1065

Bell Laboratories

subject: A Description of the UNIX File System

date: September 16, 1975

from: J. F. Maranzano
MF-75-8234-32

ABSTRACT

The UNIX File System is a hierarchical, tree structured file system that is one of the major components of the UNIX Time-sharing System.

This memorandum describes the structure of the file system as well as the applicable control blocks. It explains how block and file allocation work and the steps involved in handling file I/O. The implementation of device access as part of the file system is explained.

Finally, this document describes the operation of the unique features of the UNIX file system such as shared files, pipes, mountable file systems, and "raw", non-UNIX, I/O.

CONTENTS

1. Structure 1

2. File-System Device Formats 2

3. Block Allocation - The Super Block 3

4. File I/O 5

5. Shared Files 7

6. Pipes 8

7. Devices - Special Files 9

8. Mounted File-Systems 9

9. Raw I/O-Non UNIX File I/O 10

Bell Laboratories

subject: A Description of the UNIX File System

date: September 16, 1975

from: J. F. Maranzano MF-75-8234-32

MEMORANDUM FOR FILE

I. Structure

This memorandum describes the organization and operation of the UNIX File System. Reference [1] gives a description of the UNIX file system structure. This document assumes the reader has an understanding of the descriptions in [1].

Figure 1 is a pictorial representation of a UNIX file system where circles represent directories and squares represent files. Notice that the file system is tree structured with files and/or directories as the leaves of the tree. A UNIX file is a collection of bytes written on portions of disk devices. UNIX file names consist of any one to fourteen ASCII characters. Files are accessed using a "path name" in the file system tree. A path name is a collection of directory names followed by a directory or file name each separated with the character "/". Normally a file search begins at a default "current" directory but if the first character of a path name is a "/", then the search algorithm starts in the root directory. For example in Figure 1, file F3 can be accessed with the name /D1/D3/F3, and file F9 in the mounted file system can be referenced by /D7/D5/F9.

Each file in a file system is known internally by a unique number, the inumber, and has an associated control block, the inode, that fully describes the attributes of the file. The inumber for the root directory of each file system is one. In this way any tree search of a file system can always start in the directory associated with inumber 1.

A "directory" is nothing more than a file containing a group of 14 character names and related inumbers, and whose inode has a flag indicating it is a directory. Two special file names are used in directories: dot (.) that "points" to the directory itself and dot-dot (..) that points to the parent directory. That is, the directory entry for dot contains the inumber of the directory itself, and the entry for dot-dot contains the inumber of the parent directory. Figure 2 shows a directory (D1 from Figure 1) with inumbers versus names. Whenever a file is removed from a directory, the inumber portion of its directory entry is zeroed. In Figure 2 for example, files XYZ and WWWW existed under this directory but have since been removed. The next file to be created in this directory will use the first entry with a zeroed inumber. Dot and dot-dot are not used by the operating system itself, but are purely conventions enforced by the software that creates directories.

Figure 3 shows a representation of a file system with the dot and dot-dot notation. With this scheme a program associated with some directory can traverse the file system tree using the dot notation. For example in Figure 3, a program which is currently residing in directory P can access FILE2 by the notation ../FILE2.

Some of the naming conventions of directories in the current UNIX file system are listed in Figure 4 along with a explanation of the contents of these directories.

One important aspect of the UNIX file system is that physical devices are accessed through normal filenames. The directory /dev has been reserved for these "special files". See Section VII for more detail.

II. File-System Device Formats

File system devices in UNIX are broken into 512 byte blocks. Some file systems are generated so as to reside on a single physical device, others are generated to have many file systems on the same device, and some file systems are setup to even extend across physical devices. Each file system, whether it occupies a physical device like a single platter disk pack of 4000 blocks, or a portion of a large disk pack of 65000 blocks has the format shown in Figure 5.

Block 0 of the file system is reserved for the boot program (See [2] Section VIII) and is otherwise unused by the file system.

Block 1 is called the "superblock". The next M-2 blocks are called the "ilist" and contain inodes, the control blocks for all the files in the file system. The size of each inode is 32 bytes, therefore 16 inodes are in each ilist block. The maximum number of files in a file system is 16 times the ilist size in blocks. The remainder of the blocks in a file system are used for data files, directories, and the free list (see Section III). File systems are created with the make-file-system, MKFS [2], command in which the maximum number of blocks (N in Figure 5) can be specified. The size of the ilist can either be specified explicitly or calculated by the MKFS command.

<u>III</u>. <u>Block Allocation - The Super Block</u>

A pictorial representation of the superblock is shown in Figure 6. Whenever a file system is mounted, the superblock of that file system is read into memory. The superblock contains the size of the ilist in blocks, the size of the file system in blocks, a table of up to 100 free blocks, and a table of up to 100 free inumbers. Block allocation occurs from a "free list" of blocks. The superblock contains a pointer to a linked list of blocks, the free list. Each block in the free list contains 99 free block numbers as pictured in Figure 7. The last block in the free list contains a zero pointer indicating the end of the list. Because the UNIX file system is block oriented, a block number is synonymous to a pointer to that block.

Individual blocks are allocated from the free list using the following algorithm. The allocation scheme is to reduce NFREE by one and use the block number pointed to by NFREE. The deallocation scheme is to place the number of the block just freed at the entry pointed to by NFREE and increment NFREE by one. In the example of Figure 8, NFREE is equal to 90 and the next block to be allocated is 176. From Figure 8 to Figure 9, block 176 was allocated and block 25 was freed. Notice in Figure 9, that all entries $FREE_{90}$ to $FREE_{100}$ are obsolete and should be ignored. (In fact some block numbers of free blocks may appear there, like 25). The free list is only used to keep track of free blocks.

When NFREE equals 1 the table is replenished from another set of 100 block numbers that are contained in the block in the $FREE_0$ entry. So in Figure 9 after 89 blocks have been allocated, block 55 contains another set of 100 block numbers. This algorithm continues until $FREE_0$ contains a zero indicating the end of the chain.

Block deallocation works in reverse. Blocks that are freed are added at the NFREE entry until the table is full. At that point, (NFREE=100) the next deallocated block is used to write the full table, $FREE_0$ is set to that block number, and NFREE is set to one.

Inumber allocation works similarly to block allocation. Remember each file in a file system is internally known by an inumber, therefore inumber allocation takes place only when a file is being created whereas block allocation takes place as a new or existing file is written. A pointer, NINODE, is used as a displacement into a list of 100 inumbers (See Figure 6). To allocate an inumber, NINODE is decremented by one and the inumber at that position is used. To deallocate an inumber, place its value at NINODE and increment by one. If the table becomes depleted of free inumbers (NINODE = 0), then a sequential search is made of the ilist looking for another 100 unallocated inodes; their associated inumbers are placed into the in-core superblock. If all the inodes in the ilist are allocated, no more files can be created in the file system and an error condition exists [4].

Inumber deallocation puts entries into the list of free inumbers. If inumber deallocation overflows the table of 100, any new deallocated inumbers are ignored because their inode was already marked as unallocated and written back into the ilist where it can be picked up in any subsequent search for unallocated inumbers (as described in the preceding paragraph).

