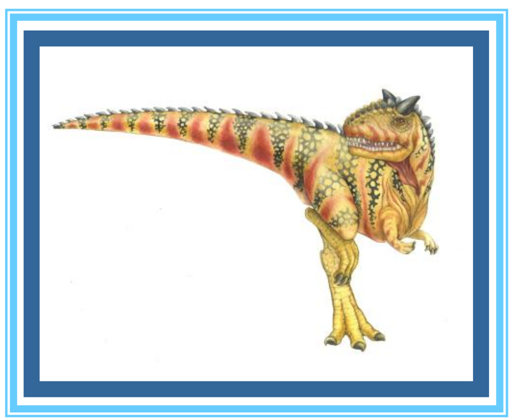
Chapter 13: File-System Interface



Fisiere

- abstractie de nivel de sistem de operare pt stocarea persistenta a datelor
- la nivelul cel mai de jos, stocarea persistenta se face pe discuri (mai recent, memorii flash, SSD, NVRAM, etc)
- sistemul de fisiere
  - componenta a sistemului de operare (i.e., parte a kernelului)
  - gestioneaza mediul de stocare persistenta a datelor
  - ofera la nivelul de aplicatie abstractia de fisier si apeluri sistem corespunzatoare
- fisierele
  - concret, containere pt stocarea persistenta a datelor
  - uzual referite prin nume (string ASCII) convertit la o reprezentare interna a kernelului de catre sistemul de fisiere
  - paradigma uzuala de folosire: open – read/write - close

File Concept

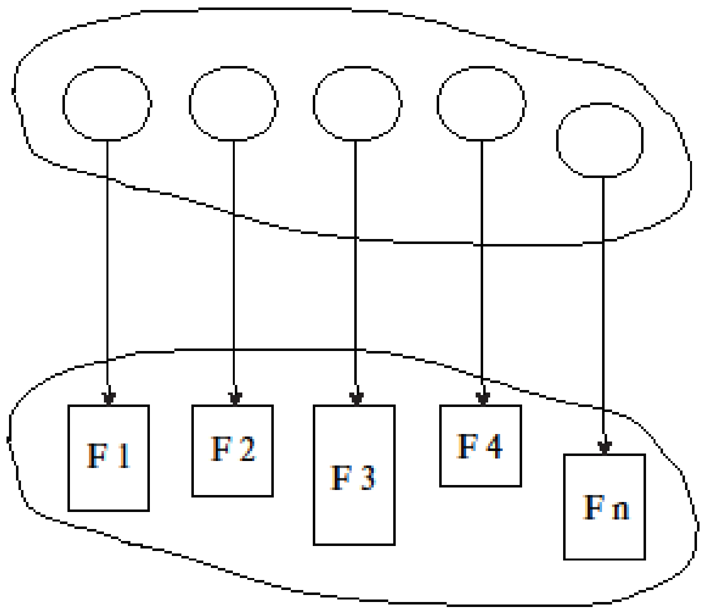
- Contiguous logical address space
- Types:
  - Data
    - Numeric
    - Character
    - Binary
  - Program
- Contents defined by file's creator
  - Many types
    - **text file,**
    - **source file,**
    - **executable file**

File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum
- Information kept in the directory structure

Directory Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk

File Operations

- **Create**
- **Write –** at **write pointer** location
- **Read –** at **read pointer** location
- **Reposition within file - seek**
- **Delete**
- **Truncate**
- ***Open*** **(*F_i*)** – search the directory structure on disk for entry ***F_i***, and move the content of entry to memory
- ***Close*** **(*F_i*)** – move the content of entry ***F_i*** in memory to directory structure on disk

Open Files

- Several pieces of data are needed to manage open files:
  - **Open-file table**: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

File Locking

- Provided by some operating systems and file systems
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

File Types – Name, Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

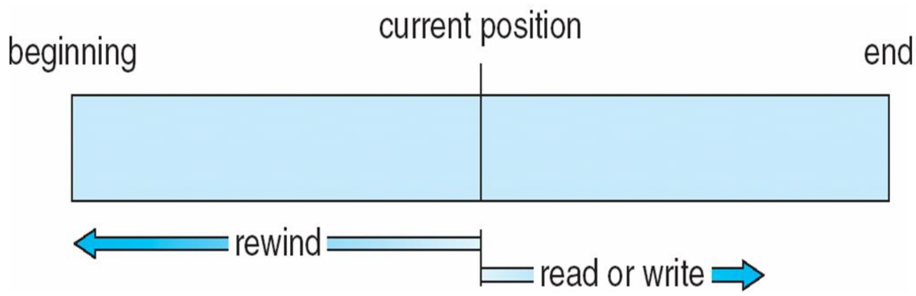
- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - Program

Access Methods

- A file is fixed length **logical records**
- **Sequential Access**
- **Direct Access**
- **Other Access Methods**

Sequential Access

- Operations
  - **`read next`**
  - **`write next`**
  - **`Reset`**
  - no read after last write (rewrite)
- Figure



Direct Access

- Operations
  - **`read n`**
  - **`write n`**
  - **`position to n`**
    - **`read next`**
    - **`write next`**
    - **`rewrite n`**

  n = **relative block number**

- Relative block numbers allow OS to decide where file should be placed

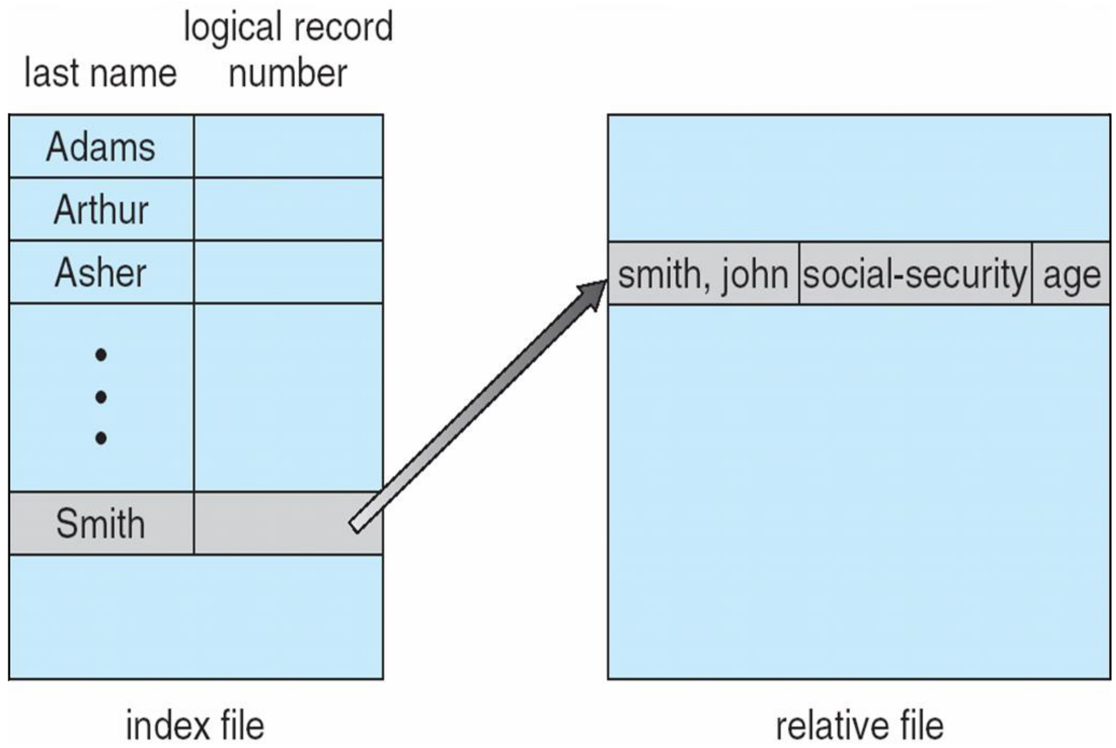
Simulation of Sequential Access on Direct-access File

sequential access	implementation for direct access
reset	$cp = 0;$
read next	*read cp*; $cp = cp + 1;$
write next	*write cp*; $cp = cp + 1;$

Other Access Methods

- Can be other access methods built on top of base methods
- General involve creation of an **index** for the file
- Keep index in memory for fast determination of location of data to be operated on (consider Universal Produce Code (UPC code) plus record of data about that item)
- If the index is too large, create an in-memory index, which an index of a disk index
- IBM indexed sequential-access method (ISAM)
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - All done by the OS
- VMS operating system provides index and relative files as another example (see next slide)

Example of Index and Relative Files



Fisiere in Unix

- perspectiva particulara: datele utilizator, organizarea fisierelor in directoare si driverele de echipamente accesibile prin intermediul interfetei sist. de fisiere
- tipuri de fisiere
  - fisiere obisnuite:
    - stocheaza datele utilizator
    - implementate ca un stream/sir de bytes (complet nestructurat)
  - fisiere director (directoare):
    - contin nume de fisiere obisnuite
    - instituie o structura ierarhica a spatiului de nume pt fisiere
  - fisiere speciale
    - interfata catre driverele de echipamente sau structuri de date speciale ale kernelului (eg, named pipes pt IPC)
- traditional, toate fisierele se accesau folosind aceleasi apeluri sistem
- ulterior, API special pt lucrul cu directoare (opendir – readdir – rewinddir – closedir)

Fisiere Unix la nivel aplicatie

- inainte de accesul la date, un fisier trebuie “deschis” (apel de sistem *open*)
- kernelul aloca un *descriptor de fisier* care este returnat aplicatiei pt folosinta
- intern, kernelul aloca si un obiect corespunzator fisierului deschis care contine un *file pointer* (setat pe 0, imediat dupa *open*) care reflecta offsetul in cadrul stream-ului de bytes
- pozitia file pointer-ului se poate schimba cu *seek*
- *read/write* lucreaza cu byte-ul referit de catre file pointerul current
- doua procese care deschid acelasi fisier folosesc offset-uri diferite in fisier
- descriptorii de fisiere se pot duplica folosind *dup/dup2*, situatie in care file pointer-ul este partajat
- fisierele sunt organizate intr-o ierarhie arborescenta cu o radacina (*root*) unica
  - in general insa, in arborele de fisiere coexista mai multe sisteme de fisiere atasate structurii arborescente de directoare cu ajutorul unei operatii speciale *mount*

Folosirea file pointer-ului

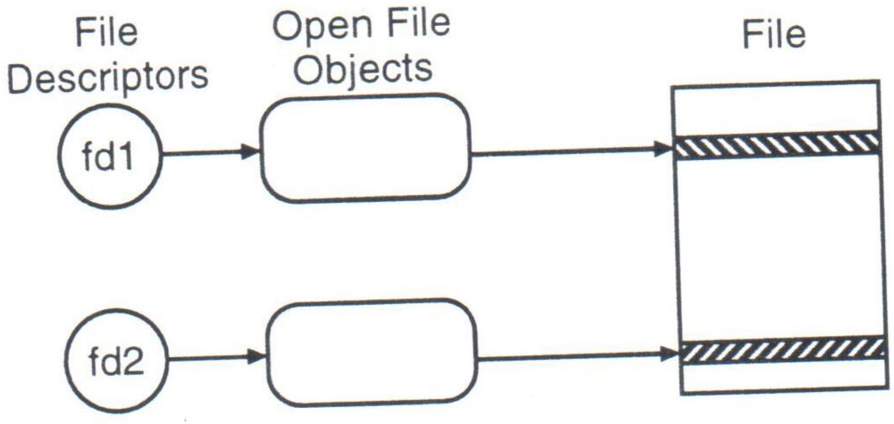


Figure 1: A file which has been opened twice.

