ФАЙЛОВИ СИСТЕМИ

Файлова система или управление на данните 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 друга медиа) 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? Това са въпроси, които вълнуват потребителите и се разглеждат в първия раздел. Вторият раздел е посветен на проблеми при физическата реализация на файлова система и подходи за тяхното решаване. Какви са операциите, наричани ***системни примитиви, системни функции, системни извиквания*** или ***system calls***, осигурявани от файловата система и как функционират? Това е предмет на разглеждане в третия раздел. Като примери са използвани файловите системи на операционните системи UNIX, MINIX, LINUX, MSDOS, OS/2 и Windows NT.

1. ЛОГИЧЕСКА СТРУКТУРА НА ФАЙЛОВА СИСТЕМА

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.p` - програма на Паскал
`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 и LINUX, такива разширения представляват съглашения, които се използват от потребителите и някои обслужващи програми (транслатори, 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***, съдържащи програми, данни, текстове или каквото друго потребителят пожелае. Но пространството от имена би могло да включва и имена на външни устройства и услуги. Такова решение е прието в много от съвременните операционни системи. Външните устройства са специален тип файлове, наречени ***специални файлове (character special u block special)*** 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 структури данни, наричани ***справочник***, ***каталог***, ***директория***, ***directory***, ***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*** и ***FIFO файл*** се използва като механизъм за комуникация между конкурентни процеси и ще бъде подробно разгледан в раздела за 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 структурата 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. Използван е 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 (наричат ги също така метеданни). Атрибутите, реализирани в различните системи се различават, но един списък от възможни атрибути е показан на Фиг. 1.

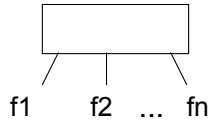
Размер	Текущия размер на файла в байтове, блокове или записи.
Време на създаване	Дата и време на създаване на файла
Време на достъп	Дата и време на последен достъп до файла
Време на изменение	Дата и време на последно изменение на файла
Собственик	Потребителят, който е текущия собственик на файла.
Права 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.

Фиг. 1. Някои възможни файлови атрибути

1.2. КАТАЛОЗИ И ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА СИСТЕМА

Вторият основен проблем, 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.

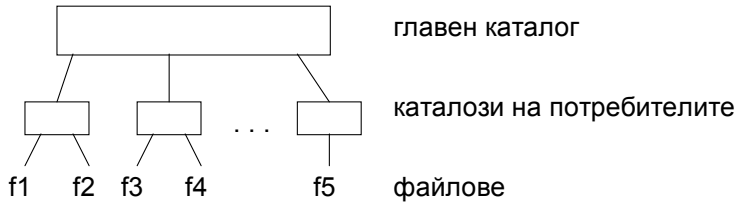
Като правило файловата система е разположена върху няколко носителя и включва файлове, принадлежащи на различни потребители. Най-простото решение, което ние ще наречем еднокаталогова файлова система, е следното. На всеки носител има по един каталог, който съдържа информация за всички файлове върху носителя (Фиг.2).



Фиг. 2. 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.

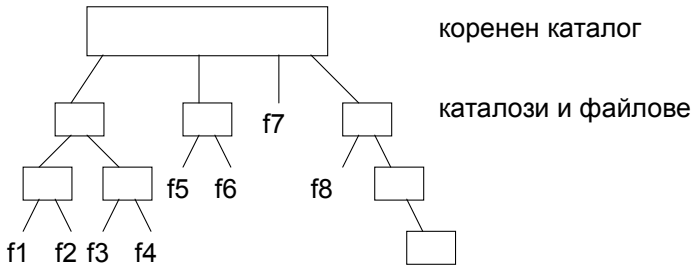
В съвременните, главно многопотребителски операционни системи, горният подход не е приемлив, най-малкото поради опасността от конфликт между еднакви имена на файлове, давани от различни потребители. В една от първите интерактивни системи, CTSS (Compatible Time-Sharing System), разработена в MTI, е реализирана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



Фиг. 3. Йехархична структура с три 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. 4).



Фиг. 4. 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 файла, разделени със специален символ, като напр. ".", ":", "\", "/". Движението нагоре по дървото се записва чрез специално име, например ".." в UNIX, LINUX, MINIX и MSDOS.

Сравнение между файловите системи на UNIX, LINUX и MSDOS

	UNIX, 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

2. 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?

Основните 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.

2.1. 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 файла се разпределя една (или малко на брой) непрекъсната дискова област с нужния размер или за файла 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 използвана в операционната система OS/360 за машини от семейството на IBM/360. Например, ако при изпълнение на програма, създаваща файл, потребителят укаже в JCL оператора DD:

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

то първоначално за файла се отделя една непрекъсната област от диска с размер 20 писти, 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 файла се ограничава от един първичен и 15 вторични екстента с размери избрани от потребителя. След това проблемът 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.

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 адреси (номера) 0,1,2, ...N.

2.2. 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 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 разпределение 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 1, то съответният блок е свободен, а ако е 0 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 за тази цел.

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 диска, 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 по-подробно 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. Индексът съдържа адресите на дисковите блокове, разпределени за файла, 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*)** и при големи файлове е корен на дърво, включващо и косвени блокове. Подробно структурата на индексния описател ще разгледаме в раздела за физическа организация на файловата система в UNIX.

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. По-нататък по-подробно ще разгледаме физическото представяне на файловата система HPFS.

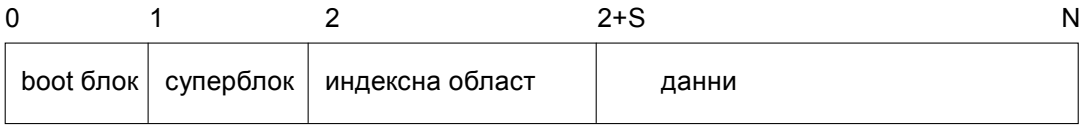
Друг тип информация, която трябва се съхранява, са общи параметри на физическата организация на файловата система. Обикновено системната структура се нарича ***суперблок***. В суперблока се съхраняват общи параметри, като:

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

2.3. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА 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 на ядрото (драйверите). Специфичните особености на дисковете, като брой цилиндри, писти, сектори, начин на разполагане на блоковете върху повърхността на диска и други са скрити. Единственото, по-което се различават дисковете е по максималния номер на блок.

При създаване на празна файлова система с командата `mkfs`, дисковото пространство на всеки дял се разделя на четири области (Фиг. 5).



