ОПЕРАЦИОННИ СИСТЕМИ

Специалност Компютърни науки

гл. ас. д-р Моника Филипова

Катедра Изчислителни системи, ФМИ

летен семестър на уч. 2012/2013г.

СЪДЪРЖАНИЕ

Първа глава

ФУНКЦИИ И СТРУКТУРА НА ОПЕРАЦИОННИТЕ СИСТЕМИ 4

1.1. ПОНЯТИЕ ЗА ОПЕРАЦИОННА СИСТЕМА 4
1.2. СТРУКТУРА НА ОПЕРАЦИОННА СИСТЕМА 7

Втора глава

КОМАНДНИ ЕЗИЦИ 8

2.1. ПРИНЦИП НА ДЕЙСТВИЕ НА КОМАНДНИЯ ИНТЕРПРЕТАТОР 8
2.2. ПРОГАМИРАНЕ НА КОМАНДЕН ЕЗИК 9
2.2.1. Метасимволи 9
2.2.2. Променливи 14
2.2.3 Оператори 24
2.2.4. Функции 31
2.2.5. Екраниране 32
2.2.6. Замествания и изпълнение на команда 33

Трета глава

ФАЙЛОВИ СИСТЕМИ 34

3.1. ЛОГИЧЕСКА СТРУКТУРА НА ФАЙЛОВА СИСТЕМА 34
3.1.1. Имена и типове файлове 34
3.1.2. Каталози и организация на файловата система 36
3.2. СИСТЕМНИ ПРИМИТИВИ ЗА РАБОТА С ФАЙЛОВЕ 39
3.3. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВА СИСТЕМА 46
3.3.1. Стратегии за управление на дисковата памет 46
3.3.2. Системни структури 48
3.3.3. Реализация на каталог 50
3.4. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВА СИСТЕМА В UNIX 52
3.5. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА СИСТЕМА В LINUX 61
3.6. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА СИСТЕМА В MSDOS 65
3.7. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА СИСТЕМА NTFS 69

Четвърта глава

ПРОЦЕСИ 74

4.1. МОДЕЛ НА ПРОЦЕСИТЕ 74
4.1.1. Йерархия на процесите 74
4.1.2. Състояние на процес 75
4.2. КОНТЕКСТ НА ПРОЦЕС 78
4.2.1. Образ на процес 79
4.2.2. Таблица на процесите 80
4.2.3. Стек на ядрото и динамична част от контекста на процес 81
4.2.4. Системни примитиви - интерфейс и изпълнение 82
4.3. СИСТЕМНИ ПРИМИТИВИ ЗА УПРАВЛЕНИЕ НА ПРОЦЕСИ 83
4.4. МЕЖДУПРОЦЕСНИ КОМУНИКАЦИИ 88
4.4.1. Взаимно изключване 88
4.4.2. Семафори 91
4.4.3. Съобщения 96

4.5. ПЛАНИРАНЕ НА ПРОЦЕСИ 101
4.5.1. Нива на планиране 101
4.5.2. Цели на планирането 101
4.5.3. Дисциплини за планиране 102
4.6. НИШКИ 108
4.6.1. Реализация на нишки 109
4.6.2. Основни операции с нишки 111

Пета глава

ДЕДЛОК 113

5.1. НЕОБХОДИМИ УСЛОВИЯ ЗА ДЕДЛОК 113
5.2. ГРАФ НА РАЗПРЕДЕЛЕНИЕ НА РЕСУРСИТЕ 114
5.3. ПРЕДОТВРАТЯВАНЕ НА ДЕДЛОК 114
5.4. ЗАОБИКАЛЯНЕ НА ДЕДЛОК 115

ФУНКЦИИ НА СИСТЕМНИТЕ ПРИМИТИВИ 118

ЛИТЕРАТУРА 119

Първа глава

ФУНКЦИИ И СТРУКТУРА НА ОПЕРАЦИОННИТЕ СИСТЕМИ

1.1. ПОНЯТИЕ ЗА ОПЕРАЦИОННА СИСТЕМА

Какво е операционна система? Ще започнем с изясняване на мястото на операционната система (ОС) в цялата компютърната система. Основата на една компютърна система е апаратната част (хардуер). Интерфейсът на този слой са машинните команди. Машинните команди са примитивни операции, които работят със сложни понятия, като прекъсвания, регистри, дискови глави, сектори и.т.н. За програмистите са нужни по-прости абстрактни понятия. Затова между човека и компютъра има дебел пласт програми, които крият истината за хардура от човека. В една компютърна система има много нива на абстракции, изграждани от софтуера, които приближават понятията до човека. На *Фиг.1.1* е представена доста опростено структурата на една компютърна система.

Комплексът от програмни модули, който създава първото (най-долното) ниво на абстракция в една компютърна система е операционна система-ядро. Интерфейсът на това ниво са ***системните извиквания*** (system calls, системни примитиви, системни функции). Системните извиквания разширяват реалната машина с нови абстрактни понятия и операции над тях. Затова наричат първите два слоя в компютърната система разширена машина.

Следващия слой включва множество различни програми, които се обединяват от програмата наречена ***команден интерпретатор*** и формират един слой. В много операционни системи тази програма се нарича ***shell*** (обвивка), защото заедно с останалите програми създава обвивка около ядрото, която осигурява по-удобен потребителски интерфейс. Интерфейсът на този слой се нарича ***команден език***. Програмите в този слой са разделени в следните групи:

- Команден интепретатор. Тази програма организира диалога на потребителя със системата. Приема командите на потребителя и ги изпълнява, като ако е необходимо активизира някоя от другите програми.
- Системи за програмиране. Тази група включва компилатори, свързващи редактори, асемблери, конструктури на сложни програмни системи и т.н. Това са програмите, с които се създава софтуер, включително и самата операционна система.
- Обслужващи програми. Тези програми са предназначени за всякакви потребители. Включват текстови редактори, програми за копиране, изтриване, преименуване на файлове, и т.н.
- Програми на администратора. Тези програми са инструмент на системния администратор. Чрез тях той регистрира или премахва потребители, проверява и ремонтира системата, преконфигурира системата и т.н.

Първите два слоя, без групата на потребителските програми (ПП), се обединяват под названието системни програми, но под операционна система в тесен смисъл на думата се разбира само ядрото. Една от разликите между тези два слоя е, че в машините, които имат няколко нива на привилегии на процесора, ОС-ядро работи в най-привилегирования режим (режим ядро или супервайзор), а командният интерпретатор и другите програми работят в по-ниско ниво на привилегии (потребителски режим). Например, Linux за Intel 80x86 използва ниво 0 за ядрото и ниво 3 за командният интерпретатор и другите програми.

Фиг. 1.1. Структура на компютърна система

И така операционната система изпълнява две основни функции: разширява възможностите на машината и управлява ресурсите на машината.

Разширява възможностите на машината

Операционната система реализира абстрактни понятия и операции за работа с тях, т.е. реализира системните примитиви. По този начин тя осигурява разширена машина, в която се програмира по-лесно чрез системните примитиви отколкото на реалната машина. Системните примитиви са различни в различните системи, но има някои общи принципи при реализацията им. Първо, системните примитиви са програмен интерфейс на ОС, т.е. извикват се от програма на някакъв език за програмиране (обикновено това е езика Си в Unix и Linux). За всеки системен примитив съществува библиотечна функция, която се извиква от програмата, затова ги наричат и системни функции. Това, което правят библиотечните фукнкции не е много, но там винаги има извикване на машинна команда `trap` или `svc`, т.е. програмно прекъсване. Следователно, всеки системен примитив предизвиква прекъсване и така управлението се предава в ядрото. По друг начин казано, всички системни примитиви са входове в ядрото (това обяснява термина системни извиквания) и това е втората им обща черта.

Друго общо в различните ОС по отношение на системните примитиви, са реализираните абстрактни понятия. Основните абстракции във всяка ОС са файл, каталог и процес.

Процес (process, задача, task) е ключово понятие в ОС. Тази абстракция се въвежда с цел да се изолира потребителя от детайлите при реализирането на едновременното изпълнение на няколко програми. Процес е програма в хода на нейното изпълнение. За разлика от програмата, която е нещо статично, процесът е дейност. Понятието процес освен програма включва и много други неща, като текущите стойности на машинните регистри (PC, PSW и други), на програмните променливи, състоянието на отворените файлове и т.н. Името на процес се нарича идентификатор на процес (process identifier или pid). ОС трябва да предоставя системни примитиви за създаване на процес (това е `fork` в Unix и Linux). Многопроцесната операционната система поддържа едновременото съществуване на няколко процеса, които са независими и работят асинхронно. В противен случай казваме, че ОС е еднопроцесна.

Файл е абстракция осигуряваща унифицирани операции за вход и изход, т.е. операции, които не зависят от входно-изходните устройства, съхраняващи данните. Да си припомним, че една от основните задачи на ОС е да скрива от потребителя

особеностите на хардуера и да предоставя удобни абстракции. Постоянен обект данни, който има име и се съхранява на входно-изходно устройство, се нарича файл.

Каталогът (directory) осигурява удобна организация на файловете, които са разположени на различни устройства, принадлежат на различни потребители и се използват за различни цели. Най-често това е йерархичната организация, т.е. всеки каталог представлява група от файлове и други каталози, която има име.

Това бе гледната точка - ОС съществува за да осигури удобен потребителски интерфейс.

Управлява ресурсите на машината

Операционната система трябва ефективно да управлява ресурсите на машината, като ги разпределя между много програми и потребители, състезаващи се за правото да ги използват. Например, ако няколко програми (процеса) се опитат едновременно да извеждат на едно и също печатащо устройство. Какво ще се случи? ОС трябва да осигури използването на този ресурс по такъв начин, че едновременно изпълняваните програми да не си пречат. Многопотребителската ОС поддържа едновременната работа на няколко потребителя. Тогава управлението на ресурсите става още по-важна функция, защото последователността, в която различните потребители ще използват ресурсите е непредсказуема. ОС трябва да решава от кого да се използва определен ресурс при наличие на няколко заявки за него, т.е. да разпределя ресурсите. Сложността на тази задача идва от там, че ресурсите са различни от гледна точка на възможностите за съвместното им използване. Кои са ресурсите? Това са процесор, оперативна памет, входно-изходни устройства.

Има два начина на разпределяне на ресурси:

a) Разпределяне във времето

Програмите или потребителите (всъщност процесите) използват ресурса последователно една след друга. Задачата на ОС е да решава: Кой да е следващия? Колко дълго да го използва? Да отнема ли насилствено предоставен ресурс? Когато ОС насилствено отнема ресурс от процес, казваме че има преразпределение. Пример за ресурси разпределяни по този начин са:

- централен процесор - обикновено с преразпределяне;
- печатащо устройство - обикновено без преразпределяне.

b) Разпределяне в пространството

Ресурса се разделя на части и всяка програма или потребител получава част от него. Проблемите, които ОС трябва да решава са следните: Да следи свободните и заети части. Да осигури защита на частите. Справедливо да разделя ресурса. Ресурси, управлявани по този начин, са оперативна памет и дискова памет.

Това бе втората гледна точка и начин на изучаване на ОС - поглед отвътре.

1.2. СТРУКТУРА НА ОПЕРАЦИОННА СИСТЕМА

За да добием по-добра представа за това, какво представлява операционната система отвътре, ще разгледаме няколко модела на структурата на тази сложна програмна система.

Монолитна структура

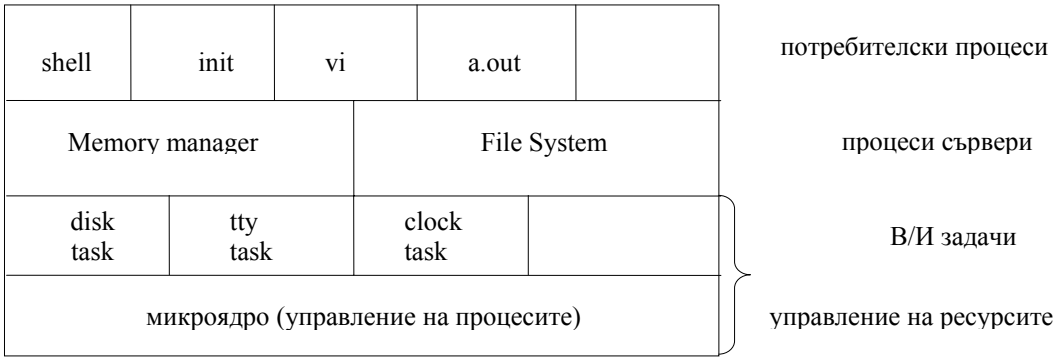
Монолитната структура или монолитна система означава, че всички програмни функции (модули) на операционната система са компилирани и свързани в един обектен модул. Всяка функция от програмите на ядрото има точно определен интерфейс, но може да вика всяка друга функция. Тази структура някои я наричат “The big mess”. Така полученото ядро работи не като отделен процес, а като част от всеки един процес. Такава е структурата на повечето версии на Unix и Linux системите.

Слоеста структура

Програмните модули на операционната система са организирани в йерархия от слоеве, като всеки следващ се изгражда върху по-долния, т.е. използва услугите на по-долния. Една от първите системи с такава структура е системата THE System (1968), създадена под ръководството на Е. Дейкстра в Университета в Айндховен. В тази система слоестата структура е използвана само при проектиране, но не и по време на работа на системата, т.е. накрая всички функции са компилирани и свързани в един обектен модул.

Микроядро

Тази структура е развитие на слоестата структура, като структура не само при проектиране, но и при работа на системата. Стремежът е колкото се може по-голяма част от програмния код на операционната система да се премести в по-горни нива и да работи като отделни процеси. Тези процеси работят като една система комуникирайки помежду си чрез някакъв механизъм. Ядрото ще реализира минимална функционалност и работи по вълшебния начин - като част от всеки един процес. Система с такава структура е MINIX (1987), създадена под ръководството на А.Таненбаум, а също така и Mach (1986) - версията на Unix. На *Фиг.1.2* е представена структурата на ОС MINIX.



Фиг. 1.2. Структура на операционната система MINIX

Втора глава

КОМАНДНИ ЕЗИЦИ

2.1. ПРИНЦИП НА ДЕЙСТВИЕ НА КОМАНДНИЯ ИНТЕРПРЕТАТОР

Командният интерпретатор (програма-shell) е програмата, която организира диалога на потребителя със системата. Потребителят задава своите операции чрез конструкции на командния език, които наричаме команди. Командният интерпретатор реализира командния език, като чете командите на потребителя и ги изпълнява. В този раздел ще разгледаме начина, по който това може да бъде направено.

Видове команди

Изпълняваните команди могат да се класифицират на различни видове, първо според реализацията им на:

- ***вътрешни*** – реализират се от програмата-shell
- ***външни*** – реализират се от програма, намираща се вън от програмата-shell.
  От своя страна програмата, реализираща външните команди може да е:
  - програма в изпълним код
  - програма на командния език (командна процедура, shell script, shell файл, команден файл)

Вторият начин за класификация на командите е според действието им на:

- ***обикновени команди***
- ***управляващи команди*** (***оператори***)

Схема на действие на програмата-shell (първи вариант)

След като потребителят успешно преодолее проверките по идентификацията си, за него автоматично се създава процес с прогамата-shell. Този процес се нарича login-shell процес (има и други видове shell-процеси). Първото му действие е да инициализира обкръжението си. След това влиза в цикъл, в който извежда покана, чете и изпълнява командите на потребителя. ***Покана*** (prompt) е съобщение, извеждано от процеса-shell, с което заявява готовността си да приеме нова команда.

В командния език има команда (logout или exit), която означава край на работата на потребителя със системата. Диалогът между потребителя и системата от идентификацията (login) до изпълнение на командата logout наричаме ***сесия***.

И така първият вариант на схема на действие на командния интерпретатор в многопроцесна ОС е следния:

```
shell() {
   инициализация на обкръжението на процеса-shell;
   while (не е край на сесия) {
        извежда покана;
        чете команда;
        if (командата е вътрешна)
                вътрешна_команда() ;
        else {
                по името на командата намира файла;
                if (файлът съдържа командна процедура)
                   превключва входа на текущия процес-shell от файла;
```

```
                else {
                  създава процес за изпълнимия код във файла;
                  if (режима не е фонов)
                    wait(); }
        }
    }
}
```

Когато ОС е многопроцесна е възможно външна команда да бъде изпълнена ***асинхронно***, т.е. паралелно с работата на процеса-shell. Такъв режим на изпълнение на команди се нарича още ***фонов режим***. Чрез функцията wait() се изчаква завършването на процеса, изпълняващ външната команда. При този вариант командна процедура не може да се изпълнява асинхронно.

Схема на действие на програмата-shell (втори вариант)

```
shell() {
   инициализация на обкръжението на процеса-shell;
   while (не е край на сесия) {
        извежда покана;
        чете команда;
        if(командата е вътрешна)
                вътрешна_команда() ;
        else {
                по името на командата намира файла;
                if (файлът съдържа командна процедура)
                   създава процес-subshell, който чете входа от файла;
                else
                   създава процес за изпълнимия код във файла;
                if (режима не е фонов)
                   wait();
        }
   }
}
```

При този вариант е възможно асинхронно изпълнение на всяка външна команда. Друго предимство на тази схема е, че командната процедура не може да смени обкръжението на текущия процес-shell, освен това могат да се пишат рекурсивни командни процедури. Но безспорно първият вариант е по-икономичен.

2.2. ПРОГАМИРАНЕ НА КОМАНДЕН ЕЗИК

В този раздел ще разгледаме по-сложните конструкции на командния език, които ни дават основание да казваме, че той е език за програмиране. По-конкретно ще разгледаме конструкции на командния език на bash (Bourne again shell) в Linux и на ksh (Korn shell) в Unix System V.

2.2.1. Метасимволи

Метасимволите, това са символи, които имат специално значение за програмата команден интерпретатор (shell). Когато се срещнат в определена конструкция на командния език, те се интерпретират по специален начин. Обикновено тази интерпретация се нарича заместване или разширение. Терминът екраниране на метасимволи (quoting и escaping) означава отменяне на специалното им значение. В

този раздел ще разгледаме няколко групи метасимволи, а именнно в имена на файлове, за пренасочване на вход/изход и конвейер, определящи режима на изпълнение на команда, и за групиране на команди в списъци. Останалите метасимволите, свързани с променливи, аритметични изчисления и др. ще бъдат разгледани в следващите раздели.

Генериране на списък от имена на файлове

Ако в дума, която трябва да е име на файл, се срещат неекранирани символите: “*”, “?” “[]”, то те имат специално значение. Думата се интерпретира от shell като шаблон за име на файл. Тя се разширява до азбучно подреден списък от имената на файловете, които съответстват на шаблона и се намират в текущия или явно указания каталог. Това разширение се нарича Pathname expansion. Ако не се открие нито едно съответствие, то думата остава непроменена. Специалното значението на символите е следното:

`*` Разширява се до произволен брой произволни символи (включително и 0).
`?` Разширява се до един произволен символ.
`[abc]` `[a-d]` Разширява се до един символ измежду изброените в скобите, или в интервала.
`[^ab]` Разширява се до един символ, различен от изброените в скобите.

Файл, чието име започва със символа “.” се третира от shell като скрит при някои операции, включително и при интерпретиране на горните метасимволи, т.е. те не се разширяват до “.” в началото на името на файла.

Пример. Нека всички файлове, намиращи се в текущия каталог са:

```
.bashrc  a.o  b.1.2  c  f  f1  fb  file.c
```

Тогава в зависимост от шаблона ще бъдат генерирани следните списъци от имена на файлове:

Шаблон	Списък от имена на файлове
`f*`	`f f1 fb file.c`
`*c`	`c file.c`
`f?`	`f1 fb`
`?b*`	`fb`
`[ab]*`	`a.o b.1.2`
`[a-c]*`	`a.o b.1.2 c`
`[^ab]*`	`c f f1 fb file.c`

Пренасочване на вход и изход

Всяка команда (всъщност всеки процес) има на разположение три стандартно отворени файла, които може да използва: стандартен вход, стандартен изход и стандартен изход за грешки. И трите файла стандартно са свързани с терминала на потребителя. С всеки отворен файл се свързва файлов дескриптор, който е цяло неотрицателно число и посредством него се идентифицира отворения файл в системните примитиви, изпълнявани от процеса. Файловите дескриптори на стандартните файлове са съответно 0, 1 и 2. Преди да изпълни определена команда shell може да свърже стандартните файлове с нещо различно от терминала и това се нарича пренасочване на вход/изход (Redirection). Синтаксисът за пренасочването на стандартния вход, изход и изход за грешки е следния:

команда < *файл*

Пренасочва стандартния вход на командата от файла.

команда > файл

Пренасочва стандартния изход на командата към файла. Ако файлът не съществува, първо се създава, а ако съществува, то старото му съдържание се изтрива.

команда >> файл

Пренасочва стандартния изход на командата към файла. Ако файлът не съществува, първо се създава, а ако съществува се отваря за добавяне.

команда 2> файл
команда 2>> файл

Пренасочва стандартния изход за грешки на командата към файла.

Пренасочване може да се прави за всеки един файлов дескриптор. Общият вид на конструкциите за пренасочване на вход/изход е:

[*n*]< *файл* По премълчаване *n* е 0 и е файлов дескриптор отворен за вход.

[*n*]> *файл,* [*n*]>>*файл* По премълчаване *n* е 1 и е файлов дескриптор отворен за изход.

[*n*]>*&m* По премълчаване *n* е 1. Тук *n* и *m* са файлови дескриптори за изход и изходите по двата дескриптора се сливат.

За една команда могат да се пренасочват няколко файла, като shell интерпретира пренасочванията отляво надясно. Преди пренасочването, в думата *файл* се интерпретират метасимволите: “$”, “~” и тези за имена на файлове (прави се заместване на променливи, аритметично разширение, заместване на символа “~” и разширение до имена на файлове). Резултатът от разширението става име на файла, но трябва да е една дума, иначе е грешка.

Пример. Пренасочване на стандартен вход и изход на команди.

```
ls dir1 > d1list
F=d1list
ls dir1 > $F
date > logfile
who >> logfile
write student < letter
cc prog.c 2> errorlist
find /home -name core –print > corelist 2> /dev/null
cat /dev/null > /var/log/wtmp
find /home -name core –print > corelist 2>&1
```

Изходът в два от примерите се пренасочва към специалния файл `/dev/null`, който представлява „черна дупка”. Това може да се използва за подтискане на стандартния изход или изход за грешки. Друго приложение е да се изтрие съдържанието на файл, като се запази името му и атрибутите, свързани с правата на достъп.

Последният пример показва как могат да се пренасочат стандартния изход и изход за грешки към един и същи файл. Има и по-кратка конструкция за това:

команда &>файл

Съществува друга форма за пренасочване на входа, която позволява командата и входните данни да се намират на едно място. Нарича се Here documents и има следния вид.

команда << низ
входни данни
низ

Тук *низ* е символен низ, който ограничава входните данни и не трябва да се среща в тях. Ако е указано << *низ*, то нито един метасимвол в данните не се интерпретира, в противен случай се интерпретират метасимволите “$”, “`” и символите за имена на файлове.

Пример. Пренасочване на стандартен вход - Here documents.

```
write student <<EOI
Hello from $USER
Happy New Year!
EOI
```

```
write student <<\EOI
Hello from $USER     # No expansion
EOI
```

Конвейер

Конвейерът (Pipeline) е конструкция, която включва две или повече команди, разделени чрез символа “|”, а именно:

команда1 | *команда2* [| *команда3*] ...

При изпълнението на конвейер shell пренасочва стандартния изход на *команда1* към стандартния вход на *команда2*, стандартния изход на *команда2* към стандартния вход на *команда3* и т.н. Командите работят като конкурентни процеси, които комуникират чрез един от механизмите за комуникация между процеси, наречен програмен канал (pipe). Команда, която чете данни от стандартния вход, извършва някаква обработка на данните и пише резултата на стандартния изход, се нарича филтър. Командният език включва голям брой команди-филтри, напр., `cat, cut, grep, head, od, pr, sort, tail, tee, tr, wc` и др., което дава много възможности за конструиране на конвейри.

Пример. Конвейер.

```
ls | wc –l
ls –l | grep "^-" | wc –l
ls –l | grep "^-" | tee filelist | wc –l
who | wc –l
who | grep "student" | wc –l
who | grep "student" | tee userlist | wc –l
```

Фонов режим на изпълнение

Изпълнение на команда във фонов (background) или асинхронен режим означава, че след като shell стартира изпълнението на командата не чака нейното завършване, а продължава работата си, т.е. извежда покана, ако е интерактивен, и чете следваща команда. Това означава, че в такъв режим могат да се изпълняват само външни команди. По премълчаване режимът на изпълнение е привилегирован (foreground) или синхронен, т.е. процесът shell се синхронизира със завършването на командата. За да бъде изпълнена една команда във фонов режим се използва друг специален символ “&”.

команда &

Ако командата използва стандартен вход, то трябва той да бъде пренасочен. Същото се препоръчва и за стандартния изход и изход за грешки, тъй като в противен случай резултатът е непредсказуем. Например, ако командата се опита да чете от

терминала, то ще й бъде изпратен сигнал, който ще предизвика спирането й (състояние stopped). Изпълнението на спряна командата може да се възстанови само в привилегирован режим (чрез командата `fg`).

Пример. Командата `find`, изпълнена във фонов режим с пренасочване на изходите.

```
find /home -name core –print > corelist 2> /dev/null&
```

Код на завършване

Всяка команда, изпълнена в привилегирован режим (всъщност всеки процес), при завършването си изработва код на завършване (exit status) и го връща към shell (родителския процес). Прието е код 0 да означава нормално (успешно) завършване на командата, а код различен от 0 (число от 1 до 255) да означава грешка или изключителна ситуация (неуспех). Код на завършване на конвейер е кода на последната команда в него.

Списък от команди

Последователност от команди и конвейри, които са разделени с някой от символите: “;”, “&”, “&&”, “||” и завършва с един от символите “;”, “&”или нов ред ще наричаме списък от команди. В зависимост от символа, разделящ командите (символ за групиране), има следните видове списъци:

- Списък за последователно изпълнение

  команда1 ; *команда2* [; *команда3*] ...

  Символът “;” е еквивалентен на символа за нов ред. Код на завършване на списъка е кода на последната изпълнена команда.

- Списък за асинхронно изпълнение

  команда1 & *команда2* [& *команда3*] ...[&]

  Командите в списъка се изпълняват асинхронно една спрямо друга. Shell стартира изпълнението на *команда1, команда2* и т.н. без да изчаква завършването им, след което ако накрая няма “&” той се синхронизира със завършването на последната команда, в противен случай не чака и нейното завършване, а продължава работата си.

- Списъци за условно изпълнение

  команда1 && *команда2*

  Изпълнява се *команда1* и ако тя върне код 0 се изпълняза *команда2*, в противен случай *команда2* не се изпълнява (AND list).

  команда1 || *команда2*

  Изпълнява се *команда1* и ако тя върне код различен от 0 се изпълняза *команда2*, в противен случай *команда2* не се изпълнява (OR list). Код на завършване на списък за условно изпълнение е кода на последната изпълнена команда.

  В един списък може да се срещне комбинация от различни символи за групиране. Символите “&&” и “||” са с еднакъв приоритет, който е по-висок приоритета на “;” и “&”.

Пример. Списъци от команди.

```
cmp –s f1 f2 && rm f2
grep "^student:" /etc/passwd > /dev/null 2>&1 || echo "No user"
```

В списък команди могат да се използват два вида скоби, които се интерпретират от shell по различен начин.

- Скоби

  { *списък_команди*; }

Списък от команди заграден във фигурни скоби се изпълнява от текущия процес shell. Конструкцията се използва за изменение на реда на интерпретиране на символите за групиране или за формиране на обединен изход или вход на няколко команди.

(*списък_команди*)

Списък от команди заграден в кръгли скоби се изпълнява от нов процес shell, който е породен от текущия процес shell (т.е. от subshell). Това означава, че всякакви изменения в обкръжението на процеса shell при изпълнението на *списък_команди* нямат ефект след завършването му (например, промяна на текущия каталог или присвояване на значения на променливи). Друга възможност, която дава тази конструкция, е изпълнение на списък от команди във фонов режим.

Пример. Списъци от команди и скоби.

```
cmp –s f1 f2 && { rm f2; mv f1 file1; }
{ date; who; } > logfile
{ cat d1list; ls dir2; } | wc -l
(cd dir; . . . )
(sleep 60; { date; who; } > logfile) &
```

2.2.2. Променливи

Променлива в shell има име, значение и евентуално атрибути. Името е символен низ от букви, цифри и символа за подчертаване "_", като започва с буква или символа за подчертаване. Значението е символен низ, включително и празен низ. Декларирането на променливи не е задължително. Променливата се обявява най-често с оператора за присвояване. С този оператор променливата се включва в обкръжението на текущия процес shell и е там докато не бъде изключена с командата `unset`. Операторът за присвояване има вида:

име_променлива=[*значение*]

Ако не е зададено *значение*, то значението е празен низ. Значението е низ, който може да е ограничен с единични или двойни кавички. Ако в низа няма интервали и метасимволи (като < > | & $ ` и други), то не е необходимо ограничаването му с кавички. Ако в низа има интервали, но няма метасимволи, то е задължително ограничаването му с кавички, независимо кои. Ако в низа има метасимволи, то двата вида кавички имат различно действие. Единичните кавички отменят специалното значение (екранират) на всички метасимволи в низа без своето собствено. Двойните кавички екранират всички метасимволи в низа с изключение на $, ` и \. Това означава, че в *значение* се прави заместване на променлива, заместване на изхода и аритметични разширения, ако съответните метасимволи не са екранирани.

В по-новите версии на shell (от Bash 2.xx) значението може да има вида: *$'низ'*. Тогава в *низ* се заместват всички последователности по ANSI C стандарта: `\n`, `\t`, `\v`, `\nnn`, където `nnn` е осмичен код на символ и др. Цялата конструкция се замества с получения низ, заграден в единични кавички.

Пример. Оператор за присвояване.

```
L=LINUX или L="LINUX" или L='LINUX'
D=/home/student/proc/dir1
OS="Red Hat LINUX" или OS='Red Hat LINUX'
A="a<b" или A='a<b'
X=$'aa\043\t\044' или X=$'aa#\t$'
```

Значението на променлива се използва, когато се срещне следната конструкция:

$*име_променлива* или ${*име_променлива*}

Тогава цялата конструкция, се замества със значението на променливата. Това се нарича заместване на променлива (Variable expansion/substitution). Ако променливата не е определена, то се замества с празен низ. Името на променливата се използва без символа „$" пред него само когато се присвоява значение, променят се атрибути или се иска информация за променливата.

Пример. Заместване на променлива.

```
OS="Red Hat $L"
cd $D
ls -l $D/a*
```

Освен тази основна и най-често използвана конструкция за заместване значението на променлива в по-новите версии на shell има по-сложни конструкции за заместване.

${*име_променлива*:=*дума*}

Ако *име_променлива* е неопределена или има значение празен низ, то й се присвоява значение *дума* и се замества това значение, иначе се замества значението на *име_променлива*.

${*име_променлива*:-*дума*}

Ако *име_променлива* е неопределена или има значение празен низ, то се замества *дума*, иначе се замества значението на *име_променлива*.

${*име_променлива*:+*дума*}

Ако *име_променлива* е неопределена или има значение празен низ, то нищо не се замества иначе се замества *дума.*

${*име_променлива*:?*дума*}

Ако *име_променлива* е неопределена или има значение празен низ, то се извежда съобщение *дума* на стандартния изход за грешки и завършва изпълнението на командната процедура с код на завършване 1, иначе се замества значението на *име_променлива*.

Ако в горните конструкции липсва символа „:", то се проверява само дали *име_променлива* е неопределена. Преди да се използва *дума*, в нея се извършва заместване на променлива, заместване на изхода, аритметично разширение и заместване на "~".

${*име_променлива*:*отместване*}

${*име_променлива*:*отместване*:*брой*}

Това са конструкции за заместване на подниз от значението на променлива. Замества се значението на *име_променлива*, започвайки от *отместване* (отместването се брои от 0) до края или максимално *брой* символи.

${*#име_променлива*}

Замества се с броя символи в значението на *име_променлива*.

Команди за променливи

С променливите са свързани няколко вътрешни команди: `echo`, `read`, `set`, `unset`, `export`, `readonly`, `declare`.

`echo [`*`опции`*`] [`*`низ`* `...]`

Извежда на стандартния изход аргументите си и символа за нов ред. Някои от опциите са:

`-n` Не извежда символа за нов ред накрая.

`-e` В *низ* се заместват всички последователности по ANSI C стандарта: `\n, \t, \v, \nnn`, където `nnn` е осмичен код на символ и др.

`read [`*`опции`*`]` *`име_променлива`* `...`

Чете ред от стандартния вход и го разделя на думи. Всяка дума присвоява на поредната променлива, указана като аргумент. Ако броят на прочетените думи не съответства на броя на променливите, то или последните променливи имат празно значение или на последната променлива се присвоява останалата част от реда. Тук името на променливата е без символа „$”, защото с `read` се присвоява значение на променлива. Някои от опциите са:

`-s` Не прави ехо на въвеждания ред.

`-p "text"` Извежда покана `text` преди да прочете реда.

`set`

Тази команда има много различни действия, но в най-простия си вид без аргументи извежда имената и значенията на всички променливи, които са определени в текущия процес shell.

`unset` *`име_променлива`* `...`

Указаните променливи се изключват от обкръжението на текущия процес shell, т.е. ще се считат за неопределени.

`export [-n][`*`име_променлива`* `...]`

Указаните променливи се отбелязват за предаване към обкръжението на процеси, които ще бъдат породени от текущия процес shell (това може да е и subshell). Такива променливи се наричат “експортни” или променливи с атрибут export. Те са подобни на глобалните променливи в езиците за програмиране, само че тук глобалността е еднопосочна. Ако има опция `-n`, то се отменя атрибута export за указаните променливи. Ако няма аргументи извежда имената и значенията на всички експортни променливи.

`readonly [`*`име_променлива`* `...]`

За указаните променливи се определя атрибут readonly, т.е. след това значението им не може да бъде променяно. Ако няма аргументи извежда имената и значенията на всички променливи с атрибут readonly.

```
declare [опции] [име_променлива ...]
```

Тази команда може да се използва вместо `export` и `readonly`. Чрез нея може да се обяви променлива с определени атрибути и дори значение, да се измени атрибут на променлива или да се получи информация за променливите от обкръжението на процеса shell. В същност, в командите `declare`, `export` и `readonly` вместо *име_променлива* може да е оператор за присвояване. Опциите определят действието на командата относно атрибутите.

`-x` Променливите се обявяват за експортни (атрибут export).
`-r` Променливите стават достъпни само за четене (атрибут readonly).
`-i` Променливите се обявяват за целочислени (атрибут integer).

Ако опцията се предшества от знак "+"вместо "-", то съответния атрибут се отменя. Ако няма аргументи променливи, командата извежда информация за променливите с указаните атрибути.

Пример. Команди `declare, export, readonly, set`.

```
declare               # set или declare -p
declare –x            # export или export -p
declare –r            # readonly или readonly -p
declare –xr           # all export or readonly variables
A=123; export A       # A=123; declare –x A
declare –x A=123
declare +x A          # export -n A
```

Системни променливи

Нормалната работа на shell, а и на цялата операционна система, зависи от ред променливи, които всеки потребител получава вече определени в началото на сесията, т.е. те са включени в обкръжението на процеса login shell. Ще ги наричаме ***системни променливи*.** Те могат да се използват, а някои да бъдат променяни от потребителя. Пълният списък от системни променливи варира в различните версии на shell, но има някои, които са общи и важни за функционирането му.

Име	Значение
`HOME`	Пълното име на началния каталог на потребителя.
`PATH`	Списък от каталози, в които shell търси външни команди.
`PS1`	Първичната покана (prompt) на shell. Възможни значения: $, # , bash$
`PS2`	Вторичната покана на shell. Стандартното значение е >.
`IFS`	(Internal Field Separator) Символите, които се разглеждат като разделители на думи при word splitting и командата `read`. Стандартно са "<space><tab><newline>".
`UID`	(User id) Вътрешен идентификатор на потребителя (с атрибут readonly).
`USER`	Име на потребителя. (Използват се също `USERNAME`, `LOGNAME`.)
`SHELL`	Име на потребителския login shell.
`PWD`	Името на текущия в момента каталог.
`OLDPWD`	Името на предишния текущ каталог.

Следващите променливи са типични най-вече за bash, но е възможно и поддържането им в други версии на shell.

`OSTYPE`	Наименование на ОС.
`HOSTTYPE`	Тип на машината.
`HOSTNAME`	Наименование на машината.
`BASH`	Името на файла с изпълнимия код на bash, стандартно е `/bin/bash`.

`BASH_VERSION` Версия на bash.

`BASH_ENV` Име на файл, който се изпълнява при извикване на командна процедура (аналог на инициализиращата процедура за интерактивен не-login shell - `.bashrc`).