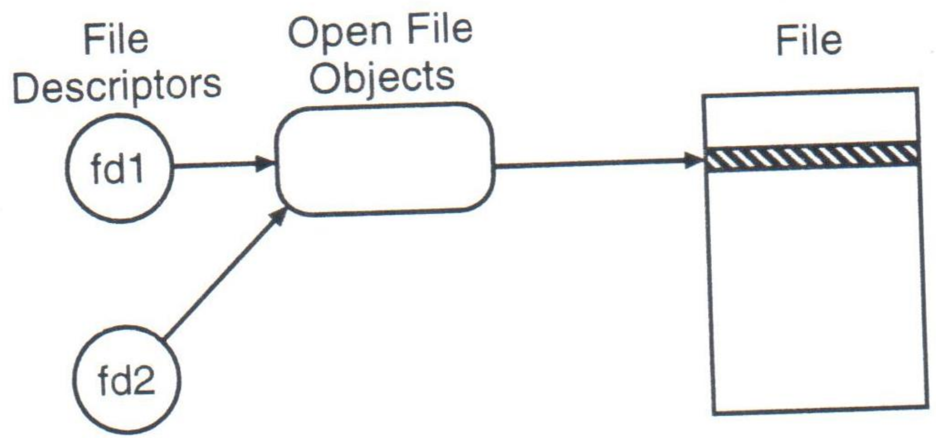


Figure 2: `fd2 = dup(fd1)`

Arhitectura discurilor

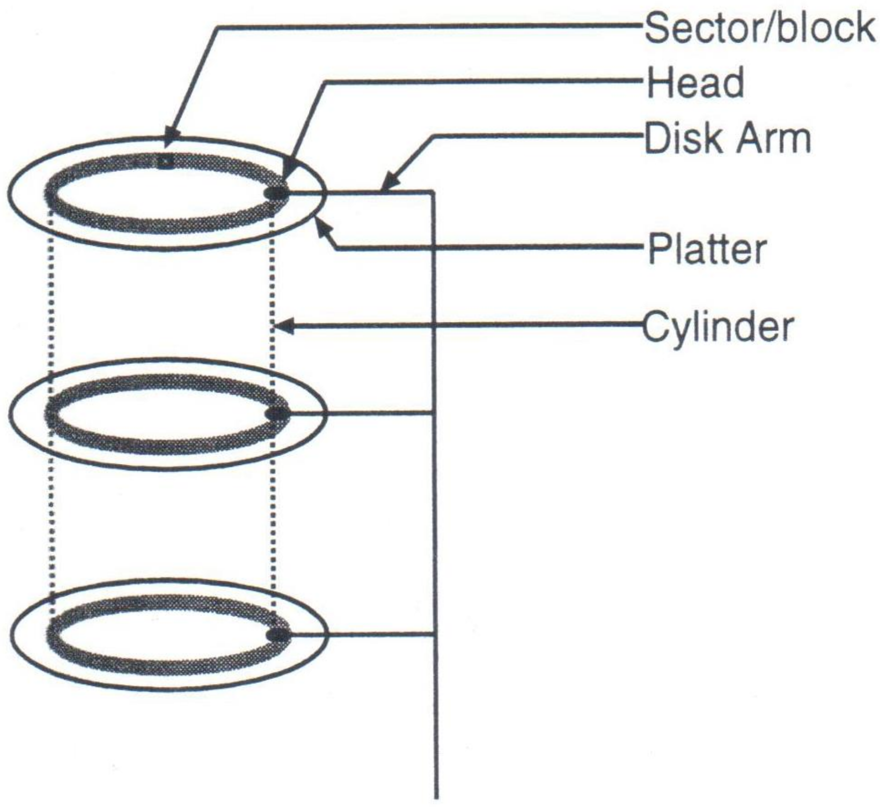


Figure 3: Geometry of a hard disk drive.

Arhitectura discurilor (cont'd)

- mediul de stocare format prin “stivuirea” unor *platane* pe un ax care formeaza un *pachet*
- acest pachet este invartit in jurul axului la o viteza de rotatie constanta (5000-10000 rpm)
- blocurile de disc (*sectoare*) situate la aceeasi distanta de centrul unui platan alcatuiesc o *pista*
- setul de piste aflat la aceeasi distanta fata de axul platanelor alcatuieste un *cilindru*
- toate datele dintr-un cilindru pot fi accesate simultan fara miscarea capului de citire
- datele se citesc in multipli de dimensiunea blocului (1KB – 4KB)

Citirea blocurilor

- se muta capul de citire deasupra cilindrului care contine blocul de date (*seek*) - cca 5 ms
- se asteapta pana cand discul se roteste a.i. datele sa ajunga sub capul de citite (*rotational delay*) – cca 4ms pt un disc cu 7200 rpm
- se transfera datele (*transfer*) prin alegerea capului de citire de deasupra pistei pe care se afla datele – cca 1ms pt 1KB
- timpul mediu de acces random in general

$$t_{seek} + t_{rotatie} + t_{transfer} = 9.1 \text{ ms}$$

- timpul mediu de acces random de pe acelasi cilindru

$$t_{rotatie} + t_{transfer} = 4.1 \text{ ms}$$

- timpul de citire a urmatorului bloc de pe aceeasi pista

$$t_{transfer} = 0.1 \text{ ms}$$

- idee centrala: minimizare timpilor de seek si rotatie

Algoritmi de disk scheduling

- incearca sa minimizeze timpul de cautare (*seek time*)
- FCFS – simplu, dar ineficient
  - ex: coada de cereri blocuri aflate pe cilindrii 53, 98, 183, 37, 122, 14, 124, 65, 67 => capetele de citire se vor muta peste 640 de cilindri

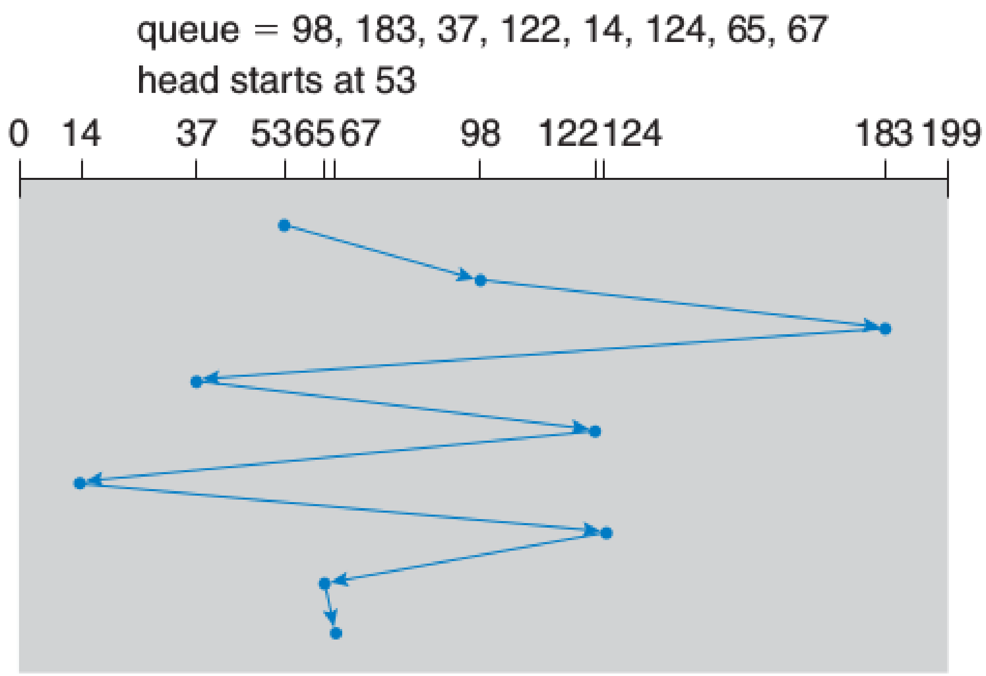


Figure 9.4 FCFS disk scheduling.

SSTF (shortest seek time first)

- serveste cererile de pe cilindrii cei mai apropiati de pozitia curenta a capetelor de citire
- ex anterior: capetele de citire se muta peste 236 de cilindri
- nu e optimal, ex: 53, 37, 14, 65, 67, 98, 122, 124, 183 => 208 cilindri parcursi
- sufera de starvation: daca apar in permanenta cereri in apropierea capetelor de citire, cererile “indepartate” sunt intarziate indefinit

SSTF (shortest seek time first)

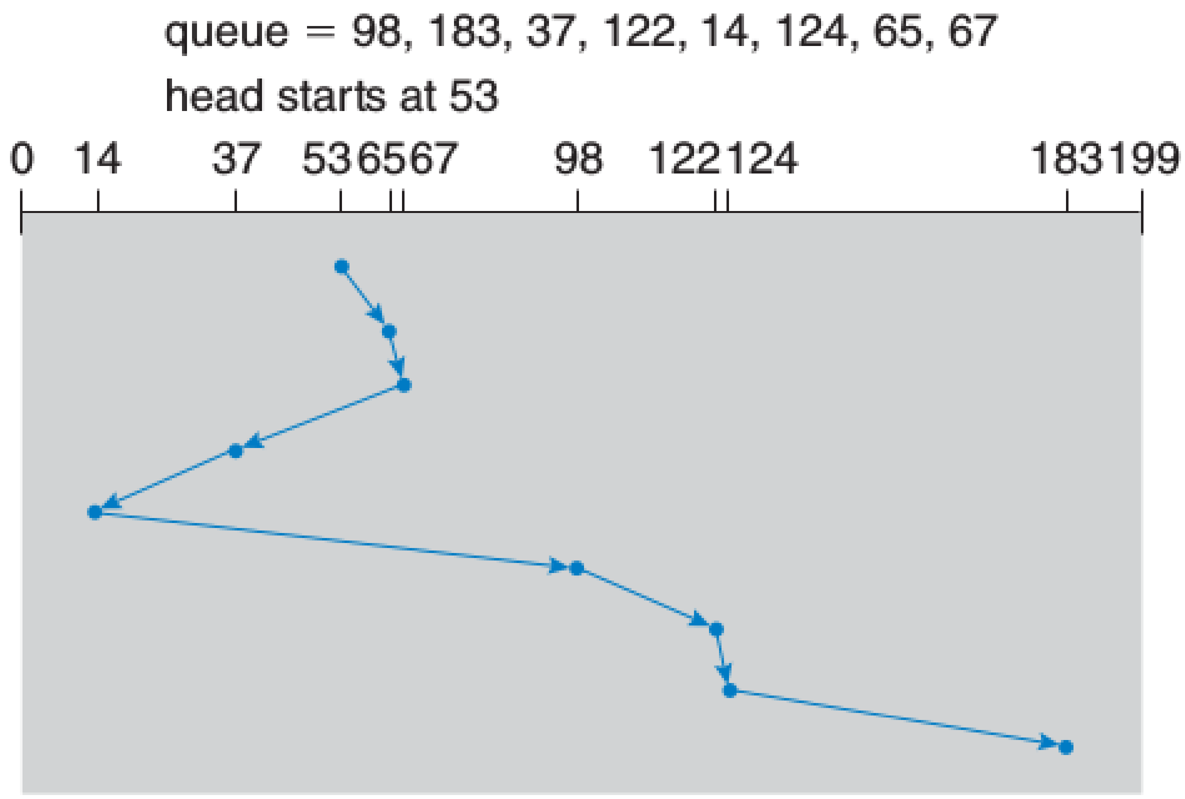


Figure 9.5 SSTF disk scheduling.

