

Файлови системи и ОС

Съдържание

ОСНОВИ на ФС

Файлове , структура;

Типове файлове и операции с тях

Структура на ФС

Разполагане на файловете в/у носител

Методи за достъп

Справочна информация за файловете

Управление на свободното пространство

Управление на достъпа – защита

Виртуална ФС

Работа с файлове в програмна среда

Съдържание

ОСНОВИ на ФС

Файлове , структура;

Типове файлове и операции с тях

Структура на ФС

Разполагане на файловете в/у носител

Методи за достъп

Справочна информация за файловете

Управление на свободното пространство

Управление на достъпа – защита

Виртуална ФС

Работа с файлове в програмна среда

Файлове

- Long-term existence
- Sharable between processes
- Structure

Понятия

- Field
- Record
- File
- Database

Основни операции

- Create
- Delete
- Open
- Close
- Read
- Write

От гледна точка на **операционната система**, целият твърд диск представлява съвкупност от клъстери с размер от 512 байта и повече. Драйверите на файловата система организират клъстерите във файлове и директории (които реално са също файлове, съдържащи списъци с файлове). Драйверите следят също така кои от клъстерите се използват в момента, кои са свободни и кои са отбелязани като повредени.

Android

YAFFS и YAFFS2

VFAT

F2FS

EXT2-EXT4

UBIFS

iOS 10.3 +

Apple File System (APFS)

Linux

Ext4

Btrfs

JFS

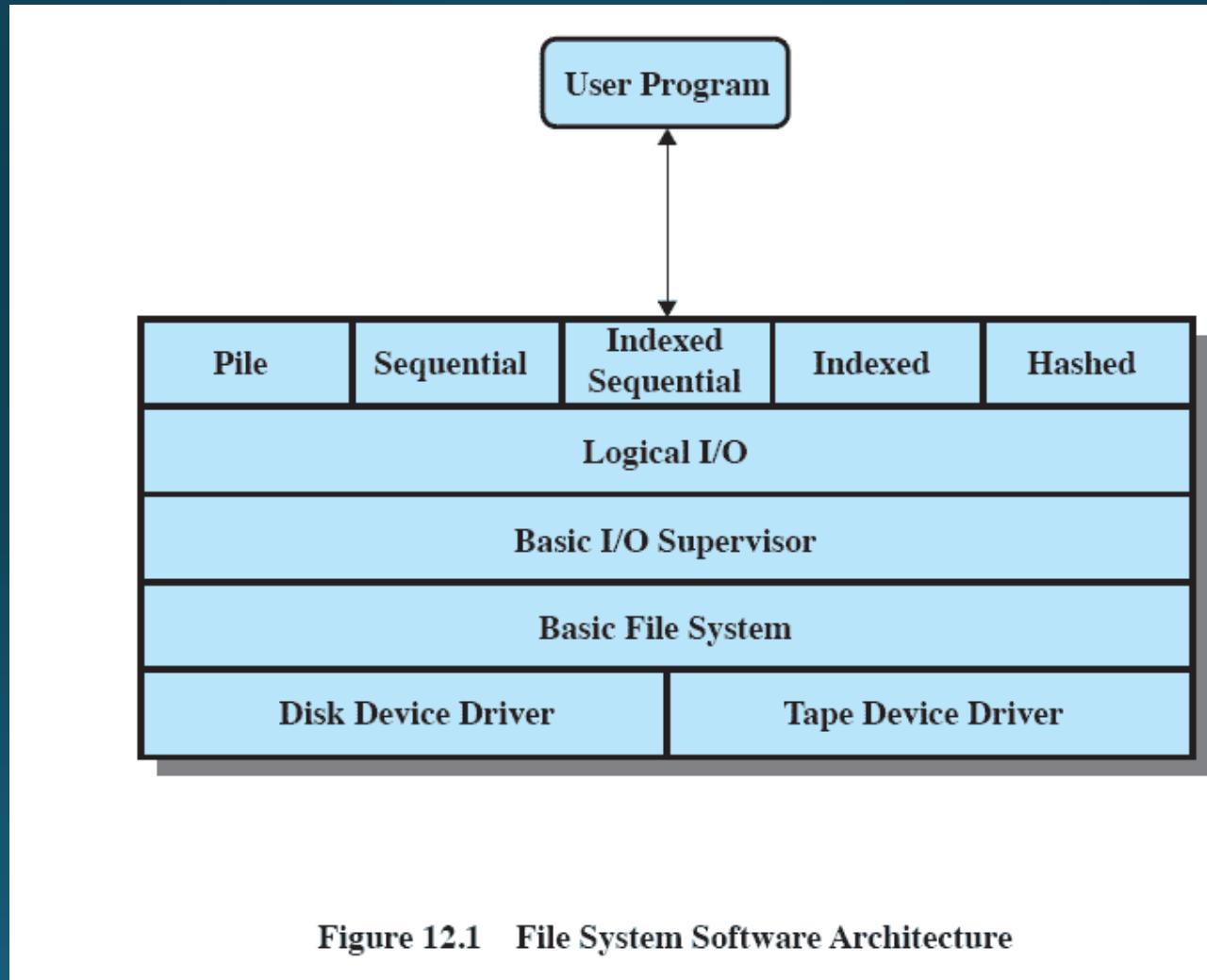
ReiserFS

ZFS

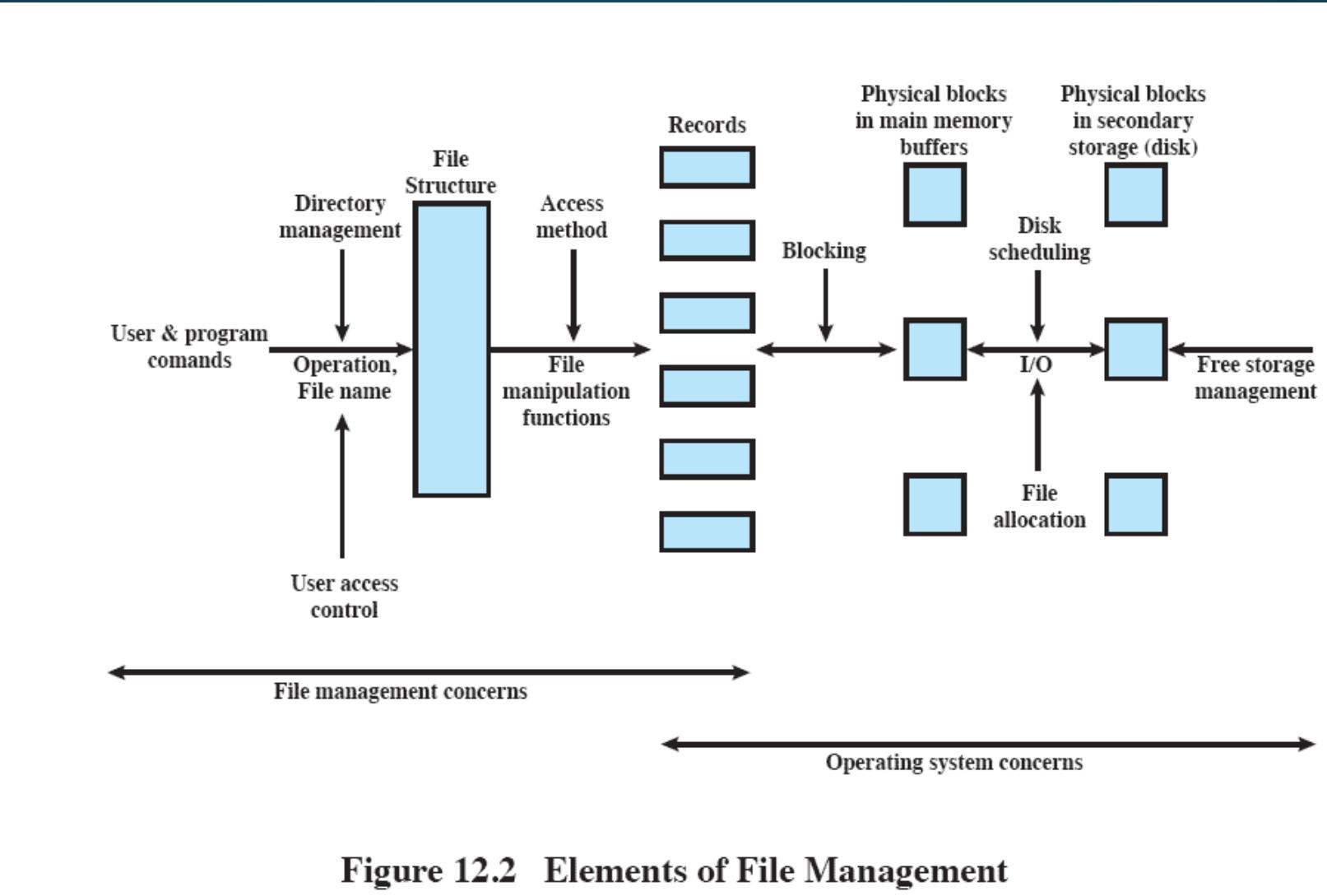
FAT32

exFAT

Типична архітектура



Елементи на управлението

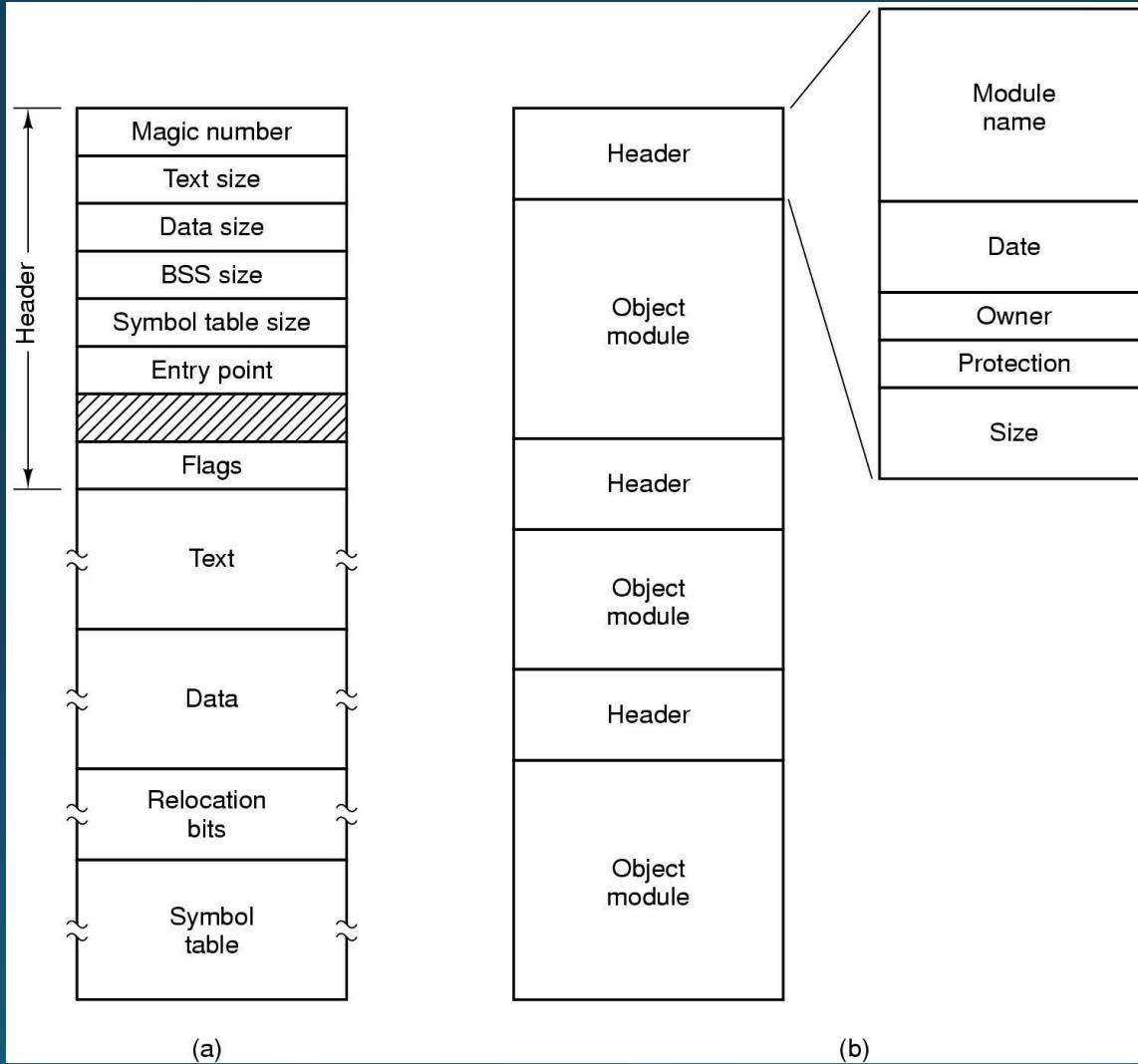


Файлови разширения

| Extension | Meaning |
|------------------|---|
| file.bak | Backup file |
| file.c | C source program |
| file.gif | Compuserve Graphical Interchange Format image |
| file.hlp | Help file |
| file.html | World Wide Web HyperText Markup Language document |
| file.jpg | Still picture encoded with the JPEG standard |
| file.mp3 | Music encoded in MPEG layer 3 audio format |
| file.mpg | Movie encoded with the MPEG standard |
| file.o | Object file (compiler output, not yet linked) |
| file.pdf | Portable Document Format file |
| file.ps | PostScript file |
| file.tex | Input for the TEX formatting program |
| file.txt | General text file |
| file.zip | Compressed archive |

Тип файл

Изпълним файл



Достъп до файл

- Последователен
 - Read all bytes/records from the beginning
 - Cannot jump around
 - May rewind or back up, however
 - Convenient when medium was magnetic tape
 - Often useful when whole file is needed
- Случаен

Файлови атрибути

| Attribute | Meaning |
|---------------------|---|
| Protection | Who can access the file and in what way |
| Password | Password needed to access the file |
| Creator | ID of the person who created the file |
| Owner | Current owner |
| Read-only flag | 0 for read/write; 1 for read only |
| Hidden flag | 0 for normal; 1 for do not display in listings |
| System flag | 0 for normal files; 1 for system file |
| Archive flag | 0 for has been backed up; 1 for needs to be backed up |
| ASCII/binary flag | 0 for ASCII file; 1 for binary file |
| Random access flag | 0 for sequential access only; 1 for random access |
| Temporary flag | 0 for normal; 1 for delete file on process exit |
| Lock flags | 0 for unlocked; nonzero for locked |
| Record length | Number of bytes in a record |
| Key position | Offset of the key within each record |
| Key length | Number of bytes in the key field |
| Creation time | Date and time the file was created |
| Time of last access | Date and time the file was last accessed |
| Time of last change | Date and time the file has last changed |
| Current size | Number of bytes in the file |
| Maximum size | Number of bytes the file may grow to |