Execution of the MKFS (make file system) command to build an empty file system works as follows. Block numbers from the end of the file system are collected together 100 at a time. The next block is then used to store these 100 numbers. This is repeated until all the free blocks have been put into the linked list on the disk with the superblock pointing to it. This implies that file allocation initially will result in nearly contiguous blocks. After some period of time, block allocation and deallocation will tend to fragment the free list. At present no programs are available to restore the sequential ordering. A DUMP of the file system, followed by a new MKFS, and a RESTOR will cause re-ordering and will reduce the fragmentation [2]. (This is true because DUMP/RESTOR retains only the data of a file not its place on the disk).

Several routines exist within UNIX to maintain file system integrity. The command SYNC [2] is used to force the in-core superblocks of file systems to be written to the disk. The program UPDATE can be invoked to write the superblocks to the disk every 30 seconds.

Still it is possible for the system to crash and the disk copy of the superblock to be incorrect. The program CHECK can be used to search through the file system tree and match inumbers against allocated inodes, make sure blocks do not appear in both the free list and an inode, and other integrity tests on the file system. (See Reference [2]).

IV. File I/O

I/O in the UNIX file system is done through read and write system calls [1,2,3]. In order to do I/O a file must be OPENed or CREATed. These two system calls return a "file descriptor", an integer number from 0 to 14. File descriptor 0 is called the standard input file and file descriptor 1 is the standard output. File descriptor 2 is generally used for diagnostic output. File descriptor 0, 1 and 2 are opened during initialization and are pointing to the user's terminal. They can be closed and reopened to point to any file (re-directed I/O) (see Reference [3] for more detail).

Read and Write system calls have a file descriptor, buffer address, and the number of bytes to be read/written as arguments. The return from these calls is the actual number of bytes read or written.

Each process has associated with it a Process File Table (u_ofile) which contains up to 15 pointers to the System File Table (see Figure 10). This Process File Table resides in the per process control block (user.h) which is swapped when the process is swapped.

At open time if the file exists its name appears in a directory with an associated inumber. The open processing uses this inumber as a index into the ilist to get its inode (the inode is the control block that fully defines the file). The inode is put in an entry in the System INODE Table (inode.h). Open processing also creates an entry in the System File Table (file.h) and finally fills in the address of this entry in the Process File Table (see Figure 10).

If the file is being created, then an inumber is allocated from the in-core copy of the super-block table of free inumbers, and an entry is made in the System INODE Table and System File Table.

The System File Table and System INODE Table both reside within the resident control program so neither is swappable. Moreover, in the current implementation both are limited to 100 entries which means that the total number of open files in the system (open by all processes) is 100.

Reads and Writes of the file take place at the byte position equal to the "offset" value in the System File Table. This value is initially set to zero by open. The offset can be changed using the Seek system call. When this offset is greater than or equal to the size of the file ("size" value in the inode) then the read/write processing assumes it is at the "end of file". The flags in the inode describe the kind of file: directory, small file, large file, special file, etc. The number of links indicate the number of directory pointers to the file; the file cannot be physically removed until this number goes to zero. The uid, and group-id contain the id of the owner of the file, and the id of the project associated with the file. The 8 addresses in the inode are block numbers of data if the file is small (less than 4096 bytes), or are block numbers of indirect blocks of addresses that in turn point to data if the file is large. To read byte N for example (offset = N), divide N by 512 bytes/block to get block number B and remainder R. If the file is small, then B is less than 8, and is used as a displacement into the list of eight addresses to fetch the block address of the data, $ADDR_B$. The value R is then used to fetch the correct byte within $ADDR_B$.

If the file is large, then B is divided by 256 (256 words/indirect block) to get I, which is less than 8, and is used as a displacement to get $ADDR_I$ that points to a block of 256 addresses. The remainder of this division, K, is used to locate the correct address within indirect block. The value R is still used to get the byte within the data block (see Figure 13).

With this algorithm, the maximum size file is 8*256*512 bytes or 1M Byte (2^{20} Bytes). This is smaller than can be specified in the size field (24 bits) of the inode.

A variation of this algorithm has been implemented on some UNIX systems to allow "huge" files. The large file has been redefined so that the last address in the inode, $ADDR_7$, points to an indirect block, which in turn has pointers to 256 double indirect blocks. In this implementation, 7*256*512 + 1*256*256*512 bytes could be addressed (a little more than 32M bytes). However the 2^{24} size limitation makes the largest file 16M bytes.

When a file is closed, either explicitly or when a program terminates, the entry in the System File Table is found and the "number of processes" value is decremented by one. When that value is zero then the inode to which it points is written back to the ILIST in the appropriate file system and the System File Table entry is cleared.

Several important characteristics can be noted.

(i) A small file is automatically (without user intervention) converted to a large file when the offset becomes greater than 4096. This is done simply by writing the eight addresses from the inode into a block (which becomes the first indirect block), placing that block number into $ADDR_0$, and changing the inode flag to indicate a large file.

(ii) A file can have holes in it since there are no restrictions about missing addresses (zero values) in the inode or indirect block.

(iii) The offset is changed by using the seek system call. It can be moved by bytes, or blocks absolutely or relative to its original value.

(iv) Two or more processes can share the same file. The System File Table contains a slot for the number of processes sharing that entry. This number of processes is decreased by one for each close of that file and the entry is not purged until the number goes to zero. More about this in the next section.

(v) By implementation convention all directories are restricted to small files to prevent exhaustive search time.

As an example of how this all works together consider the access of file /D1/F1 of Figure 1. Because the pathname begins with / the access starts in the root directory. As you recall the inumber of the root directory is 1, so the system reads the first inode from the ilist into the INODE Table. This inode contains block numbers of data blocks. The data blocks are read into memory and searched for a match on the name D1. When the name is found it will have an associated inumber that is used as a displacement into the ilist to fetch the appropriate inode. That inode is brought into memory and it points to data blocks which are searched for the name F1. Again that name has an associated inumber that is used as a displacement into the ilist to fetch another inode. At last you are ready to read or write the file.

As the pathname gets more complex, more accesses to the file system are made during open time to bring the inode of the file into the INODE Table. Once there however, that inode remains until the file is closed, so that the search overhead is only done once.

V. Shared Files

In UNIX, a process has the facility to spawn a child process [1] which inherits all of the open files of its parent. Figure 11 shows an example where process A′ is a child of process A. Process A′ inherits the open files of its parent because it inherits its Process File Table. Notice file-descriptor 2 contains the address of the same entry in the System File Table for both processes. However, because a parent and child process share the same entry, they also share the same offset into the file; if one process moves the offset, the new offset also applies to the other. This is done because these are assumed to be cooperating processes. If some other process, process B in Figure 11, wanted to read or write the same file, a separate entry in the System File Table is generated for it (with a unique offset) that points to the same inode entry in the INODE Table (file descriptor B1 points to the same file as A2). Writing to this file by B will not change the offset for process A, but will certainly change the data in the file.

