OPERATING SYSTEMS: DESIGN AND IMPLEMENTATION

Second Edition

ANDREW S. TANENBAUM

Vrije Universiteit
Amsterdam, The Netherlands

ALBERT S. WOODHULL

Hampshire College
Amherst, Massachusetts

PRENTICE HALL

Upper Saddle River, NJ 07458

1

INTRODUCTION

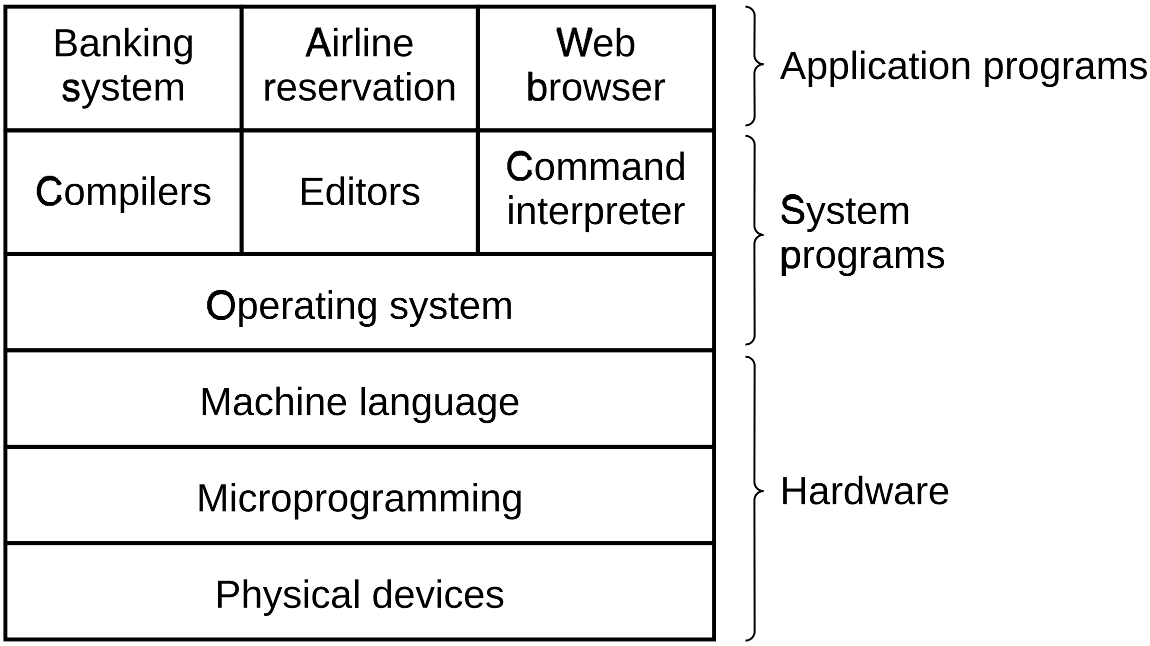

Figure 1-1. A computer system consists of hardware, system programs, and application programs.

