10

CASE STUDY 1: UNIX AND LINUX

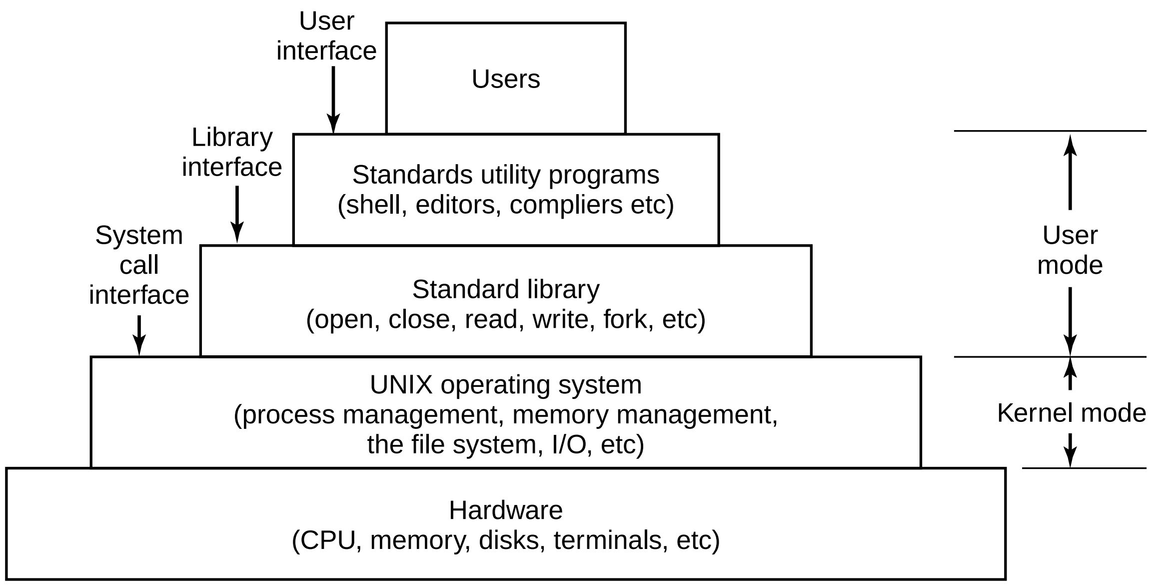

Fig. 10-1. The layers in a UNIX system.

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

Fig. 10-2. A few of the common UNIX utility programs required by POSIX.

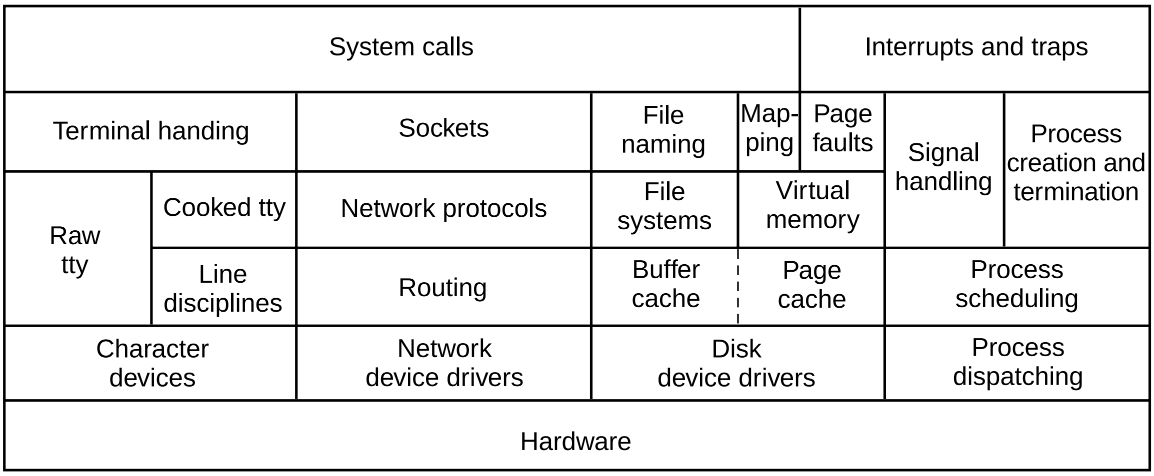

Fig. 10-3. Structure of the 4.4BSD kernel.

```
pid = fork( );              /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
    handle_error( );        /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                            /* parent code goes here. /*/
} else {
                            /* child code goes here. /*/
}
```

Fig. 10-4. Process creation in UNIX.

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Fig. 10-5. The signals required by POSIX.

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Fig. 10-6. Some system calls relating to processes. The return code *s* is –1 if an error has occurred, *pid* is a process ID, and *residual* is the remaining time in the previous alarm. The parameters are what the name suggests.

```
while (TRUE) {                          /* repeat forever /*/
    type_prompt( );                     /* display prompt on the screen */
    read_command(command, params);      /* read input line from keyboard */

    pid = fork( );                      /* fork off a child process */
    if (pid < 0) {
        printf("Unable to fork0);       /* error condition */
        continue;                       /* repeat the loop */
    }

    if (pid != 0) {
        waitpid (–1, &status, 0);       /* parent waits for child */
    } else {
        execve(command, params, 0);     /* child does the work */
    }
}
```

Fig. 10-7. A highly simplified shell.

Thread call	Description
pthread_create	Create a new thread in the caller’s address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Fig. 10-8. The principal POSIX thread calls.

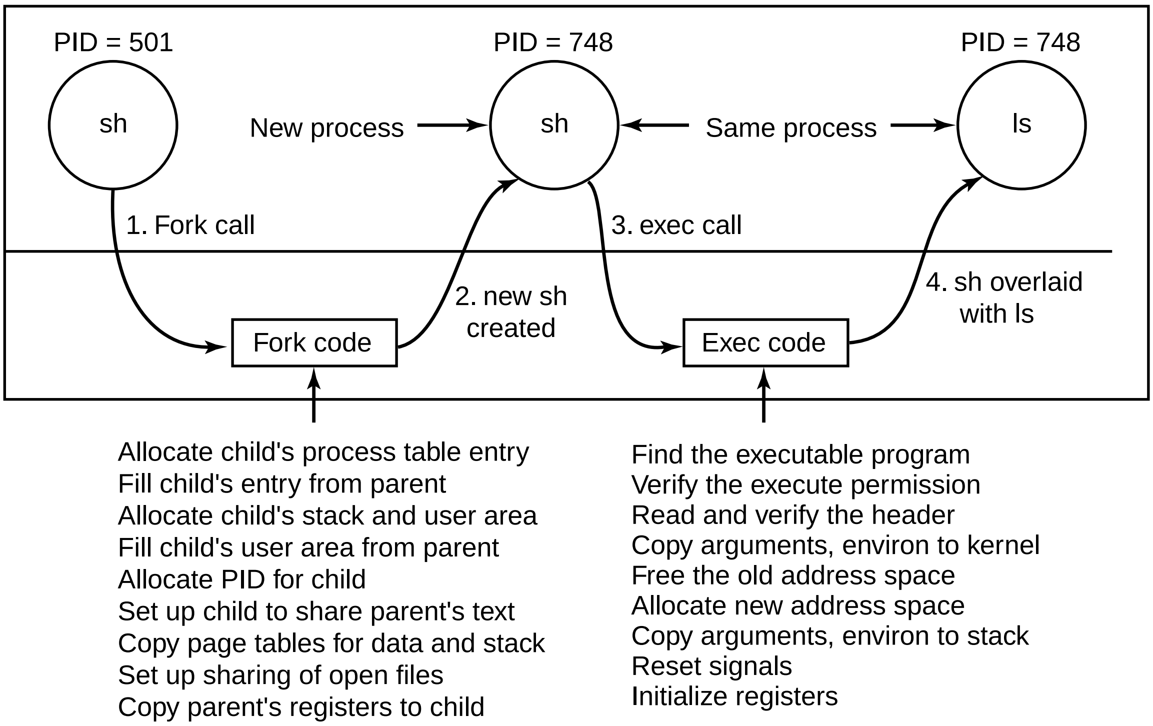

Fig. 10-9. The steps in executing the command *ls* typed to the shell.

