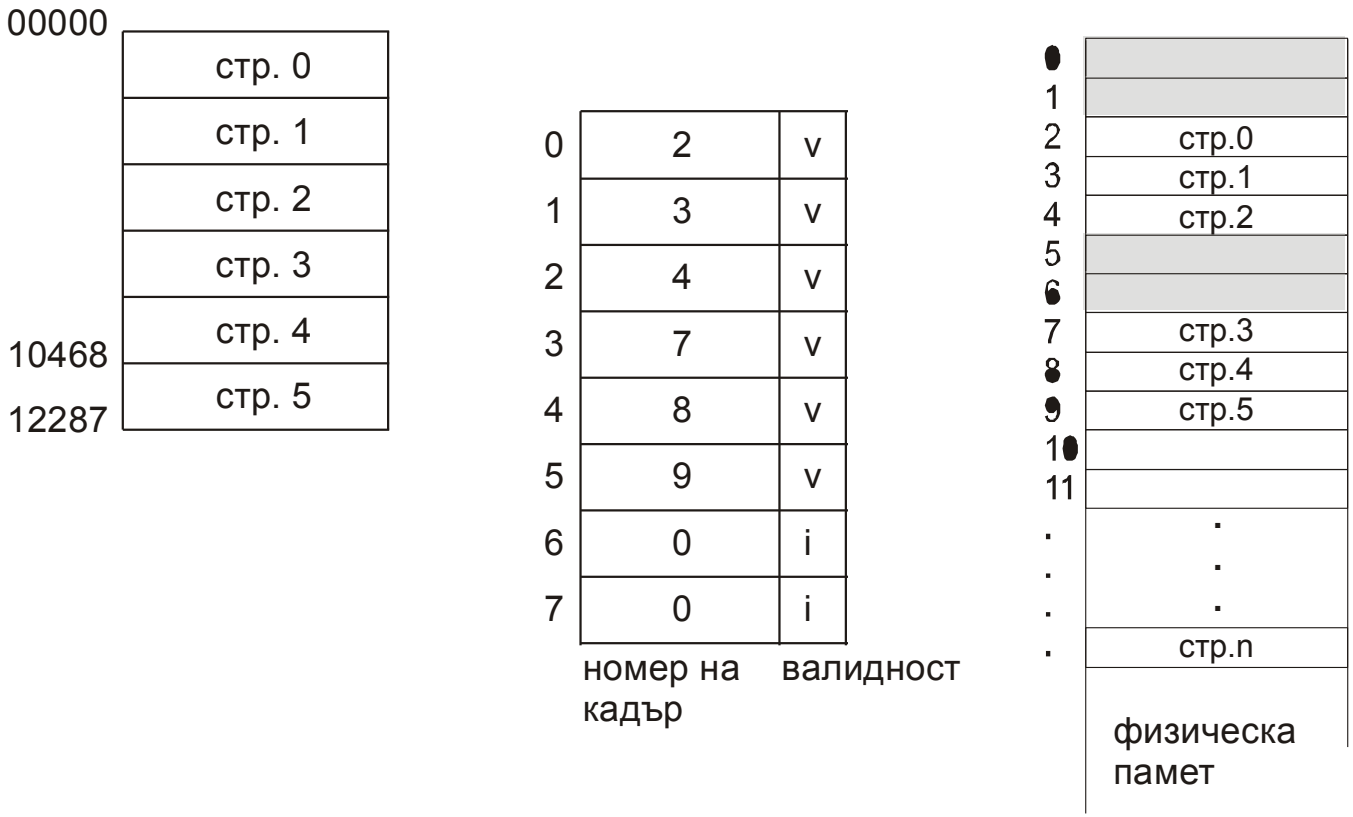
При процент на успелите попадения 98 ефективното време за достъп е

$0,98*60+0,02*70 = 61\ ns$

Защита

Защитата в странирана система се реализира посредством свързване на бит с всеки кадър. Обикновено битовете за защита се намират в таблицата на страниците. Един бит може да определи дали една страница е само за четене или може да се чете и записва в нея. Всеки достъп до паметта минава през таблицата на страниците. Едновременно с пресмятането на физическия адрес се проверява бита за защита и ако той има значение значение “само четене” не се допуска записване по този адрес. Опит за запис при забрана на записа предизвиква хардуерно изпълнение на trap-инструкция към ОС (нарушение защитата на паметта).

Освен битовете за режим на достъп с всеки вход в таблицата на страниците се свързва т.нар. бит за валидност. Ако е установен в значение “валидно”, дадената страница се намира в логическото адресно пространство на процеса, и затова тази страница е “валидна”. В противен случай страницата е извън логическото адресно пространство на процеса. Например, в система с 14 битово адресно пространство (адреси от 0 до 16383) може да има процес, на който е разрешено да ползва адресите само от 0 до 10468. При размер на страницата 2К е възможна следната ситуация:



Защита в странирана система

1. Какво е “външна фрагментация” на паметта/
2. Кои полета от машинна инструкция се наричат абсолютни? А относителни?
3. Какво означава “непрекъснато разпределяне на паметта”?
4. Колко адреса генерира процесор с 32 битова адресна шина?
5. Какъв обем памет адресира процесор с 32-битова адресна шина, ако се адресира двубайтова дума?

http://msdn.microsoft.com/en-us/library/aa366912(v=vs.85).aspx

6. Обяснете разликата между външна и вътрешна фрагментация.
7. Ако логическо адресно пространство от осем страници, всяка по 1024 думи, се проектира върху физическа памет от 32 кадъра:

   а) колко бита има логическият адрес?

   б) колко бита има физическият адрес?
8. С каква цел се използва асоциативна памет?

Лекция 6
Виртуална памет

Виртуалната памет е техника, която позволява изпълнението на процеси, които може да не са напълно в паметта. Главното достойнство на тази схема е, че програмата може да е по-голяма от физическата памет. Освен това представя главната памет като изключително голям, еднороден масив, разделяйки логическата от физическата памет. Тази техника освобождава програмиста от грижи за ограничения обем на паметта. Реализацията на виртуална памет не е проста и може съществено да снижи ефективността на ИС, ако не е добре разработена.

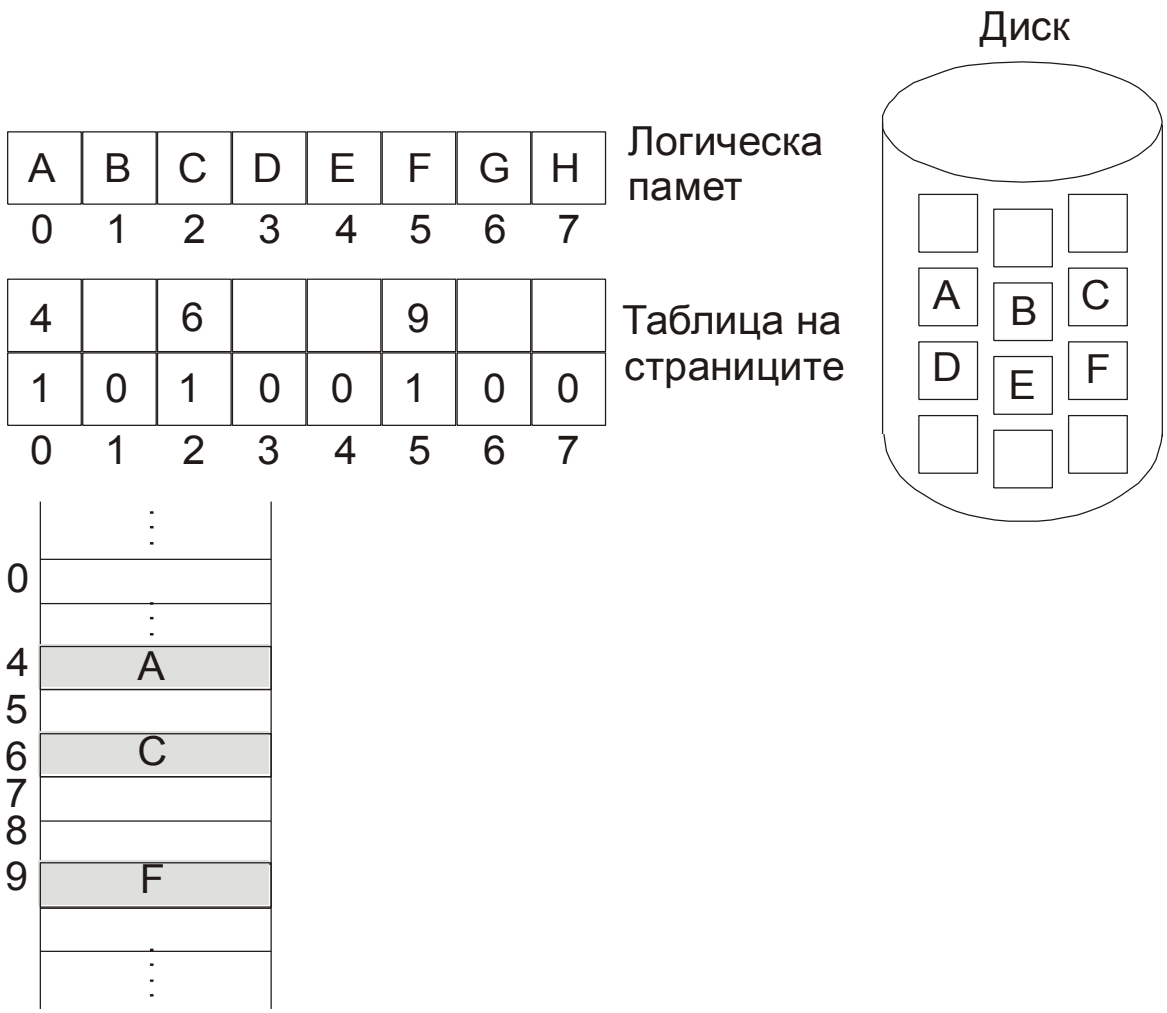
Възможността да се изпълняват програми, намиращи се само частично в паметта има следните достойнства:

- размерът на програмата вече не се ограничава от наличната физическа памет;
- тъй като всяка програма може да заема вече по-малко памет, може да се увеличи коефициента на мултипрограмиране, като съответно се увеличи ефективността на използване на процесора;

Виртуална памет със страниране

Системата, поддържаща виртуална памет със страниране, е много сходна със система, използваща страниране със суопинг. Процесите се намират във вторична памет. Когато един процес трябва да се изпълни, бива зареден (swapped in) в паметта. Само че, вместо да се зареди целият процес, се използва т. нар. lazy swapper (ленив суопинг), т.е. една страница никога не се зарежда в паметта до момента на необходимост от нея. Тъй като процесът се разглежда като редица от страници, по-правилно е да се каже pager вместо swapper.

Четенето в паметта само на страници, които се използват, позволява да се намали времето за суопинг и обема на нужната физическа памет.



Пример за задача, от която само три страници са в паметта

При опит за използване на страница, която не е в паметта, се изпълнява следната процедура:

1. Проверява се таблицата на страниците (обикновено съхранявана в блока за управление на процеса) за определяне дали заявката е валидна или невалидна;
2. Ако обръщането е към страница, която не е в паметта или е извън логическо пространство на процеса, процесът прекъсва изпълнението си. Ако страницата е в логическото му пространство, но не е в паметта, тя се зарежда:
3. Намира се празен кадър;
4. Чете се страницата в новия определен кадър.
5. След като четенето е завършило, модифицират се таблицата на процеса и таблицата на страниците;
6. Рестартира се инструкцията от мястото на прекъсване. Процесът вече има достъп до нужната страница.

Теоретично е възможно една и съща инструкция да изисква обръщане към повече от една страници, които не са в паметта. (Например едно обръщане за инструкция и няколко различни страници за данни от същата инструкция). Такава ситуация резултира в неприемливо време за обработка от страна на системата. Анализът на много програми показва, че такова поведение е изключително рядко. Програмите имат свойството локалност на обръщанията към паметта, като резултатът е приемлива ефективност на странирането със заместване.

Хардуерната поддръжка на страниране със заместване е същото като при страниране със суопинг:

- Таблица на страниците – в тази таблица може да се отбелязват невалидните входове чрез бит за валидност или специални битове за защита;
- Вторична памет: в нея се съхраняват страниците, които не се намират в главната памет. Това обикновено е диск. Известна е под името суоп-устройство, а секцията от диска, използвана за тази цел е известна като суоп – пространство или backing store.

Ето пример за най-лошия случай. Ако имаме три-адресна инструкция от вида: ADD A, B, C: събери съдържимото на A със съдържимото на B и резултата помести в C. Изпълняват се следните стъпки:

1. Извлечи и декодирай инструкцията (ADD);
2. Извлечи съдържимото на A;
3. Извлечи съдържимото на B;
4. Събери [A] и [B]
5. Запиши резултата в C.

Ако ситуацията липса на страница е възникнала на стъпка 5, при записване в C (тъй като C е в страница, която не е в паметта), необходимо е да се зареди тази страница, да се коригира таблицата на страниците и да се рестартира инструкцията. Това означава, че стъпки 1 – 4 трябва да се изпълнят отново.

IBM System 360/370 MVC има инструкция, която може да премести 256 байта от един адрес в друг адрес (move character instruction). Ако кой да е от блоковете – източник или приемник, започва от една страница и завършва на друга, може да възникне ситуацията липса на страница след частично извършване на преместването. В допълнение, ако блокът източник и блокът приемник се припокриват, източникът в момента на възникване на ситуацията липса на страница може вече да е модифициран и в този случай не можем просто да рестартираме инструкцията.