VI. Pipes

A pipe is an I/O connection between two cooperating processes that allow the processes to transmit data to each other in both directions. Figure 12 illustrates the concept where process A is writing data to process B and receiving data from process B. Each process in the mechanism has an input side of the pipe on which it reads, and an output side on which it writes. The I/O is done by a special system routine that maintains the data in a FIFO organization.

A user process initiates a system call for a pipe. The system returns two file descriptors: one for reading and one for writing. When a child process is spawned it will inherit all file descriptors including these two file descriptors for the pipe. The child process can cooperate with its parent or other descendants of its parent by reading on the output side of its parent's pipe and writing on the input side.

The implementation is to set up two entries in the System File Table pointing to the same INODE Table entry. The pipe is a real file in all respects, except that it appears in no directory (unnamed file). This fact is reflected by the link count set to zero. This automatically allows the inode to be deallocated with the last close of the pipe ends. A special system routine maintains the offset and filesize in such a way that the bytes remain in FIFO order. Seeks are illegal on pipes. Pipes are implemented as small files for efficiency. As a process attempts to write greater than 4096 bytes on a pipe end, it is roadblocked until the other end has read some data. Because the normal file system mechanisms and buffer pools are used for pipes, it is possible (if the buffer pool activity is small) that the data transfer becomes a memory to memory transfer and no disk I/O takes place.

VII. Devices - Special Files

Referencing Devices in UNIX is done through entries in the file system called special files [1]. Two classes of devices are defined: character and block. Character devices are those associated with character at a time I/O like TTY's. Block devices transfer blocks of data like disks. Each device in the system is assigned a device class, a major and minor number. The major number is associated with the software driver for that device (therefore it is generally associated with a device controller). The minor number specifies the physical drive or device, or is used to subset physical devices into logical portions. Special files are files within the file system whose inodes have been used to record the major and minor device numbers. In $ADDR_0$ of the inode for a special file the most significant byte is the major device number and least-significant byte is the minor. Filenames and protections apply identically to devices as to regular files.

VIII. Mounted File-Systems

A in-active file system is mounted onto a active file system through the use of the MOUNT command [2]. The MOUNT command specifies the special file (device) on which the file system resides and the file or directory (mount point) within the running file system upon which it is to be mounted. This requires that the inode for the mount point be brought into the INODE Table and kept there for the duration of the mount. Furthermore an entry is added to a resident Mount Table consisting of:

1. A pointer the special file containing the new file system.
2. A pointer to a buffer from the buffer pool that contains the superblock of this mounted file system.
3. A pointer to the inode entry in the INODE Table for the mount point.

The Mount Table currently has room for five mount requests. Consider access to the pathname /D7/F7 of Figure 1. The first / starts the search in the root directory for D7. When its inumber is found, a search of the ilist reveals that it is already in the INODE Table. Moreover, the inode in the INODE Table has a flag indicating that this file is a mount point, therefore the Mount Table is searched for a pointer to this inode. Once that is found, the root directory (inumber = 1) of this mounted file system is substituted and the searches continues.

In summary, the flag in the inode of the mount point acts as a switch to substitute the root inumber of a mounted file system for the inumber under consideration.

IX. Raw I/O-Non UNIX File I/O

All UNIX file system data is structured around the 512 byte block. UNIX buffer management is geared to this number. Some data requires a different physical blocking. For example, the 512 byte blocking is a poorer density then is sometimes desirable on magnetic tapes. Another application that requires a different blocking is the reading or writing of foreign (IBM compatible) magnetic tapes. To do this, UNIX provides a facility called "raw I/O" that bypasses the standard UNIX buffer management and blocking to do I/O directly to or from the user's memory. See Reference 2 and the DD command for more detail.

X. Acknowledgement

K. Thompson reviewed this memorandum for accuracy and made many helpful suggestions. C. P. Imagna, B. A. Tague, and R. C. Varney made suggestions to improve the readability of the information.

[illegible]

MH-8234-JFM-nroff J. F. Maranzano

Copy to:
UNOS
Technical Members of Dept. 8234
G. L. Baldwin
L. D. Green
H. S. London
M. M. Rochkind
B. A. Tague

REFERENCES

[1] D. M. Ritchie and K. Thompson, The UNIX Time-sharing System, CACM, July 1974.

[2] K Thompson and D. M. Ritchie, UNIX Programmer's Manual, Fifth Edition (1974).

[3] B. W. Kernighan, Programming in UNIX, Unpublished Memorandum.

[4] T. M. Raleigh Explanation of Abnormal Conditions Within the UNIX Operating System, MF 75-8234-28 March 17, 1975.

[5] T. M. Raleigh Trap and Interrupt Handling Under UNIX. MF 75-8234-29 Unpublished Memorandum.