Фиг.5. Разпределение на дисковото пространство в s5fs на UNIX

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 `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, 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 следните:

```
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.

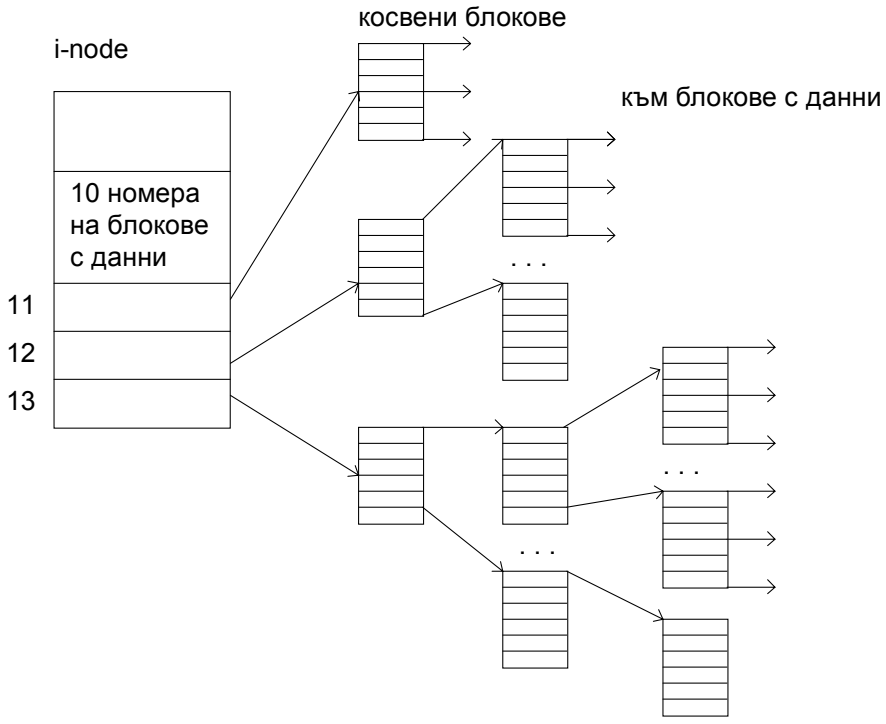
Полетата `di_uid` и `di_gid` определят собственика и групата на файла.

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.6 е представена структурата при съхраняване на номерата на блоковете, разпределени за файл.



Фиг.6. Представяне на файл в UNIX

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 1К байта, което означава, че в един косвен блок се побират 256 адреса. Тогава ограничението за размера на файл е:

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

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 2 KB, 4 KB или повече, при което ограничението за размера ще стане много по-голямо число, напр. при блок 4 KB - от порядъка на 4 TB.

Представянето на файловете в UNIX системите 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 в таблицата на индексните описатели).

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 */
      char        s_fmod;    /* super block modified flag */
      char        s_ronly;   /* mounted read only */
      . . .
      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`.

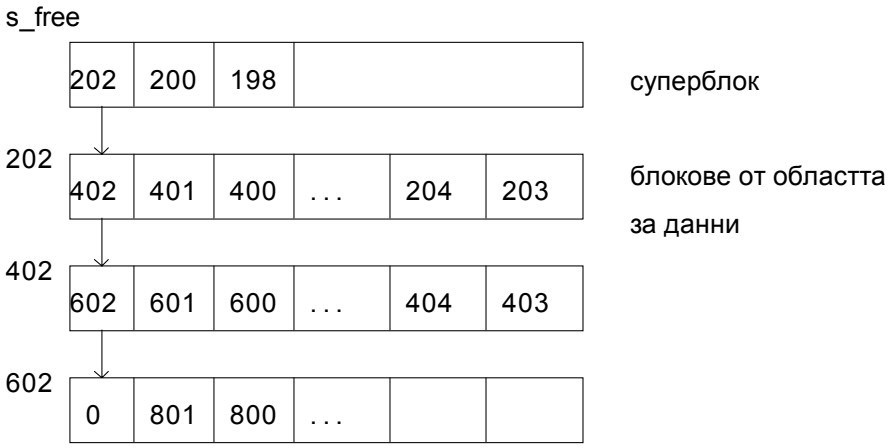
Полетата `s_flock` и `s_ilock` се използват като флагове за заключване на достъпа до списъка на свободните блокове и свободните индексни описатели по време на манипулирането им с цел избягване на състезание между конкурентни процеси.

Суперблоковете на коренната и всички монтирани файлови системи се зареждат в оперативната памет в таблицата на суперблоковете 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. 7 е изобразен примерен списък на свободните блокове.



Фиг. 7. 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.

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).

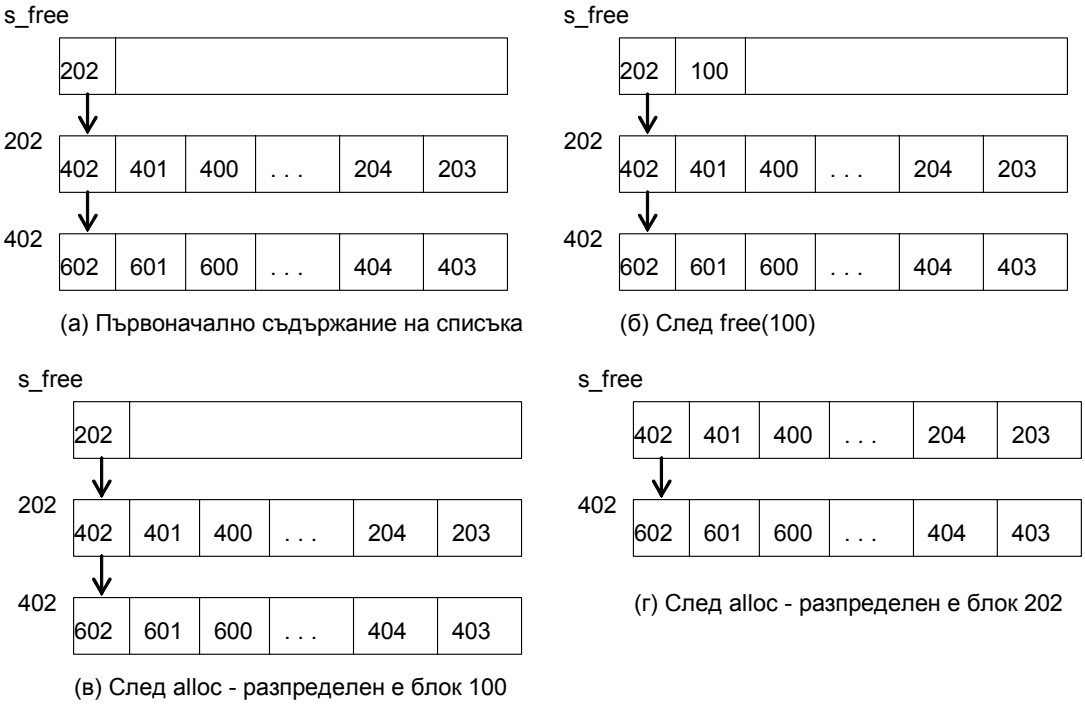
2. Ако `s_nfree` е 0, то се попълва масива `s_free`, като се прочита първия блок от списъка, чийто адрес е в `s_free[s_nfree]`. Разпределя се току що освободения блок.

Алгоритъм `free` за освобождаване на блок.

1. Ако масивът `s_free` не е пълен, то номера 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, т.е. освобождавания блок става първи блок в списъка а `s_free` е празен.

На Фиг. 8 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 (в същност организацията е стек). Затова е безмислено да се правят усилия за възстановяване на случайно изтрити файлове и такива средства отсъстват в командите на UNIX системите.



Фиг. 8. Разпределяне и освобождаване на блокове

Списък на свободните индексни описатели

Другият ресурс, който файловата система трябва да управлява, са индексните описатели и те са разположени в отделно пространство - индексната област. При създаване на файл за него трябва да се разпредели свободен индескен 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 на файл и йерархичната структура на файловата система. В разглежданата s5fs файлова система записът в каталога има следната структура:

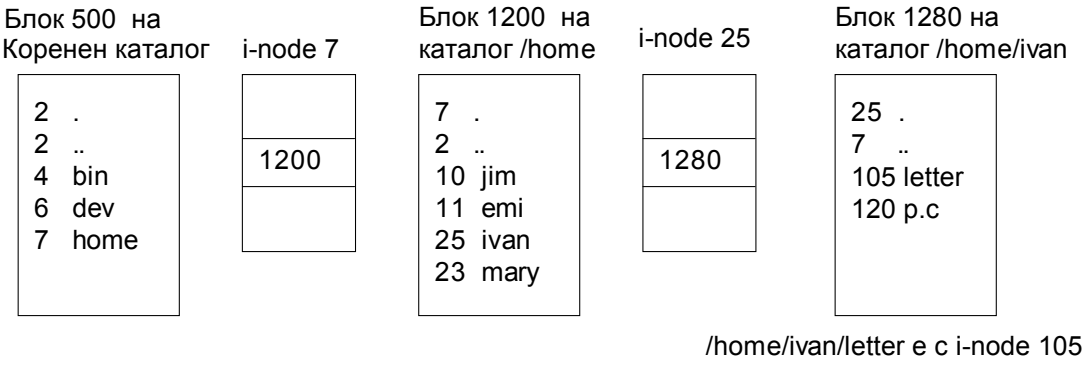
```
struct direct
{
      ushort     d_ino;
      char  d_name[DIRSIZE];
};
```

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. При реализацията на твърдите връзки, всички записи в каталози за определен файл съдържат един и същи номер на i-node.

Първите няколко индексни описателя 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 областта за данни.

При отваряне на файл файловата система трябва да преобразува пълното име на файла в индексен описател, при което се използват каталозите. Например, нека разгледаме търсенето на файл `/home/ivan/letter` и намирането на индексния му описател (Фиг.9). Търсенето започва от коренния каталог, чийто i-node е вече в паметта, в таблицата на индексните описатели. 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 таблицата на индексните.

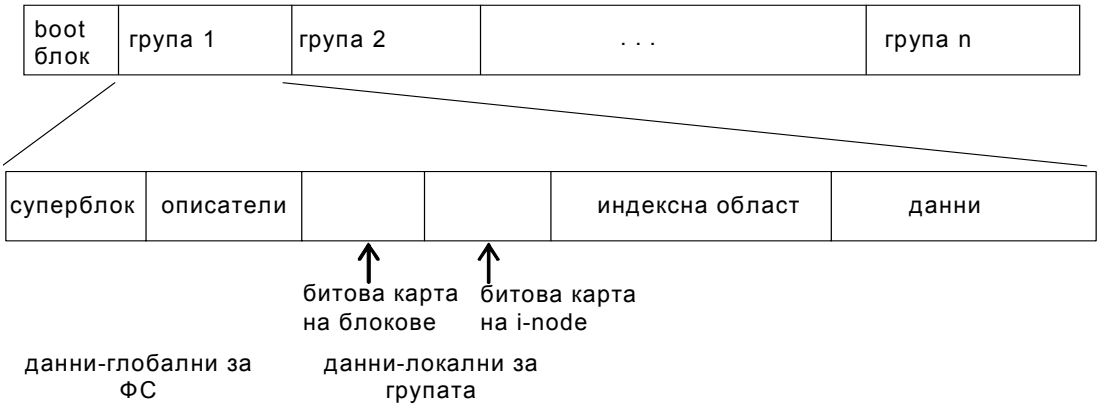


Фиг. 9. Търсене на файл `/home/ivan/letter`

Разгледаната файлова система s5fs има много преимущества, които отбелязахме в хода на разглеждането й, 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 далеко 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 UNIX, известна с наименованието Berkley Fast File System (FFS) или ufs. По-нататък ще разгледаме файловата система на LINUX, в която също са отстранени споменатите недостатъци.

2.4. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА СИСТЕМА В LINUX

Основната файлова система в LINUX е Extended File System-version 2 (ext2), а вече и 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. При управлението на дисковото пространство се използват блокове с фиксиран размер - 1KB, 2KB или 4KB. Това е и адресуемата единица на диска, единна за всички файлови системи. Фиг. 10 представя областите, в които е организирано дисковото пространство.



Фиг.10. Разпределение на дисковото пространство в LINUX

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 `e2fsck`), 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 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/ext3 е разширение на i-node от UNIX с нови полета. Размерът й е 128 байта. 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`)
- флагове:

immutable - Файлът не може да се изменя, унищожава, преименува и да се създават нови връзки към него.

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

synchronous write - Системният примитив `write` завършва след като данните са 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 бъдещо планирано развитие на файловата система.

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) се използва като разширение на `i_size` при обикновени файлове, т.е. размерът на файла се съхранява в 64 битово цяло число.

Описатели на групи блокове (Group descriptors)

Всяка група е описана чрез запис, с размер 32 байта съдържащ:

- адрес на блок с битовата карта на блоковете за групата;
- адрес на блок с битовата карта на i-node 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 резервирани за администратора блока (обикновено е 5% от общия брой блока). Тези резервирани блокове позволяват администратора да продължи работа, дори когато няма свободни блокове за другите потребители.
- брой свободни блокове и индексни описатели
- размер на блок
- брой блока в група
- брой i-node в група
- полета използвани за 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 два fsck.

Каталог

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 ext2_dir_entry {
      __u32  inode;          /* Inode number */
```

```
        __u16  rec_len;        /* Directory entry length */
        __u16  name_len;       /* Name length */
        char   name[EXT2_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 за съдържание на каталог, в който името `oldfile` е изтрито от каталога:

inode	rec_len	name_len	name							
21	12	1	.	\0	\0	\0				
22	12	2	.	.	\0	\0				
54	16	5	h	o	m	e	l	\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 и ext3 в сравнение с файловата система в UNIX е:

- Използване на битови карти при управление на свободните ресурси - блокове и i-node;
- Разделяне на дисковото пространство на групи блокове.

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 съхранявя в i-node в адресните 60 байта. По този начин се спестява един блок.

2.5. 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. 11 е изобразено разпределението на дисковото пространство в една файлова система, изградена на един том.

Фиг.11. Разпределение на дисковото пространство в том на MSDOS

Boot сектор 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 паметта, разпределена за файловете, за свободното дисково пространство, за дефектните сектори и за формата на диска. 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, кореният 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

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

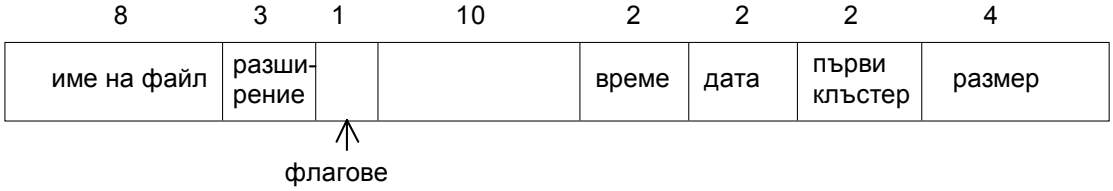
FAT съдържа елементи с дължина 12 или 16 бита, като броят им зависи от размера на тома и от размера на клъстера. Елементи 0 и 1 съдържат код, идентифициращ формата на тома. Всеки от останалите елементи съответства позиционно на клъстер от областта за данни. За удобство номерацията на клъстерите започва 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) 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), т.е. 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. 12 е показано примерно съдържание на FAT. Използвани са следните обозначения: EOF - код за последен клъстер (0xFFFF) и FREE - код за свободен клъстер (0).