Algoritmi de disk scheduling (cont'd)

- SCAN (algoritmul liftului)
  - bratul discului porneste de la un capat al acestuia catre celalalt si serveste cererile intalnite in cale
  - ajuns la capatul discului o ia in sens invers
  - ex. anterior: 203 salturi de cilindri
  - o cerere aparuta chiar inaintea capului de citire e servita imediat
  - o cerere aparuta imediat in spatele capului de citire e intarziata pana se intoace capul de citire in sens contrar

SCAN

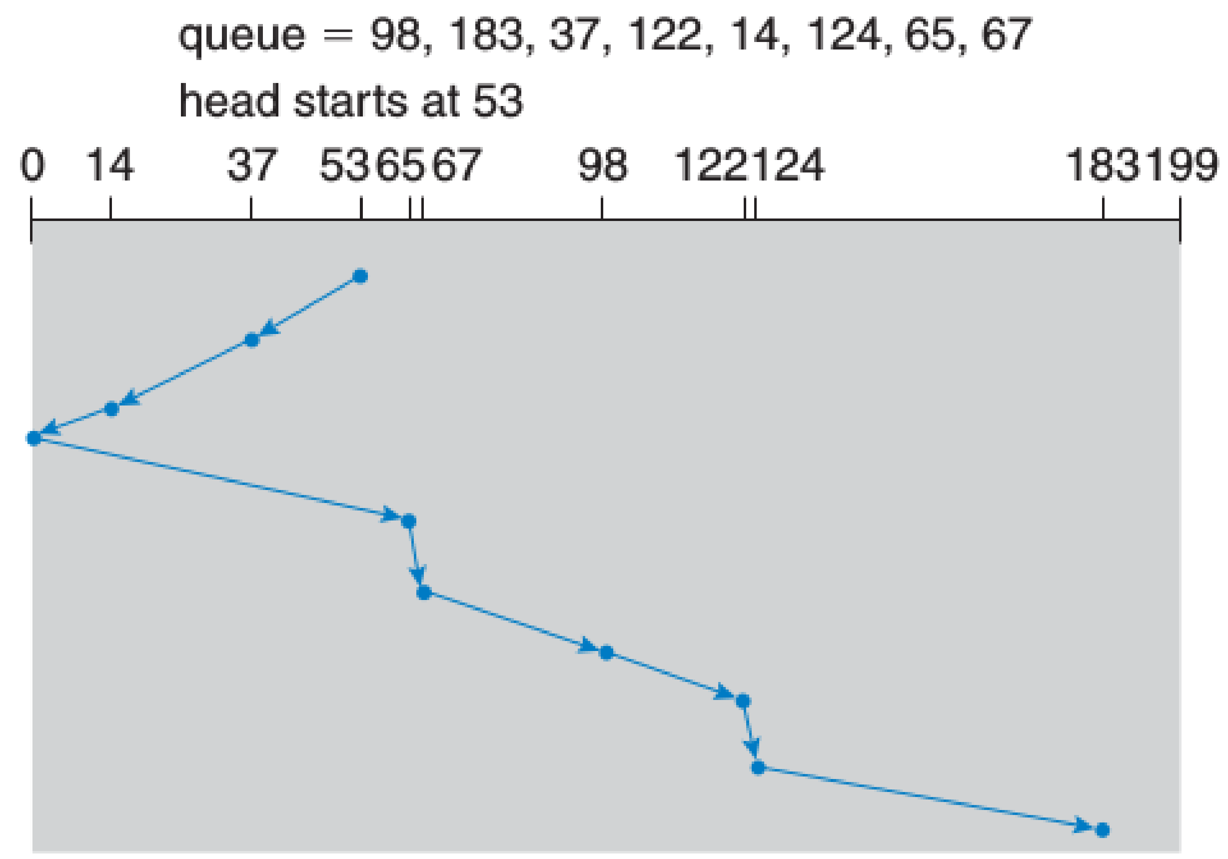


Figure 9.6 SCAN disk scheduling.

Algoritmi de disk scheduling (cont'd)

- C-SCAN (circular SCAN)
  - pt o distributie uniforma a cererilor, cand bratul ajunge la capatul discului si se intoarce, exista relativ putine cereri in fata capului pt ca acestea tocmai au fost tratate
  - densitatea mare de cereri noi e la celalalt capat al discului unde cererile au asteptat cel mai mult
  - ofera timp de asteptare mai uniform (la momentul atingerii capatului discului, capul de citire se intoarce la celalalt capat fara a mai trata cererile din fata capului de citire)
  - trateaza cilindrii discului ca pe o lista circulara
  - ex anterior: bratul se muta peste 167 de cilindri (se considera ca mutarea capului de citire la inceputul discului e o operatie f. rapida)

C-SCAN

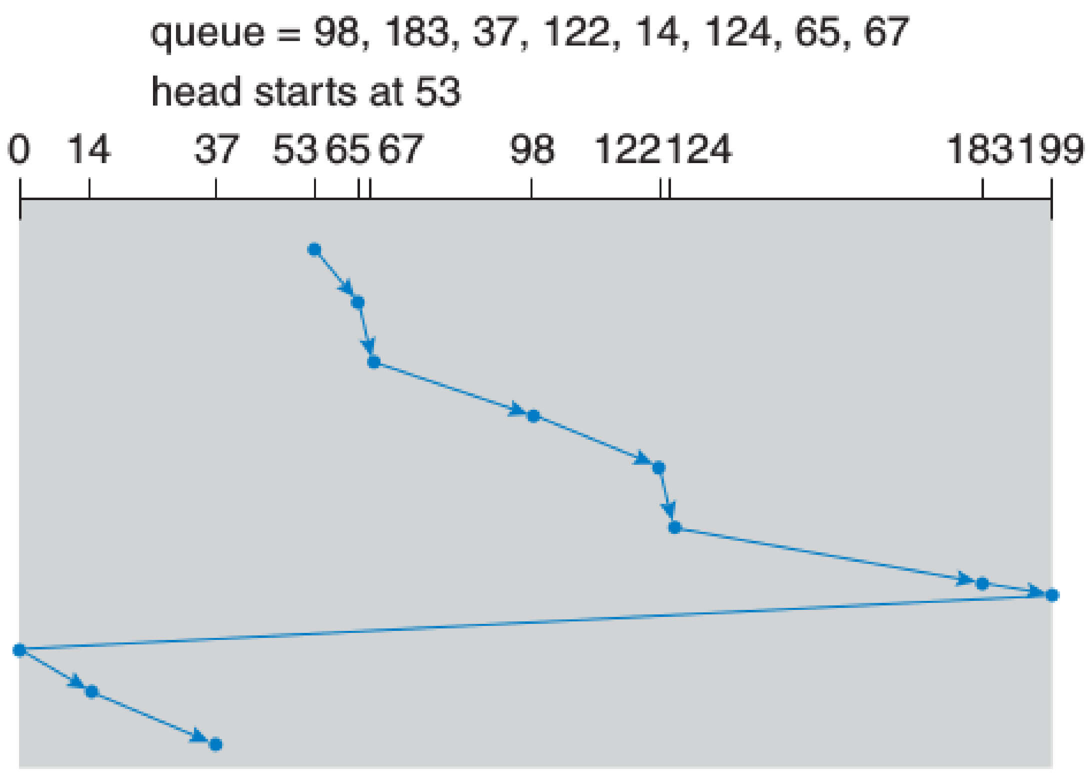


Figure 9.7 C-SCAN disk scheduling.

Algoritmi de disk scheduling (cont'd)

- LOOK
  - SCAN si C-SCAN muta capul de citire peste tot discul
  - o implementare practica ia in calcul doar cererile pt cilindrii situati intre nr minim, respectiv maxim din lista de cereri
  - algoritmii respectivi s.n. LOOK si respectiv C-LOOK
  - ex anterior: la cilindrul 183 bratul se intoarce imediat la cilindrul 14

LOOK

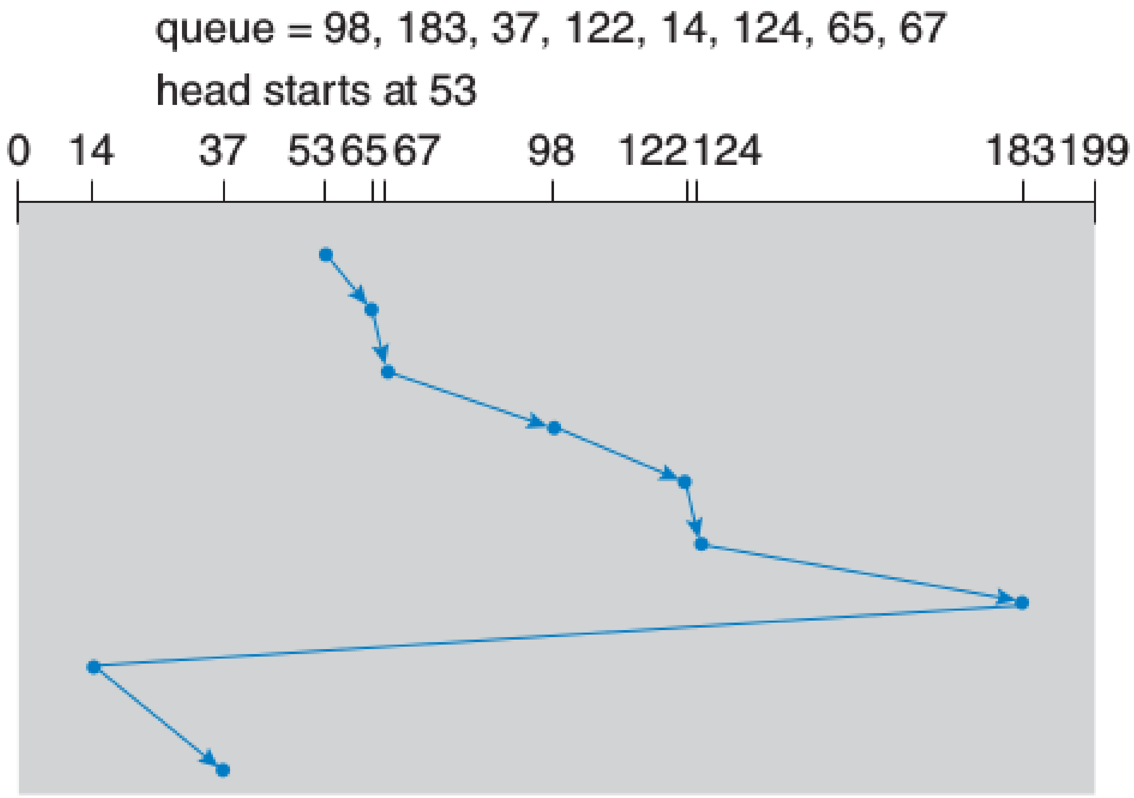


Figure 9.8 C-LOOK disk scheduling.