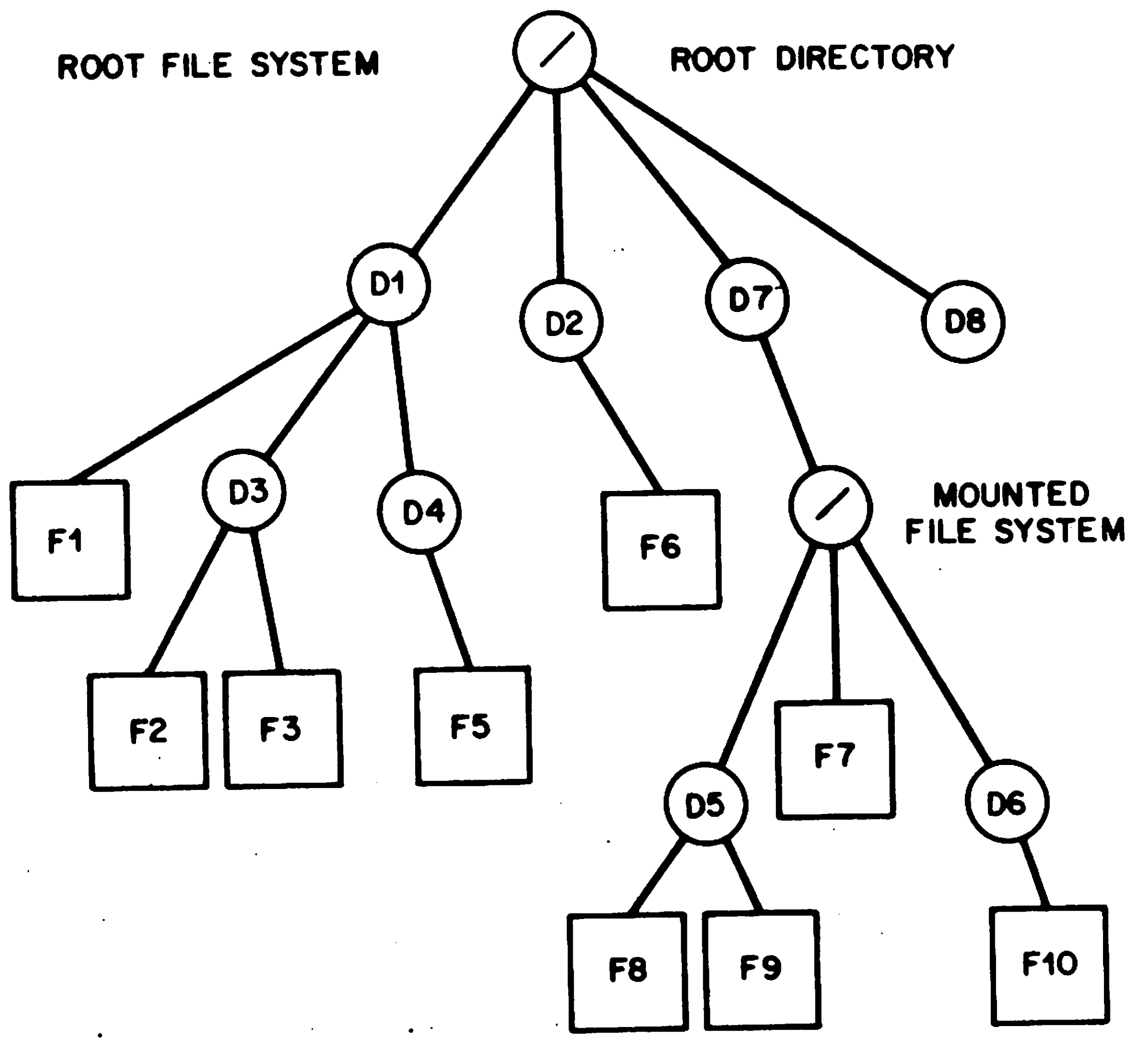


FIGURE 1: UNIX FILE SYSTEM TREE STRUCTURE

[illegible]

2 CHARS	14 CHARS
144	●
1	●●
35	F1
326	D3
0	XYZ
147	D4
0	WWWW

DIRECTORY D1
ALL NAMES ARE LEFT JUSTIFIED WITH ZERO FILL

FIGURE 2 DIRECTORY FORMAT

[illegible]

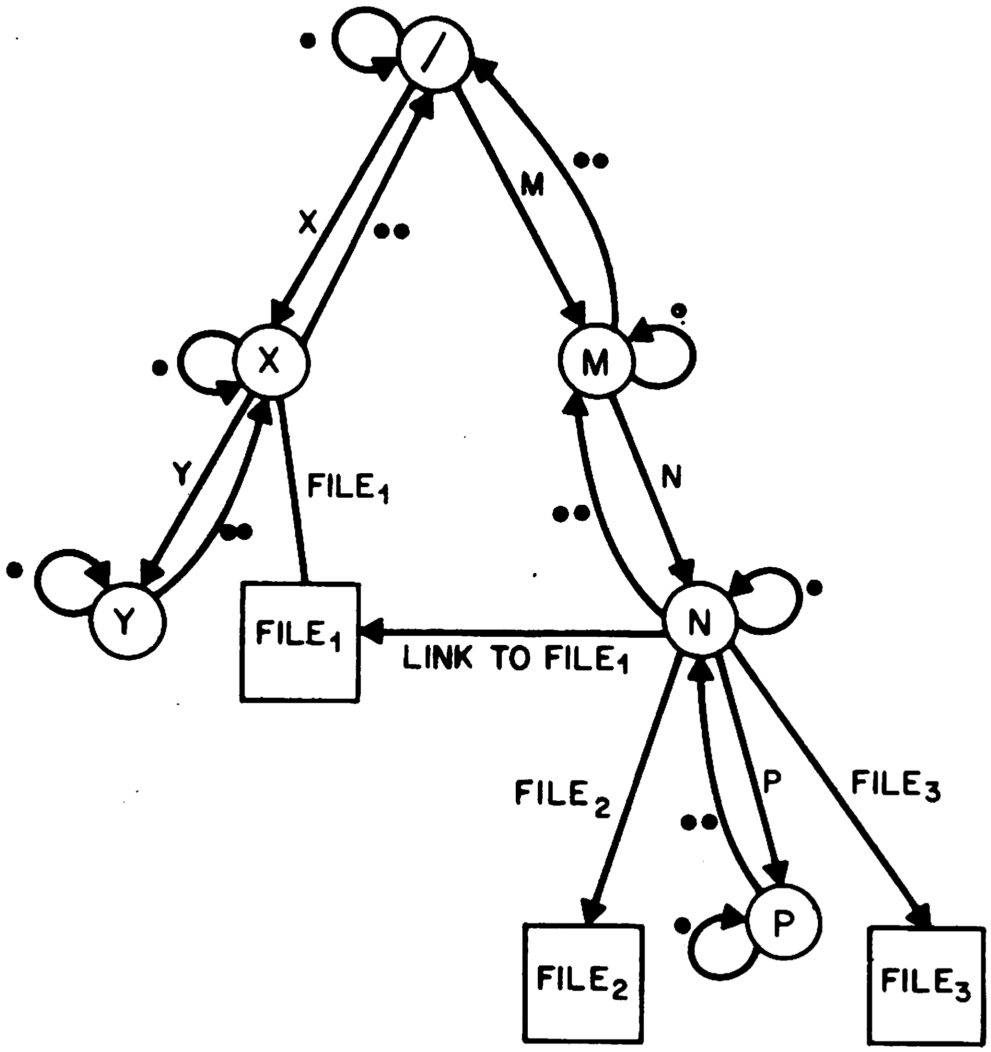


FIGURE 3 LINKS AND DOT CONVENTIONS

/	Root Directory
bin	Executable Commands
dev	Devices
etc	System Related Commands
lib	Compilers and Libraries
mnt	Node to Mount File Systems
tmp	Temporary Files

Figure 4A: Typical Directory Names in UNIX File System

usr			User Related Directory
	bin		Less Often Used Commands
	c		C Compiler Source
	fort		Fortran Compiler Source
		f1	---}
		f2	---¦ Fortran
		f3	---¦ Source
		f4	---}
		fx	
		io	Fortran I/O Source
		rt	Fortran Subroutine Library Source
		rt1	Fortran Math Library Source
		rt2	Fortran Utility Function Source
	games		Executable Games
	lib		Utility Routines Source
	mdec		Stand-alone Utility Source
	pub		Public Tables
	sno		Snobol Compiler Source

Figure 4B: Typical Directory Names in UNIX File System

	source		Command Source
		s1	Source for Commands in /bin
		s2	Source for Commands in /bin
		s3	Source for System Calls
		s4	Source for System Calls
		s7	Source for Documentation Programs
sys			Control Program Routines
	conf		Configuration Dependent Tables
	dmr		Source For Drivers and I/O
	ken		Source for File System & Scheduler