Този проблем може да бъде разрешен по два различни начина. При първия чрез микрокода да се изчислят крайните адреси на двата блока и по този начин да се установи дали ще настъпи ситуацията липса на страница. Това става предварително, преди модифицирането на каквото и да е. Преместването може да се извърши в случай, че няма липса на страница. Друго решение е да се използват временни регистри, в които да се пазят значенията от презаписаните (модифицирани) клетки. При възникване на липса на страница значенията от тях се записват обратно в паметта, като по този начин се възстановява състоянието на паметта както е било до момента на стартиране на инструкцията, следователно тя може да бъде стартирана отново.

Изпълнение на страниране със заместване

Заместването на страници по заявка може да окаже съществено влияние върху ефективността на компютърната система. За да се оцени това влияние, се изчислява ефективното време на достъп. За повечето компютърни системи времето за достъп до паметта (**m**emory **a**ccess - ma) варира между 10 и 200 ns. Ако не възниква липса на страница, ефективното време за достъп е равно на времето за достъп до паметта. Ако настъпи липса на страница, първо от диска се прочита необходимата страница, след което се извършва достъп до съответния адрес.

Нека *p* е вероятността за липса на страница,

$$0 \leq p \leq 1$$

Ефективното време за достъп е ema, обработката на липси на страница – pft (page fault time). Тогава

*ema = (1 - p) * ma + p * pft*

При възникване на такава е необходимо да се изпълнят следните стъпки:

1. Trap (обръщане) към ОС;
2. Съхраняване на регистрите и състоянието на процеса;
3. Определяне на причината за прекъсването;
4. Определяне дали указваната страница е в рамките на логическото адресно пространство на процеса и определяне местоположението й на диска;
5. Четене на страницата от диска в свободен кадър:
    а) очакване в опашка към това устройство, докато четенето се изпълни;
    б) очакване за намиране мястото й върху устройството;
    с) стартиране на четенето на страницата в свободен кадър.
6. Докато очакваме зареждането на страницата, CPU да се предостави на друг потребител (процес);
7. Прекъсване от диска (I / O е изпълнен);
8. Ако е изпълнена стъпка 6, да се съхранят регистрите и състоянието на другия процес;
9. Определяне причината за прекъсване по стъпка 7;
10. Да се коригира таблицата на страниците и други таблици с цел да се покаже че търсената страница е вече в паметта;
11. Очакване CPU да бъде отново предоставено на този процес;
12. Да се възстанови контекста на процеса и при новото състояние на неговата таблица на страниците да се възобнови прекъснатата инструкция.

Не във всички случаи е необходимо тези стъпки да се изпълняват.

При всички случаи обаче съществуват три главни компонента при обслужването на ситуацията липса на страница:

1.Обслужване на прекъсването по тази причина;
2. Четене на страницата в паметта;
3. Възобновяване на процеса.

Ако вземем средното време за обработка на липсата на страница pft да е 25 милисекунди (pft = 25 ms), а времето за достъп до паметта да е 100 ns (наносекунди) (ma = 100 ns) то ефективното време за достъп в наносекунди е:

$$ema = (1-p)*100+p*25000000=100+24999900*p$$

Вижда се, че ефективното време за достъп много силно зависи от вероятността за липса на страница. Ако в един от 1000 случая имаме липса на страница, ефективното време на достъп е 25 μs (микросекунди). Компютърът се забавя 250 пъти поради обработката на тази ситуация. (*25 μs/100 ns = 250*). Ако искаме деградацията да е по-малка от 10 %, необходимо е

*110>100+25000000*p*
*10>25000000*p*
p<0,0000004

Тоест, за да се поддържа забавянето на компютъра в разумни граници, от 250 000 достъпа до паметта само един може да предизвиква липса на страница.

Алгоритми за заместване на страници

Ако имаме 4 процеса и 40 свободни кадъра, и всеки от процесите има 10 страници, но използва само 5 от тях, възможно е да се увеличи степента на мултипрограмиране, например да се увеличи броят на процесите на 8. Тогава се говори за свръх разпределяне на паметта. Ако се изпълняват 6 процеса, всеки от които има 10 страници, но използва само 5, се увеличава степента на използване на процесора, производителността на системата и в добавка има 10 свободни кадъра. Ако обаче всеки от тези процеси, при специфичен набор от данни, внезапно “пожелае” да използва всичките си 10 страници, възниква необходимост от 60 кадъра при налични 40.

Заместването на страница включва следните действия:

1. Намиране на търсената страница върху диска;
2. Намиране на свободен кадър
   а) ако такъв има, четене на страницата в него;
   б) ако няма свободен кадър, използване на алгоритъм за избор на кадър за заместване (“жертва”).
   с) записване на заместваната страница (“жертвата”) на диска и промяна на таблицата на страниците и кадрите;
3. Четене на нужната страница от диска в новоосвободения кадър; модифициране таблиците на страници и кадри;
4. Възобновяване на процеса.

При липса на свободни кадри са нужни два трансфера на страници – един към диска и един от диска към паметта. Тази ситуация удвоява времето за обслужване на ситуацията липса на страница. Този ефект може частично да се неутрализира с използването на т.нар. бит за модифициране (dirty bit). За всяка страница или кадър може да има бит, реализиран хардуерно, който да дава индикация, че страницата е модифицирана след последното й четене в паметта. Ако дадена страница е избрана за заместване и не е била модифицирана, не е необходимо да бъде записвана на диска.

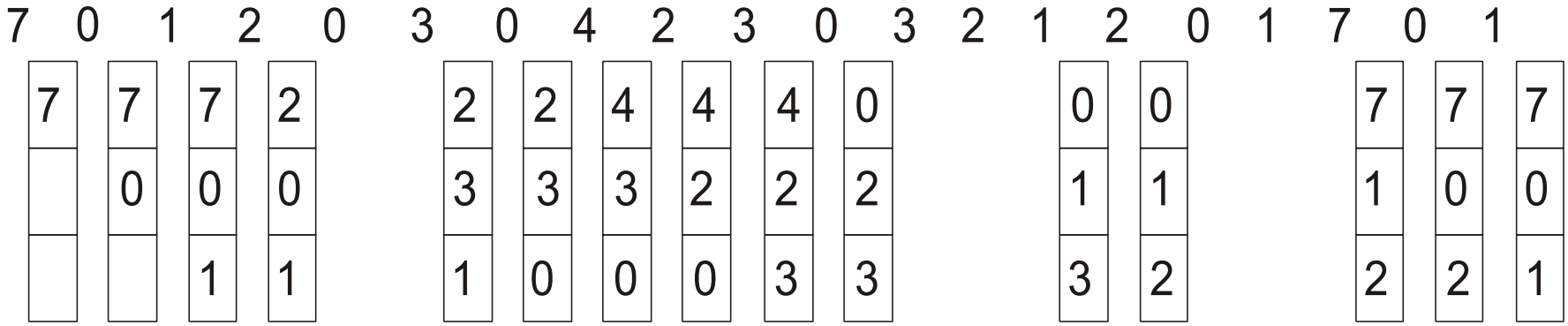
- Обикновено при избирането на алгоритъм за заместване на страница критерият е най-ниска степен на възникване липса на страница.
- За оценяване на различните алгоритми се използва една и съща редица от адреси в паметта. За даден размер на страницата е необходимо да се взема под внимание само номерът на страницата, а не целият адрес. За илюстрация на алгоритмите за заместване на страница ще използваме следната редица:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

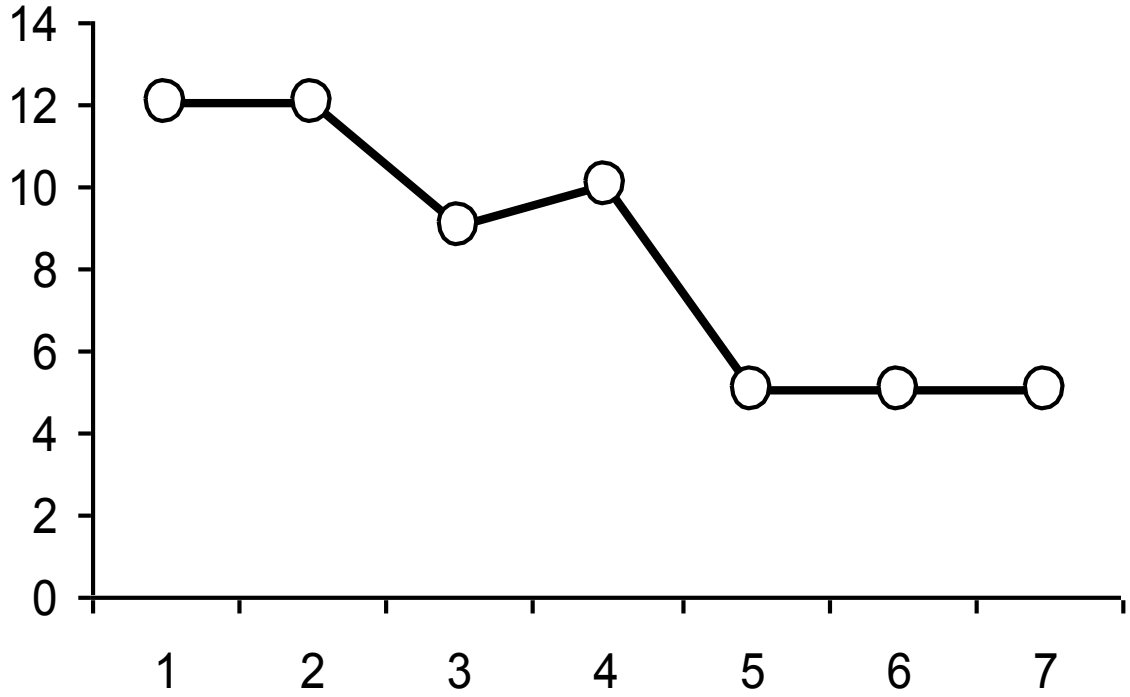
- При увеличаване броя на свободни кадри нивото на броя липси на страница пада до някаква минимална стойност.

Заместване по FIFO

Когато възникне необходимост една страница да бъде заместена, избира се тази, която най-дълго се е намирала в паметта. За страниците в паметта се поддържа FIFO опашка.



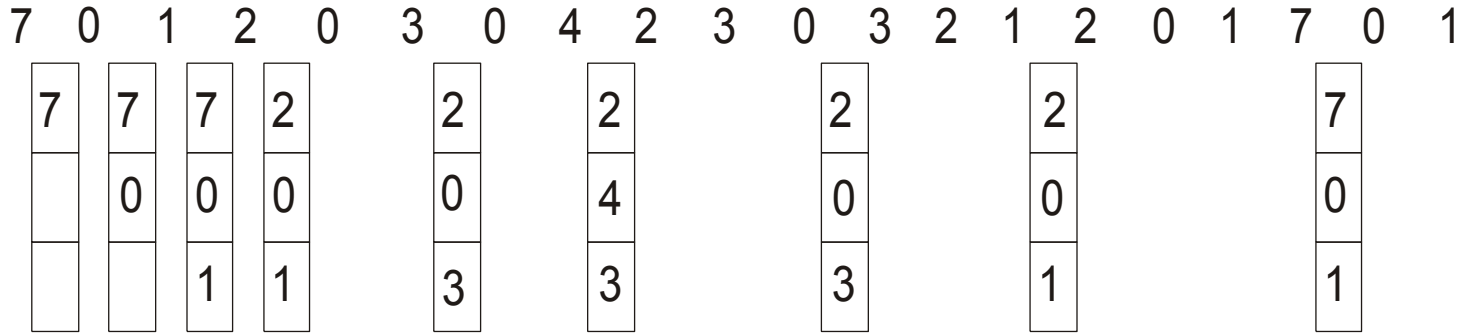
Заместване на страница по алгоритъма FIFO



Крива на зависимостта на броя липси на страници в зависимист от броя предоставени кадри за алгоритъма FIFO

Оптимално заместване на страници

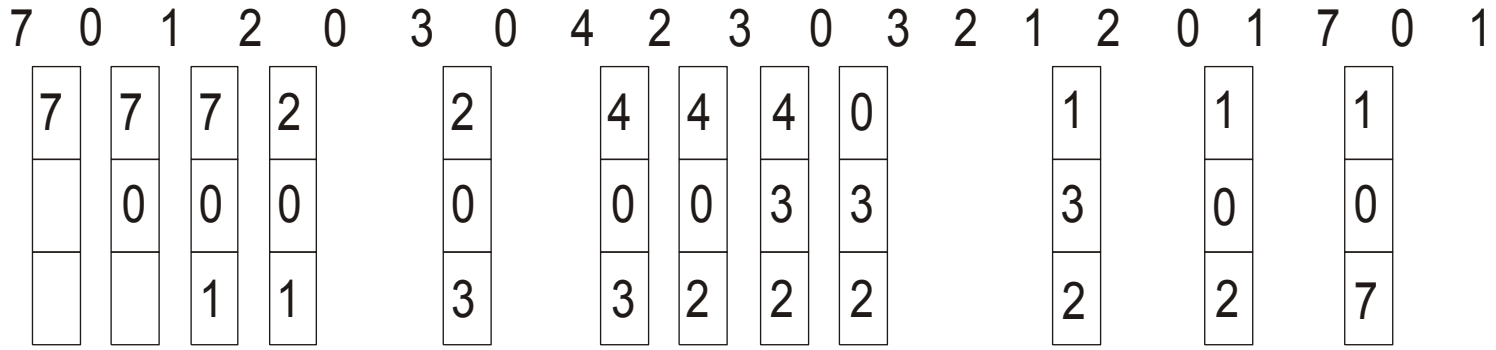
Оптималният алгоритъм за заместване на страница не показва аномалията на Белъди. В общи линии неговата стратегия е следната: заместване на страницата, която няма да бъде използвана в течение на най-дълъг период от време. Използването на този алгоритъм гарантира най-ниска честота на липсата на страници за определен брой кадри.