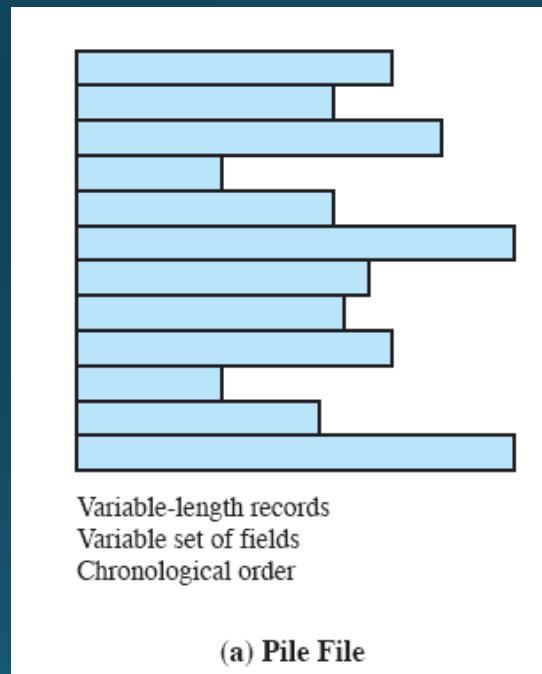
Критерии за файлова организация

- Important criteria include:
 - Short access time
 - Ease of update
 - Economy of storage
 - Simple maintenance

Типове организација на файлове

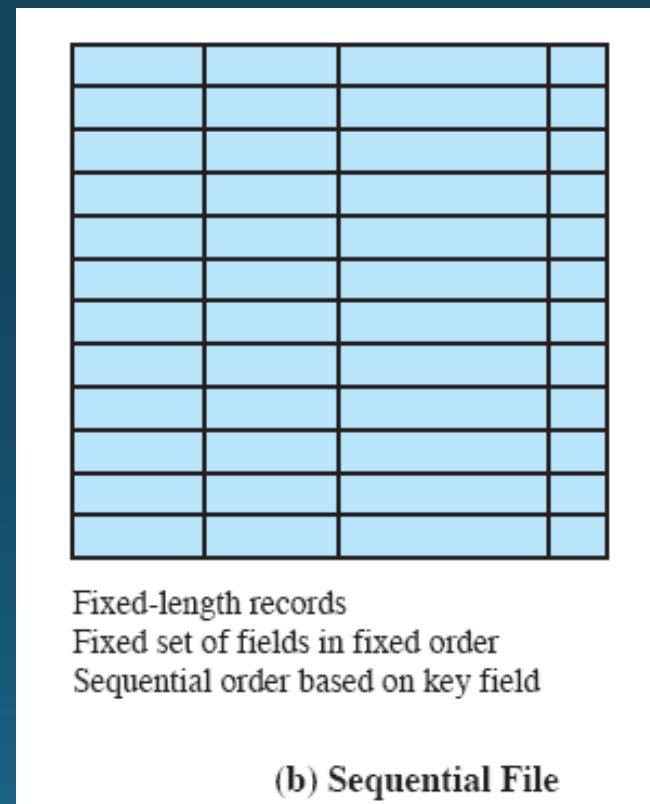
- Many exist, but usually variations of:
 - Pile
 - Sequential file
 - Indexed sequential file
 - Indexed file
 - Direct, or hashed, file

“Pile” (стълб)

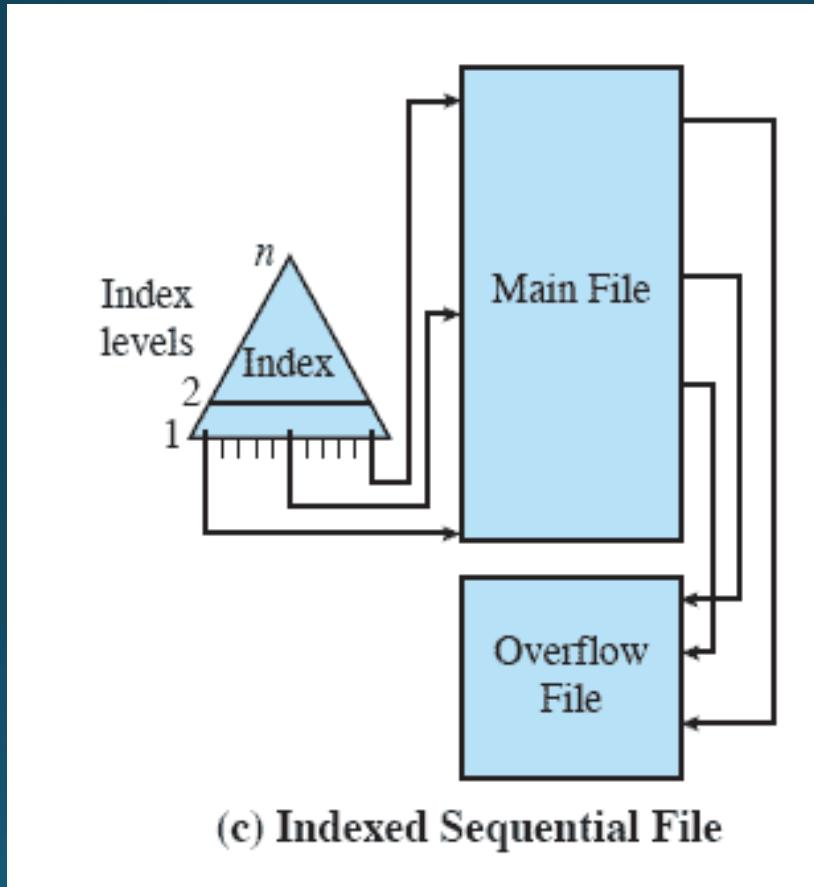


Последователни файлове

- Fixed format

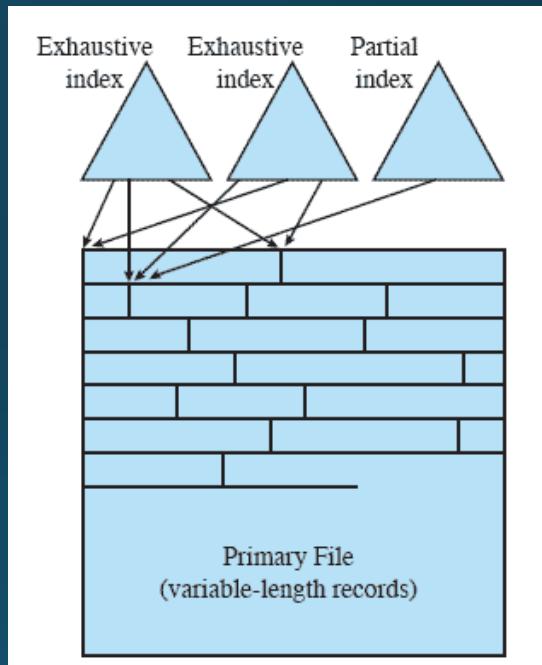


Индексни последователни файлове



Индексирани файлове

- Uses multiple indexes for different key fields
 - May contain an exhaustive index that contains one entry for every record in the main file
 - May contain a partial index
- When a new record is added to the main file, all of the index files must be updated.



(d) Indexed File

Производительность

Table 12.1 Grades of Performance for Five Basic File Organizations [WIED87]

| File Method | Space Attributes | | Update Record Size | | Retrieval | | |
|--------------------|------------------|-------|--------------------|---------|---------------|--------|------------|
| | Variable | Fixed | Equal | Greater | Single record | Subset | Exhaustive |
| Pile | A | B | A | E | E | D | B |
| Sequential | F | A | D | F | F | D | A |
| Indexed sequential | F | B | B | D | B | D | B |
| Indexed | B | C | C | C | A | B | D |
| Hashed | F | B | B | F | B | F | E |

- A = Excellent, well suited to this purpose $\approx O(r)$
B = Good $\approx O(o \times r)$
C = Adequate $\approx O(r \log n)$
D = Requires some extra effort $\approx O(n)$
E = Possible with extreme effort $\approx O(r \times n)$
F = Not reasonable for this purpose $\approx O(n^{>1})$

where

r = size of the result

o = number of records that overflow

n = number of records in file

Основни елементи на директориите

Елементи на директориите Информация за контрола на достъп

- Owner
- Access Information
- Permitted Actions

Елементи на директориите Адресна информация

- Volume
- Starting Address
- Size Used
- Size Allocated

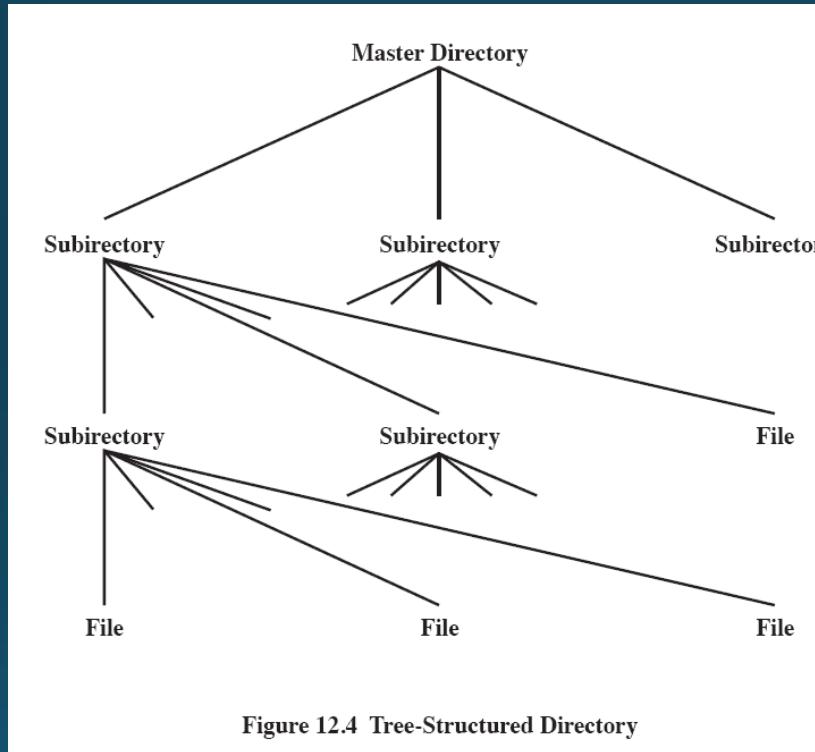
Елементи на директориите използвана информация

- Date Created
- Identity of Creator
- Date Last Read Access
- Identity of Last Reader
- Date Last Modified
- Identity of Last Modifier
- Date of Last Backup
- Current Usage
 - Current activity, locks, etc

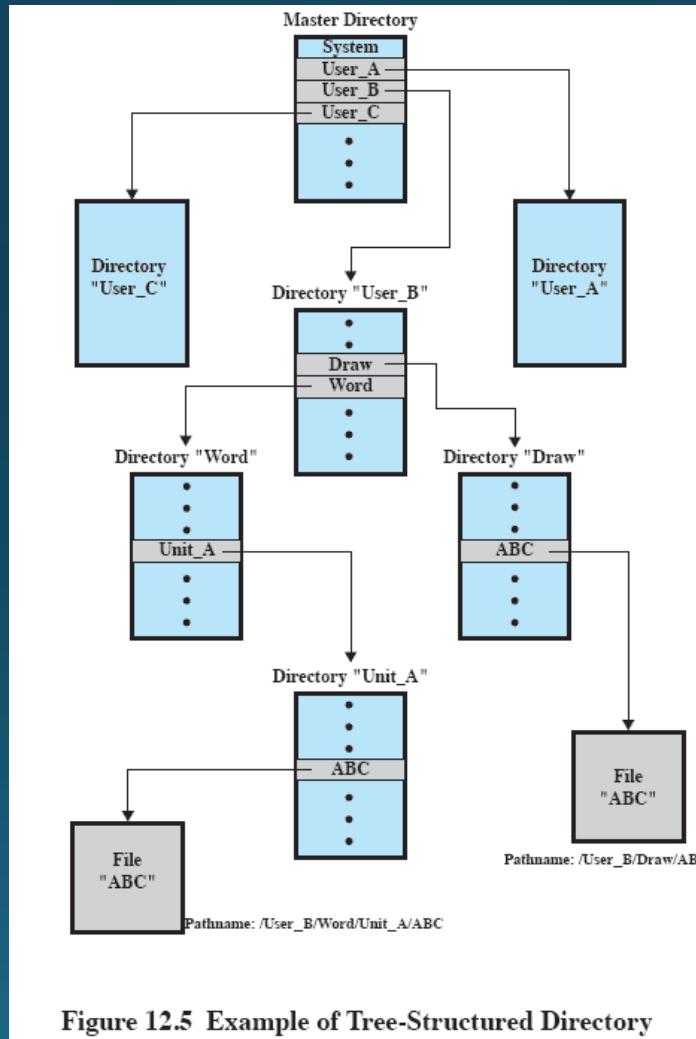
Операции извършвани от директориите

- Search
- Create files
- Deleting files
- Listing directory
- Updating directory

Йерархична дървовидна структура



Примерна структура



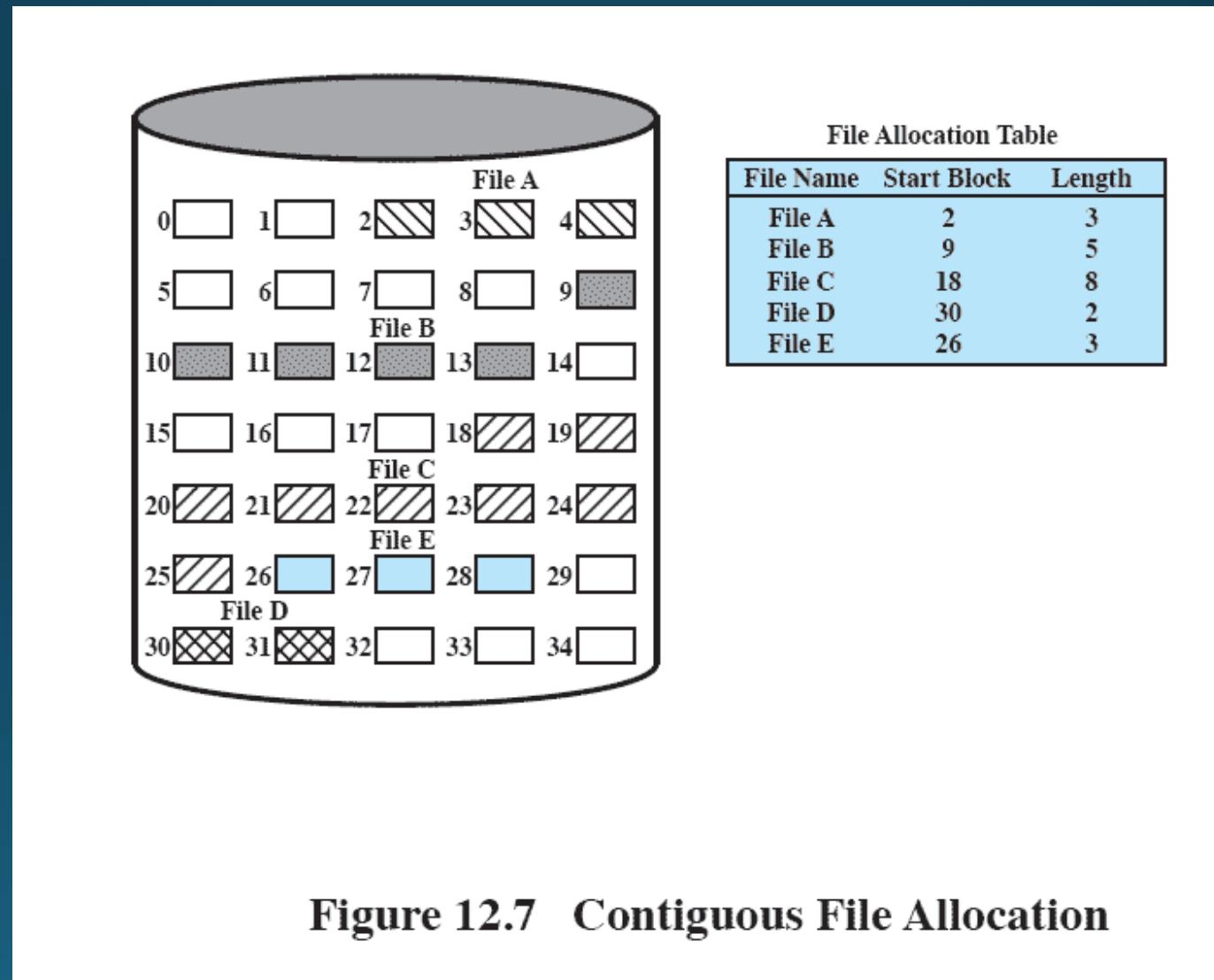
Права на доступ...