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PID	New thread gets old PID	New thread gets own PID

Fig. 10-10. Bits in the *sharing_flags* bitmap.

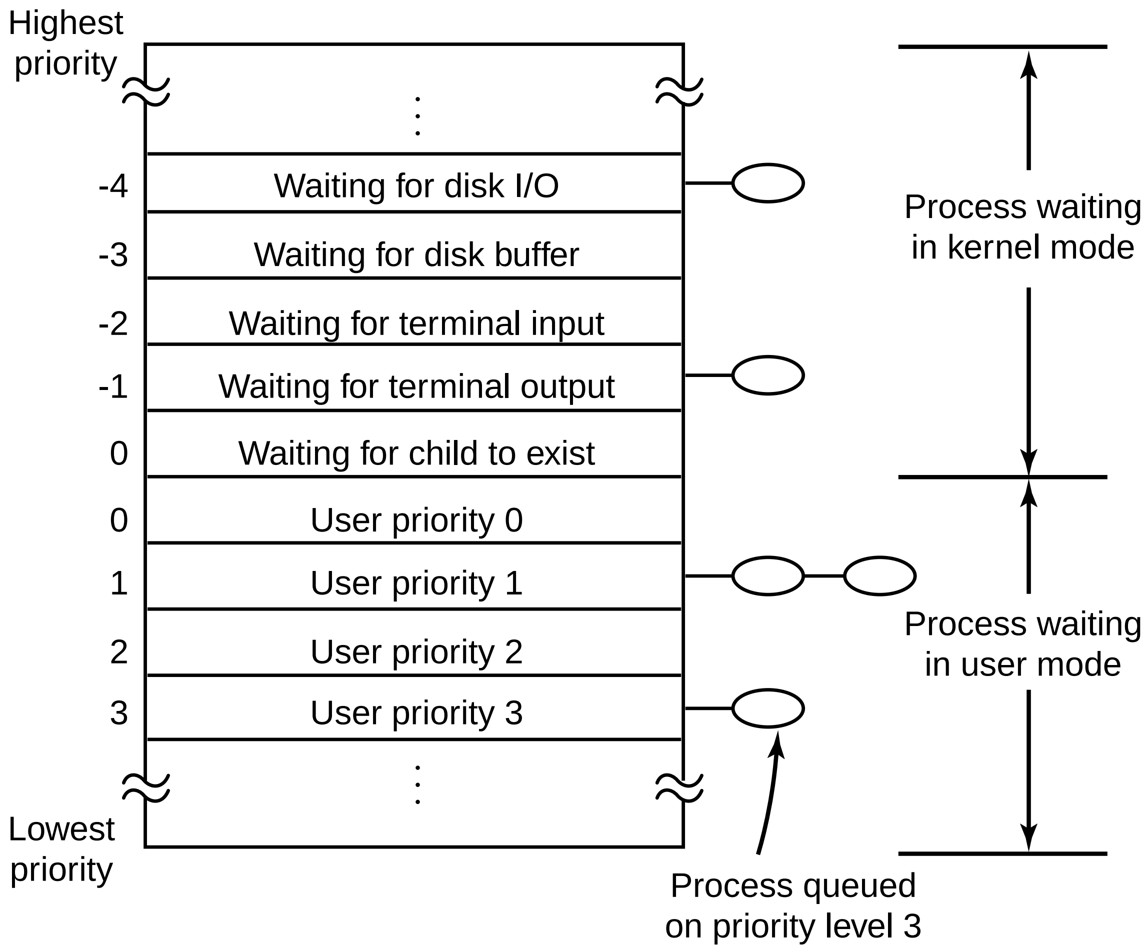

Fig. 10-11. The UNIX scheduler is based on a multilevel queue structure.

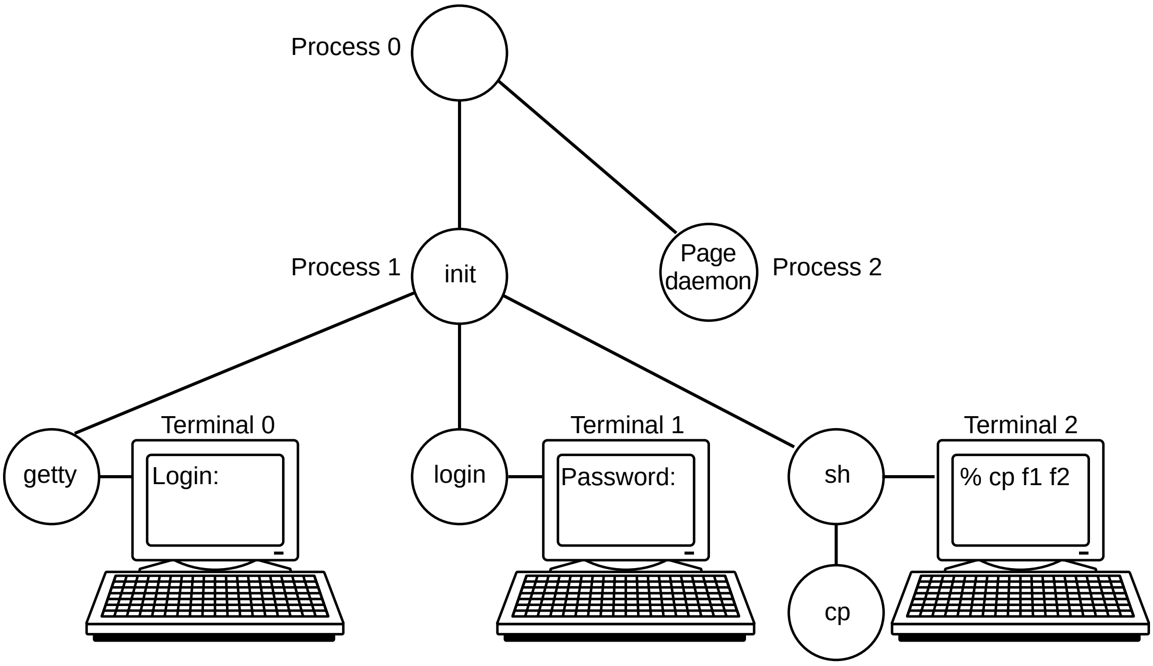

Fig. 10-12. The sequence of processes used to boot some UNIX systems.