Заместване на страници по оптимален алгоритъм

Заместване LRU (Least Recently Used)

Един компромис, основаващ се на двата предишни способа, е алгоритъмът LRU: ако се използва близкото минало за прогноза за близкото бъдеще, може да се твърди, че с голяма вероятност страниците, които най-малко са били използвани в миналото, най-малко ще бъдат използвани и в бъдещето. Стратегията на алгоритъма LRU – Least Recently Used (най-малко използвана в последно време) замества страницата, която не е била използвана в течение на най-дълъг период от време. Този алгоритъм свързва всяка страница с времето на последното ѝ използване.



Заместване на страници по LRU

Често се предлага поддръжка във формата на т. нар. reference bit. На този бит се дава стойност единица от когато има обръщане (четене или запис) към съответната страница. Всеки елемент в таблицата на страниците съдържа такъв бит. Първоначално всички битове са изчистени (нулирани) от ОС. При изпълнение на процес за всяка страница, извикана в паметта, съответният бит става 1. След определено време системата може да определи кои страници са били използвани, без да определя реда на извикването им. Използването на тази информация се реализира в много алгоритми за заместване на страници, апроксимиращи LRU - заместването.

Алгоритми, използващи броене (LFU, MFU)

Друга група алгоритми за избор на страница, която трябва да бъде заместена, се базира на честотата на използване на страниците. За всяка страница се отчита броя обръщания към нея. Две от възможните схеми са следните:

1. Замества се страницата, към която броят обръщания е най-малък – LFU (Least frequently used). Предполага се, че и за в бъдеще обръщанията към тази страница ще са малко. Този алгоритъм не отчита възможността например в началната фаза на даден процес една страница да се използва много често, а впоследствие към нея да няма нито едно обръщане. Тогава тази страница “засяда” в паметта, тъй като броячът й има голяма стойност, въпреки че от даден момент нататък не се използва изобщо.

2. Алгоритъмът, при който се замества най често използваната страница /MFU – most frequently used/ се базира на предположението, че страницата с най-малко значение на брояча току-що е била заредена в паметта и използването й още предстои.

Нито един от горните два алгоритъма не се среща често. Реализацията им е скъпа и не апроксимират достатъчно добре оптималния алгоритъм.

Алгоритъм с буфериране на страници

Като допълнение към типичните алгоритми за заместване на страница се използват и реализации на други идеи. Често системите постоянно поддържат пул от свободни кадри. При липса на страница, както обикновено, се избира страница, която ще бъде заместена, но преди да се извърши записването на тази страница във вторичната памет, нужната страница се прочита в един от кадрите на гореспоменатия пул. Това решение икономисва време. Когато заместваната страница се запише във вторичната памет, нейният кадър се добавя към пула свободни кадри.

Разпределяне на кадрите

Най-простото разпределяне на виртуалната памет е при наличието на един процес. При наличието на памет NK, където страницата е с размер 1К имаме N на брой кадри. Ако M от тях (M < N) са заети от ОС, остават N-M за потребителските процеси. Първите N – M страници на процеса (при еднопотребителска система) ще бъдат заредени успешно. При изчерпване на свободните кадри започва да работи алгоритъм за заместване на страници, който ще избере един от (N-M)-те кадри за заместване, и т.н. При спиране или завършване на процеса всички кадри се освобождават. Тази проста стратегия има множество варианти.

Разпределяне на кадрите

При буфериране част от (N-M)-те кадъра (например 3 на брой) могат да се поддържат винаги свободни. Тогава при липса на страница необходимата се зарежда в един от тези три свободни кадри, процесът продължава, а ОС избира един от останалите (N-M-3) кадъра, който да бъде освободен и добавен към останалите след зареждането 2 свободни кадъра.

Задачата за разпределяне на кадрите се усложнява при мултипрограмните системи.

1. Първото ограничение се определя от броя на наличните кадри.
2. Съществува минимален брой кадри, които могат да бъдат предоставени на процес. Този минимален брой се определя от архитектурата на компютъра, по-точно от системата машинни инструкции. При възникване на липса на страница преди да е завършило изпълнението на инструкция, тя трябва да спре и започне изпълнение отначало.

Например за едноадресна машинна инструкция са необходими два кадъра: един за самата инструкция и един за страницата, в която е адресът. Ако в този случай се допуска косвена адресация с едно ниво на вложеност, необходим е още един кадър: например load-in – инструкция, намираща се на страница с номер 18 може да съдържа адрес на страница 7, който да съдържа адрес, намиращ се на страница 35. Изводът е, че за машина, работеща само с едноадресни инструкции и косвена адресация на едно ниво на процесите трябва да се предоставят минимум по три кадъра. Ако нивото на вложеност на косвената адресация не е ограничено, теоретично е възможно адрес от инструкция да съдържа адрес, който да съдържа адрес и т.н., докато се извърши достъп до всяка страница от виртуалната памет. (В най-лошия случай цялата виртуална памет трябва да се намира във физическата, което обезсмисля идеята за виртуална памет. За избягване на този случай се налагат ограничения на нивото на вложеност на адресите при косвена адресация.)

Минималният брой кадри, които могат да бъдат предоставени на процес, зависи от архитектурата на компютъра, а максималният – от броя кадри на физическата памет.

Алгоритми за разпределяне на кадри

1. Най-простият начин да бъдат разпределени N кадъра между M процеса е като се предостави равен брой кадри на всички: по N/M. Такава схема се нарича равно разпределяне.
2. Друг алгоритъм е пропорционалното разпределяне. Базира се на получаване на необходимостта на процесите от памет. Според размера на процесите им се предоставя различен брой кадри.

Ако за процеса P_i са необходими s_i виртуални страници, то за всички процеси са необходими V_s виртуални страници:

$$V_s = \sum_i s_i$$

Ако броят на всички кадри е N, то за i-тия процес се предоставят f_i кадри, където f_i е приблизително

$$f_i = \frac{s_i}{V_s} N$$

f_i трябва да е цяло число, като $f_i > f_{min}$, където f_{min} е минималният брой кадри, определян от системата инструкции на машината. При това е необходимо да се съблюдава условието

$$\sum_i f_i \leq N$$

И в двата гореспоменати случая разпределянето на кадрите зависи от коефициента на мултипрограмиране. При увеличаване на коефициента на мултипрограмиране от всеки процес се отнемат няколко кадъра, които ще бъдат предоставени на новия (новите) процеси.

- Вижда се, че при тези способи на разпределяне високоприоритетните и нископриоритетните процеси се обслужват еднакво. При пропорционалното разпределяне е възможна модификация, според която броят предоставени кадри да зависи не от размера, а от приоритетите на процесите.

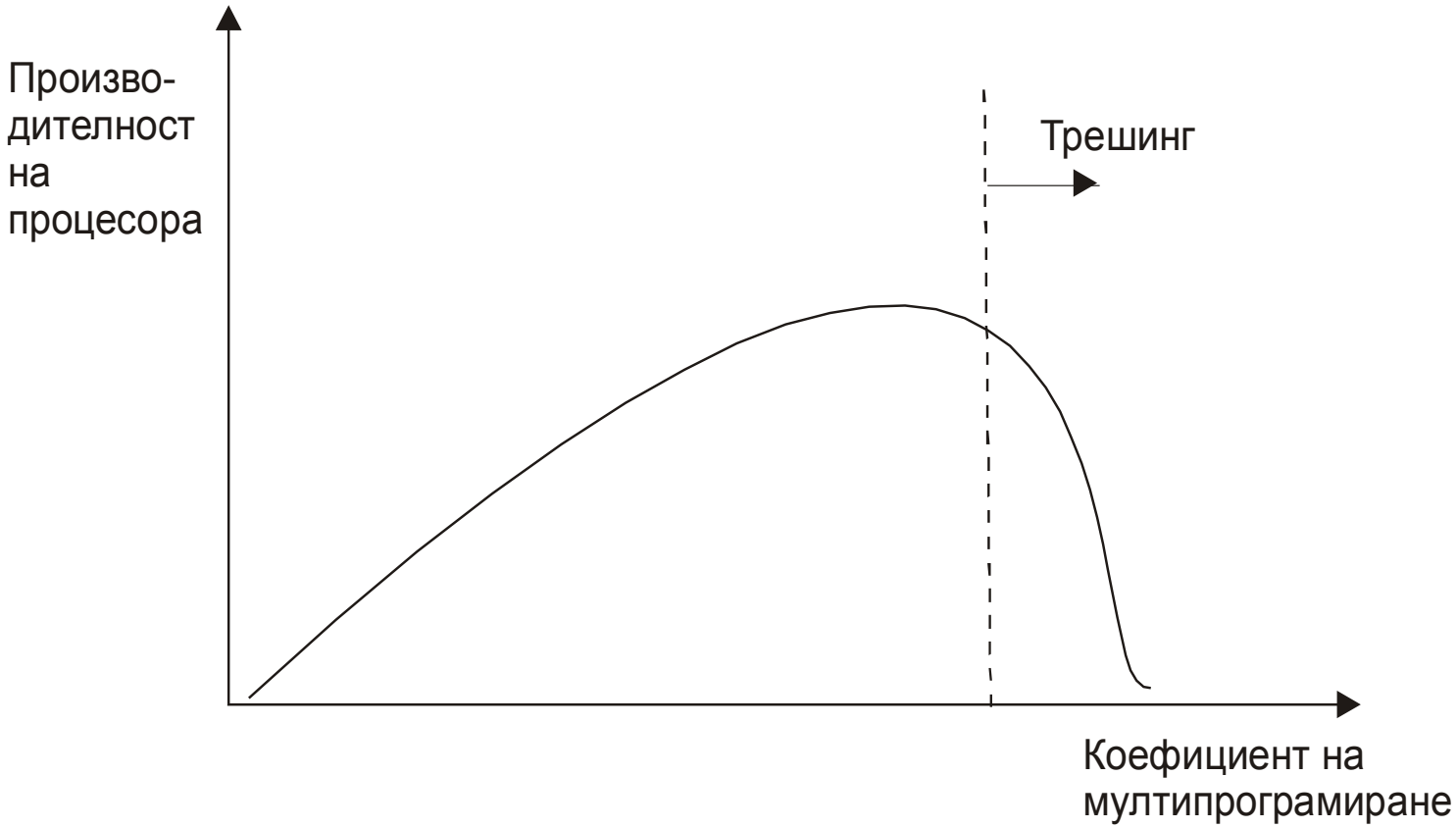
- Алгоритмите за заместване на страници се разделят на две големи групи: локални и глобални. Първите избират страница за заместване от набора кадри, принадлежащи на процеса, предизвикал липса на страница. Глобалните алгоритми избират страница за заместване от всички кадри в системата.

Размер на страницата

Изборът на размер на страницата зависи от множество изисквания, повечето противоречиви. За дадено виртуално адресно пространство колкото по-малък е размерът на страницата, толкова по-голям е броят на страниците и следователно размерът на таблицата на страниците. Виртуална памет с размер 4MB ще има 4 096 страници с размер 1 024 байта и само 512 страници, ако са с размер 8 192 байта. Тъй като всеки процес трябва да поддържа собствено копие на таблицата на страници, желателно е тя да е по-малка.

Трешинг (боксуване) на паметта

Когато броят кадри, предоставен на даден процес, падне под минимума, определян от архитектурата, този процес трябва да бъде спрян, кадрите, заемани от него – освободени и предоставени на други процеси. Оказва се, че дори когато процесите имат минималния брой кадри, те не са достатъчни за нормалното изпълнение: често възниква ситуацията липса на страница; веднага след заместването на кадър се налага отново да бъде заредена страницата, на която е бил предоставен. Тази ситуация, при която системата е заета предимно с обработка на липсата на страница, се нарича трешинг (thrashing) на паметта.