Фиг. 12. Таблица FAT в MSDOS

Дължина на елемент от FAT първоначално е 12 бита (това е FAT12). При FAT12 могат да се адресират до до 2^{12} (4096) клъстера. С навлизането на по-големи дискове, размерът на елемент се увеличава на 16 бита. При FAT16 могат да се адресират до до 2^{16} (65536) клъстера. Затова размерът на клъстера е различен (от 512 байта до 64KB), в зависимост от размера на тома, където се изгражда файловата система.

Каталози

Едно от различията при представяне на каталозите в MSDOS е, че коренният 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. 13.



Фиг.13. 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 е в един бит и предназначението им е следното:

Битове

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

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

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.

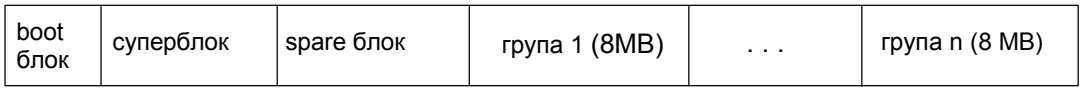
Всички каталози без коренния съдържат двата стандартни записа - с име "." за самия каталог и с име ".." за родителския каталог. Коренният каталог пък съдържа един специален запис за етикета на тома, който е символен низ с дължина до 11 символа и се записва в първите две полета на записа. Това е другата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

Ранните версии на Windows (Win 3.x, Win95, Win98) използват файловата система на MSDOS, но въвеждат дълги имена на файлове в Unicode, които не следват правилото 8.3 на MSDOS. Това налага някои промени в каталога. Ако името е дълго или в Unicode, то за файла има няколко записа в каталога. Допълнителните записи, толкова колкото са необходими, съдържат дългото име. Един главен запис съдържа съкратено име на файла във формат “xxxxxx~1.yyy”, което автоматично се генерира от файловата система, и атрибутите му. Освен това в резервираното място се добавят допълнителни атрибути – време на създаване, дата на създаване и дата на последен достъп (освен старите време и дата, които са за последно изменение).

Друга промяна във файловата система, наложена от големите дискове, е въвеждане на FAT32. В тази файлова система, освен възможността да се адресират по-голям брой клъстери (до 2^{28}, защото старшите 4 бита са резервирани), се премахва и ограничението за фиксираното място и размер на коренния каталог.

2.6. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА СИСТЕМА HPFS В OS/2

Файловата система HPFS (High Performance File System) е реализирана в OS/2. За файловата система HPFS дисковото пространство е последователност от сектори по 512 байта, с адреси от 0, 1, 2, и.т.н. Секторът е адресуемата единица дискова памет, т.е. когато по-нататък в изложението използваме термина адрес или указател, това означава номер на сектор. На Фиг. 14 е изобразено разпределението на дисковото пространство в една HPFS файлова система.



Фиг.14. Разпределение на дисковото пространство в HPFS

Boot блока заема 8KB и съдържа, както и в останалите системи, програмата за начално зареждане на системата и системна информация, като напр. име и идентификатор на тома.

Суперблокът е с размер един сектор и съдържа информация за файловата система, като:

- адреси на битовите карти;
- адрес на списъка на лошите блокове;
- адрес на групата за каталози;
- адрес на Fnode на коренния каталог;
- дата и време на последна проверка на коректността на файловата система.

Spare блок е също с размер един сектор и е допълнителнение към суперблока.

Останалата част от дисковото пространство е разделена на групи сектори по 8MB. Всяка група има собствена битова карта, описваща свободните сектори в групата. Размерът на всяка битова карта е 4 сектора, т.е. 2048 байта. Битовите карти са разположени в началото или в края на групата, така че за две съседни групи битовите им карти да се съседни. Така най-голямото непрекъснато дисково пространство, което може да се разпредели за файл е 16MB.

Една от групите, разположена около средата на диска се нарича група за каталози (directory block band) и съдържа каталозите на файловата система, но ако се запълни, каталози могат да се съхраняват и в останалите групи.

Fnode

Представянето на всеки файл или каталог включва системна структура, наречена Fnode, който заема един сектор и съдържа:

- Атрибути на файла:
  - Размер на файла
  - Флагове (от MSDOS)
  - Дължина на името на файла и първите 15 символа от името на файла;
  - ACL (Access Control Lists), съдържащ информация за правата на достъп до файла и пароли за достъп;
  - Разширени атрибути (Extended attributes).
- Структура, описваща разпределените за файла сектори (Allocation structure).

От гледна точка на описанието на разпределената дискова памет, файлът е последователност от няколко непрекъснати области на диска с различни дължини. Всяка такава непрекъсната област заема един или няколко физически съседни сектора и се нарича ***екстент*** (***extent/run***). Всеки екстент се описва с две 32 битови цели числа: адрес на първи сектор и брой сектори в екстента (нарича се run-length encoding).

Структурата, описваща разпределената дискова памет, има различни формати в зависимост от размера и степента на непрекъснатост на файла.

1. Ако файлът заема не повече от 8 екстента, цялото описание на разпределената дискова памет е в Fnode. Ако файлът е твърде голям или фрагментиран, за да бъде описан като 8 екстента, тогава структурата на описание става B+ дърво с два възможни варианта.

2. B+ дърво с две нива. Fnode е корен на B+ дървото и съдържа до 12 елемента, всеки от които съдържа адрес на сектор (***allocation sector***), съдържащ до 40 описания на екстенти. По този начин могат да се опишат до 12*40=480 екстента, което при максимална непрекъснатост на файла е размер 480*16MB = 7.68GB.

3. B+ дърво с три нива. Fnode е корен на B+ дървото и съдържа до 12 елемента, всеки от които съдържа адрес на сектор, съдържащ до 60 адреса на сектори от трето ниво, които

съдържат до 40 описания на екстенти. По този начин максималният брой екстенти за файл е 12*60*40=28800.

И при трите формата на системната структура, листата съдържат описания на екстенти, всяко по 8 байта. Възел, който не е лист в B+ дърво, съдържа записи от указател и максимален номер на файлов блок, описан в поддървото, сочено от указателя.

Каталози

Каталозите, както и файловете се представят чрез Fnode. Разпределяната за каталог дискова памет е в единици от 4 последователни сектора - каталожен блок (directory block), като се отделя памет от групата за каталози, но след запълването ѝ, от всяка друга група. Всеки каталог съдържа записи с променлива дължина, всеки описващ един файл и съдържащ:

- адрес на Fnode;
- дължина на записа;
- дължина на името;
- име на файл или каталог, описван в записа;
- указател на B дърво - адрес на блок на каталога, съдържащ записи, чиито имена са по-малки от името в текущия запис и по-големи от името в предишния запис.

Записите във всеки блок на каталог са сортирани по азбучен ред на името на файла. Когато каталог надхвърли размера на каталожен блок, се разпределят допълнителни блокове, които са организирани в B дърво. Целта на това представяне е бързо търсене на файл по име в голям каталог. Ако средната дължина на име на файл е 13 символа, то в каталожен блок се събират около 40 записа, което при B дърво с две нива означава около 1640 записа в каталога, а при B дърво с три нива - 65640. Следователно, най-много с три дискови операции може да бъде намерен файл по име или да се установи, че не съществува такъв. За сметка на бързото търсене, създаването, триенето и преименуването на файл става в общия случай бавна операция.

Ефективност

Основна цел при проектирането на файловата система на OS/2 е да се създаде високо производителна система. Това се постига чрез системните структури и подходящи стратегии и алгоритми за разпределение на дискова памет.

Системните структури дават следните предимства:

- B дърво и B+ дърво осигуряват бързо търсене на файл в каталог по име и бърз достъп до дисков блок по логически адрес - отместване във файла.
- Описанието на разпределените сектори, отразява физическото съседство. Това означава, че при нарастване на файл описанието не винаги нараства. Ако новоразпределеният сектор е физически съседен на предишния последен, то новият сектор се добавя към последния екстент, т.е. изменя се само дължината му, а не се добавя нов екстент. Това може да се използва от алгоритмите за достъп, като се четат наведнаж няколко последователни сектора.