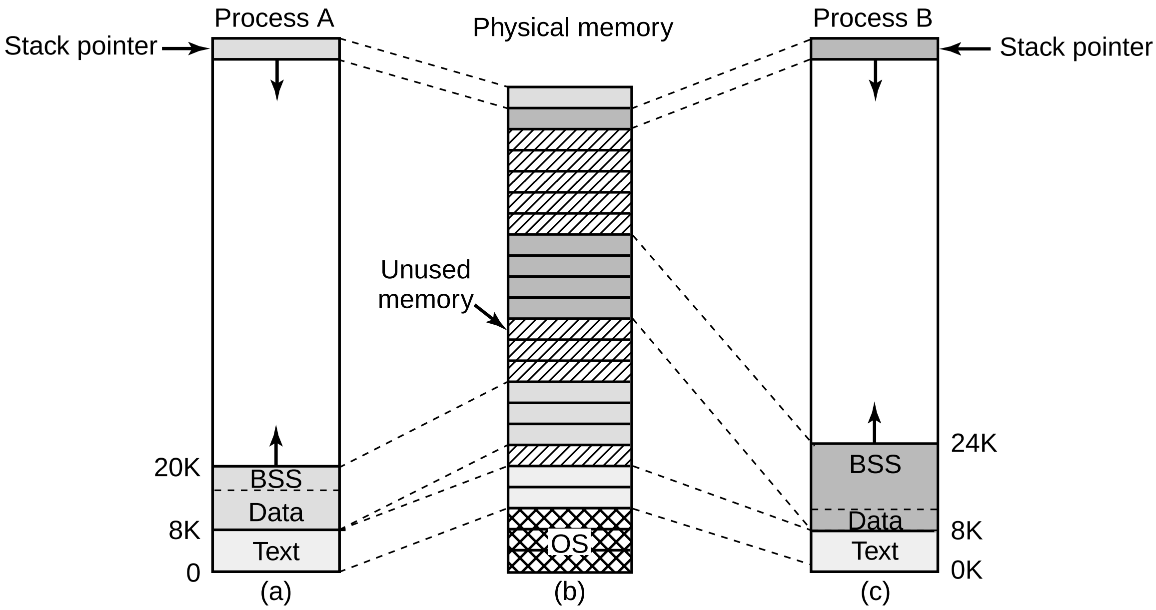

Fig. 10-13. (a) Process *A*'s virtual address space. (b) Physical memory. (c) Process *B*'s virtual address space.

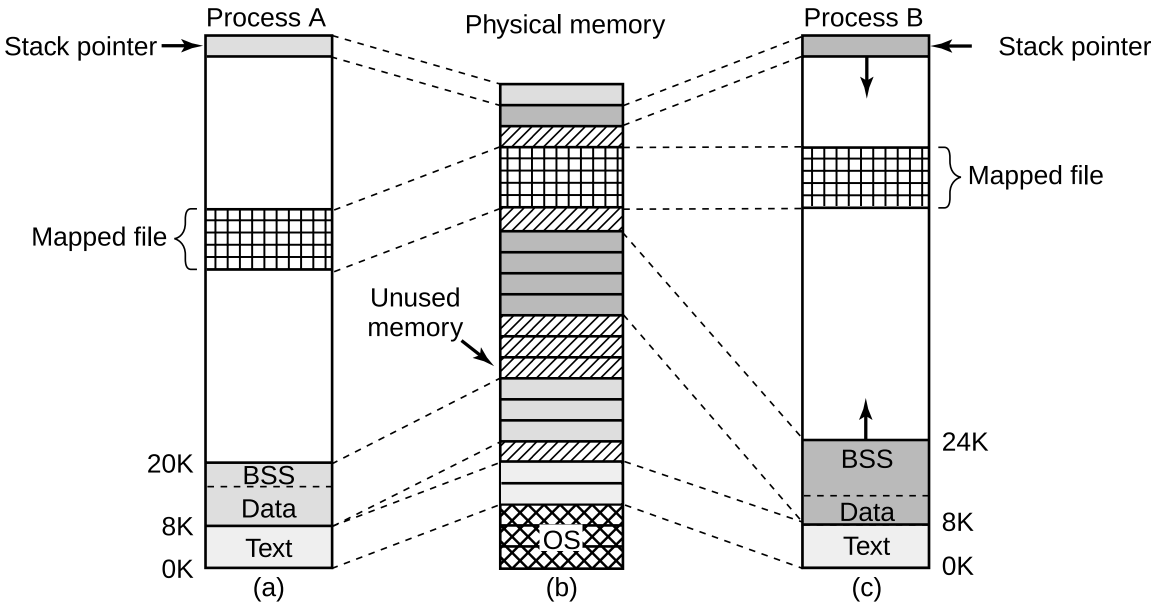

Fig. 10-14. Two processes can share a mapped file.

System call	Description
s = brk(addr)	Change data segment size
a = mmap(addr, len, prot, flags, fd, offset)	Map a file in
s = unmap(addr, len)	Unmap a file

Fig. 10-15. Some system calls relating to memory management. The return code *s* is –1 if an error has occurred; *a* and *addr* are memory addresses, *len* is a length, *prot* controls protection, *flags* are miscellaneous bits, *fd* is a file descriptor, and *offset* is a file offset.

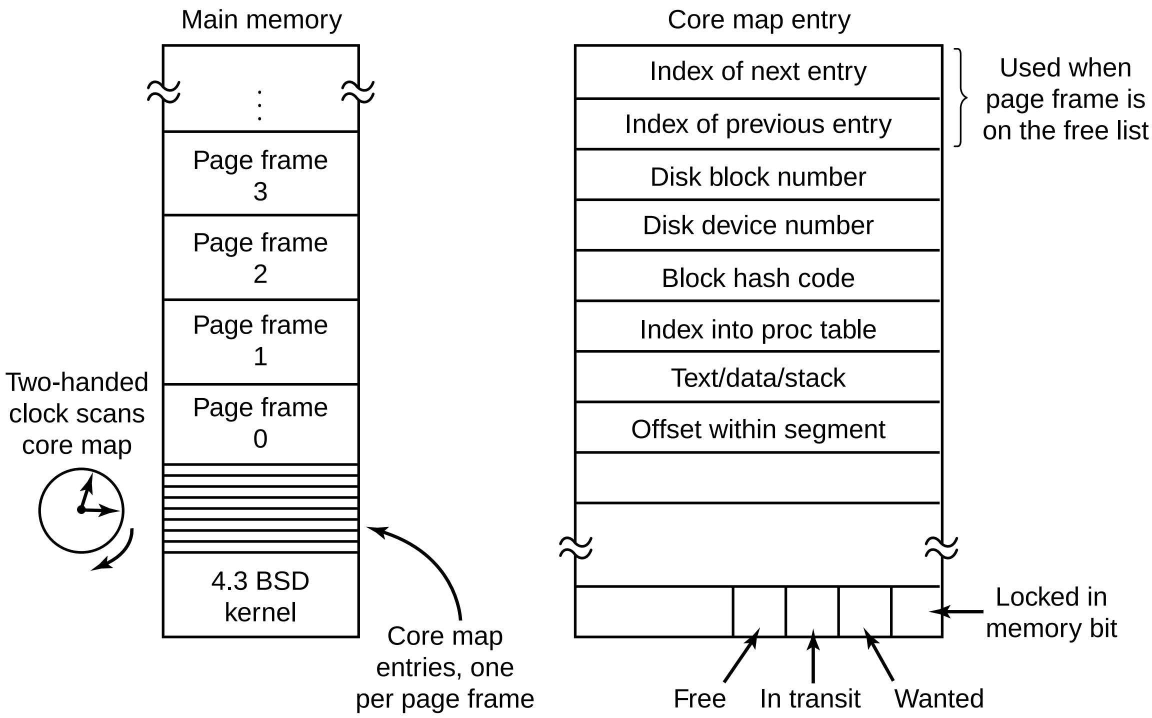

Fig. 10-16. The core map in 4BSD.