Factori care influenteaza performanta

- nr de cereri si tipul lor
  - daca exista o singura cerere in coada, toti algoritmii se comporta ca FCFS
- metoda de alocare a fisierului
  - alocare contigua – miscare limitata a capetelor de citire
  - alocare indexata – blocurile pot fi imprastiate pe disc => deplasari mari ale capetelor de citire
- plasamentul directoarelor si indexarea blocurilor
  - directoarele sunt accesate frecvent (deschiderea unui fisier pp accesarea unui director)
  - daca intrarea in director e pe cilindrul 1 si datele sunt pe ultimul cilindru, bratul de citire trebuie sa parcurga toti cilindrii
  - daca directorul e plasat pe un cilindru din mijloc, bratul se misca doar jumatare din “latimea” discului in termeni de cilindri
  - caching-ul directoarelor si al indecsilor de blocuri ajuta f. mult la evitarea miscarilor bratului discului

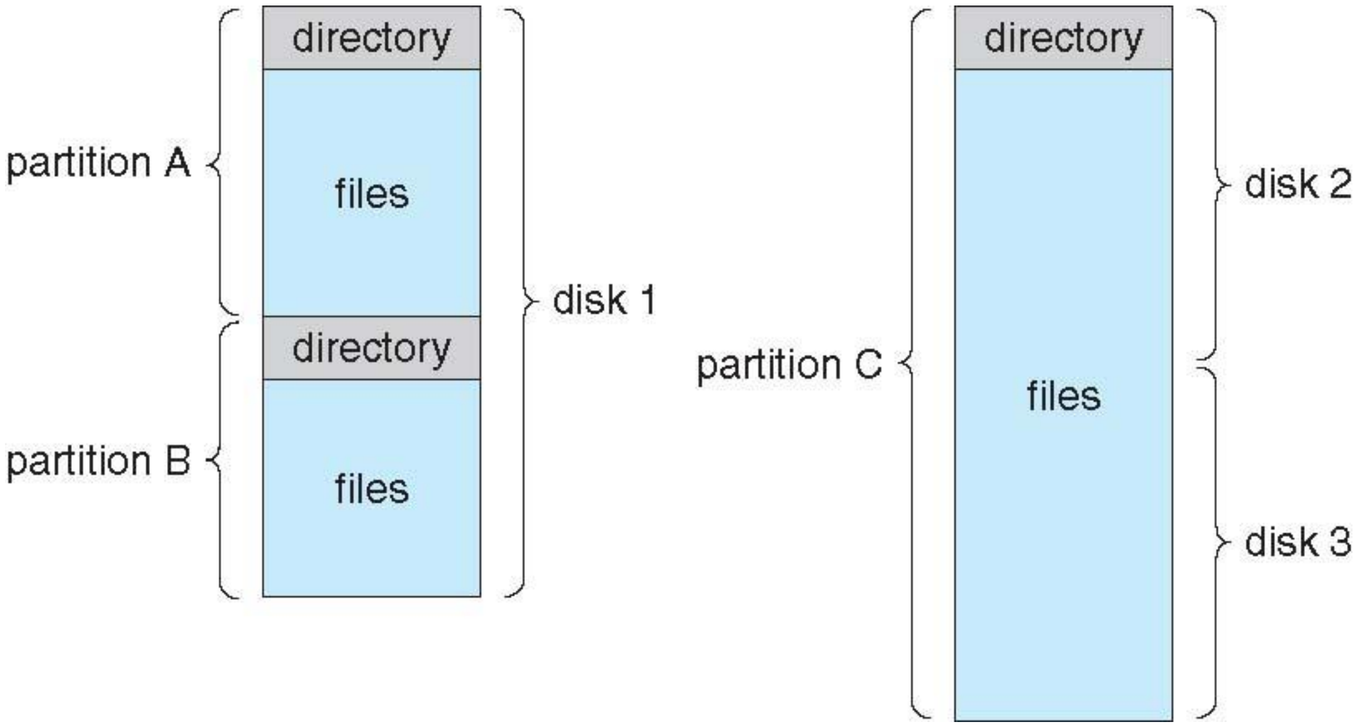
Implementare HW vs SW

- algoritmii anteriori nu iau in calcul timpul rotational care e dificil de optimizat fara cunostinte despre arhitectura HW a discului
- solutie: discurile moderne implementeaza algoritmii anteriori in HW pt a putea optimiza simultan si timpul de cautare si pe cel rotational
- pe de alta parte, sistemul de operare are propriile motive sa implementeze scheduling-ul de disc, de ex:
  - paginarea la cerere e mai importanta decat I/O-ul aplicatiilor
  - scrierile sunt mai importante decat citirile cand cache-urile din sistem nu mai au memorie suficienta
  - pt a preveni pierderea de date, e bine sa se garanteze scrierea unui anumit nr de blocuri de disc inaintea unui eventual crash

Disk Structure

- Disk can be subdivided into **partitions**
- Disks or partitions can be **RAID** protected against failure
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
- Partitions also known as minidisks, slices
- Entity containing file system is known as a **volume**
- Each volume containing a file system also tracks that file system's info in **device directory** or **volume table of contents**
- In addition to **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same operating system or computer

A Typical File-system Organization

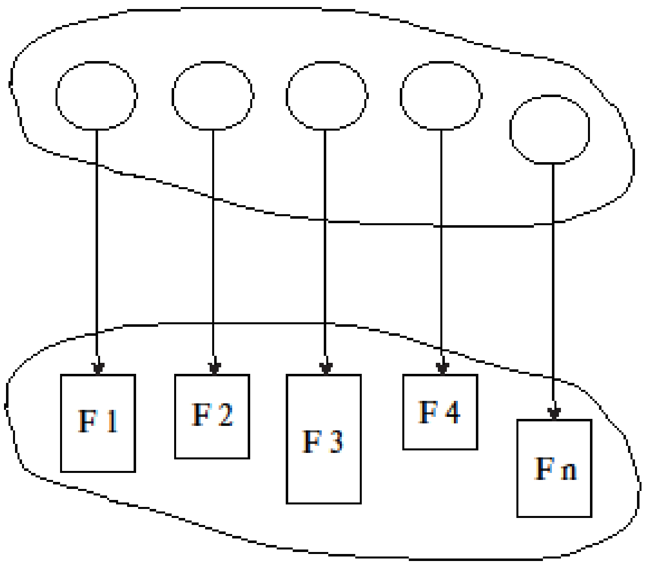


Types of File Systems

- We mostly talk of general-purpose file systems
- But systems frequently have may file systems, some general- and some special- purpose
- Consider Solaris has
  - tmpfs – memory-based volatile FS for fast, temporary I/O
  - objfs – interface into kernel memory to get kernel symbols for debugging
  - ctfs – contract file system for managing daemons
  - lofs – loopback file system allows one FS to be accessed in place of another
  - procfs – kernel interface to process structures
  - ufs, zfs – general purpose file systems