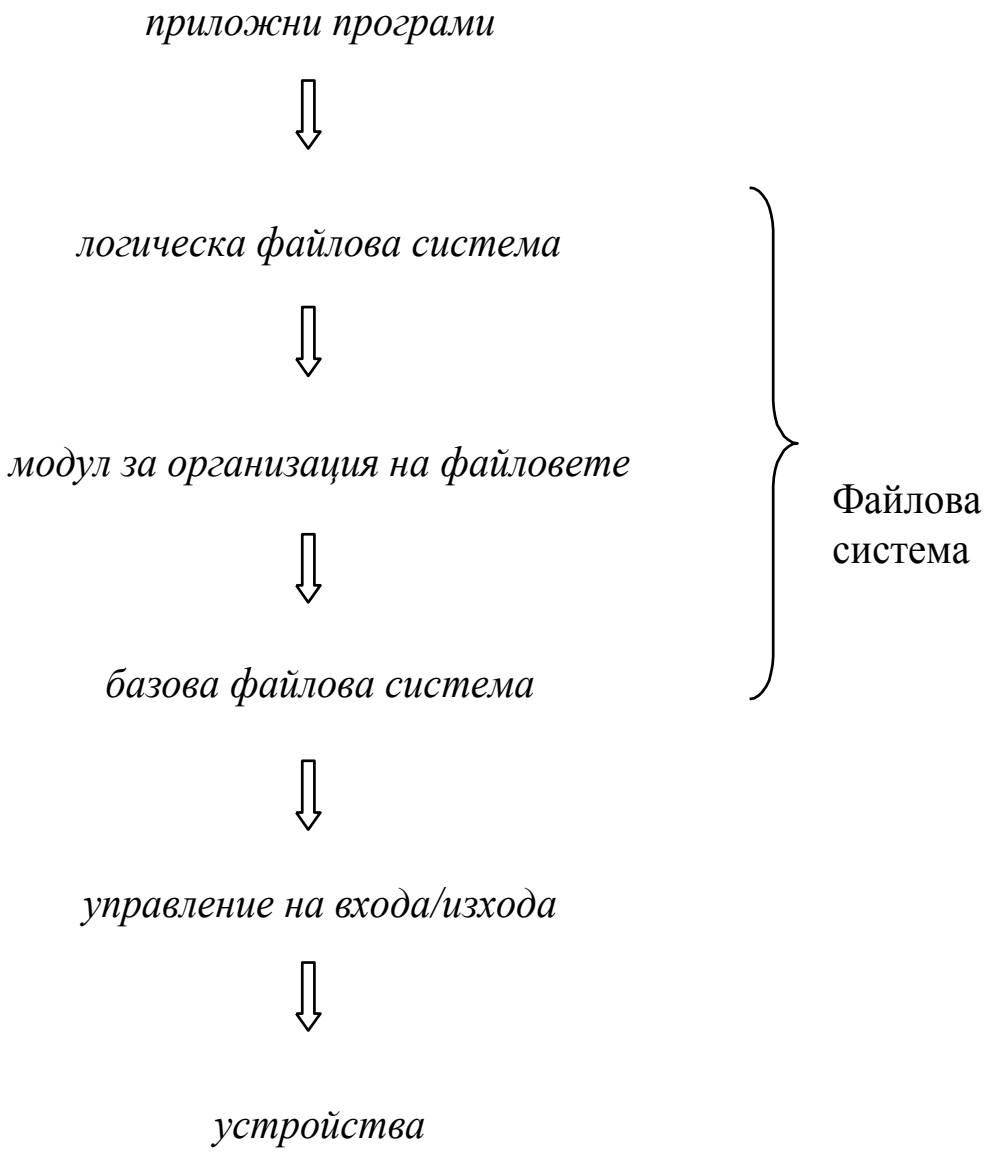
Зависимост на производителността на процесора от коефициента на многозадачност

Лекция 7

Управление на файловете

Обикновено вторичната памет за файловата система се осигурява от дискове. За по-голяма ефективност на входа/изхода трансферът между паметта и диска се извършва на блокове. Когато дискът се върти и блокът магнитни глави е фиксиран в дадено положение, всяка глава описва концентричен пръстен върху повърхността на диска. Тези пръстени се наричат **пътечки** (tracks). Информацията се записва върху тези пътечки на порции, наречени **сектори** (sectors). Пътечки с еднакъв диаметър образуват един **цилиндър** (cylinder). Един **клъстер** (cluster) е група от няколко последователно номерирани сектора (броят им е степен на 2).

Създаването на файлова система поставя две различни задачи. Първата е в определянето на това как потребителят трябва да вижда файловата система. Тази задача включва дефинирането на файл и неговите атрибути, операциите върху файлове и структура на директориите за организиране на файловете. Второ, необходимо е създаването на алгоритми и структури от данни, даващи съответствието (преобразуването) на логическата файлова система върху устройствата, осигуряващи вторичната памет.


приложни програми
логическа файлова система
модул за организация на файловете
базова файлова система
управление на входа/изхода
устройства
Файлова система

Файлове

За удобство е необходимо операционната система да предоставя работа с универсална логическа единица, която се съхранява във външната памет, скривайки от потребителя физическите особености на външните носители. Такава логическа единица се нарича **файл.**

Файлът е именувана съвкупност от смислово обвързана информация, записана във вторичната памет. От гледна точка на потребителя файлът е най-малката единица за работа с външната памет; данните не могат да бъдат записвани във вторичната памет ако не са във файл. Файловете могат да съдържат програми – както на изходния език, така и обектни, и данни. Файловете с данни могат да са числови, символни, символно-числови и двоични.

Освен името, файловете имат следните други атрибути:

- тип – за системите, поддържащи различни типове файлове;
- местоположение – информация за устройството и мястото на файла върху устройството;
- размер – съдържа броя байтове, думи или блокове на файла и възможно максимално допустимия размер;
- защита – съдържа информация за контрол на достъпа към информацията във файла относно кой е оторизиран да я чете, изменя (записва) или изпълнява, ако файлът е изпълним.
- време, дата и идентификация на потребителя – съдържа информация за създаването на файла, последната модификация и последното използване.

Атрибутите на всички файлове се съхраняват в структурата на директориите, която също е във вторичната памет. Атрибутите за всеки файл може да заемат от 16 до повече от 1000 байта.

Всички входно/изходни операции се изпълняват за блокове с размера на физическия запис. Тъй като логическите записи обикновено не съвпадат по размер с физическите, те се блокуват по няколко. (UNIX разглежда файла като поток от байтове. Всеки байт е адресуем чрез отместване спрямо началото или края на файла. В този случай големината на логическия запис е 1 байт.) Файловата система автоматично блокува и разблокува байтовете във физическите блокове за диска.

Размерът на логическия запис, на физическия запис и техниката на блокуване определят колко логически записа съдържа физическият. Блокуването може да се изпълни както от приложение, така и от ОС. Така че файлът може да се разглежда като последователност от блокове. Всички входно/изходни операции се извършват с блокове.

Операции над файлове

Файлът представлява абстрактен тип данни. Всички данни се представят освен чрез множеството си от допустими стойности, така и от операциите над тях.

а) създаване на файл.

б) записване във файл.

в) четене от файл. За четене от файл се извиква системен примитив, на който се съобщава името на файла и мястото в изпълнимата памет, в която ще се извърши четенето, например името на някаква променлива. В каталога се намира входа във файла, а системата поддържа указател към поредния елемент, който трябва да бъде прочетен. След изпълнение на четенето указателят се обновява. Тъй като файловете се отварят или само за запис, или само за четене, повечето системи поддържат само един указател – за текущата позиция.

г) унищожаване на файл. За целта в каталога се намира името на файла и там се изтрива входа към него. По този начин се освобождава цялата заемана от файла памет.

д) отсичане (truncating) на файл.

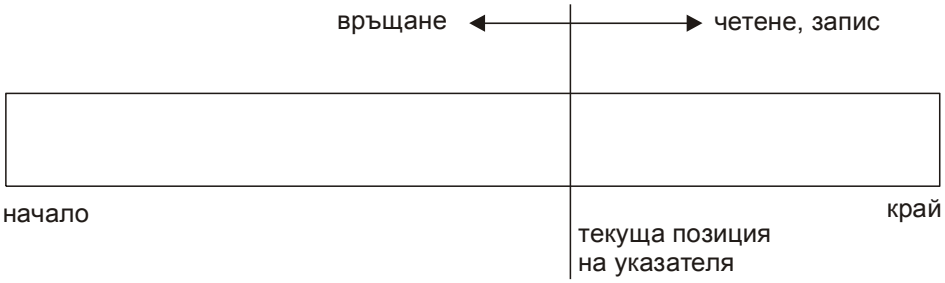
е) препозициониране във файл. Това е преместване на указателя върху даден елемент от файла и не включва вх/изх операция.

Към операциите за работа с файлове могат да бъдат добавени още две – отваряне и затваряне на файл. С тях може да бъде свързана следната информация:

- указател на файла. Някои системи не включват отместването като част от системните примитиви read и write, затова системата трябва да запомни мястото на последната вх/изходна операция като текущ указател във файла. Този указател е различен за всеки процес, работещ с файла и се съхранява отделно от дисковите атрибути на файла;
- брояч на отворените файлове. Тъй като един файл може да се използва от няколко процеса, необходимо е да се брои колко са те (колко опита за отваряне на файла има). За да бъде изтрит от таблицата на отворените файлове, един файл трябва да е затворен толкова пъти, колкото е отварян. В противен случай все още има процеси, които го използват и следователно, този файл не може да бъде премахнат от таблицата на отворените файлове;
- местоположение на файла върху диска. Повечето файлови операции предвиждат модифициране на данни във файла. Необходимата информация за намиране мястото на файла върху диска се пази в паметта за избягване на необходимостта от четене при всяка операция.

Последователен достъп

Най-простият метод за достъп е последователният. Информацията се обработва последователно, запис след запис. И досега това е най—често срещаният метод: редакторите и компилаторите ползват най-вече него. По-голяма част от файловите операции са четене и запис. При четене след операцията указателят на файла автоматично се премества върху следващия компонент. При запис към края на файла се добавя елемент и указателят автоматично се придвижва след него. Този метод може да работи както с устройства с последователен достъп, така и с устройства със случаен достъп. Указателят на файла може да бъде поставен в началото му, а в някои системи – върнат назад за n записа (дори за n = 1)



Файл с последователен достъп

Пряк достъп

Този метод се базира на дисковия модел на файл, докато последователния – на лентовия модел. Нарича се още относителен или директен достъп. Файловете се състоят от логически записи с фиксиран размер, което позволява на програмите да ги четат в произволен ред. Тъй като методът се базира на дисковия модел, дисковете позволяват случаен достъп до всеки блок от файла. За директен достъп файла се разглежда като номерирана поредица от блокове или записи. При прекия достъп е възможно примерно четене на блок 18, след това на блок 47, след това на блок 3. Няма ограничения върху реда на четене или запис.

Други методи за достъп

Други методи са развити на базата на прекия достъп. Те включват конструиране на индекс за файла и подобно на индексите в края на книга поддържат указатели към различни блокове. За намиране на запис във файла първо се претърсва индекса и по намерения указател се извършва пряк достъп до необходимия запис. Например, даден файл може да съдържа списък от кодове на някакви стоки и цени за тях. Всеки запис се състои от цифров код и 6 цифрова цена и дължината на записа е 16 байта. Ако дискът предоставя 1 024 байта за един блок, този блок може да съдържа 64 записа. Файл от 120 000 записа заема 2000 блока (2 милиона байта)

Ако файлът е сортиран по кодове, може да се определи индекс, състоящ се от първия код за всеки блок. Този индекс може да се пази в таблица, съдържаща 2000 елемента, всеки от по 10 цифри и следователно заемаща 20 000 байта. Тя се съхранява в паметта. За получаване цената на дадена стока се прави бинарно търсене в таблицата, от което се установява точно в кой блок се съдържа търсеният запис и се извършва достъп до този блок. Този метод позволява търсене в големи файлове при сравнително малък брой входно/изходни операции. За големи файлове индексът може да е твърде голям, за да се пази в паметта. Едно решение е да се създаде индекс за индекса. Главният индекс съдържа указатели към вторичен индексен файл (файлове), който от своя страна съдържа указатели към същинските блокове с информация.

Директории (папки, каталози)

Файловите системи могат да имат стотици файлове на дискове от стотици мегабайти. За управление на тези данни е необходимо те да бъдат организирани по някакъв начин. Тази организация обикновено се извършва на два етапа. Първо, файловата система се дели на дялове (минидискове за IBM или томове за PC и Macintosh). Един диск съдържа поне един дял. Дялът е структурата от ниско ниво, в нея се разполагат каталозите и файловете. Възможно е съществуването на няколко дяла в един диск и всеки се разглежда като отделно устройство. Други системи позволяват дяловете да са с по-големи размери от диск, така че няколко диска се групират в една логическа структура. Дяловете може да се разглеждат като виртуални дискове.

Вторият етап е организирането на информацията във всеки дял. Тя се съхранява в директория на устройството (device directory) или таблица на съдържанието на тома (volume table of contents). Директорията на устройството или за по-кратко просто директорията съдържа данни за всички файлове в дяла – име, местоположение, размер, тип.

Директорията може да бъде разглеждана като символна таблица, свързваща имената на файловете в елементи на директорията. Има различни начини за организация на директорите. В тях трябва да може да се включват нови елементи, да се изключват елементи, да се търси елемент, да се извежда списък на елементите.

При определяне на структурата на директориите е необходимо да се вземат под внимание операциите, които ще се изпълняват върху тях:

-търсене на файл. Файловете имат имена, съставени от символи и сходни имена може да означават, че файловете са в някаква взаимна връзка.