- Execution
- Reading
- Appending
- Updating
- Changing protection
- Deletion

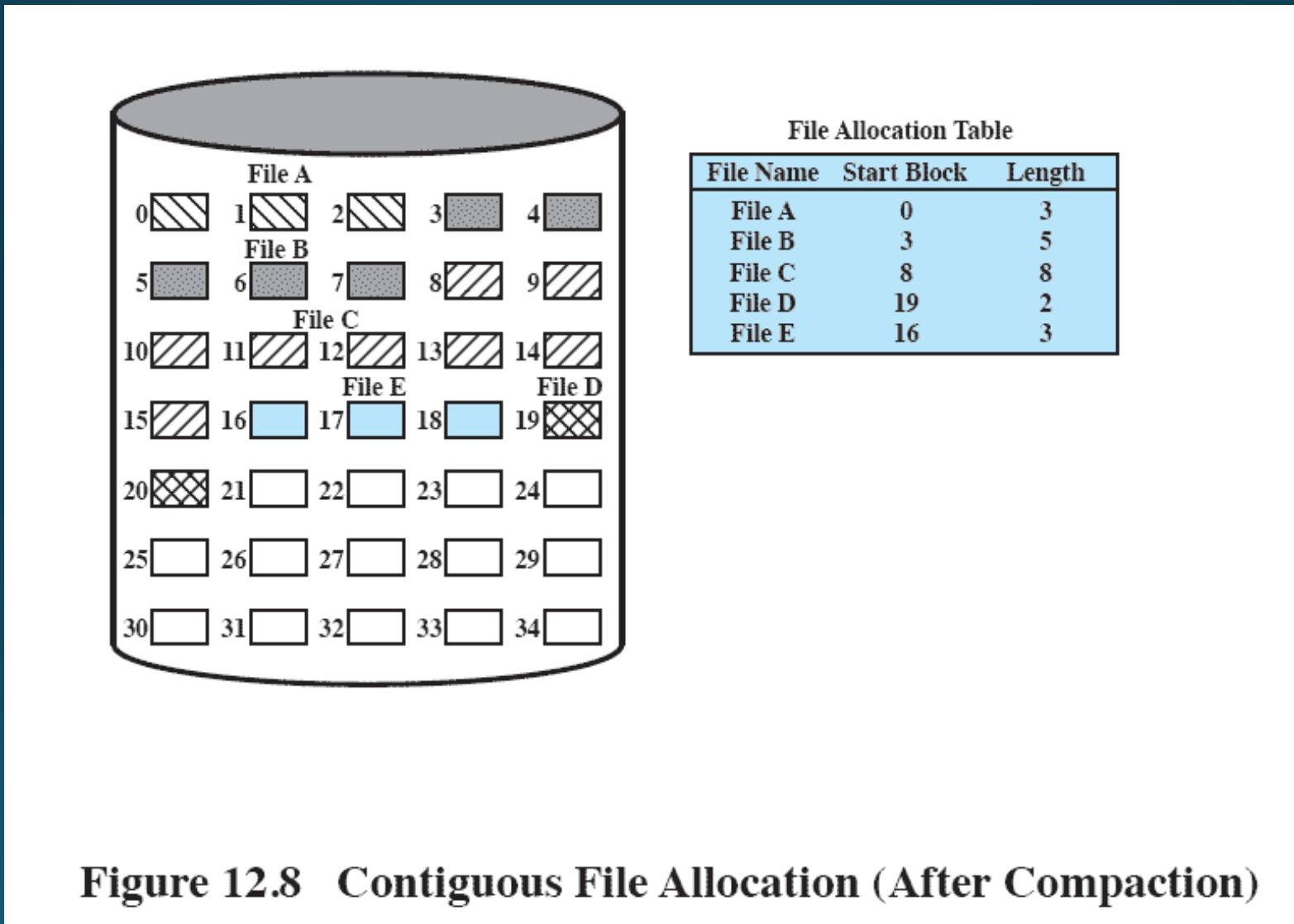
Потребителски класове

- Owner
- Specific Users
- User Groups
- All

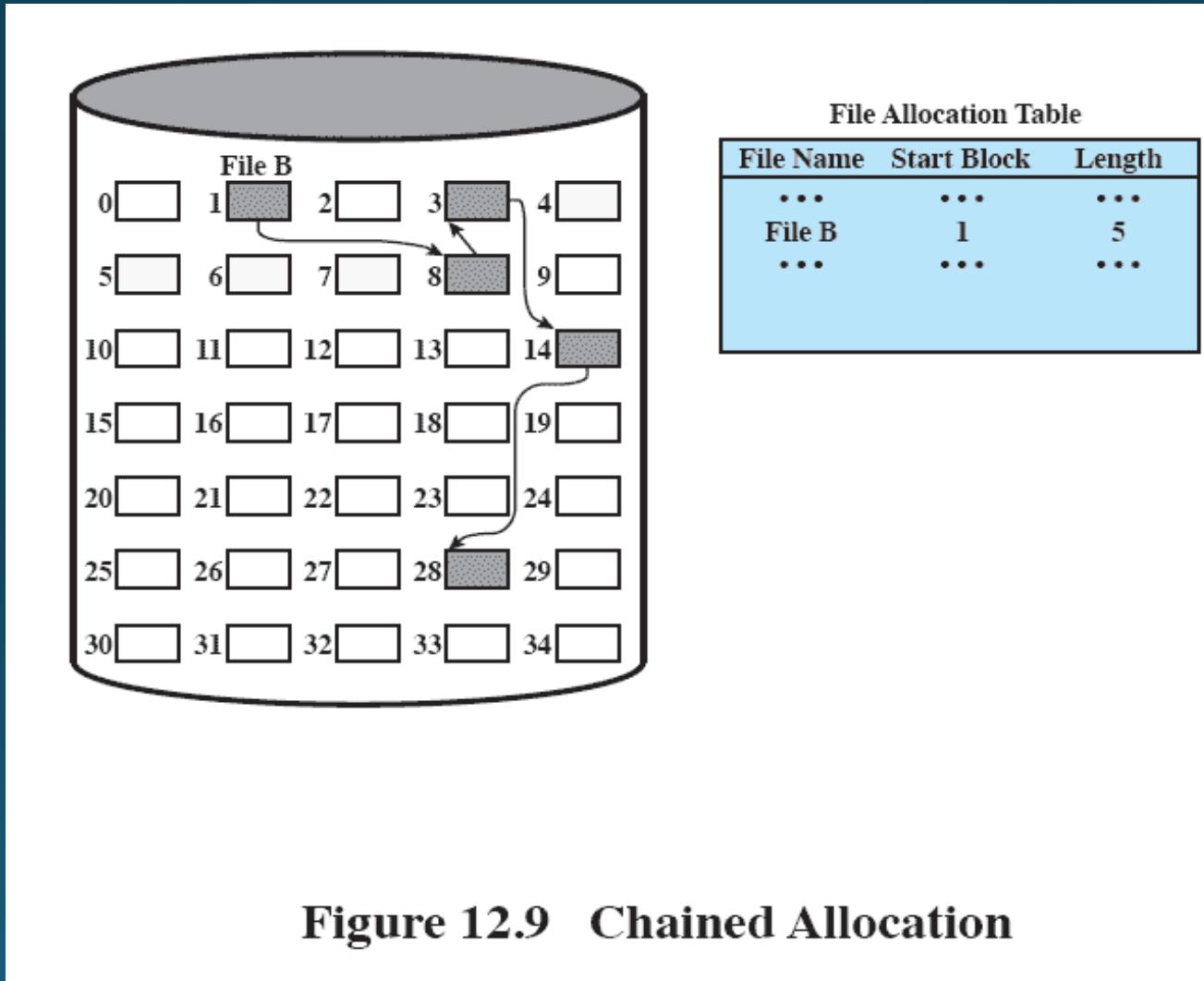
Разпределение - FAT



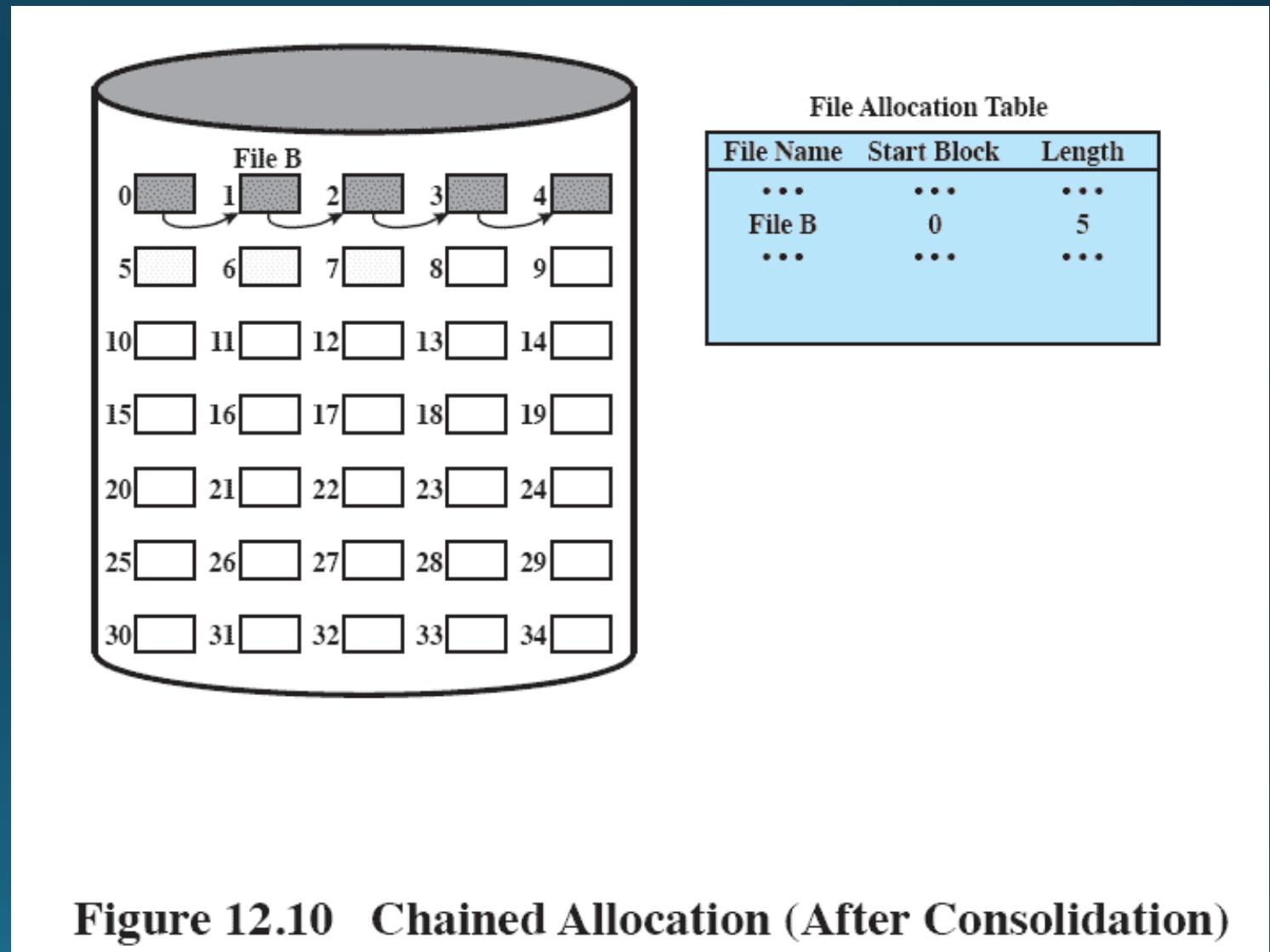
Външна фрагментация



Верижно разпределение



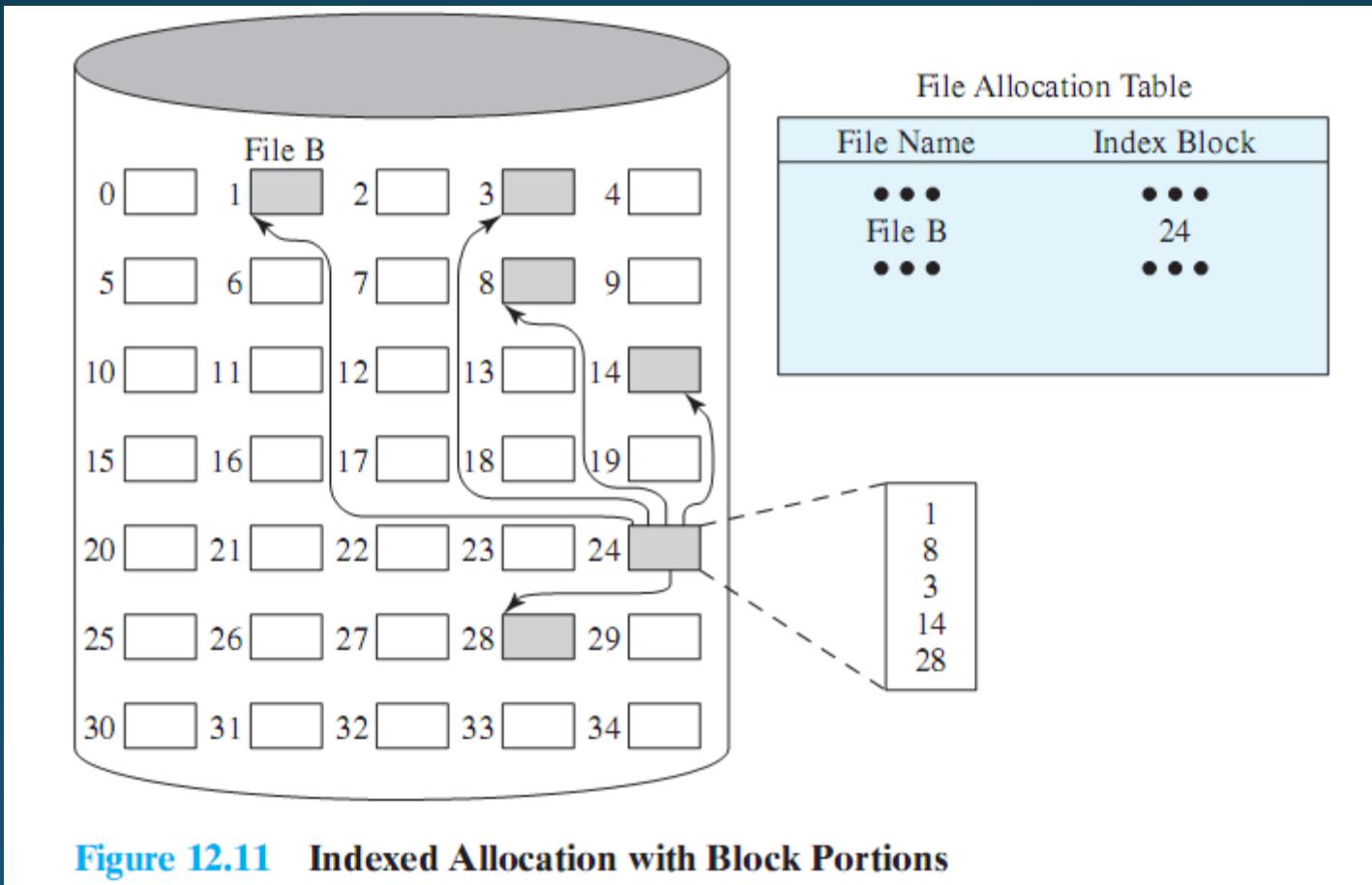
Подредено верижно разпределение



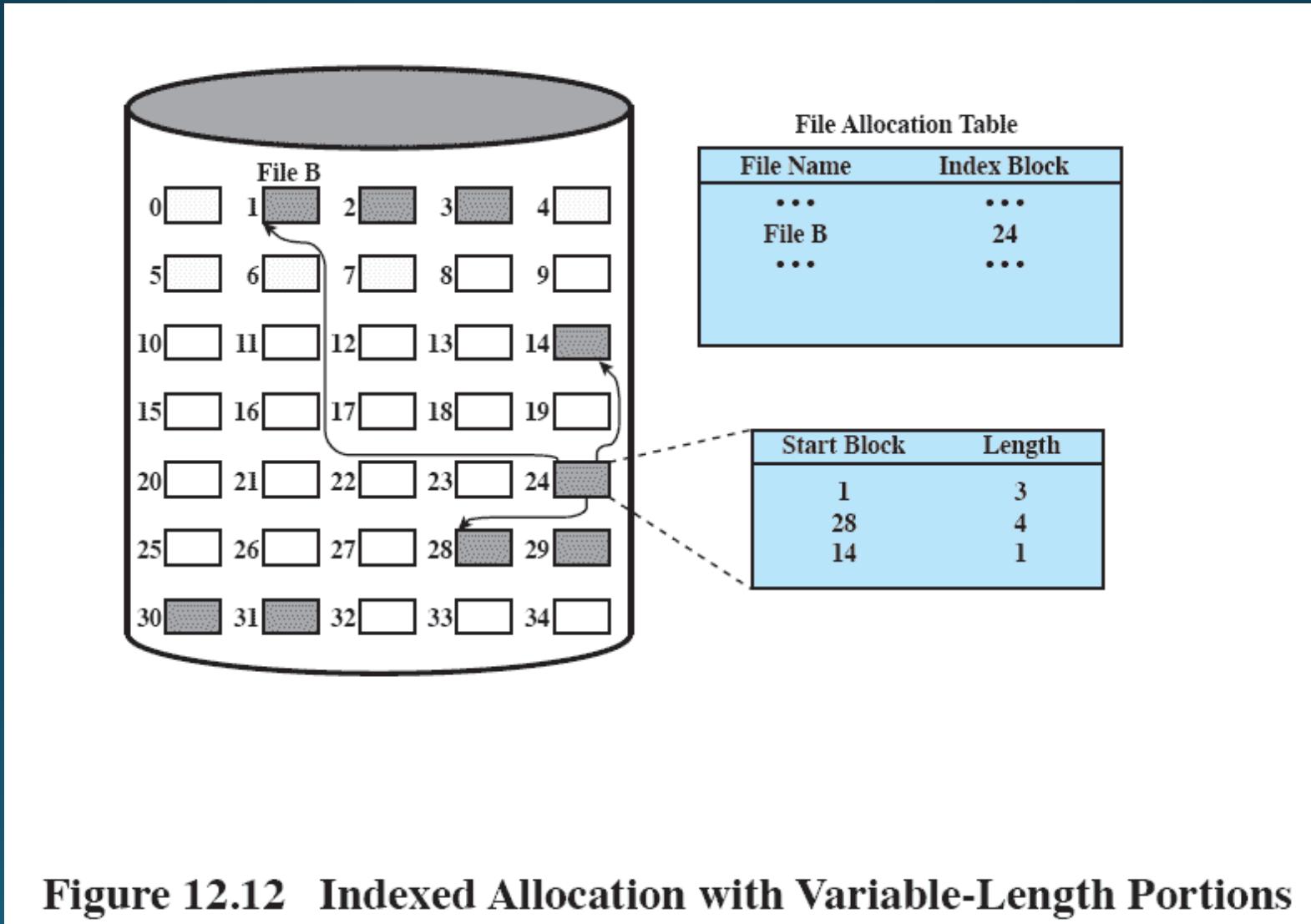
Методи за индексирано разпределение

- Fixed size blocks or
- Variable sized blocks

Индексировано разпределение в блок част



Indexed Allocation с различна дължина на



Управление на свободното пространство

Bit Таблици – таблици отбелязващи с **О ИЛИ 1** свободно/заето пространство

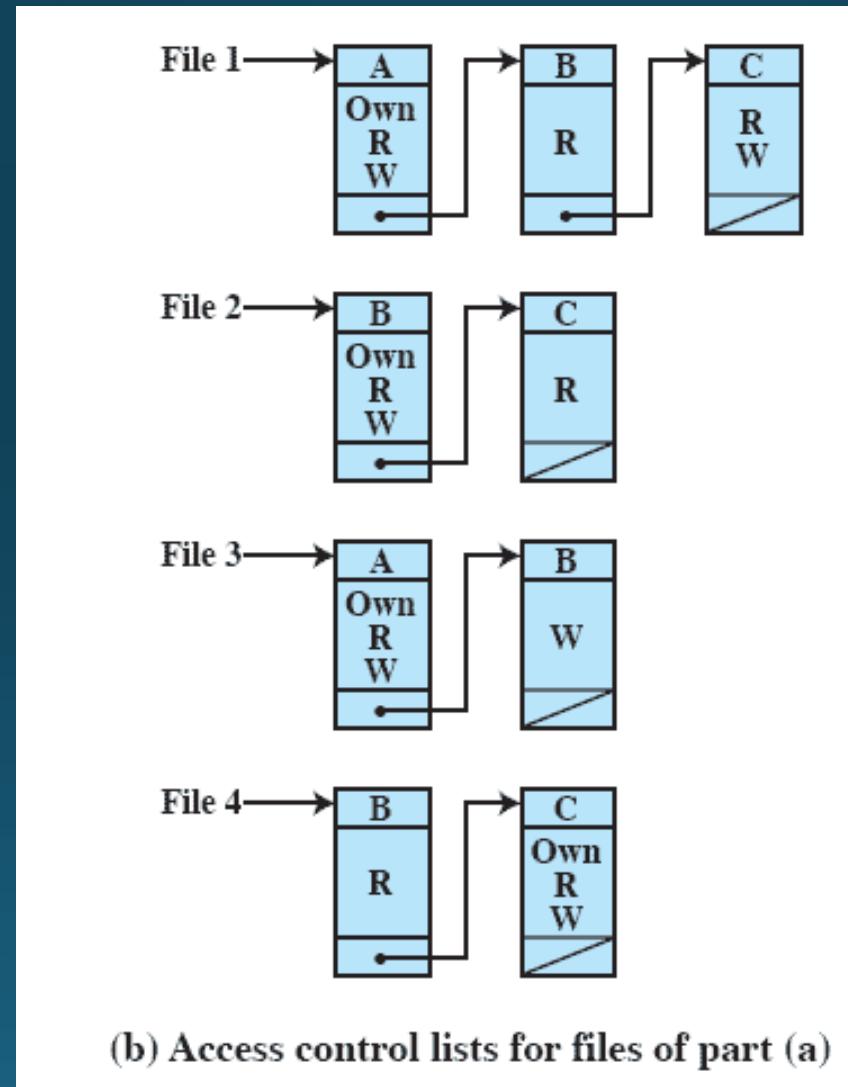
Access Matrix

- One such rule set is an Access Matrix

| | File 1 | File 2 | File 3 | File 4 | Account 1 | Account 2 |