Directory Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk

Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

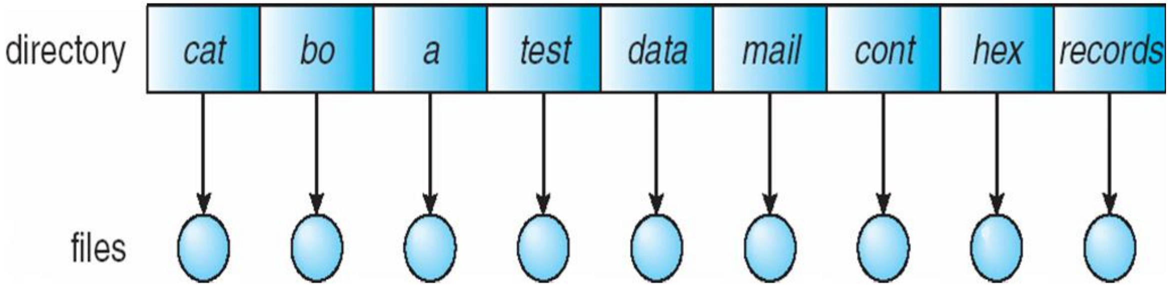
Directory Organization

The directory is organized logically to obtain

- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

Single-Level Directory

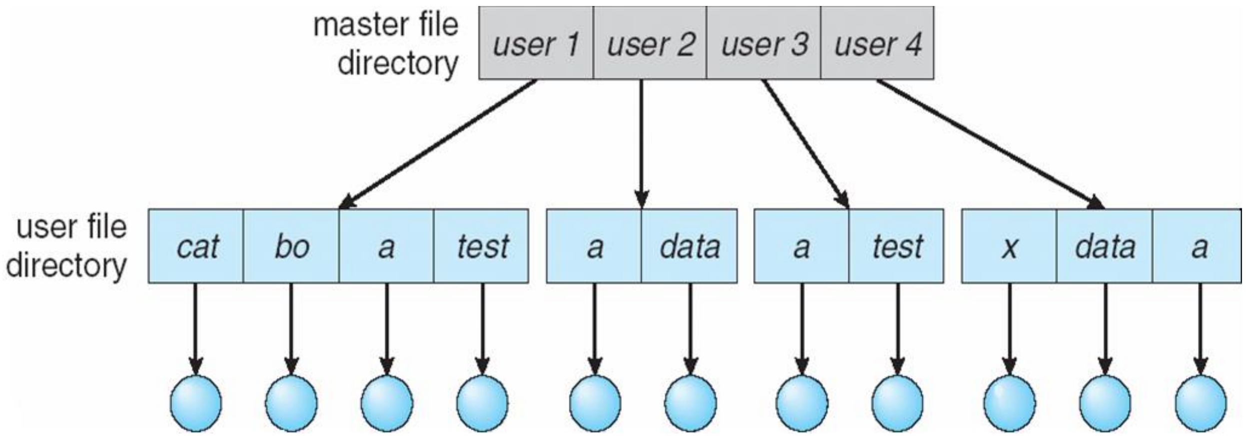
- A single directory for all users



- Naming problem
- Grouping problem

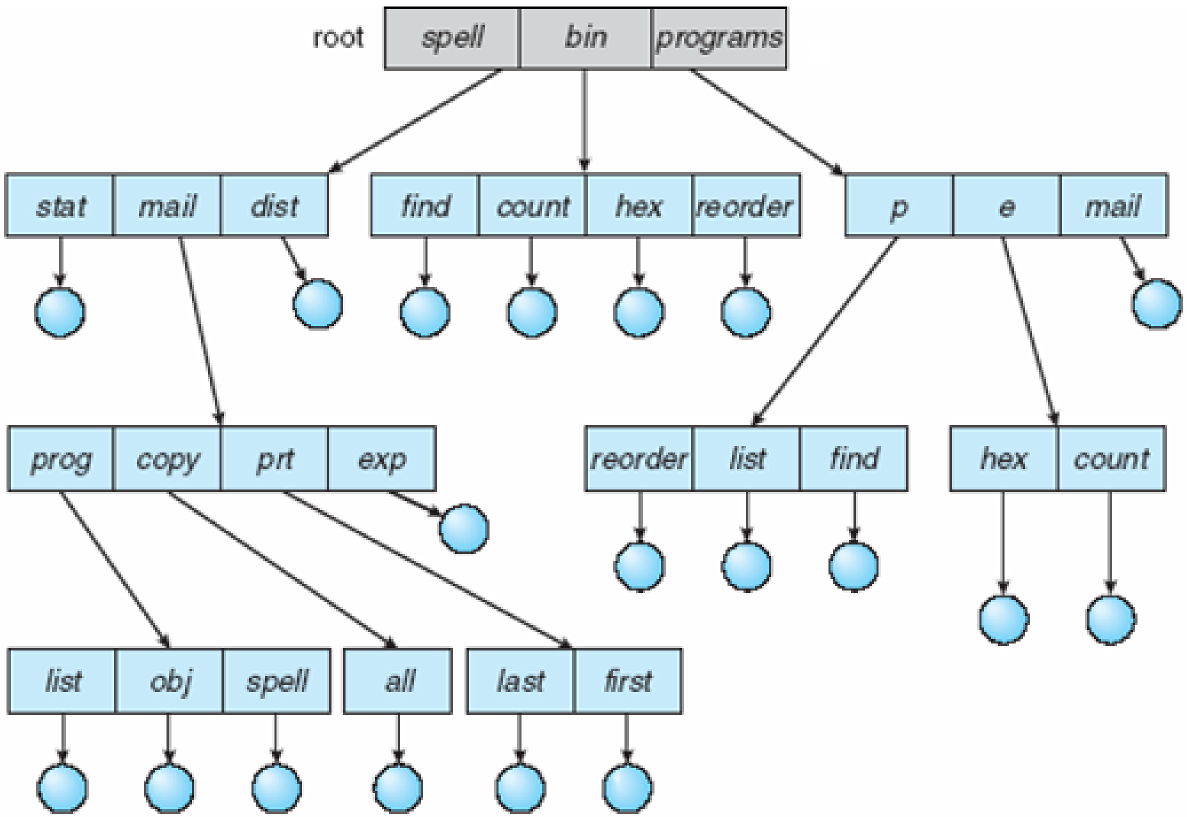
Two-Level Directory

- Separate directory for each user



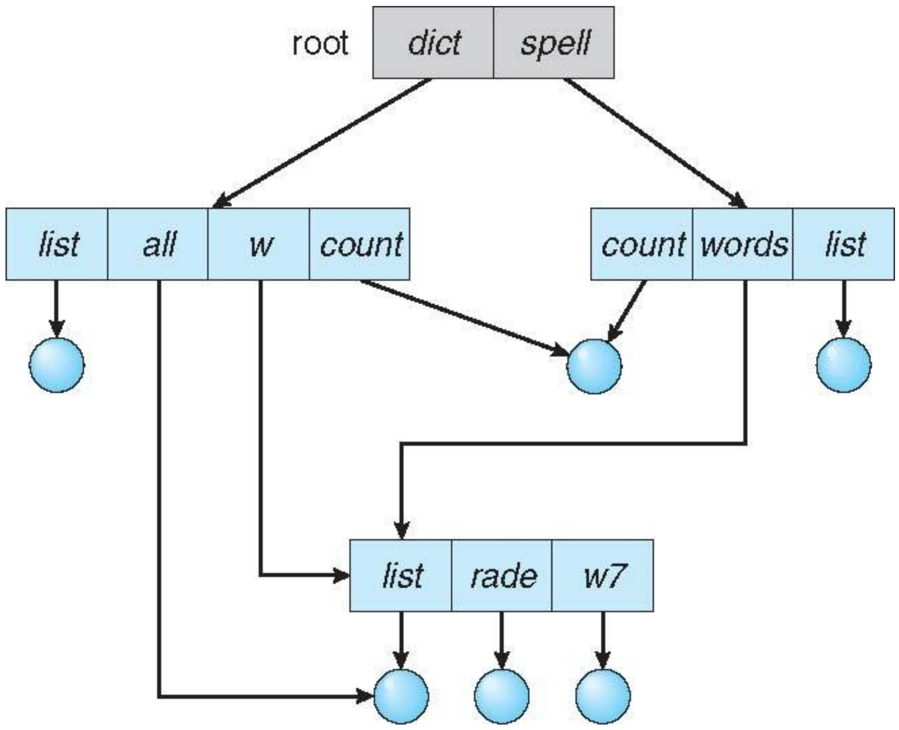
- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories



Acyclic-Graph Directories

- Have shared subdirectories and files
- Example

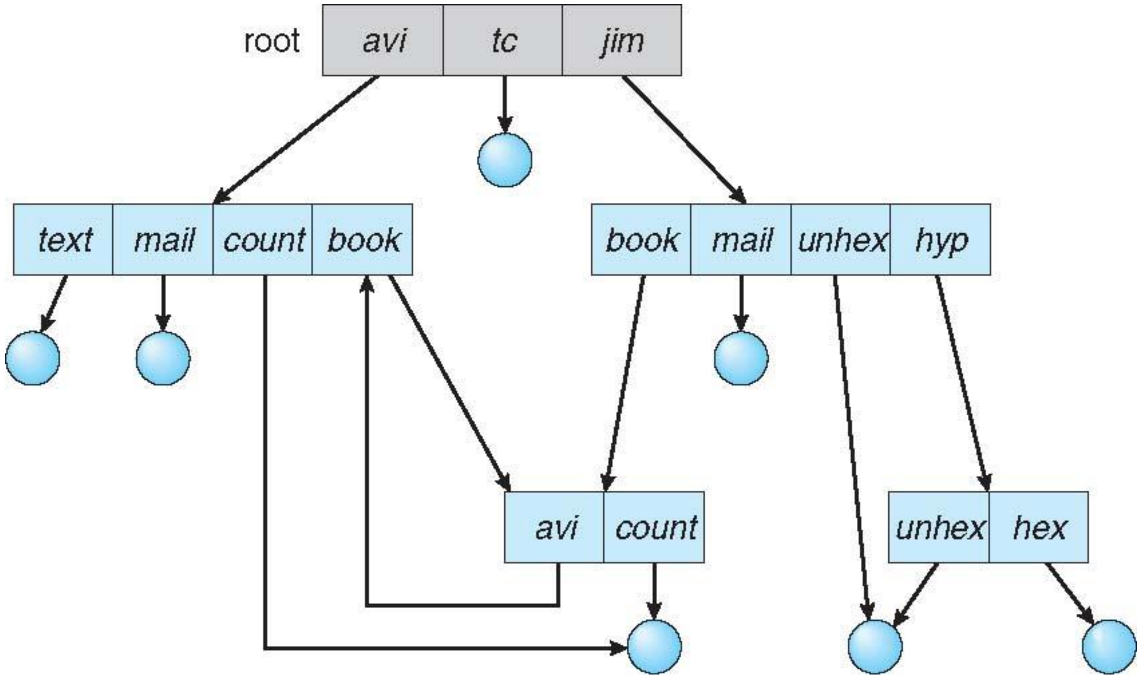


Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If ***dict*** deletes **w/*list*** $\Rightarrow$ dangling pointer

  Solutions:
  - Backpointers, so we can delete all pointers.
    - Variable size records a problem
  - Backpointers using a daisy chain organization
  - Entry-hold-count solution
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

General Graph Directory



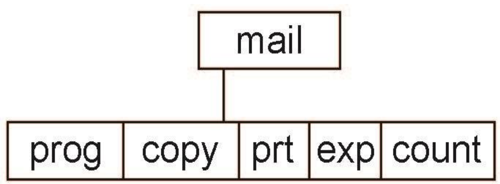
General Graph Directory (Cont.)

- How do we guarantee no cycles?
  - Allow only links to files not subdirectories
  - **Garbage collection**
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

Current Directory

- Can designate one of the directories as the current (working) directory
  - **`cd /spell/mail/prog`**
  - **`type list`**
- Creating and deleting a file is done in current directory
- Example of creating a new file
  - If in current directory is **`/mail`**
  - The command

    mkdir <dir-name>
  - Results in:



  - Deleting “mail” $\Rightarrow$ deleting the entire subtree rooted by “mail”

Protection

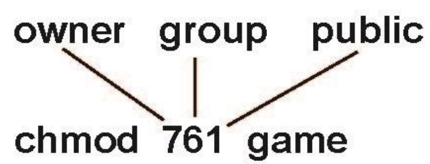
- File owner/creator should be able to control:
  - What can be done
  - By whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

Access Lists and Groups in Unix

- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

			RWX
a) **owner access**	7	⇒	1 1 1
			RWX
b) **group access**	6	⇒	1 1 0
			RWX
c) **public access**	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a file (say *game*) or subdirectory, define an appropriate access.



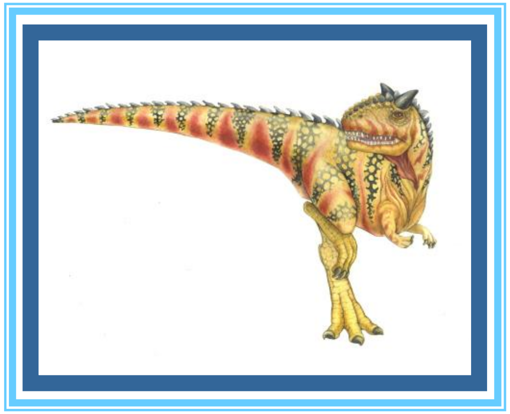
- Attach a group to a file

```
chgrp     G     game
```

A Sample UNIX Directory Listing

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

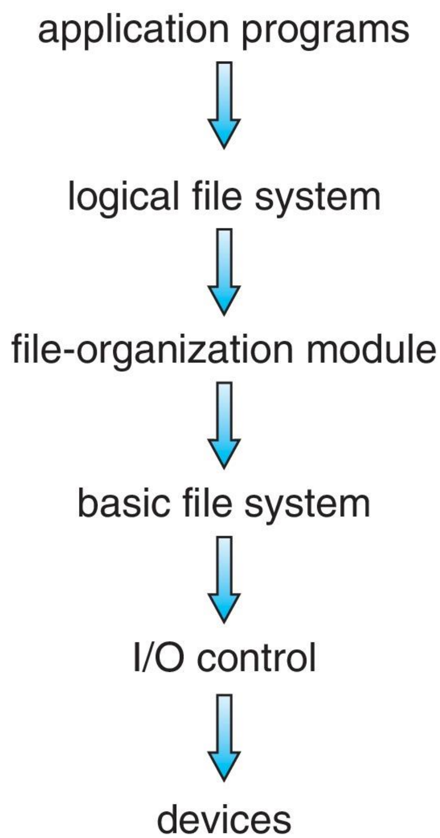
Chapter 14: File System Implementation



File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block** (**FCB**) – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers

Layered File System



File System Layers

- **Device drivers** manage I/O devices at the I/O control layer

  Given commands like

  read drive1, cylinder 72, track 2, sector 10, into memory location 1060

  Outputs low-level hardware specific commands to hardware controller

- **Basic file system** given command like "retrieve block 123" translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
- Translates logical block # to physical block #
- Manages free space, disk allocation

File System Layers (Cont.)

