1 THE FILE SYSTEM

Objectives

- Provide *long term*, *on-line*, *non-volatile* storage (for programs, data, text etc)
- Allow information sharing (editors, compiler, utilities and databases such as airline reservation information)
- Convenient means of organising and accessing data (use of symbolic names, automatic back-up etc)

Achieved via a filing system, which :

- Manages disk space (usually as blocks)
- A mapping of symbolic names to physical address (like directories, for instance what does /homes/scmsomeone map to in terms of disk address)
- Protection :against system failure like power loss etc (backup, recovery - general archiving of data)
  : against unauthorised user access (allowing sharing between co-operating users, BUT protection from others)

Can consider the smallest unit of such a system to be a **FILE** : *a named collection of data of arbitrary size* on which various operations can be performed)

2 Filing System functions

- *Create* : Allocate space and enter name+location into directory
- *Delete* : Deallocate space, invalidate+remove directory entry
- *Open* : Search directory for file, check access permissions, set pointers to file
- *Close* : Remove pointers to file
- *Read* : Access file and update current position pointers
- *Write* : Access file and update pointers
- *Reposition* : Set current position to a specific (new) value
- *Truncate* : Erase contents, but maintain other attributes

Run the command on unix **df -a** :

```
Filesystem              kbytes    used    avail capacity  Mounted on
/dev/dsk/c0t1d0s0       146967   57801    74476    44%    /
/dev/dsk/c0t1d0s6       105039   82418    12121    88%    /usr
/proc                        0       0        0     0%    /proc
fd                           0       0        0     0%    /dev/fd
/dev/dsk/c0t1d0s4       106047   32236    63211    34%    /var
/dev/dsk/c0t1d0s3       139895  110284    15631    88%    /usr/openwin
swap                    196800    8748   188052     5%    /tmp
auto_home                    0       0        0     0%    /home
auto_nfs                     0       0        0     0%    /nfs
sentinel:vold(pid538)  146967   57801    74476     44%     /vol
```

3 The Layered File System

- *Applications Programs* : High level language interface to files :
  Common access methods : *Sequential*, *Random* and *Indexed Sequential*
- *Logical File System (Directory Level)* : Mapping from symbolic names to file locations (i.e. supports directory system) ... more on this later
- *File Organisation Module* : Allocation and management of file space (free space etc)
- *Basic or Flat File System* : Basic file access facilities (read, write a block of file via a single level - flat directory structure). Block address to disk address translation (e.g. drive, cylinder, surface, sector)
- *I/O Control* : Device drivers and interrupt handlers (transfer of physical blocks of data between disk and memory)
- *Devices* : Disks on which files are stored

Archive Services : run as application programs

How can files be accessed from the programs (application) by the user ?

4 File Access Methods

- *Sequential* : Blocks processed *in order*, one after the other
  Read operation reads the next block and updates pointer
  Write appends to the end of file
  Rewind / Forward - to skip or reach certain blocks (difficult)!
- *Direct / Random* : File viewed as a numbered sequence (array) of blocks
  Blocks can be read in random (any) order
- *Indexed Sequential* : Arrangement of records in a *logical sequence* - according to key contained in record
  An *index* from key to record
  Can access sequentially in *key order* or *random*
- *Partitioned* : File of *sequential sub-files* - for libraries etc

Space allocation USUALLY NOT contiguous - logical blocks translated to physical disk blocks

5 File Attributes

Any *subset* may be held within a particular directory structure :

- **Basic Information** :
  File Name : symbolic (text) name : unique to directory
  File type : text, binary, executable or another directory
  File Organisation : sequential, random etc
  Creator : program which created it - and can be used to open it (Mac's double-click)
- **Address Information** :
  Volume : disk drive, partition of disk etc
  Start Address : cylinder, track, block number etc
  Size Used and Allocated
- **Access Control Information** :
  Owner : Who control file (often the creator - NOT always!)
  Authentication : password
  Permitted Operations : read, write, delete etc
- **Usage Information** :
  Creation date and time
  Last modify date and time
  Last read date and time
  Archive date and time
  Expiry date : when the file will be deleted
  Access activity counts : number of reads, writes (used to control file system by system managers)

```
-rw-------   1 scmofr      6549 Jan 21 19:27 conf
```

6 File Directory Structures

Maps symbolic file names to logical disk blocks (physical disk addresses) : e.g. file in your account *something.txt* to *drive 0, block 3 etc*