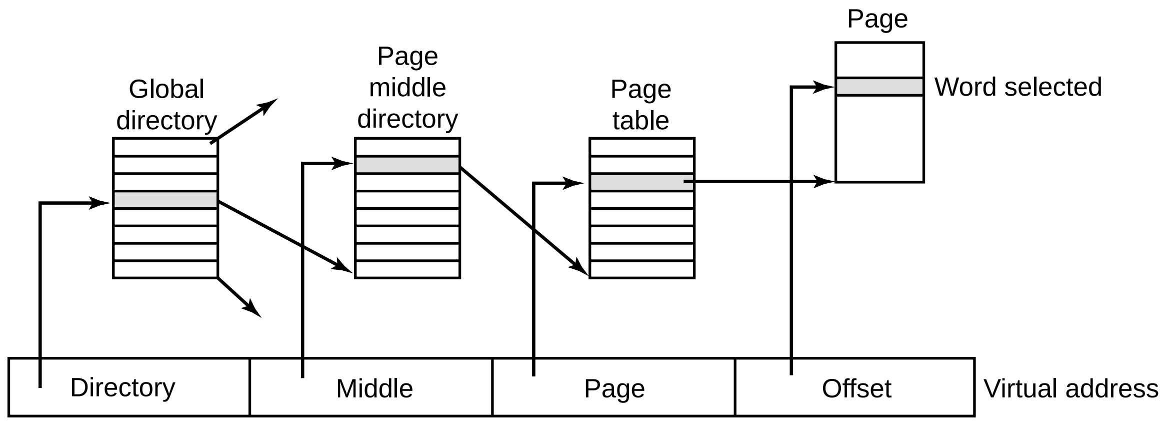

Fig. 10-17. Linux uses three-level page tables.

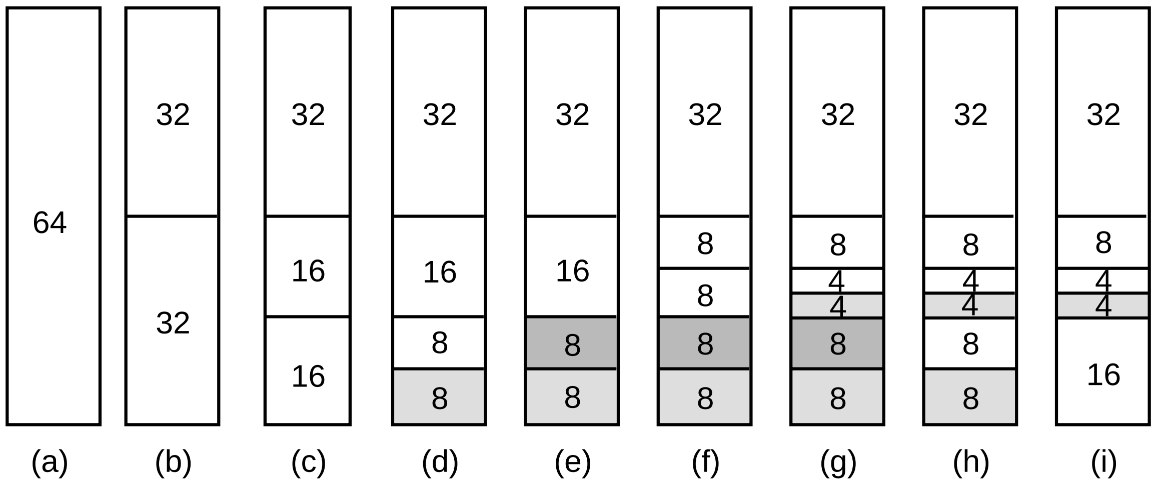

Fig. 10-18. Operation of the buddy algorithm.

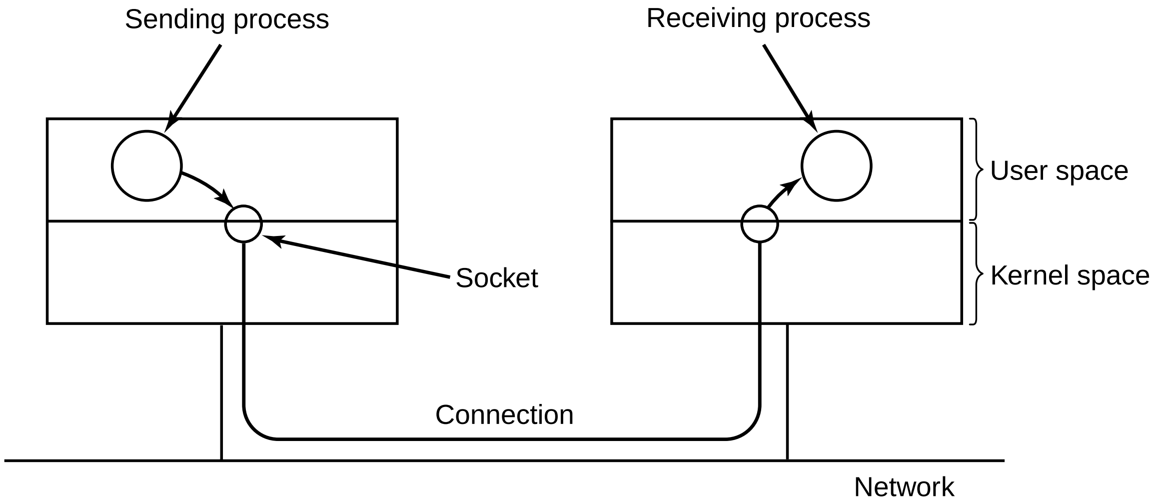

Fig. 10-19. The uses of sockets for networking.

Function call	Description
s = cfsetospeed(&termios, speed)	Set the output speed
s = cfsetispeed(&termios, speed)	Set the input speed
s = cfgetospeed(&termios, speed)	Get the output speed
s = cfgtetispeed(&termios, speed)	Get the input speed
s = tcsetattr(fd, opt, &termios)	Set the attributes
s = tcgetattr(fd, &termios)	Get the attributes

Fig. 10-20. The main POSIX calls for managing the terminal.

Device	Open	Close	Read	Write	Ioctl	Other
Null	null	null	null	null	null	...
Memory	null	null	mem_read	mem_write	null	...
Keyboard	k_open	k_close	k_read	error	k_ioctl	...
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	...
Printer	lp_open	lp_close	error	lp_write	lp_ioctl	...

Fig. 10-21. Some of the fields of a typical *cdevsw* table.