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Logical layers can be implemented by any coding method according to OS designer

File System Layers (Cont.)

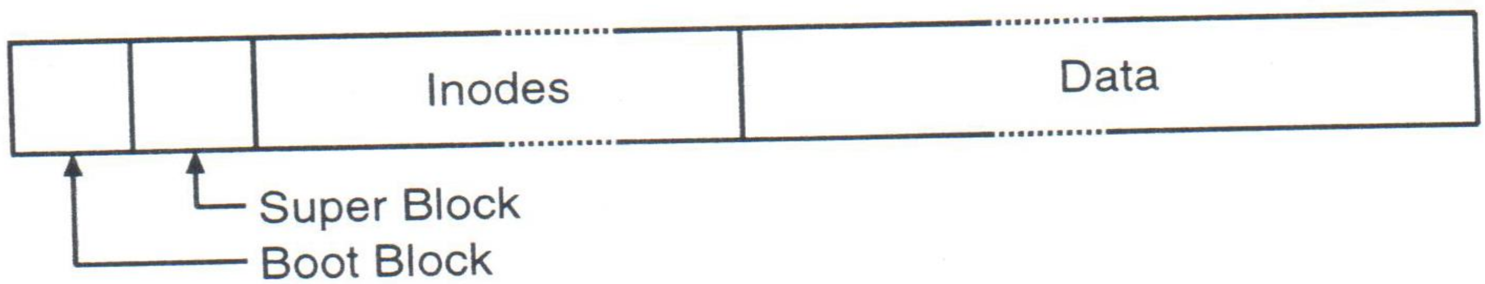
- Many file systems, sometimes many within an operating system
  - Each with its own format:
  - CD-ROM is ISO 9660;
  - Unix has **UFS**, FFS;
  - Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray,
  - Linux has more than 130 types, with **extended file system** ext3 and ext4 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

File-System Operations

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block** (**superblock**, **master file table**) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table

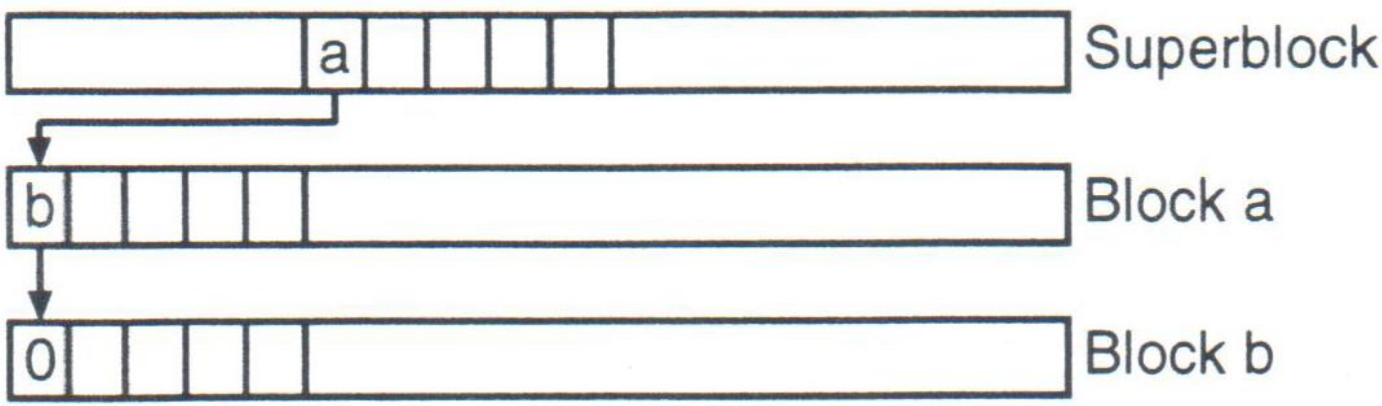
Unix System V

- primul bloc al sistemului de fisiere rezervat codului de boot (chiar daca nu e folosit, rezervarea pastreaza numerotarea restului blocurilor din sistemul de fisiere)
- superblock-ul contine:
  - dimensiunea in blocuri a sistemului de fisiere si a listei de inode-uri
  - nr de blocuri si inode-uri libere
  - lista partiala cu blocuri libere
  - lista partiala cu inode-uri libere
  - in general, cele 2 liste sunt prea mari pt a fi tinute integral in superblock



Unix System V (cont'd)

- uzual, superblock-ul contine vectori care identifica primele inode-uri, respectiv blocuri de date libere
  - cand lista de inode-uri libere e goala, kernelul scaneaza discul pt a gasi inode-uri libere si a reumple lista partiala
  - pt lista de blocuri libere nu se procedeaza similar (pt ca nu se poate decide daca un bloc de date e liber sau nu bazat doar pe continutul sau) => se folosesc liste de blocuri libere inlantuite folosind primul element al listei partiale de blocuri libere din superblock



File Control Block (FCB)

- OS maintains **FCB** per file, which contains many details about the file
  - Typically, inode number, permissions, size, dates
  - Example

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Inode-uri Unix

- fiecare fisier e reprezentat prin *i-node* (*index node*)
- lista de inode-uri de dupa superblock are lungime fixa si e alocata la crearea sistemului de fisiere (“formatarea discului”)
- inode-ul contine:
  - tipul fisierului
  - drepturile asupra fisierului
  - nr de *link*-uri ale fisierului (i.e., nr de intrari in directoare diferite)
  - dimensiunea fisierului in octeti
  - timpul ultimului acces
  - timpul ultimei modificari a fisierului
  - timpul ultimei schimbari a inode-ului
  - vectorul de adrese de disc

Vectorul de adrese de disc

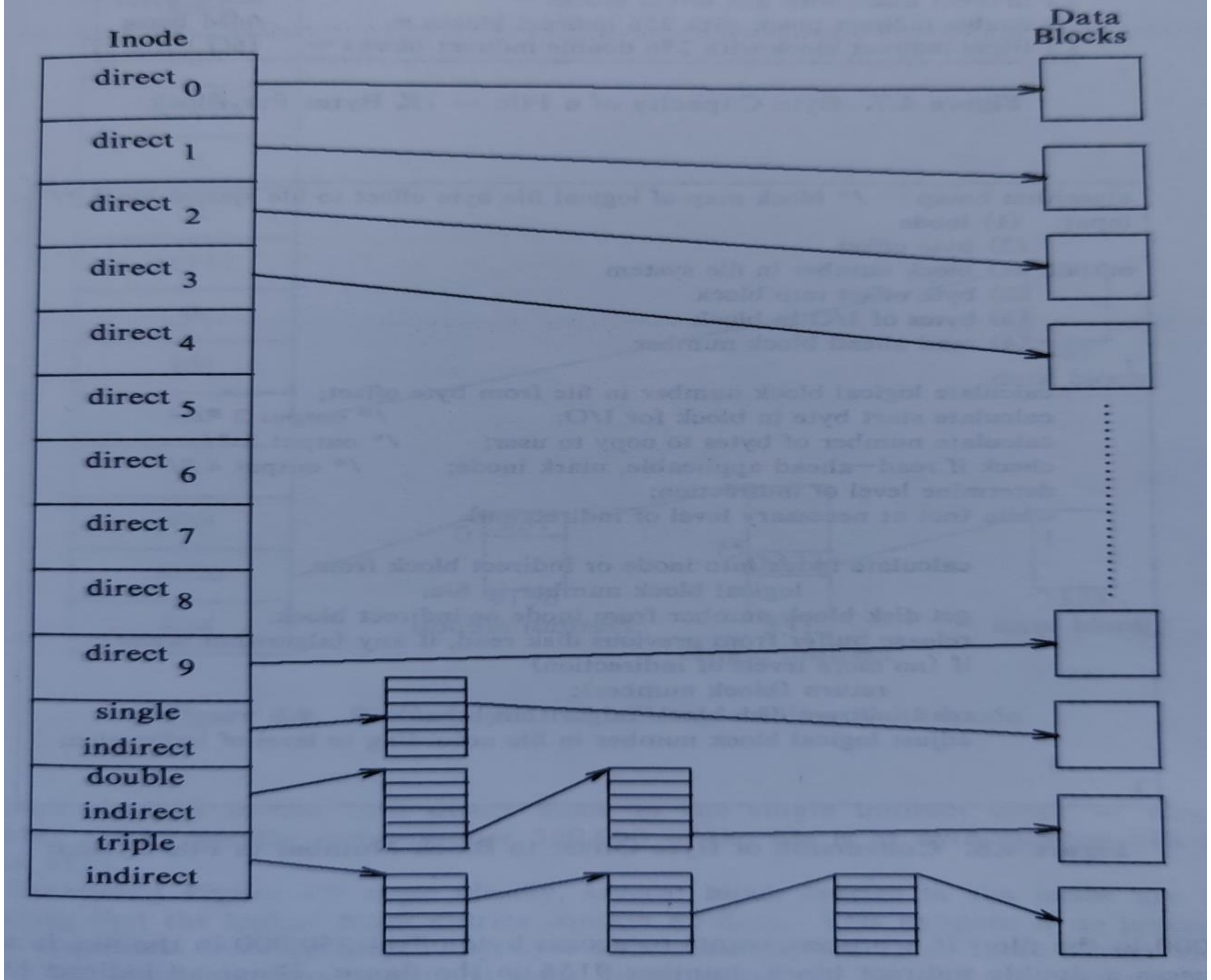


Figure 4.6. Direct and Indirect Blocks in Inode

Calcul dimensiune maxima fisier

- dimensiune bloc disk = 1KB
- adresa bloc disk reprezentata pe 4 octeti

=> 256 adrese indirecte intr-un bloc

- limita impusa de definitia inode 4GB (dimensiunea fisierului reprezentata pe 32 biti)

=> teoretic, fisierele pot avea ceva mai mult de 16GB, practic <= 4GB

10 direct blocks with 1K bytes each =	10K bytes
1 indirect block with 256 direct blocks =	256K bytes
1 double indirect block with 256 indirect blocks =	64M bytes
1 triple indirect block with 256 double indirect blocks =	16G bytes