- Битовата карта е компактна структура, позволяваща при разпределението на сектор да се отчита физическото съседство.

Стратегиите и алгоритмите се стремят:

- Да разполагат системните структури - битови карти и Fnode близко до обектите, които описват. Всяка група има своя битова карта. При създаване или нарастване на файл алгоритъма се старае сектора с Fnode и секторите с данни да са в една група и по възможност физически близко.
- За файловете да се разпределят непрекъснати области от диска когато и доколкото това е възможно. Задачата за намаляване на фрагментираността на файл в многопроцесна система, без да се затруднява потребителя не е лесна за решаване. Една от техниките, използвани тук е предварително разпределяне - всеки път, когато за файла е необходим сектор, алгоритъма се опитва да разпредели непрекъсната област с размер около 4KB. Излишните сектори се освобождават при `close`.

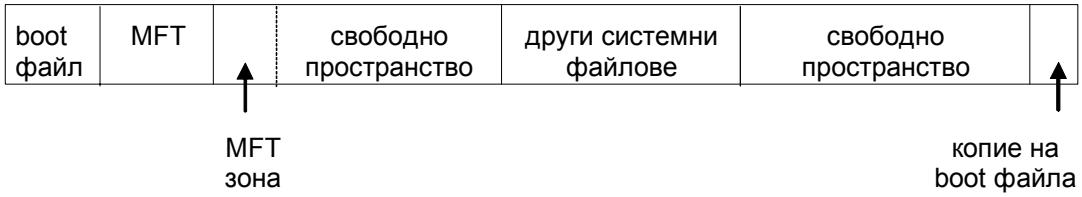
2.7. ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВАТА СИСТЕМА NTFS

NTFS се реализира като напълно нова файлова система в Windows NT и се използва в следващите версии на Windows 2000. 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. 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 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. 15 е изобразено разпределението на дисковото пространство в една новоформатирана файлова система.



Фиг.15. 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 24 записа 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 (16) 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 съвкупност 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 (общият им брой е около 14).

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 флагове, време и дата на създаване и последно изменение, брой твърди връзки.
$DATA	Данните на файла. Всеки файл има един неименован атрибут данни и може да има допълнителни именовани атрибути данни.
$INDEX_ROOT, $INDEX_ALLOCATION, $BITMAP	Три атрибута използвани при реализацията 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, $VOLUME_INFORMATION	Тези атрибути се използват само в файла $Volume. Те съхраняват иформация за тома, като етикет, версия.

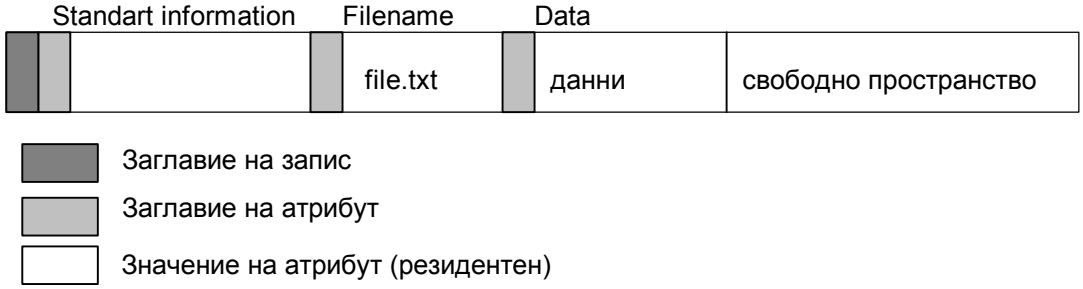
Всеки тип атрибут освен име на типа, има и числов код на типа. Този код се използва при наредба на атрибутите в MFT запис на файла. MFT записът съдържа числовия код на атрибута (който го идентифицира), значение на атрибута и евентуално име на атрибута. Значението на всеки атрибут е поток от байтове.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

Атрибутите биват резидентни и нерезидентни. Резидентен е атрибут, който се съхранява изцяло в MFT записа. Някои атрибути са винаги резидентни, например $FILE_NAME, $STANDART_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. Заглавието на

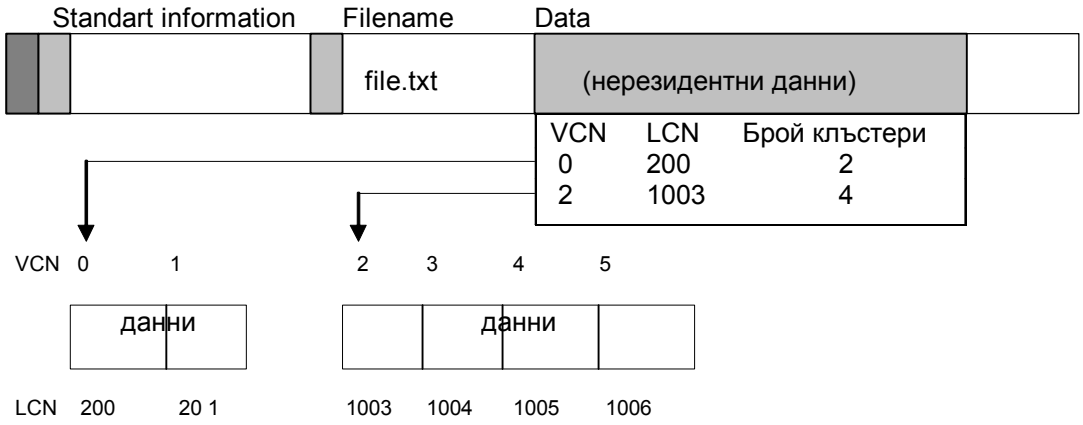
атрибута съдържа код на типа, име, флагове на атрибута и информация за разположението на данните му. Един атрибут е резидентен ако данните му се поместват в един запис заедно със заглавията на всички атрибути. На Фиг. 16 е изобразена структурата на MFT запис за малък файл, т.е. всичи атрибути на файла могат да се съхранят в MFT записа.



Фиг.16. MFT запис за файл само с резидентни атрибути

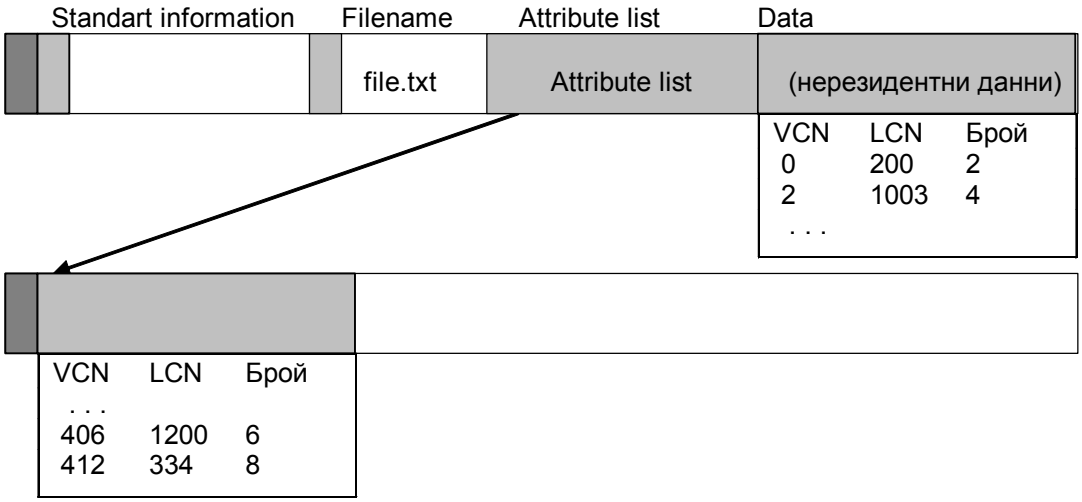
Ако един атрибут не е резидентен, заглавието му (което е винаги резидентно) съдържа информация за клъстерите, разпределени за данните му. Адресната информация се съхранява подобно на подхода в HPFS, т.е. представлява последователност от описания на екстенти (run/extent entry). Всеки екстент е непрекъсната последователност от клъстери, разпределени за данните на съответния атрибут и се описва от адрес на началния клъстер и дължина (VCN, LCN, брой клъстери). За разлика от HPFS, тук описанията на екстентите се съхраняват в последователна структура, а не в B+дърво.

На Фиг. 17 е изобразена структурата на MFT запис за по-голям файл, т.е. данните на файла се съхранят в два екстента.



Фиг.17. MFT запис за файл с нерезидентен атрибут данни

Ако 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. Адресната информация за един нерезидентен атрибут може да се съхранява в няколко записа, ако обема на данните е голям или разпределената памет е фрагментирана. На Фиг. 18 е изобразена структурата на MFT записи за голям файл, т.е. описанието на екстентите не се помества в един запис и се използва допълнителен запис и атрибут $ATTRIBUTE_LIST.

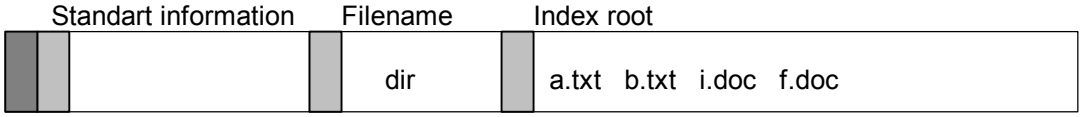


Фиг.18. MFT записи за голям файл

Каталози

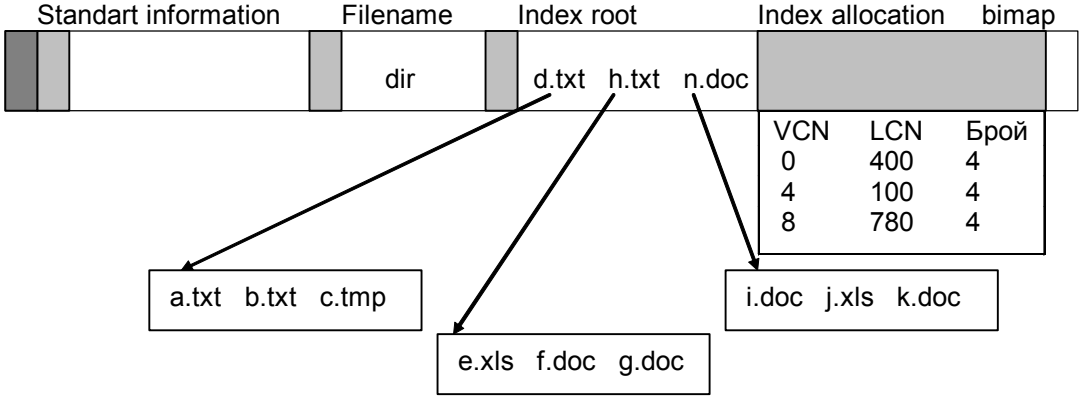
Каталогът съдържа записи с променлива дължина, всеки от които съответства на файл или подкаталог, в съответния каталог. Всеки запис съдържа името и индексът на основния MFT запис на файла, както и копие на стандартната информация на файла. Това дублиране на стандартната информация изисква две операции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

Записите в каталога са сортирани по името на файла и се съхраняват в структура B дърво, подобно на HPFS. Ако каталогът е малък, тези записи се съхраняват в атрибута $INDEX_ROOT, който е резидентен, т.е. целият каталог се намира в MFT записа си. На Фиг. 19 е изобразена структурата на MFT запис за малък каталог.



Фиг.19. 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. 20 илюстрира представянето на голям каталог (за простота всеки клъстер съдържа 1 запис).



Фиг.20. Представяне на голям каталог

3. СИСТЕМНИ ПРИМИТИВИ НА ФАЙЛОВАТА СИСТЕМА

Системните примитиви реализират операциите, които файловата система предоставя на потребителите за:

- манипулиране на отделни файлове;
- манипулиране на структура на файловата система;
- защита на файловата система.

Ше разгледаме основните системни примитиви на файловата система по стандарта POSIX, който е реализиран в повечето UNIX системи - XENIX, 4.3BSD, SunOS, AIX, HP-UX, OSF/1 и др, в LINUX и MINIX. Повечето съвременни операционни системи предоставят системни примитиви, изпълняващи същите функции, макар да има различия при реализацията им.

3.1. СИСТЕМНИ ТАБЛИЦИ

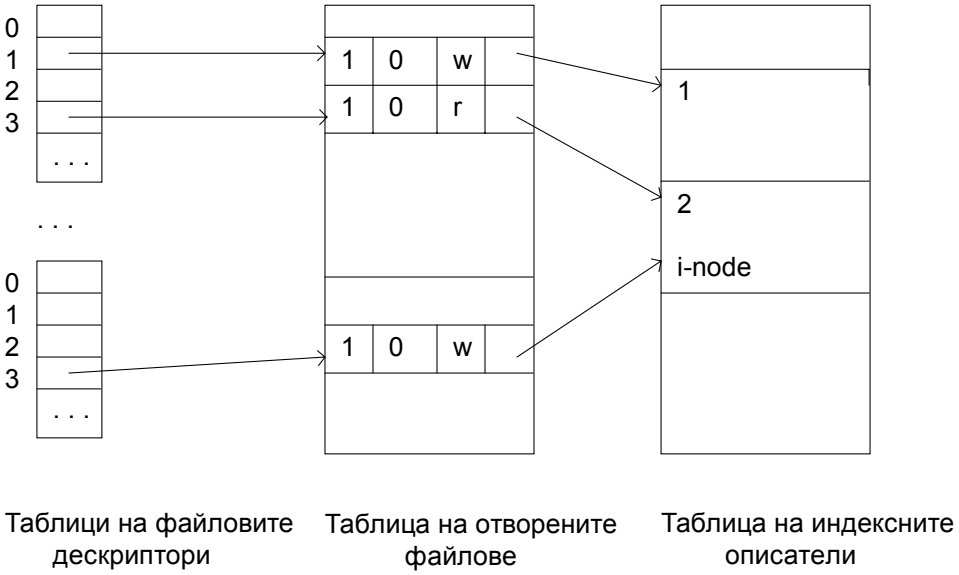
Преди да разгледаме системните примитиви ще въведем някои необходими понятия и системни структури, използвани при работа на файловата подсистема, наричани системни таблици.

Файлов дескриптор (***file descriptor***) е неотрицателно цяло число, от 0 до OPEN_MAX-1 (обикновено 19 в по-ранните и 63, 255 и повече в по-новите версии на UNIX и LINUX), което се свързва с файл при отваряне и се използва за идентифициране на файла при последващата му обработката. 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.

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 манипулиране на 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, достъпа се отказва независимо от правата на процеса.

Реализацията на системните примитиви на файловата система използва следните основни ***системни таблици*** в пространството на ядрото - ***Таблица на индексните описатели***, ***Таблица на отворените файлове*** и ***Таблици на файловите дескриптори*** (Фиг.21). За всеки отворен файл в таблицата на индексните описатели се пази точно един запис, съдържащ копие на i-node от диска и някои други полета, като номер на устройството и номер на i-node, състояние на i-node (заключен, изменен), брояч (брой указатели, насочени към записа). В таблицата на отворените файлове се пази текущата позиция и режима на отваряне на файла. При всяко отваряне на файл се отделя един запис в тази таблица, който съдържа и указател към съответния запис от таблицата на индексните описатели и брояч (брой указатели, насочени към записа, 1 след отваряне). Всеки процес има собствена таблица на файловите дескриптори. Броят на елементите в нея определя максималния брой едновременно отворени файлове за процес (`OPEN_MAX`). Записите в тези таблици съдържат само указател към запис от таблицата на отворените файлове. Файловият дескриптор всъщност представлява индекс на използвания при отварянето запис от таблицата на файловите дескриптори.



Фиг. 21. Таблици на файловата система

При всяко отваряне на файл се създава верига в системните таблици от файловия дескриптор до индексния описател, с което се осигурява достъп на процеса до данните на файла независимо от останалите процеси.

3.2. СИСТЕМНИ ПРИМИТИВИ ЗА МАНИПУЛИРАНЕ НА ФАЙЛ

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.

Създаване на обикновен файл