|--------|---------------|---------------|---------------|---------------|-------------------|-------------------|
| User A | Own R W | | Own R W | | Inquiry Credit | |
| User B | R | Own R W | W | R | Inquiry Debit | Inquiry Credit |
| User C | R W | R | | Own R W | | Inquiry Debit |

(a) Access matrix

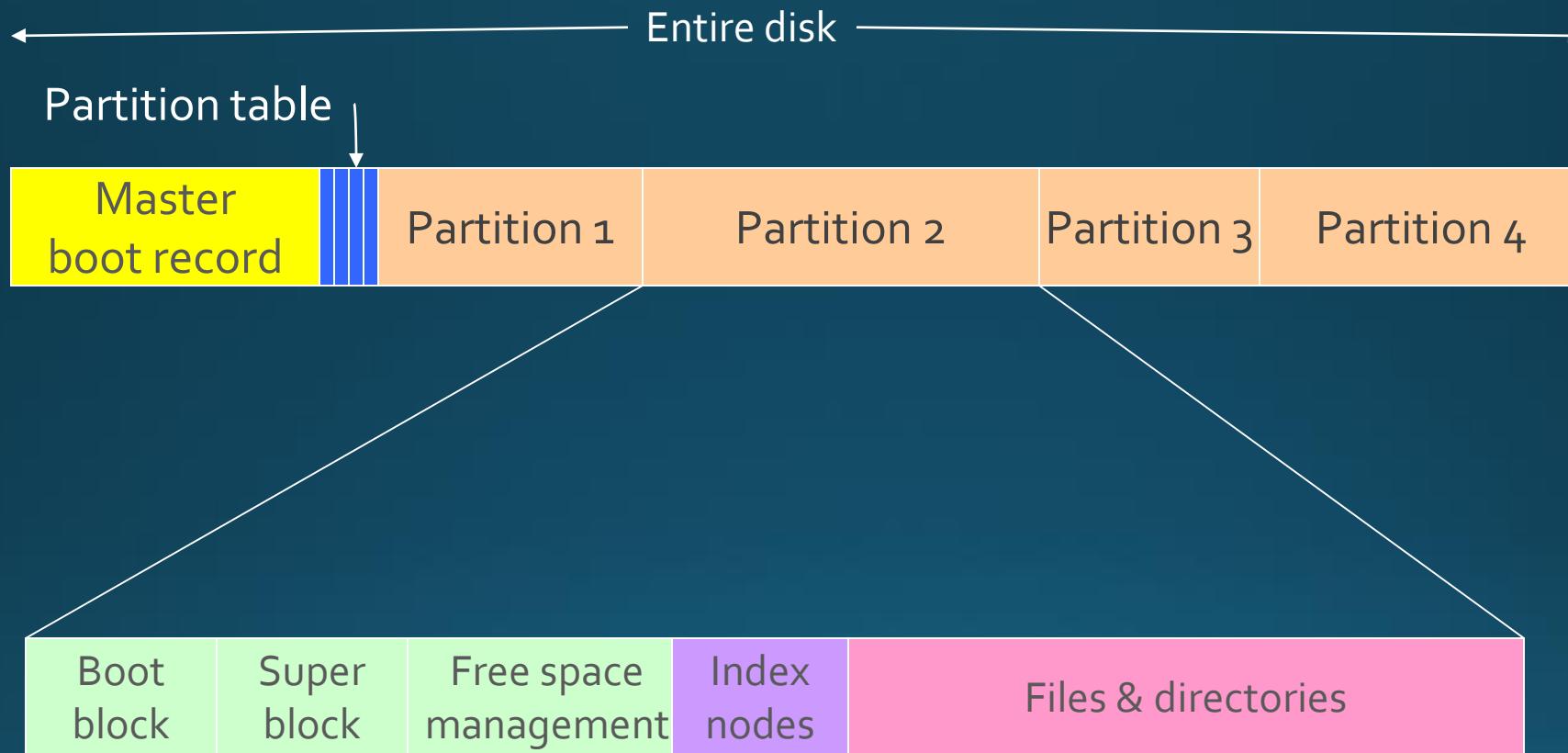
Access Control Lists



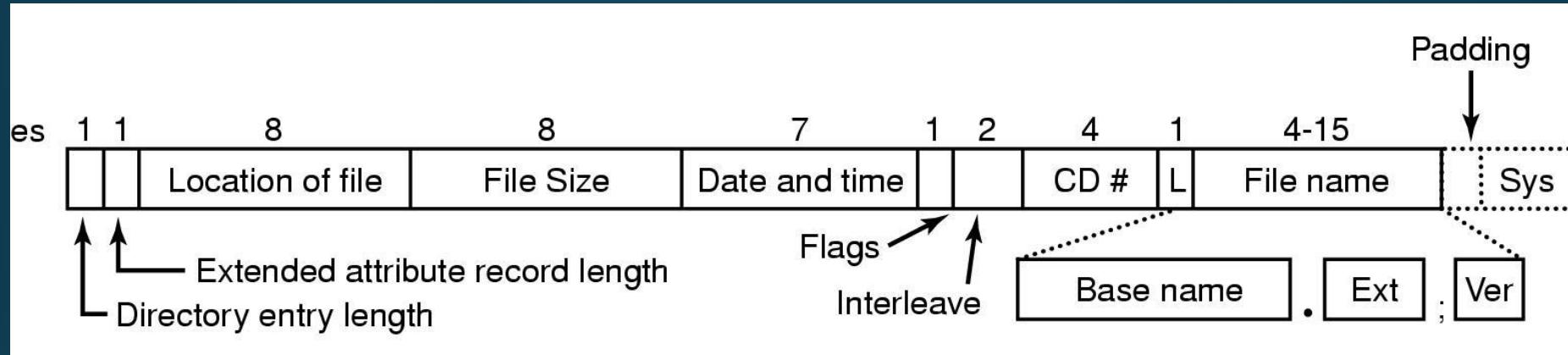
Operations on directories

- Create: make a new directory
- Delete: remove a directory (usually must be empty)
- Opendir: open a directory to allow searching it
- Closedir: close a directory (done searching)
- Readdir: read a directory entry
- Rename: change the name of a directory
 - Similar to renaming a file
- Link: create a new entry in a directory to link to an existing file
- Unlink: remove an entry in a directory
 - Remove the file if this is the last link to this file