При стартиране на процес shell могат автоматично да се изпълнят **инициализиращи командни процедури** (наричат ги profiles).Чрез тях обикновено се настройва обкръжението на процеса shell, например значенията на системните променливи. Изпълняваните командни процедури зависят от вида на процеса shell:

- login shell процес
  1. `/etc/profile`
  2. `~/.bash_profile, ~/.bash_login, ~/.profile` (първия намерен)
- интерактивен не-login shell процес
  `~/.bashrc`
- неинтерактивен shell процес (при изпълнение на командна процедура)
  `$BASH_ENV`

Ако дума започва с неекраниран символ `"~"`, той е метасимвол и въвежда така нареченото **разширение на символа `"~"`** (Tilde expansion). Вариантите на това разширение са:

`"~/"` и `"~ "`

Замества се със значението на променливата `HOME` или ако `HOME` не е определена с началния каталог на текущия потребител.

`~име_потребител`

Замества се с началния каталог на указания потребител.

`"~+/"` и `"~+ "`

Замества се със значението на променливата `PWD`.

`"~-/"` и `"~- "`

Замества се със значението на променливата `OLDPWD`.

Във всички други случаи не се прави разширение на символа `"~"`, т.е. съдържащата го дума остава непроменена.

Пример. Илюстрират се по-сложни конструкции за заместване на променливи в комбинация с други типове замествания. В коментар е даден изхода от съответната команда `echo`.

```
# paramsub
# login as moni, current directory is /home/moni/proc
#-------------------------------------------------
unset a  # or a=
i=1
echo "${a:-`whoami`}, a=$a"           # moni, a=
echo "${a:-$USER}, a=$a"              # moni, a=
echo "${a:-~+}, a=$a"                 # /home/moni/proc, a=
echo "${a:-$((i+1))}, a=$a, i=$i"     # 2, a=, i=1
a=
i=1
echo "${a:=`whoami`} a=$a"             # moni, a=moni
echo "${a:=$((i+=1))}, a=$a, i=$i"     # moni, a=moni, i=1
a=
echo "${a:=$((i+=1))}, a=$a, i=$i"     # 2, a=2, i=2
echo  ${a:+"do you want to redefine a?"} # do you want to redefine a?
echo ${a:?"a is unset or null"}        # 2
a=
echo ${a:?"unset or null"}  # paramsub: a : unset or null
                            # output to stderr and exit shell
echo "This will not execute"
```

```
exit 0
#-------------------------------------------------
```

Заместване на изхода

Конструкцията заместване на изхода (Command substitution) означава заместване на команда с това, което тя извежда на стандартния изход, т.е. стандартният изход на командата се замества в друг контекст. Какво ще се случи след това зависи от този контекст. По-старият и новият синтаксис на конструкцията са съответно:

`` `*команда*` `` и $(*команда*)

Вместо команда може да е конвейер и дори списък от команди. Възможно е влагане на конструкции заместване на изхода, но използването на стария синтаксис е доста сложно и неудобно. Много често заместването на изхода се използва в оператор за присвояване, като аргумент на друга команда, в цикъл `for`, за да генерира значения на управляващата променлива.

Пример. Заместване на изхода на команда.

```
D1=`pwd` или D1=$(pwd)
now=`date` или now=$(date)
day=`date|cut -f1 -d' '`
num=`who|wc -l` или num=$(who|wc -l)
echo "Number of sessions:`who|wc -l`"
echo `ls \`pwd\`/a*`
echo $(ls $(pwd)/a*)
echo `ls $(pwd)/a*`
echo $(ls `pwd`/a*)
```

Аритметични изчисления

Аритметични изчисления в командния език могат да се извършат по няколко начина. Една възможност е да се използва командата `expr`, която се разпространява с всички Unix и Linux системи.

`expr` *`израз`*

Командата изчислява значението на целочисления *`израз`* и го извежда на стандартния изход. *`Израз`* се конструира от константи (цели числа), променливи и аритметичните операции: +, -, *, / (цялата част при деление), % (остатъка при деление). Операциите + и – имат еднакъв приоритет, който е по-нисък от този на *, / и %. Изчислението се извършва от ляво на дясно, като скоби не могат да се използват. В израза се прави заместване на променливи и заместване на изхода преди изчислението.

Пример. Команда `expr`.

```
x=12
expr $x + 1
expr $x "*" 2
x=`expr $x + 1`
expr `who | wc -l` - 1
```

Друг начин за извършване на аритметични изчисления, който се реализира от по-новите версии на shell е така нареченото аритметично разширение (Arithmetic Expansion). Тази конструкция има стар и нов синтаксис, които са съответно:

`$[`*`израз`*`]` и `$((`*`израз`*`))`

Изчислява се целочисления *израз* и резултатът замества конструкцията. Преди да се изчисли израза се извършва заместване на променливи и заместване на изхода. Конструкциите аритметичното разширение могат да се влагат. Изчисленията се извършват в `long int`. Изразът се конструира от константи, променливи, аритметичните операции и скоби. Операциите и приоритетът им е както в езика С. Значенията на променливите се заместват в израза независимо, дали пред името има или не символ „$”. Константите могат да се записват като десетични, осмични или шестнадесетични цели числа в стила на езика Си.

Пример. Аритметично разширение.

```
x=12
echo $(($x+1))  # echo $[$x+1]
echo $(($x*2))
x=$[$x+1]
x=$(($x+1))     # x=$((x+1))
: $((x=x+1))    # : $((x+=1))
echo $((`who | wc –l` - 1))
```

Командни процедури и аргументи

Командната процедура (команден файл, shell файл, shell script) е текстов файл, съдържащ конструкции на командния език на shell, следователно това е програма на командния език, която се интерпретира от shell. Командата, която завършва изпълнението на командна процедурата е:

```
exit [n]
```

където *n* е кода на завършване, който процедура, аналогично на командите, връща. Когато процедурата завърши без команда `exit` (или с `exit,` но без аргумент), то тя връща кода на последната изпълнена команда.

Коментар в командна процедура е всичко от символа “#” до края на реда (освен ако символът е екраниран или ако е в комбинацията “#!”, която е в началото на първия ред на файла). Символът за начало на коментар се екранира, както и другите метасимволи.

Пример. Командната процедура изчислява и извежда броя на сесиите в системата. Записана е като файл в текущия каталог с име `nsession`.

```
# nsession
#-------------------------------------------------
date
n=`who | wc -l`
echo "Number of sessions: $n"
#-------------------------------------------------
```

За да бъде извикана за изпълнение една командна процедура така както се вика външна команда е необходимо:

1. Да се дадат права на потребителите да изпълняват файла, например с командата:

```
$ chmod a+x nsession
```

2. Файлът да е в каталог от списъка в променливата `PATH`.

Ако са изпълнени и двете условия, командната процедура се извиква като команда:

```
$ nsession
```

Ако файлът не е в каталог от PATH, то при извикване трябва да се задава пълно име на файла – абсолютно или относително:

`$` **`/home/student/nsession`** или **`./nsession`**, ако е в текущия каталог.

Друг начин за изпълнение на командна процедура е да се извика shell и да му се предаде като първи аргумент името на файла с командната процедура:

```
$ bash nsession
```

При този начин файлът се търси в каталозите от PATH, ако не е зададен с пълно име, но не е необходимо потребителят да има право за изпълнение. Този метод е удобен при тестване на командни процедури, тогава shell може да се извика с опция `-x` или `-v`. И още един начин за извикване е:

```
$ . nsession
```

Командната процедура се изпълнява от текущия процес shell. Файлът се търси в каталозите от PATH, ако не е зададен с пълно име.

В командна процедура могат да се използват ***позиционни параметри*** (Positional parameters). Значения им се присвояват при извикване на процедурата чрез аргументите в командния ред или с команда `set`, но не и с оператор за присвояване. Имената на позиционните параметри са:

```
$0, $1, $2, $3, $4, ... ${10}, ...
```

Нулевият параметър `$0` съдържа името на файла с командната процедура, а останалите съответстват позиционно на действителните аргументи, зададени при извикване на процедурата след името на файла. Ако при извикване не са зададени необходимия брой действителни аргументи, то съответните променливи `${i}` ще имат значение празен низ.

Пример. Командната процедура е с един аргумент, който трябва да е име на потребител, и изчислява и извежда броя на сесиите на потребителя.

```
# userlog user_name
#-------------------------------------------------
n=`who | grep "^$1 " | wc -l`
echo "User $1: $n sessions"
#-------------------------------------------------
```

Съществува още една група от променливи, които се поддържат от shell, могат да се използват от потребителя, но само shell променя значението им. Ще ги наричаме ***вътрешни променливи***:

Име	Значение
`$#`	Брой аргументи, предадени на shell без нулевия.
`$*`, `$@`	Всички аргументи, предадени на shell без нулевия, разделени с интервал.
`$?`	Код на завършване на последната изпълнена команда в привилегирован режим.
`$$`	Идентификатор (pid) на процеса shell.
`$!`	Идентификатор (pid) на последния изпълнен фонов процес.
`$_`	Последния аргумент на последната изпълнена команда.

Когато се използват в командна процедура, променливите `$#`, `$*` и `$@` се отнасят до аргументите, предадени при извикването й. Има разлика между `$*` и `$@` :

`"$*"` е еквивалентно на `"$1`*c*`$2`*c*`$3`*c*`…"`, където *c* е първия символ от значението на променливата `IFS` или интервал

`"$@"` е еквивалентно на `"$1" "$2" "$3"` …
Тази разлика се проявява при наличие на аргументи, съдържащи интервали в себе си, и когато върху променливите не се извършва така нареченото действие word splitting. Word splitting представлява заместването на всяка последователност от символи от променливата `IFS` с един интервал. Не се прави при заграждането на низ в двойни кавички. Следващият пример илюстрира тези различия.

Пример. Командната процедура илюстрира разликата между променливите `$*` и `$@`. В коментар е показан един резултат от изпълнението ѝ.

```
# arglist arg1 arg2 ...
#-------------------------------------------------
index=1
echo "Listing of arguments with \"\$*\":"
for a in "$*"
do
     echo "Arg $index = $a"
     index=$(( index + 1 ))
done
echo
index=1
echo "Listing of arguments with \"\$@\":"
for a in "$@"
do
     echo "Arg $index = $a"
     index=$(( index + 1 ))
done
echo
index=1
echo "Listing of arguments with \$*:"
for a in $*
do
     echo "Arg $index = $a"
     index=$(( index + 1 ))
done
echo
index=0
echo "Listing of arguments with \$@:"
for a in $@
do
     echo "Arg $((index+=1)) = $a"
done
exit 0
#-------------------------------------------------
#  arglist "aa nn" bb cc
#-------------------------------------------------
# Listing of arguments with "$*":
# Arg 1 = aa nn bb cc
#
# Listing of arguments with "$@":
# Arg 1 = aa nn
# Arg 2 = bb
# Arg 3 = cc
#
# Listing of arguments with $*:
# Arg 1 = aa
```

```
# Arg 2 = nn
# Arg 3 = bb
# Arg 4 = cc
#
# Listing of arguments with $@:
# Arg 1 = aa
# Arg 2 = nn
# Arg 3 = bb
# Arg 4 = cc
#-------------------------------------------------
```

Друг начин за присвояване на значения на позиционните параметри и на вътрешните променливи `$*`, `$@` и `$#` е чрез командата `set`.

Пример. Командната процедура илюстрира този начин за присвояване.

```
# argset arg1 arg2 ...
#-------------------------------------------------
echo "Positional parameters before set"
echo "Number: $#"
index=1
for a
do
     echo "Arg $index = $a"
     index=$(( index++ ))
done
echo
set `date`
echo "Positional parameters after set"
echo "Number: $#"
index=1
for a
do
     echo "Arg $index = $a"
     index=$(( index++ ))
done
exit 0
#-------------------------------------------------
```

Върху значенията на позиционните параметри влияе командата `shift`.

```
shift [n]
```

Измества значенията на аргументите с *n* позиции наляво, т.е. към малките номера, а значенията на първите *n* аргумента се губят. По премълчаване *n* е 1. Променят се съответно и значенията на променливите `$*`, `$@` и `$#`. Например, при изпълнение на:

```
shift
```

се извършва следното изместване `$2->$1, $3->$2`, и т.н. Старото значение в `$1` се губи, а `$0` не участва в изместването.

2.2.3 Оператори

Условен оператор и команди за проверка на условия

```
if списък_команди1
then списък_команди2
[elif списък_команди3
then списък_команди4] …
[else списък_команди5]
fi
```

В условните конструкции на командния език *списък_команди1* и *списък_команди3* са произволни списъци от команди, които тук изпълняват и ролята на условие. Когато след като се изпълни списъкът от команди върне код на завършване 0, условието е истина, а ако върне код различен от 0 е лъжа. Следователно, изпълнява се *списък_команди1* и ако върне код 0 се изпълнява *списък_команди2*, иначе се продължава с клона `elif`, ако го има, или `else` ако го има.

Пример. Условен оператор.

```
if cmp –s f1 f2
then echo "Files f1 and f2 are identical"
else echo "Files f1 and f2 differ"
fi
```

```
if cd dir 2>/dev/null
then echo "Now in dir"
else echo "Can't change to dir"
fi
```

```
if grep "^student:" /etc/passwd >/dev/null 2>&1; then :
else echo "No user student"
fi
```

```
if who | grep "^student "
then :
else echo "User student is not working"
fi
```

Код на завършване на оператор `if` е кода на последната изпълнена команда в `then` или `else` списъка, или е 0 ако нито едно условие не е истина и няма `else`. Операторите `if` могат да се влагат.

Много често в `if` се използва командата `test` и синонима й [, която проверява различни условия и връща код 0 при истина, и 1 при лъжа.

`test` *условие* или `[` *условие* `]`

Условието, което се проверява включва проверки за файлове, сравнения на низове и сравнения на числови низове, които могат да се комбинират с логическите операции, които в ред на намаляване на приоритета са:

! (логическо NOT), `-a` (логическо AND), `-o` (логическо OR)

- Условия за файлове

-e *файл*	истина, ако *файл* съществува
-f файл	истина, ако *файл* съществува и е обикновен файл
-d *файл*	истина, ако *файл* съществува и е каталог
-c *файл*	истина, ако *файл* съществува и е символен специален файл

-b *файл* истина, ако *файл* съществува и е блоков специален файл
-L *файл* истина, ако *файл* съществува и е символна връзка
-s *файл* истина, ако *файл* не е празен, размерът му е по-голям 0
-r *файл* истина, ако *файл* е достъпен за четене за потребителя, изпълняващ test
-w *файл* истина, ако *файл* е достъпен за писане за потребителя, изпълняващ test
-x *файл* истина, ако *файл* е достъпен за изпълнение за потребителя, изпълняващ test
файл1 -nt *файл2* истина, ако *файл1* е по-нов от *файл2* според датата на последно изменение
файл1 -ot *файл2* истина, ако *файл1* е по-стар от *файл2* според датата на последно изменение

- Условия за низове

низ1 = низ2 истина, ако *низ1* и *низ2* са еднакви
низ1 != *низ2* истина, ако *низ1* и *низ2* са различни
[-n] *низ1* истина, ако *низ1* не е празен
-z *низ1* истина, ако *низ1* е празен

- Условия за сравнение на числа

n1 –eq *n2* истина, ако *n1* и *n2* са равни

Другите операции за сравнение на числа са : `-ne, -lt , -le, -gt, -ge`.

Преди да се провери условието в него се прави заместване на променливи, заместване на изхода и аритметично разширение.

Пример. Команда `test`.

```
if test –e file; then
      echo "File exists"
fi
```

```
dir=/home/student/proc
if [ -d $dir –a –r $dir ]; then
      echo "You have read permission on directory $dir"
fi
```

В новите версии на bash (от bash2.02) е въведена и разширена команда за проверка на условия, която има вида:

```
      [[ условие ]]
```

Разширението се състои в добавянето на скоби, нови логически операции: && (логическо AND) и || (логическо OR) и нови операции за сравнение на низове:
низ1 == низ2 еквивалентно на операцията =
низ1 < низ2 истина, ако *низ1* е преди *низ2* според ASCII наредбата
низ1 > низ2 истина, ако *низ1* е след *низ2* според ASCII наредбата

Пример. Разширена команда `test`.

```
if [[ -d $dir && –r $dir ]]; then
      echo "You have read permission on directory $dir"
fi
```

Пример. Нов вариант на командната процедура, която изчислява и извежда броя на сесиите на потребител, чието име е зададено като аргумент.

```
# userlog2 user_name
#-----------------------------------------------
if [ $# -eq 0 ]
then echo "usage: $0 user_name" >&2; exit 1
fi
if grep "^$1:" /etc/passwd > /dev/null 2>&1
then
     n=`who | grep "^$1 " | wc -l`
     echo "User $1: $n sessions"
else echo "No user $1" >&2; exit 2
fi
exit 0
#-----------------------------------------------
```

Оператори за цикъл

В операторите `while` и `until` итерациите на цикъла се управляват от условие. Условието, както и при оператора `if`, е списък от команди. Ако списъкът върне код 0 условието се счита за истина, иначе е лъжа. Тялото на цикъла също е списък от команди между ключовите думи `do` и `done`.

```
while списък_команди1
do списък_команди2
done
```

Цикълът `while` се изпълнява дотогава докато условието е истина. Условието се проверява в началото на итерацията, което означава, че *списък_команди1* се изпълнява поне веднаж и когато върне код различен от 0, управлението се предава на командата след `done`.

```
until списък_команди1
do списък_команди2
done
```

Цикълът `until` се изпълнява дотогава докато условието е лъжа. Условието също се проверява в началото на цикъла, т.е. *списък_команди1* се изпълнява поне веднаж и когато върне код 0, управлението се предава на командата след `done`. Код на завършване на оператор `while` и `until` е кода на последната изпълнена команда в тялото на цикъла *списък_команди2*, или е 0 ако тялото на цикъла не е изпълнено нито веднаж.

Пример. Командната процедура е с един аргумент, който е име на потребител. Тя чака включването на потребителя в системата и му изпраща директно съобщение.

```
# waitu user_name
#-----------------------------------------------
if [ $# -eq 0 ]
   then echo "usage: waitu user_name" >&2; exit 1
fi
until who | grep "^$1 " >/dev/null
do
   echo "waiting for $1"; sleep 30
done
echo "User $1 is working"
write $1 << !
Hello
```

```
!
exit 0
#-------------------------------------------------
```

В оператор `for` итерациите на цикъла се управляват от променлива. Фразата `in` определя значенията, които се присвояват на променливата. За всяко значение се изпълнява тялото на цикъла, което е *списък_команди1*.

```
for променлива [in дума …]
do списък_команди1
done
```

В *дума* се извършва заместване на променливи, заместване на изхода, аритметично разширение и разширение до имена на файлове (т.е. интерпретират се метасимволите $, `, *****, ?, []), в резултат на което се получава списък от елементи. След това на променливата се присвоява всеки един елемент и се изпълнява *списък_команди1*. Ако след като се извърши разширение на метасимволите във фразата `in` не се генерира нито една дума, то тялото на цикъла не се изпълнява нито веднаж. Ако фразата `in` не е зададена по премълчаване се подразбира `in "$@"`, т.е. цикълът се изпълнява по веднаж за всеки един позиционен аргумент. Код на завършване на оператор `for` е кода на последната изпълнена команда в тялото на цикъла или е 0 ако тялото не се изпълни нито веднаж.

Операторите за цикъл могат да се влагат. Върху изпълнението на цикъл оказват влияние две вътрешни команди, които могат да се използват само в тялото на цикъл.

```
break [n]
```

Прекратява изпълнението на съдържащите командата *n* вложени цикъла. Ако *n* е по-голямо от броя на вложените цикли, съдържащи `break`, то се прекратяват всичките. По премълчаване значението на *n* е 1.

```
continue [n]
```

Започва нова итерация на *n*-тия цикъл, броейки отвътре навън вложените цикли, съдържащи командата. По премълчаване *n* е 1. Кодът на завършване на `break` и `continue` е 1, ако не се съдържат в цикъл, иначе е 0.

Пример. Командната процедура изпълнява цикъл с 10 итерации по два начина.

```
# Two ways to count up to 10
#-------------------------------------------------
echo "With for"
for i in 1 2 3 4 5 6 7 8 9 10
do
     echo $i
done
echo "With while"
i=1
while [ "$i" -le 10 ]
do
     echo $i
     i=`expr $i + 1`  # i=$(($i+1)) or i=$((i+1))
done
#-------------------------------------------------
```

Пример. Командната процедура е с един аргумент - име на каталог, който по премълчаване е текущия каталог. Извежда пълните имена на всички подкаталози, съдържащи се в каталога. Следват два варианта - `ldir` и `ldir2`.

```
# ldir [ directory ]
# Lists directories in a directory (one level only)
#-------------------------------------------------
if test $# -ne 0
then if test -d "$1"
     then cd $1
     else echo "$1 is not directory" >&2; exit 1
     fi
fi
d=`pwd`
for i in *      # $(ls) or `ls`
do
    if test -d "$i"
    then echo $d/$i
    fi
done
exit 0
#-------------------------------------------------

# ldir2 [directory]
# Lists directories in a directory (one level only)
#-------------------------------------------------
if [ $# -eq 0 ]
then d=`pwd`
else
     if [ -d "$1"]
     then d=$1
     else echo "$1 is not directory">&2; exit 1
     fi
fi
for i in `ls $d`
do
    if [ -d $d/$i ]
    then echo $d/$i
    fi
done
exit 0
#-------------------------------------------------
```

Пример. Командната процедура е с един аргумент - име на каталог, който по премълчаване е текущия каталог. Извежда имената на всички подкаталози, съдържащи се в поддървото на каталога.

```
# listd [directory]
# Lists directories in the subtree of directory (including directory)
# with absolute or relative path name (depends on parameter)
#-------------------------------------------------------------
if [ $# -eq 0 ]
then d=.
else
     if [ -d "$1" ]
     then d=$1
     else echo "$1 is not directory" >&2; exit 1
```

```
     fi
fi
for i in `find $d -type d -print`
do
    echo $i
done
exit 0
#-------------------------------------------------------------
```

Пример. Друг вариант на решение на горната задача, в който се използва рекурсивно извикване на командната процедура.

```
# listd2 [directory]
#-------------------------------------------------
if test $# -eq 0
then d=.
else if test -d "$1"
     then d=$1
     else echo "$1 is not directory" >&2; exit 1
     fi
fi
echo $d
for i in `ls $d`
do
   if test -d $d/$i
   then listd2 $d/$i  # if listd2 in PATH, if not
   # /home/…/listd2 $d/$i
   fi
done
exit 0
#-------------------------------------------------
```

Пример. Още един вариант на решение, в който се използва рекурсивно извикване на командната процедура. Извежда винаги пълните имена на каталози, включително и на самия аргумент, независимо как е зададен.

```
# listd3 [directory]
#-------------------------------------------------
if test $# -eq 1
then if test -d "$1"
     then cd $1
     else echo "$1 not directory" >&2; exit 1
     fi
fi
echo `pwd`
for i in `ls`
do
   if test -d $i
   then listd3 $i  # if listd3 in PATH, if not
   # then /home/…/listd3 $i
   fi
done
exit 0
#-------------------------------------------------
```

Пример. Командната процедура е с един аргумент - име на каталог, който по премълчаване е текущия каталог. Изчислява и извежда имената на обикновените и достъпни за изпълнение файлове в каталога и накрая броя им.

```
# lfile [directory]
# Lists and counts regular/execute files in a directory (one level only)
#-------------------------------------------------
if [ $# -ne 0 ]
then if [ -d "$1" ]
     then cd $1
     else echo "$1 is not directory" >&2; exit 1
     fi
fi
count=0
for i in `ls`
do
    if [ -f $i -a -x $i ]
    then echo $i
      count=$((count + 1))
    fi
done
echo "Regular&execute: $count"
#-------------------------------------------------
```

Пример. Командната процедура може да се извиква с произволен брой аргументи, които са имена на потребители. За всеки аргумент изчислява и извежда броя на сесиите на съответния потребител или съобщение ако не съществува такъв потребител.

```
# userlogs user_name ...
#-------------------------------------------------
if [ $# -eq 0 ]
then echo "usage: userlogs name_name ..." >&2; exit 1
fi
for u
do
     if grep "^$u:" /etc/passwd >/dev/null 2>&1
     then
          n=`who | grep "^$u " | wc -l`
          echo "User $u: $n sessions"
     else
          echo "No user $u" >&2
     fi
done;
exit 0
#-------------------------------------------------
```

Пример. Друг вариант на решение, който използва командата `shift`.

```
# userlogs2 user_name ...
#-------------------------------------------------
if [ $# -eq 0 ]
then echo "usage: $0 name_name ..." >&2; exit 1
fi
while [ -n "$1" ]
do
     if grep "^$1:" /etc/passwd >/dev/null 2>&1
     then
          n=`who | grep "^$1 " | wc -l`
          echo "User $1: $n sessions"
     else
          echo "No user $1" >&2
```

```
      fi
      shift
done
exit 0
#-------------------------------------------------
```

Оператор case

```
case  дума  in
шаблон1 [ | шаблон2 ] …) списък_команди ;;
. . .
esac
```

В *дума* се извършва заместване на променливи, заместване на изхода и аритметично разширение (т.е. интерпретират се метасимволите: $, `). След това се търси съответствие на получената дума с шаблоните по реда на задаването им. Изпълнява се списъка команди при първото намерено съответствие и изпълнението на оператора завършва. В шаблоните могат да се използват метасимволите: *, ?, [], които се разширяват по същите правила, както при имена на файлове, но тук се разширяват до произволни думи. Оператор `case` връща кода на завършване на последната изпълнена команда или 0 ако не е намерено нито едно съответствие.

Пример. Фрагмент от командна процедура, който чете отговор на потребителя и го интерпретира като „Да”, „Не” или „Изход”.

```
read answer
case "$answer"  in
[Qq]*|[Ee]*) echo "Answer is Quit"; exit 1;;
[Yy]*| "") echo "Answer is YES";;
[Nn]*) echo "Answer is NO";;
*) echo "Invalid answer";;
esac
```

2.2.4. Функции

Функция с име *име_функция* и тяло *списък_команди* се определя по следния начин:

```
[function] име_функция () { списък_команди; }
```

Определението на функцията трябва да предшества извикването ѝ. Функция се извиква така както и обикновена команда, чрез името си. Тогава се изпълнява *списък_команди* в контекста на текущия процес shell, т.е. не се създава нов процес shell, както при изпълнение на командна процедура. Това означава, че всички променливи и техните значения, определени в текущия shell са достъпни и при изпълнение на функцията. Единствено позиционните параметри `$1`, `$2`, и т.н. и променливите `$#`, `$*`, `$@` се изменят по време на изпълнението на функцията от аргументите, зададени при извикването ѝ.

Във тялото на функцията може да се определят локални променливи чрез командата `local`. Опциите са същите, както на командата `declare`.

```
local [опции] променлива[=значение]
```

При завършване функцията връща код на завършване, който е кода на последната изпълнена команда в *списък_команди*, ако това не е команда `return`.

```
return [n]
```

Командата `return` явно завършва изпълнението на функцията и връща изпълнението към мястото на викането й. Аргументът *n* (число от 0 до 255) е код на завършване и ако не е указан се използва пак кода на последната изпълнена преди `return` команда. След завършване изпълнението на функцията се възстановяват значенията на временно изменените позиционни параметри и вътрешни променливи (`$1`, `$2`, и т.н., и `$#`, `$*`, `$@`) такива, каквито са били преди извикването. Променливата `$?` ще съдържа кода на завършване, върнат от функцията.

Пример. Командната процедура може да се извиква с произволен брой аргументи (вкючително и 0), които са имена на потребители. За всеки аргумент изчислява и извежда броя на сесиите на съответния потребител или съобщение ако не съществува такъв потребител. Ако е извикана без аргументи изчислява и извежда общия брой сесии в системата.

```
# userls [user_name . . .]
#-------------------------------------------------
function scnt()
{
if [ $# -eq 0 ]
then
     n=`who | wc -l`
else
     if grep "^$1:" /etc/passwd >/dev/null 2>&1
     then n=`who | grep "^$1 " | wc -l`
     else return 1
     fi
fi
return 0
}
if [ $# -eq 0 ]
then
     scnt
     echo "Number of sessions: $n"
else
     for u
     do
          if scnt $u
          then echo "User $u: $n sessions"
          else echo "No user $u" >&2; fi
     done
fi
#-------------------------------------------------
```

2.2.5. Екраниране

Под термина екраниране се разбира отменяне на специалното значение на символи или думи. Съществуват три начина за това.

- Чрез символа \ (escape character)

Ако не е екраниран символът \ запазва нормалното значение на символа след него, освен ако това не е символ за нов ред. Комбинацията \<newline> се интерпретира като игнориране на символа за нов ред, т.е. като продължение на реда. Използва се за запис на команда на няколко реда (последния пример). Символите $ и > в началото на реда са първичната и вторична покана.

```
$ echo \x              # x
$ echo \\x             # \x
$ echo \\\x            # \x
$ echo xxx\            # xxxzzz
> zzz
```

- Чрез двойни кавички - "*низ*"

Запазва се нормалното значение на всички символи в низа, с изключение на $, `, \. Чрез символа \ в низа се екранират само символите $, `, \, " и <newline>.

```
dd=123
$ echo "\x"                      # \x
$ echo "\\x"                     # \x
$ echo "\\\x"                    # \\x
$ echo "aa\xbb\"cc$dd"           # aa\xbb"cc123
$ echo "aa\xbb\"cc\$dd"          # aa\xbb"cc$dd
$ echo "xxx\                     # xxxzzz
> zzz"
```

- Чрез единични кавички - ‘*низ*’

Запазва се нормалното значение на всички символи в низа.

```
$ echo '\x'            # \x
$ echo '\\x'           # \\x
$ echo '\\\x'          # \\\x
```

2.2.6. Заместванията и изпълнение на команда

Преди да предаде управлението на команда shell изпълнява определена последователност от действия:

1. Анализира синтаксиса на командата в съответствие със собствените си синтактически правила.
2. Извършва заместванията, а именно:
   - Заместване на "~"
   - Заместване на име на променлива със значението
   - Заместване на изхода на команда
   - Аритметично разширение
   - Word splitting
   - Разширение до имена на файлове
3. Извършва пренасочвания на вход и изход.
4. Изпълнява командата, като й предава аргументите след направените заместванията. При това, ако името на командата не е пълно име на файл, то проверява дали има:
   - Функция с указаното име
   - Вътрешна команда с указаното име
   - Търси файл в каталозите на променливата `PATH` с указаното име.

Изпълнява първата намерена команда. Ако името на командата е пълно име на файл - относително или абсолютно, тогава направо се опитва да изпълни програмата във файла. Ако не намери файла или намерения файл не е изпълним (потребителят няма право `x`), извежда съобщение за грешка и завършва изпълнението на командата с код съответно 127 или 126.

Трета глава

ФАЙЛОВИ СИСТЕМИ

Файлова система (подсистема) това е тази компонента на операционната система, която е предназначена да управлява постоянните обекти данни, т.е. обектите, които съществуват по-дълго отколкото процесите, които ги създават и използват. Постоянните обекти данни се съхраняват на външна памет (диск или друг носител) в единици, наричани файлове.

Файловата система трябва да:

- осигурява операции за манипулиране на отделни файлове, като например create, open, close, read, write, delete и др.
- изгражда пространството от имена на файлове и да предоставя операции за манипулиране на имената в това пространство.

Какво съдържа пространството от имена? Каква е организацията на файловата система? Какви са операциите, наричани ***системни примитиви, системни функции, системни извиквания*** или ***system calls***, осигурявани от файловата система? Това са въпроси, които вълнуват потребителите и се разглеждат в първите два раздела. Останалите раздели на тази глава са посветен на проблеми при физическата реализация на файлова система и подходи за тяхното решаване. Като примери са използвани файловите системи на операционните системи Unix, Linux, MSDOS и Windows NT, но се споменават и други.

3.1. ЛОГИЧЕСКА СТРУКТУРА НА ФАЙЛОВА СИСТЕМА

3.1.1. Имена и типове файлове

Най-важната характеристика на една абстракция за потребителите са правилата, по които се именуват обектите, в случая файловете. Най-често името на файл е низ от символи с определена максимална дължина, като в някои системи освен букви и цифри са разрешени и други символи. Много често името се състои от две части, разделени със специален символ, например ".". Втората част се нарича разширение на името и носи информация за типа или формата на данните, съхранявани във файла. Например, следните имена имат разширения, показващи типа на данните във файла.

```
file.c   - програма на Си
file.o   - програма в обектен код
file.exe - програма в изпълним код
file.txt - текстов файл
```

В някои операционни системи, като Unix, MINIX и Linux, такива разширения представляват съглашения, които се използват от потребителите и някои обслужващи програми (компилатори, свързващи редактори, текстови процесори и др.), но ядрото не ги налага и използва. В други операционни системи се реализира по-строго именуване. Например, няма да бъде зареден и изпълнен файл, ако името му няма разширение "`.exe`" или някое друго.

Какво включва пространството от имена или какви са типовете файлове? Преди всичко имена на ***обикновени файлове*** (***regular files***), съдържащи програми, данни, текстове или каквото друго потребителят пожелае. Но пространството от имена би могло да включва и имена на външни устройства, системни структури и услуги. Такова решение е прието в много от съвременните операционни системи. Външните устройства са специален тип файлове, наречени ***специални файлове*** (***character special и block special device file***) в Unix, Linux, MINIX и др. Системните примитиви на

файловата система са приложими както към обикновените файлове така и към специалните файлове. Следователно, всяка операция за четене или писане, осъществявана от програмата, е четене или писане във файл. Най-съществено предимство на този подход е, че позволява да се пишат програми, които не зависят от устройствата, тъй като действията, изпълнявани от програмата зависят само от името на файла.

Трябва да се съхранява информация за файловете във файловата система и за тази цел се използват специални системни структури, наричани ***каталог***, ***справочник***, ***директория***, ***directory***, ***folder***, ***VTOC***, които осигуряват връзката между името и данните на файла и реализират организацията на файловете. В много системи каталогът е тип файл с фиксирана структура на записите и съдържа по един запис за всеки файл.

И така, типовете файлове, поддържани от файловите системи на Unix, Linux, MINIX и др. са:

- обикновен файл
- каталог
- специален файл
- програмен канал, FIFO файл
- символни връзки

Типът ***програмен канал*** (***pipe***) и ***FIFO файл*** се използва като механизъм за комуникация между конкурентни процеси. Чрез ***символните връзки*** (***symbolic link, soft link, junction***) един файл може да има няколко имена, евентуално в различни каталози, или както се казва реализират се няколко връзки към файл. Този тип файл е един от механизмите за осигуряване на общи файлове за различните потребители. Друг начин за работа с общи файлове са ***твърдите връзки*** (***hard links***), но те не са тип файл, а са няколко имена на обикновен файл.

Тясно свързан с въпроса за типовете файлове е вътрешната структура на файл. Файлът най-често представлява последователност от обекти данни. Какви са тези обекти данни? Възможни са два отговора - запис или байт:

Файлът е последователност от записи с определена структура и/или дължина. Основното в този подход е, че всеки системен примитив `read` или `write` чете или пише един запис. Такъв подход е използван в DOS/360, OS/360, CP/M.

Другата възможност, реализирана в много от съвременните операционни системи, Unix, Linux, MINIX, MSDOS, Windows и др., е последователност от байтове. Това е структурата на файл, реализирана от файловата система. Всяка по-нататъшна структура на файла може да се реализира от програмите на потребителско ниво, като операционната система не помага за това, но и не пречи. Този подход позволява разработката на прости системни примитиви, които освен това да са приложими както за обикновени файлове, така и за останалите типове файлове.

Всеки файл има име и данни. В допълнение операционната система може да съхранява и друга информация за файла, която ние ще наричаме ***атрибути*** на файла. Атрибутите, реализирани в различните системи се различават, но един списък от възможни атрибути е показан в *Таблица 3.1*.

Таблица 3.1. Някои възможни файлови атрибути

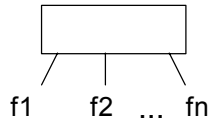
Размер	Текущия размер на файла в байтове, блокове или записи
Време на създаване	Дата и време на създаване на файла
Време на достъп	Дата и време на последен достъп до файла
Време на изменение	Дата и време на последно изменение на файла
Собственик	Потребителят, който е текущия собственик на файла
Права на достъп	Кой и по какъв начин може да осъществява достъп до файла

Парола за достъп	Парола за достъп до файла
Флагове:	
Read-only флаг	1 - само за четене, 0 - за четене и писане
Hidden флаг	1 - не се вижда от командите, 0 - видим за командите
System флаг	1 - системен файл, 0- нормален файл
Archive флаг	1 - трябва да се архивира, 0 - не е променян след архивирането
Secure deletion	При унищожаване на файла блоковете, заемани от него, се форматират

3.1.2. Каталози и организация на файловата система

Вторият основен въпрос, засягащ външния вид на файловата система е: Колко каталога има и ако са повече от един каква е организацията на системата от каталози? Каталогът съдържа по един запис (елемент) за всеки файл, който съдържа като минимум името на файла. Освен това може да съдържа атрибутите на файла и дисковите адреси на данните на файла или указател към друга структура, където се съхраняват дисковите адреси на данните и евентуално атрибутите на файла.