Figure 4.7. Byte Capacity of a File – 1K Bytes Per Block

Exemplu

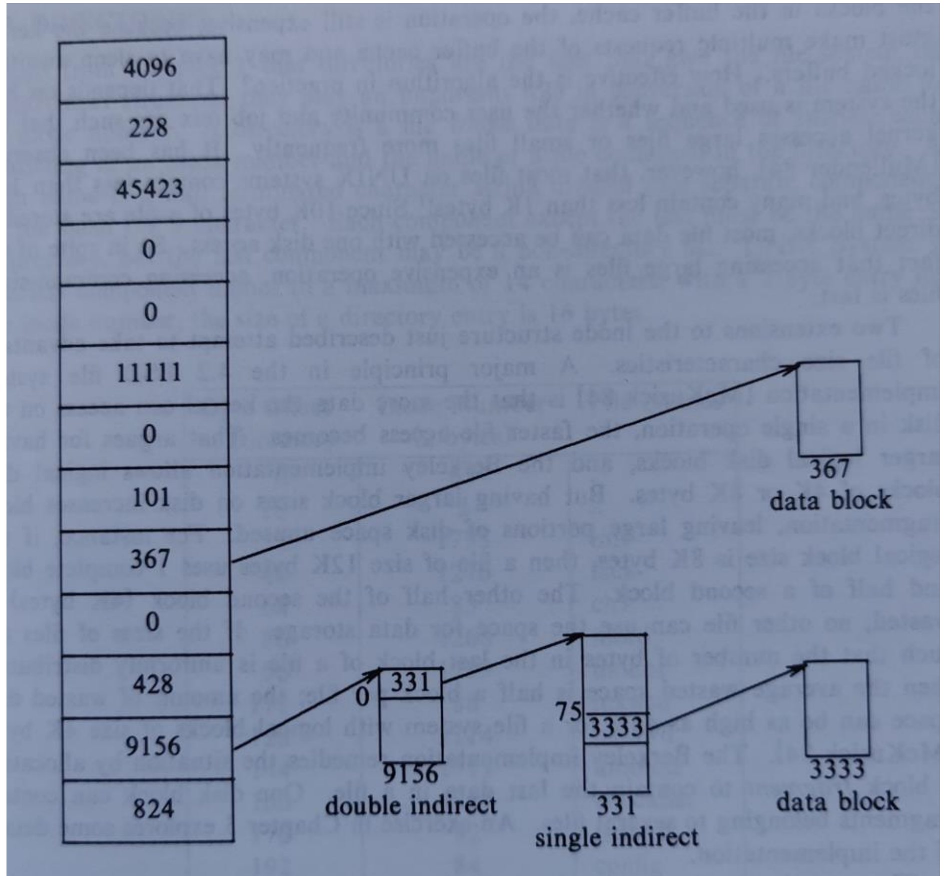


Figure 4.9. Block Layout of a Sample File and its Inode

Virtual Filesystem Switch (VFS)

- inainte de a fi folosite, sistemele de fisiere se instaleza in ierarhia de directoare prin operatia *mount*
  - discul formatat cu un anumit sistem de fisiere se asociaza ("incaleca") cu un subdirector (numit *mount point*) din ierarhia de directoare

  Ex:

  mount –t ext4 /dev/sda1 /mnt

  mount –t nfs 192:168.0.45:/home /home

  mount –t iso9660 –o loop ubuntu.iso /dev/cdrom

- in sistemele de operare moderne, multiple sisteme de fisiere pot fi instalate in aceeasi ierarhie de directoare prin intermediul VFS

Virtual Filesystem Switch (VFS)

- VFS defineste abstractia de *v-node*
  - la instalarea unui sistem de fisiere, kernelul aloca o structura VFS care inregistreaza in campul *v_op* pointeri catre functiile necesare implementarii apelurilor de sistem (*open*, *close*, *read*, *write*, etc) pt acel tip de sistem de fisiere
  - de fapt, un fisier deschis e reprezentat in kernel de un *v-node* (nu *inode*), eventual proaspat alocat, daca fisierul nu exista
  - restul kernelului invoca functiile din *v_op* prin functii generice care redirectioneaza executia catre operatiile specifice sistemului de fisiere respective, eg

    VOP_OPEN(vnode *vp) { *(vp->v_op->vop_open)(vp, …) ; }

Directoare Unix

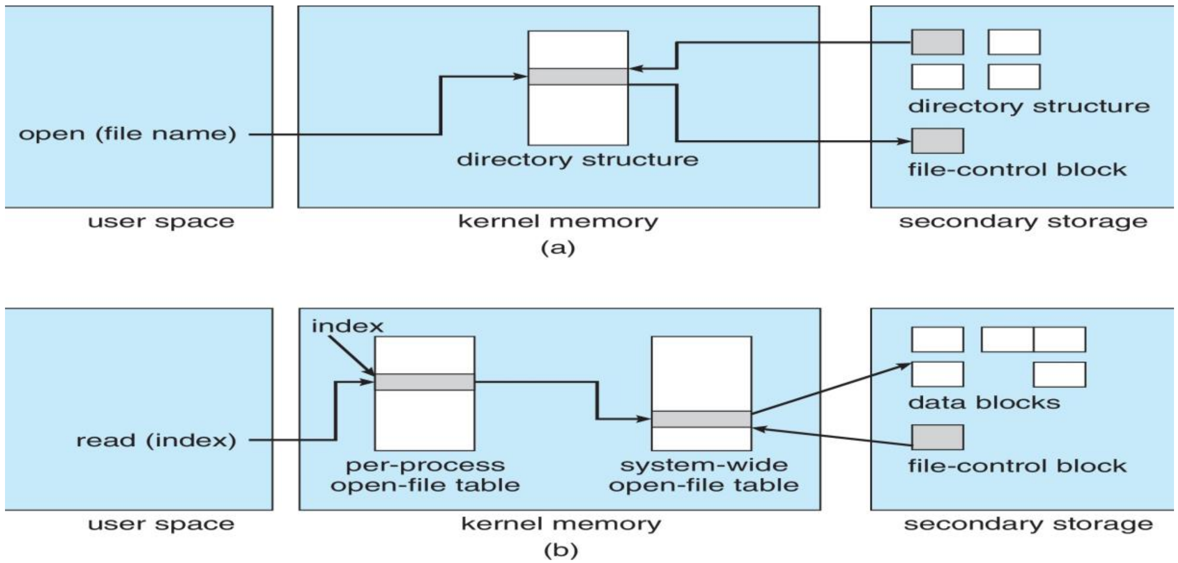
- implementate ca perechi (nr inode, nume)
- contin automat intrari pt. "." si ".."
- *link*
  - intrari in directoare diferite care refera acelasi inode
  - *hard*: intrari pe acelasi sistem de fisiere (disc)
  - *soft*: intrari pe diferite sisteme de fisiere (discuri), de fapt fisiere speciale in care continutul fisierului este numele fisierului linkat
- *metadate*
  - blocuri folosite pt a stoca informatii administrative despre sistemul de fisiere si NU date obisnuite din fisiere
  - ex: superblock, free list-uri, inode, blocuri indirecte, etc

In-Memory File System Structures

- **Mount table** storing file system mounts, mount points, file system types
- **System-wide open-file table** contains a copy of the FCB of each file and other info
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info

In-Memory File System Structures (Cont.)

- Figure 12-3(a) refers to opening a file
- Figure 12-3(b) refers to reading a file



Allocation Method

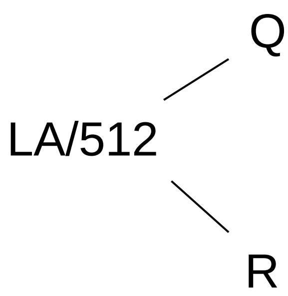
- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous
  - Linked
  - File Allocation Table (FAT)

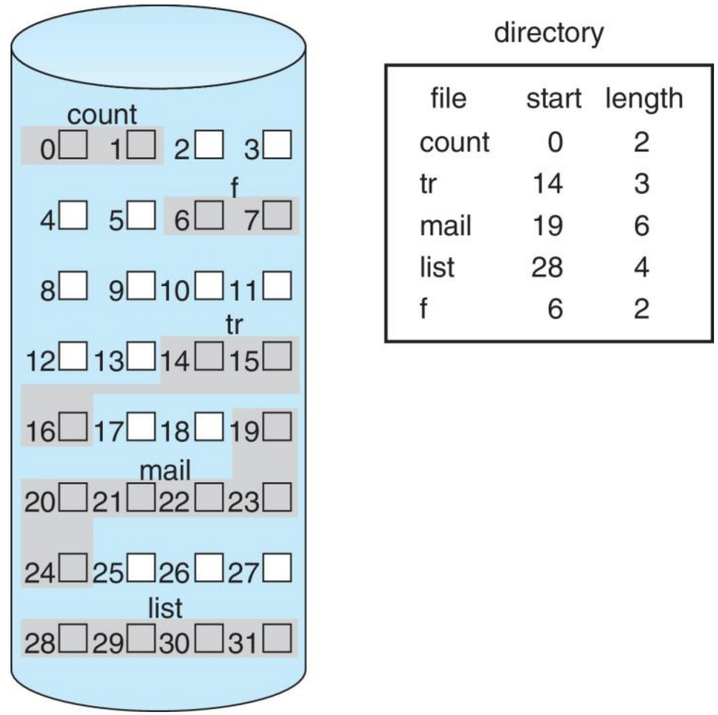
Contiguous Allocation Method

- An allocation method refers to how disk blocks are allocated for files:
- Each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include:
    - Finding space on the disk for a file,
    - Knowing file size,
    - External fragmentation, need for **compaction off-line** (**downtime**) or **on-line**

Contiguous Allocation (Cont.)