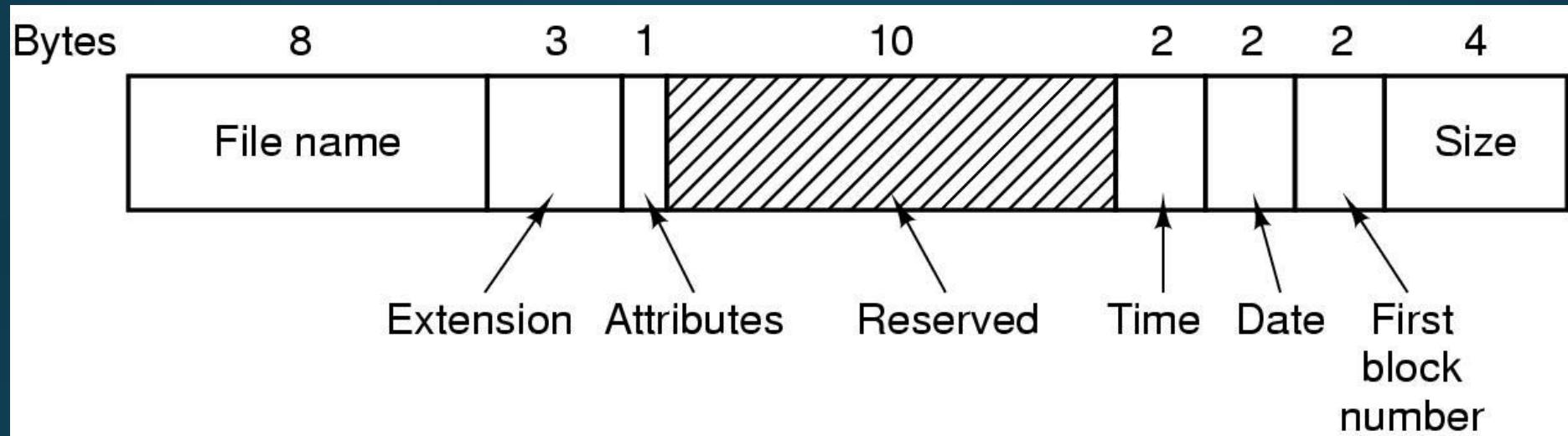
Подялба на диска



CD-ROM file system



Directory entry in MS-DOS



MS-DOS File Allocation Table

| Block size | FAT-12 | FAT-16 | FAT-32 |
|-------------------|---------------|---------------|---------------|
| 0.5 KB | 2 MB | | |
| 1 KB | 4 MB | | |
| 2 KB | 8 MB | 128 MB | |
| 4 KB | 16 MB | 256 MB | 1 TB |
| 8 KB | | 512 MB | 2 TB |
| 16 KB | | 1024 MB | 2 TB |
| 32 KB | | 2048 MB | 2 TB |

Inodes

- Index node

Структура от данни в Unix подобни на файлови системи, която съдържа цялата информация за обект на файловата система, с изключение на данните на файла и името му

inode number (номер на инод): Уникален идентификатор на инода в рамките на файловата система.

2. **File type** (тип на файл): Информация за типа на файла, като обикновен файл, директория, блоково устройство и други.

3. **Permissions** (права за достъп): Правата за достъп до файла, които определят какви операции могат да бъдат извършвани върху него от различни потребители и групи.

4. **Owner** (собственик): Информация за потребителя, който е собственик на файла.

5. **Group** (група): Информация за групата, към която принадлежи файла.

6. **File size** (размер на файл): Размерът на файла в байтове.

7. **Timestamps** (времеви маркери): Информация за времето на създаване, последна модификация и последен достъп до файла.

8. **Pointers to data blocks** (указатели към блокове с данни): Указатели към блоковете с данни, които съхраняват фактическите данни на файла или указатели към други иноди, ако става въпрос за директория.

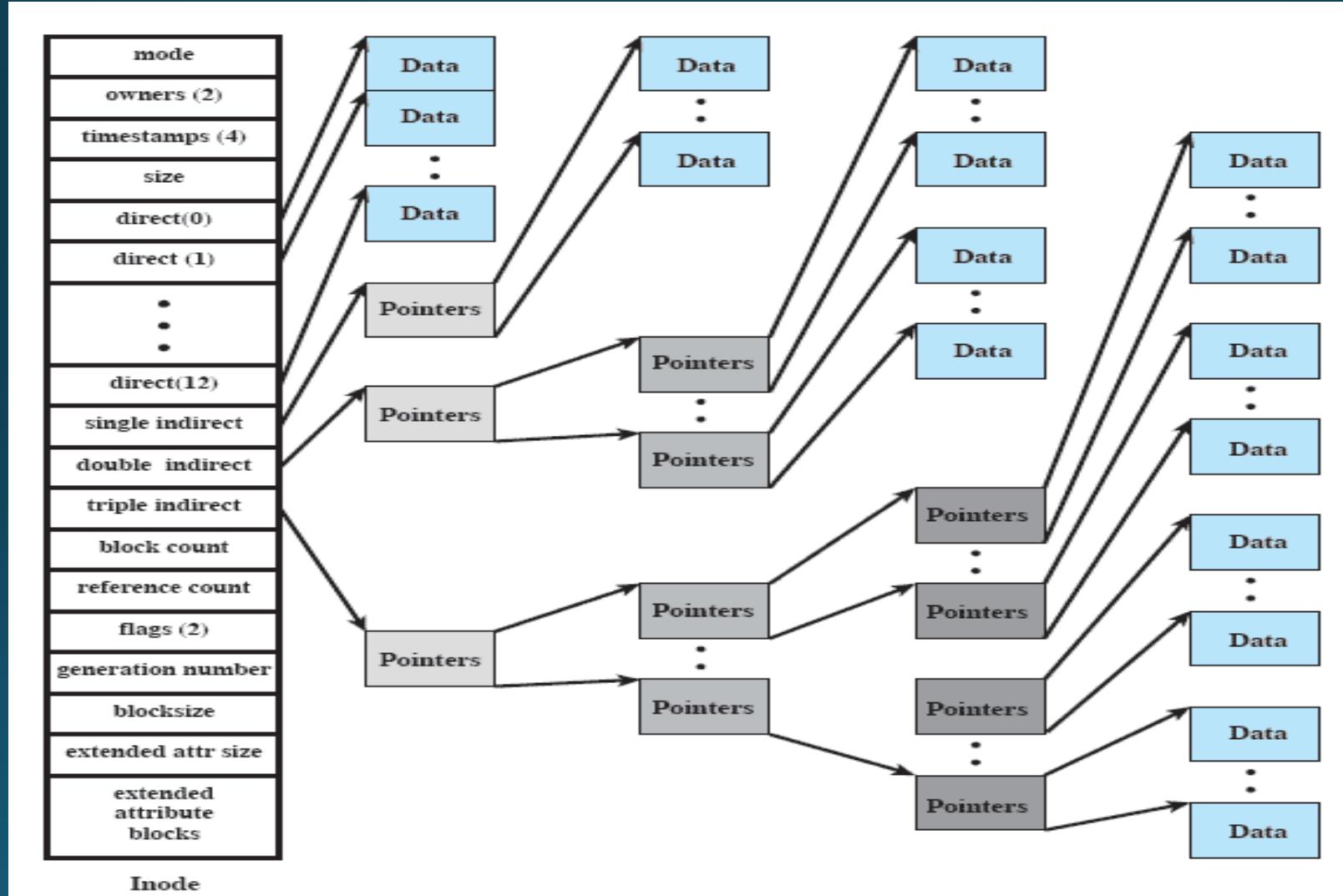
9. **Extended attributes** (разширени атрибути): Допълнителни атрибути, които могат да бъдат свързани с инода, като например защитени атрибути, криптиране и други.

Free BSD

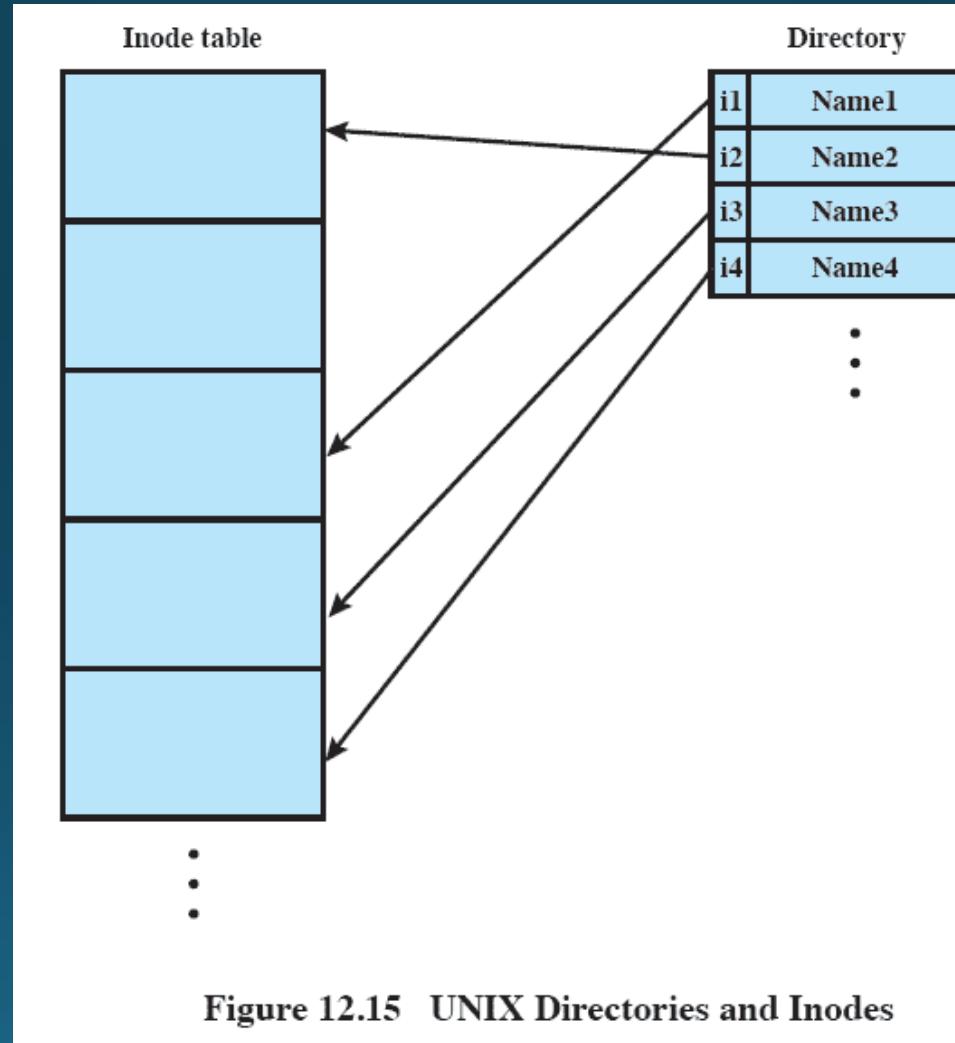
Inodes include:

- The type and access mode of the file
- The file's owner and group-access identifiers
- Creation time, last read/write time
- File size
- Sequence of block pointers
- Number of blocks and Number of directory entries
- Blocksize of the data blocks
- Kernel and user setable flags
- Generation number for the file
- Size of Extended attribute information
- Zero or more extended attribute entries