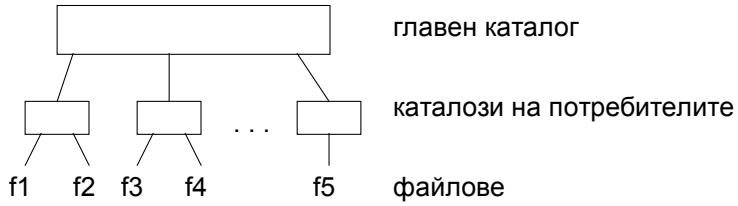
Като правило файловата система е разположена върху няколко носителя и включва файлове, принадлежащи на различни потребители. Най-простото решение, което ние ще наречем еднокаталогова файлова система, е следното. На всеки носител има по един каталог, който съдържа информация за всички файлове върху носителя (*Фиг.3.1*).



Фиг. 3.1. Еднокаталогова файлова система

Такава е организацията в някои ранни операционни системи, като DOS/360, или в по-нови, но примитивни, предназначени за персонални компютри системи, като например CP/M. Преимущество на тази организация е простотата и възможността за бързо търсене на файл по името му.

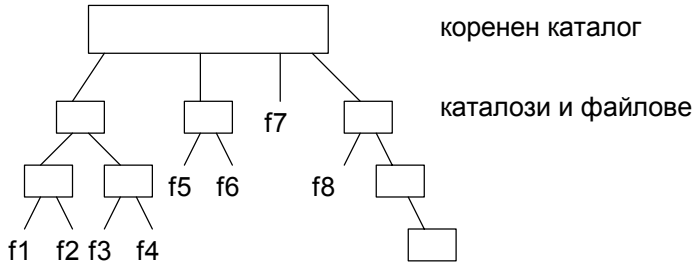
В съвременните, главно многопотребителски операционни системи, горният подход не е приемлив, най-малкото поради опасността от конфликт между еднакви имена на файлове, давани от различни потребители. В една от първите интерактивни системи, CTSS (Compatible Time-Sharing System), разработена в MIT, е реализирана следната организация (*Фиг.3. 2*).



Фиг. 3. 2. Йехархична структура с фиксиран брой нива

Всеки потребител има свой каталог, който съдържа записи за всички файлове на потребителя. Има главен каталог, който съдържа по един запис за каталозите на потребителите. Тогава имената, които потребителят дава на своите файлове, трябва да са уникални в рамките на неговия каталог. Логическата структура тук представлява йерархия с фиксиран брой нива.

Следващата стъпка към подобряване на организацията е да се обобщи файловата структура до дървовидна или йерархична структура с произволен брой нива (*Фиг.3.3*).



Фиг. 3.3. Йерархична файлова система

Вътрешните възли на дървото трябва да са каталози, а листата могат да бъдат каталози, обикновени файлове, специални файлове и други типове фалове, ако се поддържат такива. За потребителя каталогът представлява група от файлове и каталози (подкаталози). Този подход, използван при файловите системи на повечето съвременни операционни системи Unix, Linux, MSDOS, MINIX, Windows и др., дава възможност за естествено и удобно групиране на файловете, отразяващо предназначението и формата на данните, съхранявани в тях.

Всеки връх в дървото, каталог или друг тип файл, има име, което потребителят избира да е уникално в рамките на съдържащия го каталог и което ние ще наричаме ***собствено име***. Тогава всеки файл ще има име, което уникално го идентифицира в рамките на цялата йерархична структура и ние ще го наричаме ***пълно име***. Използват се два начина за формиране на пълно име.

- ***абсолютно пълно име*** (***absolute path name***)

Всеки файл или каталог притежава едно абсолютно пълно име, което съответства на единствения път в дървото от корена до съответния файл или каталог.

- ***относително пълно име*** (***relative path name***)

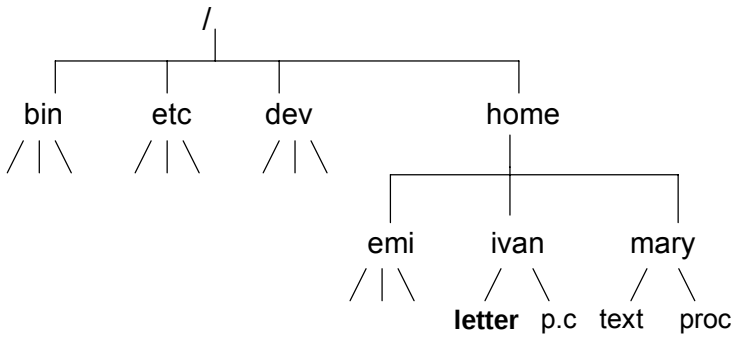
Този начин за формиране на пълно име е свързан с понятието ***текущ или работен каталог*** (***current/working directory***). Във всеки един момент от работата на потребителя със системата, той е позициониран в един от каталозите на дървото. Операционната система предоставя средства за избор на текущ каталог. Относителното пълно име на файл или каталог съответства на пътя в дървото от текущия в даден момент каталог до съответния файл или каталог. Като в този случай е разрешено и движение нагоре по дървото.

Пълното име на файл, абсолютно или относително, се състои от компоненти - собствените имена на каталозите в пътя и на самия файл, разделени със специален символ-разделител, като напр. ">", "\", "/". Движението нагоре по дървото се записва чрез специално име, което обикновено е "..". Ако първият символ в пълното име е символа-разделител, това означава, че името е абсолютно пълно. Всяко име, което не започва със символа-разделител, се приема за относително пълно или собствено име. За илюстрация да разгледаме една типична йерархия на файлова система в Unix или Linux, показана на *Фиг.3.4*.

Абсолютно пълно име на файл е `/home/ivan/letter`. Ако текущ каталог е `/home`, то относителното пълно име на същия файл е `ivan/letter`. Ако текущ каталог е `/home/emi`, то относителното пълно име на същия файл е `../ivan/letter`. Следователно, ако от текущ каталог `/home/emi` искаме да създадем копие на файла под име `copy_letter`, имаме различни възможности за именуване на оригиналния файл и копието:

```
$ cp /home/ivan/letter /home/emi/copy_letter
$ cp /home/ivan/letter copy_letter
```

```
$ cp ../ivan/letter  copy_letter
```



Фиг. 3.4. Файлова система в Unix или Linux

В *Таблица 3.2* е направено сравнение на файловите системи в Unix, Linux и MSDOS по основните характеристики на логическата структура.

Таблица 3.2. Сравнение между файловите системи на Unix, Linux и MSDOS

	UNIX и LINUX	**MSDOS**
1. Имена на файлове	До 14 или 255с., може с няколко разширения. Прави се разлика между малки и главни букви.	До 8с. първа част на името и до 3с. разширение, без разлика между малки и главни букви.
2. Типове файлове	Обикновени, каталози, специални и др., като имената им са вградени в структурата на ФС.	Обикновени, каталози и специални, но имената на специалните не са вградени в структурата на ФС.
3. Структура на ФС	Йерархична	Йерархична
Монтиране на ФС	Да - единна йерархия	Не - няколко йерархии; имена на устройства и текущо устройство
Коренен каталог	/	\
Текущ каталог	Да - за всеки процес	Да - за всяко устройство
Абсолютно и относително пълно име	Да - с разделител "/"	Да - с разделител "\"
4. Атрибути за защита	Собственик, група и права на достъп	Флагове: read-only, hidden, system
5. Много имена на файл	Да - твърди и символни връзки	Не

3.2. СИСТЕМНИ ПРИМИТИВИ ЗА РАБОТА С ФАЙЛОВЕ

Системните примитиви (системни функции, системни извиквания, system calls) реализират операциите, които файловата система предоставя на потребителите за:

- работа с отделни файлове;
- изграждане на структура на файловата система;
- защита на файловата система.

Ще разгледаме няколко основни системни примитиви за работа с файлове по стандарта POSIX, който е реализиран в повечето Unix системи - UNIX System V, 4.3BSD, SunOS, AIX, HP-UX и др, в Linux и MINIX. Повечето съвременни операционни системи предоставят системни примитиви, изпълняващи същите функции, макар да има различия в имената, броя аргументи и при реализацията им.

Основното проектно решение, което определя набора от операции над файл, е структурата на файла. За потребителя файлът представлява последователност от 0 или повече байта. Това е структурата на файла, реализирана от файловата система и съобразно тази структура са проектирани системните примитиви. Преди да разгледаме системните примитиви ще въведем някои необходими понятия, използвани при работа на файловата подсистема.

Файлов дескриптор (***file descriptor***) е неотрицателно цяло число, от 0 до 19 в по-ранните версии на Unix или до 63, 255 и повече в по-новите версии на Unix и в Linux, което се свързва с файл при отваряне и се използва за идентифициране на файла при последващата му обработка от процеса. Връзката между файловия дескриптор и файла се разрушава при затваряне на файла. Файловите дескриптори имат локално значение за всеки процес, което означава, че файлов дескриптор напр., 5 в два процеса най-вероятно е свързан с различни файлове. Прието е файловите дескриптори 0, 1 и 2 да се свързват със стандартно отворените файлове, които всеки процес получава, съответно стандартния вход, стандартния изход и стандартния изход за грешки. Стандартно тези три файлови дескриптора са свързани с един и същи специален файл, съответстващ на управляващия терминал на процеса. Но за ядрото на операционната система няма нищо специално в тези файлови дескриптори. Това е просто съглашение, което ако се приеме от всички потребителски програми, прави лесно пренасочването на стандартния им вход/изход/изход за грешки и свързването на програми в конвейер, с помощта на командния интерпретатор (спомнете си конструкциите на командния език, разгледани във втора глава).

С всяко отваряне на файл се свързва и указател на ***текуща позиция*** във файла (***file offset***, ***file pointer***), който определя позицията във файла, от която ще бъде четено или записвано и на практика стойността му е отместването от началото на файла, измерено в байтове. Този указател се зарежда, изменя или използва от повечето примитиви за работа с файлове.

С всяко отваряне на файл се свързва и ***режим на отваряне*** на файла, който определя начина на достъп до файла чрез съответния файлов дескриптор докато е отворен от процеса, напр., само четене, само писане, четене и писане. При всеки следващ опит за достъп до файла се проверява дали той не противоречи на режима на отваряне, и ако е така, достъпът се отказва независимо от правата на процеса.

Създаване на обикновен файл

```
#include <sys/types.h>
int creat(const char *filename, mode_t mode);
```

Примитивът `creat` създава нов файл с указаното име *`filename`* и код на защита *`mode`*. Ако такъв файл вече съществува, то старото му съдържание се

унищожава при условие, че процесът има право за писане (`w`). Процесът трябва да има право за позициониране (`x`) за всички каталози на пътя в пълното име и право `w` за родителския каталог. Създаденият файл се отваря за писане и функцията на `creat` връща файлов дескриптор свързан с файла. При грешка връща -1.

Отваряне на файл

```
#include <fcntl.h>
int open(const char *filename, int oflag [, mode_t mode]);
```

В по-ранните версии на Unix примитивът `open` е предназначен само за отваряне на съществуващ файл, т.е. създава връзка между процеса и указания файл, която се идентифицира чрез файлов дескриптор върнат от `open` и включва режима на отваряне на файла и текуща позиция във файла. След `open` текущата позиция сочи началото на файла. В по-новите версии на Unix и в Linux функционалността на `open` е разширена. Чрез `open` може да се създаде файл, ако не съществува и след това да се отвори. Действието на системния примитив се конкретизира чрез параметъра *`oflag`*. Значенията на *`oflag`* се конструират чрез побитово ИЛИ (|) от следните символни константи:

- Определящи начина на достъп (само един от тези три флага)

`O_RDONLY` - (0) четене
`O_WRONLY` - (1) писане
`O_RDWR` - (2) четене и писане

- Определящи действия, извършвани при изпълнение на `open`

`O_CREAT` - създаване на нов файл, ако не съществува
`O_EXCL` - (заедно с `O_CREAT`) връща грешка, ако файлът съществува
`O_TRUNC` - изменяне дължината на файла на 0, ако съществува

- Определящи действия, извършвани при писане във файла

`O_APPEND` - добавяне в края на файла при всяка операция писане
`O_SYNC` - синхронно обновяване на данните на диска при всяко извикване на системния примитив `write`.

Следователно, системният примитив `creat` е излишен (но е запазен заради съвместимост с по-ранните версии и може би за удобство), тъй като е еквивалентен на

```
open(filename, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

Един файл може да бъде отворен едновременно от няколко процеса и дори няколко пъти от един процес. При всяко отваряне процесът, изпълняващ `open`, получава нов файлов дескриптор, с който са свързани независими текуща позиция и режим на отваряне.

Затваряне на файл

```
int close(int fd);
```

Примитивът `close` затваря отворен файл, т.е. прекратява връзката между процеса и файла, като освобождава файловия дескриптор, който след това може да бъде използван при последващи изпълнения на `creat` или `open`. Връща 0 при успех или -1 при грешка.

Четене и писане във файл

```
#include <sys/types.h>
ssize_t read(int fd, void *buffer, size_t nbytes);
```

Примитивът `read` чете *`nbytes`* последователни байта от файла, идентифициран чрез файлов дескриптор *`fd`*, като започва четенето от текущата позиция. Прочетените байтове се записват в област на процеса, чийто адрес е зададен в *`buffer`*. Указателят на текуща позиция се увеличава с действителния брой прочетени байта, т.е. сочи байта след последния прочетен байт. Връща действителния брой прочетени байта, който може да е по-малък от *`nbytes`* ако до края на файла има по-малко байта, 0 ако текущата позиция е след края на файла или -1 при грешка.

```
#include <sys/types.h>
ssize_t write(int fd, void *buffer, size_t nbytes);
```

Примитивът `write` има същите аргументи. *`Nbytes`* байта се предават от областта на процеса с адрес *`buffer`* към файла, идентифициран с файлов дескриптор *`fd`*. Аналогично на `read`, мястото на началото на писане във файла се определя от текущата позиция, като след завършване на обмена текущата позиция се увеличава с броя записани байта, т.е. се премества след последния записан байт. За разлика от `read`, `write` може да пише и в позиция след края на файла, при което се извършва увеличаване размера на файла. Ако е вдигнат флаг `O_APPEND` при `open`, то при всяко изпълнение на `write` текущата позиция първо се установява в края на файла и след това се пише. Ако е вдигнат флаг `O_SYNC` при `open`, то `write` връща управлението след физическото записване на данните и управляващата информация на диска. Връща действителния брой записани байта или -1 при грешка.

Алгоритмите на `read` и `write` са подобни. Всеки `read` и `write` се реализира като неделима операция. Докато не завърши изпълнението на определен `read` или `write`, друг процес не може да осъществи достъп до същия файла. Това е важно когато няколко процеса са отворили и работят едновременно с един и същи файл. Различието в алгоритмите на `read` и `write` е в следното:

- При опит да се чете от текуща позиция след края на файла, `read` връща 0, а `write` записва данните и увеличава размера на файла, като ако се наложи разпределя нов дисков блок.

- Ако `write` пише байтове, които заемат част от блок, то първо чете блока от диска, след което изменя съответната част в системния буфер.

- При `write` се използва подход наречен отложен запис (delayed write), т.е. данните се изменят в системния буфер, но при завършване на `write` физическото записване на диска може да не е извършено. За да сме сигурни, че записът наистина е извършен трябва да се използва флаг `O_SYNC` при `open`.

Изпълнявайки `read` или `write` в цикъл след `open` може последователно да се прочете или запише файл, т.е. системните примитиви, разгледани до тук осигуряват последователен достъп до файл.

Позициониране във файл

Примитивът `lseek` премества указателя на текуща позиция на произволна позиция във файла или след края му. Използването на `read` или `write` съвместно с `lseek` осигурява произволен достъп до файл.

#include <sys/types.h>

off_t lseek(int *fd*, off_t *offset*, int *flag*);

Параметърът *offset* задава отместването, с което текущата позиция ще се промени, а *flag* определя началото, от което се отчита отместването: 0 - от началото на файла, 1 - от текущото значение на указателя, 2 - от края на файла, т.е. новото значение на текущата позиция във файла (fp) се изчислява в зависимост от значението на *flag* по следния начин:

```
0 (SEEK_SET)    fp = offset
1 (SEEK_CUR)    fp = fp + offset
2 (SEEK_END)    fp = file_size + offset
```

Значението в *offset* може да е положително или отрицателно число.

При изпълнението на примитива lseek не се изисква достъп до диска. Изменя се единствено полето за текуща позиция, което е в паметта. Ако новото значение е след края на файла, то размерът на файла не се увеличава. Това ще стане по-късно, при изпълнение на последващия write. Ако за ново значение се получи отрицателно число, то това се счита за грешка и текущата позиция не се изменя. При успех lseek връща новото значение на текущата позиция, а при грешка -1.

Тъй като при успех lseek връща новото значение на текущата позиция, то можем да използваме lseek за да определим текущата позиция без да я изменяме:

```
off_t curpos;
curpos = lseek(fd, 0, SEEK_CUR);
```

Този начин на извикване можем да използваме и за проверка дали файлът позволява позициониране. Някои типове файлове, като програмните канали, специалния файл за терминал, не позволяват позициониране чрез lseek и тогава примитивът връща -1.

Ако новото значение на текущата позиция след изпълнение на lseek е на разстояние след края на файла, то последващия write ще започне писането от текущата позиция и ще увеличи размера на файла. Така може да се създаде „файл с дупка”. При четене всички байтове от дупката са нулеви (\0). Впоследствие там може да се запише нещо различно.

Пример. Програмата създава „файл с дупка”.

```
/* ------------------------------------------------------------ */
#include <stdio.h>
#include <fcntl.h>
char buf1[] = "ABCDEF";
char buf2[] = "abcdef";

main(void)
{
  int fd;

  if ((fd = open("file.hole", O_WRONLY|O_CREAT|O_TRUNC, 0640)) < 0){
          fprintf(stderr, "can't create file\n");
          exit(1); }
  if (write(fd, buf1, 6) != 6){
          fprintf(stderr, "write buf1 error\n");
          exit(1); }
  if (lseek(fd, 10, SEEK_CUR) == -1){
```

```
            fprintf(stderr, "lseek error\n");
            exit(1); }
  if (write(fd, buf2, 6) != 6){
            fprintf(stderr, "write buf2 error");
            exit(1); }
  exit(0);
}
/* ------------------------------------------------------------ */
```

Като изпълним програмата и след това проверим резултата с командите `ls` и `od`, ще получим следните резултати:

```
$ ls -l file.hole
-rw-r-----   1 moni     moni           22 Apr 12 05:58 file.hole
$ od -c file.hole
0000000   A   B   C   D   E   F  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020   a   b   c   d   e   f
0000026
```

Използваме командата `od` с аргумент `-c`, за да видим съдържанието на файла по символи. От изхода се вижда, че десетте незаписани байта в средата се четат като символ `\0`.

Информация за файл

Част от съхраняваната за файла информация (атрибутите) може да се получи чрез примитивите `stat` и `fstat`.

```
#include <sys/stat.h>
int stat(const char *filename, struct stat *sbuf);
int fstat(int fd, struct stat *sbuf);
```

Разликата между `stat` и `fstat` се състои в начина на указване на файла - в `stat` чрез *`filename`* се задава име на файла, а в `fstat` чрез *`fd`* се задава файловия дескриптор на отворения файл. Примитивът `fstat` е полезен при работа с наследени отворени файлове, чиито имена може да са неизвестни на процеса. Вторият аргумент *`sbuf`* е указател на структура `stat`, в която примитивът записва информацията за файла. Полетата на структурата съдържат атрибути на файла, като: тип на файл, код на защита, собственик и група на файла, размер, дата и време на последен достъп, последно изменение и др.

Пример. Програмата `copy` създава ново копие на съществуващ обикновен файл.

```
/* ------------------------------------------------------------ */
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#define BUFS 1024

main(int argc, char *argv[])
{
int fdr, fdw, n;
struct stat sbuf;
char buff[BUFS];

  if (argc != 3) {
      fprintf(stderr, "usage: copy from_file to_file\n");
      exit(1); }
```

```
  if ( stat(argv[1], &sbuf) == -1) {
      fprintf(stderr, "copy: %s: No such file\n", argv[1]);
      exit(1); }
  if ( ! S_ISREG(sbuf.st_mode) ) {
      fprintf(stderr, "copy: %s: Not a regular file\n", argv[1]);
      exit(1); }
  if (( fdr = open(argv[1], O_RDONLY)) == -1) {
      fprintf(stderr, "copy: %s: can't open\n", argv[1]);
      exit(1); }

  if (( fdw = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, 0644)) == -1) {
      fprintf(stderr, "copy: %s: can't create\n", argv[2]);
      exit(1); }

  while (( n = read(fdr, buff, BUFS)) > 0)
      write(fdw, buff, n);
  exit(0);
}
/* ------------------------------------------------------------ */
```

Програмата трябва да се извиква с два аргумента - името на копирания файл се задава като първи аргумент, а името на създаваното копие, като втори. Например, ако се извика чрез командния ред:

$ **`copy fileA xyz`**

ще създаде файл с име `xyz`, който е копие на файл с име `fileA`. Ако преди изпълнението на `copy` съществува файл с име `xyz`, той ще бъде презаписан. Ако не съществува файл `fileA` или съществува, но не е обикновен файл, ще бъде изведено съобщение за грешка.

Пример. Програмата `typef` извежда на стандартния изход съдържанието на файлове.

```
/* ------------------------------------------------------------ */
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
struct stat sbuf;
int fdr, i;
char buf;

  if (argc < 2) {
      fprintf(stderr,"usage:typef file ...\n");
      exit(1); }

  for ( i=1; i<argc; i++ ) {
      if ( stat(argv[i], &sbuf) ==-1) {
         fprintf(stderr, "typef: %s: No such file\n", argv[i]);
         continue; }

      if ( ! S_ISREG(sbuf.st_mode) )  {
         fprintf(stderr, "typef: %s: is not a regular file\n", argv[i]);
         continue; }

      if (( fdr = open(argv[i], O_RDONLY)) == -1) {
         fprintf(stderr, "typef: %s: can't open\n", argv[i]);
         continue; }
```

```
      while ( read(fdr, &buf, 1) )
         write(1, &buf, 1);
      close(fdr);
  }
  exit(0);
}
/* ------------------------------------------------------------ */
```

Програмата се извиква с произволен брой (поне един) аргументи - имена на файлове. Например, ако се извика чрез командния ред:

$ **`typef fileA xyz abc`**

ще изведе на стандартния изход последователно съдържанието на файловете `fileA` , `xyz` и `abc`. Ако не съществува някой от файловете или съществува, но не е обикновен файл, ще бъде изведено съобщение за грешка.

3.3. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВА СИСТЕМА

При физическата реализация на една файлова система трябва да бъдат взети решения по редица въпроси, като:

1. Каква да е стратегията за управление на дисковото пространство при създаване, нарастване и унищожаване на файлове?
2. Как да се съхраняват файловете - данни и атрибути на файла?
3. Как да се реализира логическата структура на файловата система, включително и каква да е структурата на каталог?
4. Как да се осигури защита на данните от неправомерен достъп и от повреда или разрушение?

Основни цели при решаването на тези въпроси трябва да са:

1. **Ефективност**
   Файловата система трябва да осигурява бърз достъп до данните и ефективно използване на дисковата памет.
2. **Надеждност и сигурност**
   Файловата система трябва да е устойчива в условията на конкурентен достъп от много потребители, при възможни сривове и да е защитена от несанкциониран достъп.
3. **Разширяемост**
   Физическата организация трябва да е адекватна на съвременното състояние и тенденциите в развитието на хардуера.

Изложението в този раздел е посветено на тези въпроси и методи за тяхното решаване, използвани в съвременните операционни системи.

3.3.1. Стратегии за управление на дисковата памет

Стратегията за управление на дисковата памет трябва да даде отговор на два въпроса.

1. Кога се разпределя дискова памет за файл?

Едната възможност е това да се прави статично (предварително) при създаването на файла, а другата е да се прави динамично при нарастване на файла.

2. Колко непрекъснати дискови области може да заема един файл?

Дали за файл се разпределя една (или малко на брой) непрекъсната дискова област с нужния размер или за файл могат да се разпределят много порции дискова памет, които не е задължително да са физически съседни, а броят им най-често се ограничава от капацитета на диска.

Най-често реализираните стратегии са две.

Статично и непрекъснато разпределение

За файл с размер n байта се отделя една непрекъсната област от диска, в която могат да се съхранят всичките n байта и това се прави при създаването на файла. За тази цел обаче, системата трябва да има предварителна информация за максималния размер на файла, който той може да достигне по време на съществуването си. Естествено тази информация може и трябва да се осигури от потребителя по време на създаване на файла и тогава системата ще разпредели дискова памет за файла. Преимуществото, което този метод дава, е високата производителност на системата при последователна обработка на файла, тъй като последователните байтове на файла са разположени в съседни сектори, писти и цилиндри на диска.

Проблем може да възникне при **нарастване** на файла, което е нещо обичайно, ако размерът на файла надмине разпределената му предварително памет. Едно възможно решение на този проблем е разпределяне на нова непрекъсната област с по-голям размер и преместване на файла в новата област. Това обаче не е добро решение, тъй като операцията може да се окаже бавна и скъпа при големи файлове. Друго решение е да се направи компромис при искането за непрекъснатост на дисковата памет, т.е. един файл да може да заема не една, а няколко, но малко на брой непрекъснати области от диска с размери, заявени от потребителя.

Такава стратегия е реализирана в операционната система OS/360 за машини от семейството на IBM/360. Например, ако при изпълнение на програма, създаваща файл, потребителят укаже в съответния JCL оператор:

//DD . . . SPACE=(TRK, (20, 5)), . . .

то първоначално за файла се отделя една непрекъсната област от диска с размер 20 писти и това е така наречения първичен екстент. При запълване на тази област системата извършва вторично разпределение - разпределя втора непрекъсната област от диска с размер 5 писти. При продължаващо нарастване на файла вторичното разпределение се повтаря, но максимално 15 пъти. Следователно максималният размер на файл се ограничава от един първичен и 15 вторични екстента с размери избрани от потребителя. След това проблемът с нарастване на файла отново се появява.

Друг недостатък на тази стратегия е известен като **фрагментация** на свободната дискова памет. Това означава съществуване на достатъчно свободни участъци дисковата памет, които обаче не са съседни и поради това не може да бъде удоволетворена заявка за първично или вторично разпределение на памет за файл. Причината за този проблем е изискването за непрекъснатост и нефиксирания размер на единицата за разпределение на дискова памет.

Тази стратегия е използвана в по-старите операционни системи. Но в последните години с развитието на технологията и появата на CD-ROM и други еднократно записвани носители, статичното и непрекъснато разпределение отново се оказа приложимо. При създаване на CD-ROM диск размерите на файловете са предварително известни и не се изменят при последващо използване на файловата система.

Динамично и поблоково разпределение

Файлът се дели на части с фиксирана дължина, които ще наричаме блокове. Последователните блокове на файла при записването им на диска не е задължително да са физически съседни. Това дава възможност дискова памет за файл да се разпределя по блокове тогава, когато е необходима, т.е. динамично при нарастване на файла. Така се решава и проблема с нарастване на файла и проблема с фрагментацията на свободната дискова памет, тъй като памет се разпределя динамично по блокове, които са с еднакъв фиксиран размер.

При прилагане на тази стратегия трябва се избере **размер на блока**. Отчитайки организацията на диска, естествени кандидати са сектор, писта, цилиндър. Избор на голям блок, напр. цилиндър, би довел до неефективно използване на дисковата памет за сметка на частично запълненен последен блок на файл. Избор на малък блок пък означава, че големите файлове ще се състоят от много блокове, които е възможно да са несъседни и пръснати по дисковата повърхност. А това означава неефективност при последователна обработка на файла. Необходим е компромис, който най-често е 1К байта, 2К байта, 4К байта. Ако дисковият сектор е 512 байта, то всеки блок ще заема 2, 4 или 8 последователни сектора.

При по-нататъшното изложение ще предполагаме, че се използва динамично и поблоково разпределение на дискова памет, както е в повечето съвременни системи.

Единицата за разпределение на дискова памет ще наричаме блок, въпреки че в някои операционни системи се използват и други термини, като зона, клъстер или тя е просто дисков сектор. Следователно, за файловата система дисковото пространство е последователност от блокове с фиксиран размер и адреси (номера) 0,1,2, ...N.

3.3.2. Системни структури

Файловата система трябва да съхранява информация за разпределението на дисковата памет, а именно за свободните блокове и за блоковете разпределени за всеки един файл. Под системни структури (наричат ги също метаданни) тук ще разбираме структури, съхраняващи такава информация постоянно на диска.

Информация за всички свободни блокове трябва да бъде съхранявана, тъй като само по съдържанието на блока не може да се отличи свободния от заетия блок. Някои възможни структури данни, използвани за тази цел са следните.

Свързан списък на свободните блокове

Всички свободни блокове са организирани в едносвързан списък, т.е. първата дума от всеки свободен блок съдържа адрес на следващ свободен блок. Тази структура е реализирана във файловата система на XINU. Недостатък на този метод е, че системната структура, т.е. свързаният списък, е пръсната по всички свободни блокове, което крие опасности за повредата й при системни сривове.

Свързан списък на блокове с номера на свободни блокове

Свободните блокове се групират и номерата на блоковете от една група се записват в първия свободен блок. Следователно информацията се съхранява в едносвързан списък от дискови блокове, които съдържат номера на свободни блокове. Самите блокове от списъка вече не са свободни за разлика от предходната структура. Такава структура е по-компактна, големината й е пропорционална на свободното дисково пространство и позволява разработката на ефективни алгоритми за разпределяне на блокове. Този подход е използван във файловата система на UNIX System V (s5fs).

Карта или таблица

Структурата представлява масив от елементи, като всеки елемент съответства позиционно на блок от диска, т.е. на блок с номер равен на индекса на елемента в масива. Съдържанието на всеки елемент описва състоянието на блока. В най-простия случай може да се помнят две състояния - свободен и зает. Тогава елемент от картата може да е просто бит. Ако битът е 0, то съответният блок е свободен, а ако е 1 е зает (разпределен за файл или друга структура на диска) или обратното. Такава системна струтура се нарича ***битова карта (bit map).*** Преимущество на битовата карта е, че е компактна и с фиксиран размер, а при разпределяне на памет позволява да се отчита физическото съседство на блоковете. Битова карта е използвана във файловите системи на Minix, Linux , OS/2 (HPFS) и Windows (NTFS).

Друг тип информация, която трябва да бъде съхранявана, е за дисковата памет, разпределена за всеки един файл. Всеки файл от гледна точка на физическото му представяне представлява последователност от блокове, съдържащи последователните му данни, като последователните блоковете на файла може да не са физически последователни. Следователно файловата система трябва да съхранява информация за блоковете, разпределени за всеки един файл в съответната логическа последователност.

Ще разглеждаме някои възможни структури, съхраняващи информация за разпределените за файл блокове.

Свързан списък на блоковете на файла

Всеки блок на файла ще съдържа в първата си дума номер на следващ блок на файла и данни в останалата част. Основният недостатък на тази структура е неефективната реализация на произволен достъп до файла. За да се прочете произволен байт от файла системата трябва да прочете всички предходни блокове в списъка докато стигне до данните, искани от програмата. Друг недостатък е, че адресната информация е пръсната по всички блокове на файла, а това прави файловата система неустойчива при повреди. И още един недостатък, който в някои случаи е критичен, е че броят на байтовете за данни в блока вече не е степен на 2.

Карта или таблица

Същността на проблема при предходната реализация се състои в това, че адресите и данните се съхраняват заедно. Идеята на картата (таблицата) на файловете е свързаните списъци от номера на блокове за всички файлове на диска да се съхраняват в една структура отделно от данните. В същност това може да е същатата системна структура карта, в която се съхранява информация за свободните блокове и която описахме по-горе, като елемент на масива е достатъчно голям, така че да позволи съхраняване на следните състояния на блок - свободен, повреден, а ако е зает да съдържа номера на следващия блок на файла. Тази структура е използвана в MSDOS, където се нарича таблицата ***FAT (File Allocation Table).*** По-нататък ще разгледаме физическото представяне на файловата система в MSDOS.

Индекси

Основният недостатък на предишния подход е, че информацията за всички файлове се съхранява в една структура, което при големи дискове създава проблеми. При големи дискове естествено голяма става и картата на файловете, а тя трябва цялата да се зарежда и съхранява в оперативната памет по време на работа дори ако е отворен само един файл. Противното би довело до значително понижаване на ефективността на работа на системата. Следователно по-добре би било ако списъците на различните файлове се съхраняват в отделни структури и тогава в паметта ще се зарежда само информацията за отворените файлове.

Структурата, в която се съхранява информацията за блоковете, разпределени за определен файл, се нарича индекс. Всеки индекс съдържа адресите на дисковите блокове, разпределени за един файл, като наредбата на адресите отразява логическата наредба на блоковете във файла. При физическата реализация на индекса трябва да се отчита размера на файла, т.е. представянето му да не ограничава размера на файла и при много големи файлове да се осигурява бърз произволен достъп до файла.

Една възможна реализация на индекса е ***индексен списък*** в XINU. Индексният списък е едносвързан списък от индексни блокове. Индексните блокове са с размер, различен от този на блоковете за данни и се намират в област на диска, отделна от областта на блоковете за данни, т.е. адресното пространство за индексните блокове е различно от това на дисковите блокове. Всеки индексен блок, освен адрес на следващ индексен блок, съдържа и определен брой адреси на дискови блокове с данни, като наредбата на адресите отразява логическата наредба на блоковете с данни във файла.

Друга възможна реализация е чрез дърво.Такъв подход е използван в Unix, Linux и MINIX, където структурата се нарича ***индексен описател*** (***i-node*** от ***index node***) и

при големи файлове е корен на дърво, включващо и косвени блокове. Подробно структурата на индексния описател ще разгледаме в разделите за физическа организация на файловата система в Unix и Linux.

Използваната във файловата система HPFS на OS/2 реализация на индекса е B+ дърво. Структурата се нарича ***Fnode***, който може да е корен на B+ дърво с две или три нива в зависимост от размера и фрагментираността на файла.

Общи параметри

На всеки диск или дял от диск (ако е разделен на дялове) се изгражда файлова система с определена физическа структура, обикновено наричана том (volume). Друг тип информация, която трябва се съхранява, са общи параметри на физическата структура на файловата система, т.е. на тома. Обикновено системната структура се нарича ***суперблок***. В суперблока се съхраняват общи параметри, като:

- размер на файловата система (максимален номер на блок на тома)
- размер на блок
- размери и адреси на различни области от тома, съдържащи системни структури, като напр., на индексната област, на битовата карта, на FAT
- общ брой свободни блокове на тома и други ресурси, напр., индексни описатели.

3.3.3. Реализация на каталог

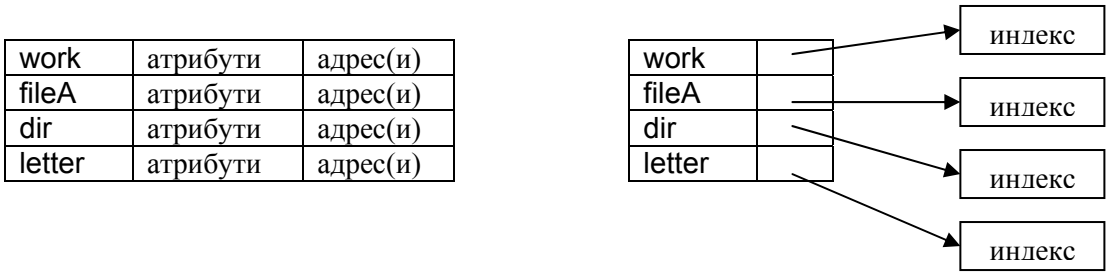
Каталозите също съдържат системна информация. Основната функция на каталога е да преобразува името на файла в информация, необходима за намиране на данните му. Каталогът съдържа по един запис (в някои системи може и повече от един) за всеки файл или подкаталог. Записът съдържа собственото име на файла и информация, необходима за намиране на данните му (адресна информация). Адресната информация в запис на каталог зависи от стратегията за разпределение на дисковото пространство и използваните системни структури за разпределението на дисковата памет.

При непрекъснато разпределение всеки файл заема една непрекъсната дискова област, следователно записът съдържа един дисков адрес и брой блокове (сектори или друга единица за разпределение на дискова памет) на данните на файла.

При поблоковото разпределение с карта на файловете, записът съдържа адреса на първия блок на файла, а номерата на останалите блокове са в картата. При поблоковото разпределение с използване на индекс, в записа на каталога се съдържа адрес на индекса, където е цялата адресна информация за файла.

Вторият проблем е съхраняване на атрибутите на файла. Една възможност е те да се съхраняват в записа на каталога заедно с адресната информация. Всеки запис съдържа собствено име на файл, всички атрибути, реализирани от операционната система и един или повече дискови адреси, осигуряващи достъп до данните. Това е подход използван във файловите системи на CP/M, MSDOS и ISO 9660 за CD-ROM (*Фиг.3.5а*).

Друг вариант, който може да се реализира при използване на индекс, е атрибутите да се съхраняват в индекса заедно с адресната информация. Това силно опростява структурата на запис в каталог. Той съдържа само собствено име на файл и адрес на индекса. Този подход е показан на *Фиг.3.5б* и е реализиран във файловите системи на Unix, MINIX, Linux.



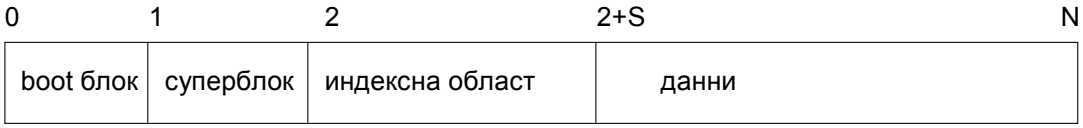
Фиг. 3.5 Структура на каталог - (а) с атрибути и дискови адреси и (б) с указател на индекс

И при двете описани схеми предполагахме, че записите в каталога са в хронологичен (случаен) ред. При добавяне на запис в каталог, той се записва на първото свободно място. Когато се търси файл, последователно се обхождат всички записи в каталога, докато се намери файла или се достигне края на каталога. При големи каталози последователното търсене може да се окаже бавно. Една възможност за ускоряване на търсенето на файл в каталог е да се реализира друга организация на каталога. Например, в HPFS на OS/2 записите във всеки блок на каталога са сортирани по името на файла, а блоковете на един каталог са организирани в B дърво. Аналогична организация на каталог, реализирана в NTFS, ще бъде разгледана в следващия раздел.

Чрез каталога се реализира и йерархичната организация на файловата система. Всеки каталог съдържа записи за файловете и каталозите, за които той е родителски каталог. Освен това всеки каталог съдържа и два стандартни записа, известни като записи “.” и “..”. Записът “.” описва самия каталог, а записът “..” описва родителския му каталог.

3.4. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВА СИСТЕМА В UNIX

Като пример ще разгледаме базовата файлова система на UNIX System V (s5fs). Всеки диск се състои от един или няколко дяла (partitions). Дяловете се разглеждат като независими устройства (на всеки дял съответства различен специален файл). За файловата система дисковото пространство на всеки диск или дял от диск е последователност от блокове с фиксиран размер и адресирани с номера 0, 1, 2,... N. Тази абстрактна представа за дисковете се осигурява от по-долния слой във входно/изходната подсистемата (драйверите). Специфичните особености на дисковете, като брой цилиндри, писти, сектори, начин на разполагане на блоковете върху повърхността на диска и др. са скрити. Единственото, по което се различават дисковете е по максималния номер на блок. При създаване на празна файлова система, дисковото пространство на всеки дял се разделя на четири области (*Фиг.3.6*).



Фиг. 3.6. Разпределение на дисковото пространство в том на s5fs

Блок с номер 0, наричан boot блок, съдържа програма за първоначално зареждане на операционната система. Използва се само в коренната файлова система, но заради еднотипността присъства и в другите файлови системи.

Блок с номер 1 е наречен суперблок, тъй като съдържа общи параметри на физическата структура на файловата система.

От блок 2 започва индексната област, където се съхраняват индексните описатели (i-node) на всички файлове. Размерът на тази област S блока е функция от общия размер на файловата система N и трябва внимателно да бъде изчислен така, че да не ограничава броя на файловете на диска. Този размер се задава като параметър при изграждане на празна файлова система с командата `mkfs` и не може да бъде променян след това.

В последната област - данни, се съхраняват блоковете на всички файлове и каталози, косвените блокове, блоковете от списъка на свободните блокове и евентуално някои блокове са свободни.

Индексни описатели

Всеки файл от произволен тип има точно един индексен описател (i-node), независимо от това в кой и колко каталога е включен и под какви имена. Индексните описатели се номерират от 1 и номерът съответства на позицията му в индексната област. Индексният описател е с размер 64 байта и има следната структура:

```
struct dinode
{
     ushort     di_mode;   /* mode and type of file */
     short      di_nlink;  /* number of links to file */
     ushort     di_uid;    /* owner's user id */
     ushort     di_gid;    /* owner's group id */
     off_t      di_size;   /* number of bytes in file */
     char       di_addr[40];    /* disk block addresses */
     time_t     di_atime;  /* time last accessed */
     time_t     di_mtime;  /* time last modified */
     time_t     di_ctime;  /* time last  changed */
};
```

В описанието на тази структура, а и в следващите се използват производни типове, дефинирани във файла <sys/types.h>. Някои от тях са:

```
typedef long    time_t;
typedef long    off_t;
typedef long    daddr_t;
typedef unsigned short ushort;
```

В полето di_mode се съхранява типа на файла в старшите 4 бита и кода на защита на файла в останалите 12 бита.

Полето di_nlink съдържа броя на твърдите връзките или имената на файла, т.е. колко пъти в записи на каталози е цитиран този номер на индексен описател.

Полетата di_uid и di_gid определят собственика и групата на файла.

Полето di_size при обикновени файлове и каталози съхранява размера на файла в брой байтове.

В полетата di_atime, di_mtime и di_ctime се записват дата и време, съответно на последен достъп, последно изменение и последно изменение на i-node на файла.

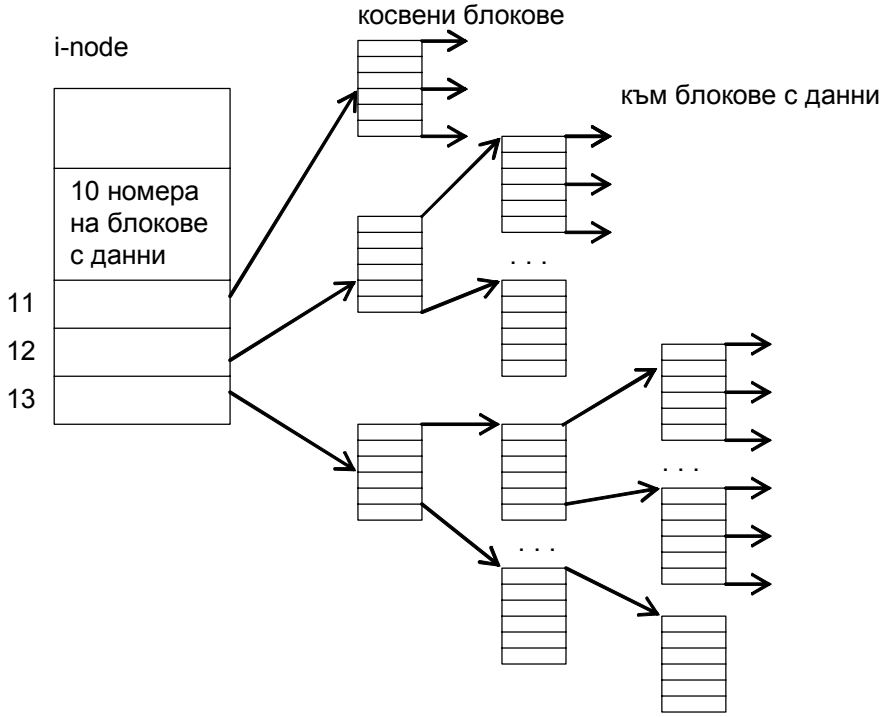
Ако файлът е специален в полето di_addr се съхранява номера на устройството, на което съответства специалния файл.

За другите типове файлове в полето di_addr се съхраняват 13 дискови адреса (номера на блокове от областта за данни) всеки в 3 байта. Първите 10 адреса са директни адреси на първите 10 блока с данни на файла. Ако файлът стане по-голям от 10 блока, тогава се използват косвени блокове. Косвените блокове се намират в областта за данни, но съдържат номера на блокове, а не данни на файла. Единадесетият адрес съдържа номер на косвен блок, който съдържа номерата на следващите блокове с данни на файла. Това се нарича единична косвена адресация. Чрез дванадесетия адрес се реализира двойна косвена адресация, т.е. там е записан номер на косвен блок, който съдържа номера на косвени блокове, които вече съдържат номера на блокове с данни. За представяне на много големи файлове се използва тринадесетия адрес и тройна косвена адресация. На *Фиг.3.7* е представена структурата при съхраняване на номерата на блоковете, разпределени за файл.

Да видим какво означава много голям файл и дали този начин за представяне на файл не поставя практически ограничения за размера на файла. Нека предположим, че блокът е с размер 1KB, което означава, че в един косвен блок се побират 256 адреса по 4 байта. Тогава ограничението за размера на файл е:

$1024 * (10 + 256 + 256^2 + 256^3)$ байта, което е от порядъка на 16GB.

В същност това дълго време е било по-скоро теоретично ограничение за размера на файла, тъй като звучи невероятно да се наложи създаването на такъв файл. Сега вече не е толкова невероятно. Но дори и това да се случи, то би могло да се избере размер на блок 2KB, 4KB или повече, при което ограничението за размера ще стане много по-голямо число, напр., при блок 4KB - от порядъка на 4TB.



Фиг. 3.7. Представяне на файл в UNIX

Какво представлява **кода на защита** в полето `di_mode`? Той определя правата на различните потребители за достъп до файла. Спрямо определен файл всички потребители се класифицират в следните класове:

- администратор или привилегирован потребител (`root`)
- собственик - потребител, който е собственик на файла (чийто идентификатор е в полето `di_uid`);
- група - потребители, които не са собственик на файла, но принадлежат на групата на собственика (групата в полето `di_gid`);
- други - потребители, които не са в първите три класа.

Администраторът има неограничен достъп до цялата файлова система. За всеки от останалите три класа, типовете разрешен достъп до файла са кодирани в кода на защита. Битовете в полето `di_mode` се интерпретират по следния начин:

`04000` при изпълнение се изменя uid на процеса (set UID бит)
`02000` при изпълнение се изменя gid на процеса (set GID бит)
`01000` Sticky bit
`00400` четене за собственика (`r`)
`00200` писане за собственика (`w`)
`00100` изпълнение за собственика (`x`)
`00040` четене за групата
`00020` писане за групата
`00010` изпълнение за групата
`00004` четене за другите
`00002` писане за другите
`00001` изпълнение за другите

Трите типа достъп за различните типове файлове означават следното.
За обикновен файл:

- Право `r` означава правото да отворим файла за четене, т.е. с флаг `O_RDONLY` или `O_RDWR` в `open`.
- За да отворим файл в режим `O_WRONLY` и `O_RDWR` трябва да имаме право `w` (за режима `O_RDWR` е необходимо да имаме `r` и `w`).
- Право `x` е необходимо за да извикаме файл за изпълнение (системен примитив `exec`).

За каталог:

- Право `r` означава правото да четем съдържанието на каталога, напр., с `ls -l dir`.
- Право `w` означава правото да създаваме или унищожаваме на файлове в каталога.
- Право `x` означава търсене на файлове в каталога и позициониране в каталога, напр., с `cat dir/text` или `cd dir`.

Например, за да изпълним системния примитив:

```
fd = open("/home/ivan/letter", O_RDWR);
```

трябва да имаме права `x` за `/`, `x` за `/home`, `x` за `/home/ivan`, `r` и `w` за `letter`.
За да създадем нов файл `ff1` с примитива:

```
fd = creat("/home/ivan/ff1", 0644);
```

трябва да имаме права `x` за `/`, `x` за `/home`, `x` и `w` за `ivan`.

Правата `r` и `x` при каталози действат независимо, `x` не изисква `r` и обратно, следователно при комбинирането им могат да се получат интересни резултати. Например, каталог с право `x` и без право `r` за дадена категория потребители е така наречения "тъмен каталог". Потребителите имат право да четат файловете в каталога, само ако им знаят имената. Този метод се използва в FTP сървери, когато някои раздели от архива трябва да са достъпни само за посветени потребители.

Чрез Sticky bit (показва се като `t` от командата `ls -l`) за каталог може да се осигури допълнителна защита на файловете в каталога. Ако този бит не е вдигнат за каталог, е достатъчно потребител да има право `w` за каталога, за да може да унищожи всеки файл в него. Но ако битът е вдигнат за каталог, потребител може да унищожи файл ако има право `w` за каталога и е едно от трите: собственик на файла, собственик на каталога или привилегирован потребител. Пример за каталог с вдигнат Sticky bit е `/tmp`.

Представянето на файловете в UNIX System V съчетава следните предимства: малък индексен описател с възможност за представяне на големи файлове при осигуряване на бърз достъп до произволен байт от файла. И при най-големи файлове за достъп до произволно място във файла са необходими най-много три допълнителни достъпа до диска за трите нива на косвени блокове (Индексният описател се зарежда в паметта при отваряне на файла и се съхранява там до затварянето му.).

Суперблок

Суперблокът съдържа изключително важна информация, характеризираща физическата структура на файловата система. Основните данни, съдържащи се в него, са описани в следната структура:

```
struct filsys
{
      ushort      s_isize;   /* size in blocks of i-list */
      daddr_t     s_fsize;   /* size in blocks of entire volume */
      short       s_nfree;   /* number of addresses in s_free */
```

```
daddr_t     s_free[NICFREE];    /* free block list */
short       s_ninode;  /* number of i-nodes in s_inode */
ushort      s_inode[NICNODE];   /* free i-node list */
char        s_flock;  /*lock during free list manipulation*/
char        s_ilock;   /* lock during i-list manipulation */
      . . .
daddr_t     s_tfree;   /* total free blocks */
ushort      s_tinode;  /* total free i-nodes */
      . . .
};
```

Полето `s_isize` съдържа дължината на индексната област в блокове, която се задава и зарежда в полето при създаване на файловата система.

Полето `s_fsize` съдържа максималния номер на блок във файловата система и също както `s_isize` се задава и зарежда при изграждане на файловата система.

Полето `s_nfree` съдържа брой свободни блокове, чиито номера са записани в масива `s_free`.

Полето `s_ninode` съдържа брой свободни индексни описатели, чиито номера са записани в масива `s_inode`.

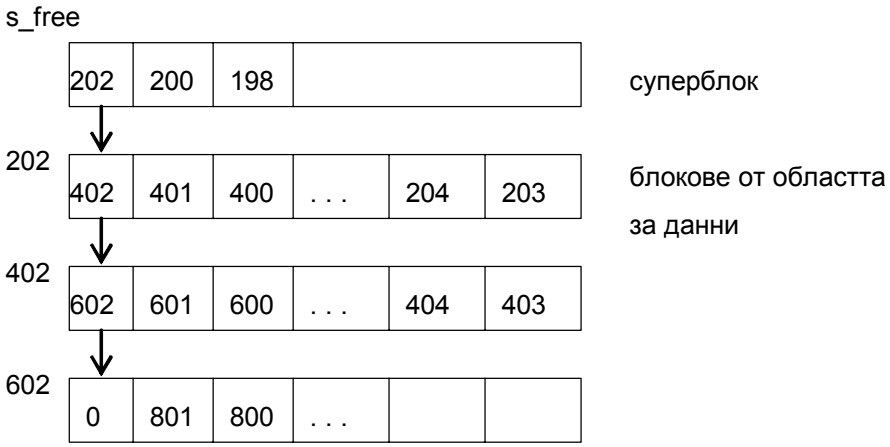
Полетата `s_flock` и `s_ilock` се използват като флагове за заключване на достъпа до списъка на свободните блокове и свободните индексни описатели по време на манипулирането им с цел избягване на състезание между конкурентни процеси.

Полетата `s_tfree` и `s_tinode` съдържат общия брой свободни блокове и свободни индексни описатели на диска.

Суперблоковете на коренната и всички монтирани файлови системи се зареждат в оперативната памет и остават там през цялото време на работа на системата. Това осигурява бърз достъп до най-важните характеристики на файловите системи и се използва от алгоритмите за разпределяне и освобождаване на ресурсите - блокове и индексни описатели.

Списък на свободните блокове

Всички блокове от областта за данни, които не се разпределени за файлове, каталози, за косвени блокове, т.е не съдържат данни на файловата система са свободни. Но файловата система не е в състояние да отличи свободния от заетия блок само по съдържанието му. Затова информацията за всички свободни блокове се съхранява в специална структура - едносвързан списък от блокове, които съдържат номера на свободни блокове. Масивът `s_free` в суперблока представлява глава на списъка. Останалите елементи от списъка на свободните блокове, които са дискови блокове, са разположени в областа за данни. На *Фиг.3.8* е изобразен примерен списък на свободните блокове.



Фиг.3. 8. Списък на свободните блокове в s5fs на UNIX

Алгоритмите за разпределяне и освобождаване на блокове съществено използват факта, че първият елемент от списъка - масивът `s_free` е в паметта (суперблоковете на всички монтирани файлови системи се кешират в паметта).

Алгоритъм `alloc` за разпределяне на блок.

1. Ако `s_nfree` не е 0, то се разпределя блок, чийто номер е в `s_free[s_nfree--]` (и `s_nfree` се намалява с единица).

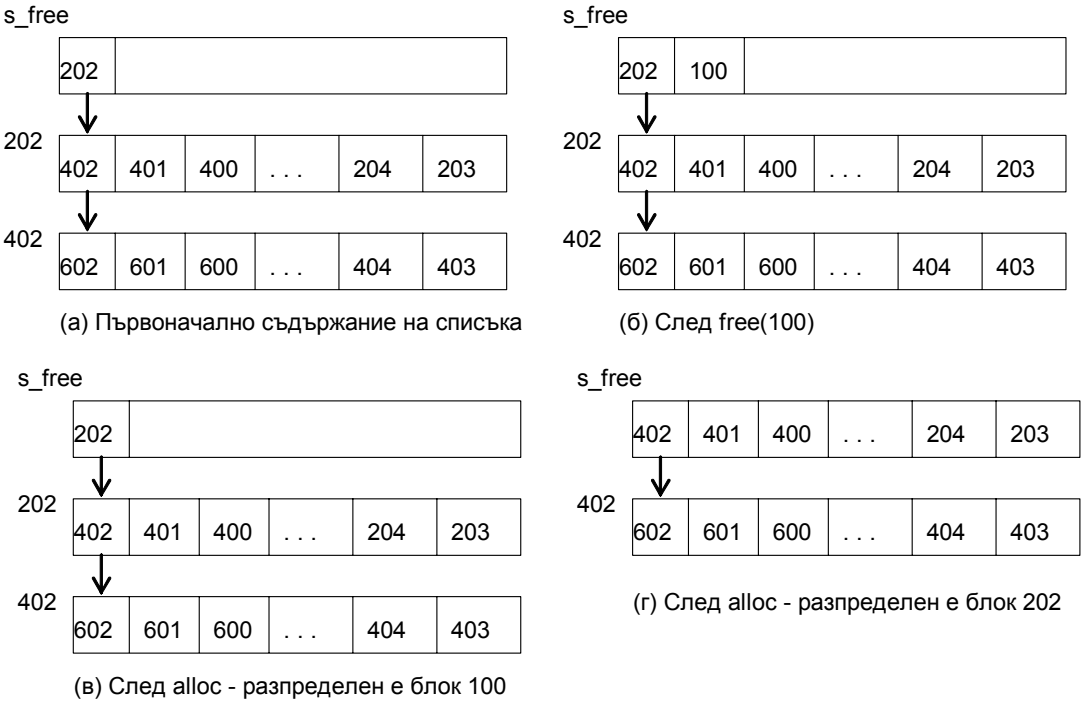
2. Ако `s_nfree` е 0, то се попълва масива `s_free`, като се прочита първия блок от списъка, чийто адрес е в `s_free[s_nfree]`. Разпределя се току що освободения блок.

Алгоритъм `free` за освобождаване на блок.

1. Ако масивът `s_free` не е пълен, то номерът на освобождавания блок се записва в `s_free[++s_nfree]` (и `s_nfree` се увеличава с 1).

2. Ако масивът `s_free` е пълен, то съдържанието му се копира в освобождавания блок, а неговия адрес се записва в `s_free[0]` и в `s_nfree` се записва 0 (освобождаваният блок става първи блок в списъка, а `s_free` е празен).

На *Фиг.3.9* е показан пример за работата на алгоритмите `alloc` и `free`. Тъй като суперблокът е в паметта, алгоритмите осигуряват ефективно разпределяне и освобождаване на блокове. Разчита се, че в много от случаите манипулирането на списъка на свободните блокове засяга само масива `s_free` и не изисква достъп до диска. От друга страна обаче, последните освободени блокове първи се разпределят. Затова е безмислено да се правят усилия за възстановяване на случайно изтрити файлове и такива средства отсъстват в командите на UNIX системите.



Фиг. 3.9. Разпределяне и освобождаване на блокове

Списък на свободните индексни описатели

Другият ресурс, който файловата система трябва да управлява, са индексните описатели. Те са разположени в отделно пространство - индексната област. При създаване на файл за него трябва да се разпредели свободен индексен описател, а при унищожаване на файл индексния му описател трябва да се отбележи като свободен.

В суперблока е разположен масив `s_inode`, в който са записани известен брой номера на свободни индексни описатели. За разлика от управлението на блоковете, този масив не продължава в свързан списък от блокове. Това означава, че освен номерата в масива `s_inode` в индексната област може да има и други свободни индексни описатели. Това не създава проблеми при управлението им, тъй като ядрото може да различи свободния от заетия i-node по полето `di_mode` - битовете за тип са 0 в свободен i-node. В същност би могло и без масива `s_inode`, т.е. когато е необходим свободен i-node ще се търси в индексната област. Но такъв алгоритъм би бил неефективен, ако всеки път когато се създава файл се обхождат блоковете на индексната област в търсене на свободен i-node. Затова е въведен масива `s_inode`, в който са кеширани известен брой номера на свободни индексни описатели, но структурата не продължава в пълен списък.

Каталози

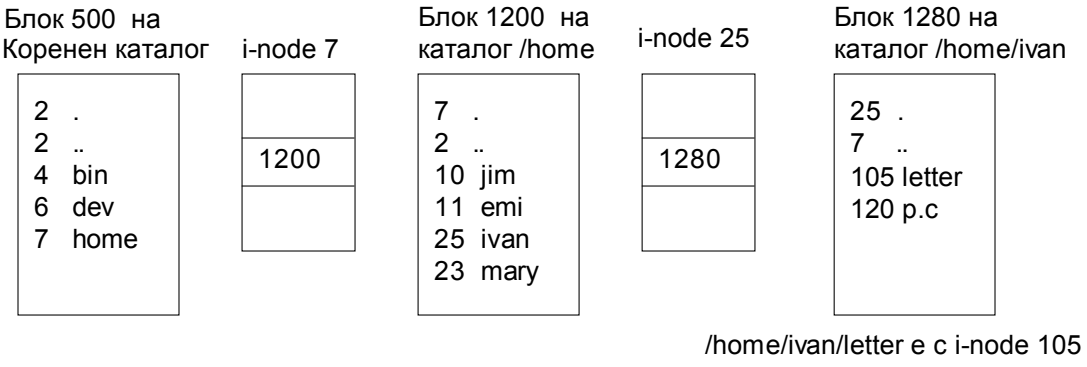
Каталозите са тип файлове, които осигуряват връзката между името и данните и атрибутите на файл, и йерархичната организация на файловата система. В разглежданата s5fs файлова система записът в каталога има следната структура:

```
struct direct
{
     ushort     d_ino;
     char  d_name[DIRSIZE];
};
```

Полето `d_name` съдържа собственото име на файл или подкаталог, а полето `d_ino` съдържа номера на индексния му описател. Във всеки каталог има два стандартни записа, в единия полето `d_name` съдържа "..", а `d_ino` номера за родителския каталог, в другия полето `d_name` съдържа ".", а `d_ino` номера за самия каталог. Тези два записа присъстват и в празен каталог и са част от представянето на дървовидната структура на файловата система. Някои записи може да са празни. Номер 0 в полето `d_ino` означава, че записът е освободен при унищожаване на файл.

Първите няколко индексни описатели са резервирани за някои важни файлове, като коренния каталог, файла на лошите блокове и др. Така номерът на i-node на коренния каталог е известен, а самия каталог е разположен в произволни блокове от областта за данни.

При отваряне на файл файловата система трябва да преобразува пълното име на файла в индексен описател, при което се използват каталозите. Напр., нека разгледаме търсенето на файл `/home/ivan/letter` и намирането на индексния му описател (*Фиг.3.10*). Търсенето започва от коренния каталог, чийто i-node е вече в паметта. Търси се запис в коренния каталог, съдържащ първата компонента от името - `home`, и се намира номер на i-node 7. Чрез i-node 7 получаваме достъп до блоковете на каталога `/home`, където търсим следващата компонента от името - `ivan` и намираме номер на i-node 25. Чрез i-node 25 получаваме достъп до блоковете на каталога `/home/ivan` и продължаваме с търсене на следващата, в случая последна, компонента от името - `letter`. Така намираме търсения i-node на файла `/home/ivan/letter`, който в разглеждания пример е 105. При относително пълно име се търси по същия начин, с тази разлика, че се тръгва от текущия каталог, чийто i-node също се намира в вече в паметта.

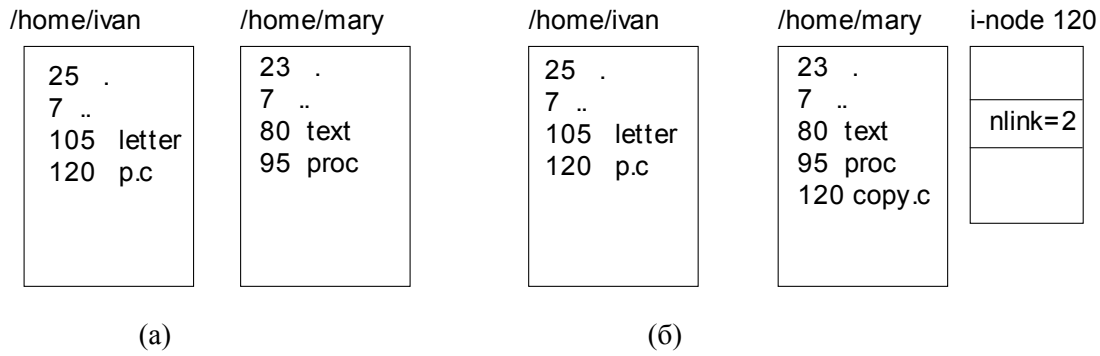


Фиг. 3.10. Търсене на файл с име `/home/ivan/letter`

При реализация на **твърдите връзки** (hard links), всички записи в каталози за определен файл съдържат един и същи номер на i-node. За да се разбере по-добре представянето на твърдите връзки да разгледаме следния пример. Нека има два каталога със съдържание показано на *Фиг.3.11(а)*. След изпълнение на командата:

$ **`ln /home/ivan/p.c /home/mary/copy.c`**

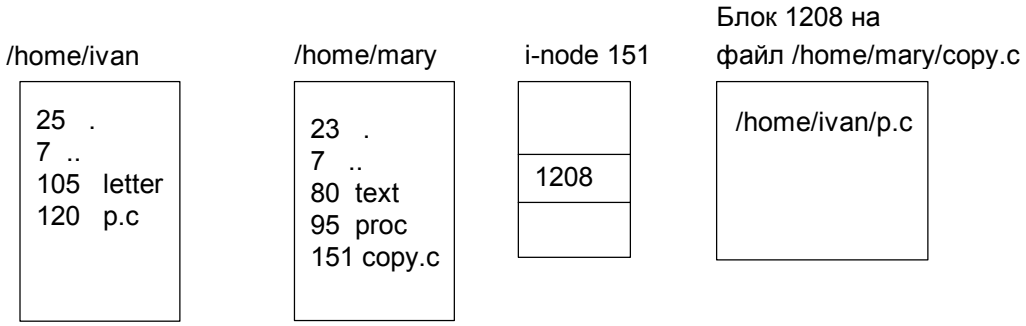
съдържанието на каталога `/home/mary` и i-node на файла `/home/ivan/p.c` се изменят както е показано на *Фиг.3.11(б)*. Имената `/home/mary/copy.c` и `/home/ivan/p.c` сочат към един и същи индексен описател с номер 120, т.е. отнасят се до един и същи файл.



Ф*иг. 3.11*. Съдържание на каталозите и i-node - (а) преди и (б) след `ln`

Символната връзка (symbolic link или soft link) е файл, който сочи към друг файл, т.е. съдържанието на файл от тип символна връзка е името на друг файл. Нека съдържанието на каталозите е същото, както на *Фиг.3.11(а)*. На *Фиг.3.12*. е показано представянето на новия файл `copy.c` след изпълнение на на командата:

```
$ ln -s /home/ivan/p.c /home/mary/copy.c
```



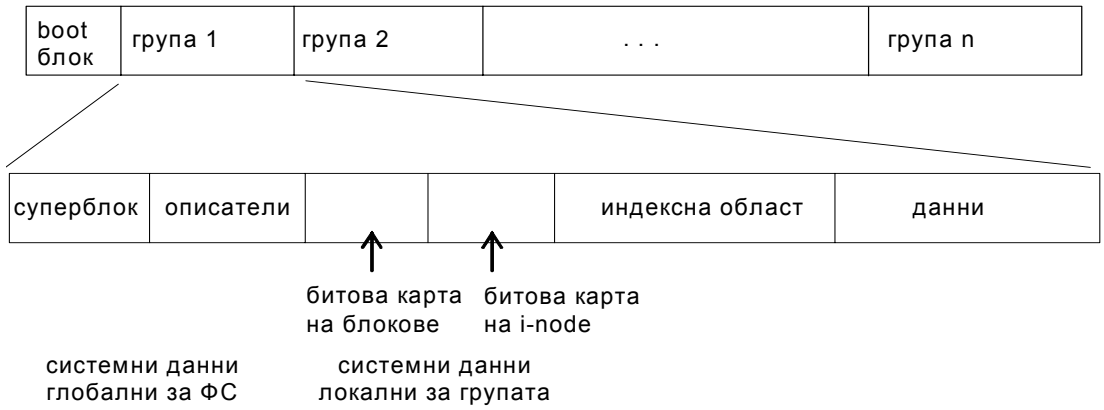
Фиг. 3.12. Съдържание на каталозите и i-node след `ln -s`

Символната връзка, както и твърдата връзка, позволява един файл да има няколко имена. При твърдата връзка се гарантира съществуването на файла и след като оригиналното име е унищожено, докато при символната връзка това не е така. Всъщност дори не се проверява съществуването на файл `/home/ivan/p.c` при изпълнението на командата `ln -s`. Символната връзка се интерпретира при опит за достъп до файла чрез нея. Друга разлика между двата типа връзки е, че символна връзка може да се създава през границите на файловата система и към файл от произволен тип - обикновен файл, специален файл, каталог.

Разгледаната файлова система s5fs има много преимущества, които отбелязахме в хода на разглеждането, но има и слаби места. От гледна точка на надеждността слабо място е суперблока, за който се съхранява само едно копие. За производителността на системата е критично, че индексните описатели са разположени в началото на файловата система и са далеч от данните на файла. Също така, системните структури и алгоритмите за разпределение на блокове с времето могат да доведат до фрагментираност на файловете по целия диск. Индексната област е с фиксиран размер, който се задава при създаване на файловата система, и неподходящия му избор може да доведе до липса на свободни индексни описатели при наличие на свободно дисково пространство. И накрая, ограничението за дължина на името на файл (14 символа) и максимален общ брой на i-node (65535). Тези недостатъци и ограничения са довели до разработката на нови версии на Unix файлови системи, като версията в 4.2BSD, известна с наименованието Berkley Fast File System или ufs. Следващият пример е файловата система на Linux, в която са отстранени споменатите недостатъци.

3.5. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА СИСТЕМА В LINUX

Основната файлова система в Linux е Extended File System-version (ext2/ ext3). Тя се базира на файловата система в Unix, но използва идеи от 4.2BSD и MINIX. Дори в ранните версии на Linux се използва директно файловата система на MINIX. И тук на всеки диск или дял от диск съответства различен специален файл. За файловата система дисковото пространство на всеки диск или дял е последователност от блокове с фиксиран размер - 1KB, 2KB или 4KB. Това е и адресуемата единица на диска, единна за всички файлови системи. *Фиг.3.13* представя областите, в които е организирано дисковото пространство на един том.



Фиг. 3.13. Разпределение на дисковото пространство в том на ext2/ext3

Всяка група съдържа част от файловата система и копие на глобални системни структури, критични за цялостността на системата - суперблока и описателите на групите. Ядрото работи със суперблока и описателите от първата група. Когато се извършва проверка на непротиворечивостта на файловата система (изпълнява се командата `fsck`), ако няма повреди, то двете системни структури от първата група се копират в останалите групи. Ако се открие повреда, тогава се използва старо копие на системните структури от друга група и обикновено това позволява да се ремонтира файловата система.

Битови карти

Битовите карти описват свободните ресурси - блокове и индексни описатели в съответната група. Значение 0 означава свободен, а 1 използван блок или индексен описател. Размерът на групите е фиксиран и зависи от размера на блок, като стремежът е битовата карта на блоковете да се събира в един блок. Групата е с размер 8*b блока, където b е размер на блок в брой байта, т.е. ако размерът на блок е 1KB, то групата е с размер 1024*8 блока по 1KB, което е 8MB, ако размерът на блок е 2KB, то групата е с размер 32MB и при блок 4KB групата е с размер 128MB. Броят на групите е N/(8*b), където N е размер на диска в брой блокове.

Индексни описатели

Индексните описатели са разпределени равномерно във всички групи, но се адресират в рамките на файловата система. Структурата на индексния описател в ext2 и ext3 е разширение на i-node от UNIX System V с нови полета. Размерът му е 128 байта и има следната структура:

```
struct ext3_inode {
      __le16     i_mode;          /* File mode */
      __le16     i_uid;           /* Low 16 bits of Owner Uid */
      __le32     i_size;          /* Size in bytes */
      __le32     i_atime;   /* Access time */
      __le32     i_ctime;   /* Creation time */
      __le32     i_mtime;   /* Modification time */
      __le32     i_dtime;   /* Deletion Time */
      __le16     i_gid;           /* Low 16 bits of Group Id */
      __le16     i_links_count;  /* Links count */
      __le32     i_blocks;  /* Blocks count */
      __le32     i_flags;   /* File flags */
      __le32     i_block[EXT3_N_BLOCKS];/* Pointers to blocks */
      . . .
};
```

Адресните полета са 12+1+1+1, т.е. използва се косвена адресация на три нива, но броят на директните адреси е увеличен с два, т.е. е 12. Освен това адресните полета вместо по 3 байта са по 4 байта, което позволява адресирането на 2^{32} блока. Следователно, при размер на блок 4KB, максималният размер на файловата система е $4096*2^{32}$, което е от порядъка на 16TB.

Добавени са нови атрибути на файл, например:

- размер на файла в брой блокове по 512 байта (`i_blocks`)
- още едно четвърто поле за дата и време (`i_dtime`)
- атрибути флагове:

immutable - Файлът не може да се изменя, унищожава, преименува и да се създават нови връзки към него (дори от администратора).

append only - Писането във файла е винаги добавяне в края му.

synchronous write - Системният примитив `write` завършва след като данните бъдат записани на диска.

secure deletion - При унищожаване на файла блоковете му се форматират.

undelete - При унищожаване на файла съдържанието му се съхранява за евентуално възстановяване впоследствие.

compress file - При съхраняване на файла ядрото автоматично го компресира.

Някои от добавените атрибути са за планирано бъдещо развитие на файловата система.

В този индексен описател има две полета за размер на файл `i_size` и `i_blocks`. Файлът се съхранява в цяло число блокове и сл., в общия случай `i_size`<= 512*`i_blocks`. Но е възможно и обратното - `i_size`> 512*`i_blocks`, ако файлът съдържа „дупка”. И тук полето `i_size` (32 бита) ограничава размера на файла до 4GB, но ext2 позволява работа с големи файлове на 64-битова архитектура. Полето `i_dir_acl` (32 бита), което не се използва при обикновени файлове, се използва като разширение на `i_size`, т.е. размерът на файла се съхранява в 64 битово цяло число.

Описатели на групи блокове

Всяка група е описана чрез запис с размер 32 байта (Group descriptor**)**, съдържащ следната информация за групата:

- адрес на блок с битовата карта на блоковете за групата;
- адрес на блок с битовата карта на индексните описатели за групата;
- адрес на първи блок на индексната област в групата;
- брой свободни блокове в групата;

- брой свободни i-node в групата;
- брой каталози в групата;
- резервирано поле от 14 байта.

Описателите на всички групи са събрани в областта `описатели`, копие на която се съдържа във всяка група.

Суперблок

Суперблокът съдържа общите параметри на физическата структура на файловата система. Някои от основните данни, съдържащи се в него, са следните:

- общ брой блокове - размер на файловата система;
- общ брой индексни описатели;
- брой блокове резервирани за администратора (обикновено е 5% от общия брой блокове). Тези резервирани блокове позволяват на администратора да продължи работа, дори когато няма свободни блокове за другите потребители.
- общ брой свободни блокове и индексни описатели;
- размер на блок;
- брой блокове в група;
- брой i-node в група;
- полета използвани при автоматична проверка на файловата система (`fsck`) при boot, напр., дата и време на последен `fsck`, брой монтирания след последния `fsck`, максимален брой монтирания преди следващ `fsck`, максимален интервал време между две изпълнения на `fsck`.

Каталози

Ограничението за максимална дължина на името на файл е 255 символа, затова записите в каталога са с променлива дължина и съдържат:

- номер на i-node;
- дължина на записа;
- дължина на името на файла;
- име на файла, съхранявано в толкова байта, колкото са небходими.

Всеки запис в каталога е подравнен на границата на 4 байта Следователно, в края името на файла може да е допълнено с няколко символа ‘\0’. Структурата на запис в каталога е следната:

```
struct ext3_dir_entry {
      __le32     inode;          /* Inode number */
      __le16     rec_len;        /* Directory entry length */
      __le16     name_len;       /* Name length */
      char  name[EXT3_NAME_LEN]; /* File name */
};
```

При изтриване на име на файл от каталог в полето `inode` се записва 0, а полето `rec_len` в предходния запис се увеличава с броя байтове на освобождавания запис. Следва пример, илюстриращ изтриването на файл от каталог. Нека съдържанието на каталог преди да е изтрито името `oldfile` е следното.

inode	rec_len	name_len	name							
21	12	1	.	\0	\0	\0				
22	12	2	.	.	\0	\0				
54	16	5	h	o	m	e	1	\0	\0	\0
62	12	3	u	s	r	\0				
199	16	7	o	l	d	f	i	l	e	\0
35	12	4	n	e	x	t				

След изтриване на името `oldfile` съдържанието на каталога ще е:

inode	rec_len	name_len	name							
21	12	1	.	\0	\0	\0				
22	12	2	.	.	\0	\0				
54	16	5	h	o	m	e	1	\0	\0	\0
62	28	3	u	s	r	\0				
0	16	7	o	l	d	f	i	l	e	\0
35	12	4	n	e	x	t				

И така, новото във физическото представяне на ext2/ext3 в сравнение с файловата система s5fs е:

- Разделяне на дисковото пространство на групи блокове.
- Използване на битови карти при управление на свободните ресурси - блокове и индексни описатели.

И двете промени, както и съответните промени в стратегиите и алгоритмите, имат за цел да се постигне по-висока степен на локалност на файловете и системните им структури (да се намали разстоянието между i-node и блоковете на файл), а от там и по-добра производителност.

- За системните структури, съдържащи критична за файловата система информация се съхраняват няколко копия.
- При реализацията на символните връзки са въведени така наречените бързи символни връзки. Ако името, към което сочи символната връзка, е до 60 символа, то се съхранява в адресните 60 байта на i-node. По този начин се спестява един блок и следването на символната връзка е по-ефективно.

3.6. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА СИСТЕМА В MSDOS

За файловата система на MSDOS дисковото пространство е последователност от сектори по 512 байта. Броят им зависи от типа на диска. Твърдите дискове могат да се разделят на дялове (partitions), като всеки дял е последователност от сектори върху диска и се описва от адреса на началния сектор, размера и др. В по-ранните версии на MSDOS броят на дяловете на един диск е ограничен до четири. Схемата на разделяне в MSDOS 5 се усъвършенства, при което не се ограничава броя на дяловете. Във всеки дял от диск или цял диск (ако не е разделян или е дискета) се изгражда независима файлова система, наричана том (volume). На *Фиг.3.14* е изобразено разпределението на дисковото пространство в една файлова система, изградена на един том.

Фиг. 3.14. Разпределение на дисковото пространство в том на MSDOS

Boot сектора съдържа програмата, която стартира зареждането на операционната система в паметта и параметри на тома, като размер на клъстер, размер на тома, размер и брой копия на FAT, размер на коренния каталог. Тази област присъства на всички томове, дори и да не са системни, но при опит да се стартира системата от несистемен диск се извежда съобщение за грешка.

Следващата област е FAT (File Allocation Table) или Таблица за разпределение на дисковото пространство. Тя представлява карта на файловете и съдържа цялата информация за дисковата памет, разпределена за файловете, за свободното дисково пространство и за дефектните сектори. От гледна точка на файловата система това са най-важните и критични данни на диска, затова се пазят две копия на FAT, разположени едно след друго.

Следва областта, в която се съхранява коренния каталог на файловата система. Файловата система в MSDOS е йерархична, но за разлика от Unix, тук всеки том се разглежда като независима дървовидна структура. По тази причина, може би, коренният каталог на всеки диск е разположен в отделна област, след FAT.

В областта данни се съхраняват всички файлове и каталозите, различни от коренния.

FAT - Таблица за разпределение на дисковото пространство

Дискова памет за файлове се разпределя динамично и единицата за разпределение се нарича клъстер (cluster). Клъстерът е последователност от 1 или повече сектора (степен на двойката брой сектори) и всички клъстери на един том са с еднакъв размер. За различните томове в една система клъстерите може да са с различни размери.

FAT съдържа елементи с фиксирана дължина (12, 16 или 32 бита), като броят им зависи от размера на тома и от размера на клъстера. Елементи 0 и 1 съдържат код, идентифициращ формата на тома. Всеки от останалите елементи съответства позиционно на клъстер от областта за данни. За удобство номерацията на клъстерите започва от 2, т.е. първият клъстер в областта за данни е с номер 2. Съдържанието на елемент от FAT определя състоянието на съответния клъстер - свободен, разпределен за файл или повреден. Код 0 в елемент означава свободен клъстер. Код 2, 3, 4, ..., N (максимален номер на клъстер на тома) означава, че съответният клъстер е разпределен за файл, а числото в елемента е указател към следващ елемент на FAT (което е и адрес на следващ клъстер на файла). В този случай елементът е част от верига елементи на

FAT, представяща клъстерите, разпределени за файл. Максималният код, който може да се запише в елемент, означава край на верига, т.е. съответният клъстер е последен във файл. На *Фиг.3.15* е показано примерно съдържание на FAT. Използвани са следните обозначения: EOF - код за последен клъстер (0xFFFF) и FREE - код за свободен клъстер (0).

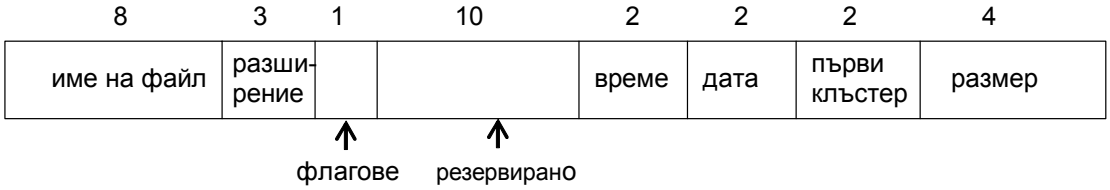
Фиг. 3.15. Таблица FAT в MSDOS

Има три версии на файловата система - FAT12, FAT16 и FAT32 (би трябвало да се нарича FAT28, но степен на 2 звучи по-добре). Дължина на елемент от FAT първоначално е 12 бита. При FAT12 могат да се адресират до 2^{12} (4096) клъстера, но в началото се използват само флопи дискове с капацитет 360KB. С появата на твърдите дискове проблемът се решава като се увеличава размера на клъстер на 1KB, 2KB, 4KB, което ограничава размера на дял до 16MB и на диска до 64MB (при 4KB клъстер и 4 дяла на диск).

С навлизането на по-големи дискове, се увеличава размерът на елемент от FAT на 16 бита. При FAT16 могат да се адресират до 2^{16} (65536) клъстера. Освен това се използва различен размер на клъстера - 2KB, 4KB, 8KB, 16KB, 32KB в зависимост от размера на тома, където се изгражда файловата система. Това позволява работа с дял с максимален размер 2GB и диск до 8GB (при 32KB клъстер и 4 дяла на диск). Най-накрая се появява версията FAT32, която осигурява работа с дискове по-големи от 8GB и с по-нормален размер на клъстер. Могат да се адресират до 2^{28} клъстера, защото старшите 4 бита не се използват. Максималният размер на дял в тази версия е 2TB.

Каталози

Едно от различията при представяне на каталозите в MSDOS е, че кореният каталог е в отделна област и с фиксиран по време на форматирането размер (до 256 записа). Всички останали каталози са разположени в областта за данни, имат променлива дължина и памет за тях се разпределя динамично, както и при обикновените файлове. Но всички каталози имат еднаква структура. Съдържат записи с дължина по 32 байта, всеки от които описва файл или подкаталог. Структурата на запис от каталога е показана на *Фиг. 3.16*.



Фиг. 3.16. Структура на запис от каталог в MSDOS

В полетата `име` (8 байта) и `разширение` (3 байта) е записано собственото име на файла. Компонентата разширение на името не е задължителна.

В полето `флагове` (1 байт) се съхраняват различните атрибути-флагове, които определят характеристики на файла, като една част са свързани със защитата на данните, а друга с типа на файла (записа в каталога). Всеки флаг се съхранява в един бит и предназначението им е показано в *Таблица 3.3*.

Таблица 3.3. Значение на битовете в атрибута флагове

7	6	5	4	3	2	1	0	Предназначение
							1	само за четене
						1		скрит файл
					1			системен файл (от CP/M)
				1				етикет на тома
			1					каталог
		1						бит при архивиране

В полето `време` (2 байта) се съхранява времето на създаване или последно изменение на файла. Кодирано е като цяло число без знак, което се изчислява от астрономическото време по формулата:

час * 2048 + минути * 32 + секунди / 2

Секундите се кодират в младшите 5 бита, минутите в 6 бита и часът в старшите 5 бита.

Полето `дата` (2 байта) съдържа датата на създаване или последно изменение на файла и се изчислява от календарната дата по формулата:

(година - 1980) * 512 + месец * 32 + ден

Денят е кодиран в младшите 5 бита, месеца в 4 бита и годината-1980 в старшите 7 бита (ще има проблем 2108г.).

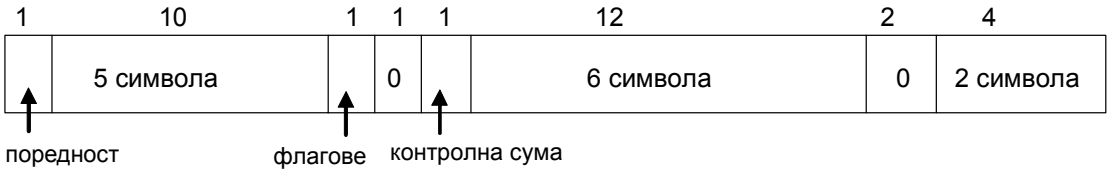
В полето `първи клъстер` (2 байта) е записан номера на първия елемент от FAT, от който започва веригата, представяща разпределените за файла клъстери. Ако за файла още не е разпределена памет, полето съдържа 0.

В полето `размер` (4 байта) е записан размера на файла в брой байта, въпреки че отделената дискова памет може да е повече, тъй като се разпределя в клъстери. Това ограничава размера на файл до 4GB.

Всички каталози без коренния съдържат двата стандартни записа - с име "." за самия каталог и с име ".." за родителския каталог. Коренният каталог пък съдържа един специален запис за етикета на тома, който е символен низ с дължина до 11 символа и се записва в първите две полета на записа. Това е другата разлика в представянето на коренния и останалите каталози.

Ранните версии на Windows (Windows 3.x, Windows95, Windows98) използват само файловата система FAT на MSDOS. Но от Windows95 се въвеждат дълги имена на файлове в Unicode, които не следват правилото 8.3 на MSDOS. Това налага някои промени в каталога, които са реализирани по такъв начин, че да има обратна съвместимост със старите системи. Ако името е дълго или в Unicode, то за файла има

няколко записа в каталога. Един главен запис съдържа съкратено име на файла във формат “xxxxxx~1.yyy”, което автоматично се генерира от файловата система, и атрибутите му. Допълнителни записи, толкова колкото са необходими, съдържат дългото име и предшестват основния запис. Наредбата на допълнителните записи е кодирана в първото поле - `поредност`. Структурата на запис от каталога, съдържащ част от дългото име (до 13 символа, всеки кодиран в 2 байта) е представена на *Фиг.3.17*. Полето `флагове` съдържа код 0x0F, което е невъзможна комбинация от флагове. Полето `контролна-сума` се използва за да се избегнат проблеми при манипулиране на файла от MSDOS чрез краткото име. Достъп до файла се осигурява чрез всяко от двете имена (те са функционално еквивалентни).



Ф*иг.3.17*. Структура на запис от каталог с част от дълго име

На *Фиг.3.18* е показан пример за записите в каталог, съхраняващи информация за файл с дългото име “My Long document file name.doc”. Необходими са три допълнителни записа за дългото име, които предшестват основния запис.

67	. d o c	Ф	0	CS		0	
2	e n t f	Ф	0	CS	i l e n a	0	m e
1	M y L o	Ф	0	CS	n g d o c	0	u m
M	YLONG~1DOC	Ф	NT	S	дати/време	първи	размер

Фиг. 3.18. Пример за съхраняване на дълго име на файл

Друга промяна във файловата система, която не е свързана с дългите имена и големите дискове, е добавяне на допълнителни атрибути в резервираното място на основния запис за файл. Освен старите време и дата, които са за последно изменение, се добавят:

- време на създаване;
- дата на създаване;
- дата на последен достъп;
- разширение на полето за номер на първи клъстер с още два байта.

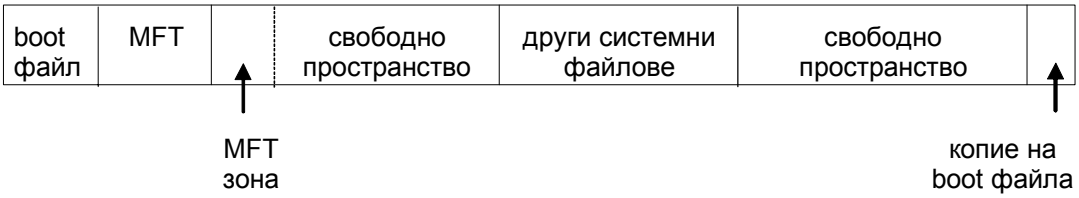
3.7. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА СИСТЕМА NTFS

NTFS се реализира като напълно нова файлова система в Windows NT и се използва в следващите версии на Windows 2000 и по-нови. Схемата на разделяне на дялове и изграждане на томове е усъвършенствана. Освен обикновените томове (simple volumes), се реализират се многодялови томове (multipartition volumes). Многодялов том означава, че една файлова система се изгражда върху няколко дяла.

Единицата за разпределяне и адресиране на дисковото пространство от NTFS е клъстер с размер 1KB, 2KB или 4KB. Адресът на клъстер относно началото на тома се нарича LCN (Logical cluster number), а адресът в рамките на определен файл се нарича VCN (Virtual cluster number).

Една от ключовите новости в NTFS е принципа „**всичко на диска е файл**": и данните и метаданните (системните структури, както ги наричаме) се съхраняват в тома като файлове, т.е. на всеки том има обикновени файлове, каталози и ***системни файлове***. Това, че метаданните се съхраняват като файлове, позволява динамично разпределяне на дискова памет при нарастване на метаданните, без да са необходими фиксирани области върху диска за тях. Сърцето на файловата система и главният системен файл е MFT (Master File Table).

На *Фиг.3.19* е изобразено разпределението на дисковото пространство в една новоформатирана файлова система.



Фиг. 3.19. Разпределение на дисковото пространство в том на NTFS

Файлът MFT представлява индекс на всички файлове на тома. Съдържа записи по 1KB и всеки файл на тома е описан чрез един запис, включително и MFT. Освен MFT има и други файлове с метаданни, чиито имена започват със символа $ и са описани в първите записи на MFT. Самите системни файлове са разположени към средата на тома. В началото на тома е boot файла. Скрито в края на тома е копие на boot файла. За да се намали фрагментирането на MFT файла, се поддържа буфер от свободно дисково пространство - MFT зона, докато останало дисково пространство не се запълни. Размерът на MFT зоната се намалява на половина винаги когато останалата част от тома се запълни.

MFT файл

Всеки файл на тома е описан в поне един запис на MFT файла. Индексът (номерът) на началния MFT запис за всеки файл се използва като идентификатор на файла във файловата система (File reference number). Първите записи са резервирани за системните файлове, т.е. те имат предопределени индекси. Следва списък на част от тези файлове.

Име на файл	Индекс	Описание на съдържанието на файла
$Mft	0	MFT файл
$MftMirr	1	Копие на първите записи от MFT файла
$LogFile	2	Журнал при поддържане на транзакции
$Volume	3	Описание на тома
$AttrDef	4	Дефиниции на атрибутите

\	5	Коренен каталог на тома
$Bitmap	6	Битова карта на тома
$Boot	7	Boot сектори на тома
$BadClus	8	Списък на лошите клъстери на тома

Адресът на MFT файла (на началото му) се намира в Boot файла. Първият запис в MFT файла описва самия файл и следователно съдържа адресна информация, осигуряваща достъп до целия MFT файл, ако той е фрагментиран и заема няколко области на тома. Файлът $MftMirr съдържа копие на първите няколко записа от $Mft. Целта на това дублиране е по-висока надеждност на файловата система. Файлът $LogFile се използва за поддържане на транзакции при манипулиране структурата на файловата система. Всяка последователност от дискови операции, които реализират една операция на файловата система, като създаване, преименуване, унищожаване на файл и др., представлява транзакция. NTFS създава записи за тези операции в $LogFile. Тези записи се използват за възстановяване на коректността на файловата система след сривове. Действието на всички незавършили транзакции се отменя и файловата система се възстановява до състоянието си преди началото на тези транзакции. Файлът $Volume съдържа информация за тома, като сериен номер и име на тома, версия на NTFS, дата на създаване и др. Файлът $Bitmap представлява битова карта на клъстерите на тома. Причината и boot сектора да е файл, е може би за да се спази правилото всичко на диска е файл. Всички повредени сектори на тома са организирани във файл на лошите клъстери на тома $BadClus.

Атрибути на файл

Всеки файл се съхранява като последователност от двойки „атрибут/значение”. Един от атрибутите са данните на файла (наричан unnamed data attribute). Други атрибути са име на файл, стандартна информация и други. Всеки атрибут се съхранява като отделен поток от байтове. Обикновен файл може да има и други атрибути данни, наричани named data attribute. Това променя представата ни за файл, като една последователност от байтове, т.е. файлът може да има няколко независими потока данни. Следва списък на част от типовете атрибути.

Име на тип атрибут	**Описание на значението на атрибута**
$FILE_NAME	Името на файла. Един файл може да има няколко атрибута от този тип, при твърди връзки или ако се генерира кратко име в MSDOS стил.
$STANDARD_INFORMATION	Атрибути на файл, като размер, време и дата на създаване и последно изменение, флагове, брой твърди връзки.
$DATA	Данните на обикновен файл. Всеки файл има един неименован атрибут данни и може да има допълнителни именовани атрибути данни.
$INDEX_ROOT, $INDEX_ALLOCATION, $BITMAP	Три атрибута при представяне на каталозите.
$ATTRIBUTE_LIST	Този атрибут се използва, когато за файл има повече от един запис в MFT. Съдържа списък от атрибутите на файла и индексите на записите, където се съхраняват.
$VOLUME_NAME,	Два атрибута използвани в системния файл $Volume.

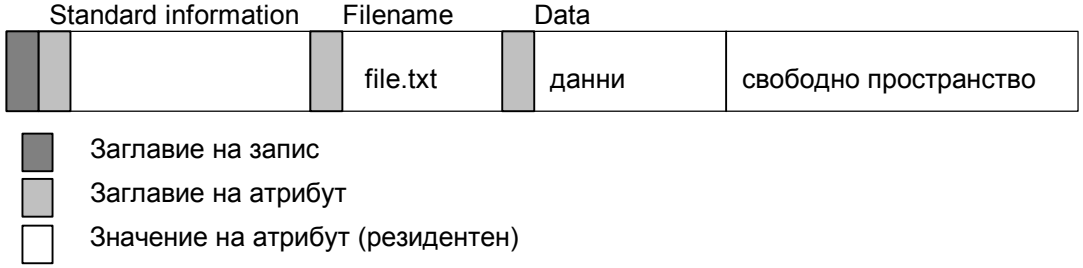
$VOLUME_INFORMATION	Те съхраняват информация за тома, като етикет, версия и др.

Всеки тип атрибут освен име на типа, има и числов код на типа. Този код се използва при наредба на атрибутите в MFT запис на файла. MFT записът съдържа числовия код на атрибута (който го идентифицира), значение на атрибута и евентуално име на атрибута (когато има няколко атрибута от един тип). Значението на всеки атрибут е поток от байтове. Един файл може да има няколко атрибута от един тип, например няколко $FILE_NAME или $DATA атрибута.

Атрибутите биват резидентни или нерезидентни. Резидентен е атрибут, който се съхранява изцяло в MFT записа. Някои атрибути са винаги резидентни, напр., $FILE_NAME, $STANDARD_INFORMATION, $INDEX_ROOT. Ако значението на атрибут, като данните на голям файл, не може да се съхрани в MFT записа, то за него се разпределят клъстери извън MFT записа. Такива атрибути се наричат нерезидентни. Файловата система решава как да съхранява един атрибут. Нерезидентни могат да бъдат само атрибути, чиито значения могат да нарастват, например $DATA.

MFT запис

Всеки MFT запис съдържа заглавие на записа (record header) и атрибути на файла. Всеки атрибут се съхранява като заглавие на атрибута (attribute header) и данни (значение). Заглавието на атрибута съдържа код на типа, име, флагове на атрибута и информация за разположението на данните му. Един атрибут е резидентен ако данните му се поместват в един запис заедно със заглавията на всички атрибути на файла. На *Фиг.3.20* е изобразена структурата на MFT запис за малък файл, т.е. всичките му атрибути могат да се съхранят в MFT записа.



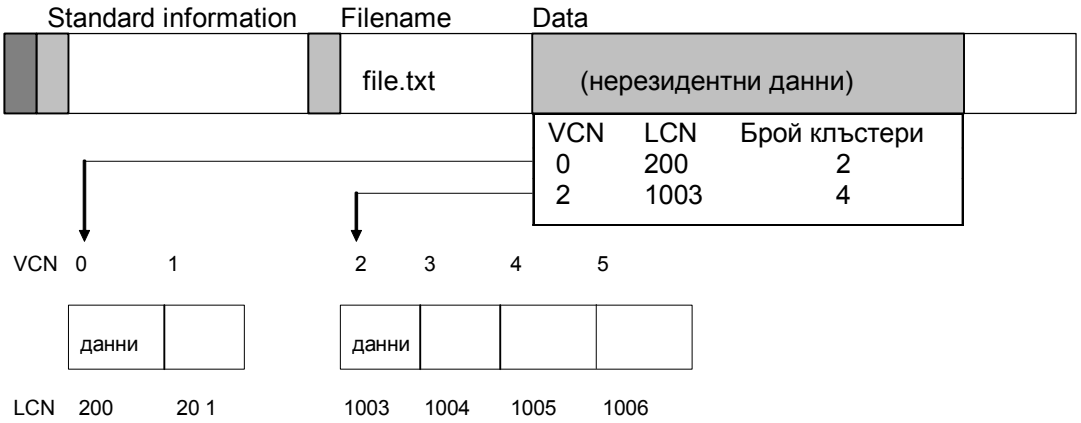
Фиг. 3.20. MFT запис за файл само с резидентни атрибути

Ако един атрибут не е резидентен, заглавието му (което е винаги резидентно) съдържа информация за клъстерите, разпределени за данните му. Адресната информация се съхранява подобно на подхода в HPFS на OS/2, т.е. представлява последователност от описания на екстенти (run/extent entry). Всеки екстент е непрекъсната последователност от клъстери, разпределени за данните на съответния атрибут и се описва от адрес на началния клъстер и дължина, т.е. с тройката:

(VCN, LCN, брой клъстери).

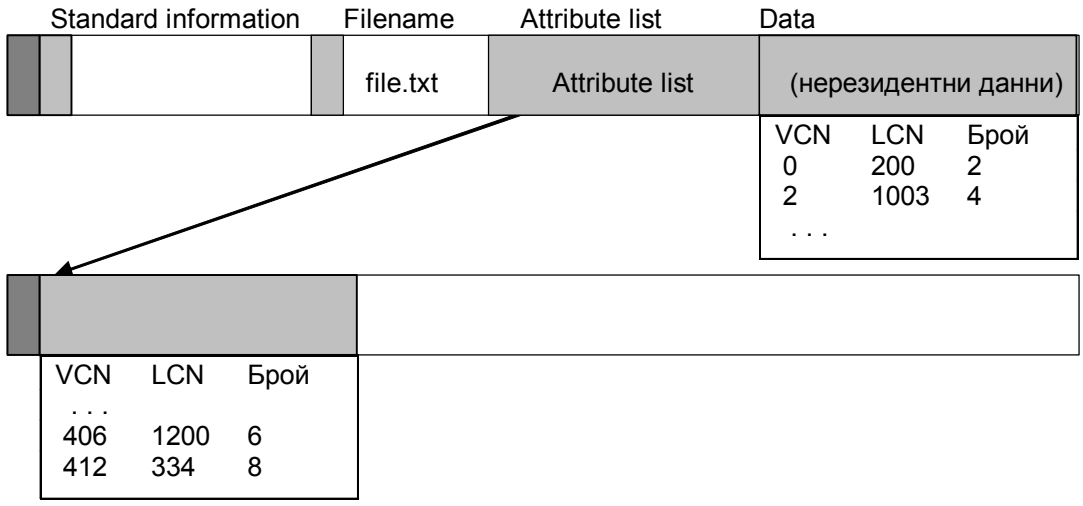
За разлика от HPFS, тук описанията на екстентите се съхраняват в последователна структура, а не в B+ дърво.

На *Фиг.3.21* е изобразена структурата на MFT запис за по-голям файл, т.е. данните на файла се съхранят в два екстента.



Фиг.3.21. MFT запис за файл с нерезидентен атрибут данни

Ако един файл има много атрибути и не може да се опише в един MFT запис, то се разпределят допълнителни записи. В основния (първи) запис има атрибут $ATTRIBUTE_LIST, съдържащ указатели към допълнителните записи (за всеки атрибут на файла съдържа код на типа и номер на MFT записа, съхраняващ атрибута). Адресната информация за един нерезидентен атрибут може да се съхранява в няколко записа, ако обемът на данните е голям или разпределената памет е фрагментирана. На *Фиг.3.22* е изобразена структурата на MFT записите за още по-голям обикновен файл, т.е. описанието на екстентите не се помества в един запис и се използва допълнителен запис и атрибут $ATTRIBUTE_LIST.

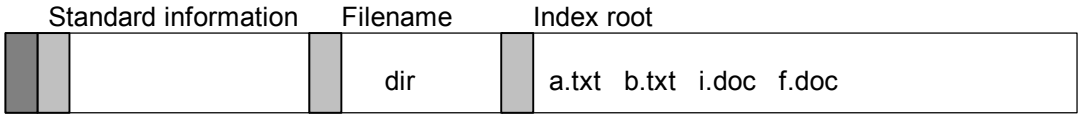


Фиг. 3.22. MFT записи за голям файл

Каталози

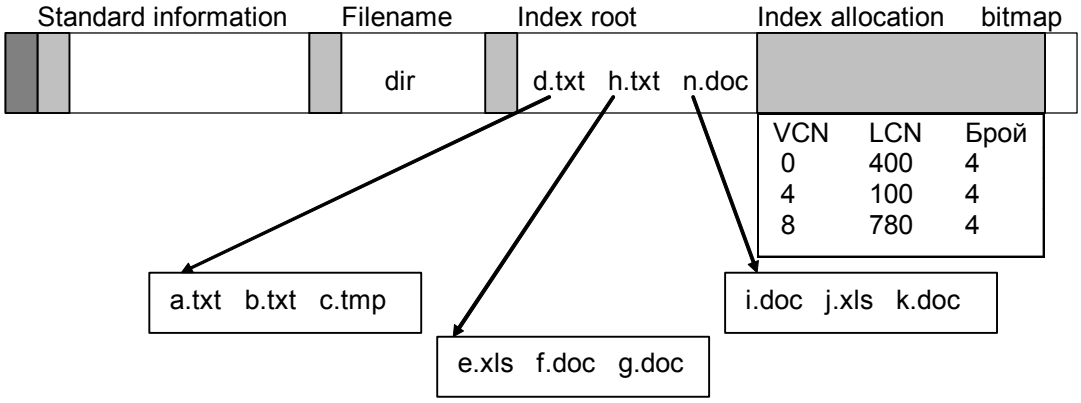
Каталогът съдържа записи с променлива дължина, всеки от които съответства на файл или подкаталог, в съответния каталог. Всеки запис съдържа името на файла и индекса на основния MFT запис на файла, както и копие на стандартната информация на файла. Това дублиране на стандартната информация изисква две операции писане при изменението ѝ, но ускорява извеждането на справки за съдържанието на каталога.

Записите в каталога са сортирани по името на файла и се съхраняват, подобно на HPFS, в структура B дърво. Ако каталогът е малък, всичките му записи се съхраняват в атрибута $INDEX_ROOT, който е резидентен, т.е. целият каталог се намира в MFT записа си. На *Фиг.3.23* е изобразена структурата на MFT запис за малък каталог.



Фиг. 3.23. MFT запис за малък каталог

Когато каталогът стане голям, за него се разпределят екстенти с размер 4KB, наречени индексни буфери (index buffers). Атрибутът $INDEX_ROOT и тези екстенти са организирани в B дърво. В този случай каталогът има и атрибут $INDEX_ALLOCATION, който съхранява адресна информация за разположението на екстентите-индексни буфери. Атрибутът $BITMAP е битова карта за използването на клъстерите в индексните буфери. *Фиг.3.24* илюстрира представянето на голям каталог (за простота всеки екстент съдържа 3 записа).



Фиг. 3.24. Представяне на голям каталог

Четвърта глава

ПРОЦЕСИ

В операционните системи понятието процес е централно, всичко останало, включително и абстракцията файл, се гради върху него. Съвременните компютри са способни да изпълняват няколко операции едновременно. Докато централният процесор (ЦП) изпълнява команди, дисковото устройство може да чете и терминалът или принтерът да извеждат данни. Съществуването на този реален паралелизъм довежда до революционна за операционните системи идея, която в началото бе наречена не много удачно ***мултипрограмиране*** (***multiprogramming***). В оперативната памет са заредени няколко програми и макар, че ЦП във всеки един момент изпълнява само една команда, той може да се превключва от изпълнение на една програма към друга и да го прави много бързо. Първоначално мултипрограмирането се въвежда с цел по-ефективното използване на ресурсите на компютъра. Но също така у потребителя се създава впечатление за едновременното изпълнение на няколко програми, което е в същност една илюзия, поддържана от операционната система. Истината е, че ЦП изпълнява няколко последователни дейности, като за тази цел трябва да се съхранява информация за тези дейности, за да е възможно да се възстановява изпълнението на прекъсната дейност. За да се изолира потребителя от детайлите по поддържането на този псевдо паралелизъм, е необходим модел, който да опрости работата на човека в операционната система.

4.1. МОДЕЛ НА ПРОЦЕСИТЕ

В такъв един модел се въвежда понятие за обозначаване на дейността по изпълнение на програма(и) за определен потребител. В различни операционни системи са използвани понятията ***задание*** (***job***)***, задача*** (***task***), ***процес*** (***process***). Терминът процес за първи път е бил използван в операционната система MULTICS на MIT през 60-те години и преобладава в съвременните операционни системи. Най-краткото определение за **процес е програма в хода на нейното изпълнение**. Следователно, има връзка между понятията процес и програма, но те не са идентични. За разлика от програмата, която е нещо статично - файл записан на диска и съдържащ изпълним код, процесът е дейност. В понятието процес освен програмата се включват и текущите стойности на регистъра PC (programm counter), на другите регистри, на програмните променливи, състоянието на отворените файлове и други. В операционна система, която реализира такъв един модел, всичкия софтуер работещ на компютъра е организиран в процеси, т.е. ЦП винаги изпълнява процес и нищо друго. Когато операционната система поддържа едновременото съществуване на няколко процеса се казва, че е многопроцесна, в противен случай е еднопроцесна. Операционните системи Unix, MINIX и Linux са типични многопроцесни системи.

4.1.1. Йерархия на процесите

Операционна система, която реализира абстракцията процес, трябва да предоставя възможност за създаване на процеси и за унищожаване на процеси, а също така и начин за идентифициране (именуване) на процес.

В Unix, Linux и MINIX процес се създава със системния примитив `fork`. В тези системи най-точното определение за **процес е обект, който се създава от fork**. Когато един процес изпълни `fork` ядрото създава нов процес, който е почти точно негово копие. Първият процес се нарича процес-баща, а новият е процес-син. След `fork` процесът-баща продължава изпълнението си паралелно с новия процес-син.

Следователно, след това процесът-баща може да създаде с `fork` и други процеси-синове, т.е. един процес може едновременно да има няколко процеса-синове. Процесът-син също може да изпълни `fork` и да създаде свой процес-син. Следователно, ако разглеждаме връзките на пораждане на процеси, т.е. връзката баща-син, то всички едновременно съществуващи процеси са свързани в йерархия. Всеки процес се идентифицира чрез уникален номер, наричан ***pid*** (***process identifier***), който освен, че е уникален за процеса е и неизменен през целия му живот.

Нека разгледаме какво се случва при стартиране на Unix или Linux и някои важни процеси, които се създават. В същност първият процес не може да бъде създаден нормално, тъй като преди него няма друг процес. Програмата за начално зареждане, която е в boot блока, зарежда ядрото в паметта и предава управлението на модул, който инициализира структурите в ядрото (системните таблици) и ръчно създава процес с pid 0. От тук нататък ядрото продължава да работи като процес с pid 0 и създава нормално с `fork` процес с pid 1, в който пуска за изпълнение програмата `init`. След това процес 0 създава няколко други процеси, които се наричат системни процеси (kernel processes), и самия той става системен процес (това важи за някои версии).

Процес `init` е първият нормално създаден процес и затова ядрото го счита за корен на йерархията на процесите. Той се грижи за инициализация на процесите. Когато процес `init` заработи, чете файл `/etc/inittab` и създава процеси според съдържанието му. Например, за всеки терминал в системата `init` създава процес, в който пуска за изпълнение специална програма - `getty` в повечето версии на Unix (`mingetty` в Linux). Програмата `getty` чака вход от съответния терминал и когото той постъпи приема, че потребител иска вход в системата. Тогава програмата `getty` в процеса се сменя с програмата `login`, която извършва идентификация на потребителя и при успех се сменя с програмата `shell` за съответния потребител (или поражда нов процес за програмата `shell` в някои версии). Следователно, в живота на един процес могат последователно да се сменят различни програми, които той изпълнява, а също така няколко едновременно съществуващи процеса могат да изпълняват една и съща програма, но те са различни обекти за ядрото.

След като процесът `init` приключи с инициализацията на процесите според описанието в `/etc/inittab`, той изпълнява безкраен цикъл, в който чака завършване на свой процес-син и изпълнява довършителни дейности.

4.1.2. Състояние на процес

В многопроцесните операционни системи едновремено съществуват много процеси, а ЦП е един и във всеки един момент може да изпълнява само един от тези процеси. За този процес казваме, че се намира в състояние текущ (running). Останалите процеси са в някакво друго състояние. Най-опростеният модел на процесите включва три вида състояния:

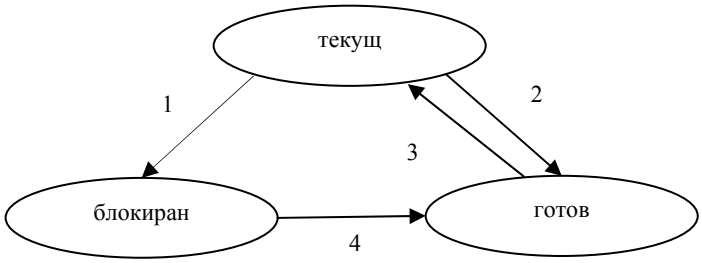
1. **текущ** (**running**)
   ЦП изпълнява команди на процеса.

2. **готов** (**ready**)
   Процесът може да продължи изпълнението си, ако му се предостави ЦП, който в момента се използва от друг процес.

3. **блокиран** (**blocked**)
   Процесът чака настъпването на някакво събитие, много често завършването на входно-изходна операция.

Логически състоянията 1 и 2 са подобни, в смисъл, че и в двата случая процесът логически е работоспособен, но при състояние 2 няма свободен ЦП. При състояние 3 е различно, процесът логически не може да работи дори ако ЦП няма какво друго да прави.

На *Фиг.4.1* е изобразена диаграмата на състоянията и възможните преходи от едно състояние в друго, които са показани с дъги. Между тези три състояния са възможни четири прехода. Преход 1 се случва когато процес открие, че не може да продължи изпълнението си, например докато не завърши входно-изходната операция, която той е поискал, изпълнявайки системния примитив `read`. Преход 4 се извършва когато настъпи събитието, което процесът чака. Преходи 2 и 3 се управляват от модул на операционната система, наричан ***планировчик*** (***scheduler***), без процесът да ги желае и дори да знае за тях. Когато планировчикът реши, че процес достатъчно дълго е използвал ЦП, той насилствено му го отнема, за да го предостави на друг готов процес, т.е. извършва за него прехода 2. Когато ЦП се освободи поради блокиране, завършване или сваляне на текущия процес, планировчикът избира един от готовите процеси за текущ, т.е. извършва за него прехода 3. Планирането, т.е. решаването кой процес да работи, кога и колко време, е важна функция в многопроцесните операционни системи, критична за производителността им. Планиране, при което се реализира прехода 2 се нарича ***планиране с преразпределение*** на ЦП (preemptive scheduling). Планирането ще бъде разгледано по-нататък.



Фиг. 4.1. Модел на процесите с три състояния и диаграма на преходите

Описаният модел дава начална представа за понятието състояние на процес, но е прекалено опростен, например в него не е ясно как работи ядрото. Моделът на процесите в UNIX System V включва девет състояния:

1. **текущ в потребителска фаза** (**user running**)
   Процесът се изпълнява в потребителски режим, като ЦП изпълнява команди от потребителската програма свързана с процеса.

2. **текущ в системна фаза** (**kernel running**)
   Процесът се изпълнява в режим ядро, като ЦП изпълнява команди от ядрото, т.е. от името на процеса работят модули на ядрото, които вършат системна работа.

3. **готов в паметта** (**ready in memory**)
   Процесът е готов за изпълнение и се намира в паметта.

4. **блокиран в паметта** (**blocked in memory**)
   Процесът чака настъпването на някакво събитие и се намира в паметта.

5. **готов на диска** (**ready, swapped**)
   Процесът е готов за изпълнение, но планировчикът трябва да го зареди в паметта преди да може да бъде избран за текущ.

6. **блокиран на диска** (**blocked, swapped**)

   Процесът е блокиран и планировчикът го е изхвърлил от паметта в специална област на диска - свопинг област, която представлява разширение на паметта, за да освободи място за други процеси.

7. **преразпределен** (**preempted**)

   Процесът е бил на път да се върне в състояние 1, след като е завършила системната фаза - състояние 2, но планировчикът му е отнел ЦП насилствено, за да го предостави на друг процес (преразпределил е ЦП).

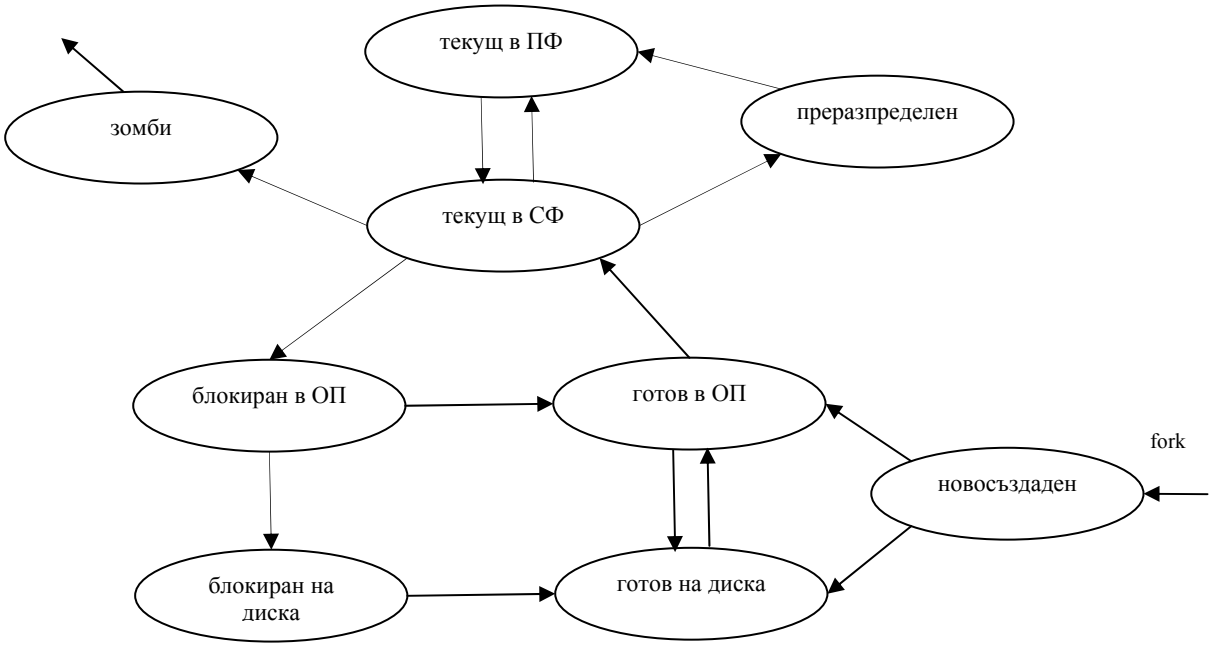
8. **новосъздаден** (**created**)

   Това е началното състояние, в което процес влиза в системата. Той е почти създаден, но още не е напълно работоспособен. Това е преходно състояние при създаване на всеки процес.

9. **зомби** (**zombie**)

   Процесът е изпълнил системния примитив `exit` и вече не съществува, но от него все още има някакви следи в системата. Съхранява се информация за него, която може да бъде предадена на процеса-баща. Това е крайното състояние на всеки процес преди да изчезне напълно от системата.

На *Фиг.4.2* са изобразени възможните преходи от едно състояние в друго.



Фиг. 4.2. Диаграма на състоянията и преходите в UNIX System V

Сега да разгледаме събитията, които предизвикват смените на състоянията. Всеки процес влиза в системата в състояние "новосъздаден", когато процес-баща изпълнява `fork`. След това преминава в състояние "готов в паметта" или "готов на диска" в зависимост от ядрото и наличните в момента свободни ресурси. Нека приемем, че процесът е преминал в състояние "готов в паметта". След време планировчикът ще го избере за текущ и той ще влезе в състояние "текущ в системна фаза", където ще довърши своята част от `fork`. След това той може да премине в състояние "текущ в потребителска фаза", където ще започне изпълнението на потребителската си програма. След време ще настъпи някакво прекъсване или в потребителската програма ще има

системен примитив. И двете събития ще предизвикат преход в състояние "текущ в системна фаза". Когато завърши обработката на прекъсването, ако това е била причината за вход в състоянието, ядрото може да реши да върне процеса обратно в състояние "текущ в потребителска фаза", или да го свали от ЦП, т.е. да извърши преход в състояние "преразпределен". Ако причината за прехода от потребителска към системна фаза е била извикване на системен примитив, то за някои примитиви, например `read` или `write`, се налага процесът да изчака, т.е. да премине в състояние "блокиран в паметта". Когато входно-изходната операция завърши, устройството ще предизвиква прекъсване и някой друг процес, който в момента е в състояние "текущ в потребителска фаза", ще влезе в състояние "текущ в системна фаза", а модулът обработващ прекъсването ще смени състоянието на първия процес от "блокиран в паметта" в "готов в паметта".

В многопроцесна система обикновено е невъзможно всички процеси да се съхраняват едновременно в паметта. Затова част от процесите временно се съхраняват в специална област на диска.Тази техника се нарича свопинг (swapping), а дисковата област - свопинг област. Специален планировчик (swapper) решава кой от процесите в състоние "блокиран в паметта" или "готов в паметта" да бъде изхвърлен временно от паметта и кой от процесите в свопинг областта да бъде върнат обратно в паметта, т.е. управлява преходите на състояния 3 в 5, 5 в 3 и 4 в 6.

Състоянията "готов в паметта" и "преразпределен" са подобни в смисъл, че процесът е готов за изпълнение когато му се предостави ЦП, но са отделени в модела, за да се подчертае факта, че процес работещ в системна фаза не може да бъде свален докато не я завърши. Следователно свопинг на процес може да се извършва и от състояние "преразпределен", т.е. има преходи от състояние 7 в 5 и обратно, които не са изобразени *на Фиг. 4.2*.

Когато процес завършва той изпълнява системния примитив `exit` и състоянието му се сменя от "текущ в потребителска фаза" в "текущ в системна фаза". Когато системната фаза на `exit` завърши състоянието на процеса се сменя в "зомби". В това състояние процесът остава докато неговият процес-баща не се погрижи чрез специален системен примитив `wait`, след което процесът напълно изчезва от системата.

4.2. КОНТЕКСТ НА ПРОЦЕС

За да се реализира този модел и възможността операционната система да управлява преходите, трябва да се съхранява информация за процесите. Сега ще разгледаме основните структурите в паметта, които представят един процес или от какво се състои един процес, имайки предвид основно Unix и Linux, въпреки, че принципите са общи за операционните системи. В Unix системите всичко, което реализира един процес, се нарича контекст на процес. Компонентите на контекста могат да се разделят на три части:

Потребителска част - определя работата на процеса в потребителска фаза и включва образа на процеса.

Машинна (регистрова) част - съдържанието на машинните регистри - PC, съдържащ адреса на следващата машинна команда; PSW, определящ режима на ЦП по отношение на процеса, признак за резултата от последната команда и др.; други регистри, съдържащи данни на процеса.

Системна част - структури в пространството на ядрото, описващи процеса, някои от които ще разгледаме по-нататък.

4.2.1. Образ на процес

Образът на процеса съдържа програмния код и данните, използвани от процеса в потребителска фаза. Той се създава като се използва файл с изпълним код. Обикновено образът на процеса се състои от логическите единици:

- ***код*** (***text***) - съдържа машинните команди, изпълнявани от процеса в потребителска фаза. Зарежда се от изпълнимия файл.
- ***данни*** (***data***) - съдържа глобалните данни, с които процесът работи в потребителска фаза. Инициализираните данни се зареждат от изпълнимия файл. За неинициализираните данни в изпълнимия файл се съхранява размера им и при създаване на образа на процеса се отделя съответния обем памет.
- ***стек*** (***stack***) - създава се автоматично с размер определян от ядрото. Чрез него се реализира обръщението към потребителски функции. Той представлява стек от слоеве, като има по един слой за всяка извикана функция, която още не е изпълнила return. Всеки слой съдържа данни, локални за функцията и достатъчно данни за връщане от функция. Това е стекът, използван при работа на процеса в потребителска фаза. При работа в системна фаза се използва друг стек - стек на ядрото, който има същата структура, но в него се записват данни при обръщение към функции от ядрото.

В Unix системите тези логически единици, на които се дели образа на процеса, се наричат области или региони (regions). Всеки регион е последователна област от виртуалното адресно пространство на процеса и се разглежда като независим обект за защита или съвместно използване. Например, регион за код обикновено е read-execute. Това позволява няколко процеса, които изпълняват една и съща програма, да могат да разделят достъпа до един и същи регион за код, т.е. да използват едно копие. Има и други типове региони, които могат да са общи за няколко процеса, например ***обща памет*** (***shared memory***). Следователно, на базата на регионите, няколко процеса могат да делят части на своите образи, но общите региони се считат за част от образа на всеки от тези процеси. Понятието регион е логическо и не зависи от реализираното управление на паметта.

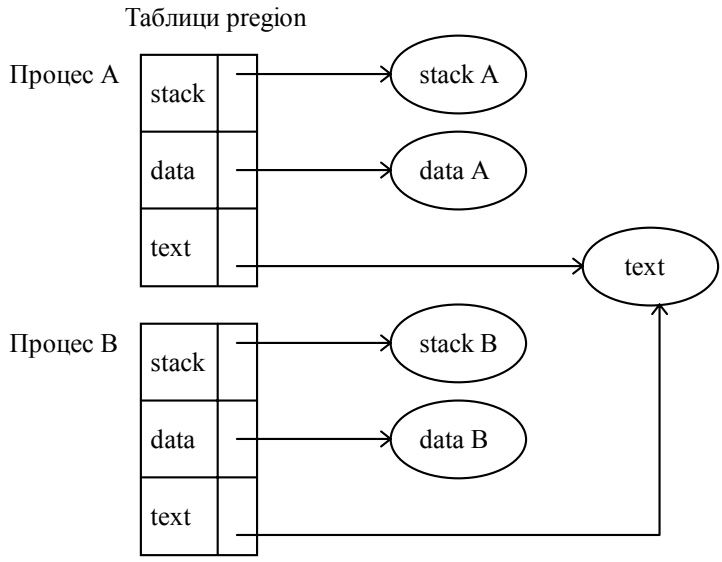
Информацията за всички активни региони се съхранява в ***Таблица на регионите***. Всеки запис в тази таблица описва един регион и съдържа:

- тип на региона - код, данни, стек и др.
- размер на региона
- адрес на региона в паметта
- състояние на региона - заключен, зарежда се в паметта и др.
- брой процеси, използващи региона.

Тъй като един регион може да е общ за няколко процеса, то всеки процес има собствена системна структура, описваща неговите региони - ***Таблица pregion*** (***Per process region table***). Всеки запис в тази таблица описва един регион на процеса и съдържа:

- тип на разрешения на процеса достъп до региона - read-only, read-write, read-execute
- виртуален адрес на региона в процеса
- указател към запис от Таблицата на регионите.

Таблицата pregion и съответните записи от Таблицата на регионите са част от системното ниво на контекста на процес, както и други системни таблици, необходими за изобразяване на виртуалното адресно пространство във физическо, в зависимост от реализираното управление на паметта. На *Фиг.4.3* са изобразени два процеса, които разделят достъпа до общ регион за код, т.е. изпълняват една програма.



Фиг. 4.3. Процеси и региони

4.2.2. Таблица на процесите

Тази структура представлява масив от записи в пространството на ядрото, като всеки запис описва един процес. Записът съдържа информация за процеса, всичко което ядрото трябва да знае независимо от състоянието му. Освен наименованието таблица на процесите, в литературата се срещат и други, като ***дескриптор на процес***, ***блок за управление на процес***. Независимо от терминологичните различия структура с такова предназначение присъства във всяка операционна система. Така че, друго определение за процес, което можем да приемем, е: **процес е обект, който е описан в запис от таблицата на процесите**.

Точната структура на запис от таблицата на процесите е различна в различните операционни системи, дори за Unix и Linux системите няма абсолютна еднаквост. Информацията, описваща един процес в тези системи, обикновено е разделена в две системни структури:

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

Потребителска област (***User area*** или ***U area***) - съдържа данни за процеса, които са необходими и достъпни за ядрото само когато той е в състояние текущ.

Следват някои от полетата, които се включват в Таблицата на процесите:

- идентификатор на процеса (pid)
- идентификатор на процеса-баща (ppid - parent pid)
- идентификатор на група процеси (pid на процеса-лидер на групата)
- идентификатор на сесия (pid на процеса-лидер на сесията)
- състояние на процеса
- събитие, настъпването на което процеса чака в състояние блокиран
- полета, осигуряващи достъп до образа на процеса и U area (адрес на таблица pregion)
- полета, определящи приоритета на процеса при планиране
- полета, съхраняващи времена - използваното от процеса време на ЦП в системна и потребителска фази и други
- код на завършване на процеса.

Следващите полета са в Потребителската област:

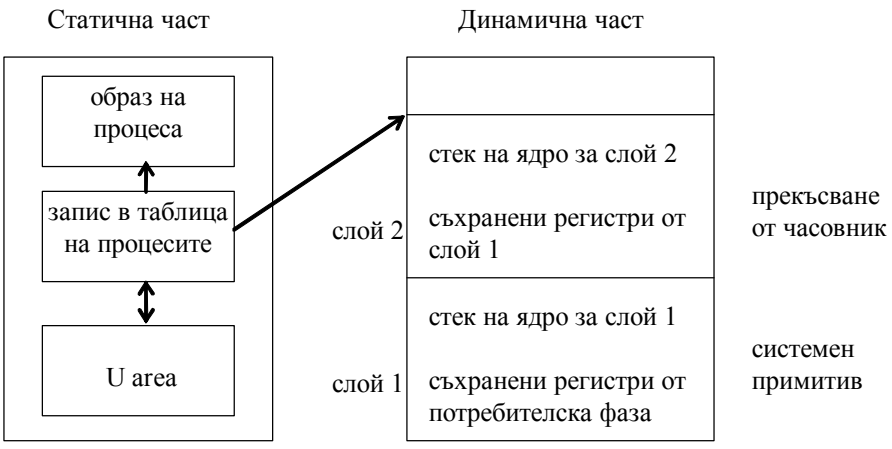
- файлови дескриптори
- текущ каталог на процеса
- управляващ терминал на процеса или `NULL` ако няма такъв
- реален потребителски идентификатор (ruid) - определя потребителя, създал процеса
- ефективен потребителски идентификатор (euid) - определя правата на процеса
- реален идентификатор на потребителска група (rgid)
- ефективен идентификатор на потребителска група (egid)
- полета за параметри на текущия системен примитив и върнати от него стойности.
- указател към записа от таблицата на процесите

4.2.3. Стек на ядрото и динамична част от контекста на процес

Когато процес работи в системна фаза е необходим стек на ядрото. Всеки процес трябва да има свой собствен стек на ядрото, който да съответства на неговото обръщение към ядрото. Освен това, когато процес преминава от потребителска в системна фаза е необходимо преди това да се съхрани съдържанието на машинните регистри, за да може по-късно процесът да се върне към прекъснатата потребителска фаза. Следователно за всеки процес контекстът включва и стек на ядрото и област за съхранение на регистрите.

В същност нещата са по-сложни, тъй като в диаграмата на преходите (*Фиг.4.2*) не е показан един преход от състояние 2 (текущ в системна фаза) в състояние 2 и обратно. Докато процес работи в системна фаза, обработвайки системен примитив или прекъсване, може да настъпи прекъсване с по-висок приоритет. Тогава процесът прекъсва първата системна фаза и започва нова системна фаза, което означава, че трябва да се съхранят регистрите от старата системна фаза и да се започне нов стек на ядрото за новата системна фаза. Така когато приключи обработката на новата системна фаза ще е възможно процесът да се върне към прекъснатата системна фаза.

Затова контекстът включва динамична част, която представлява стек от слоеве. Слой се записва при прекъсване или системно извикване и включва съхранените регистри от работата на процеса преди прекъсването и стек на ядрото, използван при обработка на новото прекъсване. Слой се изключва при връщане след обработка на прекъсването или на системния примитив. Когато процес работи в потребителска фаза, динамичната част от контекста е празна. Процес, работещ в системна фаза, се изпълнява в контекста на последния записан слой. Броят на нивата на прекъсване е хардуерно зависим и това ограничава максималния брой слоеве в динамичната част. *Фиг.4.4* илюстрира компонентите, съставляващи контекста на процес.



Фиг. 4.4. Компоненти на контекста на процес

Регистровият контекст на процес трябва да се съхранява и когато текущият процес се сваля от ЦП (процесът преминава в състояние преразпределен или блокиран) и да се възстановява когато планировчикът отново избере процеса за текущ. Смяната на контекста на текущия процес с контекста на друг процес се нарича ***превключване на контекста*** (***context switch***). Има три случая, в които се прави превключване на контекст:

1. Текущият процес се блокира.
2. Текущият процес е изпълнил системния примитив `exit` и минава в състояние зомби.
3. Текущият процес е завършил системната си фаза, след обработка на системен примитив или на прекъсване и ще трябва да се върне в потребителска фаза, но има готов процес с по-висок приоритет и планировчикът решава да свали текущия процес от ЦП.

И в трите случая текущият процес, който е в състояние "текущ в системна фаза", излиза от това състояние (всички системни операции са завършени и структурите в ядрото са в коректно състояние) и не може или не трябва да продължи изпълнението си. Тогава ядрото трябва да избере друг готов процес (в състояние "готов в паметта" или "преразпределен") за текущ. Стъпките, които ядрото изпълнява са следните:

1. Решава дали да прави превключване на контекста.
2. Съхранява контекста на "стария" процес, т.е. записва слой в динамичната част на неговия контекст (push в динамичната част).
3. Избира "най-подходящия" процес за текущ, използвайки алгоритъма за планиране.
4. Възстановява контекста на избрания процес, използвайки най-горния слой в динамичната част на неговия контекст (pop от динамичната част). От тук нататък системата продължава да работи в контекста на "новия" процес.

4.2.4. Системни примитиви - интерфейс и изпълнение

В програма на езика Си извикването на системен примитив изглежда като обръщение към функция, но всеки системен примитив е вход в ядрото. Как се реализира това? За всеки системен примитив стандартната библиотека на Си включва функция (в някои случаи има няколко функции за един системен примитив). Тези функции се свързват с потребителската програма и се изпълняват в потребителска фаза. Това, което извършват тези функции, е по принцип едно и също за всеки системен примитив, а именно:

1. Зарежда номера системния примитив в регистър.
2. Изпълнява команда, предизвикваща програмно прекъсване.
3. След връщане от прекъсване проверява за грешка регистъра PSW и преобразува връщаните стойности към формата, използван от функциите на системните примитиви (код на грешката в глобалната променлива `errno` и връщаната от функцията стойност в регистър 0).

Следователно фактическата обработка на системните примитиви се върши от модули в ядрото, а кода на библиотечните функции служи като обвивка, която осигурява по-удобен потребителски интерфейс. За простота на езика обаче, ние наричаме библиотечните функции системни примитиви.

При изпълнение на команда за програмно прекъсване в библиотечната функция заработва системата за прекъсване. Ядрото обработва всички прекъсвания по следния сценарий:

1. Съхранява текущия регистров контекст, като записва слой в динамичната част на контекста (слой 1).
2. Определя източника на прекъсване и от вектора на прекъсване извлича адреса на съответния обработчик на прекъсването.
3. Извиква обработчика на прекъсването.
4. Възстановява регистровия контекст (съхранен в стъпка 1), т.е. става връщане от прекъсване.

При програмно прекъсване управлението в стъпка 3 на горния сценарий се предава на модул от ядрото, който получава като вход номера на системния примитив. Следва алгоритъма на този модул.

Алгоритъм на syscall (номер на системния примитив)

1. Намира записа в таблицата на системните примитиви, който съответства на получения номер, т.е. определя адреса на модула в ядрото за извикания системен примитив и необходимия брой параметри.
2. Копира параметрите от потребителския стек в област на ядрото - U area.
3. Извиква модула в ядрото за системния примитив.

Връщаните от системния примитив стойности се записват в областта за съхранените регистри от потребителска фаза (слой 1 от динамичната част на контекста). При грешка в регистър PSW се вдига бит, а в регистър 0 се записва код на грешката, иначе при нормално завършване в регистри 0 и 1 се записват връщаните от системния примитив стойности.

4.3. СИСТЕМНИ ПРИМИТИВИ ЗА УПРАВЛЕНИЕ НА ПРОЦЕСИ

Ще разгледаме системните примитиви по стандарта POSIX, които реализират основните операции за процеси, а именно създаване на процес, завършване на процес и други.

Създаване на процес