- **One level directory** (flat structure) e.g. CP/M, RT 11
- Usually simple system holding subset of file attributes - address and access control information
- In Unix a flat file system maps index (i-node) number of addresses (mapping of a tree structure into a flat structure before performing the physical mapping)

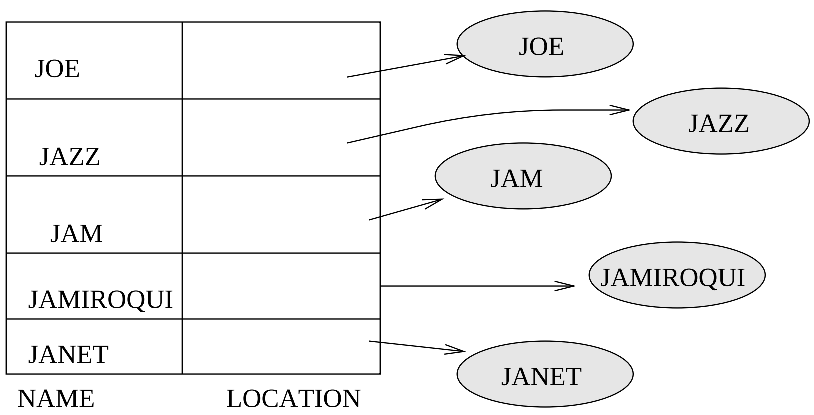

Figure 1: Flat directory structure

- **Two Level Directory Structure** : Minimum requirement for multi-user systems
- Two levels of directories : a **Master Directory** containing one entry for every user of the system, and a **User File Directory** : with one entry per file

- User directories can get very long - and name clashes may occur (this is the equivalent of the one level directory structure above)

So what's the solution for having too many files ?
Better directory organisation dictates **a tree structure**

- **Multi level Directory Structure** (as in Unix, MS-DOS, Mac etc)

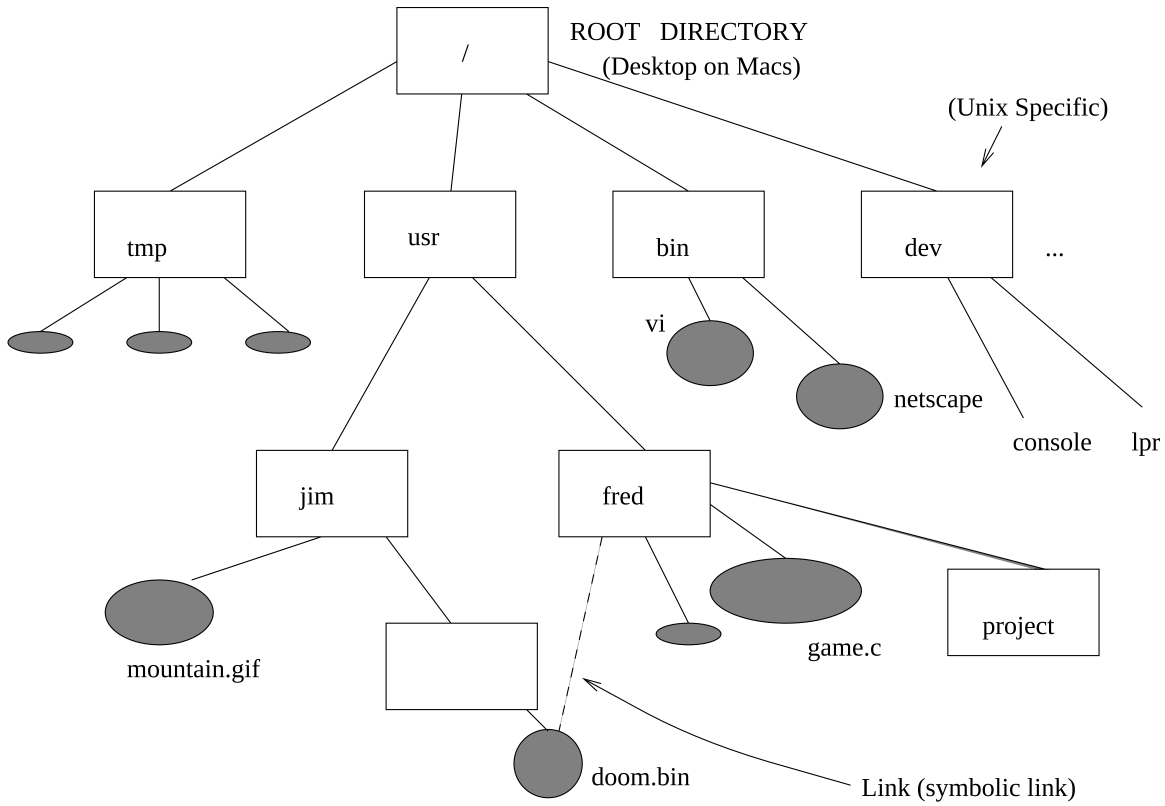

Figure 2: Multi Level directory structure

7 Directory Operations

Operations that can be performed on a directory include :

