Disk Hardware

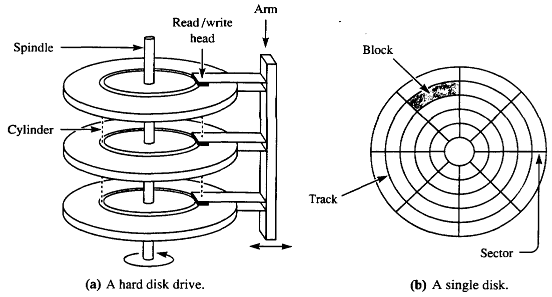


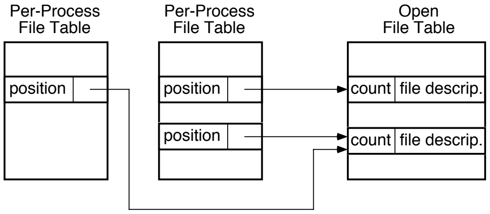
Diagram from *Computer Science*, Volume 2, J. Stanley Warford, Heath, 1991.

- Arm can move in and out
  - Read / write head can access a ring of data as the disk rotates
- Disk consists of one or more *platters*
  - Each platter is divided into rings of data, called *tracks*, and each track is divided into *sectors*
  - One particular platter, track, and sector is called a *block*

Data Structures for Files

- Every file is described by a *file descriptor*, which may contain (varies with OS):
  - Type
  - Size
  - Access times — when created, last accessed, last modified
  - Owner, group
  - Access permissions — read, write, etc.
  - Link count — number of directories that contain this file
  - Blocks where file is located on disk
- Not included:
  - Name of file

OS Data Structures for Files


Per-Process File Table

position

Per-Process File Table

position

position

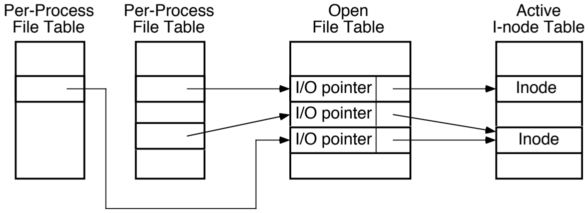
Open File Table

count file descrip.

count file descrip.

- *Open file table* (one, belongs to OS)
  - Lists all open files
  - Each entry contains:
    - A *file descriptor*
    - Open count — number of processes that have the file open
- *Per-process file table* (many)
  - List all open files for that process
  - Each entry contains:
    - Pointer to entry in open file table
    - Current position (offset) in file

UNIX Data Structures for Files


Per-Process File Table

Per-Process File Table

Open File Table

I/O pointer

I/O pointer

I/O pointer

Active I-node Table

Inode

Inode

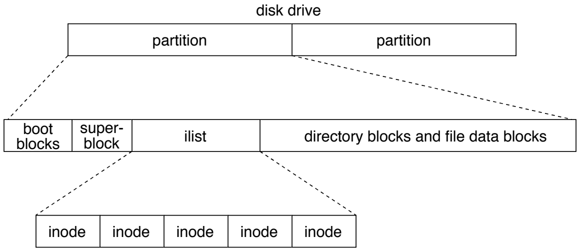
- *Active Inode table* (one, belongs to OS)
  - Lists all active *inodes* (file descriptors)
- *Open file table* (one, belongs to OS)
  - Each entry contains:
    - Pointer to entry in active inode table
    - Current position (offset) in file
- *Per-process file table* (many)
  - Each entry contains:
    - Pointer to entry in open file table

UNIX File System

- A file descriptor (*inode*) represents a file
  - All *inodes* are stored on the disk in a fixed-size array called the *ilist*
    - The size of the ilist array is determined when the disk is initialized
    - The index of a file descriptor in the array is called its *inode number*, or *inumber*
  - Inodes for active files are also cached in memory in the *active inode table*
- A UNIX disk may be divided into *partitions*, each of which contains:
  - Blocks for storing directories and files
  - Blocks for storing the ilist
    - Inodes corresponding to files
    - Some special inodes
      - Boot block — code for booting the system
      - Super block — size of disk, number of free blocks, list of free blocks, size of ilist, number of free inodes in ilist, etc.

UNIX File System (cont.)