-създаване на файл. Необходима е възможност за създаване на нови файлове и добавяне към директорията;

-унищожаване на файл. Необходима е възможност за отстраняване на файл от директорията след като престане да е нужен;

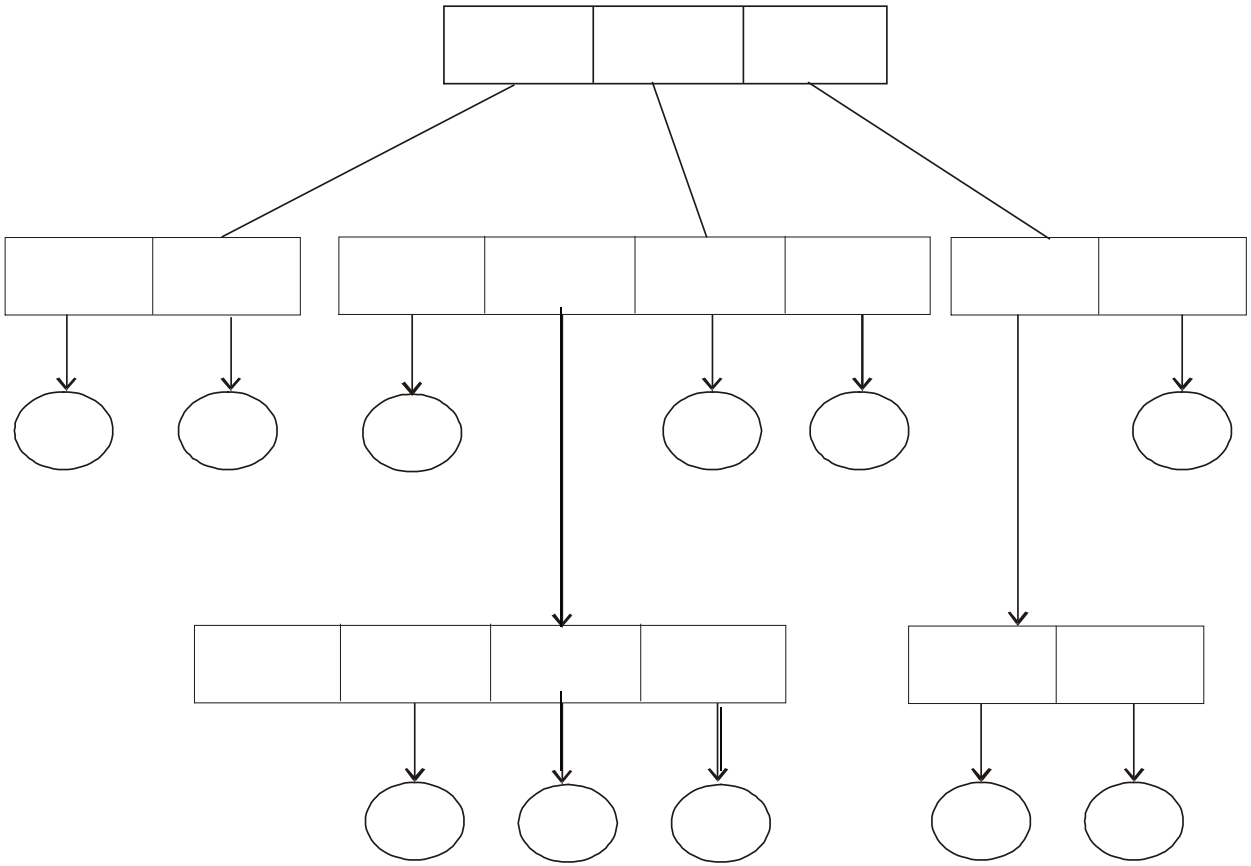
-извеждане на списък на директорията. Извеждане на имената и атрибутите на файла в директорията;

-преименуване на файл. Тъй като името на файла до голяма степен дава представа за съдържанието или предназначението му, необходима е възможност за промяна на името ако съдържанието или предназначението му се изменят;

-обхождане на файловата система

Директории с дървовидна структура.

От дървовидна структура с дълбочина 2 лесно се преминава към дървовидна структура с произволна дълбочина. Дървото има корен – коренова директория. Всеки файл в системата има уникално пълно име. Директория или поддиректория съдържа множество файлове или поддиректории. Всъщност директорията е файл, към който има специален подход. Всички директории имат еднакво вътрешно представяне. Един бит за всеки елемент на директория определя този елемент като файл (0) или поддиректория (1). Специални системни директиви създават и унищожават директории.



Директория с дървовидна структура

Реализация на файловите системи

Реализация на файлове

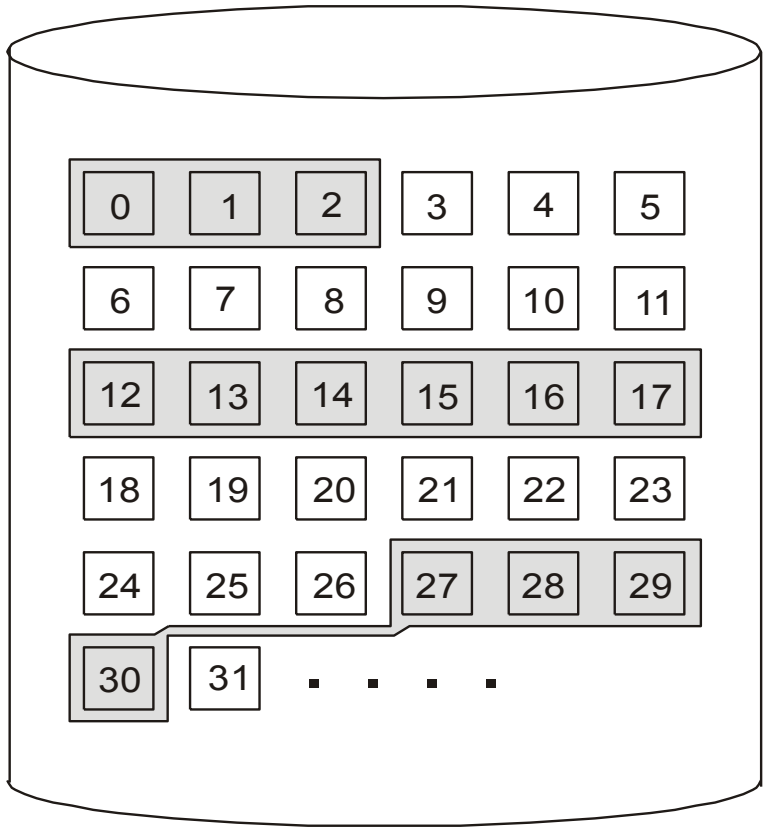
Основен момент в реализацията на файловете е поддържане на сведения кои дискови блокове принадлежат на всеки файл. В системите се използват различни методи. Тук ще разгледаме най-разпространените.

Последователно разпределяне

Най-простият начин на реализация е записването на файл върху диска в последователни блокове. Така например върху диск с размер на блока 1К, файл с размер 60 К ще заема 60 последователни блока. Тази схема има две съществени достойнства: първо, тя е проста за реализиране, тъй като е нужно да се съхранява само адреса на първия блок. Второ, четенето на файла може да се изпълни за една операция. Никой от другите методи дори не се доближава до показателя време за входно/изходна операция. За съжаление обаче тази методика има и два много сериозни недостатъка: първо, той е неприложим ако при създаването на файла не е известен максималния му размер. Без да знае максималния размер, ОС не може да резервира пространство на диска. Второ, в резултат от това разпределяне дискът страда от недопустима фрагментация. Уплътняване трябва да се извършва много често, а това е скъпо струваща операция.

каталог

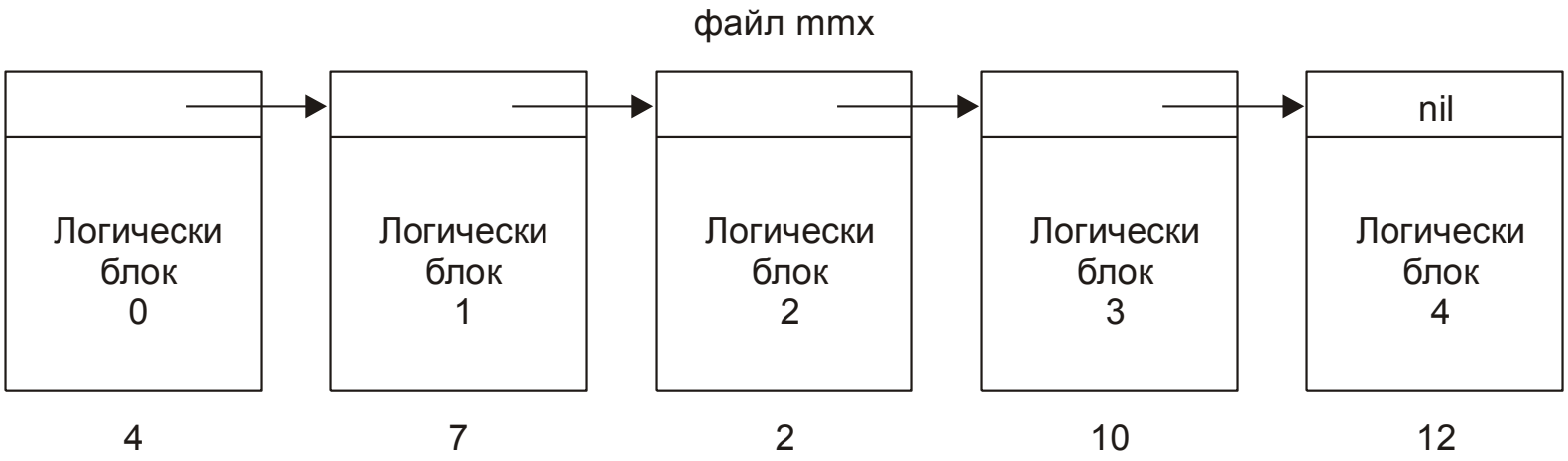
файл	начало	размер
emy	0	3
mail	12	6
x	27	4



Последователно разпределяне на дисковото пространство.

1.2.Свързан списък

Вторият метод на разпределяне предвижда блоковете на файла да се организират в свързан списък. Първата дума на всеки блок се използва като указател към следващия. Останалата част от блока съдържа данни.



Организиране на файл като списък от блокове

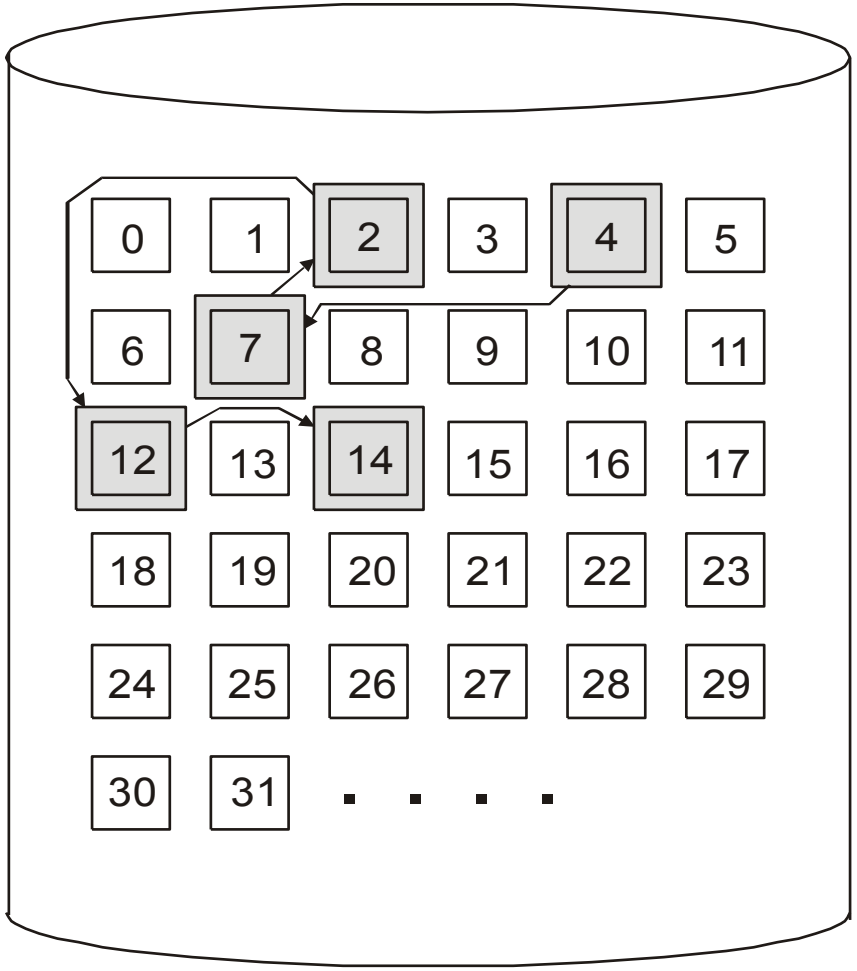
Свързано разпределяне

Този метод решава проблема с външната фрагментация, възникващ при последователното разпределяне. Всички дискови блокове могат да се използват. Всеки елемент от директорията съдържа указател към първия блок на файл. Ако файлът е празен, указателят има стойност null, а размерът е 0. При запис във файла системата намира свободен блок, записва в него и го присъединява в края на списъка. За четене на файла последователно се четат блоковете, като се следват указателите.

Освен че решава проблема с външната фрагментация, този метод не изисква деклариране размера на файла при създаване. Файлът може да расте докато има свободни блокове. От тук още следва, че не се налага уплътняване на диска.