- *Open/Close* a directory
- *find a file* in directory using a pattern (wildcards etc)
- *Create/Delete* : how do we deal with non-empty directories ?
- *Link/Unlink* : An indirection across directory structure
- *Change directories* : Move across directory structures
- *List* : files present within a directory
- *Attributes* : Read attributes of a file or change them (protection information) - in Unix - the *sticky bit*

Need to traverse directory structure for *archiving* (backups), a useful command in unix :

```
find . -name 'something_nice' -print
```

or something more complex :
*Remove all files in your home directory named a.out or * .o that have not been accessed for a week:*

```
my_prompt% find $HOME \( -name a.out -o -name '*.o' \
   -atime +7 \ -exec rm {} \;
```

8 Links

Provide a reference to a directory in another part of the tree or a file in another directory - *allows alternative names for the same file*

- **Hard Link** : references address of the file *(only for files in Unix)*
- **Symbolic Link** : references full path name of file *(or directory in Unix)*
- **Two types** : Cyclic or acyclic linkages
- sometimes links *across* file systems not allowed

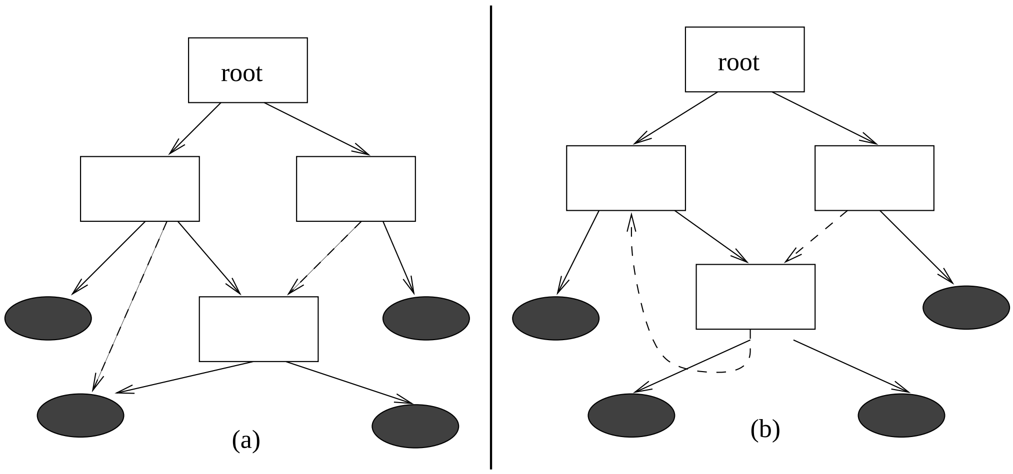

Figure 3: (a) Acyclic graph directory ; (b) Cyclic graph directory

- **Problems** : *File Deletion* :
  Search for links and remove them
  Leave links (causes exception when used e.g. symbolic links, Mac aliases)
  Keep link count with file - delete when count =0 (e.g. hard links)
- *Loops* : Directory traversal algorithm may loop (in archiving utilities for instance) - in cyclic graph directories

9 Access Control

Passwords

- *Directory* : User/Owner checked at the Master File Directory level (password verification at login)
- *File* : allow sharing by giving password

Password protection is an *all or nothing* protection - i.e. not flexible - *Passwords MUST be encrypted*

Access Privileges

Specify *type* of access allowed to others (in increasing order of privilege) :

1. No Access (N)
2. Execute only (X)
3. Read only (R)
4. Append only (e.g. log files) (A)
5. Read/Write (update) (W)
6. Change Protection (C)
7. Delete (D)

Access privileges can be encoded hierarchically - i.e. each access right includes those below it (therefore, ability to write implies ability to read and execute also) - alternately, encode in a key with one but for each access mode.
Owner must always be able to change protection mode - WHY ?