```
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 `w`. Процесът трябва да има права `x` за всички каталози на пътя в пълното име и право `w` за родителския каталог. Създаденият файл се отваря за писане и `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 функциите на `open` са разширени. Чрез `open` може да се създаде файл, ако не съществува и след това да се отвори. Функциите на системния примитив се конкретизират чрез параметъра *`oflag`*. Значенията на *`oflag`* се конструират чрез побитово ИЛИ (|) от следните символни константи:

- Определящи начин 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.

Алгоритъм на open.

1. Преобразува пълното име на файла в номер на i-node.
2. Ако файлът не съществува, то ако е зададен флаг O_CREAT, създава файл с име *filename*, в противен случай завършва и връща -1.
3. Ако файлът съществува и са зададени флагове O_CREAT и O_EXCL, то завършва и връща -1.
4. Зарежда i-node на файла в таблицата на индексните описатели, ако вече не е там.
5. Проверява правата на процеса за достъп към файла.
6. Разпределя свободен запис от таблицата на отворените файлове и го инициализира (брояч - 1; текуща позиция - 0; режим на отваряне - от параметъра *oflag*; указател към записа от таблицата на индексните описатели).
7. Разпределя свободен запис от таблицата на файловите дескриптори и го инициализира с указател към записа от таблицата на отворените файлове. Запомня индекса на записа, който ще е файловият дескриптор.
8. Ако е зададен флаг O_TRUNC, то се освобождават всички блокове на файла.
9. Връща файловия дескриптор.

Пример. Да предположим, че процес А изпълни следния код:

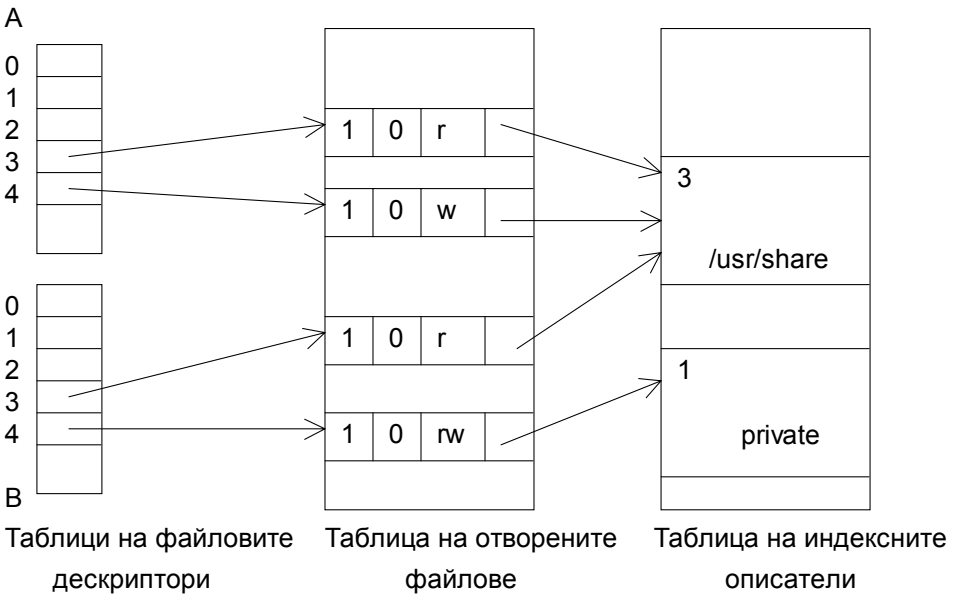
```
fd1 = open("/usr/share", O_RDONLY);
fd2 = open("/usr/share", O_WRONLY);
```

а процес В изпълни системните примитиви:

```
fd1 = open("/usr/share", O_RDONLY);
fd2 = open("private", O_RDWR);
```

Фиг. 22 показва връзките между съответните структури, докато двата процеса (и никои други) са отворили тези файлове.

Всеки open разпределя нов запис в съответната таблица на файловите дескриптори и нов запис в таблицата на отворените файлове. Защо тогава има отделна структура - Таблица на отворените файлове? От всичко казано до сега се вижда, че съответствието между записите в тези структури е едно към едно. Но това е за сега. Системните примитиви dup и fork, които ще разгледаме по-нататък, позволяват няколко записа от таблиците на файловите дескриптори да сочат към един запис от таблицата на отворените файлове.