- High-level view:



- Low-level view:

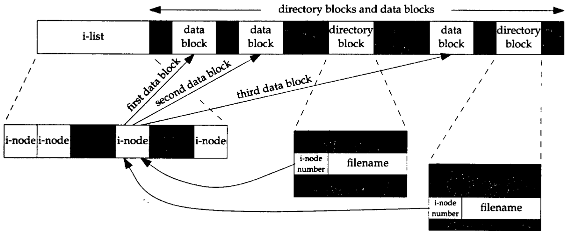


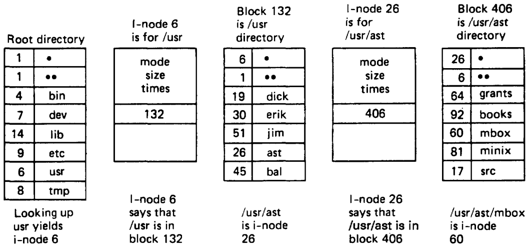
Diagram from *Advanced Programming in the UNIX Environment*, W. Richard Stevens, Addison Wesley, 1992.

Working with Directories in UNIX

(Think about how this compares to Windows or to the Macintosh OS)

- UNIX keeps track of the inode number of current working directory for each process; directory searches begin there
- However, a file can also be specified as the full pathname from the “root”
  - If filename begins with “ / ”, start at root of the file system tree (inode 2)
- Other characters have special meaning:
  - If filename begins with “ ~ ”, start at the user’s home directory
  - If filename begins with “ . ”, start at the current working directory
  - If filename begins with “ .. ”, start at the parent directory

Working with Directories (Lookup)


Root directory
I-node 6 is for /usr
mode size times
132
Block 132 is /usr directory
I-node 26 is for /usr/ast
mode size times
406
Block 406 is /usr/ast directory
Looking up usr yields i-node 6
I-node 6 says that /usr is in block 132
/usr/ast is i-node 26
I-node 26 says that /usr/ast is in block 406
/usr/ast/mbox is i-node 60

Root directory

1	•
1	••
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Block 132 is /usr directory

6	•
1	••
19	dick
30	erik
51	jim
26	ast
45	bal

Block 406 is /usr/ast directory

26	•
6	••
64	grants
92	books
60	mbox
81	minix
17	src

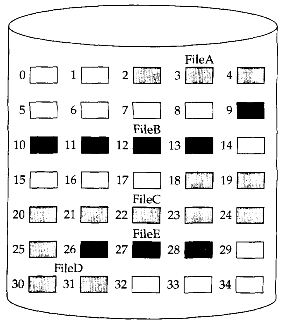
Fig. 4-16. The steps in looking up */usr/ast/mbox*.

- A directory is a table of entries:
  - 2 bytes — inumber
  - 14 bytes — file name (improved in BSD 4.2 and later)
- Search to find the file begins with either root, or the current working directory
  - Inode 2 points to the root directory (“ / ”)
  - Example above shows lookup of /usr/ast/mbox

Working with Directories (Links) in UNIX

- UNIX supports “links” — two directories containing the same file
  - Think of “shortcuts” in Windows, or “aliases” in the Macintosh OS
- Hard links (“ ln *target_file directory* ”)
  - Specified directory refers to the target file
    - Both directories point to same inode
- Soft / symbolic links (“ ln –s *target_file directory* ”)
  - Adds a pointer to the target file (or target directory) from the specified directory
    - Special bit is set in inode, and the file just contains the name of the file it’s linked to
    - View symbolic links with “ls –F” and “ls –l”
  - Can link across disk drives
  - Similar to linking in Windows / Mac OS

Organization of Files (Contiguous Allocation)



File Allocation Table

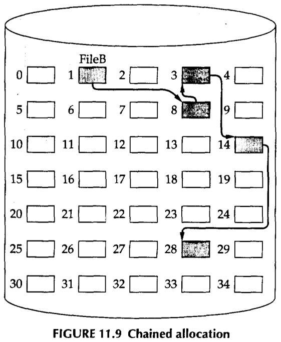
File Name	Start Block	Length
FileA	2	3
FileB	9	5
FileC	18	8
FileD	30	2
FileE	26	3

FIGURE 11.7 Contiguous file allocation

Diagram from *Operating Systems*, William Stallings, Prentice Hall, 1995.