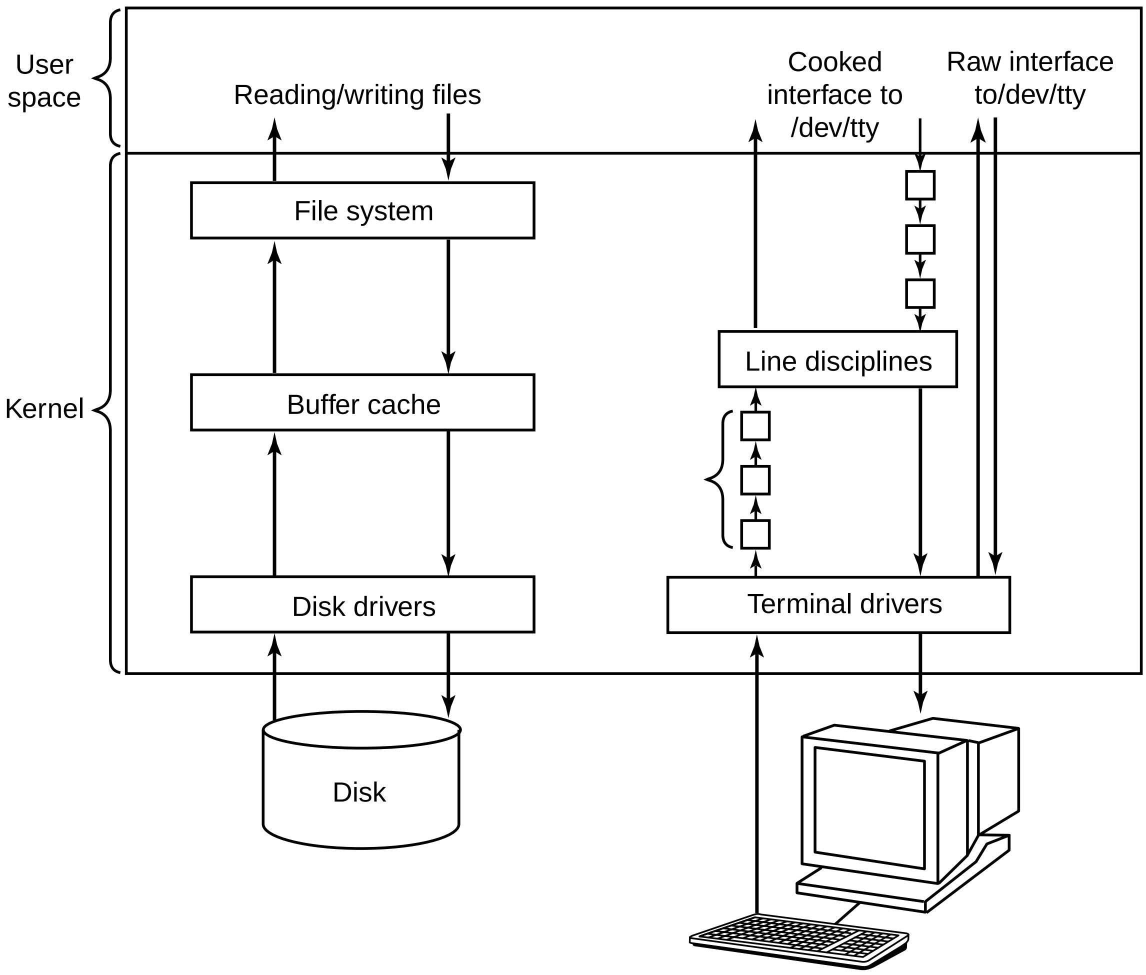

Fig. 10-22. The UNIX I/O system in BSD.

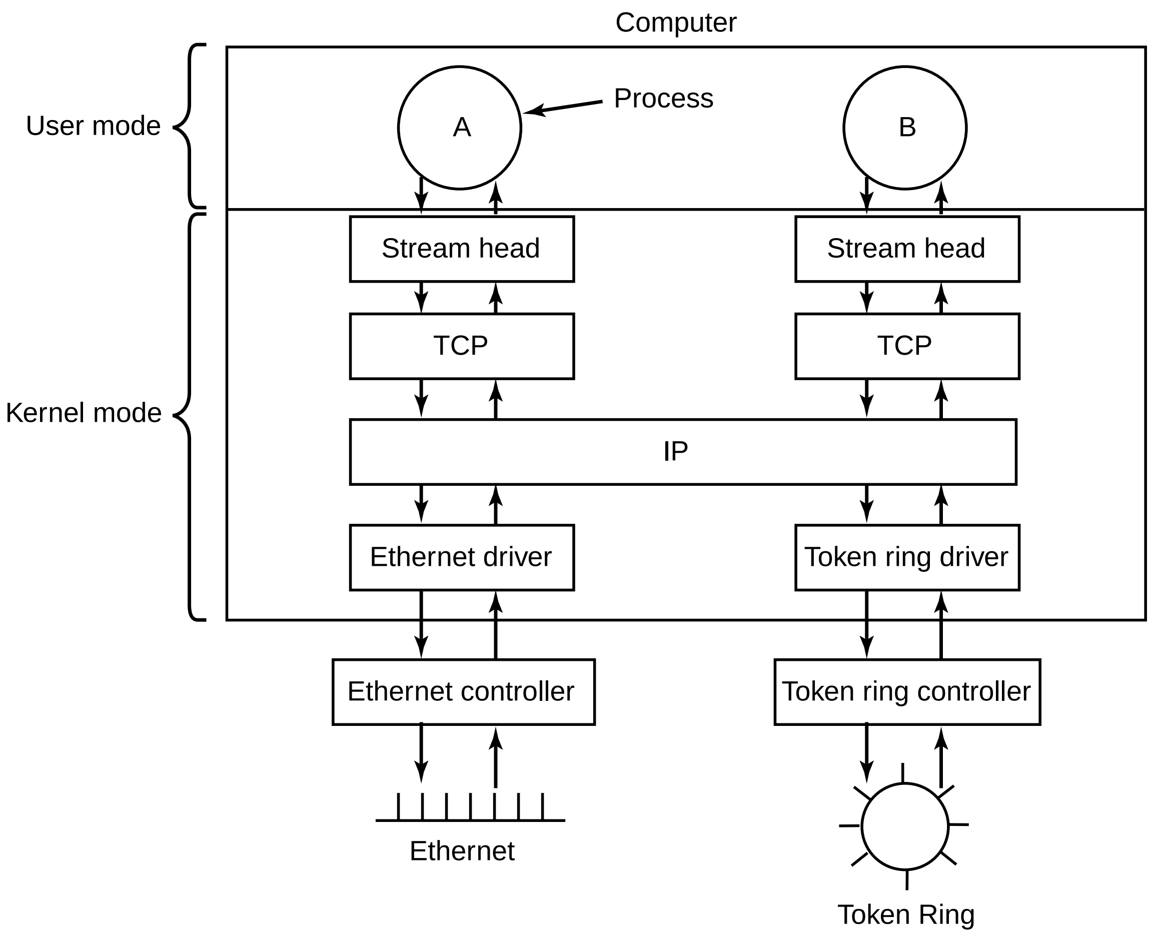

Fig. 10-23. An example of streams in System V.

Directory	Contents
bin	Binary (executable) programs
dev	Special files for I/O devices
etc	Miscellaneous system files
lib	Libraries
usr	User directories

Fig. 10-24. Some important directories found in most UNIX systems.

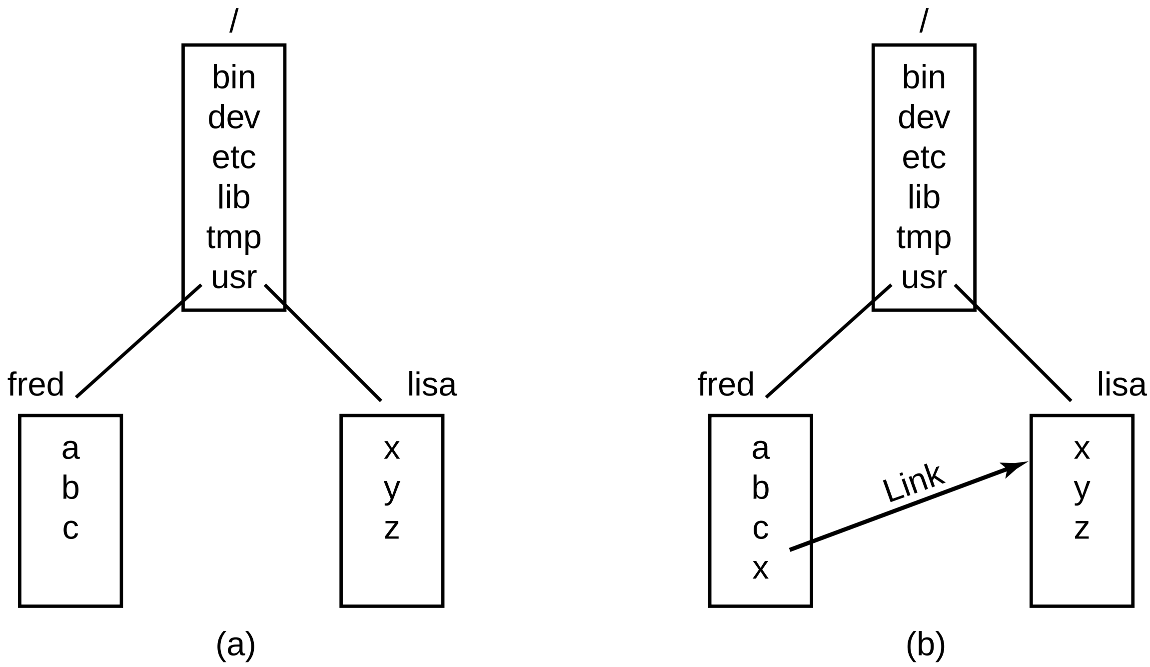

Fig. 10-25. (a) Before linking. (b) After linking.