Фиг. 22. Системните таблици след `open`

Затваряне на файл

`int close(int fd);`

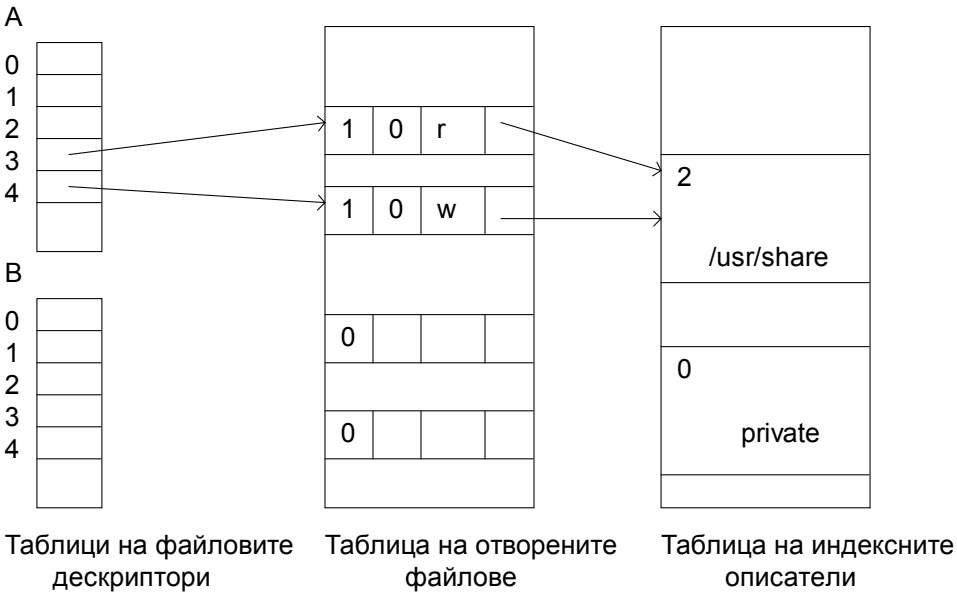
Примитивът `close` затваря отворен файл, т.е. прекратява връзката между процеса и файла, като освобождава файловия дескриптор, който след това може да бъде използван при последващи изпълнения на `creat` или `open`. Връща 0 при успех или -1 при грешка.

Алгоритъм на `close`.

1. Намалява с 1 полето брояч в съответния запис от таблицата на отворените файлове.
2. Ако новото значение е по-голямо от 0, то освобождава елемента от таблицата на файловите дескриптори и завършва като връща 0.
3. Ако новото значение е 0, то освобождава елемента от таблицата на файловите дескриптори и записа от таблицата на отворените файлове.
4. Намалява с 1 полето брояч в съответния запис от таблицата на индексните описатели.
5. Ако новото значение е равно на 0 (никой друг не осъществява достъп до файла), то i-node се записва на диска (ако е бил изменен) и записа се счита за свободен.
6. Връща 0.

Пример. На Фиг.23 е показано съдържанието на съответните записи в системните таблици след като процес B е изпълнил `close` за двата си отворени файла:

```
close(fd1);
close(fd2);
```



Фиг. 23. Системните таблици след `close`

Четене и писане във файл

```
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 EOF или -1 при грешка.

```
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 преди всяко изпълнение на `write` текущата позиция се установява в края на файла. 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. Докато не се прочете или запише искания брой байта, i-node на файла е заключен в таблицата на индексните описатели. Отключва се преди завършване на примитива. Това означава, че друг процес не може да осъществи достъп до файла докато не завърши изпълнението на определен `read` или `write`, т.е. всеки `read` или `write` може да се счита за неделима операция когато се разглежда проблема за състезания между процеси при достъп до общи файлове.

Различието в алгоритмите на `read` и `write` е в следното:

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

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

- При `write` се използва подход наречен отложен запис (delayed write), т.е. данните се изменят в буфера, 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.

`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 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 в съответния запис от таблицата на отворените файлове. Ако новото значение е след EOF, 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 файла 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 `lseek` е на разстояние след края на файла, то последващия `write` ще започне писането от текущата позиция и ще увеличи размерът на файла. Така може да се създаде „файл с дупка”. При четене всички байтове от дупката са нулеви (\0).

Пример. Програмата създава „файл с дупка”.

```
#include <stdio.h>
#include <fcntl.h>
char buf1[] = "ABCDEF";
char buf2[] = "abcdef";
main(void)
{
  int fd;
  if ((fd = creat("file.hole", 0640)) < 0){
            fprintf(stderr, "can't create\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);
}
```

Информация за файл

Част от съхраняваната за файла информация в индексния му описател 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 параметър *`sbuf`* е указател на структура `stat`, в която примитивът записва информацията за файла.

```
struct stat {
   dev_t st_dev;           /* device where inode belongs */
   ino_t st_ino;           /* inode number */
   mode_t st_mode;         /* mode word */
   nlink_t st_nlink;       /* number of hard links */
   uid_t st_uid;           /* user ID of owner*/
   gid_t st_gid;           /* group ID of owner */
   dev_t st_rdev;          /* for special files */
   off_t st_size;          /* file size */
   blksize_t st_blksize;   /* blocksize for filesystem I/O */
   blkcnt_t st_blocks;     /* number of blocks allocated */
   time_t st_atime;        /* time of last access */
   time_t st_mtime;        /* time of last modification */
   time_t st_ctime;        /* time of last change */
};
```

Пример. Програмата `copy` създава ново копие на съществуващ файл, чието име е зададено като първи параметър, а името на създавания файл, като втори параметър.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#define BUFS 1024
#define MODE 0644
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 = creat(argv[2], MODE)) == -1) {
              fprintf(stderr, "copy: %s: can't create\n", argv[2]);
              exit(1); }
        while (( n = read(fdr, buff, BUFS)) > 0)
              write(fdw, buff, n);
}
```

Копиране на файлов дескриптор

Последният системен примитив от този раздел `dup` извършва странно на пръв поглед манипулиране на файлови дескриптори.

`int dup(int oldfd);`

`int dup2(int oldfd, int newfd);`

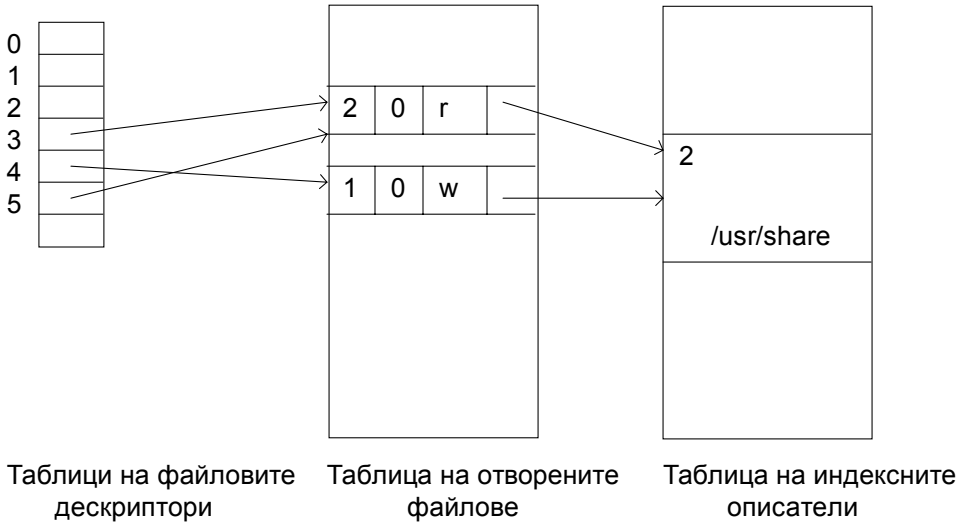
Параметърът *`oldfd`* е файлов дескриптор на отворен файл (върнат от `creat`, `open`, `dup`). Примитивът `dup` разпределя първия свободен в момента файлов дескриптор и там копира *`oldfd`*. Увеличава с 1 полето брояч в съответния запис от таблицата на отворените файлове. Примитивът `dup2` копира *`oldfd`* на мястото на *`newfd`*, като затваря *`newfd`* преди това ако е необходимо. И двата примитива връщат новия файлов дескриптор при успех или -1 при грешка. Следователно двата файлови дескриптора имат следното общо:

- един и същи отворен файл;
- общ указател на текуща позиция;
- еднакъв режим на отваряне на файла.

Пример. Да предположим, че процес е изпълил системните примитиви:

```
fd1 = open("/usr/share", O_RDONLY);
fd2 = open("/usr/share", O_WRONLY);
fd3 = dup(fd1);
read(fd1, buf1, sizeof(buf1));
read(fd3, buf2, sizeof(buf2));
```

На Фиг.24 е показано съдържанието на съответните записи в системните таблици. От там се вижда, че двата буфера `buf1` и `buf2` няма да съдържат еднакви, а последователни данни на файла. След `close(fd1)`, четенето от файла може да продължи нормално чрез `fd3`.



Фиг 24. Системните таблици след `dup`

Примитивът `dup` се използва при пренасочване на стандартен вход, изход или изход за грешки и при организиране на конвейер от програми.

Пример. Програмен фрагмент за пренасочване на стандартен вход от файл:

```
if ( (fd = open(infile, O_RDONLY)) == -1 ) {
      fprintf(stderr, "can't open file %s\n", infile);
      exit(1); }
close(0);
dup(fd);
close(fd);
```

3.3. СИСТЕМНИ ПРИМИТИВИ ЗА МАНИПУЛИРАНЕ НА СТРУКТУРАТА НА ФАЙЛОВАТА СИСТЕМА

Тази група включва системни примитиви, чрез които се изгражда и манипулира системата от каталози, създават се и се унищожават връзки към файл и се изгражда единната йерархична структура на файловата система.

Създаване и унищожаване на каталог

`int mkdir(const char *`*`dirname`*`, mode_`*`t mode`*`);`

Примитивът `mkdir` създава нов празен каталог с име *`dirname`* и код на защита *`mode`*. Празен каталог означава, че той съдържа само двата стандартни записа с име ".", съответстващо на самия каталог и с име ".." , съответстващо на родителския му каталог. Процесът трябва да има права `x` за всички каталози на пътя в пълното име и право `w` за родителския каталог. Грешка ще е и ако вече съществува файл с име *`dirname`*. При успех връща 0, а при грешка -1.

`int rmdir(const char *`*`dirname`*`);`

Примитивът `rmdir` унищожава каталог с име *`dirname`*, който трябва да е празен, т.е. да съдържа само двата стандартни записа - "." и "..". Процесът трябва да има права `x` за всички каталози на пътя в пълното име и право `w` за родителския каталог. При успех връща 0, а при грешка -1.

`int mknod(const char *`*`name`*`, mode_`*`t mode`*`, dev_t `*`dev`*`);`

Примитивът `mknod` първоначално е бил предназначен за създаване на произволен тип файл - обикновен, каталог, специален файл и др. Параметърът *`mode`* съдържа типа на създавания файл и кода на защита. Параметърът *`dev`* се използва при създаване на специален файл, в другите случаи трябва да е 0. В по-новите версии на UNIX и LINUX този примитив се използва само за създаване на специални файлове, обикновени файлове и FIFO файлове, а за каталози е добавен `mkdir`. Процесът, в който се изпълнява `mknod`, трябва да принадлежи на привилегирования потребител (`root`) ако типа на създавания файл е различен от FIFO файл. При успех връща 0, а при грешка -1.

Създаване и унищожаване на връзка към файл

Предназначението на `link` и `symlink` е да позволи достъп към един файл чрез различни имена, евентуално разположени в различни каталози на файловата система.

`int link(const char *`*`oldname`*`, const char * `*`newname`*`);`

Създава нова ***твърда връзка*** (***hard link***) за съществуващ файл, като добавя нов запис за файла в каталог. Параметърът *`oldname`* задава име на съществуващ файл, а *`newname`* - новото име на файла. Процесът трябва да има права `x` за всички каталози в пълното име *`newname`* и *`oldname`*, и право `w` за родителския каталог на *`newname`*. Това, което всъщност извършва `link` е да включи нов запис в родителския каталог на *`newname`* с новото име и номер на i-node от записа за *`oldname`*. Освен това в индексния описател на файла се увеличава с 1 броя на твърдите връзки за файла. В по-ранните версии на UNIX `link` е бил разрешен и за каталози,

което е било необходимо при създаване на каталози с примитива `mknod`. В по-новите версии тази необходимост не съществува, тъй като има примитив `mkdir` и затова `link` не е разрешен за каталози. При успех връща 0, а при грешка -1. При успешно завършване двете имена на файла са напълно равноправни и на практика не може да се отличи старото от новото име.

`int symlink(const char *`*`toname`*`, const char * `*`fromname`*`);`

Създава нова ***символна връзка*** (***symbolic link*** или ***soft link***) към файл. Това означава, че създава нов файл от тип символна връзка с име *`fromname`*, който сочи към файл *`toname`*, т.е. съдържанието на новия файл е низа *`toname`*. Процесът трябва да има права `x` за всички каталози в пълното име *`toname`* и *`fromname`*, и право `w` за родителския каталог на *`fromname`*. При успех връща 0, а при грешка -1.

Както и твърдата връзка, символната 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 *`toname`* при създаване на символната връзка. 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 към обикновен файл, специален файл и към каталог.

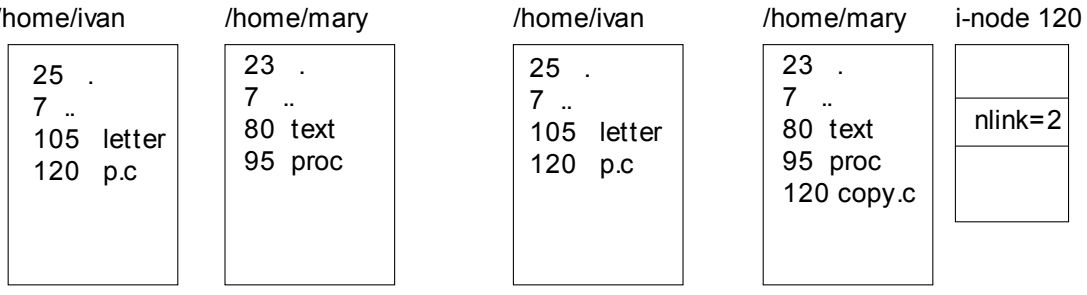
`int unlink(const char * `*`name`*`);`

Указаното име на файл *`name`* се изключва от файловата система, т.е. изключва се записа за *`name`* от каталога. Освен това в индексния описател на файла се намалява с 1 броя на твърдите връзки за файла. Ако броят стане 0, т.е. това е било последното име за файла, то той се унищожава (освобождават се блоковете с данни, индексния описател и косвените блокове), но след като последният процес, в който файлът е бил отворен, го затвори. Процесът трябва да има права `x` за всички каталози в пълното име *`name`* и право `w` за родителския каталог. В по-ранните версии на UNIX `unlink` е бил разрешен и за каталог, в по-новите не е, тъй като там има `rmdir`, но е разрешен за символна връзка, специален файл и FIFO файл. При успех връща 0, а при грешка -1.

За да се разбере по-добре действието на примитивите `link` и `symlink` ще разгледаме следния пример. Нека има два каталога със съдържание показано на Фиг. 25a. След изпълнение на примитива:

```
link("/home/ivan/p.c", "/home/mary/copy.c");
```

съдържанието на каталога и i-node се изменя както е показано на Фиг.25b. 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.



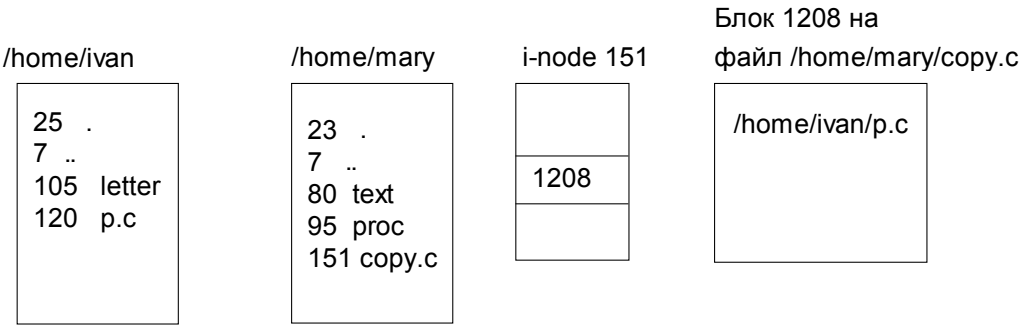
(а) (б)

Фиг. 25. Съдържание на каталозите - (а) преди и (б) след `link`

Нека съдържанието на каталозите е същото, както на Фиг. 25a. След изпълнение на примитива:

```
symlink("/home/ivan/p.c", "/home/mary/copy.c");
```

представянето на новото име е показано на Фиг.26.



Фиг. 26. Представянето на имената след `symlink`

Пример. Програмният фрагмент показва как може да се организира работата с временни (работни) файлове, т.е. файлове, които трябва винаги да се унищожават при завършване на процеса, дори и при аварийното му завършване.

```
main()
{
     . . .
     fd = open(temporary, O_RDWR | O_CREAT | O_TRUNC, mode);
     unlink(temporary);
     . . .
      /* четене-запис във временния файл чрез файловия дескриптор fd */
     close(fd);  /* унищожаване на временния файл */
}
```

Пример. Програмата `del` унищожава файлове, които не са каталози и чиито имена се задават като аргументи.

```
#include <sys/stat.h>
#include <stdio.h>

main(int argc, char *argv[])
{
int i;
struct stat sbuf;
      if (argc < 2) {
            fprintf(stderr, "usage: del file ...\n");
            exit(1); }
      for ( i = 1; i < argc; i++) {
            if ( stat(argv[i], &sbuf) == -1 ) {
                  fprintf(stderr, "del: %s: No such file\n", argv[i]);
                  continue; }
            if ( S_ISDIR(sbuf.st_mode) ) {
                  fprintf(stderr, "del: %s: is a directory\n", argv[i]);
                  continue; }
            if ( unlink(argv[i]) == -1)
                  fprintf(stderr, "del: %s: can't delete\n", argv[i]);
      }
}
```

Смяна на текущ каталог

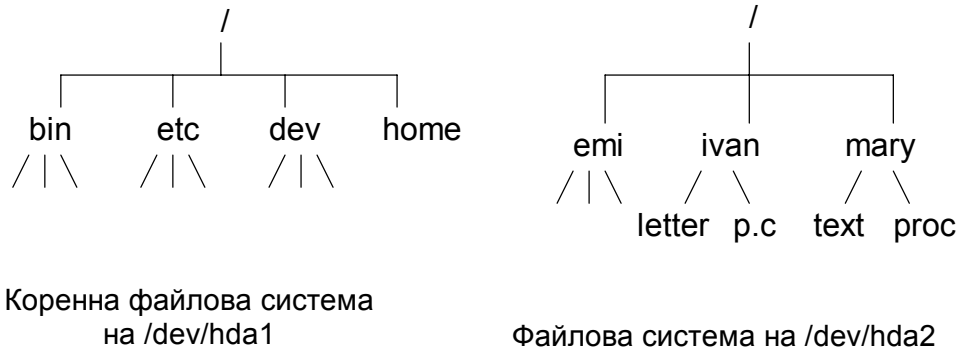
Позиционирането във файловата система, т.е. установяването на текущ каталог се извършва чрез примитива `chdir`.