9.1 Specifying permitted Users

User Classes

- Owner (from userid)
- Group (from groupid)
- Others (the rest)

In Unix : *ordinary file* : R, W, X privileges
directory : R, W, X (search for name and execute file)
Superuser (root) has over-riding permission to access in any mode
OwnerID can be substituted for userid for execute access to files - change protection domain e.g. to allow user to execute a system utility with system capabilities

List of Users

- with each directory
- with each file

e.g. VAX/VMS : record - username, access mode (R, W or E)

There are other variations to these models, for instance, the concept of *DOMAINS* in WinNT - we shall cover these in *Security* later

10 Consistency Semantics

There should be *consistency* maintained by the file system for all users sharing the system :

Unix Semantics

- Writes performed by one user are immediately seen by other user(s)
- Can optionally share current location pointer :
  User 1 reads block 510, User 2 then reads block 511 (Single File Image)

Session Semantics

- Writes to open file are not visible to other users
- When file closed, changes are made visible to new sessions, but are not seen by current sessions

Multiple copies of files ?

Immutable Files

- Read only sharing - files cannot be changed

Locks

- Sharing of files $\equiv$ locking to give exclusive access
- Prevent others reading while the file is being updated
- Locks can be : *On whole files* or *Individual record of a file*

11 Disk Organisation

Size of a file is variable - $\equiv$ the need for *dynamic space management*
Generally, space is allocated in *blocks* (typically 512, 1024 or 2048 bytes)

Since a disk is composed of blocks, and files are translated into these blocks, how is this structure actually implemented ?

11.1 Block Linkage or Chaining

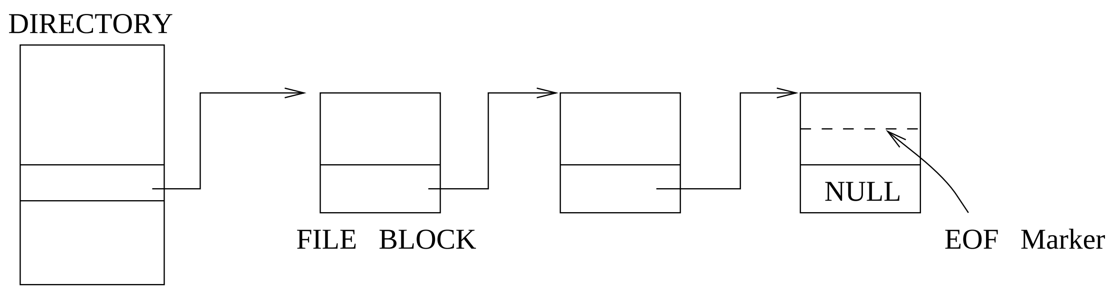

Figure 4: Block Linkage (Chaining)

- Search chain to find required logical block
- Problems : Only suitable for sequential file access (many accesses needed to find the N^th file element)
- Wastes pointer space in each block (perhaps better to use 4 contiguous blocks and then use pointer to next cluster. How many blocks in cluster ?)

11.2 File Allocation Table (FAT)

aka *File Map*

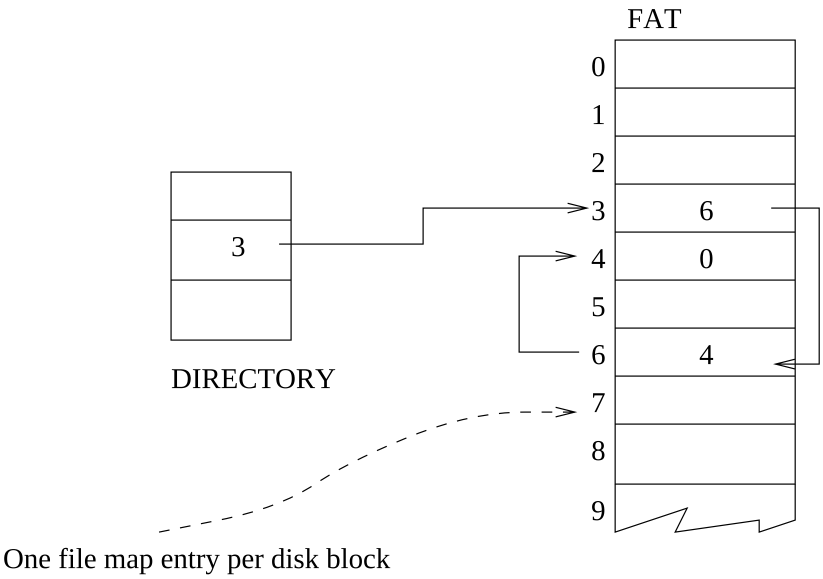

Figure 5: File Allocation Table

- Allocation of disk blocks to files recorded in a *file allocation table* (e.g. MS-DOS, OS/2)
- Sequential Access
- Hold all or part of file allocation table in store to reduce number of disk accesses
- Maximum of two disk accesses per file block access if blocks for the file are referenced on one file map block (i.e. ideally file blocks should be clustered on the disk)