Figure 4C: Typical Directory Names in UNIX File System

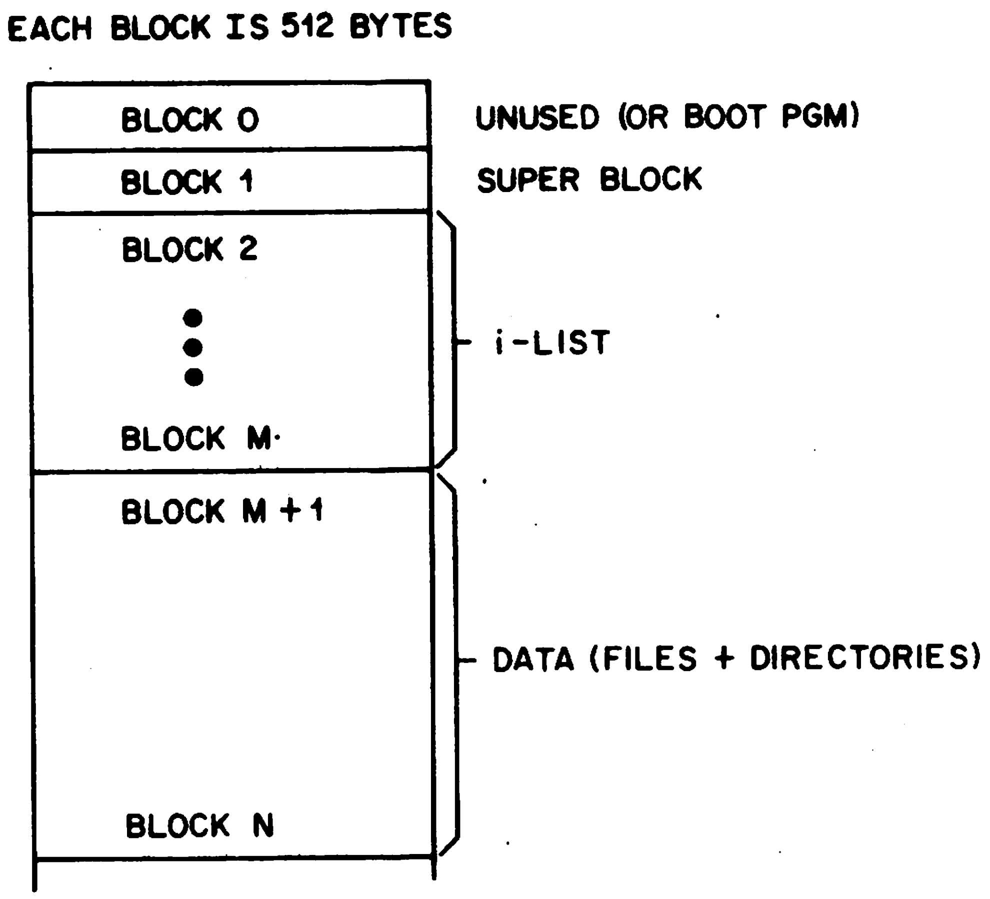


FIGURE 5 FILE SYSTEM DEVICE FORMAT

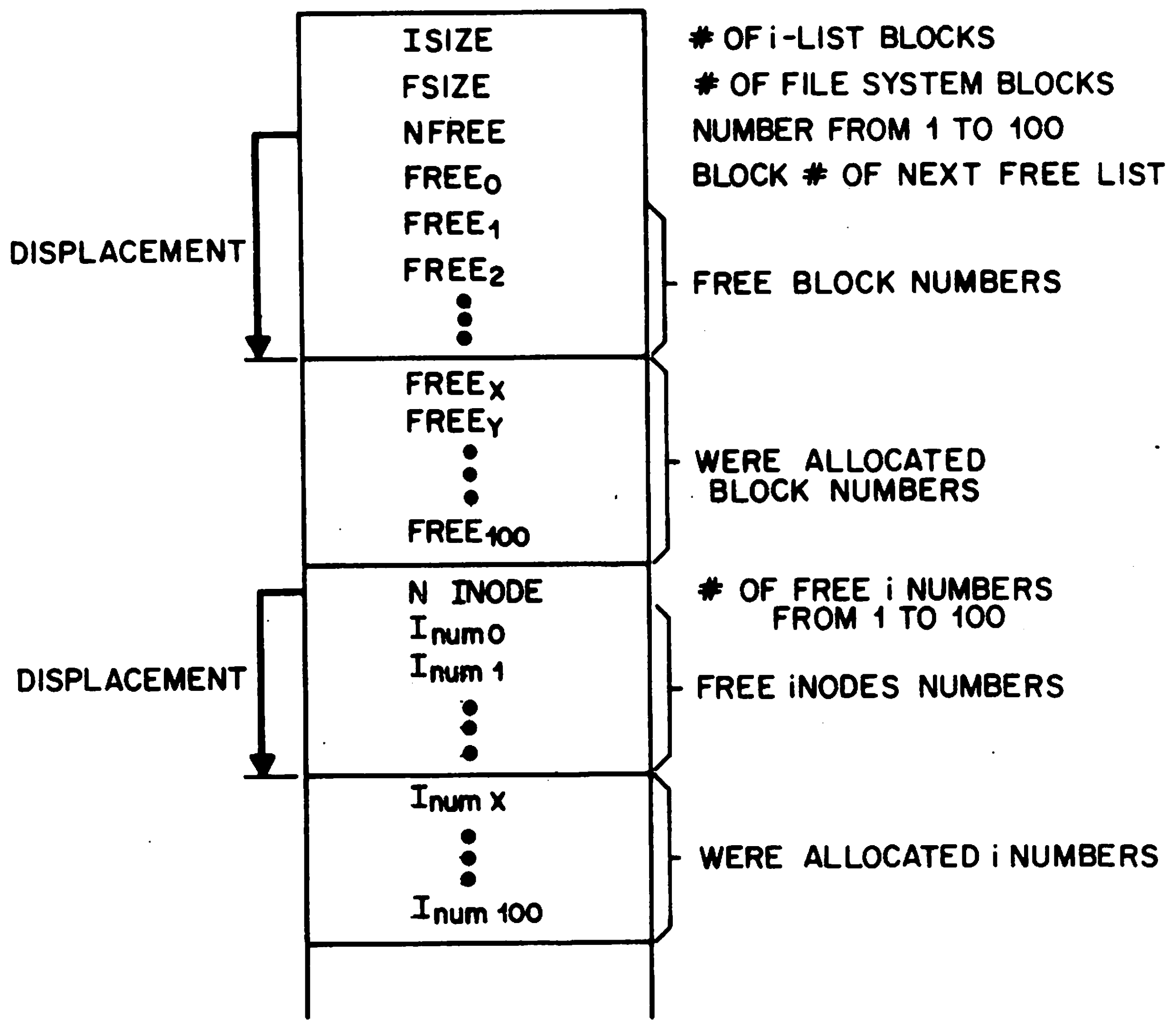


FIGURE 6 SUPER BLOCK-MEMORY COPY

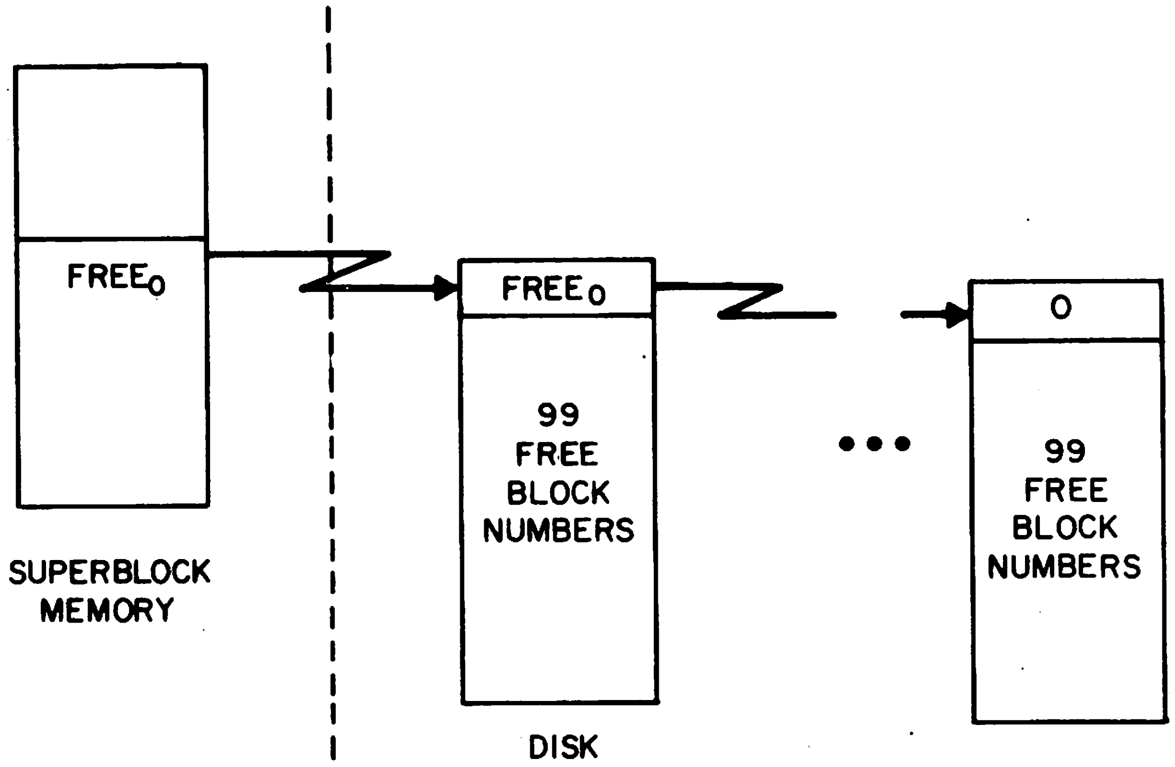