FreeBSD Inode and File Structure

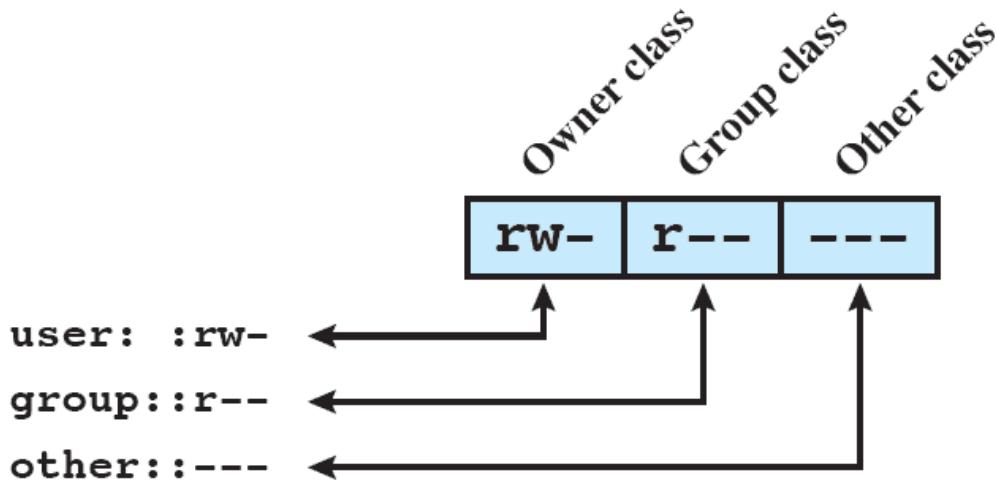


UNIX Directories and Inodes



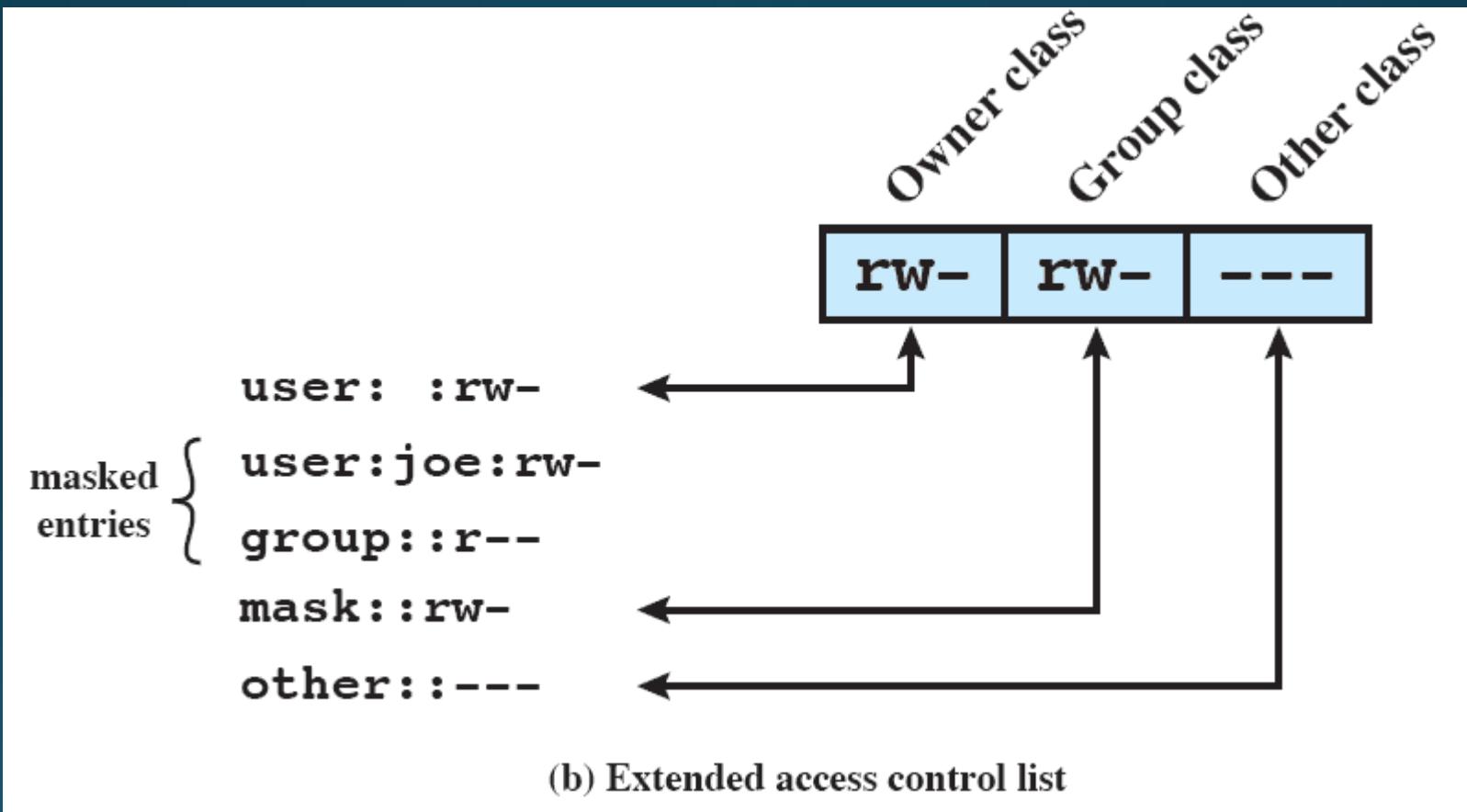
UNIX File Access Control

- User ID
- Group ID
- Everyone else



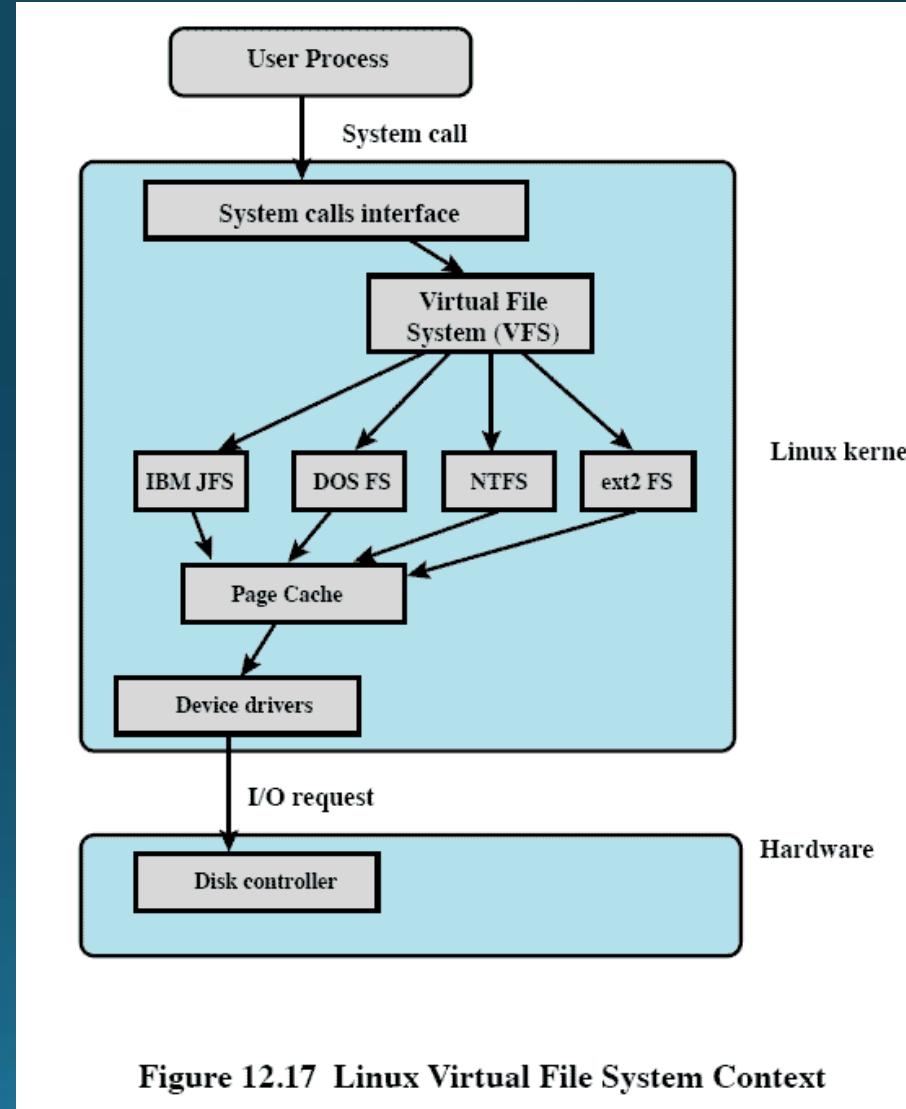
(a) Traditional UNIX approach (minimal access control list)

UNIX File Access Control



Основни VFS стратегии

Съществуват и **виртуални файлови системи**, както и мрежови файлови системи, които представляват начин за достъп до файлове на отдалечен компютър.



Файлови системи детайли

Virtual File Systems - VFS

Абстрактни слоеве на софтуер, които предоставят стандартизиран интерфейс за работа с различни файлови системи. Те позволяват на приложенията да работят с файлови системи, без да се интересуват от конкретната реализация на тези файлови системи.

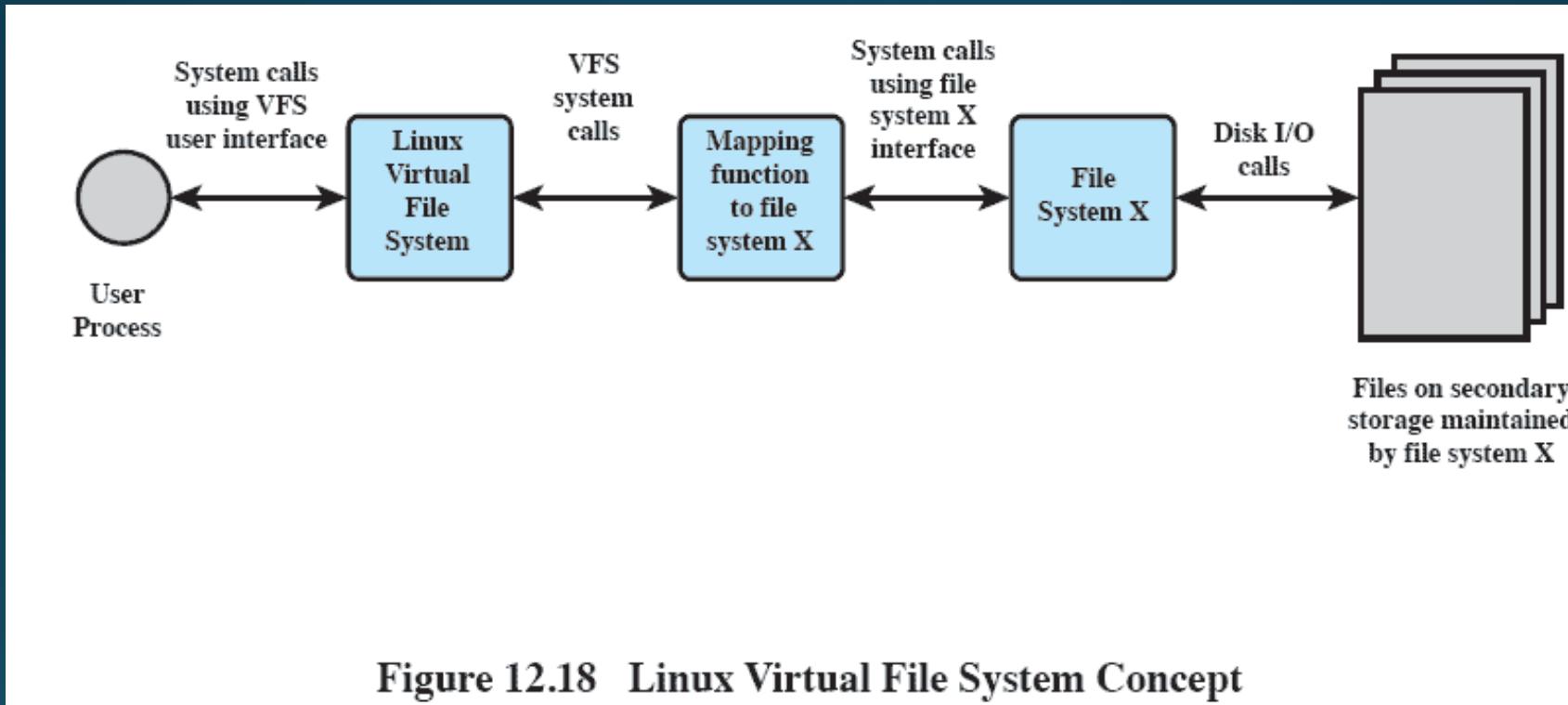
VFS предоставя общи операции за работа с файлове и директории, като отваряне, затваряне, четене, писане и преместване. Това позволява на приложенията да бъдат написани така, че да бъдат независими от конкретната файлова система, с която работят.

Предимствата на виртуалните файлови системи включват възможността за прозрачен достъп до различни типове файлови системи, като например локални, мрежови или виртуални файлови системи. Те също така позволяват на операционната система да предоставя допълнителни функционалности, като например криптиране на данни или компресия на файлове, без да се налага да променя реализацията на конкретната файлова система.

Някои примери за виртуални файлови системи включват **FUSE (Filesystem in Userspace)** за **Linux**, **Dokan** за **Windows** и **VFS** във **FreeBSD**.

Тези системи предоставят API, което позволява на разработчиците да създават собствени виртуални файлови системи да бъдат монтирани и използвани като обикновени файлови системи.

Ролята на VFS



Основни обекти на VFS

- Superblock object
- Inode object
- Dentry object
- File object

Windows File System

- Key features of NTFS
 - Recoverability
 - Security
 - Large disks and large files
 - Multiple data streams
 - Journaling
 - Compression and Encryption

| FEATURE | FAT32 | NTFS |
|------------------------|---------------------|----------------------|
| Max. Partition Size | 2TB | 2TB |
| Max. File Name | 8.3 Characters | 255 Characters |
| Max. File Size | 4GB | 16TB |
| File/Folder Encryption | No | Yes |
| Fault Tolerance | No | Auto Repair |
| Security | Only Network | Local and Network |
| Compression | No | Yes |
| Conversion | Possible | Not Allowed |
| Compatibility | Win 95/98/2K/2K3/XP | Win NT/2K/XP/Vista/7 |

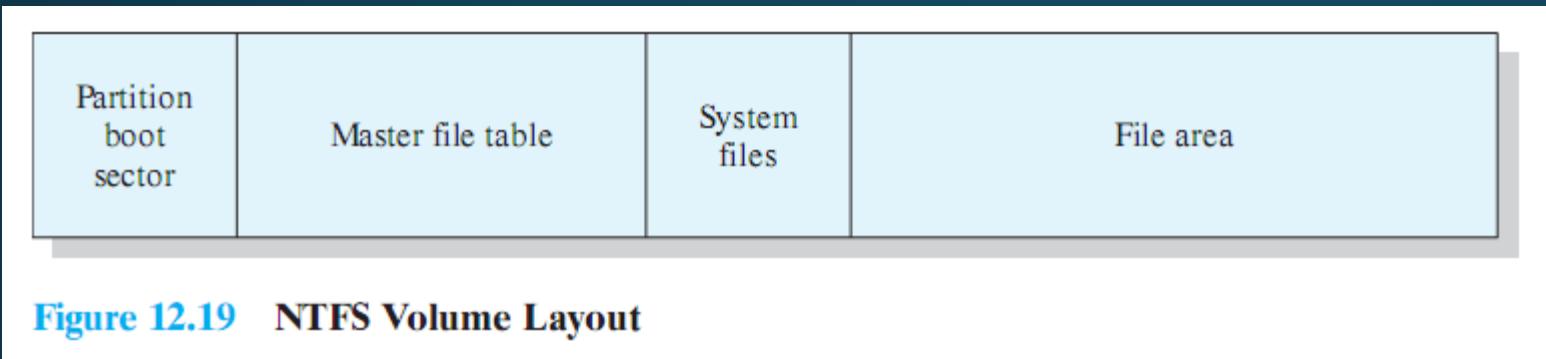
NTFS структура

- Sector
 - The smallest physical storage unit on the disk
 - Almost always 512 bytes
- Cluster
 - One or more contiguous sectors
- Volume
 - Logical partition on a disk