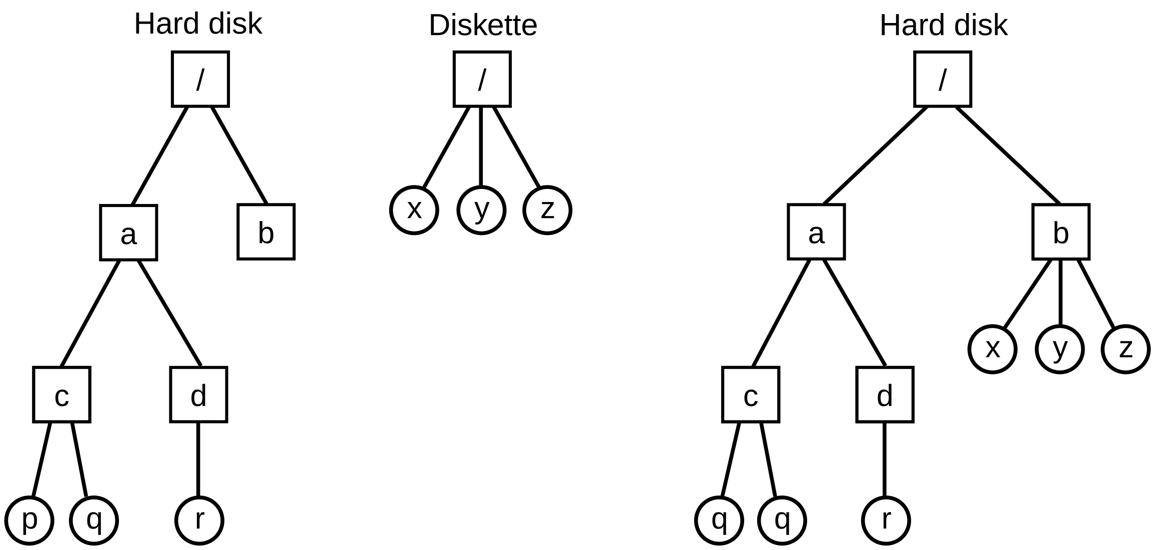

Fig. 10-26. (a) Separate file systems. (b) After mounting.

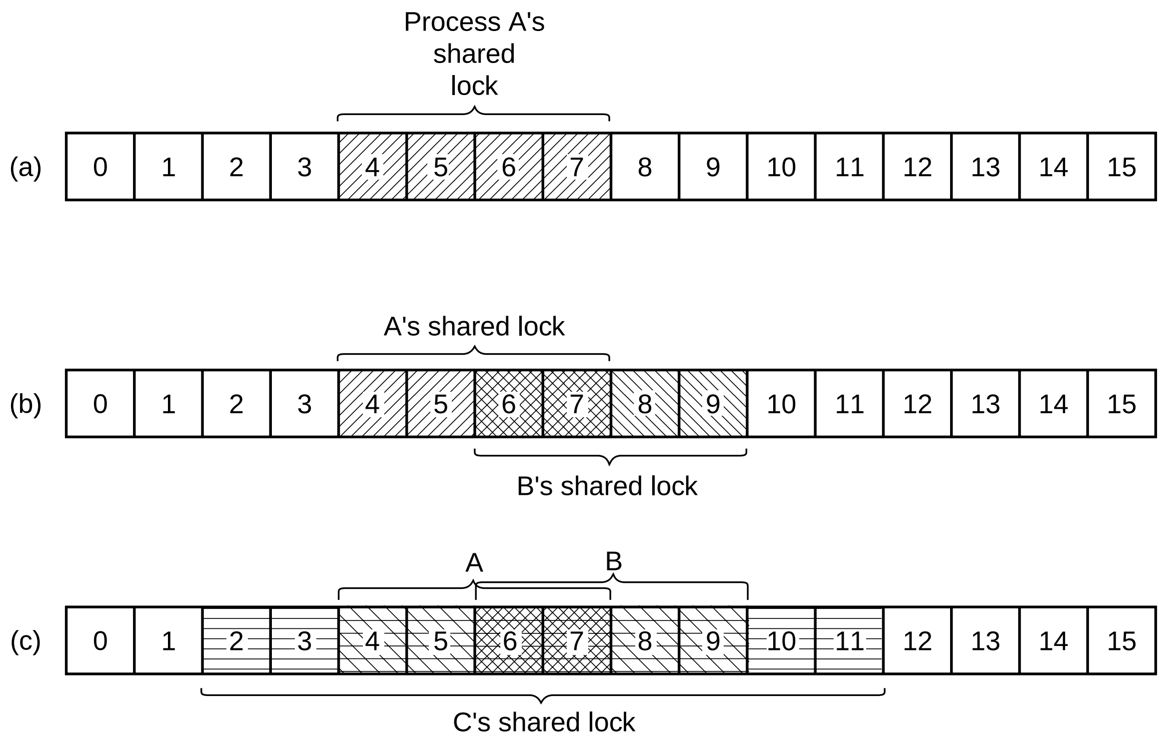

Fig. 10-27. (a) A file with one lock. (b) Addition of a second lock. (c) A third lock.

System call	Description
fd = creat(name, mode)	One way to create a new file
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information
s = fstat(fd, &buf)	Get a file's status information
s = pipe(&fd[0])	Create a pipe
s = fcntl(fd, cmd, ...)	File locking and other operations

Fig. 10-28. Some system calls relating to files. The return code *s* is −1 if an error has occurred; *fd* is a file descriptor, and *position* is a file offset. The parameters should be self explanatory.

Device the file is on
I-node number (which file on the device)
File mode (includes protection information)
Number of links to the file
Identity of the file’s owner
Group the file belongs to
File size (in bytes)
Creation time
Time of last access
Time of last modification

Fig. 10-29. The fields returned by the stat system call.

System call	Description
s = mkdir(path, mode)	Create a new directory
s = rmdir(path)	Remove a directory
s = link(oldpath, newpath)	Create a link to an existing file
s = unlink(path)	Unlink a file
s = chdir(path)	Change the working directory
dir = opendir(path)	Open a directory for reading
s = closedir(dir)	Close a directory
dirent = readdir(dir)	Read one directory entry
rewinddir(dir)	Rewind a directory so it can be reread

Fig. 10-30. Some system calls relating to directories. The return code *s* is −1 if an error has occurred; *dir* identifies a directory stream and *dirent* is a directory entry. The parameters should be self explanatory.