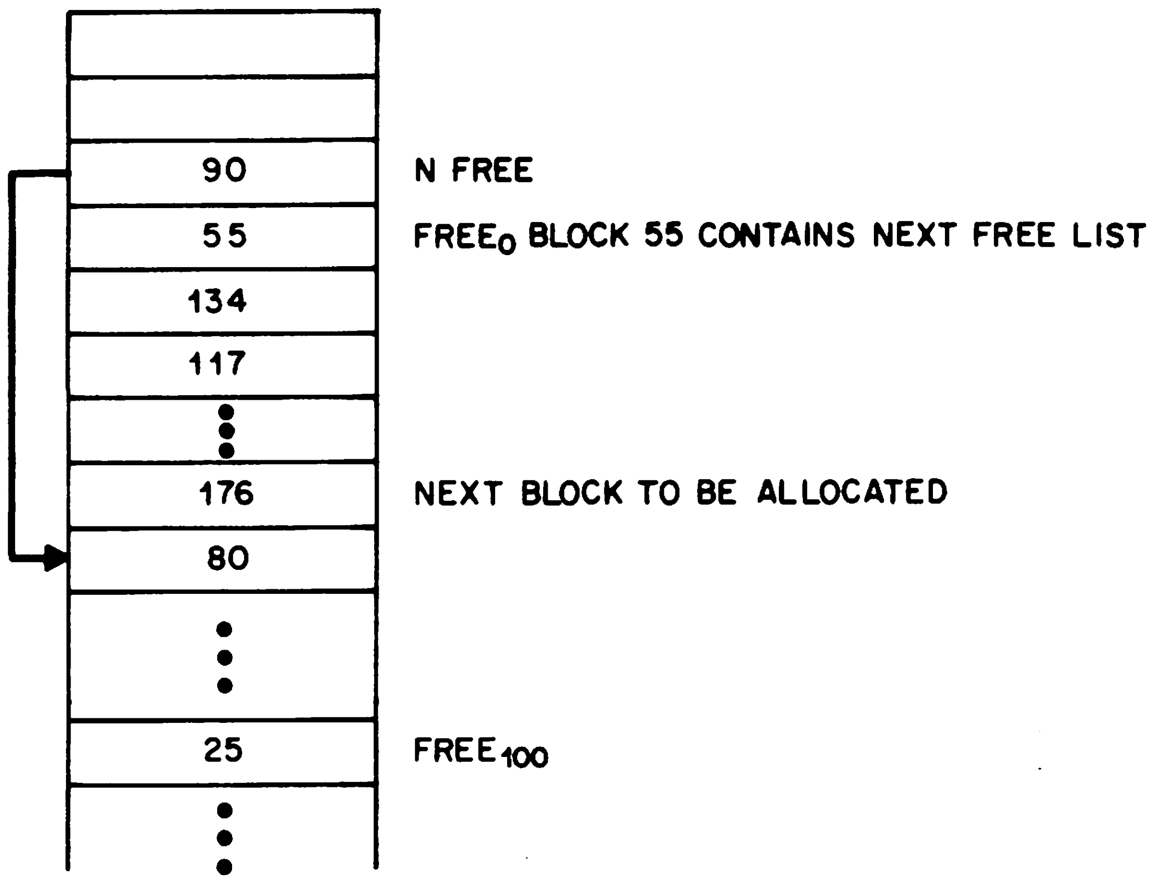


FIGURE 7 CHAINED LIST OF FREE BLOCKS – FREE LIST



90
N FREE
55
FREE$_0$ BLOCK 55 CONTAINS NEXT FREE LIST
134
117
176
NEXT BLOCK TO BE ALLOCATED
80
25
FREE$_{100}$

FIGURE 8 EXAMPLE OF FILE ALLOCATION/DEALLOCATION (1)