Ефективна с големи файлове

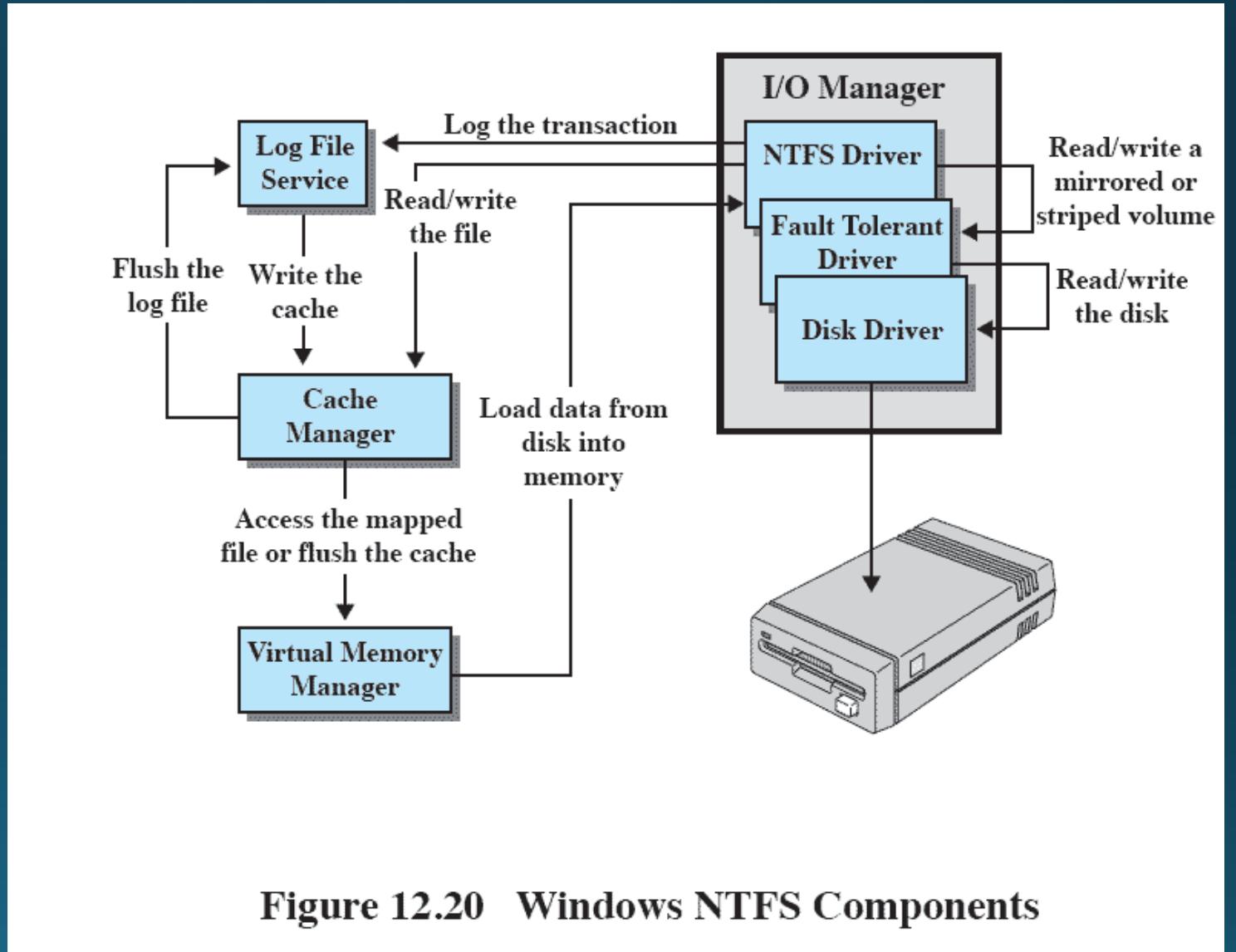
| Volume Size | Sectors per Cluster | Cluster Size |
|--------------------|---------------------|--------------|
| 512Mbyte | 1 | 512bytes |
| 512Mbyte – 1 Gbyte | 2 | 1K |
| 1–2 Gbyte | 4 | 2K |
| 2–4 Gbyte | 8 | 4K |
| 4–8 Gbyte | 16 | 8K |
| 8–16 Gbyte | 32 | 16K |
| 16–32 Gbyte | 64 | 32K |
| >32Gbyte | 128 | 64K |

NTFS разпределение на паметта



Windows NTFS

КОМПОНЕНТИ



NTFS

- ~ 12% MTF
- ~ 88% data

MTF файлове:

\$Mft

\$MftMirr

\$LogFile

\$Volume

\$AttrDef

\$Bitmap

\$Boot

\$BadClus

| Filename | Description |
|-----------|---|
| \$Boot | Contains bootstrap code and a BIOS parameter block including a volume serial number and cluster numbers of \$MFT and \$MFTMirr. |
| \$MFTMirr | Duplicate of the first vital entries of \$MFT, usually 4 entries (4 Kilobytes). |
| \$LogFile | Used to record all disk operations that affect the NTFS volume structure such as file creations, file copying, file deletion, etc. After a system |
| \$Volume | Contains information about the volume (e.g. file system version and volume label) |
| \$Bitmap | An array of bit entries - each bit indicates whether its corresponding cluster is used (allocated) or free (available for allocation). |
| \$Secure | Stores security information for each object on the file systems objects |
| \$BadClus | A file that contains all the clusters marked as having bad sectors. |

NTFS

1. **Boot Sector** (Заглавен сектор): Това е първият сектор на дял или диск, който съдържа информация за файловата система и помага на операционната система да я разпознае и зареди.
2. **Master File Table** (MFT) (Таблица на главните файлове): Това е централната таблица, която съхранява информация за всички файлове и директории в NTFS. Всеки запис в MFT представлява един файл или директория и съдържа метаданни за тях, като име, размер, дати на създаване и модификация, атрибути и т.н.
3. **File Record Segments** (Сегменти на записите за файлове): Всеки запис в MFT може да бъде разделена на няколко сегмента, ако размерът на записа надвишава размера на един MFT запис. Това позволява на NTFS да съхранява големи файлове, които надвишават размера на един MFT запис.
4. **Attribute List** (Списък с атрибути): Това е структура, която съхранява информация за всички атрибути на файловете и директориите. всяка атрибут може да съдържа данни като съдържание на файла, атрибути за сигурност, индексиране и други.
5. **Bitmaps** (Битови карти): NTFS използва битови карти, за да отбележва свободното и заето пространство на дяла или диска. Това позволява на файловата система да управлява и разпределя дисковото пространство.
6. **Log File** (Файл с протокола): Това е специален файл, който съдържа протокол на операциите, извършени върху файловата система. Той се използва за възстановяване на данни в случай на системни сбои или други проблеми.

MFT

Размер и записи: MFT е разделен на фиксирани големини записи, обикновено с размер от 1 KB. Всяка записка съхранява информацията за един файл или директория, включително метаданни като име, атрибути, указатели към данни и други.

2. **Метаданни на системни файлове:** Първите няколко записа в MFT са резервираны за системни файлове и метаданни като \$MFT (самата MFT), \$MFTMirr (резервно копие на MFT), \$LogFile (журнал на операционната система) и други.

3. **Индексиране и йерархия:** MFT използва индексирана структура, която позволява бързо намиране на файлове и директории. Записите в MFT са свързани чрез указатели, които образуват йерархия от директории и поддиректории. Това позволява ефективно търсене и навигация във файловата система.

4. **Алокация на пространство:** MFT се управлява от самата файлова система и може да се разширява или свива, в зависимост от нуждите. Когато се добавят нови файлове или директории, MFT автоматично увеличава своя размер, за да съхранява новите записи. Ако MFT е запълнена, файловата система може да използва допълнителни MFT зони на диска, за да съхранява останалите записи.

5. **Фрагментация:** При неправилно управление на файловата система или често добавяне и изтриване на файлове, MFT може да се фрагментира, като записите се разпръсват по различни части на диска. Това може да намали ефективността на достъпа до файловата система и да забави операциите за търсене и изпълнение на файлови операции.

Организацията на MFT в NTFS е проектирана да осигури бърз и ефективен достъп до файловата информация, като в същото време позволява гъвкавост и автоматично управление на пространството. Правилното поддържане на MFT е важно за оптималното функциониране на NTFS файловата система.

NTFS

- *Възстановяване* – способност на файловата система да се възстановява при проподане на ОС или грешка на диска. Използва допълнителна памет за дублиране на критичните данни.
- *Устойчивост на грешки*. Използва отказоустойчив драйвер и възможностите му за реализация на RAID 1 RAID 5 . Заменя повредените клъстери с добри и лошите ги включва в специален файл.
- *Безопасност* - Отворен файл има дескриптор на безопасност.
-
- *Големи дискове и файлове* – поддържат се ефективно.
- *Множествени потоци от данни* – свързва набор от атрибути с всеки файл (име, собственик, данни и т.н.). Всеки атрибут се съхранява като отделен поток от байтове.
- *Индексиране* – индексира атрибутите на файловете върху том, за по бързо търсене.
- *Дискова структура (структура на данните)*
- *Сектор* – обикновено 512 байта.
- *Клъстер* – един или повече съседни сектори в една писта с дължина в сектори 2^n .
- *Том* – логически раздел на диск или цял диск.

FAT32 - NTFS

Размер на максималния обем: FAT32 поддържа максимален обем на диска от около 2 терабайта и максимален размер на файла от 4 гигабайта. С друга страна, NTFS поддържа много по-големи обеми на диска и файлове – до 16 екзабайта за дисковете и до 16 терабайта за файловете.

2. **Сигурност и разрешения за достъп:** NTFS предлага по-напреднал контрол върху достъпа до файловете и разрешенията за потребителите. Можете да определите различни нива на достъп и да зададете разрешения за четене, писане, изпълнение и други за отделни потребители или групи. FAT32 няма подобни възможности за разрешения.

3. **Възстановяване на данни:** NTFS има по-добра възможност за възстановяване на данни в случай на повреда или срив на системата. Тя включва журналиране на операциите, което позволява по-бързо възстановяване след срив или нежелани прекъсвания на захранването. FAT32 няма подобни механизми за възстановяване на данни.

4. **Поддръжка на компресия и шифроване:** NTFS предлага вградена поддръжка на компресия на файлове и папки, което може да помогне за спестяване на дисково пространство. Освен това, NTFS също така поддържа шифроване на файлове и папки, което осигурява по-висока сигурност на данните. FAT32 няма тези възможности.

5. **Фрагментация:** FAT32 е по-склонна към фрагментация на файловете, което може да намали ефективността на достъпа до данните. NTFS има по-добра управление на фрагментацията и може да оптимизира разположението на файловете на диска.

Linux

- /bin
- /boot
- /dev
- /etc
- /home
- /lib
- /mnt
- /root
- /sbin
- /tmp
- /usr
- /var

Journaling

Файлова система, която използва „journaling“ има специфична зона от клъстери, наричани журнал (journal), в които се записва какво иска да свърши файловата система. Когато ние решим да запишем нашият 3 гигабайтов файл, файловата система може

- (1) да отбележи в журнала, че иска да запише файл;
- (2) да запише файла;
- (3) да отбележи в журнала факта, че файла е успешно записан;
- (4) да регистрира файла;
- (5) да изтрие журнала. Ако захранването спре, при следващото си зареждане файловата система първо ще прочете журнала и ще прецени какво да предприеме, като по този начин ще ни остави последователна файлова система

Ext2fs, Ext3fs

- Second Extendet Filesystem

- EXT фамилията от файлови системи са най-разпространени при използването на Linux операционни системи. Поредицата включва ext1, ext2, ext3, и ext4. Ext1 почти не се използва. Ext2 е много надеждна и е издържала теста на времето (15 г. към момента . всъщност стандартната Linux файлова система, е същата като предшественика ѝ, но с допълнение на функцията “**journaling**”.

Ext4

1. **Superblock (Суперблок):** Това е първият блок на файловата система и съдържа информация за параметрите и статистиката на файловата система, като размер на блока, брой блокове, индекси и други.
2. **Inode Table (Таблица на инодите):** Всяка файлова система Ext4 съдържа таблица на инодите, която съхранява метаданни за всички файлове и директории. Всяка запис в таблицата на инодите съдържа информация като размер, права за достъп, създаване и модификация, атрибути и указатели към блокове на данните.
3. **Data Blocks (Блокове с данни):** Ext4 използва блокове с фиксиран размер, които съхраняват фактическите данни на файловете и директориите. Размерът на блока може да бъде конфигуриран при създаването на файловата система.
4. **Journal (Журнал):** Ext4 използва журнал, който записва промените, извършени върху файловата система, преди те да бъдат приложени. Това позволява на Ext4 да бъде по-устойчива на сбои и осигурява по-бързо възстановяване при необходимост.
5. **Directory Structure (Структура на директориите):** Ext4 използва подобна на Ext2/Ext3 структура на директориите, която съхранява информация за имената на файловете и указатели към инодите, свързани с тях.

HPFS - os/2

- High Performance File System

Работа с файлове в .NET

- Какво представляват потоците?
- Потоците в .NET Framework. Базови и преходни потоци. Основни операции
- Типът `System.IO.Stream`
- Буферирани потоци
- Файлови потоци
- Четци и писачи
- Двоични четци и писачи
- Текстови четци и писачи
- Операции с файлове. Класове `File` и `FileInfo`

Работа с директории

- Работа с директории. Класове `Directory` и `DirectoryInfo`
- Наблюдение на файловата система с `FileSystemWatcher`
- Работа с `IsolatedStorage`

Какво представляват потоците?

- Абстракцията "поток" е основният начин за осъществяване на входно-изходна активност в съвременните обектно-ориентирани езици (C#, C++, Java, Delphi)
- Потоците:
 - са подредени серии от байтове
 - представляват абстрактни канали за данни, до които достъпът се осъществява последователно
 - предоставят механизъм за четене и писане на поредица байтове от и към устройства за съхранение или пренос на данни
- В .NET Framework повечето входно-изходни операции използват потоци

Потоците в .NET Framework

- В .NET Framework потоците са два вида:
 - Базови потоци (base stream)
 - четат и пишат данни от и към външен механизъм за съхранение на данни
 - примери: FileStream, MemoryStream, NetworkStream
 - Преходни потоци (pass-through streams)
 - четат и пишат в други потоци, като добавят допълнителна функционалност (напр. буфериране, кодиране и компресиране)
 - например: BufferedStream и CryptoStream

Типът System.IO.Stream

- **По-важни методи на класа Stream:**
 - int Read(byte[] buffer, int offset, int count) – чете най-много count байта от входен поток, увеличава текущата позиция и връща колко байта е прочел или 0 при край на потока
 - Read(...) може да блокира за неопределено време докато прочете поне 1 байт
 - Read(...) може да прочете по-малко от заявения брой байтове
 - Write(byte[] buffer, int offset, int count) – записва в потока поредица от count байта, започвайки от дадена позиция в масив
 - Write(...) може да блокира за неопределено време, докато изпрати всички байтове към местоназначението им

Типът System.IO.Stream

- По-важни методи на класа Stream:
 - Flush () – изпраща данните от вътрешните буфери към механизма за съхранение/пренос на данни
 - Close () – извиква Flush (), затваря връзката към механизма за съхранение/пренос на данни и освобождава използваните ресурси
 - Seek (offset, SeekOrigin) – премества текущата позиция в потока (ако се поддържа) с дадено отместване спрямо началото, края или текущата позиция в потока
 - SetLength (long) – променя дължината на потока (ако се поддържа)
 - Ако позициониране не се поддържа, се хвърля изключение NotSupportedException

Буферирани потоци

- Буферираните потоци:
 - Буферират данните и така значително подобряват производителността
 - При заявка за прочитане на 1 байт се прочитат още няколко килобайта напред и се записват във вътрешен буфер
 - При следващо четене се връщат данни от буфера (което е много бърза операция)
 - При писане данните се записват във вътрешен буфер (което става много бързо)
 - При препълване на буфера или при извикване на `Flush()` данните от буфера се изпращат към механизма за съхранение/пренос
- В .NET Framework се използва класа `System.IO.BufferedStream`