```
#include <sys/types.h>
pid_t fork(void);
```

Единственият начин да се създаде процес в Unix и Linux системите е чрез `fork`. Процесът, който изпълнява `fork` се нарича процес-баща, а новосъздаденият е процес-син. При връщане от `fork` двата процеса имат еднакви образи, с изключение на връщаната стойност: в процеса-баща `fork` връща pid на процеса-син при успех или -1 при грешка, а в сина връща 0. Сложността при този примитив се състои в това, че един процес "влиза" във функцията `fork`, а два процеса "излизат от" `fork` с различни връщани значения. Да разгледаме по-подробно това, което става когато процес-баща изпълнява `fork` и така ще разберем какво е общото между процесите баща и син.

Алгоритъм на `fork`

1. Определя уникален идентификатор за новия процес и създава запис в таблицата на процесите, като инициализира полетата в него (група и сесия от процеса-баща, състояние "новосъздаден" и т.н.).
2. Създава U area, където повечето от полетата се копират от процеса-баща (файловите дескриптори, текущ каталог, управляващ терминал, потребителските идентификатори - euid, ruid, egid и rgid).

3. Създава образ на новия процес - копие на образа на процеса-баща (регионът за код обикновено е общ за двата процеса).
4. Създава динамичната част от контекста на новия процес: Слой 1 е копие на слой 1 от контекста на бащата. Създава нов слой 2, в който записва съхранените регистри от работата в слой 1, като регистър PC е изменен така, че синът да започне изпълнението си в `fork` от стъпка 7.
5. Изменя състоянието на процеса-син в "готов в паметта".
6. В процеса-баща връща pid на новосъздадения процес-син.
7. В процеса-син, връща 0.

Следователно, когато по-късно планировчикът избере новосъздадения процес за текущ той ще заработи според най-горното ниво на динамичната част от контекста си (слой 2) - в системна фаза на `fork` и ще върне 0.

И така процесите син и баща имат следното общо помежду си:

- Двата процеса изпълняват една и съща програма. Дори процесът-син, който преди това не е работил, започва изпълнението на потребителската програма от оператора след `fork`. Но връщаните стойности в двата процеса са различни, което позволява да се определи кой процес е бащата и кой сина и да се раздели тяхната функционалност макар, че изпълняват една и съща програма.
- Процесът-син наследява от бащата файловите дескриптори, т.е. двата процеса ще разделят достъпа до файловете, които бащата е отворил преди да изпълни `fork`. Това позволява пренасочване на входа и изхода и свързване на процеси с програмни канали.
- Двата процеса имат един и същ текущ каталог, управляващ терминал, група процеси и сесия.
- Двата процеса имат еднакви права - реален и ефективен потребителски идентификатори, което гарантира неизменност на привилегиите на потребител при работа в системата.

Има два основни начина за използване на `fork`. Процес иска да създаде свое копие, което да изпълнява една операция, докато другото копие изпълнява друга, например при процеси сървери. Процес иска да извика за изпълнение друга програма, тогава първо създава свое копие, след което в едното копие (обикновено в процеса-син) се извиква другата програма. Това се използва от командния интерпретатор.

Завършване на процес

`void exit(int status);`

Системата не поставя ограничение за времето на съществуване на процес. Има процеси, например процес `init` (с pid 1), които съществуват вечно в смисъл от boot до shutdown. Когато процес завършва той изпълнява `exit`. Процесът може явно да го извика или неявно в края на програмата, но може и ядрото вътрешно да извика `exit` без знанието на процеса, например когато процеса получи сигнал, за който не е предвидил друга реакция. Значението на аргумента `status` е кода на завършване, който се предава на процеса-баща, когато той се заинтересува. Този системен примитив не връща нищо, защото няма връщане от него, винаги завършва успешно и след него процесът почти не съществува, т.е. той става зомби.

Алгоритъм на `exit`

1. Изпълнява `close` за всички отворени файлови дескриптори и освобождава текущия каталог.

2. Освобождава паметта, заемана от образа на процеса и потребителската област.
3. Сменя състоянието на процеса в зомби. Записва кода на завършване в таблицата на процесите. Ако процесът завършва по сигнал, код на завършване е номера на сигнала.
4. Урежда изключването на процеса от йерархията на процесите. Ако процесът има синове, то техен баща става процесът `init` и ако някой от тези синове е зомби изпраща сигнал "death of child" на `init`. Изпраща същия сигнал и на процеса-баща на завършващия процес.

Изчакване завършването на процес-син

```
#include <sys/wait.h>
#include <sys/types.h>

pid_t wait(int *status);
```

Процес-баща може да синхронизира работата си със завършването на свой процес-син, т.е. да изчака неговото завършване ако още не завършил и да разбере как е завършил, чрез `wait`. Функцията на системния примитив връща pid на завършилия син, а чрез аргумента *`status`* кода му на завършване.

Алгоритъм на `wait`

1. Ако процесът няма синове, това е грешка и функцията връща -1.
2. Ако процесът има син в състояние зомби, т.е. синът вече е изпълнил `exit`, освобождава записа му от таблицата на процесите, като взема кода на завършване и неговия pid и ги връща.
3. Ако процесът има синове, но никой от тях не е зомби, той се блокира, като чака сигнал "death of child". Когато получи такъв сигнал, което означава, че някой негов син току що е станал зомби, продължава както в точка 2.

Сега след като разгледахме системните примитиви `fork`, `exit` и `wait`, става по-ясно значението на състоянието зомби. След `fork` двата процеса - баща и син съществуват едновременно и се изпълняват асинхронно. Това означава, че бащата може да изпълни `wait` както преди така и след `exit` на сина. Освен това не бива да се задължава процес-баща да изпълнява `wait`, той може да завърши веднага след като е създал син. Но тогава в системата ще останат вечни зомбита, които заемат записи в таблицата на процесите. Затова когато процес завършва неговите синове се осиновяват от процеса `init`, който се грижи за изчистване на системата от зомбита-сираци.

Изпълнение на програма

Когато с `fork` се създава нов процес, той наследява образа си от бащата, т.е. продължава да изпълнява същата програма. Но чрез `exec` всеки процес може да смени образа си с друга програма по всяко време от своя живот, както веднага след създаването си така и по-късно, дори няколко пъти в своя живот. За този системен примитив има няколко функции в стандартната библиотека, които се различават по начина на предаване на аргументите и използване на променливите от обкръжението на процеса. Следва синтаксиса на част от тях.

```
int execl(const char *name, const char *arg0
          [, const char *arg1]..., 0);

int execlp(const char *name, const char *arg0
          [, const char *arg1]..., 0);
```

```
int execv(const char *name, const char *argv[]);
int execvp(const char *name, const char *argv[]);
```

Първият аргумент *name* указва към името на файл, съдържащ изпълним код, от който ще се създаде новия образ. При `execvp` и `execlp` *name* може да е собствено име на файл и тогава файлът се търси в каталозите от променливата `PATH`. В останалите случаи *name* трябва да е пълно име на файл. Останалите аргументи в `execl` и `execlp` са указатели към аргументите, които ще се предадат на функцията `main` когато новият образ започне изпълнението си. Във функциите `execv` и `execvp` има един аргумент *argv*, който е масив от указатели на аргументите за функцията `main`, който също трябва да завършва с елемент `NULL`.

Алгоритъм на `exec`

1. Намира файла, чието име е в аргумента *name* и проверява дали процесът има право за изпълнение на този файл.
2. Проверява дали файлът съдържа изпълним код.
3. Освобождава паметта, заемана от стария образ на процеса.
4. Създава нов образ на процеса, използвайки изпълнимия код във файла и копира аргументите на `exec` в новия потребителски стек.
5. Изменя значенията на някои регистри в областта за съхранени регистри в слой 1 от динамичната част на контекста, напр. на PC, указател на стек. Така, когато процесът се върне в потребителска фаза ще заработи от началото на функцията `main` на новия образ.
6. Ако програмата е set-UID прави съответните промени на потребителските идентификатори на процеса.

При успех, когато процесът се върне от `exec` в потребителска фаза, той изпълнява кода на новата програма, започвайки от началото, но това си остава същия процес. Не е променен идентификатора му, позицията му в йерархията на процесите дори голяма част от потребителската област, като файловите дескриптори, текущия каталог, управляващия терминал, групата на процесите, сесията. При грешка по време на `exec` става връщане в стария образ, така че функцията връща -1 при грешка, а при успех не връща нищо защото няма връщане в стария образ.

Разделянето на създаването на процес и извикването на програма за изпълнение в два системни примитива играе важна роля. Това дава възможност програмата на процес-баща да определя поне в началото функционалността на свой процес-син и например, да се реализира пренасочване на входа и изхода, и свързването на роднински процеси чрез програмни канали (заслуга за това има и наследяването на файловите дескриптори).

Информация за процес

```
pid_t getpid(void);
pid_t getppid(void);
```

Системният примитив `getpid` връща идентификатора на процеса, който го изпълнява, а `getppid` връща идентификатора на неговия процес-баща. И двата примитива винаги завършват успешно.

Пример. Програмата илюстрира създаване на нов процес.

```
/* ------------------------------------------------------------ */
#include <sys/types.h>
main(void)
{
  pid_t pid;
  pid = fork();
  if (pid < 0) {                /* in parent process */
      perror("Cannot fork");
     exit(1); }
  else if (pid == 0)            /* in child process */
      printf("PID %d: Child started, parent is %d\n",
              getpid(),         /* Current (child) PID */
              getppid());       /* Parent PID */
  else {                              /* in parent process */
      printf("PID %d: Started child PID %d\n",
              getpid(),         /* Current (parent) PID */
              pid);             /* Child PID */
      sleep(3);                 /* Wait 3 seconds */
  }
  exit(0);
}
/* ------------------------------------------------------------ */
```

Процесът, в който се изпълнява програмата, създава нов процес. В процеса-син се изпълнява същата програма, но двата процеса извеждат различни съобщения на стандартния изход.

Пример. Програмата илюстрира създаване на процес и изпълнение на друга програма.

```
/* ------------------------------------------------------------ */
#include <sys/types.h>
main(void)
{
  pid_t pid;
  int status;
  pid = fork();
  if (pid == 0) {          /* in child process */
     execl("/bin/ls", "ls", "-l", 0);
     perror("Cannot exec ls");
     exit(1);}
  else                     /* in parent process */
     if (pid < 0) {
        perror("Cannot fork");
       exit(1);}
     else {
        wait(&status);
        printf("Parent after death of child: status=%d.\n", status);
        exit(0);}
}
/* ------------------------------------------------------------ */
```

Процесът, в който се изпълнява програмата, създава нов процес. В процеса-син се изпълнява командата `ls`. Процесът баща изчаква завършването на сина и извежда съобщение на стандартния изход за кода му на завършване с функцията `printf`. Съобщенията за грешки се извеждат с функцията `perror`. Тя извежда на стандартния изход за грешки, като освен текста в аргумента си извежда и системно съобщение за грешката в изпълнения преди нея системен примитив.

4.4. МЕЖДУПРОЦЕСНИ КОМУНИКАЦИИ

В този раздел ще разгледаме проблемите при междупроцесни комуникации (Interprocess Communication или IPC) и някои механизми за решаването им. Проблемите при комуникации между процеси имат два аспекта. Първият е предаване на информация между процесите. Другият е свързан със съгласуване на действието на процесите, които работят асинхронно, така че да се гарантира правилното им взаимодействие.

Най-простият начин, по който два или повече процеса могат да взаимодействат е да се конкурират за достъп до общ ресурс. Например, два процеса P и Q четат и пишат в обща променлива брояч `counter`, като всеки увеличава променливата. Нека значението на `counter` е 7 и достъпът на двата процеса до нея се извърши в следния ред:

Процес P	**Процес Q**
1.Чете `counter` в локална променлива `pa`.	
	2.Чете `counter` в локална променлива `pb`.
	3. `pb = pb + 2`
	4. Записва `pb` в `counter`.
5. `pa = pa + 1`	
6. Записва `pa` в `counter`.	

При тази последователност на изпълнение на двата процеса резултатът в `counter` ще е неправилен - 8, а не 10, тъй като изменението на процеса Q ще се загуби. Такава ситуация, при която два или повече процеса четат и пишат в обща памет и крайният резултат зависи от реда, в който работят процесите се нарича ***състезание*** (***race condition***). Как да се избегне състезанието? Решението на този проблем се нарича ***взаимно изключване*** (***mutual exclusion***), т.е. по такъв начин да се организира работата на двата (или повече) процеса, че когато един от тях осъществява достъп до общия ресурс (изпълнява трите стъпки в примера) за другия (другите) да се изключи възможността да прави същото.

Друг по-сложен начин на взаимодействие на два или повече процеса е когато те извършват обща работа. Съществуват няколко класически модела на взаимодействие на процеси, извършващи обща работа, например:

- Производител-Потребител (The Producer-Consumer Problem)
- Читатели-Писатели (The Readers and Writers Problem)
- Задачата за обядващите философа (The Dining Philosophers Problem)

Това, което е необходимо за коректното взаимодействие на процесите (освен евентуално взаимно изключване), се нарича ***синхронизация*** (***synchronization***). Например, в задачата Производител-Потребител синхронизацията изисква всяка произведена от Производителя данна да се предаде на Потребителя и обработи точно веднаж, като нищо не се загуби.

4.4.1. Взаимно изключване

Проблемът за избягване на състезанието е бил формулиран от Е. Дейкстра (E.Dijkstra) чрез термина ***критичен участък*** (***critical section***). Част от кода на процеса реализира вътрешни изчисления, които не могат да доведат да състезание. В друга част от кода си процесът осъществява достъп да обща памет или върши неща, които могат да доведат до състезание. Тази част от програмата ще наричаме критичен участък и ще казваме, че процес е в критичния си участък, ако е започнал и не е завършил изпълнението му, независимо от състоянието си. За избягване на състезанието и

коректното взаимодействие на конкуриращите се процеси трябва да са изпълнени следните условия:

1. Във всеки един момент най-много един процес може да се намира в критичния си участък (това е взаимно изключване).
2. Никой процес да не остава в критичния си участък безкрайно дълго.
3. Никой процес, намиращ се вън от критичния си участък, да не пречи на друг процес да влезе в своя критичен участък.
4. Решението не бива да се основава на предположения за относителните скорости на процесите.

Ще разгледаме решения на задачата за взаимното изключване чрез използване само на обща памет. Холандският математик Т. Декер (T.Dekker) пръв предложи решение на тази задача, а по-късно бе предложено още едно решение от Питерсон (G.Peterson). Ще разгледаме двата алгоритъма в най-простите им варианти за два процеса. Но преди това ще започнем с едно не съвсем коректно решение.

Алгоритъм с редуване на процесите

В този алгоритъм и останалите в раздела общата памет ще записваме като деклариране на променлива с думата `shared`. Този алгоритъм използва една обща променлива `turn`, чието значение е номер на процес, който е на ред да влезе в критичния си участък.