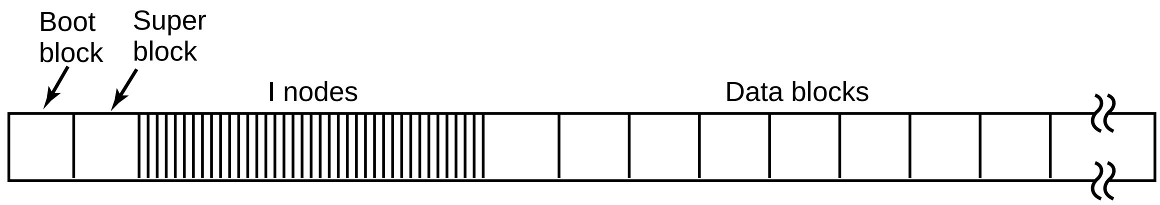

Fig. 10-31. Disk layout in classical UNIX systems.

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Fig. 10-32. Structure of the i-node in System V.

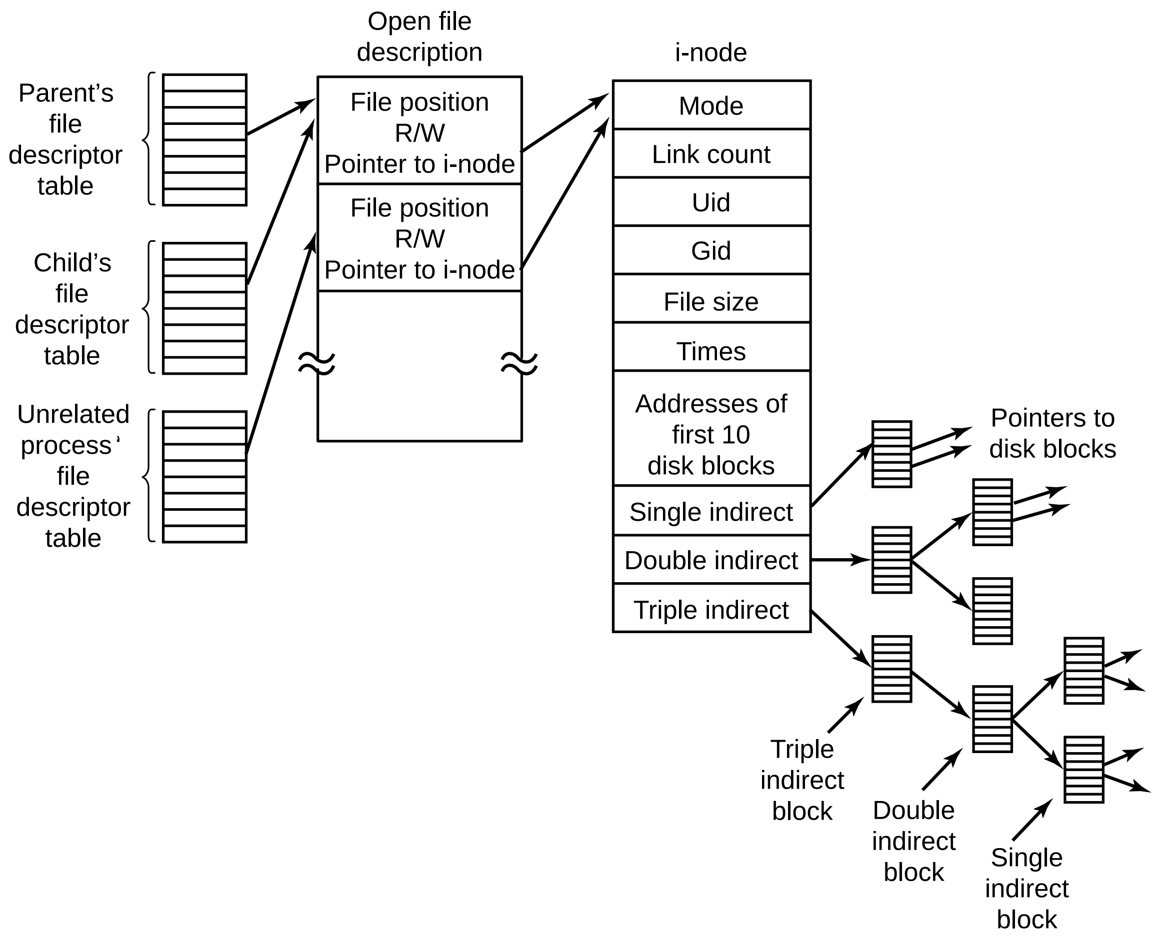

Fig. 10-33. The relation between the file descriptor table, the open file description table, and the i-node table.

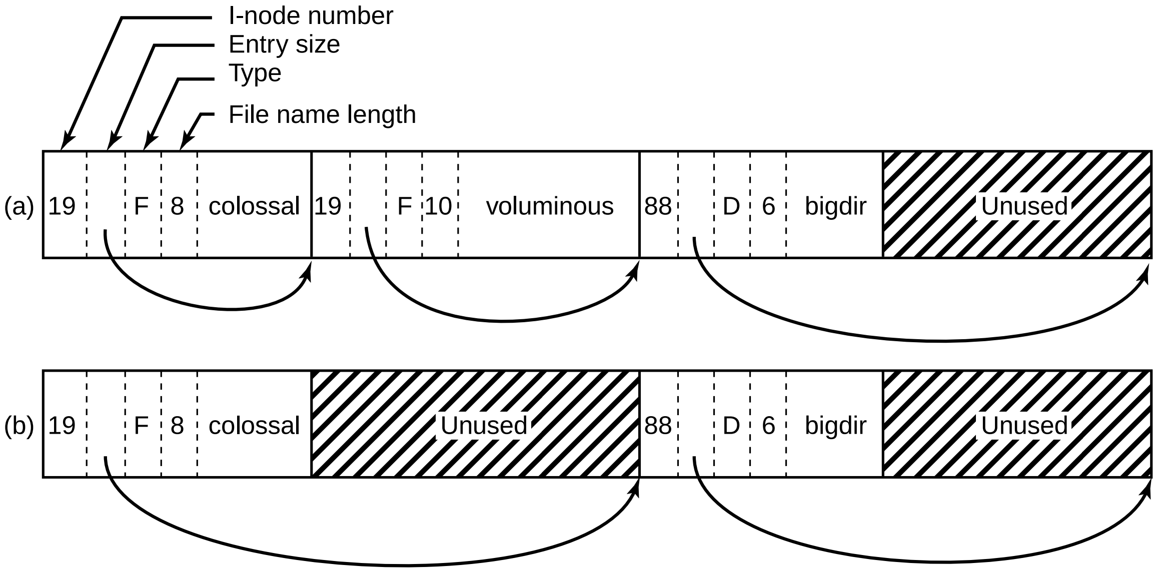

Fig. 10-34. (a) A BSD directory with three files. (b) The same directory after the file *voluminous* has been removed.

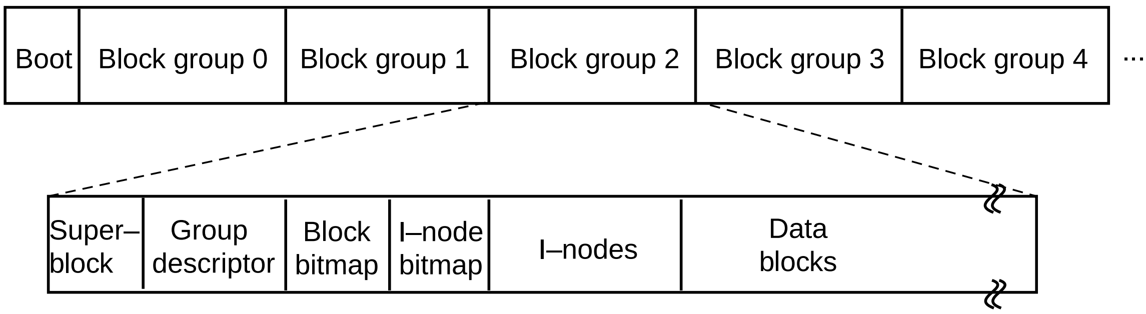

Fig. 10-35. Layout of the Linux Ext2 file system.