каталог

файл	начало	край
mmx	4	14



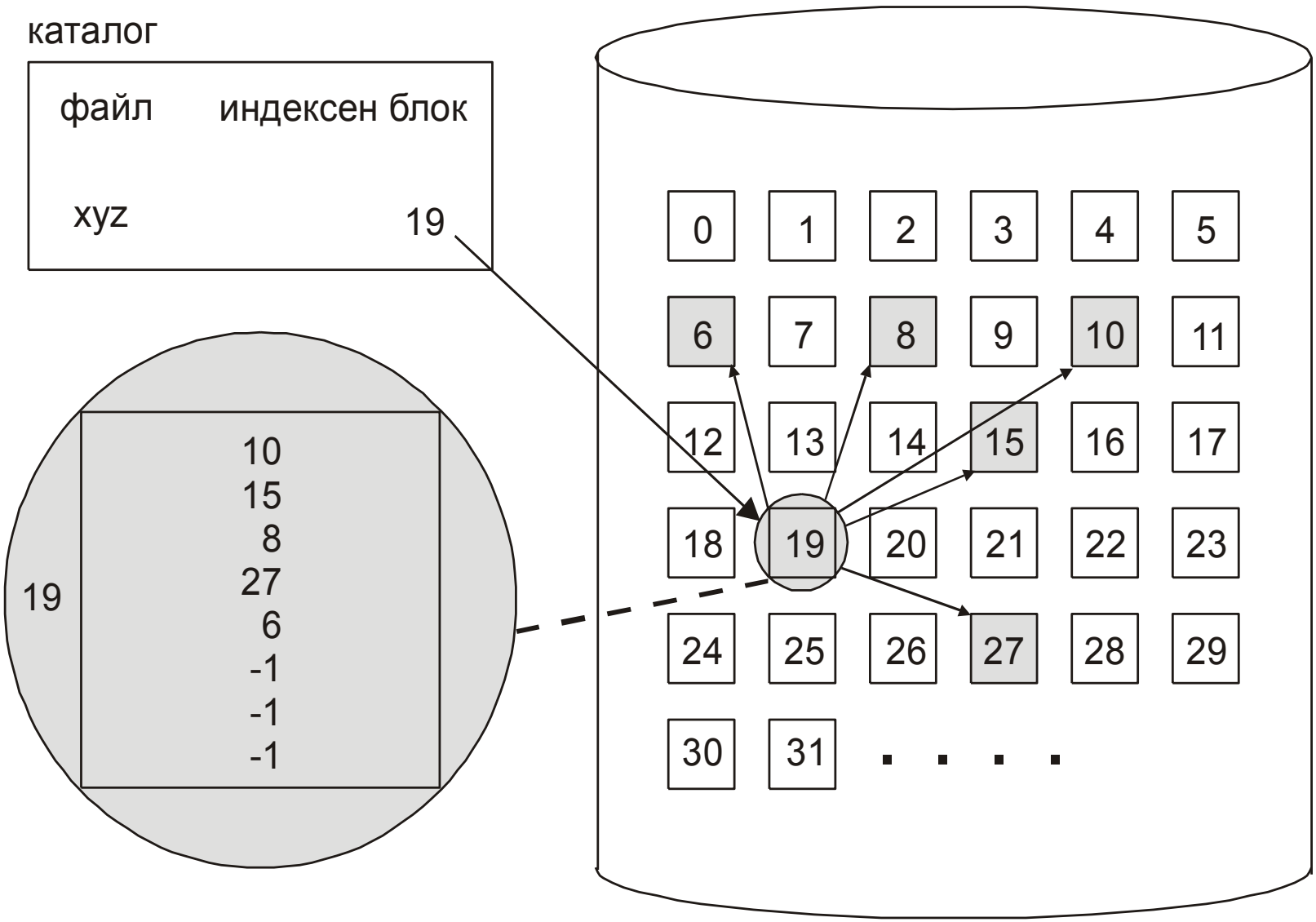
Свързано разпределя на диска

1.3. Индексирано разпределяне

Свързаното разпределяне решава проблемите с външната фрагментация и декларирането на размера на файла, съществуващи при последователното разпределяне. Но без реализация на FAT не поддържа ефективен пряк достъп. Този проблем е решен при събиране на всички указатели към блокове в един блок – индексен. Всеки файл има собствен индексен блок, наричан още индексен възел (i-node; i-възел), като i-тият вход (елемент) на блока съдържа указател към i-тия блок на файла. Директорията съдържа адреса на индексния блок. За прочитане на i-тия блок се използва указателя в i-тия елемент на индексния блок.

При създаване на файл всички указатели в индексния блок се установяват в null. Когато се записва блок, адреса му се получава от мениджъра на свободното дисково пространство и се записва в съответния елемент на индексния блок.

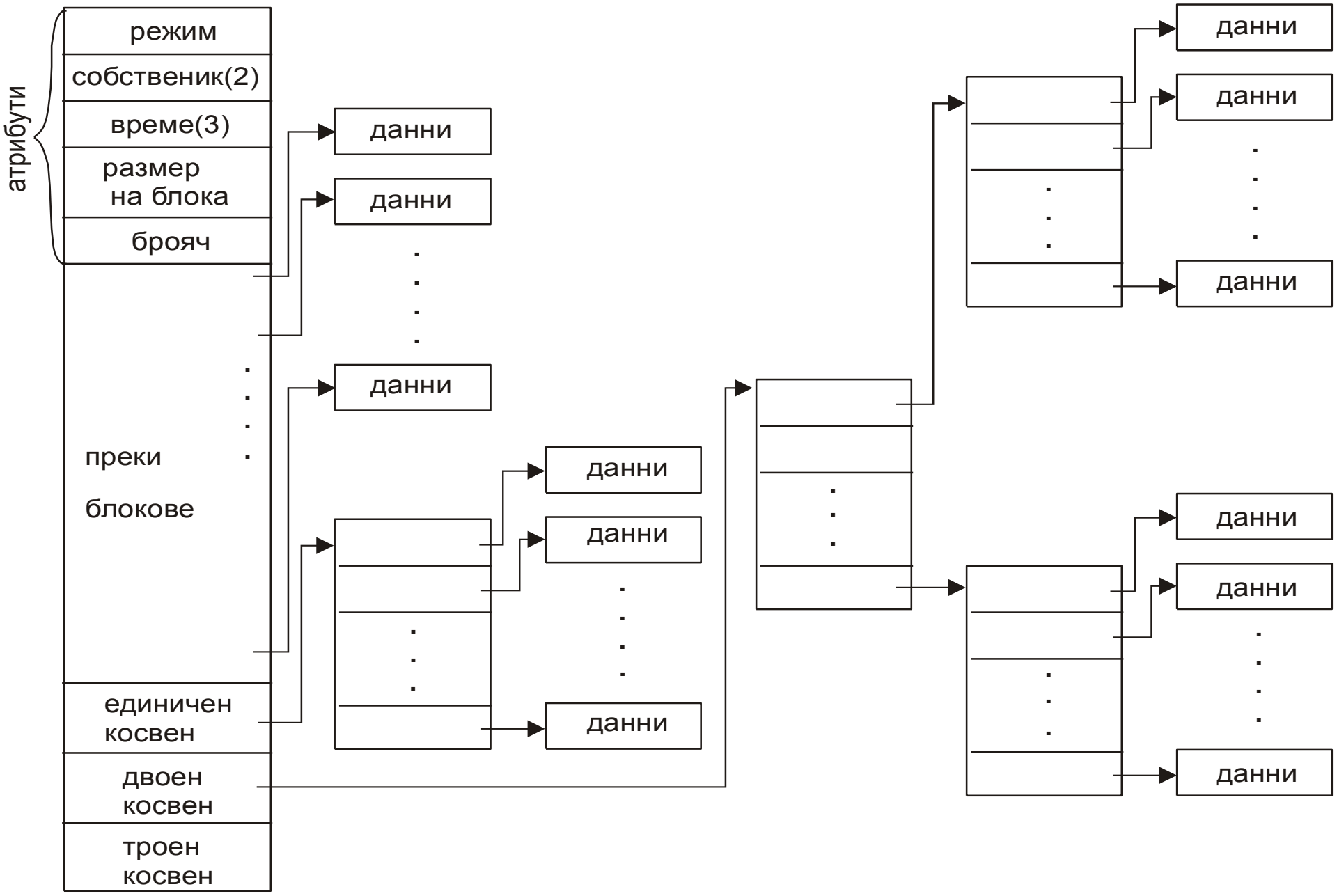
Индексното разпределяне поддържа прекия достъп до дисковите блокове без външна фрагментация, тъй като всеки свободен блок може да бъде предоставен за удовлетворяване нуждите на файловете от по-голямо пространство.



Индексно разпределяне на дисковото пространство

- Недостатък на този метод е загубата на дискова памет за съхраняване на идексните блокове: по-голяма отколкото при свързаното разпределяне. Например, ако един файл има само един или два блока, при свързано разпределяне за реализиране на указателите са необходими 8 байта (ако един указател заема 8 байта). За същия файл индексният метод предоставя цял блок за i-възела, въпреки че само един или два указателя са различни от null. Затова възниква въпросът за големината на индексния блок. Ако е малък, няма да има достатъчно указатели за блоковете на голям файл. Ако е голям, за малките файлове ще се губи много дискова памет при неизползвани указатели. За решаване на въпроса са възможни следните схеми:

1. Свързана схема: индексният блок е един дисков блок. Той може да бъде четен и записван директно. За големи файлове няколко индексни блока могат да се свържат в списък. Например, индексен блок може да съдържа малко “заглавие” с името на файла и първите 100 адреса на дискови блокове. За малки файлове последният адрес съдържа null, а за големи – указатели към следващия индексен блок.
2. Многостепенно индексиране. Вариант на свързаната схема предвижда индексиране на две нива: индексният блок от първо ниво съдържа указатели към индексни блокове от второ ниво, които вече съдържат указатели към дисковите блокове на файла. Тази схема може да се изпълни и за три нива индексирани блокове, в зависимост от размера на файла. Ако блокът има размер 4096 байта, ще съдържа 1024 указателя с размер 4 байта. За двустепенно индексиране това са 1 048 576 блока с данни или файл с размер до 4 гигабайта.
3. Комбинирана схема. Една алтернативна схема, използвана в BSD UNIX, е да се предоставят първите, например 15 указатели в индексния блок на файла. Една част от указателите, например 12, да са директно към дискови блокове. Тогава малки файлове с размер до 12 блока не се нуждаят от друг индексен блок. Ако размерът на дисковия блок е 4К, файлове до 48 К са достъпни директно. Следващите три указателя са към индиректни (косвени) блокове.


атрибути
режим
собственик(2)
време(3)
размер на блока
брояч
преки блокове
единичен косвен
двоен косвен
троен косвен
данни
данни
данни
данни
данни
данни
данни
данни
данни
данни
данни
данни

UNIX i-възел

Въпроси:

1. Дайте определение за файл.
2. Една от долуизброените не е операция върху файл (от гледна точка на файловата система):

а) запис;

б) отваряне;

в) прикачване;

г) отсичане.

3. Избройте методите за достъп до данните във файловете.
4. Формулирайте поне един недостатък на последователното разпределяне при реализация на файлове.
5. Какво съдържа един индексен възел?

Лекция 8

Управление на устройствата

Двете главни задачи на една компютърна система са процесиране и въвеждане и извеждане на данни, или т.нар. вход/изход. В много случаи второто е главната работа в системата, а процесирането е инцидентно. Ролята на операционната система е да управлява входно/изходните операции и входно/изходните устройства (външни устройства, периферия).

Една от класификациите на устройствата може да се извърши според предназначението им:

1. Устройства за съхраняване на информация (storage devices). Такива са магнитните и оптични дискове.
2. Устройства за предаване на информация – модеми, мрежови карти.
3. Устройства за интерфейс с потребителя – дисплеи, клавиатури, мишки.

Друга категоризация се извършва според характера на приеманата и предавана информация:

1. Блочни устройства. Работят с информация, организирана в блокове с фиксиран размер, всеки от който е със собствен адрес. Дисковете са типични блочни устройства с размер на блока от 128 до 1024 байта. Важно свойство за блочните устройства е възможността всеки блок да бъде четен или записван независимо от останалите.

2. Символни устройства. Приемат или предават поток от символи. Той не е адресуем и операцията търсене в него (seek) не е възможна. Терминали, принтери (печатащи текст), хартиени ленти, мрежови интерфейси, мишки (показалци) са символни устройства.

Контролери

Устройствата общуват с компютърната система, изпращайки сигнали. Това може да стане през точка за комуникация, наречена **порт**. Ако едно или повече устройства използват обща съвкупност от съединения, то тя се нарича шина. По-формалното определение за шина е съвкупност от съединения и строго определен протокол, определящ множеството съобщения и начина им на изпращане по шината. Шините са широко използвани в компютърната архитектура. Рис. 1. показва типична шинна PC структура. Това е PCI шина, свързваща подсистемата процесор-памет с бързите устройства и допълнителна шина (expansion bus) за връзка със сравнително бавните устройства, такива като клавиатурата и последователния порт. Дисковете са свързани чрез SCSI-шина, интерфейсът на която с PCI-шината е SCSI-контролер.

Контролерът е процесор за управление на устройство, порт или шина. Контролерът на последователния порт е пример за прост контролер. SCSI-контролерът е пример за сложен контролер.

Контролерите имат един или повече регистри за данни и управляващи сигнали. Процесорът общува с контролера като чете или записва в тези регистри определени поредици от битове. Един от двата начина на общуване е чрез специални входно/изходни инструкции, определящи трансфера на байт или дума към адрес на входно/изходния порт. Чрез входно/изходната инструкция се определя нужното устройство и се записва или чете определена редица значения на битове в регистъра (регистрите) на контролера. Другият начин е регистрите да са част от адресното пространство на процесора. В този случай специални входно/изходни команди не са нужни, тъй като се използват командите за работа с паметта.