- OS keeps an ordered list of free blocks
  - Allocates contiguous groups of blocks when it creates a file
  - File descriptor must store start block and length of file

Organization of Files (Linked / Chained Allocation)



File Allocation Table

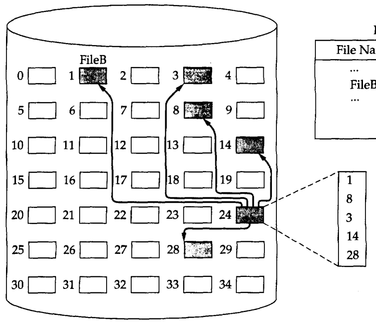
File Name	Start Block	Length
...	...	...
FileB	1	5
...	...	...

FIGURE 11.9 Chained allocation

Diagram from *Operating Systems*, William Stallings, Prentice Hall, 1995.

- OS keeps an ordered list of free blocks
  - File descriptor stores pointer to first block
  - Each block stores pointer to next block
- File-Allocation Table variation keeps all pointers in one table

Organization of Files (Indexed Allocation)



File Allocation Table

File Name	Index Block
...	...
FileB	24
...	...

FIGURE 11.11 Indexed allocation with block portions

Diagram from *Operating Systems*, William Stallings, Prentice Hall, 1995.

- OS keeps a list of free blocks
  - OS allocates an array (called the index block) to hold pointers to all the blocks used by the file
  - Allocates blocks only on demand
  - File descriptor points to this array

Organization of Files (Multilevel Indexed Allocation)

- Used in UNIX (numbers below are for traditional UNIX, BSD UNIX 4.1)
- Each inode (file descriptor) contains 13 *block pointers*
  - First 10 pointers point to data blocks (each 512 bytes long) of a file
    - If the file is bigger than 10 blocks (5,120 bytes), the 11th pointer points to a *single indirect block*, which contains 128 pointers to 128 more data blocks (can support files up to 70,656 bytes)
      - If the file is bigger than that, the 12th pointer points to a *double indirect block*, which contains 128 pointers to 128 more single indirect blocks (can support files up to 8,459,264 bytes)
        - If the file is bigger than that, the 13th pointer points to a *triple indirect block*, which contains 128 pointers to 128 more double indirect blocks
  - Max file size is 1,082,201,087 bytes

Organization of Files (Multilevel Indexed Allocation) (cont.)

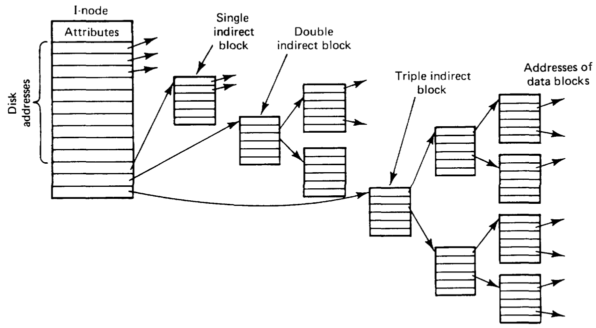


Diagram from *Modern Operating Systems*, Andrew Tanenbaum, Prentice Hall, 1992.

- BSD UNIX 4.2, 4.3:
  - Maximum block size is 4096 bytes
  - Inode contains 14 block pointers
    - 12 to data
    - 13 to single indirect block containing 1024 pointers, 14 to double indirect block…
  - Max file size is 2^{32} bytes

Improving Performance with Good Block Management

- OS usually keeps track of free blocks on the disk using a *bit map*
  - A bit map is just an array of bits
    - 1 means the block is free,
    - 0 means the block is allocated to a file
  - For a 12 GB drive, there are about 3,070,000 4KB blocks, so a bit map takes up 384 KB (usually kept in memory)
- Try to allocate the next block of the file close to the previous block
  - Works well if disk isn't full
  - If disk is full, this is doesn't work well
    - Solution — keep some space (about 10% of the disk) in reserve, and don't tell users; never let disk get more than 90% full
    - With multiple platters / surfaces, there are many possibilities (one surface is as good as another), so the block can usually be allocated close to the previous one