11.3 Index Blocks

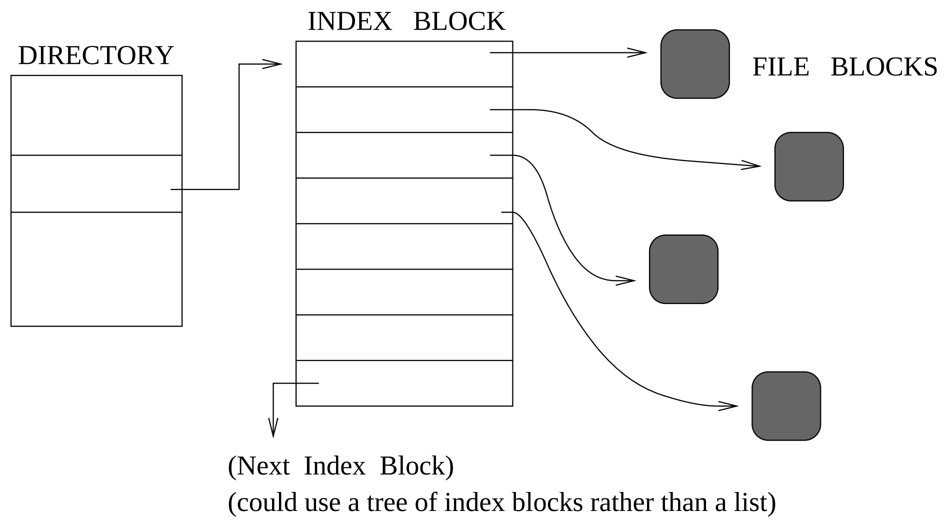

Figure 6: Index Blocks

- After first file access - the index block can be kept in memory
- Good for *random access*
- A whole index block at a time must be allocated even for small files
  large disk space overheads than for previous schemes
- Difficult to insert or delete blocks

an example of a unix directory structure (with symbolic link)

```
-rw-r--r--  1 ofr  1902 Sep  6  1995 Nn_Code
-rw-r--r--  1 ofr  1780 Jul  9  1996 PVM_vs_MPI
lrwxrwxrwx  1 root 27 Sep  6  1995 PetriNets -> /usr/local/petrinets/
drwxr-xr-x  2 ofr  512 Oct 26 14:52 Pmosaic
```

11.4 Criteria for choosing Disk Block Size

1. *Too large* : waste space for small files
2. *Too small* : Too many pointers
3. Memory buffer space - needed for :
   current index block
   block being read or written

Typical Block Sizes :
Unix : 512 bytes or 1024 bytes
IBM 370 : can differ between files

11.5 Management of free space

- **Free space file** : Keep a file of free blocks on disk (for allocation)
- *Bitmap* : An array of bits (with an entry for each block). The bit corresponding to a block is set or cleared (depending on whether the block is empty or being used). The bitmap can be held in main store for efficiency - however, any system crash may cause inconsistencies between bitmap on disc and bitmap in main store

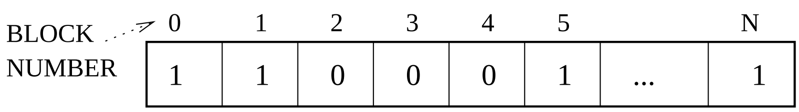

Figure 7: Bitmap

12 Archiving and Recovery

Information in file store can be lost through hardware or software failures : Hence, the need for **Backup and Recovery** (usually to magnetic tape, Digital Audio Tape (DAT) or CD-ROM).

- *Periodic (total) Dump* : Copy everything in file store.
  File system may be very large : time consuming - there may not be that many changes between the periods being considered.
  Recovery of individual files by scanning entire dump tape

- *Incremental Dump* : Only copy files *updated, changed or created* since last dump
  Using *flags* per file or per directory to determine files which need to be saved
  Dump program usually run as a low priority process and clears flags associated with files or directories

- Must be performed frequently to avoid loss of data (could generate a large number of dump tapes)

Recovery:

- Involves working *backwards* through incremental dump tapes

- Restore any dumped files *except* those already restored (overwriting with old copies!!)

- Finally use last periodic dump

this is better than working forwards - WHY ?