Файлови потоци

- Конструиране на файлов поток:

```
FileStream fs = new FileStream(string fileName,  
    FileMode [, FileAccess [, FileShare]]);
```

- FileMode – начин на отваряне на файла
 - Open, Append, Create, CreateNew, OpenOrCreate, Truncate
- FileAccess – режим на отваряне на файла
 - Read, Write, ReadWrite
- FileShare – режим на достъп за други потребители докато ние държим отворен файла
 - None, Read, Write, ReadWrite

Файлови потоци – пример

```
// Замяна на всички нули (0x00) в двоичен файл с 255 (0xFF)
```

```
FileStream fs = new FileStream("input.bin",
    FileMode.Open, FileAccess.ReadWrite, FileShare.None);
using (fs)
{
    byte[] buf = new byte[16384];
    while (true)
    {
        int bytesRead = fs.Read(buf, 0, buf.Length);
        if (bytesRead == 0)
            break;
        for (int i=0; i<bytesRead; i++)
        {
            if (buf[i] == 0)
                buf[i] = 255;
        }
        fs.Seek(-bytesRead, SeekOrigin.Current);
        fs.Write(buf, 0, bytesRead);
    }
}
```

Файлови потоци – пример

```
using System;
using System.IO;

// Намиране броя на нулевите байтове в двоичен файл
class CountZeros
{
    static void Main(string[] args)
    {
        long countOfZeros = 0;
        FileStream fs =
            new FileStream("input.bin", FileMode.Open,
                           FileAccess.Read, FileShare.Read);
        try
        {
            byte[] buf = new byte[4096];
            while (true)
            {
                int bytesRead =
                    fs.Read(buf, 0 , buf.Length);
                if (bytesRead == 0)
                    break;          (примерът продължава)
            }
        }
    }
}
```

Файлови потоци – пример

```
for (int i=0; i<bytesRead; i++)
{
    if (buf[i] == 0)
        countOfZeros++;
}
}
}
finally
{
    fs.Close();
}

Console.WriteLine(
    "The count of zeros in the file is {0}.",
    countOfZeros);
}
}
```

Четци и писачи

- Четците и писачите са класове, които:
 - улесняват работата с потоци
 - позволяват четене и писане на различни структури от данни, например примитивните типове, текстова информация и други типове
 - биват двоични и текстови
- Класовете `BinaryReader` и `BinaryWriter`
 - осигуряват четене и записване на примитивните типове данни в двоичен вид
 - `ReadChar()`, `ReadChars()`, `ReadInt32()`, `ReadDouble()`, ...
 - `Write(char)`, `Write(char[])`, `Write(Int32)`, `Write(Double)`, ...
 - позволяват четене и писане на `string`, като го записват като масив от символи, предхождан от дължината му: `ReadString()`, `Write(string)`

Бинарни четци и писачи – пример

- Имаме бинарен файл със записи във формат (име: string, възраст: int)
- За добавяне и четене на записи можем да ползваме следния код:

```
static void AppendPerson(BinaryWriter aWriter,  
    string aName, int aAge)  
{  
    aWriter.Write(aName);  
    aWriter.Write(aAge);  
}  
  
static void ReadPerson(BinaryReader aReader,  
    out string aName, out int aAge)  
{  
    aName = aReader.ReadString();  
    aAge = aReader.ReadInt32();  
}
```

Бинарни четци и писачи – пример

- Ето и цялостен пример за записване и прочитане на няколко записи:

```
static void Main(string[] args)
{
    //Създаваме файла data.bin и записваме
    //в него няколко записи

    FileStream fs =
        new FileStream("data.bin", FileMode.Create);
    using (fs)
    {
        using (BinaryWriter writer = new BinaryWriter(fs))
        {
            AppendPerson(writer, "Бай Иван", 57);
            AppendPerson(writer, "Цар Киро", 48);
            AppendPerson(writer, "Кака Мара", 26);
        }
    }
}                                (примерът продължава)
```

Бинарни четци и писачи – пример

- Ето и цялостен пример за записване и прочитане на няколко записи:

//Прочитаме и отпечатваме всички записи от файла data.bin

```
fs = new FileStream("data.bin", FileMode.Open);
using (fs)
{
    using (BinaryReader reader = new BinaryReader(fs))
    {
        while (fs.Position < fs.Length - 1)
        {
            string name;
            int age;
            ReadPerson(reader, out name, out age);
            Console.WriteLine("{0} - {1}", name, age);
        }
    }
}
```

Текстови четци и писачи

- Класовете `TextReader` и `TextWriter`
 - осигуряват четене и записване на текстова информация (низове, разделени с нов ред)
 - използват се по същия начин като класа `Console` (има `ReadLine()`, `WriteLine(...)`, ...)
 - символът за нов ред е различен за различните платформи:
 - `LF` (`0x0A`) – в Unix и Linux
 - `CR LF` (`0x0D 0x0A`) – в Windows и DOS
 - `ReadLine()` – прочита текстов ред
 - `ReadToEnd()` – прочита всичко до края на потока
 - `Write(...)` – пише текст в потока
 - `WriteLine(...)` – пише текстов ред в потока

Текстови четци и писачи

- Класовете `TextReader` и `TextWriter` са абстрактни и не се използват директно
- Използват се следните класове:
 - `StreamReader` – чете текстови данни от поток
 - `StringReader` – чете текстови данни от низ
 - `StreamWriter` – пише текстови данни в поток
 - `StringWriter` – пише текстови данни в низ, използва вътрешно `StringBuilder`
- Пример:
 - Даден е текстов файл. Искаме да добавим номерация в началото на всеки ред от файла
 - Използваме `StreamReader` и `StreamWriter`, четем всеки ред и го отпечатваме като добавяме номерация

Текстови четци и писачи – пример

```
// Номериране на редовете на текстов файл
static void Main(string[] args)
{
    StreamReader reader = new StreamReader("in.txt");
    using (reader)
    {
        StreamWriter writer = new StreamWriter("out.txt");
        using (writer)
        {
            int lineNumber = 0;
            string line = reader.ReadLine();
            while (line != null)
            {
                lineNumber++;
                writer.WriteLine("{0,5} {1}",
                    lineNumber, line);
                line = reader.ReadLine();
            }
        }
    }
}
```

Текстови четци и писачи – пример

```
// Find-text-in-file Search Utility

static void Main(string[] args)
{
    if (args.Length < 2)
    {
        Console.WriteLine("Use: FindInFile file text");
        return;
    }

    string fileName = args[0];
    if (!File.Exists(fileName))
    {
        Console.WriteLine(
            "Could not find file: " + fileName);
        return;
    }

    string textToFind = args[1].Trim();
```

(примерът продължава)

Текстови четци и писачи – пример

```
using (StreamReader reader = File.OpenText(fileName))
{
    bool found = false;
    int lineNumber = 0;
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        lineNumber++;
        if (line.IndexOf(textToFind) > -1)
        {
            Console.WriteLine("{0}: {1}",
                lineNumber, line);
            found = true;
        }
    }
    if (!found)
    {
        Console.WriteLine("Text not found!");
    }
}
```

Четци, писачи и кодировки

- Четците и писачите използват кодировки за да боравят с текстова информация
 - по подразбиране се използва кодиране UTF-8
 - в конструкторите могат да се задават и други кодировки, инстанции на `System.Encoding`
- Създаване на windows-1251 четец:

```
Encoding win1251 = Encoding.GetEncoding("windows-1251");
StreamReader reader = new StreamReader("in.txt", win1251);
```

- Създаване на windows-1251 писач:

```
Encoding win1251 = Encoding.GetEncoding("windows-1251");
StreamWriter writer =
    new StreamWriter("out.txt", win1251);
```

Класовете File и FileInfo

- File и FileInfo са помощни класове, които предоставят функционалност за:
 - създаване на файл – Create (), CreateText ()
 - отваряне на файл – Open (), OpenRead (), OpenWrite (), AppendText ()
 - копиране на файл – CopyTo (...)
 - местене (преименуване) на файл – MoveTo (...)
 - изтриване на файл – Delete ()
 - извличане на времето на последен достъп и промяна – LastAccessTime и LastWriteTime
 - проверка за съществуване – Exists (...)
- Класът File предоставя тези методи статични, а FileInfo чрез инстанция

File и FileInfo – пример

```
static void Main(string[] args)
{
    StreamWriter writer = File.CreateText("test1.txt");
    using (writer)
    {
        writer.WriteLine("Налей ми бира!");
    }

    FileInfo fileInfo = new FileInfo("test1.txt");
    fileInfo.CopyTo("test2.txt", true);
    fileInfo.CopyTo("test3.txt", true);

    if (File.Exists("test4.txt"))
    {
        File.Delete("test4.txt");
    }

    File.Move("test3.txt", "test4.txt");
}
```

Directory и DirectoryInfo

- **Directory** и **DirectoryInfo** са класове, които предоставят функционалност за:
 - създаване на директории и поддиректории – `Create()`, `CreateSubdirectory()`
 - извличане на всички файлове – `GetFiles()`
 - извличане на всички поддиректории – `GetDirectories()`
 - местене (преименуване) – `MoveTo(...)`
 - изтриване – `Delete()`
 - извличане на горната директория – `Parent`
 - проверка за съществуване – `Exists()`
 - извличане на пълното име – `FullName`
- Класът **Directory** предоставя тези методи статични, а **DirectoryInfo** чрез инстанция

Класът Directory – пример

```
class DirectoryTraversal
{
    private static void Traverse(string aPath)
    {
        Console.WriteLine("[{0}", aPath);
        string[] subdirs = Directory.GetDirectories(aPath);
        foreach (string subdir in subdirs)
        {
            Traverse(subdir);
        }
        string[] files = Directory.GetFiles(aPath);
        foreach (string f in files)
        {
            Console.WriteLine(f);
        }
    }

    static void Main(string[] args)
    {
        string winDir = Environment.SystemDirectory;
        Traverse(winDir);
    }
}
```

Класът Path

- Класът `System.IO.Path` предоставя функционалност за работа с пътища
- По-важни свойства и методи:
 - `DirectorySeparatorChar` – символът, който отделя директориите в пътя ("\\" за Windows и "/" за Unix и Linux файлови системи)
 - `Combine (...)` – комбинира пълен път с релативен път
 - `GetExtension (...)` – извлича разширението на даден файл (ако има)
 - `GetFileName (...)` – извлича името на файла от даден пълен път (ако има)
 - `GetTempFileName (...)` – създава временен файл с уникално име и нулема дължина и връща името му

Временни файлове – пример

```
using System;
using System.IO;

class TempFilesDemo
{
    static void Main(string[] args)
    {
        String tempFileName = Path.GetTempFileName();
        try
        {
            using (TextWriter writer =
                new StreamWriter(tempFileName))
            {
                writer.WriteLine("This is just a test");
            }
            File.Copy(tempFileName, "test.txt");
        }
        finally
        {
            File.Delete(tempFileName);
        }
    }
}
```

Специални директории

- Специалните директории са достъпни от `System.Environment.GetFolderPath (Environment.SpecialFolder)`:
 - `SpecialFolder.DesktopDirectory` – връща работния плот на текущия потребител
 - `SpecialFolder.ApplicationData` – връща директория за приложенията на текущия user
 - `SpecialFolder.Favorites` – директорията "My Documents" на текущия потребител
 - `SpecialFolder.Favorites` – директорията с любимите връзки на текущия потребител
 - `SpecialFolder.MyPictures/MyMusic` – директория "моите картинки/музика" на текущия потребител

Специални директории – пример

```
string myDocuments = Environment.GetFolderPath(  
    Environment.SpecialFolder.Personal);  
Console.WriteLine(myDocuments);  
// C:\Documents and Settings\Administrator\My Documents  
  
string myDesktop = Environment.GetFolderPath(  
    Environment.SpecialFolder.DesktopDirectory);  
Console.WriteLine(myDesktop);  
// C:\Documents and Settings\Administrator\Desktop  
  
string myFavourites = Environment.GetFolderPath(  
    Environment.SpecialFolder.Favorites);  
Console.WriteLine(myFavourites);  
// C:\Documents and Settings\Administrator\Favorites  
  
string myMusic = Environment.GetFolderPath(  
    Environment.SpecialFolder.MyMusic);  
Console.WriteLine(myMusic);  
// C:\Documents and Settings\Administrator\My Documents\My Music
```

Класът FileSystemWatcher

- Класът `FileSystemWatcher` позволява наблюдение на файловата система за различни събития като:
 - създаване, промяна, преименуване и изтриване на файл или директория
- По-важни свойства и събития:
 - `Path` – директорията, която се наблюдава
 - `Filter` – филтър за наблюдаваните файлове, например `"*.*"` или `"*.exe"`
 - `NotifyFilter` – филтър за наблюдаваните събития, например `FileName`, `LastWrite`, `Size`
 - `Created`, `Changed`, `Renamed`, `Deleted` – събития, които се извикват при промяна. Внимание: извикват се от друга нишка!

FileSystemWatcher – пример

```
static void Main()
{
    string currentDir = Environment.CurrentDirectory;
    FileSystemWatcher w =
        new FileSystemWatcher(currentDir);
    w.Filter = "*.*";
    w.NotifyFilter = NotifyFilters.FileName |
        NotifyFilters.DirectoryName |
        NotifyFilters.LastWrite;
    w.Created += new FileSystemEventHandler(OnCreated);
    w.Changed += new FileSystemEventHandler(OnChanged);
    w.Renamed += new RenamedEventHandler(OnRenamed);
    w.EnableRaisingEvents = true;

    Console.WriteLine("{0} is being watched now...",
        currentDir);
    Console.WriteLine("Press [Enter] to exit.");
    Console.ReadLine();
}
```

(примерът продължава)

FileSystemWatcher – пример

```
private static void OnCreated(object aSource,
    FileSystemEventArgs aArgs)
{
    Console.WriteLine("File: {0} created - {1}",
        aArgs.Name, aArgs.ChangeType);
}

private static void OnChanged(object aSource,
    FileSystemEventArgs aArgs)
{
    Console.WriteLine("File: {0} changed - {1}",
        aArgs.Name, aArgs.ChangeType);
}

private static void OnRenamed(object aSource,
    RenamedEventArgs aArgs)
{
    Console.WriteLine("File: {0} renamed to {1}",
        aArgs.OldName, aArgs.Name);
}
```

Работа с IsolatedStorage

- Технологията IsolatedStorage:
 - се използва когато приложение, което няма права за достъп до локалния твърд диск, трябва да съхранява някъде файлове
 - например при приложения, стартирани от Web-страница в Интернет
- IsolatedStorage предоставя изолирана виртуална файлова система:
 - ограничена по обем
 - без да позволява на приложението достъп до останалите файлове на твърдия диск
- Класовете за достъп до изолирани файлови системи се намират в пакета System.IO.IsolatedStorage (вж. MSDN).