```
#define TRUE 1
shared int turn=0;
```

```
P0()
{
while (TRUE) {
      while (turn != 0);
      critical_section0();
      turn = 1;
      noncritical_section0();
      }
}
```

```
P1()
{
while (TRUE) {
      while (turn != 1);
      critical_section1();
      turn = 0;
      noncritical_section1();
      }
}
```

Двата процеса строго се редуват. Недостатък в това решение е нарушение на изискване 3. Ако един от процесите е по-бавен това ще пречи на другия процес да влиза по-често в критичния си участък, или ако един от процесите завърши другият повече няма въобще да може да влиза в критичния си участък.

Алгоритъм на Декер

Този алгоритъм използва три общи променливи за осигуряване на взаимно изключване. Променливите `wants0` и `wants1` са флагове за всеки от процесите. Флаг `FALSE` означава, че съответният процес не е в критичния си участък и не иска вход. Когато процес иска вход в критичния си участък, той вдига флага си (`TRUE`). И този алгоритъм използва променлива `turn`, чието значение е номер на процес, който е на ред да влезе в критичния си участък, но двата процеса не се редуват строго. Променливата се използва когато и двата процеса желаят вход в критичните си участъци, за да разреши конфликта.

```
#define FALSE 0
#define TRUE 1
```

```
shared int wants0=FALSE, wants1=FALSE;
shared int turn=0;
```

```
P0()
{
while (TRUE) {
      wants0 = TRUE;
      while (wants1)
        if (turn == 1) {
            wants0 = FALSE;
            while (turn == 1);
      wants0 = TRUE; }
      critical_section0();
      turn = 1;
      wants0 = FALSE;
      noncritical_section0();
      }
}
```

```
P1()
{
while (TRUE) {
      wants1 = TRUE;
      while (wants0)
        if (turn == 0) {
            wants1 = FALSE;
            while (turn == 0);
            wants1 = TRUE; }
      critical_section1();
      turn = 0;
      wants1 = FALSE;
      noncritical_section1();
      }
}
```

Алгоритъм на Питерсон

Алгоритъмът на Питерсон също използва три общи променливи за осигуряване на взаимно изключване на два процеса. Променливите `interested[2]` също са флагове на процесите. И тук се използва променлива `turn`, чието значение е номер на процес, който е на ред да влезе в критичния си участък. Но начинът, по който променливата `turn` се изменя и проверява, тук е по-различен.

```
#define FALSE 0
#define TRUE 1
shared int turn=0;
shared int interested[2]={FALSE, FALSE};
```

```
P0()
{
while (TRUE) {
      enter_region(0);
      critical_section0();
      leave_region(0);
      noncritical_section0();
      }
}
```

```
P1()
{
while (TRUE) {
      enter_region(1);
      critical_section1();
      leave_region(1);
      noncritical_section1();
      }
}
```

```
enter_region(int process)
{
int other;
      other = 1 - process;
      interested[process] = TRUE;
      turn = process;
      while (turn == process && interested[other] == TRUE);
}

leave_region(int process)
{
      interested[process] = FALSE;
}
```

Основният недостатък и на двата алгоритъма е, че за осигуряване на взаимното изключване се използва активно чакане (busy waiting). Всеки от процесите, който желае да влезе в критичния си участък изпълнява цикъл, в който непрекъснато проверява дали това е възможно, докато стане възможно. Много по-естествено и ефективно би било, когато процес не може да влезе критичния си участък да бъде блокиран и когато влизането стане възможно операционната система да го събуди. Освен това и двата алгоритъма реализират взаимно изключване на два процеса. Алгоритъм за взаимно изключване на n процеса, известен като Bakery algorithm, е бил предложен от Лампорт (L.Lamport), но е по-сложен.

4.4.2. Семафори

През 1965г. Дейкстра предложи нов механизъм за междупроцесни комуникации и го нарече ***семафори*** (***semaphores***). С всеки семафор се свързва цяла променлива - брояч и списък на чакащи в състояние блокиран процеси. Значението на брояча е неотрицателно число. За семафорите Дейкстра определи три операции - инициализация, операции `P` и `V`. При инициализацията се създава нов обект семафор и се зарежда начално значение в брояча му, което е неотрицателно цяло число. Тази операция ще записваме като деклариране и инициализиране на променлива, например:

```
semaphore s = 1;
```

Операцията `P` проверява и намалява значението на брояча на семафора, ако това е възможно, в противен случай блокира процеса и го добавя в списъка на чакащи процеси, свързан със съответния семафор. Събитието, което блокирания процес чака, е увеличение на брояча на семафора. Това събитие ще настъпи когато друг процес изпълни операцията `V` над същия семафор. Операцията `V` събужда един блокиран по семафора процес, ако има такива, а в противен случай увеличава брояча на семафора, т.е. запомня едно събуждане. За операциите `P` и `V` се използва и обозначението `down` и `up`. Алгоритмите на двете операции са следните:

```
P(s)
{
   if ( s > 0 )  s = s - 1;
   else  блокира процеса, изпълняващ P по семафора s;
}

V(s)
{
   if (има блокирани по семафора s процеси) събужда един процес;
   else  s = s + 1;
}
```

Съществото за двете операции е, че са неделими (атомарни), т.е. никой процес, изпълняващ `P(s)` или `V(s)` не може да бъде преразпределен и докато един процес изпълнява операция над семафор `s` друг процес не може да започне операция над `s` докато първата не завърши. Тази неделимост на операциите може да бъде осигурена при реализацията им в ядрото като системни примитиви. Модулите, реализиращи операциите, са част от ядрото на операционната система и там за осигуряване на взаимно изключване може да се използва забрана на прекъсванията или други апаратни средства за взаимно изключване.

Взаимно изключване чрез семафор

За осигуряване на взаимното изключване на произволен брой процеси е необходим един семафор, инициализиран с 1. Освен това всеки от процесите трябва да загради критичния си участък с операциите `P` и `V` над този семафор. Такъв семафор се нарича двоичен, тъй като значенията, които заема брояча му са само две - 0 и 1.

```
#define TRUE 1
semaphore mutex=1;
```

`P0()`
`{`
`while (TRUE) {`
`P(mutex);`
`critical_section0();`
`V(mutex);`
`noncritical_section1();`
`}`
`}`
	`P1()`
	`{`
	`while (TRUE) {`
	`P(mutex);`
	`critical_section1();`
	`V(mutex);`
	`noncritical_section1();`
	`}`
	`}`
		`...`

Синхронизация чрез семафори

Съществуват няколко класически задачи за междупроцесни комуникации и качествата на всеки нов механизъм се демонстрират чрез решаването на тези задачи. Сега ще разгледаме решаването на някои от тези задачи чрез механизмите обща памет и семафори, но преди това един по-прост пример. Имаме два процеса `P1` и `P2`, които работят асинхронно и искаме операторите `S1` в процеса `P1` да се изпълнят преди `S2` в `P2`. Решението е следното:

```
semaphore s=0;
```

`P0()`
`{`
`S1;`
`V(s);`
`}`
	`P1()`
	`{`
	`P(s);`
	`S2;`
	`}`

Задачата Производител - Потребител

Задачата Производител-Потребител е модел на един начин за взаимодействие на два процеса, познат ни от командните езици в Unix и Linux. Когато командният интерпретатор изпълнява конвейер, например:

```
who | wc -l
```

той решава задачата Производител-Потребител. Има два процеса, които взаимодействат, като извършват обща работа. Процесът-производител (`who`) произвежда данни, които се предават на процеса-потребител (`wc`), който ги използва. Командните интерпретатори в Unix и Linux решават тази задача чрез програмен канал.

Тук ще разгледаме решение, в което използваме обща памет за предаване на данните от производителя към потребителя и семафори за синхронизация на работата им. Предполагаме, че двата процеса използват общ буфер, който може да поеме N елемента данни (нека за простота елементите са цели числа). Организацията на буфера няма отношение към проблема за синхронизация, затова няма да я конкретизираме. Производителят записва в буфера всеки произведен от него елемент, а потребителят

чете от буфера елементите, за да ги обработи. Двата процеса работят едновременно, с различни и неизвестни относителни скорости. Следователно, задачата за синхронизация се състои в това да не се позволи на производителя да пише в пълен буфер, да не се позволи на потребителя да чете от празен буфер и всеки произведен елемент да бъде обработен точно един път. Освен това трябва да се осигури и взаимно изключване при достъп до общия буфер. Решението на Дейкстра използва три семафора: `mutex` за взаимното изключване, `empty` и `full` за синхронизацията. Броячът на `empty` съдържа броя на свободните места в буфера и по него производителят ще се блокира когато няма място в буфера. Семафорът `full` ще брои запълнените места в буфера и по него потребителят ще се блокира когато в буфера няма данни.

```
#define N 100
#define TRUE 1

shared buffer buf;    /* общ буфер с капацитет N елемента */
semaphore mutex = 1;  /* за взаимното изключване */
semaphore empty = N;  /* брои свободните елементи в буфера */
semaphore full = 0;   /* брои запълнените елементи в буфера */
```

```
producer()
{
int item;
while (TRUE) {
      produce_item(&item);
      P(empty);
      P(mutex);
      insert_item(&item);
      V(mutex);
      V(full);
      }
}
```

```
consumer()
{
int item;
while (TRUE) {
      P(full);
      P(mutex);
      remove_item(&item);
      V(mutex);
      V(empty);
      consume_item(item);
      }
}
```

Задачата Читатели - Писатели

Тази задача е модел на достъп до обща база данни от много конкурентни процеси, които се делят на два вида. Единият вид са процеси-читатели, които само четат данните в базата, а другият вид са процеси-писатели, които изменят по някакъв начин данните в базата. Искаме във всеки момент достъп до базата данни да могат да осъществяват или много процеси-читатели или един процес-писател. Тази задачата за синхронизация е решена от Куртоа, Хейманс и Парнас чрез една обща променлива брояч и два двоични семафора.

```
#define TRUE 1
shared int rcount=0;  /* брой процеси-читатели, които четат */
semaphore mutex=1;    /* управлява достъпа до rcount */
semaphore db=1;       /* управлява достъпа до базата данни */

reader()
{
while(TRUE) {
      P(mutex);
      rcount  = rcount + 1;
      if (rcount == 1) P(db);
      V(mutex);
      read_data_base();
```

```
      P(mutex);
      rcount  = rcount - 1;
      if (rcount == 0) V(db);
      V(mutex);
      use_data_read();
      }
}

writer()
{
while (TRUE) {
      collect_data();
      P(db);
      write_data_base();
      V(db);
      }
}
```

Ако никой от процесите не осъществява достъп до базата данни, то двата семафора са 1 и `rcount` е 0, тогава първият процес, който се появи ще изпълни `P(db)` и ще затвори този семафор (броячът му ще стане 0). Ако това е процес-читател, то `rcount` ще стане 1 и следващите читатели няма да проверяват семафора `db`, а само ще увеличават `rcount` и ще започват да четат базата данни. Когато читател завърши четенето той намалява `rcount`, а последният читател изпълнява `V(db)` и ще отвори семафора `db` (ще събуди един писател, блокиран по `db`, или броячът на `db` ще стане 1).

Ако първият процес, получил достъп до базата данни е писател, то първият читател ще се блокира по семафора `db`, следващите читатели ще се блокират по `mutex`, а следващите писатели ще се блокират по `db`. Когато писателят завърши писането той ще изпълни `V(db)` и с това ще събуди един процес, чакащ за достъп до базата данни. Ако това е първият чакащ читател, той ще изпълни `V(mutex)` и ще събуди следващия чакащ читател, който ще събуди следващия и т.н. докато всички читатели бъдат събудени.

Недостатък на това решение е, че то дава преимущество на читателите, което в една реална база данни не е добра стратегия.

Задачата за обядващите философи

Задачата е модел на поведението на процеси, които се състезават за монополен достъп до ограничен брой ресурси. И тя е поставена и решена за първи път от Дейкстра чрез семафори и обща памет. Задачата се състои в следното. Пет философа седят около кръгла маса, като пред всеки от тях има чиния със спагети, а между всеки две чинии има само по една вилица. Животът на всеки философ представлява цикъл, в който той размишлява, в резултат на което огладнява и се опитва да вземе двете вилици около своята чиния. Ако успее известно време се храни, а след това връща вилиците. Задачата е да се напише програма, която да прави това, което се очаква от философа.
Едно очевидно, но грешно решение е следното.

```
#define N 5           /* броя на философите */
#define TRUE 1

philosopher(int i)    /* i е номер на философа, от 0 до 4 */
{
while(TRUE) {
   think();
   take_fork(i);     /* взима лявата вилица,като чака докато тя
```

```
                        стане достъпна */
   take_fork((i+1)%N); /*взима дясната вилица, като също чака*/
   eat();
   put_fork(i);        /* връща лявата вилица */
   put_fork((i+1)%N);  /* връща дясната вилица */
   }
}
```

Всеки философ изпълнява функцията `philosopher`. Грешката в това решение е, че може да доведе до дедлок. Ако всичките пет философа вземат едновременно левите си вилици, то никой няма да може да вземе дясна вилица и всички ще останат вечно блокирани.

Теоретически правилно и несложно решение, може лесно да се получи от горното като петте оператора след `think()` се направят критичен участък. Но от практическа гледна точка това решение не е добро, защото с наличните пет вилици във всеки момент биха могли да се хранят едновременно два философа, а при това решение във всеки момент се храни най-много един философ, а останалите гладни чакат.

Решението на Дейкстра използва общ масив `state[N]`, като всеки елемент описва състоянието на съответния философ. Възможните състояния са: мисли, гладен е (опитва се да вземе двете вилици) и храни се. Достъпът до общия масив се регулира от двоичен семафор `mutex`. Освен него се използва масив от семафори `s[N]`, по един за всеки философ, по който той се блокира когато трябва да чака освобождаването на вилиците.

```
#define N 5
#define TRUE 1
#define LEFT  (i-1)%N   /* номер на левия съсед на философ i */
#define RIGHT (i+1)%N   /* номер на десния съсед на философ i */
#define THINKING 0      /* философът мисли */
#define HUNGRY 1        /* философът се опитва да вземе вилици */
#define EATING 2        /* философът се храни */

shared int state[N]={0,0,0,0,0};
semaphore mutex=1;          /* за взаимно изключване на state[N] */
semaphore s[N]={0,0,0,0,0};  /* семафори, по един на философ */

philosopher(int i)
{
while(TRUE) {
      think();
      take_forks(i);
      eat();
      put_forks(i);
      }
}

take_forks(int i)
{
      P(mutex);
      state[i] = HUNGRY;
      test(i);
      V(mutex);
      P(s[i]);
}
```

```
put_forks(int i)
{
      P(mutex);
      state[i] = THINKING;
      test(LEFT);
      test(RIGHT);
      V(mutex);
}

test(int i)
{
if(state[i]==HUNGRY&&state[LEFT]!=EATING&&state[RIGHT]!=EATING) {
      state[i] = EATING;
      V(s[i]);
    }
}
```

При използването на обща памет и семафори за междупроцесни комуникации отговорността за правилното взаимодействие на процесите е на програмиста. Поради това този метод не е безопасен. Грешки в програмите могат да доведат до дедлок, състезание и други форми на трудно предсказуемо и невъзпроизводимо поведение. Съществува друг по-безопасен метод за комуникация, при който процесите обменят съобщения.

4.4.3. Съобщения

За да комуникират два процеса чрез механизъм на съобщения между тях трябва да се установи комуникационна връзка (communication link). След това процесите обменят съобщения чрез два примитива:

```
send(destination, message)

receive(source, message)
```

Чрез `send` процес изпраща съобщение `message` към процес `destination`. Процес, който иска да получи съобщение, трябва да изпълни `receive`, при което съобщението се записва в `message`. Следователно, за да се осъществи предаване на едно съобщение между два процеса, е необходимо единият процес да изпълни `send`, а другият `receive`. При реализацията на механизъм на съобщенията е необходимо да се дадат отговори на следните въпроси, определящи логическите свойства на комуникационната връзка.

1. Как се установява комуникационна връзка между процесите?
2. Може ли една комуникационна връзка да свързва повече от два процеса?
3. Колко комуникационни връзки може да има между два процеса?
4. Еднопосочна или двупосочна е комуникационната връзка?
5. Какво е капацитет на комуникационната връзка?
6. Има ли някакви изисквания за размер и/или структура на съобщенията предавани по определена комуникационна връзка?

В операционните системи има различни реализации на механизъм на съобщения.

Адресиране на съобщенията

Ще разгледаме първите четири от поставените въпроси, отговорът на които зависи от това, как се адресира получателя и изпращача в примитивите `send` и `receive`.

Директна комуникация

Всеки процес именува явно процеса-получател или процеса-изпращач, т.е. `destination` и `source` са идентификатори/имена на процеси. За да се предаде съобщение от процес P към процес Q, трябва да се изпълнят примитивите.

Процес P (изпращач)	**Процес Q (получател)**
`send(Q, message);`	`receive(P, message);`

При този начин на адресация отговорите на първите четири въпроса са следните.

1. Комуникационната връзка се установява автоматично между двойката процеси, но всеки от тях трябва да знае идентификатора на другия.
2. Комуникационната връзка свързва точно два процеса.
3. Между всяка двойка процеси може да има само една комуникационна връзка.
4. Комуникационната връзка е двупосочна.

При този начин съществува симетрия при адресацията, всеки от процесите трябва да укаже идентификатора на другия процес. Може да се реализира асиметричен вариант на директна комуникация. В примитива `receive` се указва идентификатор на процес, както по-горе или може да е от вида:

```
receive(ANY, message);
```

Това означава, че процесът иска да получи съобщение от всеки, който му изпрати такова. Функцията ще върне идентификатора на процеса, от който е полученото съобщение.

Косвена комуникация

Въвежда се нов обект, наричан пощенска кутия (mailbox), опашка на съобщенията (message queue) или порт (port). Съощенията се изпращат в или се получават от пощенска кутия. Следователно, `destination` и `source` са идентификатори на пощенска кутия. За да се предаде съобщение от процес P към процес Q, трябва да се използва обща пощенска кутия, например с име A и да се изпълнят примитивите.

Процес P (изпращач)	**Процес Q (получател)**
`send(A, message);`	`receive(A, message);`

При този метод на адресация отговорите на въпросите са следните:

1. Комуникационната връзка се установява между процесите когато те използват обща пощенска кутия.
2. Комуникационната връзка може да свързва и повече от два процеса.
3. Между два процеса може да съществува повече от една комуникационна връзка.
4. Комуникационната връзка може да е еднопосочна или двупосочна.

В този случай освен примитивите `send` и `receive` трябва да има и примитив за създаване на пощенска кутия. Също така възниква и въпросът за собствеността на пощенската кутия и за правата на процесите да изпращат съобщения в или да получават съобщения от определена пощенска кутия. Друг въпрос е как се унищожава пощенска кутия.

Една възможна реализация е собственик на пощенската кутия да става процесът, който я създаде. Всеки друг процес, който знае името на пощенската кутия и на който собственикът е дал права, може да я използва. Унищожаването на пощенска кутия може да се реализира чрез системен примитив, извикван явно от собственика й. Друга възможност е процесите да могат да ползват обща пощенска кутия чрез механизма за

създаване на процеси и когато последният процес, ползващ определена пощенска кутия, завърши тя да се унищожава автоматично от системата.

Буфериране на съобщенията

Ще разгледаме и последните два от поставените въпроси, отнасящи се до логическите свойства на комуникационната връзка. Какъв е капацитета на комуникационната връзка? Той определя броя на временно съхраняваните в комуникационната връзка изпратени, но още неполучени съобщения.

Съобщения без буфериране

Комуникационната връзка има капацитет нула, т.е. не може да има временно чакащи съобщения. Това означава, че ако първо се изпълни `send`, то изпращачът ще трябва да изчака докато получателят изпълни `receive`. Процесите изпращач и получател синхронизират работата си в момента на предаване на съобщението, затова този метод се нарича още "рандеву".

Съобщения с автоматично буфериране

Комуникационната връзка има ограничен капацитет. Определен брой (обем) съобщения могат временно да се съхраняват в комуникационната връзка, докато получателят изпълни `receive`. Когато се изпълнява `send`, ако комуникационната връзка не е пълна, съобщението се съхранява в нея и изпращачът продължава работата си. Ако комуникационната връзка е пълна, изпращачът ще трябва да чака, докато се освободи място в нея. И в двата случая изпращачът не може да знае дали съобщението му е получено след като `send` завърши. Ако това е важно за комуникацията, то процесите трябва явно да прилагат протокол за потвърждаване. Следният фрагмен илюстрира предаването на едно съощение от процес P към процес Q с потвърждаване.

Процес P (изпращач)	**Процес Q (получател)**
`send(Q, message);`	`receive(P, message);`
`receive(Q, message);`	`send(P, "acknowledgement");`

В MINIX механизм на съобщенията е реализиран с директна адресация, без буфериране и съобщенията са с фиксирана дължина.

В IPC пакета на UNIX System V, който се поддържа и от други Unix и Linux системи, са включени съобщения. Там се използва косвена адресация и автоматично буфериране.

Обект, в който се изпращат и от който се получават съобщения, се нарича опашка на съобщенията (message queue). Всяка опашка на съобщенията има външно име, което е цяло положително число и се нарича ключ. Това позволява една опашка на съобщенията да се използва за комуникация между неродствени процеси. Опашка на съобщенията се създава явно чрез системен примитив `msgget`. Чрез същия системен примитив процес може да получи достъп до създадена вече опашка. Процесът, създал една опашка на съобщенията, става неин собственик и има право да определя правата на другите процеси за достъп до нея. При определяне правата за достъп до опашка на съобщенията се използва същия метод както при файловете, а именно код на защита. Правото `r` означава правото да се получават съобщения от опашката, а правото `w` - да се изпращат съобщения в опашката. Една опашка на съобщенията съществува докато не се унищожи явно чрез системен примитив `msgctl` и само собственикът има право за това. Чрез този системен примитив се извършват и други операции по управление на опашка, например промяна на правата на достъп. Изпращането и получаването на съобщение се извършва с примитивите `msgsnd` и `msgrcv`.

По отношение на структурата на съобщенията има следните изисквания. Всяко съобщение се състои от тип-цяло положително число и текст-масив от байтове с променлива дължина. Структурата на съобщение е следната:

```
struct msgbuf {
      long mtype;
      char mtext[N]; }
```

Чрез типа на съобщението може да се реализират няколко потока за предаване на съобщения в рамките на една опашка на съобщенията, включително и двупосочна комуникация. За това съществена роля има и примитива msgrcv, който позволява да се поиска първото съобщение от определен тип.

Пример. Програмата илюстрира механизма на съобщенията в Unix и Linux, като реализира задачата Производител-Потребител. Производителят изпраща цели числа на Потребителя, който ги умножава по две и извежда резултата на стандартния изход.

```
/* ------------------------------------------------------------ */
/* Producer - Consumer with MESSAGES */

#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>
#define MSG_KEY 123
#define MODE    0600
struct msgb{
  long mtype;
  char mtext[256];
};
int mid;

main(void)
{
struct msgb msg;
int i, buf, status;

  if ( (mid=msgget(MSG_KEY,IPC_CREAT|IPC_EXCL|MODE)) == -1) {
     perror("Msgget error"); exit(1);}

  if (fork()== 0) {                       /* child - producer */
     for (i=0;i<=50 ; i++) {
       msg.mtype = 1;
       *(int*)msg.mtext = i;
       msgsnd(mid, &msg, sizeof(int), 0); /* send message */
     }
  } else {                                /* parent - consumer */
     for (i=0;i<=50 ; i++) {
       msgrcv(mid, &msg, 256,1,0);         /* receive message */
       buf = *((int *)msg.mtext);
       buf *= 2;
       printf("Consumer: %d \n", buf);
     }
     wait(&status);
     if (msgctl(mid, IPC_RMID, 0) == -1 ){
        perror("Msgctl IPC_RMID error"); exit(1); }
     printf("MQ destroyed\n");
  }
  exit(0);
}
/* ------------------------------------------------------------ */
```

Следва примерен изход от изпълнението на програмата.