13 Disk File I/O

Request file : /usr/pub/userlist

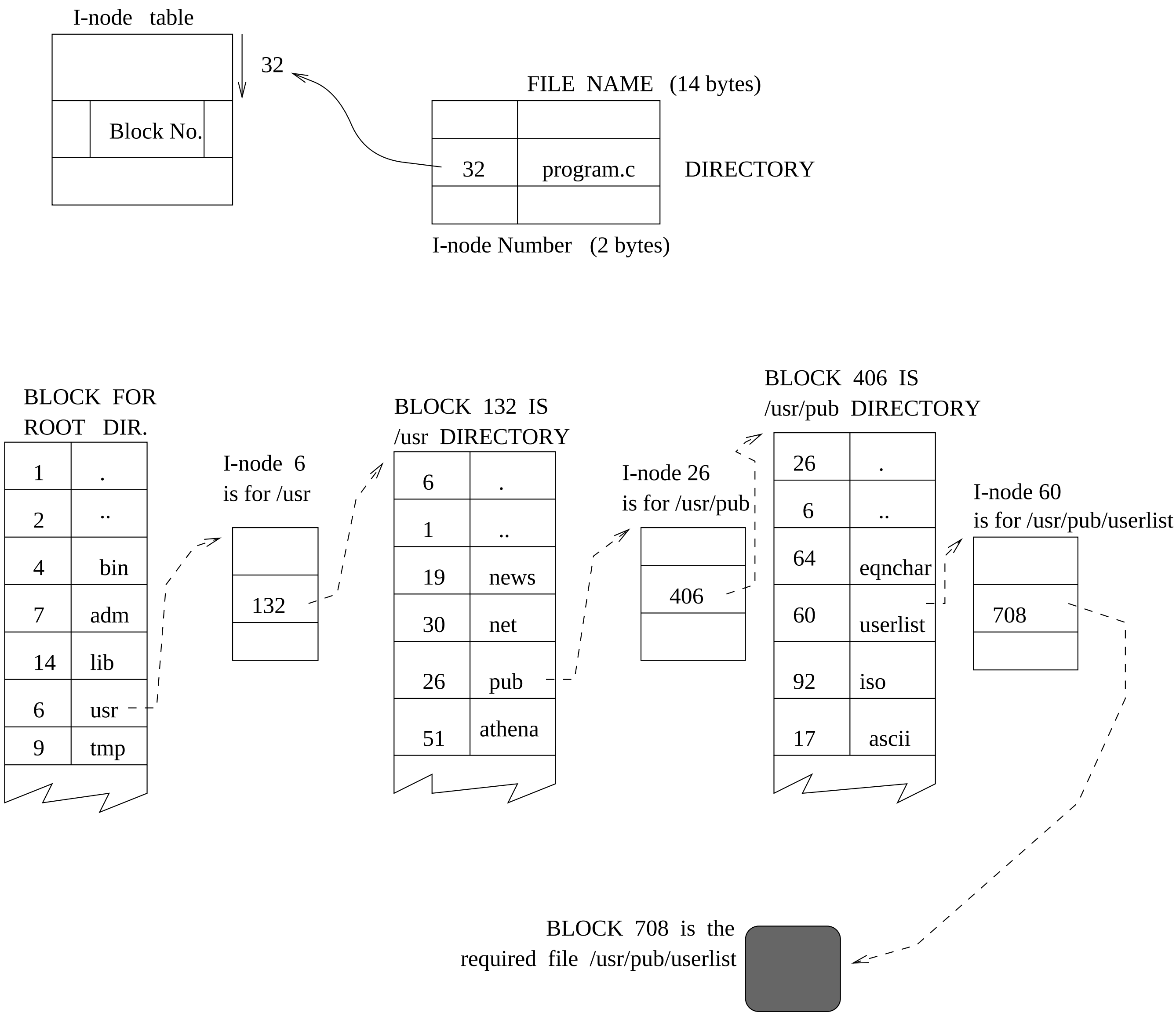

Figure 8: Unix I-nodes

13.1 FileMan:

Information required for each process includes, *userid, root directory* and *current directory* :

- Use process ID to get User ID
- Look up directory entry
- Check access privilege for userid
- Check file state - only 1 user can have file open in writemode
- Directory $\equiv$ file location (disk no. and address on disk)
- Create entry in process file descriptor table

13.2 Information needed on open files

- File name or unique file identifier (cf unix i-node number)
- Device driver id
- Location of first block in file
- Location of next block to be read or written (sequential access)
- Access mode

13.3 Multi-User Systems

- Open and close takes time - fileman is a bottleneck
- Move device independent I/O into user process
- Provide file access procedures (run in kernel mode)