- Mapping from logical to physical (block size =512 bytes)





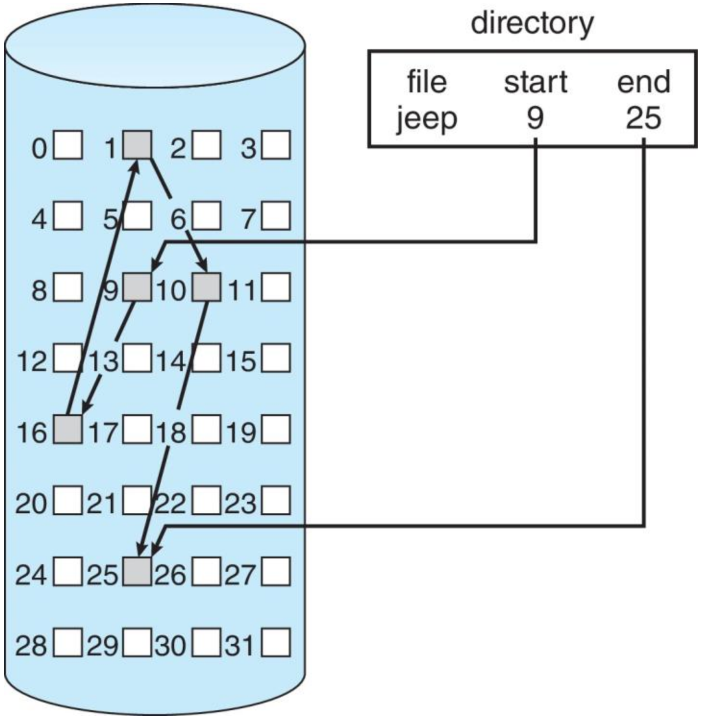
- Block to be accessed = starting address + Q
- Displacement into block = R

Linked Allocation

- Each file is a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block
- No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks

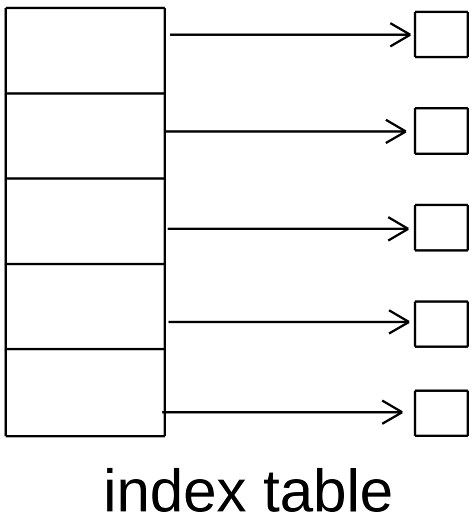
Linked Allocation Example

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- Scheme

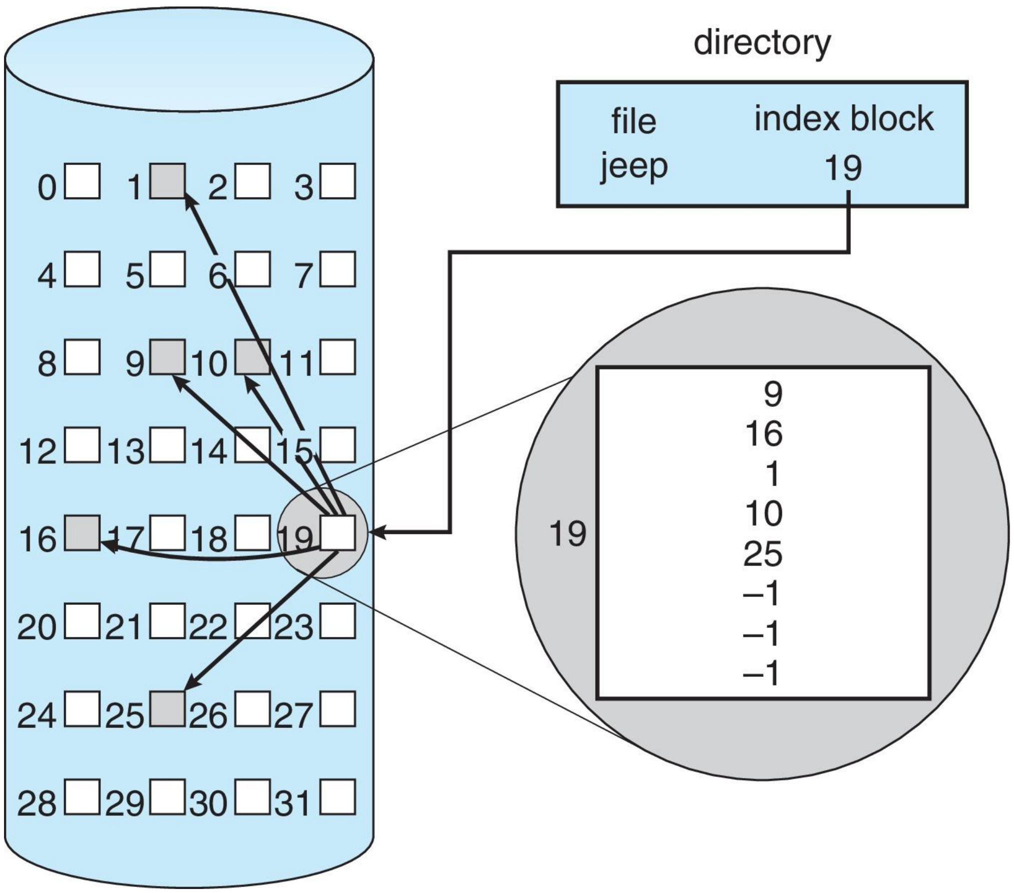


Indexed Allocation Method

- Each file has its own **index block**(s) of pointers to its data blocks
- Logical view



Example of Indexed Allocation



Performance

- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation
  - Select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead
- For NVM, no disk head so different algorithms and optimizations needed
  - Using old algorithm uses many CPU cycles trying to avoid non-existent head movement
  - Goal is to reduce CPU cycles and overall path needed for I/O

Efficiency and Performance

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures

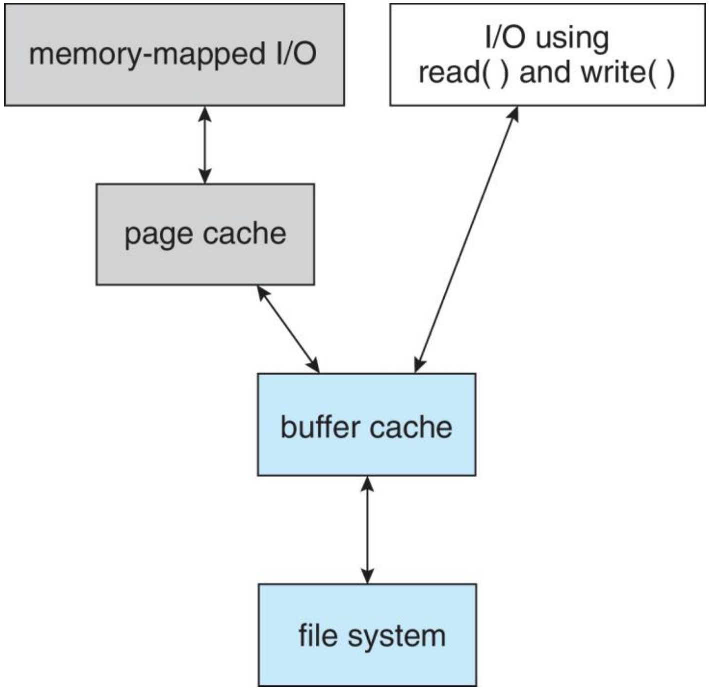
Efficiency and Performance (Cont.)

- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes

Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

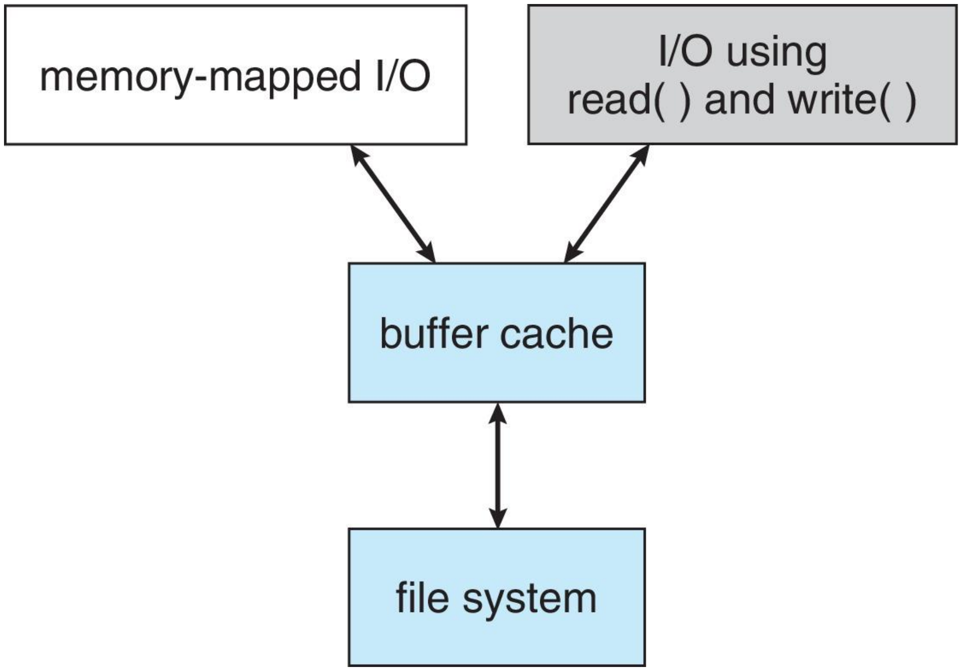
I/O Without a Unified Buffer Cache



Unified Buffer Cache

- A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache



Recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup

Log Structured File Systems

- **Log structured** (or **journaling**) file systems record each metadata update to the file system as a **transaction**
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

BSD-LFS

- adauga la sfarsitul log-ului nu doar actualizarile metadatelor, ci si pe cele ale datelor
- log-ul e circular si e compus din segmente de dimensiune fixa (cca 512KB)
- cand se ajunge la sfarsitul discului e nevoie de un garbage collector (*cleaner*) care recupereaza blocurile vechi din segmente si creeaza segmente curate
- header segmente
  - sume de control
  - adrese de disc pt fiecare inode din segment
  - nr de inode
  - versiunea de nr de inode
  - numerele de blocuri logice pt fiecare fisier din segment
  - timpul de creare
  - flag-uri

Operare

- pe masura ce se scrie in fisiere, se aduna blocurile modificate pana cand se umple un segment
- blocurile sunt sortate dupa nr de bloc logic din fisier
- se asigneaza adrese de disc blocurilor fisierului
- adresele de blocuri din inode sunt actualizate pt a reflecta noua asignare pe disc
- inode-ul si blocurile de metadate sunt adaugate la segment (append)
- segmentul e scris pe disc in log

Efectele modului de operare

- blocurile sunt contigue pe disc
- inode-ul si metadatele (blocuri de indirectare, de ex) sunt citite simultan cu datele fara cautari suplimentare pe disc
- cf principiului localitatii de referinta => probabilitate mare ca fisierele unui director sa fie in acelasi segment cu directorul lor, inode-urile si blocurile lor indirecte => context bine localizat in cache

Localizarea fisierelor

- scanare ierarhie de directoare pt mapare nume fisier -> inode
- tabela *ifile* de inode-uri cu adresele de disc corespunzatoare
- tabela de utilizare a segmentelor: mentine nr de bytes de date “live” din fiecare segment (informatie necesara cleaner-ului)
- cele doua tabele sunt scrise periodic pe disc (checkpoints)

Recuperare dupa crash

- se localizeaza ultimul checkpoint
- se reiau actualizarile de la sfarsitul ultimului log
- verificarea completa a sistemului de fisiere se poate face cu un program de tip *fsck* in background, dupa pornirea sistemului

Operare cleaner

- compacteaza segmentele pe disc
- elimina blocurile vechi ca urmare a stergerii lor sau a aparitiei unor versiuni noi ale lor adaugate la sfarsitul logului
- cleaner-ul se poate implementa ca un proces un spatiul utilizator cu ajutorul unor apeluri sistem specifice

End of Chapter 14