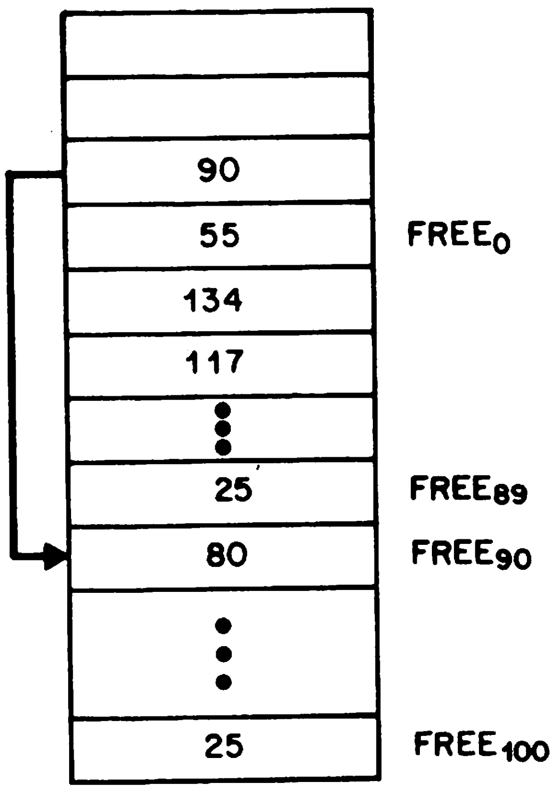


FIGURE 9 FILE ALLOCATION/DEALLOCATION ②

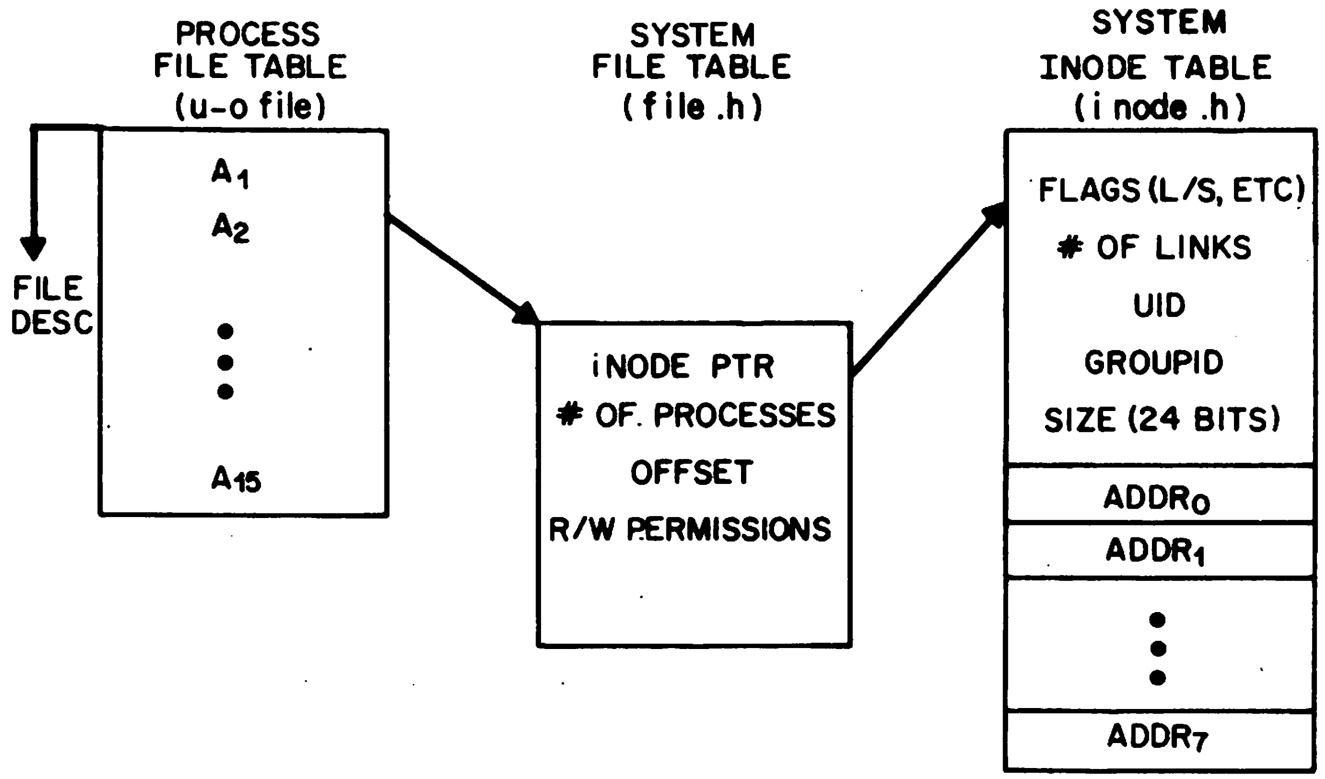


FIGURE 10 FILE SYSTEM TABLES EXPANDED

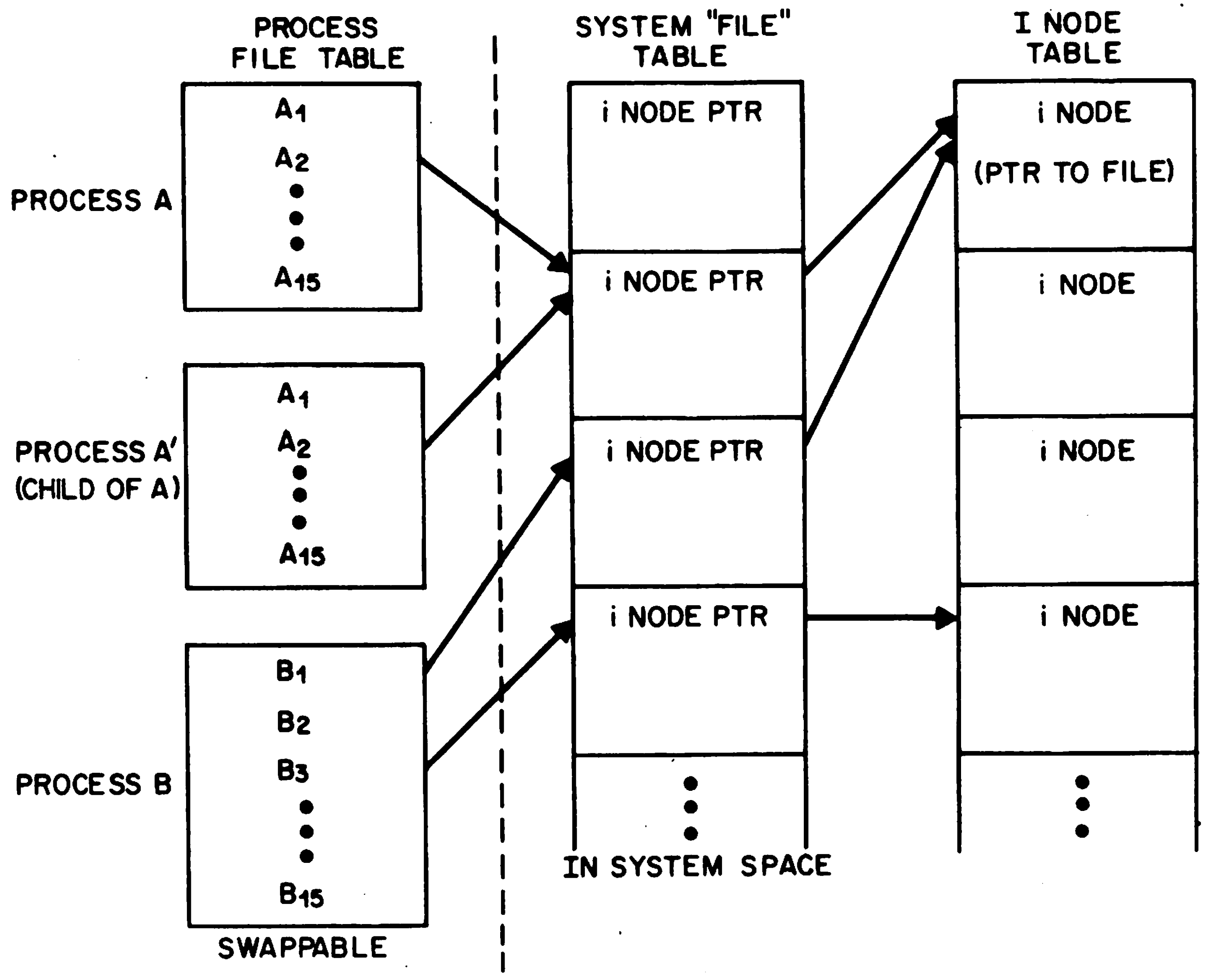
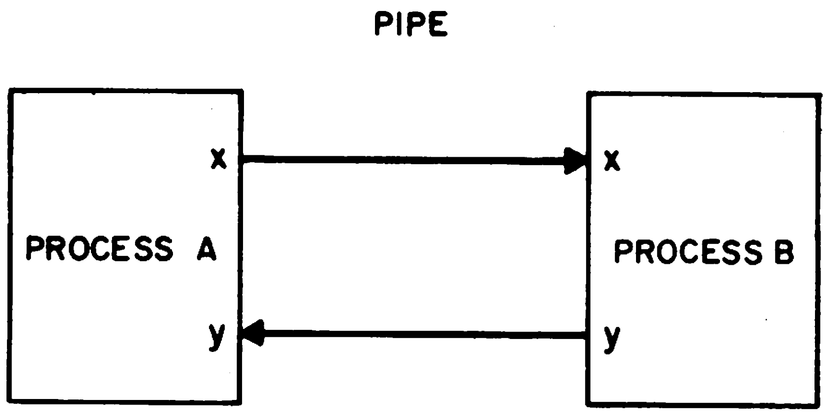
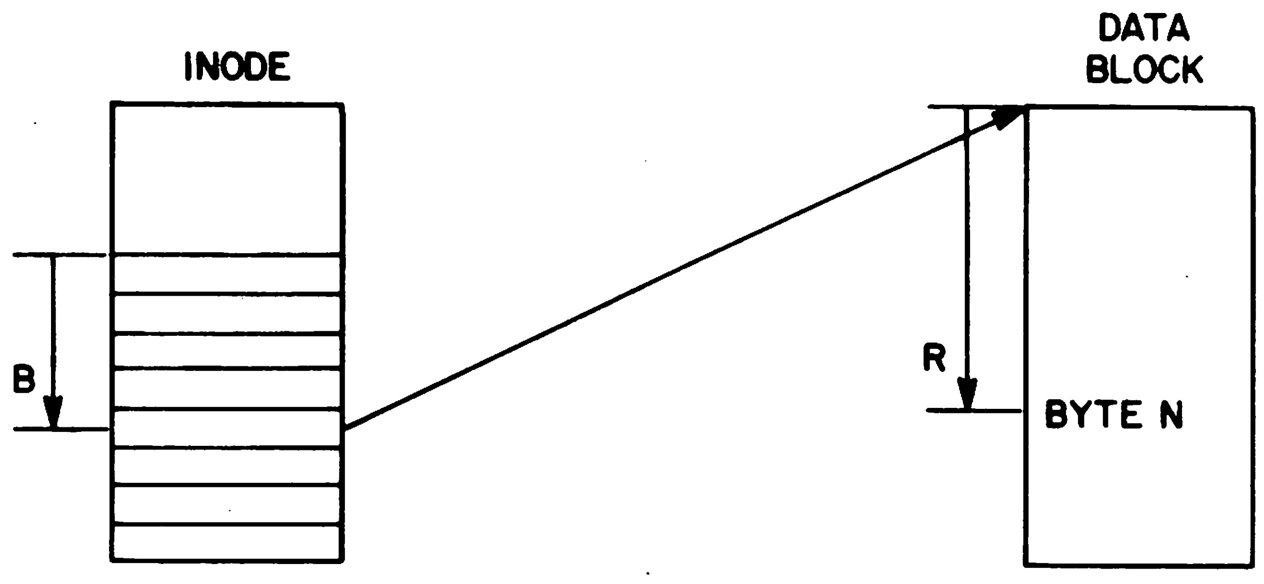