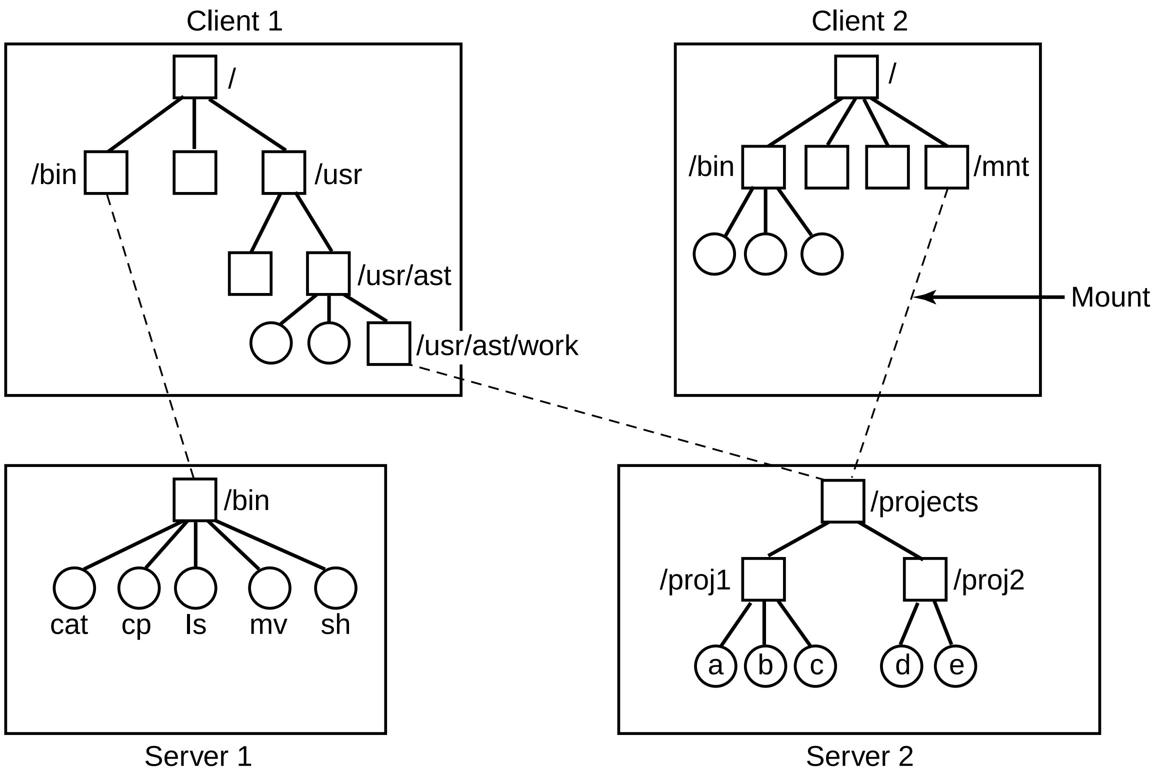

Fig. 10-36. Examples of remote mounted file systems. Directories are shown as squares and files are shown as circles.

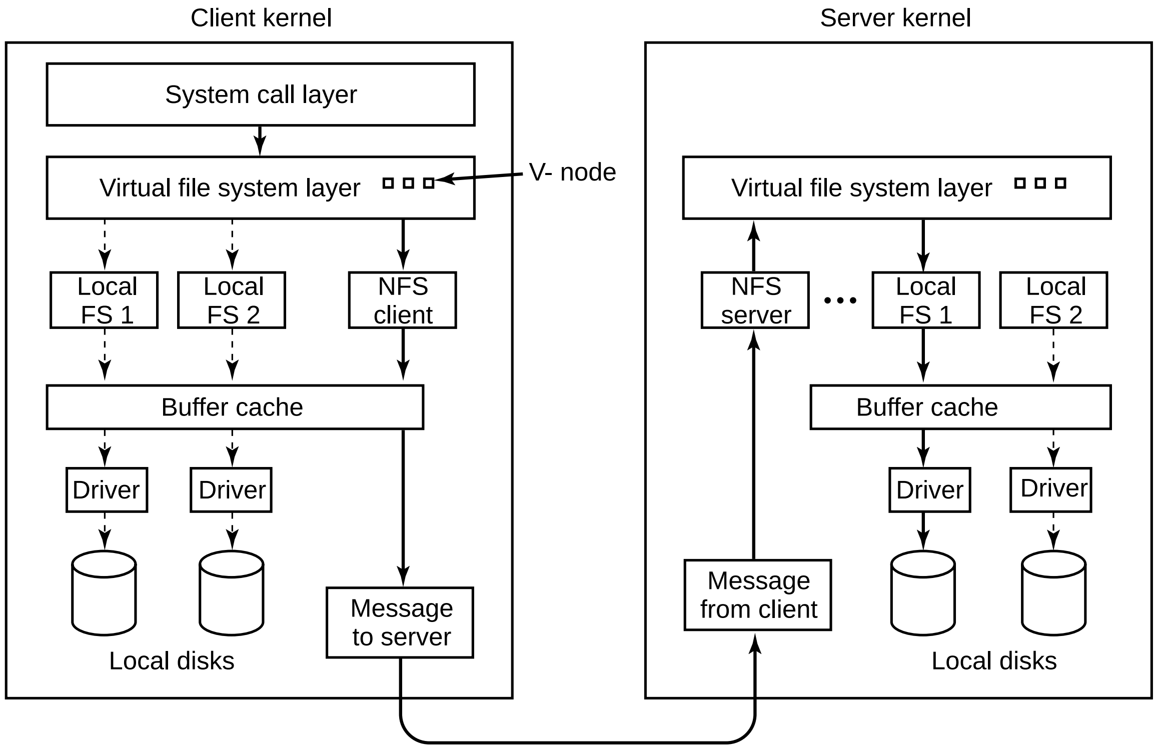

Fig. 10-37. The NFS layer structure.

Binary	Symbolic	Allowed file accesses
111000000	rwx––––––	Owner can read, write, and execute
111111000	rwxrwx–––	Owner and group can read, write, and execute
110100000	rw–r–––––	Owner can read and write; group can read
110100100	rw–r––r––	Owner can read and write; all others can read
111101101	rwxr–xr–x	Owner can do everything, rest can read and execute
000000000	–––––––––	Nobody has any access
000000111	––––––rwx	Only outsiders have access (strange, but legal)

Fig. 10-38. Some example file protection modes.

System call	Description
s = chmod(path, mode)	Change a file’s protection mode
s = access(path, mode)	Check access using the real UID and GID
uid = getuid( )	Get the real UID
uid = geteuid( )	Get the effective UID
gid = getgid( )	Get the real GID
gid = getegid( )	Get the effective GID
s = chown(path, owner, group)	Change owner and group
s = setuid(uid)	Set the UID
s = setgid(gid)	Set the GID

Fig. 10-39. Some system calls relating to security. The return code *s* is –1 if an error has occurred; *uid* and *gid* are the UID and GID, respectively. The parameters should be self explanatory.