```
$ a.out
Consumer: 0
Consumer: 2
Consumer: 4
Consumer: 6
Consumer: 8
Consumer: 10
Consumer: 12
 . . .
MQ destroyed
```

В Unix, Linux и MINIX системите се реализира един традиционен механизъм за комуникация между процеси - програмен канал. Програмният канал осигурява еднопосочно предаване на неформатиран поток от данни (поток от байтове) между процеси и синхронизация на работата им. Реализират се два типа програмни канали:

- **неименован програмен канал** (**unnamed pipe** или **pipe**) - за комуникация между родствени процеси
- **именован програмен канал** (**named pipe** или **FIFO файл**) - за комуникация между независими процеси.

Като механизъм за комуникация те са еднакви. Реализират се като тип файл, който се различава от обикновените файлове и има следните особености:

- За четене и писане в него се използват системните примитиви `read` и `write`, но дисциплината е FIFO.
- Каналът има доста ограничен капацитет.

Двата типа програмни канала се различават по начина, по който се създават и унищожават и по начина, по който процес първоначално осъществява достъп към канала. Програмният канал би могъл да се разглежда като механизъм на съобщения с косвена адресация и автоматично буфериране. Разликата е, че в програмния канал няма граници между съобщенията, т.е. процес P може да запише едно съобщение от 1000 байта, а процес Q да го прочете като 10 съобщения от по 100 байта или обратното.

При използване на съобщения и двата аспекта на комуникацията - предаването на данните и синхронизацията се решават от механизма. Затова съобщенията са по-безопасни. Но двата разгледани метода за междупроцесни комуникации:

- чрез обща памет и семафори
- чрез съобщения

не са взаимно изключващи се, т.е. могат да са реализирани и използват в една операционна система. IPC пакета на UNIX System V включва и трите механизма.

4.5. ПЛАНИРАНЕ НА ПРОЦЕСИ

В разгледания модел на многопроцесна операционна система в системата обикновено има много процеси в състояние готов и значително по-малко на брой ЦП. Тази част от ядрото, която избира най-подходящия процес за текущ и решава колко дълго ще работи се нарича ***планировчик*** (***scheduler***). При наличие на много заявки за използване на определен ресурс, вземането на решение коя от тях да бъде удоволетворена се нарича ***планиране***. Планирането е една от важните функции на операционните системи, тъй като обикновено заявките за използване на определен ресурс са повече от възможностите му. Тук ще разглеждаме планирането на ЦП, като го наричаме съкратено планиране.

4.5.1. Нива на планиране

В операционните системи често се извършва планиране на няколко нива.

- **Планиране на високо ниво** (планиране на заданията)

Това ниво присъства в пакетните ОС, където обикновено постъпват повече задания отколкото могат да бъдат изпълнени. При постъпване на ново задание то първо се поставя в опашка, съхранявана на диска. Планировчикът на това ниво решава кое от чакащите задания да бъде заредено в системата, т.е. да се създадат задачи (процеси) за него. Целта на този планировчик е получаването на добра смес от задания и увеличаване на пропусквателната способност на системата.

- **Планиране на ниско ниво**

На това ниво се определя кой от готовите процеси да бъде избран за текущ и колко време да използва ЦП. Това е най-важното ниво, защото този планировчик работи най-често, (много пъти в секундата), винаги го има в ОС и е критичен за производителността на системата. Когато в ОС се използва думата планировчик, обикновено се разбира този.

- **Планиране на междинно ниво** (планиране на свопинга)

В някои ОС при управление на паметта се реализира метод, наречен свопинг. Планировчикът на това ниво управлява изхвърлянето на процеси на диска в свопинг областта и обратното им връщане в паметта.

Планиране на три нива се реализира в някои пакетните ОС. В интерактивните ОС обикновено планирането е на две нива - без високото ниво. Всеки нов процес се приема в системата, като стабилността се контролира от физически ограничения (брой терминали, таблица на процесите и др.) и от човешката природа.

4.5.2. Цели на планирането

За да се разработи добър алгоритъм за планиране е необходимо да се определят целите, които трябва да постига планировчикът. Ще разгледаме някои от основните цели.

- **Справедливост**

Времето на ЦП да се разпределя справедливо между процесите. Планировчикът да не дискриминира едни процеси като ги отлага безкрайно дълго.

- **Баланс**

Да се осигури пълно натоварване на всички ресурси на системата, като се създаде добра смес от процеси. Например, в паметта да има както ограничени от ЦП така и ограничени от В/И процеси.

- **Ефективност**

Голяма част от времето на ЦП да се използва за изпълнение на потребителски процеси, а не за служебни цели, като превключване на контекста или просто ЦП да престоява, защото няма готови процеси.

- **Време за отговор** (response time)

Това е времето за обслужване на една потребителска заявка, т.е. времето от въвеждане на данни или команда от потребителя до получаване на поредния отговор. Тази цел е важна при интерактивни процеси (процеси, изпълнявани в интерактивен или привилегирован режим). За потребителя е важно системата бързо да реагира след вход от клавиатура или мишка на поредната му заявка.

- **Време за чакане** (waiting time)

Това е общото време, през което процесът чака обслужване в състояние готов. От него зависи времето за изпълнение на процеса, т.е. от създаване до получаване на окончателния резултат. Тази цел е важна при пакетни процеси (процеси, изпълнявани във пакетен или фонов режим). Там не е важно как процесът работи във времето, а да се намали общото време, през което той е в системата.

- **Пропусквателна способност**

Това е още една цел важна при пакетните системи. Пропусквателна способност е брой процеси или задания, преминали през системата за единица време. Целта естествено е тя да се увеличи.

- **Предсказуемост**

Процесите да се изпълняват приблизително за едно и също време независимо от натоварването на системата. Алгоритъмът за планиране да гарантира завършването на всеки процес, т.е. да не се случва безкрайно отлагане.

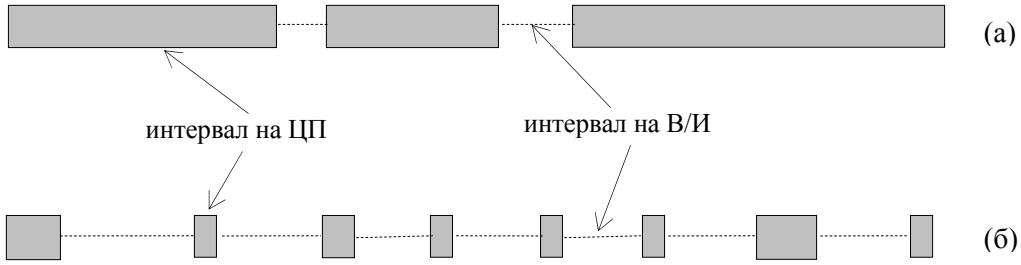
От изброените цели се вижда, че има такива, които са общи, но има и цели, които зависят от типа на процеса. Освен това някои от целите са противоречиви. Това прави планирането една доста сложна задача. Затова има много алгоритми за планиране, реализиращи различни дисциплини.

4.5.3. Дисциплини за планиране

От гледна точка на планирането работата на всеки процес представлява последователност от редуващи се интервали от два типа:

- ***интервал на ЦП*** (CPU burst) - интервал, в които процесът изпълнява команди
- ***интервал на В/И*** (I/O burst) - интервал, в който процесът чака завършването на входно-изходна операция.

Броят и дължината на тези интервали са индивидуални за всеки процес и предварително незвестни на планировчика. Има процеси, които по-голяма част от времето изпълняват някакви изчисления, т.е. те имат малко на брой но дълги интервали на ЦП (*Фиг.4.5а*). Такива процеси се наричат CPU bound или ограничени от възможностите на ЦП. Други процеси често извикват входно-изходна операция и по-голяма част от времето чакат завършването й, т.е. те са с много на брой и къси интервали на ЦП (*Фиг.4.5б*). Такива процеси се наричат I/O bound или ограничени от възможностите на В/И устройства.



Фиг. 4.5. Процес ограничен от ЦП (а); процес ограничен от В/И (б)

Трудността при планиране идва от това, че всеки процес е уникален и непредсказуем за планировчика. Когато планировчикът избере процес за текущ, той не знае колко дълъг е поредния интервал на ЦП, т.е. колко време ще мине преди процесът да се блокира и доброволно да освободи ЦП. Затова важен въпрос при планирането е кога работи планировчикът? Има ситуации, в които той очевидно трябва да се намеси.

- Когато текущият процес завърши като изпълни примитива `exit` и стане еомби.
- Когато текущият процес извика примитив, изискващ блокиране като `read`, `write`, `wait` или друг.

И в двата случая текущият процес доброволно освобождава ЦП защото повече не е в състояние да работи. Ако планировчикът се намесва само в тези два случая, се казва че реализира ***планиране без преразпределение*** (***nonpreemptive scheduling***).

Възможно е планировчикът да се намесва още в един случай. Апаратният таймер изработва прекъсване периодично, например 50 или 60 пъти в секундата. При всяко прекъсване от таймера работи обработчика на прекъсването в ядрото. След като обработката завърши и преди процесът да се върне в потребителска фаза, може да се намеси планировчикът и да избере друг процес за текущ. Планировчикът може да се намесва и след обработката на други апаратни прекъсвания. Например, ако прекъсването е от входно-изходно устройство, което е завършило работа, то ще бъде събуден процесът, извикал съответната операция и той ще мине в състояние готов. Планировчикът може да избере този процес за текущ. Планиране, при което ЦП може да се отнема насилствено от текущия процес, се нарича ***планиране* с *преразпределение*** (***preemptive scheduling***).

Ще разгледаме няколко популярни дисциплини за планиране, които могат да бъдат използвани на различни нива.

Дисциплина FCFS (First-Come-First-Served)

Процесите се обслужват в реда на появяването им. Има една опашка на готовите процеси, в която новите процеси се добавят в края. Същото се прави и за процесите, които са били блокирани, след като бъдат деблокирани. Когато работи планировчикът избира първия процес в опашката и го изключва от нея. Всеки избран процес работи докато се блокира или завърши, т.е. това е дисциплина без преразпределение и без приоритети на процесите.

Основното преимущество е, че това е лесна за разбиране и реализация дисциплина. Освен това е справедлива, в смисъл че предоставя на всички процеси еднакви услуги, т.е. еднакво средно време за чакане. Недостатъците са повече: Не е приложима в интерактивни ОС, защото няма преразпределение. Кратките процеси чакат толкова колкото и дългите, което някой ще каже, че не е справедливо.

Дисциплина SJF (Shortest-Job-First или най-краткото задание първо)

Това е също дисциплина без преразпределение, но за разлика от FCFS е приоритетна дисциплина. За планировчика процесите не са еднакви, а имат

приоритети, които са обратно пропорционални на времето за изпълнение. Точното време за изпълнение е неизвестно предварително, затова се използва очаквано време, задавано от потребителя. Това е един от недостатъците, тъй като потребителят сам определя приоритета на процесите си. Освен това е неприложима за интерактивни процеси, тъй като се иска оценка за необходимото време за изпълнение.

Предимство на дисциплината е, че кратките процеси се обслужват с предимство, което ще намали тяхното време за чакане, а така ще се намали и средното време за чакане в системата. Например, нека четири процеса A, B, C и D, които имат времена за изпълнение 8, 4, 4 и 4 съответно, се появят в този ред в опашката на готовите. Да предположим, че те не се блокират, т.е. след като всеки от тях получи ЦП работи съответното време и завършва. При дисциплина FCFS те ще бъдат обслужени в същия ред, следователно времето за чакане на A е 0, на B - 8, на C - 12 и на D - 16, а средното време за чакане е 9. При дисциплина SJF, процесите ще се подредят в реда B, C, D, A. Времената им за чакане ще са 0, 4, 8, 12 и средното време за чакане ще е 6.

Дисциплина SRT (Shortest-Remaining-Time или най-малко оставащо време)

Идеята на тази дисциплина е винаги да работи процесът, на който му остава най-малко до завършване. Това е приоритетна дисциплина, като приоритетът на процес е обратно пропорционален на оставащото му време за изпълнение, което се изчислява по формулата:

оставащо време = очаквано време - получено време

Прилага се с преразпределение, т.е. когато се появи нов готов процес, ако неговото оставащо време е по-малко от това на текущия в момента, се прави превключване на контекста. При тази дисциплина кратките процеси печелят още повече и се постига минимално средно време за чакане. Но въпреки, че е с преразпределение, и SRT е неприложима за интерактивни процеси, защото се иска оценка на очакваното време.

Общ недостатък на последните две дисциплини е задържането на дългите процеси, което може да доведе до безкрайното им отлагане.

Циклична дисциплина или **дисциплина RR** (Round-Robin)

Една от най-използваните в интерактивните ОС е цикличната дисциплина, наричана още дисциплина RR. Всички готови процеси са подредени в опашка по реда на появяването им (нов или деблокиран процес се добавя в края). Планировчикът избира първия процес от опашката и му определя интервал време, наричан квант (quantum), през който той може да използва ЦП. Ако при изтичане на кванта процесът не е завършил или не се е блокирал, то му се отнема ЦП и се поставя в края на опашката на готовите процеси. Следователно това е дисциплина с преразпределение. Предимствата на дисциплината RR са много:

- Дава предимство на кратките процеси без да дискриминира дългите.
- Осигурява приемливо време за отговор в интерактивните ОС за процесите ограничени от В/И.
- Проста за реализация е.
- Справедлива е.

Основен параметър на тази дисциплина е размерът на кванта. Голям или малък да е кванта, с фиксиран или променлив размер да е за различните процеси. От отговорите на тези въпроси зависи ефективността на RR. Ако дисциплината използва голям квант, то ще нарастне времето за чакане на процес между два последователни кванта, което е недостатък при процесите ограничени от В/И, тъй като влошава времето за отговор. При достатъчно голям квант, дисциплината става FCFS за повечето процеси. Малък квант означава често превключване на контекста, което намалява

ефективността на системата. Например, нека превключването на контекста изисква 5 msec. Ако квантът е 20 msec, то 20% от времето на ЦП ще се използва за служебни цели. Ако квантът е 500 msec, то по-малко от 1% от времето ще е за превключване на контекста (предполагаме, че процесите използват целия си квант). От друга страна, ако предположим, че 10 потребителя почти едновременно натиснат клавиш ENTER, то 10 процеса ще се наредят в опашката на готовите процеси. При квант 500 msec, последният процес ще чака около 5 sec.

Дисциплини с няколко опашки

В дисциплината RR всички процеси са равноправни, но често в ОС това не е вярно. Има процеси, които трябва да се изпълняват с по-висок приоритет, например системни процеси, процеси демони или процесите ограничени от В/И да са по-приоритетни от процесите ограничени от ЦП. Идеята на тази група дисциплини е процесите да се класифицират в няколко класи и за всеки клас да се поддържа опашка на готовите процеси. Всяка опашка има свой приоритет и може да се обслужва с различна дисциплина. Когато ЦП се освободи планировчикът избира процес от непразната опашка с най-висок приоритет и го обслужва според дисциплината на опашката му. Процес от опашка с по-нисък приоритет се избира само, ако опашките с по-висок приоритет са празни.

Важен въпрос е как процесите се разпределят по опашките. Съществуват два основни принципа за това:

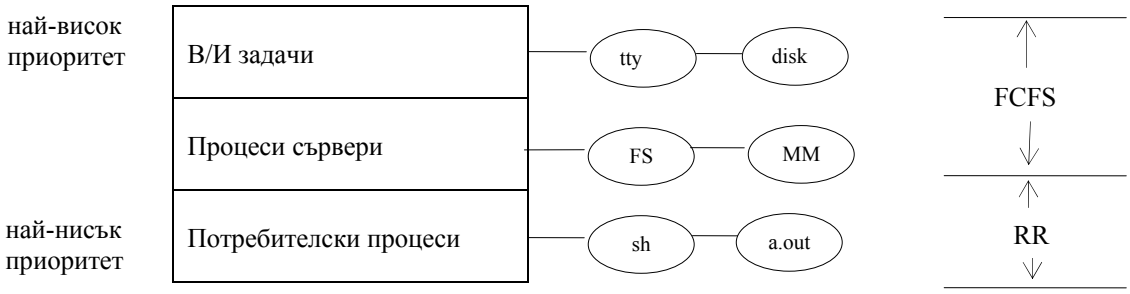
Връзката между процес и опашка да е статична, което означава, че определен процес винаги попада в една и съща опашка когато е готов. Такава дисциплина е използвана в MINIX.

Друга възможност е тази връзка да е динамична, т.е. един процес през своя живот може да попада в различни опашки в зависимост от поведението си. Такава дисциплина се нарича ***опашки с обратна връзка***, защото дисциплината се адаптира към поведението на процеса като повишава или намалява приоритета му. Пример за система, реализираща такава дисциплина е Unix.

Планиране на процесите в MINIX

Алгоритъмът за планиране на процесите в MINIX използва три опашки. Всяка опашка има свързан с нея приоритет. Връзката между процесите и опашките е статична и е показана на *Фиг.4.6*.

.



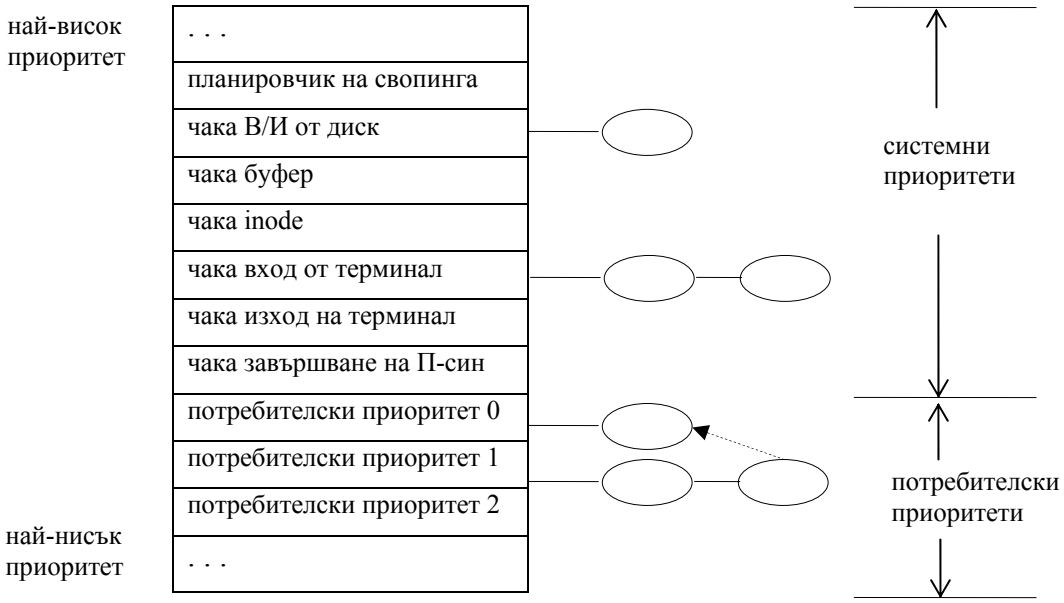
Фиг. 4.6. Планиране на процесите в MINIX

В опашката с най-висок приоритет попадат един специален вид системни процеси, наречени входно-изходни задачи. Във всяка В/И задача работи част от операционната система, която реализира драйвер на някой тип В/И устройство. Затова тази опашка се обслужва по дисциплината FCFS. В средната опашка се нареждат така

наречените процеси сървери. Те са два: FS (File Server) и MM (Memory Manager). Те реализират системните примитиви на операционната система, съответно FS - за работа с файлове и MM - за управление на процеси. Затова и тази опашка се обслужва по дисциплината FCFS. В опашката с най-нисък приоритет са потребителските процеси, които се обслужват с дисциплина RR с квант 100 msec. В/И задача или процес-сървер никога не се свалят от ЦП, независимо колко дълго са работили.

Планиране на процесите в UNIX

Алгоритъмът за планиране на процесите в Unix използва няколко опашки с обратна връзка. Всяка опашка има свързан с нея приоритет. Приоритетите са цели числа, като по-малко значение означава по-висок приоритет. Диапазонът на приоритетите се дели на два непресичащи се класа: потребителски приоритети и системни приоритети, които са по-високи от първите. *Фиг.4.7*. илюстрира използваните опашки за планиране в UNIX.



Фиг. 4.7. Планиране на процесите в UNIX

Системен приоритет се дава на процес, изпълняван в системна фаза, който се е блокирал. Значението на системния приоритет зависи от събитието, което процесът чака. Когато такъв процес се деблокира и мине в състояние готов, ще бъде добавен в опашката на съответния системен приоритет.

Потребителски приоритет има процес, изпълняван в потребителска фаза и такъв, който е бил свален от планировчика в преразпределен.

Кога и как планировчикът преизчислява приоритетите на процесите?

- Когато процес минава в състояние блокиран, му се дава съответен системен приоритет.
- Когато процес се връща от системна в потребителска фаза му се дава потребителски приоритет, защото може да е бил блокиран и приоритетът му е системен.
- При обработка на прекъсване от апаратния таймер на всяка секунда се преизчисляват всички потребителски приоритети по формулата:

  `priority = CPU/2 + base + nice`

В тази формула `CPU` и `nice` са полета в таблицата на процесите. Полето `CPU` отчита използването на ЦП от процеса напоследък. При всяко прекъсване от таймера неговото значение се увеличава с 1 за текущия процес. Така приоритетът на текущият процес се понижава, но наказанието за използването на ЦП не е вечно. Периодично полето `CPU` се намалява (дели се на 2 в горната формула или по друг начин). Компонентата `base` е прагов приоритет между системни и потребителски приоритети, който не позволява процес в потребителска фаза да получи системен приоритет. Полето `nice` се зарежда чрез едноименен системен примитив, изпълнен от процеса. Чрез него процес може да понижи или повиши приоритета си, но само процес на администратора може да зададе значение, с което да повиши приоритета.

Основната идея на този алгоритъм за планиране е процесите бързо да преминават през системна фаза.

4.6. НИШКИ

Нишките (***threads***) са сравнително нова абстракция в операционните системи. Понякога ги наричат ***lightweight processes*** или ***подпроцеси*** и макар, че това наименование представлява известно опростяване, то е добра начална точка. Нишките не са процеси, но те имат нещо общо с процесите, представляват част от процес. Да си припомним какво е процес, така както го разглеждахме до сега. Той включва програмен код, данни, различни други ресурси, като файлови дескриптори, текущ каталог, управляващ терминал и др. и досега един набор от машинни регистри, т.е. един регистър PC. Това означава, че всеки процес има една последователност на изпълнение на команди или накратко е последователен процес.

В същност разгледаният модел на процесите се базира на две независими концепции:

- групиране на ресурсите - процесът има различни ресурси;
- изпълнение на програма - процесът е последователност на изпълнение на команди (thread of execution или накратко само thread).

Тези два аспекта на процеса могат да бъдат разделени и така се появява понятието нишка. Процесът се използва за групиране на ресурсите, а нишките са обектите, изпълнявани от ЦП. Всяка последователност на изпълнение в рамките на процес се нарича нишка, т.е. нишката е част от процес, която има собствен набор от регистри и стек.

Идеята на въвеждането на понятието нишка е процесът да може да има няколко последователности на управление, т.е. да е конкурентен, а не последователен процес. Такъв процес, който включва няколко нишки се нарича ***multithreaded process***. Различните нишки в един процес не са така независими, както различните процеси. Следва списък на елементите на нишка и тези общи за всички нишки в един процес.

Елементи на нишка	**Елементи на процес**
PC и други регистри стек състояние	адресно пространство глобални променливи отворени файлове текущ каталог управляващ терминал

Всяка нишка има свой собствен стек, който съдържа по един слой за всяка извикана в нишката функция, която още не е завършила. Една нишка може да се намира в едно от няколко състояния: текуща, готова, блокирана, завършила (зомби). Преходите от едно състояние в друго са както в разгледания модел на процесите.

Поддържането на нишки дава възможност да се програмират конкурентни приложения, които могат да работят по-ефективно като в еднопроцесорна така и в многопроцесорна среда. Една причина за появата на нишки са приложения, които изпълняват няколко дейности. Проектът на програмата ще е по-ясен ако приложението се декомпозира на няколко последователни нишки, които работят в едно адресно пространство. Като пример за такова приложение, където нишките са полезни, да разгледаме текстообработваща програма (Word processor). Обикновено тя визуализира документа на екрана форматиран така, както ще изглежда на печат. Освен това периодично и автоматично тя записва редактирания файл на диска. Нека проектираме нашия Word с три нишки. Една нишка ще взаимодейства с потребителя (интерактивна нишка) и ще прима неговите команди за изтриване, добавяне на текст и т.н. Друга нишка ще преформатира документа при необходимост, т.е. по команда от интерактивната нишка. Третата нишка периодически ще записва документа на диска.

Такъв модел на програма Word е по-ясен и по-прост за реализация. Освен това в този случай модел с три процеса, комуникиращи с някакъв механизъм, не е подходящ.

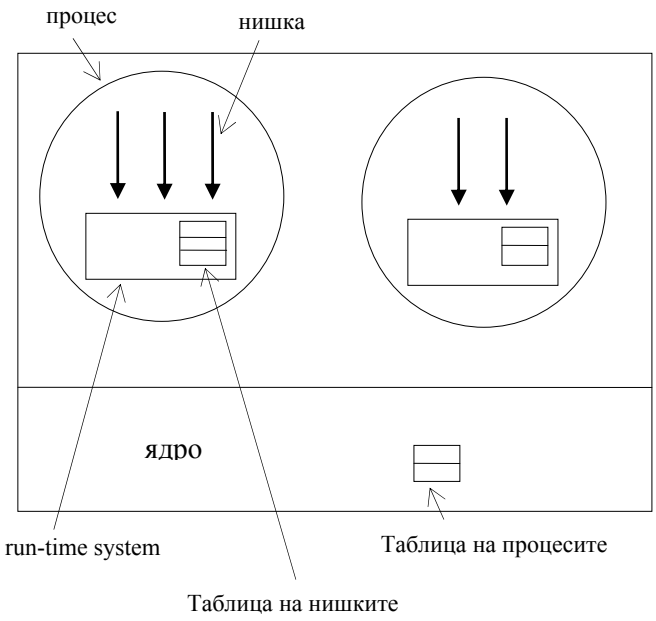
Друг аргумент в полза на нишките е производителността. Операциите с нишки - създаване и унищожаване, ще са по-бързи отколкото при процеси, защото нишката няма собствени ресурси. Също така по-ефективно може да е изпълнението на един многонишков процес. Когато някоя от нишките на процеса се блокира, чакайки например входно-изходна операция, друга нишка на процеса може да продължи изпълнението. Така ще се увеличи общата скорост на работа на процеса.

4.6.1. Реализация на нишки

Пакетът, реализиращ абстракцията нишка, трябва да осигури набор от операции, подобни на тези при процеси: създаване на нова нишка, завършване на нишка, изчакване на завършването на друга нишка (аналози на `fork`, `exit`, `wait`) и др. Има два основни начина за реализация на нишки:

- в потребителското пространство
- в ядрото.

Първият начин означава, че ядрото не знае нищо за нишки и управлява последователни процеси. Общата схема при тази реализация е показана на *Фиг.4.8*.



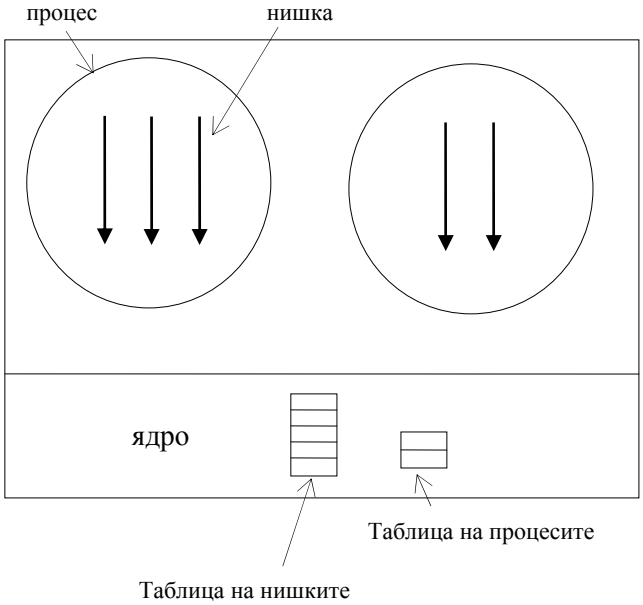
Фиг. 4.8. Нишки в потребителското пространство

Целият пакет функции, реализиращи нишките (run-time system), работи в потребителското пространство. Всеки процес има собствена Таблица на нишките, която съдържа информация за нишките на процеса, като състояние, съхранени регистри и др. Тази таблица също се намира в потребителското пространство и се управлява от run-time системата. Например, когато една нишка трябва да изчака завършването на друга нишка на процеса, тя извиква съответната функция от run-time системата. Там се променя състоянието на нишката в блокирана, съхраняват се регистрите на нишката, избира се друга нишка на процеса, зареждат се съхранените регистри на новата нишка и така се прави превключване на нишки (thread switching). Има едно съществено различие при управлението на нишки и процеси. Превключването на нишки не изисква прекъсване и вход в ядрото. Всички функции, реализиращи описаните действия, са в потребителското пространство. Това е едно от предимствата, което този начин за

реализация на нишки, дава, а именно по-бързо превключване между нишки в един процес. Друго предимство е, че този метод може да се използва в съществуваща операционна система без изменение в ядрото, а с добавяне на нова библиотека функции.

Но съществуват и проблеми. Един от тях е свързан със системните примитиви, които блокират процеса, като `read`, `write` и др. Ако при изпълнението на такъв примитив се наложи, се блокира целият процес, а не само нишката, извикала примитива. Такова поведение противоречи на основната идея за използване на нишки в приложения, а именно когато една нишка в процеса се блокира друга негова нишка да продължи работа. Друг проблем се проявява при планиране на нишки. Когато една нишка започне работа тя ще работи докато доброволно не освободи ЦП. Това е така, защото планировчикът на нишките не може да работи докато управлението не се предаде в run-time системата. Следователно, планирането на нишките в един процес е без преразпределение.

Вторият начин за реализация на нишки е в пространството на ядрото, т.е. ядрото знае за съществуването им (*Фиг.4.9*). В този случай в потребителското пространство няма run-time система и Таблица на нишките. В ядрото има една глобална Таблица на нишките, съхраняваща информация за всички нишки в системата. Освен това, както и преди, в ядрото има Таблица на процесите, описваща процесите. Всички операции с нишки са реализирани като системни примитиви, т.е. в ядрото. Това означава, че когато нишка трябва да се блокира, ядрото може да избере друга нишка от същия или от друг процес. Следователно, системните примитиви като `read` не създават проблеми. Недостатък на този метод е, че операциите с нишки стават по-бавни (включват прекъсване, съхранение на контекста и възстановяването му).



Фиг. 4.9. Нишки в пространство на ядрото

По-нататък ще разгледаме по-подробно нишките по стандарта POSIX 1003.1c, известни още като pthreads, които могат да се използват в съвременните версии на Unix и Linux системите. В някои версии, като UNIX System V, Solaris, Linux и др., нишките се реализират в ядрото. В други Unix системи, като BSD нишките не се поддържат от ядрото и реализацията им е по първия начин в една от библиотеките. Функциите за работа с нишки, които ще разгледаме, осигуряват основни операции с нишки, като създаване, завършване и др.

4.6.2. Основни операции с нишки

Първата - главна нишка във всеки процес се създава автоматично при създаване на процес. Друга нишка може да се създаде, когато една нишка изпълни функцията:

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Всяка нишка има идентификатор, който й се присвоява при създаването и е от тип `pthread_t`. При успех се връща идентификатора на новата нишка чрез аргумента *`thread`*.

Аргументът *`attr`* определя някои характеристики на създаваната нишка или както ги наричат атрибути. Ако е указано `NULL`, то атрибутите имат значения по премълчаване. Един от атрибутите определя типа на нишката: joinable или detached. Тип joinable означава, че друга нишка в процеса може да се синхронизира с момента на завършването й, т.е. след завършване на нишката тя не изчезва веднага (подобно на състояние зомби при процеси). Тип detached означава, че при завършване на нишката веднага се освобождават всички ресурси, заемани от нея, т.е. тя изчезва в момента на завършване. По премълчаване нишката се създава joinable. Има и други атрибути, които определят дисциплината и параметри на планиране на нишки.

Когато нишката бъде създадена тя започва да изпълнява функцията *`start_routine`*, на която се предава аргумент *`arg`*. За разлика от процесите, където бащата и синът продължават изпълнението си след `fork` от една и съща точка, тук не е така, защото нишките на един процес имат общи ресурси. При успех функцията връща 0, а при грешка връща различен от 0 код на грешка.

Така създадената нишка завършва когато:

- Изпълни return от *`start_routine`*;
- Извика явно `pthread_exit`;
- Друга нишка я прекрати чрез `pthread_cancel`.

```
void pthread_exit(void *retval);
```

Аргументът *`retval`* е код на завършване, който нишката изработва. Той е предназначен за всяка друга нишка на процеса, която изпълни `pthread_join`. Но ако нишката е от тип detached, то след `pthread_exit` от нея не остава никаква следа, следователно и кода не се съхранява. Няма връщане от тази функция.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

Чрез тази функция една нишка може да синхронизира изпълнението си със завършването на друга нишка (аналог на `wait` при процеси, но тук няма изискването текущата нишка да е баща на чаканата). Аргументът *`thread`* е идентификатор на нишката, чието завършване се чака от текущата нишка. Текущата нишка се блокира докато не завърши нишка *`thread`*. Най-много една нишка може да изчака завършването на коя да е друга нишка, т.е. ако няколко нишки изпълнят `pthread_join` за една и съща нишка, вторият `pthread_join` ще върне грешка. При успех функцията връща 0 и чрез аргумента *`value_ptr`* се предава кода на завършване на нишката *`thread`*. При грешка връща код на грешка различен от 0.

Пример. Програмата "Hello World" чрез нишки.

```
/* ------------------------------------------------------------ */
#include <pthread.h>
main(void)
{
int ret;
pthread_t th1, th2;
void *print_mess();
char *mess1 = "Hello ";
char *mess2 = "World\n";

/* Create a thread to print 'Hello' */
   ret = pthread_create(&th1, NULL, print_mess, mess1);
   if (ret != 0) {
      perror("Error in pthread_create 1");
      exit(1); }
/* Create a thread to print 'World' */
   ret = pthread_create(&th2, NULL, print_mess, mess2);
   if (ret != 0) {
      perror("Error in pthread_create 2");
      exit(1); }
/* Wait the threads to finish */
   pthread_join(th1, NULL);
   pthread_join(th2, NULL);
   exit(0);
}

/* Thread function */
void* print_mess(void *str)
{
char *msg;
   msg = (char *)str;
   printf("%s", msg);
   pthread_exit((void *)0);
}
/* ------------------------------------------------------------ */
```

Програмата извежда "Hello World" или "World Hello", защото всяка от двете думи се извежда от отделна конкурентна нишка.

Пета глава

ДЕДЛОК

Казваме, че множество от два или повече процеса са в дедлок (deadlock), ако всички те са в състояние блокиран и всеки чака настъпването на събитие, което може да бъде предизвикано само от друг процес в множеството. Тъй като всички процеси са блокирани, никой от тях не може да работи и да предизвика събитие, което да събуди някой друг процес от множеството. Следователно, ако системата не се намеси, всички процеси ще останат вечно блокирани. Събитието, което процесите чакат най-често е предоставяне на ресурс на системата.

Кои са участниците в проблема дедлок? Системата включва различни типове ресурси и състезаващи се за тях процеси. Всеки тип ресурс може да има няколко идентични екземпляра. Ресурси могат да са:

- апаратни устройства – печатащо устройство, лентово устройство и др.
- данни – запис в системна структура, файл или част от файл и др.

Ресурсите са обектите, които процесите искат да използват монополно. Последователността от действия при използване на ресурс от процес е:

1. заявка за ресурс (request)
2. използване на ресурс (use)
3. освобождаване на ресурса (release)

Ако ресурсът не е свободен при стъпка 1, то процесът бива блокиран и се събужда, когато друг процес изпълни стъпка 3. Начинът по който се изпълняват тези стъпки зависи от типа на ресурса, например може да е системен примитив или част от системен примитив.

5.1. НЕОБХОДИМИ УСЛОВИЯ ЗА ДЕДЛОК

Дедлок може да възникне ако в системата са изпълнени следните условия, формулирани от Кофман (E.Coffman) като необходими условия за дедлок.

1. Взаимно изключване (Mutual exclusion)

   В системата има поне един ресурс, който трябва да се използва монополно, т.е. или е свободен или е предоставен на точно един процес.

2. Очакване на допълнителен ресурс (Hold and Wait)

   Процесите могат да получават ресурси на части, като съществува поне един процес, който задържа получени ресурси и чака предоставяне на допълнителен ресурс.

3. Непреразпределение (No preemption)

   Ресурс, предоставен на процес, не може насилствено да му бъде отнет. Процесите доброволно освобождават ресурсите, след като приключат работата си с тях.

4. Кръгово чакане (Circular wait)

   Трябва да съществува множество от блокирани процеси $\{ p_1, p_2, \ldots, p_k\}$, такива че p_1 чака ресурс държан от p_2, p_2 чака ресурс държан от p_3 и т.н. p_k чака ресурс държан от p_1.

Последното условие предполага изпълнението на другите три, така че условията не са напълно независими, но е полезно да се разглеждат отделно.

5.2. ГРАФ НА РАЗПРЕДЕЛЕНИЕ НА РЕСУРСИТЕ

Холт (R.Holt) е предложил математически модел на дедлока. Състояние дедлок в системата може формално да бъде описано чрез двуделен ориентиран граф G={V, E}. Множеството на върховете V е обединение на две непресичащи се подмножества:

$\{ p_1, p_2, \ldots, p_n\}$ – множество на процесите в системата

$\{ r_1, r_2, \ldots, r_m\}$ – множество на типовете ресурси в системата.

Множеството на ребрата E включва два вида ребра:

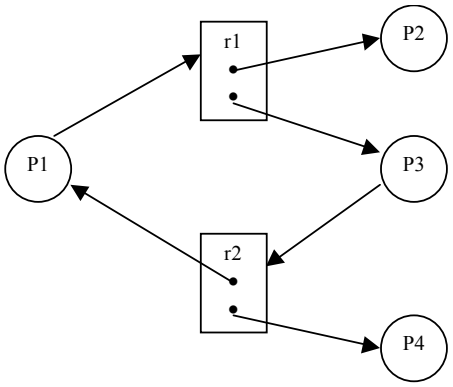
(p_i, r_j) – означава, че процес p_i иска ресурс r_j (request edge)

(r_j, p_i) – означава, че един екземпляр от ресурс r_j е предоставен на процес p_i (assignment edge).

Когато процес p_i поиска ресурс r_j, в графа се добавя ребро (p_i, r_j). Когато тази заявка бъде изпълнена, това ребро се трансформира в ребро (r_j, p_i). Когато процесът освободи ресурса, реброто (r_j, p_i) се изключва от графа. Така графът представя разпределението на ресурсите и чакащите заявки във всеки един момент.

Може ли по графа на разпределение на ресурсите да се открие наличието на дедлок? Ако в графа няма цикъл, то в системата няма дедлок. Ако графът съдържа цикъл, то може да има дедлок. Ако всеки тип ресурс, участващ в цикъла има един екземпляр, тогава има дедлок, в който участват процесите, включени в цикъла. Наличието на цикъл в графа е необходимо, а в горния случай и достатъчно условие за дедлок. В същност наличието на цикъл в графа е равносилно на четвъртото от условията на Кофман.

Следващият пример, показан на *Фиг.5.1*, илюстрира горното твърдение, т.е. в графа има цикъл, включващ процесите p_1 и p_3 и ресурсите r_1 и r_2, но няма дедлок.



Фиг. 5.1. Граф на разпределение на ресурсите

5.3. ПРЕДОТВРАТЯВАНЕ НА ДЕДЛОК

Методите за предотвратяване (Deadlock Prevention) налагат ограничения на процесите при получаване на ресурси, така че дедлок става принципно невъзможен. Четирите необходими условия са ключ към някои възможни решения. Ако при разпределението на ресурсите в системата можем да осигурим неизпълнението на поне едно от тези условия, тогава дедлок няма да настъпи. Най-известните методи за предотвратяване са стратегиите на Хавендер (J.Havender).

1. Взаимно изключване

Ако никой ресурс никога не се предоставя за монополно използване, то дедлок няма да настъпи. За съжаление обаче има ресурси, които не могат да не се използват монополно, например запис в таблицата на процесите, печатащо устройство, лентово устройство. Затова това условие не може да бъде нарушено за всички ресурси.

2. Очакване на допълнителен ресурс

За да се наруши това условие трябва да се гарантира, че когато един процес иска ресурс той не задържа други ресурси.

Един възможен протокол на разпределение е следният. Всеки процес трябва да иска всичките необходими му ресурси наведнаж и преди започване на работа, и те да му бъдат предоставени преди началото на изпълнение.

Друг алтернативен протокол позволява на процес да иска ресурси и след започване на изпълнението, само когато не задържа никакви други ресурси. Процес може да иска ресурси, да ги използва, но преди да поиска допълнителни ресурси трябва да е освободил всички дадени му преди това.

Макар, че има разлика между двете стратегии, общият им недостатък е неефективното използване на ресурсите, тъй като процесите ще трябва да ги задържат по-дълго време отколкото са им необходими.

3. Непреразпределение

Ако процес, който е получил и държи някакви ресурси, поиска допълнителни и системата не може да му ги предостави веднага, то процесът бива блокиран и всички дадени му до момента ресурси му се отнемат. Те се добавят към списъка на ресурсите, които процесът е поискал допълнително и които чака. Процесът ще бъде събуден и ще продължи изпълнението си, когато системата може да му даде всичките ресурси – новите (поисканите) и старите (отнетите му). Проблем при този метод е, че той може да се прилага, само по отношение на ресурси, чието състояние лесно може да се съхрани при насилствено отнемане от процес и след това да се възстанови (например, регистрите на ЦП, оперативна памет). Но как да се приложи към печатащо устройство или лентово устройство.

4. Кръгово чакане

Въвежда се наредба на всички типове ресурси, като с всеки тип ресурс се свързва цяло число, което е поредния му номер. Нека $R = \{ r_1, r_2, . . . , r_m \}$ е множеството на типовете ресурси. Следователно, определяме функция $F: R \rightarrow N$.

Един възможен протокол е следния. Всеки процес може да иска ресурси само по нарастване на номера на типа. Ако първоначално е получил ресурси от тип r_i, то след това може да иска само ресурси от тип r_j, където $F(r_j) > F(r_i)$. Ако от определен тип са му необходими няколко екземпляра, трябва да ги поиска наведнаж.

Друг възможен протокол е следния. Когато процес иска ресурс от тип r_j, той трябва да е освободил всички ресурси от тип r_i , за които $F(r_i) \geq F(r_j)$.

5.4. ЗАОБИКАЛЯНЕ НА ДЕДЛОК

Дедлок може да се избегне и без да се поставят такива строги правила на процесите, а чрез внимателно анализиране на всяка заявка с цел да се прецени дали удоволетворяването ѝ е безопасно. Когато опасността от бъдещ дедлок се увеличи, ресурсът не се дава на процеса, за да се избегне дедлока. За тази цел системата трябва да разполага с информация, за това как процесите ще искат ресурси по време на своето изпълнение.

Най-популярният метод за заобикаляне на дедлок (Deadlock Avoidance) е известен като алгоритъм на банкера и е предложен от Дейкстра за 1 тип ресурс и от Хаберман (Habermann) за m типа ресурса. Всеки процес трябва да даде предварителна информация за максималния брой екземпляри от всеки тип ресурс, които може да поиска. Състоянието на разпределение на ресурсите се описва чрез:

- Брой свободни ресурси
- Максимален брой ресурси за всеки процес

- Брой ресурси дадени на всеки процес
- Брой ресурси, които процесът може още да поиска

Казваме, че системата се намира в ***надеждно състояние*** (***safe state***) на разпределение на ресурсите, ако съществува поне една последователна наредба на процесите $<p_1, p_2, \ldots, p_n>$, за която е изпълнено следното: нуждите на всеки процес p_i могат да се удоволетворят от свободните в момента ресурси и ресурсите държани от процесите p_j, където $j<i$. Следователно, от текущото разпределение на ресурсите съществува някаква последователност от други състояния, в която системата може да удоволетвори максималните потребности на всеки процес и той след време да завърши. Когато за текущото разпределение на ресурсите не съществува нито една такава последователност се казва, че състоянието е ненадеждно.

Нека системата разполага с 12 екземпляра от един тип ресурс и текущото разпределение е следното:

Процеси	**Максимален брой**	**Получен брой**	**Оставащ брой**
p1	10	5	5
p2	4	2	2
p3	9	2	7

Следователно в момента системата разполага с 3 свободни екземпляра. Това е пример за надеждно състояние, защото съществува последователността $<p_2, p_1, p_3>$, в която се гарантира завършването и на трите процеса.

Ако предположим, че процес p_3 поиска една допълнителна бройка и системата му я даде, то новото състояние ще е ненадеждно, тъй като се гарантира завършването само на процес p_2.

Процеси	**Максимален брой**	**Получен брой**	**Оставащ брой**
p1	10	5	5
p2	4	2	2
p3	9	3	6

Алгоритъмът на банкера анализира всяка заявка за ресурс и я удоволетворява само ако новото състояние е надеждно. В противен случай процесът трябва да чака, т.е. блокира се и ще бъде събуден когато системата е в състояние да удоволетвори заявката му, като остане в надеждно състояние. Следва описание на Алгоритъма на банкера.

Структурите данни, представящи разпределението на ресурсите при n процеса и m типа ресурса, са следните масиви:

- Available[m] - брой свободни ресурси за всеки тип
- Max[n,m] - максимален брой ресурси за всеки процес. Max[i,j] е броят ресурси от тип r_j, който процес p_i може да поиска.
- Allocation[n,m] - разпределените в момента ресурси. Allocation[i,j] е броят ресурси от тип r_j, който процес p_i е получил и държи в момента.
- Need[n,m] - оставащите ресурси за всеки процес, т.е. Need[i,j] = Max[i,j] - Allocation[i,j]

При описанието на алгоритъма ще използваме следните обозначения. Нека X[n] иY[n] са едномерни масиви. Ще казваме, че X<=Y, когато X[i] <=Y[i] за i=1,2,…n. Ще казваме, че X<Y, когато X<=Y и X!=Y. Всеки ред на матрицата Allocation ще

разглеждаме като вектор, който описва разпределените на процеса p_i ресурси и ще означаваме $Allocation_i$. Аналогично, с $Need_i$ ще означаваме нуждите на процеса p_i.

Алгоритъм на Банкера

Нека масивът Request[m] е една заявка на процес p_i за ресурси от различни типове.

1. if $Need_i$ < Request then { "Грешка"; return; }
2. if Availabe < Request then { "ресурсите не са свободни"; блокира p_i; return; }
3. Available = Available - Request;
   $Allocation_i = Allocation_i + Request;$
   $Need_i = Need_i - Request$;
4. if safe_state() then return;
   else { възстановява се старото състояние; блокира p_i; return; }

Алгоритъм на safe_state

Нека Work[m] и Finish[n] са работни масиви, а flag е променлива.

```
1. Work = Available; flag = true; for (i = 1; i <= n; i++) Finish[i] = false;
2. while (flag == true) {
        flag = false;
        for (i = 1; i <= n; i++) {
                if ( Finish[i] == false and Need_i <= Work ) {
                        Work = Work + Allocation_i;
                        Finish[i] = true;
                        flag = true; }
        }
}
3. for (i = 1; i <= n; i++)
        if  ( Finish[i] == true ) then continue; else return false;
   return true;
```

Макар, че алгоритъмът на банкера е доста популярен (има го във всеки учебник по операционни системи), използването му в практиката е трудно. Броят на процесите не е фиксиран, а се изменя динамично, тъй като потребителите непрекъснато създават нови процеси. Възможна е и динамична промяна на броя на ресурсите от определен тип, например поради повреда на някое печатащо устройство. И накрая, сложността на алгоритъма при n процеса и m типа ресурса е от порядъка на $m*n^2$. Това означава допълнително време, тъй като алгоритъмът трябва да се изпълнява при всяка заявка на процес за ресурс.

Приложение 1

ФУНКЦИИ НА СИСТЕМНИТЕ ПРИМИТИВИ

Файлова система

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>

int creat(const char *filename, mode_t mode);
int open(const char *filename, int oflag [, mode_t mode]);
int close(int fd);
ssize_t read(int fd, void *buffer, size_t nbytes);
ssize_t write(int fd, void * buffer, size_t nbytes);
off_t lseek(int fd, off_t offset, int flag);

#include <sys/stat.h>

int stat(const char *filename, struct stat *sbuf);
int fstat(int fd, struct stat *sbuf);
```

Процеси

```
#include <sys/types.h>
pid_t fork(void);
void exit(int status);

#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

int execl(const char *name, const char *arg0
          [,const char *arg1]..., 0);
int execlp(const char *name, const char *arg0
          [,const char *arg1]..., 0);
int execle(const char *name, const char *arg0
          [,const char *arg1]..., 0, char *envp[]);
int execv(const char *name, char *argv[]);
int execvp(const char *name, char *argv[]);
int execve(const char *name, char *argv[], char *envp[]);
pid_t getpid(void);
pid_t getppid(void);
```

Нишки

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **value_ptr);
```

ЛИТЕРАТУРА

1. Tanenbaum A.S. Operating Systems: Design and Implementation, NJ: Prentice Hall, 1987; 2nd ed. 1997.
2. Tanenbaum A.S. Modern Operating Systems, NJ: Prentice Hall, 1992; 2nd ed. 2001.
3. Peterson J. L., Silberschatz A. Operating Systems Conceps, Addison-Wasley Publishing Company Inc., 1985.
4. Bach M. J. The Design of the UNIX Operating System, Prentice Hall Inc., Englewood Cliffs, N.J., 1986.
5. Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, O'Reilly, 2000; 3rd ed. 2005.
6. Николов Л. Операционни системи._София: Сиела, 1998.