FIGURE 11 FILE TABLES FOR SEVERAL USERS

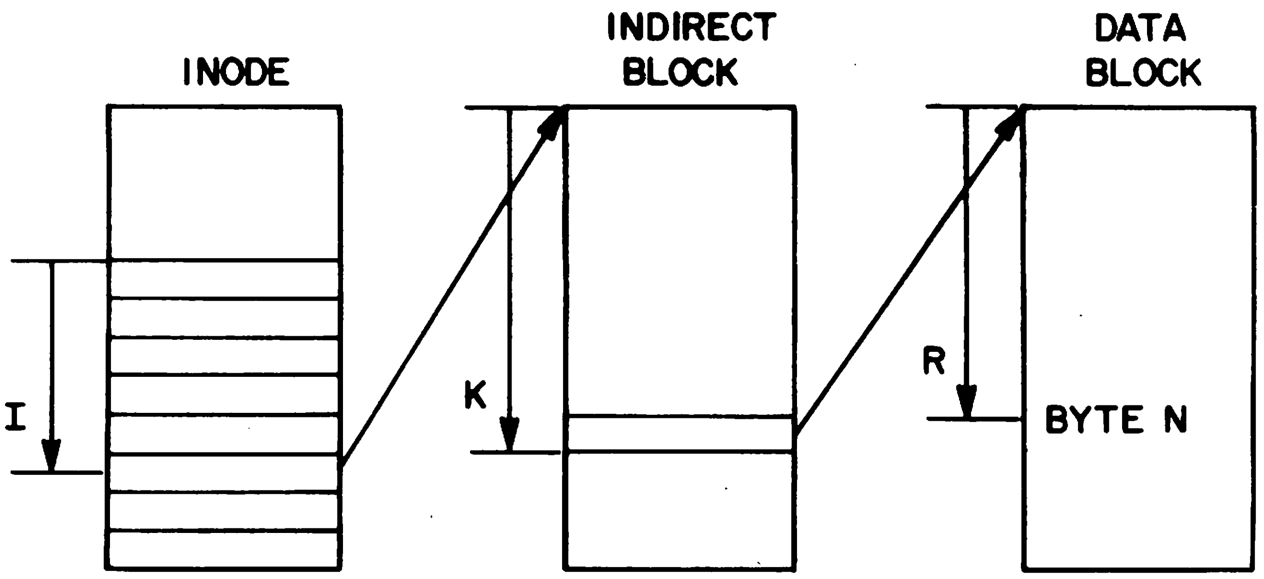


PROCESS A WRITES ON FILES DESCRIPTOR x AND READS ON y. PROCES B READS ON x AND WRITES ON y.

FIGURE 12 CONCEPTUAL PICTURE OF PIPES



SMALL FILE ALGORITHM



LARGE FILE ALGORITHM

FIGURE 13: FETCHING BYTE N FROM A FILE