Взаимодействията между елементи от различни нива на йерархия, каквито са процесорът и контролерите на устройствата, се извършва по два начина. Ако процесорът иска да изпълни входно/изходна операция, записва в регистрите на контролера съответната инструкция. След приемане на командата процесорът може да продължи работата си паралелно с изпълнението на входно/изходната операция. След изпълнението на инструкцията контролерът генерира **прекъсване** към процесора. То се обработва от съответната процедура, като се проверява резултата от операцията. Процесорът получава резултата и състоянието на устройството, като чете данните от регистрите на контролера.

Другият метод е периодично **запитване** от страна на процесора към контролерите с цел да се определи кое устройство е в състояние на готовност. Предимството на втория метод е в това, че запитването се извършва само тогава, когато процесорът е готов да предостави необходимото обслужване. Главен недостатък е загубата на време за запитвания в случай, че готовност отсъства.

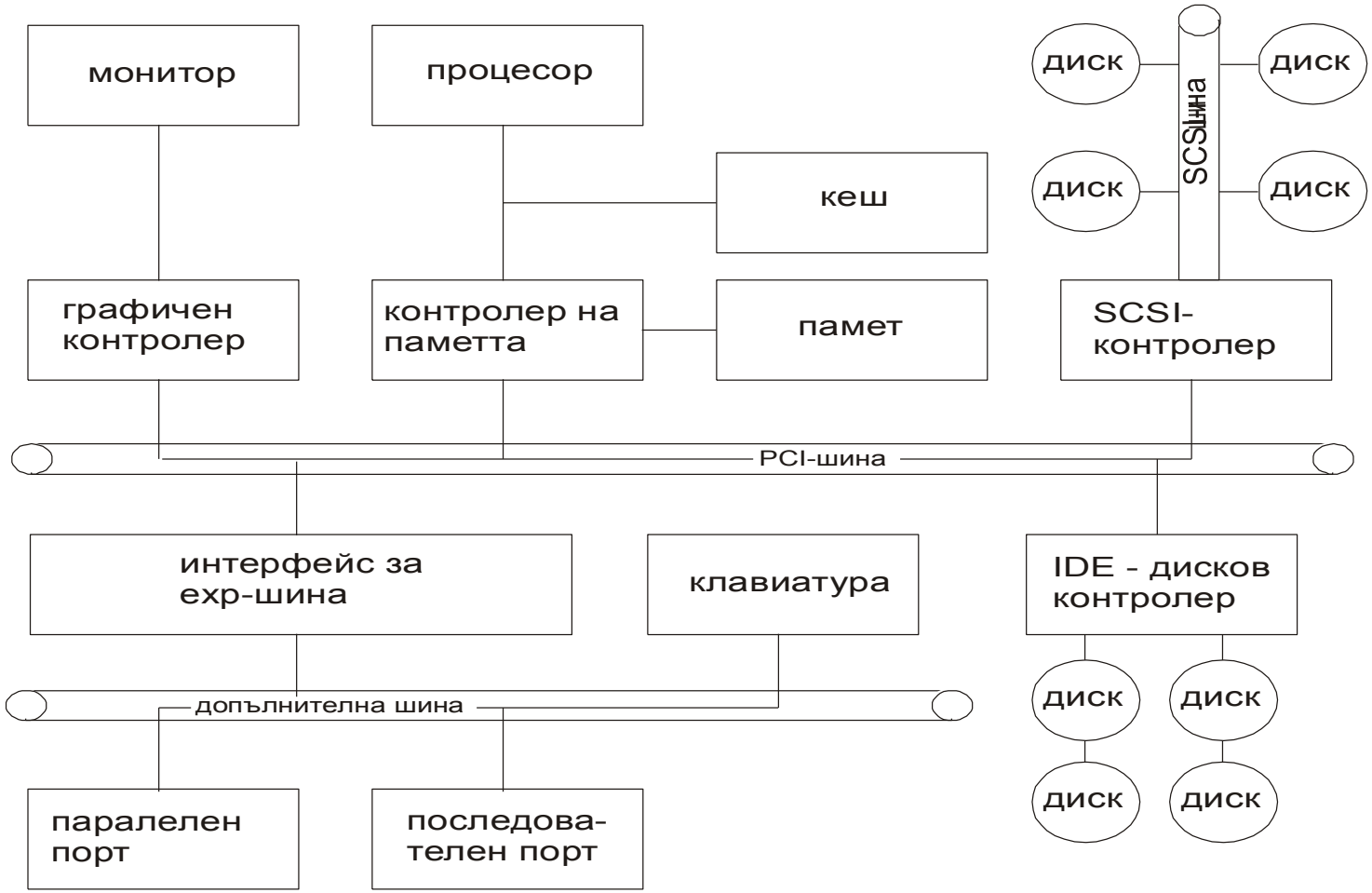
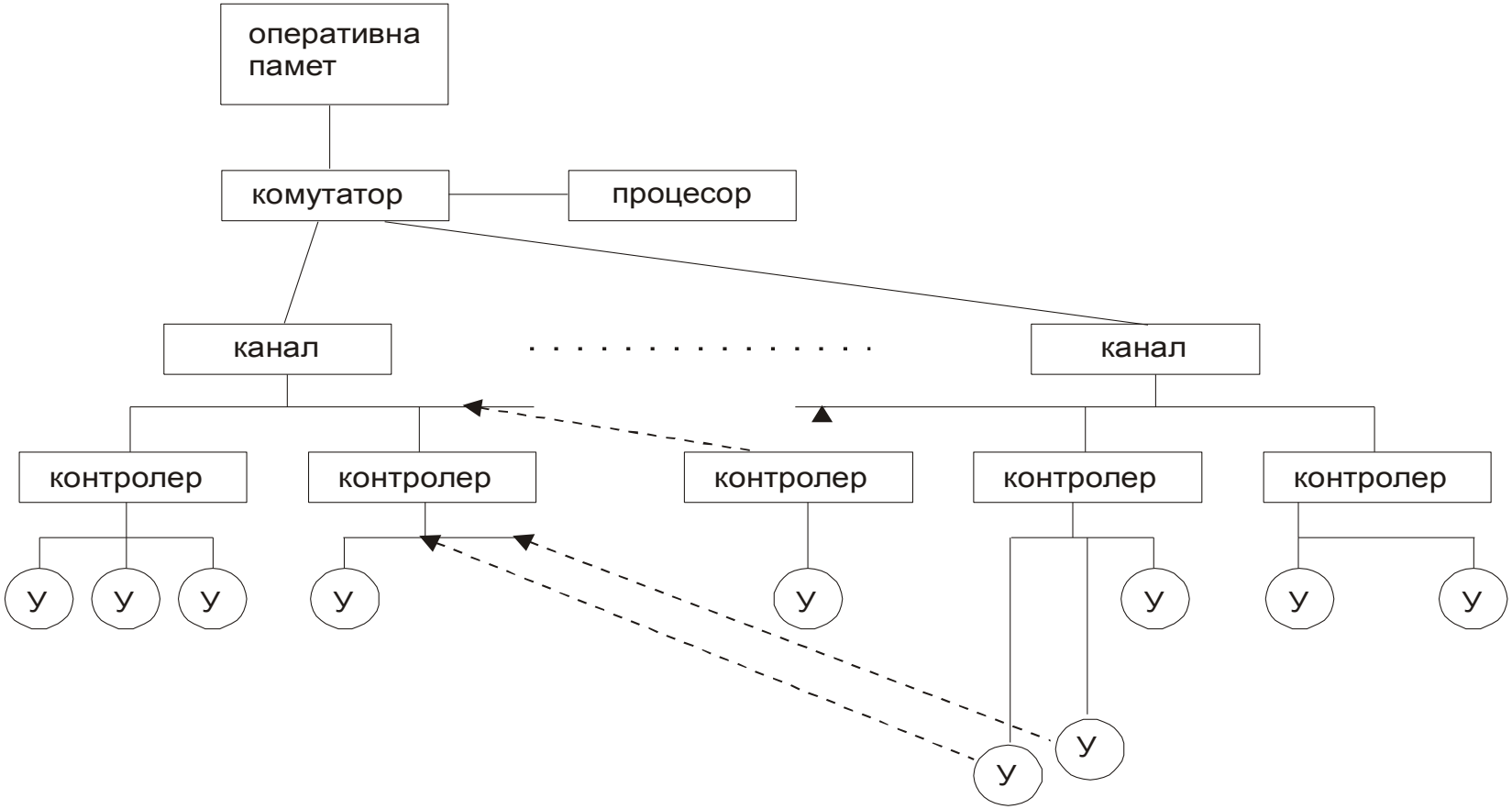


Рис. 1. Типична шинна структура за PC

Канали

В големите системи за организация на паралелна работа на процесора и устройствата между процесора и контролерите се използват специални посредници, наречени канали или входно-изходни процесори. Каналите са устройства, осигуряващи предаването на данни между паметта и контролерите. В системата може да има няколко селекторни и мултиплексорни канала. Каналите работят паралелно с централния процесор (процесори) и един с друг под управлението на канални програми. Най-простите канали се ограничават с изпълнение на еднокомандни програми; други могат да изпълняват редица инструкции. Тези инструкции определят операциите, които трябва да се изпълнят от устройствата, а също управляват прехвърлянето на данни между контролерите и буферите в паметта.

В подсистемата за вход-изход данните се предават от паметта чрез канал в контролер и след това към устройството и обратно. С паметта физически могат да бъдат свързани няколко канала, с всеки канал – няколко контролера, с всеки контролер – няколко устройства. За да може към ограничен брой канали от различен тип да се подключват всякакви устройства, необходимо е те да удовлетворяват някакъв стандартен интерфейс. Това позволява въвеждането на нови устройства без модифициране на процесора, паметта, каналите или шините, способства за осигуряване на независимост от устройствата.



Дървовидна организация на входа/изхода с възможност за превключване на устройства и контролери

Прекъсвания

Всеки процесор притежава поне един вход, чрез който постъпват сигналите за прекъсване. След изпълнението на всяка инструкция процесорът проверява наличието на сигнал за прекъсване. Когато някой контролер е изпратил сигнал за прекъсване, процесорът съхранява част от състоянието си – програмния брояч, и стартира изпълнението на програма за обработка на прекъсванията, започваща от точно определен адрес в паметта. Програмата за обработка на прекъсванията (ПОП) определя причината за прекъсването, обслужва го и изпълнява връщане към състоянието на процесора в точката преди прекъсването. Казваме, че контролерът на даденото устройство "**вдига**" прекъсване чрез сигнал по входа за прекъсване на процесора, процесорът **прехваща** прекъсването и го диспечира към ПОП, а ПОП **изчиства** прекъсването като го обслужва.

Повечето процесори притежават поне два входа за прекъсвания: един за **немаскируемо** прекъсване, резервирано само за много важни събития като например невъзстановими грешки на паметта; другият вход за прекъсване се нарича **маскируем** – той може да бъде изключван от процесора преди изпълнението на критични поредици от инструкции, които не трябва да бъдат прекъсвани. Точно маскируемият вход за прекъсване се използва от контролерите на устройствата за подаване на заявките за готовност.

Механизмът на прекъсването приема адрес – число, избиращо точно определена подпрограма за обслужване на прекъсване. Такива подпрограми не са много на брой. В повечето архитектури този адрес е величината на отместването в таблица, наречена вектор на прекъсванията. Той съдържа началните адреси на специализираните подпрограми за обработка на прекъсванията. Целта е да се минимизира търсенето на всички възможни причини за определяне коя от тях се нуждае от обслужване. На практика компютрите имат повече устройства, отколкото са слотовете във вектора на прекъсванията /и следователно, повече подпрограми за обработката им/. Един от начините за решаване на проблема е всеки елемент от вектора на прекъсванията да има указател към списък подпрограми за обслужване на прекъсванията. При възникване на прекъсване ПОП в съответния списък се обхождат до намиране на нужната. Такъв подход е компромисен между огромна таблица на прекъсванията и единствена и неефективна ПОП.

Съвременните операционни системи взаимодействат по различни начини с механизма на прекъсването. По време на зареждане операционната система проверява шините, за да установи наличието на устройства и инсталира съответните ПОП във вектора на прекъсвания. По време на входно/изходните операции прекъсвания се вдигат от различни контролери, когато са готови за обслужване. Тези прекъсвания индицират завършване на изход или че входни данни са вече налице, или входно/изходна грешка. Механизмът на прекъсвания управлява и голям брой изключителни ситуации като опит за делене на нула, опит за достъп до несъществуващ адрес, опит за изпълнение на привилегирована инструкция от потребителски режим. Всяко от тези събития изисква изпълнение на специална процедура за обработка на ситуацията.

Пряк достъп до паметта

В случаите на интензивен трансфер с някои устройства контролирането на статуса и изпращането на данни байт по байт в контролера губи ценно процесорно време. Тази дейност /т.нар. програмиран вход/изход/ често се възлага на специализиран процесор, като се цели освобождаване на централния процесор за други, по-важни функции. Този специализиран процесор се нарича контролер за директен достъп до паметта (DMA – Direct Memory Access). За инициализиране на DMA-трансфер процесорът записва команден блок в паметта. Този блок съдържа указатели към източника и приемника на информация и броя байтове за предаване. Процесорът записва адреса на този команден блок в контролера за директен достъп и преминава към други свои дейности. Тогава DMA-контролерът извършва трансфера на данни, работейки директно с паметта и съответното устройство, без участие на централния процесор.