14 Implementing file sharing

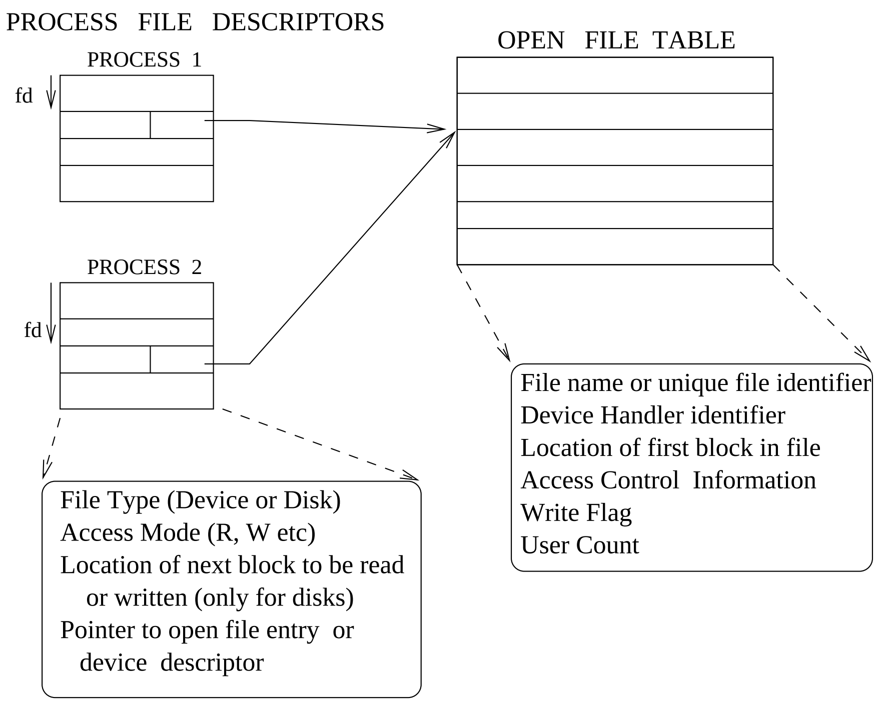

Figure 9: File Sharing

Assume :

Only one process can have file open with write access

Permit multiple readers

If *User Count* $= 0$ and *Write Flag* $=$ false then permit open in *writemode*

If *Write Flag* $=$ false then permit open in *readmode*

Increment *User Count*

Require mutually exclusive access to whole *Open File table* or individual entries in table, but access only required for short time

15 Scheduling of disk accesses

Servicing requests to read/write data from/to disk. Intention is to order pending requests with respect to present position on disk, to minimise seek and/or latency times (see scheduling in processes/CPU).

- *No re-ordering of requests* : **FIFO** (First In first out) or first come first serve.
  Implies random seek patterns (Ok for lightly loaded disks, but poor performance for heavy loads).
  Fair

- *Order according to seek time* : **SSTF** (Shortest Seek time first)
  Order requests based on shortest distance from current head position
  Discrimination against innermost and outermost tracks (hence, unpredictable and unfair performance)

- **Scan Scheduling** : Choose requests which result in shortest seek time in preferred direction.
  Change direction only when reached outermost (innermost) cylinder (or when no further requests can be serviced in preferred direction)
  Most common scheduling algorithm
  Long delays for requests at outer locations

- **N-Step Scan** : As for scan scheduling, but services only requests waiting when a sweep began. Requests arriving during a sweep are serviced on return sweep.

- **C Scan** : Service requests in 1 direction ONLY. When reached innermost request, jump to outermost request
- **Rotational Optimisation** : Once head reaches a track, service requests for that track in rotational order