```
int chdir(const char *dirname);
```

Указаният чрез аргумента каталог *dirname* става новия текущ каталог на процеса. Текущият каталог представлява активен файл, т.е. при изпълнение на chdir индексният описател на новия каталог се зарежда в таблицата на индексните описатели, а i-node на стария текущ каталог се освобождава. Процесът трябва да има права x за всички каталози в пълното име *dirname*. При успех връща 0, а при грешка -1 и текущия каталог не се променя.Текущият каталог, както и отворените файлове, се наследява от породените процеси.

Монтиране и демонтиране на файлова система

Системните примитиви mount и umount позволяват за потребителя винаги да се изгражда единна йерархична файлова система, независимо от броя на носителите (флопи дискове, твърди дискове или техни дялове). На всеки носител е изградена йерархична файлова система чрез командата mkfs, една от които съдържа програмата за начално зареждане и ядрото на операционната система, и се нарича коренна файлова система. Чрез mount некоренна файловата система на определен носител може да бъде присъединена (монтирана) към коренната файлова система. Например, нека конфигурацията включва две дискови устройства и върху носителите са изградени следните файлови системи (Фиг.27):

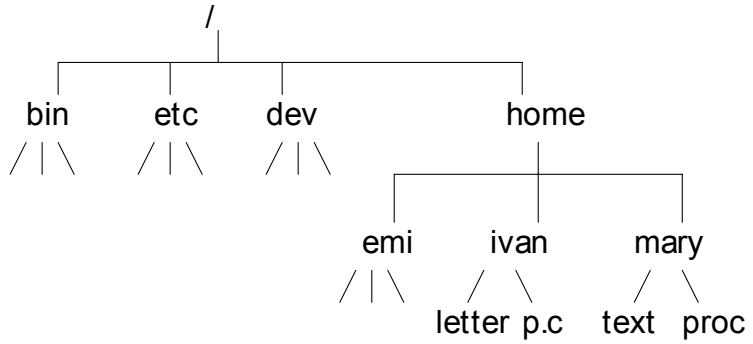


Фиг. 27. Две файлови системи преди монтирането

След изпълнение на системния примитив

```
mount("/dev/hda2", "/home", . . . );
```

за потребителя файловата система ще има следната структура (Фиг.28):



Фиг. 28. Файловата система след монтирането на /dev/hda2

Следователно, файловете, разположени на устройството /dev/hda2, ще са достъпни за потребителя чрез пълно име, напр., "/home/ivan/letter".

Разрушаване на връзката между коренната и коя да е друга монтирана файлова система се извършва чрез системния примитив umount. Точният синтаксис на двата примитива е системно зависим и в LINUX например е следния:

```
int mount(const char *special, const char *dirname,
```

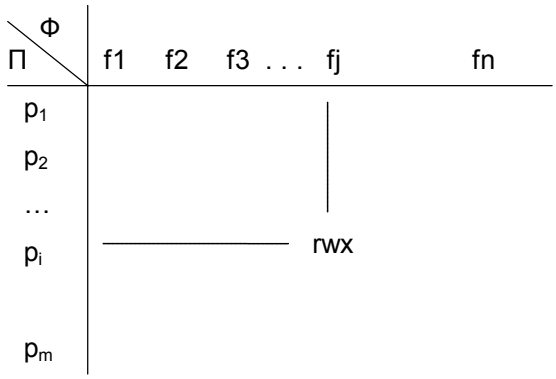
```
    const char *fstype, unsigned long flags, const void *data);
int umount(const char * dirname);
```

Процесът, в който се изпълнява `mount` или `umount` трябва да принадлежи на привилегирования потребител (`root`). Параметърът *`special`* е име на специалния файл на монтирана файлова система. При изпълнение на `mount` каталогът *`dirname`*, в който се монтира трябва да съществува, в него вече да не е монтирана файлова система и да не е активен в момента на монтиране (напр. да не е нечий текущ каталог). Аналогично, изпълнението на `umount` завършва с грешка и не се демонтира файловата система, ако в момента там има активен файл (отворен обикновен файл или текущ каталог). Останалите параметри са специфични за LINUX. При успех и двата примитива връщат 0, а при грешка -1.

3.4. СИСТЕМНИ ПРИМИТИВИ ЗА ЗАЩИТА НА ФАЙЛОВАТА СИСТЕМА

В многопотребителските системи не всички данни е разумно да са достъпни за всеки потребител. Администраторът трябва да определи на ***кого (субект)*** към ***кои данни (обекти) какъв тип достъп*** е позволен, а операционната система трябва да следи за спазването на тези правила. Обикновено защитата, реализирана от операционната система, е на ниво файл, т.е. обекти са файловете (от всякакъв тип). Типовете достъп могат да са четене, запис и изпълнение. Субекти са отделните потребители, регистрирани в системата.

Най-естествената структура за съхраняване на разрешените права на достъп за всички субекти към всички обекти е ***матрица на достъпа*** (Фиг. 29). Тя представлява двумерна матрица, в която всеки ред съответства на потребител, а всеки стълб на файл. В елемента на матрицата с индекси i и j, са записани правата на i-тия потребител към j-тия файл.



Фиг. 29. Матрица на достъпа

Недостатък на този подход е, че матрицата на достъпа ще е много голяма и разредена. Тези проблеми може да се разрешат като:

- Правата на достъп се свържат с обекта, т.е. станат част от атрибутите на файла.
- Субектите се класифицират и права на достъп към определен обект се определят не за всеки субект поотделно, а за класовете субекти.

Такъв подход е избран при реализацията на защитата на файловете в UNIX, MINIX и LINUX системите. Потребителите, регистрирани в определена система, са групирани в определени от администратора потребителски групи. Всеки потребител в определен момент принадлежи на една група. Когато се създава файл (от произволен тип) с него се свързва потребител - собственик на файла и група. Това са потребителят, на когото принадлежи процеса, създаващ файла и групата му в момента. Спрямо определен файл всички потребители се класифицират в следните класове:

- администратор или привилегирован потребител (`root`)
- собственик - потребител, който е собственик на файла;
- група - потребители, които не са собственик на файла, но принадлежат на групата на собственика (групата, свързана с файла);
- други - потребители, които не са в първите два класа.

За всеки клас без първия се определят типовете разрешен достъп до файла и това представляват ***правата на достъп*** до файла или както още се нарича ***код на защита*** на файла и той се съхранява в индексния описател на файла като 12 битов низ. Администраторът има неограничен достъп до цялата файлова система.

Кодът на защита на файл се задава при създаване на файла като аргумент в системния примитив `creat`, `open`, `mkdir` или `mknod`. Правата на достъп на процес се проверяват при отваряне на файл, създаване на нов файл или унищожаване на файл, т.е. при всеки системен примитив, в който се задава име на файл. За да се разреши изпълнението на съответната операция над файла значение имат:

- категорията, към която принадлежи потребителя, собственик на процеса;
- само правата на достъп, установени за съответната категория.

И така проверката, която изпълнява ядрото следва следните четири стъпки.

1. Ако процесът е с правата на `root`, достъпа се разрешава.

2. Ако процесът е с правата на собственика на файла, то:
   - ако в кода на защита на файла бита за съответния тип достъп за собственика е вдигнат, достъпа се разрешава.
   - иначе достъпа не се разрешава.
3. Ако процесът е с правата на групата на файла, то:
   - ако в кода на защита на файла бита за съответния тип достъп за групата е вдигнат, достъпа се разрешава.
   - иначе достъпа не се разрешава.
4. Ако в кода на защита на файла бита за съответния тип достъп за другите е вдигнат, достъпа се разрешава, иначе не се разрешава.

Ще разгледаме системните примитиви, чрез които се изменя кода на защита или собствеността на файла след създаването му.

`int chmod(const char *name, mode_t mode);`

`int fchmod(int fd, mode_t mode);`

Примитивът `chmod` изменя кода на защита на файла *`name`* според указаното в аргумента *`mode`*. Битовете в *`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
`00200` писане за собственика
`00100` изпълнение за собственика
`00040` четене за групата
`00020` писане за групата
`00010` изпълнение за групата
`00004` четене за другите
`00002` писане за другите
`00001` изпълнение за другите

Какво означават трите типа достъп за различните типове файлове.
За обикновен файл правата означават следното:

- 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 с `exec`.

За каталог правата означават следното:

- 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, например `cat dir/text` или `cd dir`.

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 `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.

Например, за да изпълним:

```
open("/home/ivan/file1", O_RDWR);
```

трябва да имаме `x` за `/`, `x` за `/home`, `x` за `/home/ivan`, `r` и `w` за `file1`. За да създадем нов каталог `dir1` с примитива:

```
mkdir("/home/ivan/dir1", 0755);
```

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

Чрез Sticky bit (показва се като t при 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 процес може да унищожи файл ако има право w за каталога и е едно от трите:

- собственик на файла
- собсвеник на каталога
- принадлежи на привилегирования потребител (root)

Пример за каталог с вдигнат Sticky bit е /tmp.

Процесът, който изпълнява chmod трябва да принадлежи на собственика на файла или на привилегирования потребител (root) и трябва да има права x за всички каталози в пълното име *name*. При успех връща 0, а при грешка -1 и кода на защита не се променя.

Примитивът fchmod има същото действие, но файлът се идентифицира чрез файлов дескриптор *fd*.

int chown(const char *_name_, uid_t _owner_, gid_t _group_);

int fchown(int _fd_, uid_t _owner_, gid_t _group_);

Примитивът chown изменя собственика и групата на файла *name* според указаното в аргументите *owner* и *group*. Ако значението на аргумента *owner* или *group* е -1 то не се изменя съответно собственика или групата. За да се измени собственика на файл, процесът трябва да принадлежи на привилегирования потребител (root), а за промяна на групата на файл процесът трябва да принадлежи на собственика на файла или на привилегирования потребител (root). Освен това процесът трябва да има права x за всички каталози в пълното име *name*. При успех връща 0, а при грешка -1 и собствеността не се променя.

Примитивът fchown има същото действие, но файлът се идентифицира чрез файлов дескриптор *fd* .

int umask(mode_t _cmask_);

Чрез umask се зарежда маската за създаване на файл като се използва аргумента *cmask*&0777. Тази маска влияе на примитивите, изпълнявани от процеса за създаване файлове - creat, open, mkdir и mknod като модифицира кода на защита на създаваните файлове. Действителният код на защита при създаване е: *mode&~cmask*, т.е. от значение са младшите 9 бита на *cmask* и битовете, които са 1 в *cmask* стават 0 в действителния код на защита, независимо от значението им в аргумента *mode* на системния примитив creat, open, mkdir или mknod. Системният примитив връща старото значение на маската, което може да бъде съхранено и по-късно възстановено. Така този примитив дава възможност на потребителите да ограничават по премълчаване достъпа до своите файлове, но само по време на създаването им. Маската се наследява при пораждане на процеси.