Връзката между DMA-контролера и контролера на устройство се осъществява по двойка съединения. DMA-request и DMA-acknowledge, контролерът на устройството изпраща сигнал по линията DMA-request когато дума е готова за предаване. Когато получи този сигнал, DMA-контролерът монополизира шината на паметта, установява адреса на адресната шина и изпраща сигнал по линията DMA- acknowledge. При получаване на последния сигнал контролерът на устройството изпраща по шината за данни думата към гореуказания адрес в паметта, и снема DMA-request сигнала. При завършване на трансфера DMA-контролерът изпраща прекъсване до процесора. За времето, когато DMA-контролерът е обсебил шината на паметта, процесорът остава без достъп до шината и следователно до паметта, като за това време има достъп само до първичния и вторичния кеш. Въпреки че подобно действие забавя работата на процесора, като цяло производителността се увеличава, поради отпадане на необходимостта процесорът да се занимава с гореописания трансфер.

Трансферът на данни при директен достъп до паметта може да се опише в следните шест етапа:

1. Драйверът на устройството (напр. диска) получава команда да изпрати порция данни в буфер (в паметта) по адрес A.

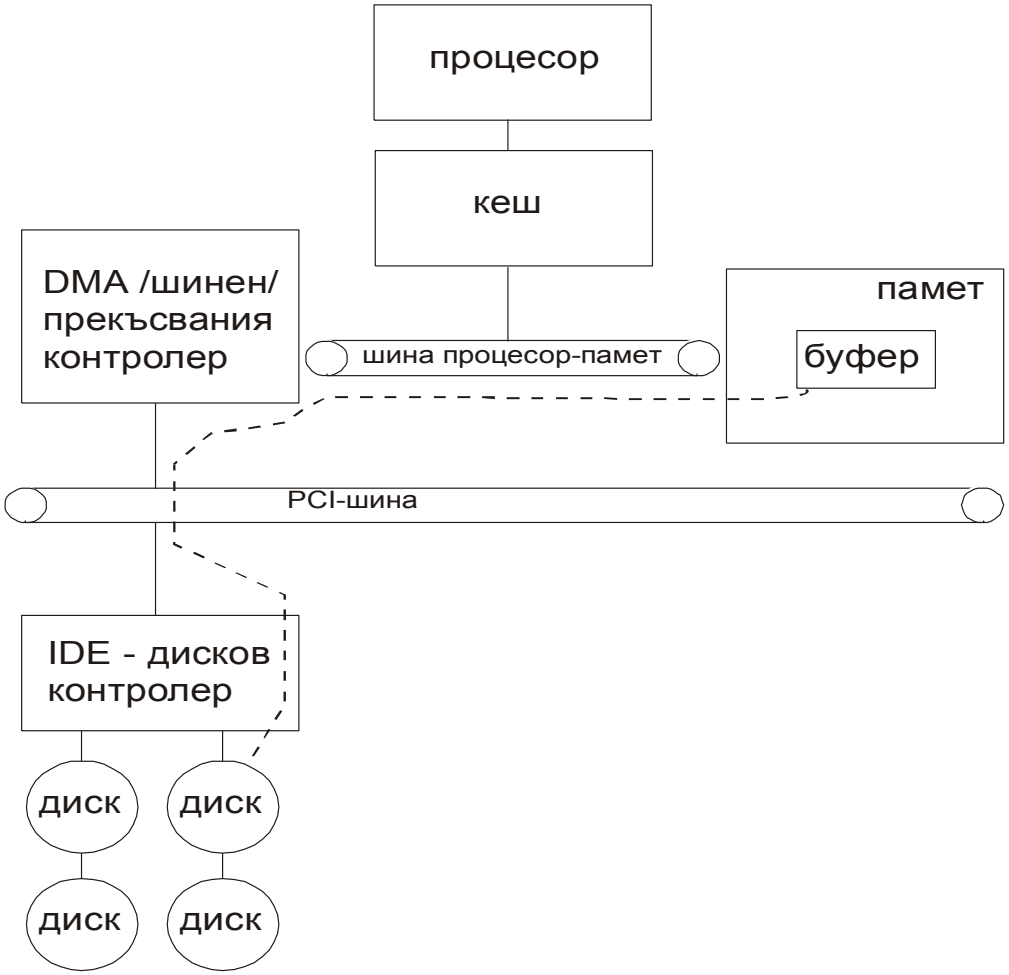
2. Драйверът на устройството дава команда на контролера да изпрати C байта от устройството в буфер по адрес A.

3. Контролерът на устройството инициира DMA-трансфер;

4. Контролерът на устройството изпраща всеки байт до DMA-контролера;

5. DMA-контролерът изпраща байтовете в буфера, започвайки от адрес A в паметта (или пресметнатия физически адрес A_{ph}, ако A е виртуален адрес). След изпращане на всеки байт контролерът увеличава адреса и намалява значението на C, докато C стане 0.

6. Когато C = 0, DMA изпраща сигнал за прекъсване на процесора с цел да съобщи, че трансферът е завършен.



Път на данните при пряк достъп до паметта

Принципи на входно/изходния софтуер

Главна цел при разработването на входно-изходния софтуер е конструиране на поредица слоеве, като слоевете от ниско ниво трябва да скрият особеностите на апаратурата и да представят някаква виртуална машина за слоевете от по-високо ниво. Най-горният слой цели да предостави на потребителя ясен и удобен интерфейс.

1. Ключова концепция в разработването на входно/изходния софтуер е независимост от устройствата.
2. Друго изискване, което е добре да се спазва, е имената на файловете и устройствата да са универсални, да не зависят от устройствата.
3. Важна задача за входно/изходния софтуер е откриване и обработване на грешки. Обикновено стремежът е грешките да се локализират и обработват на възможно най-ниско ниво, най-близо до апаратурата.
4. Следващ ключов момент е деленето на входно-изходните трансфери на синхронни и асинхронни (с използване на прекъсване). Работа и с двата вида.
5. Освен това, устройствата могат да се разделят на два големи класа: такива, които позволяват поделяне (съвместно използване) и други, които могат да бъдат ползвани монополно, само от един процес. Работа и с двата вида.

Реализирането на тези цели и принципи се постига чрез структуриране на входно/изходния софтуер на следните четири нива:

1. Програми за обработка на прекъсванията;
2. Драйвери на устройства;
3. Програмно осигуряване, независещо от устройствата;
4. Програми от потребителско ниво.

Програми за обработка на прекъсванията

Тези програми са почти невидими за потребителя. Възможно най-малко модули от системата трябва да работят с тях. Най-добрият начин за постигане на това е всеки процес, стартирал вход/изход, да блокира: чрез изпълнение на операция P върху семафор или wait върху условна променлива и пр. След завършване на входно/изходния трансфер устройството инициира прекъсване, а съответната процедура за обработка на прекъсването трябва да изпълни определени действия за деблокиране на процеса, който е предизвикал стартирането й. В някои системи това може да е V операция върху семафор, в други – send върху условна променлива в монитор, в трети – изпращане на съобщения. При всички случаи крайният резултат от прекъсването е блокираният процес да бъде приведен в готовност и впоследствие възобновен.

Драйвери на устройства

В драйверите на устройства е съсредоточен целият програмен код, зависещ от спецификата на устройствата. Всеки драйвер обслужва един тип устройство или един клас много близки по тип устройства.

Контролерите на устройствата притежават един или повече регистри за команди. Драйверите изпращат тези команди и контролират правилното им изпълнение. Затова дисковият драйвер е единствената част от операционната система, която знае колко регистъра има дисковият контролер и за каква цел се използва всеки от тях. Единствено драйверът знае секторите, пътечките, цилиндрите, главите и прочие подробности, които се използват за поддържане на правилна и точна работа с диска. Иначе казано, главната функция на драйвер на устройство е да получава абстрактни заявки от по-високото ниво на независимите от устройствата програми и да осигури тяхното изпълнение.

След изпращане на командите към контролера е възможна една от следните две ситуации: при първата драйверът трябва да чака, докато контролерът завърши работа. Тогава драйверът блокира и след получаване на прекъсването от контролера деблокира. В други случаи операциите завършват без забавяне и не е необходимо драйверът да блокира.

Независимо дали драйверът блокира или не, след изпълнение на операцията той трябва да направи проверка за грешки. При отсъствие на такива налице е блок данни, който може да бъде изпратен към източника на заявката от апаратно-независимия софтуер. Заедно с това връща и някаква статус-информация за изпълнението на операцията и проверката за грешки. Ако в опашката към драйвера има други заявки, избира една от тях и я обслужва. Ако опашката е празна, блокира и чака постъпване на заявка.

Апаратно-независим софтуер.

Голяма част от входно/изходните управляващи програми не зависят от спецификата на устройствата. Не може да се разделят точно функциите, изпълнявани от драйверите, от тези, изпълнявани от апаратно-независимите програми, тъй като някои функции, които могат да бъдат реализирани по независим от устройствата начин е по-ефективно да бъдат вградени в драйверите. Обикновено следните функции се изпълняват от апаратно-независимия софтуер:

1. Единен интерфейс с драйверите на устройствата;
2. Именуване на устройствата;
3. Защита на устройствата;
4. Осигуряване на апаратно-независим размер на блоковете данни;
5. Буфериране;
6. Разпределяне на паметта – за блочни устройства;
7. Разпределяне на монополно използуеми устройства;
8. Съобщения за грешки.

Потребителски входно/изходни програми

По-голямата част от входно/изходния софтуер е в операционната система. Съществуват обаче програми, изпълняващи се вън от ядрото и това обикновено са библиотеки. Системните директиви, включително за вход/изход, се изпълняват от библиотечни процедури. Когато програма на С съдържа следния ред

count = write(fd, buffer, nbytes);

библиотечната процедура write се свързва в програмата и в двоичен вид се намира в паметта по време на изпълнение. Съвкупността от всички такива процедури е част от входно/изходната подсистема. Освен процедури от вида на read и write форматирането на входа и изхода също е функция на библиотечни процедури.

Планиране на входа-изхода

Планирането на входно/изходните заявки означава определяне на подходящ ред на изпълнението им. Редът на генериране на тези заявки е произволен и обикновено не е най-добрият за ефективната работа на системата. Планирането на заявките за вход/изход може да повиши производителността на системата, да понижи времето на очакване за завършване на входно-изходни операции.

Преподреждането на заявките при обслужване е същността на планирането на входа/изхода. Планирането се реализира чрез поддържане на опашка от входно/изходни заявки към всяко устройство. При генериране на блокираща входно/изходна системна директива заявката постъпва в опашката към даденото устройство. Планировчикът на входа/изхода преподрежда заявките в опашката с цел подобряване на производителността на системата като цяло и средното време на отговор за приложенията. От друга страна операционната система трябва да гарантира, че нито едно приложение няма да получи лошо обслужване поради отлагане на входно/изходните му заявки.

Буфериране

Буфер е област от главната памет, използвана за съхраняване на данни при трансфера им между две устройства или между устройство и приложение. Буферирането се прави поради следните три причини:

1. Поради разлика в скоростите на производителя и потребителя на потока от данни.
2. Друга причина за използване на буфери е различният размер на данните за един трансфер при различните устройства. Такива различия са особено актуални за компютърните мрежи, където се използват буфери за фрагментиране и реасемблиране на съобщенията.
3. Трета причина за буферирането е поддържане на т.нар. семантика на копиране за приложния вход/изход: Нека едно приложение има буфер с данни, които иска да запише на диск. То извиква системната функция write, предоставя указател към буфера и цяло число, задаващо броя байтове за запис. Възможно е след връщането от write приложението да е модифицирало данните в буфера. Семантиката на копиране гарантира, че версията на записаните на диска данни е тази от времето на извикване на write независимо от последвали изменения в буфера на приложението.

Кеширане

Кешът е бърза памет, която съдържа копие на данни. Достъпът на данните в кеша е по-бърз отколкото до оригинала. Например инструкциите на текущия процес се намират върху диска, кеширали са основната памет и още веднъж във вторичния и първичен кеш на процесора. Разликата между буфер и кеш е в това, че докато в буфера може да се намира единственият екземпляр от данни, кешът по определение само използва физически по-бърза памет за съхраняване на данни, чийто оригинал съществува някъде на друго място.

Кеширането и буферирането имат различни функции и цели, но понякога област от паметта може да се използва и за двете. Например, за реализиране семантиката на копиране и ефективно планиране на дисковия вход/изход, операционната система използва буфери в основната памет. Тези буфери се използват и като кеш за повишаване скоростта на входа/изхода за поделени между различни приложения файлове или файлове, в които се пише или чете бързо (и често).

Спулинг и резервиране на устройства

Спулът е буфер, съдържащ изход към устройство, което не може да приема редуващи се потоци от данни от различни процеси, например принтер. Принтерът за определен период от време може да обслужва само един процес, но в същото време е възможно много други процеси да искат отпечатване на резултат. Операционната система приема всички заявки за печат на принтера. Изходът на всяко приложение се спулира към отделен дисков файл. Когато едно приложение приключи с изхода в спул-файла, спулинг системата го поставя в опашка към принтера. Тази система копира на принтера спул-файловете един по един. В някои операционни системи спулинга се управлява от специален системен процес: спул-демон. В други ОС – от системна нишка. И в двата случая операционната система предоставя управляващ интерфейс, който позволява на потребителите и системния администратор да разпечатват опашката и видят какви процеси чакат печат, да отстраняват от опашката нежелани процеси, да прекъсват печатането за обслужване на принтера и пр.

Въпроси и задачи:

1. Дайте определение или описание за:

- контролер;
- маскируемо прекъсване;
- буфер;
- Кеш.

2. Обяснете какво представлява прекият достъп до паметта (DMA).