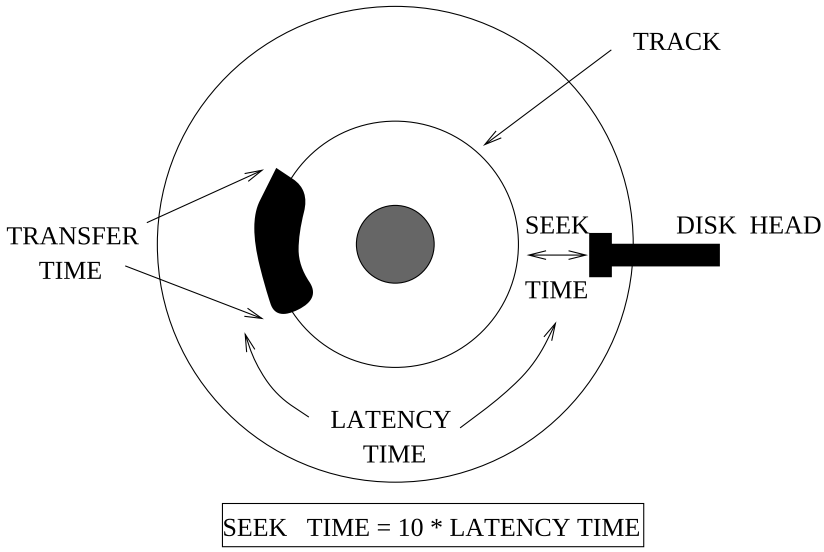

Figure 10: Disk Scans

For Scans : *Request queue* : 98, 183, 37,122,14,124,65,67 (FIFO)
When handling request 14, new requests arrive for 50, 70, 100 (long delay before 183 is serviced) (SSTF)
Head starts at 53 ; direction towards 0 (Scan)
Requests 80, 140 arrive when head moving outwards (N-Step Scan)
Head starts at 53; direction outwards (C Scan)

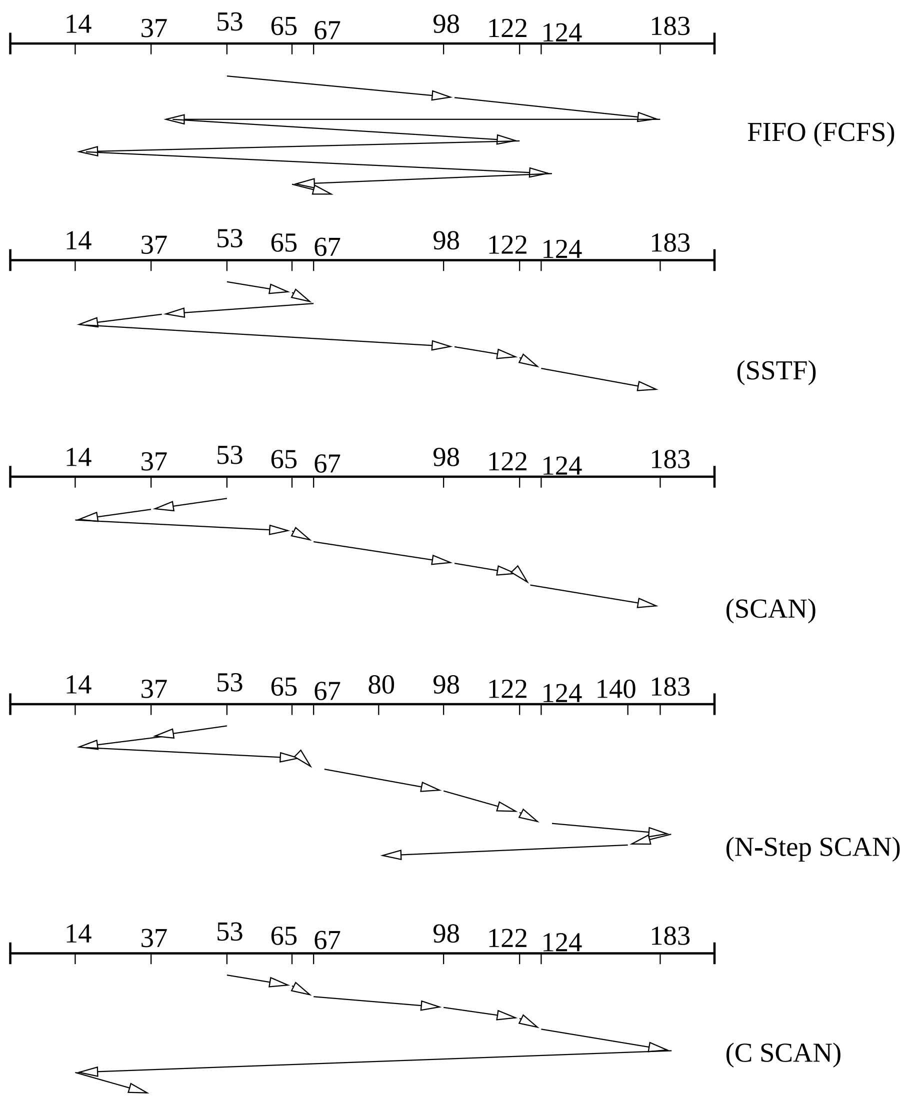

Figure 11: Disk Scans