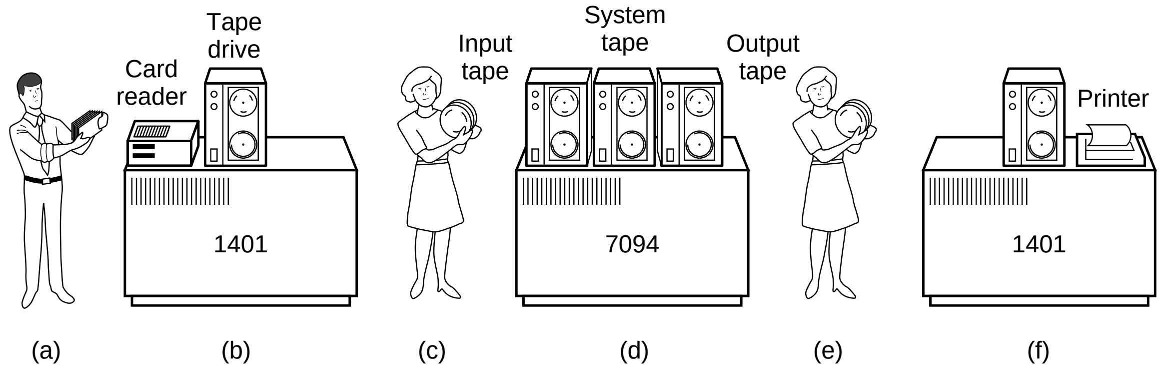

Figure 1-2. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

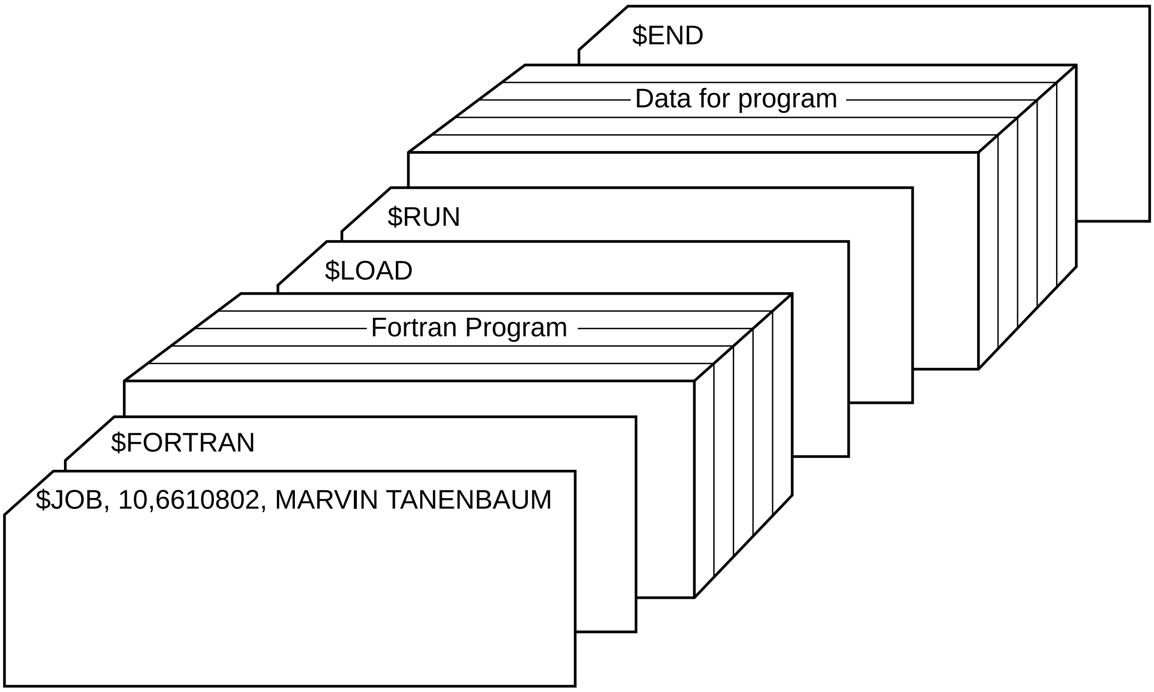

Figure 1-3. Structure of a typical FMS job.

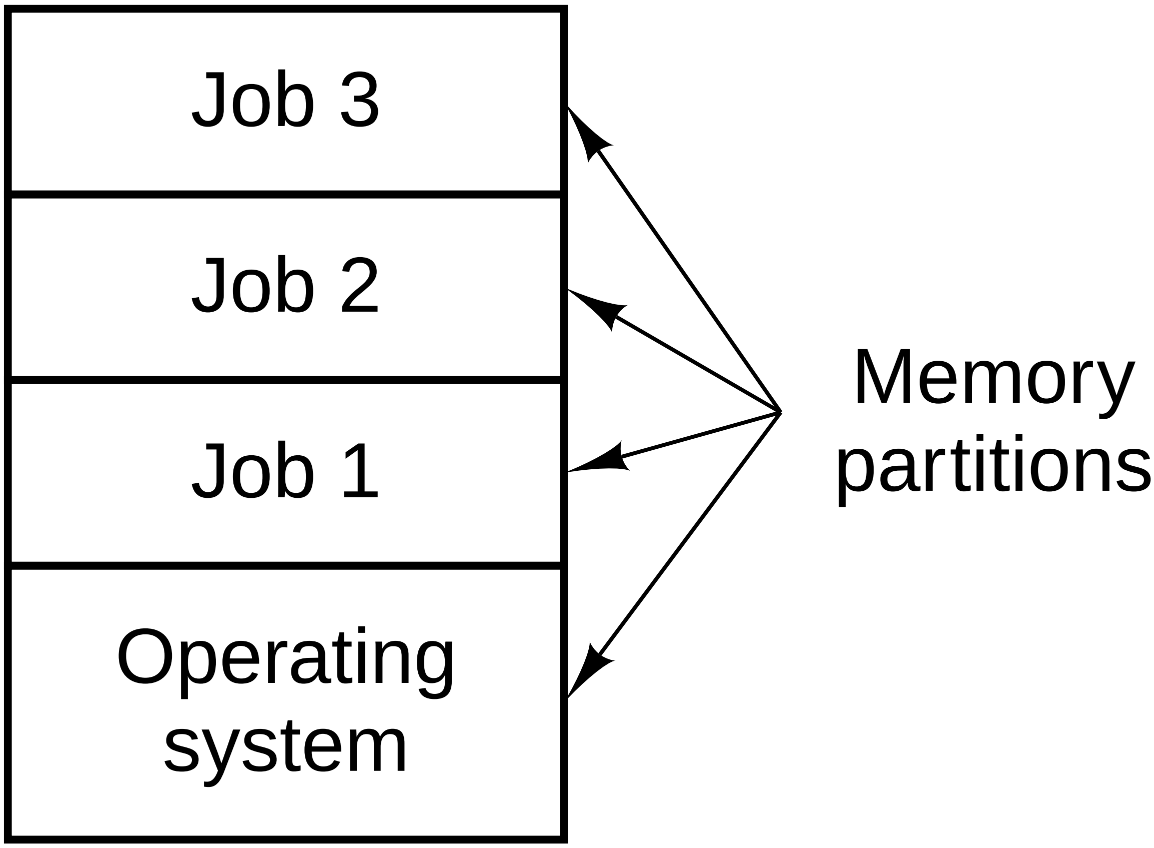

Figure 1-4. A multiprogramming system with three jobs in memory.

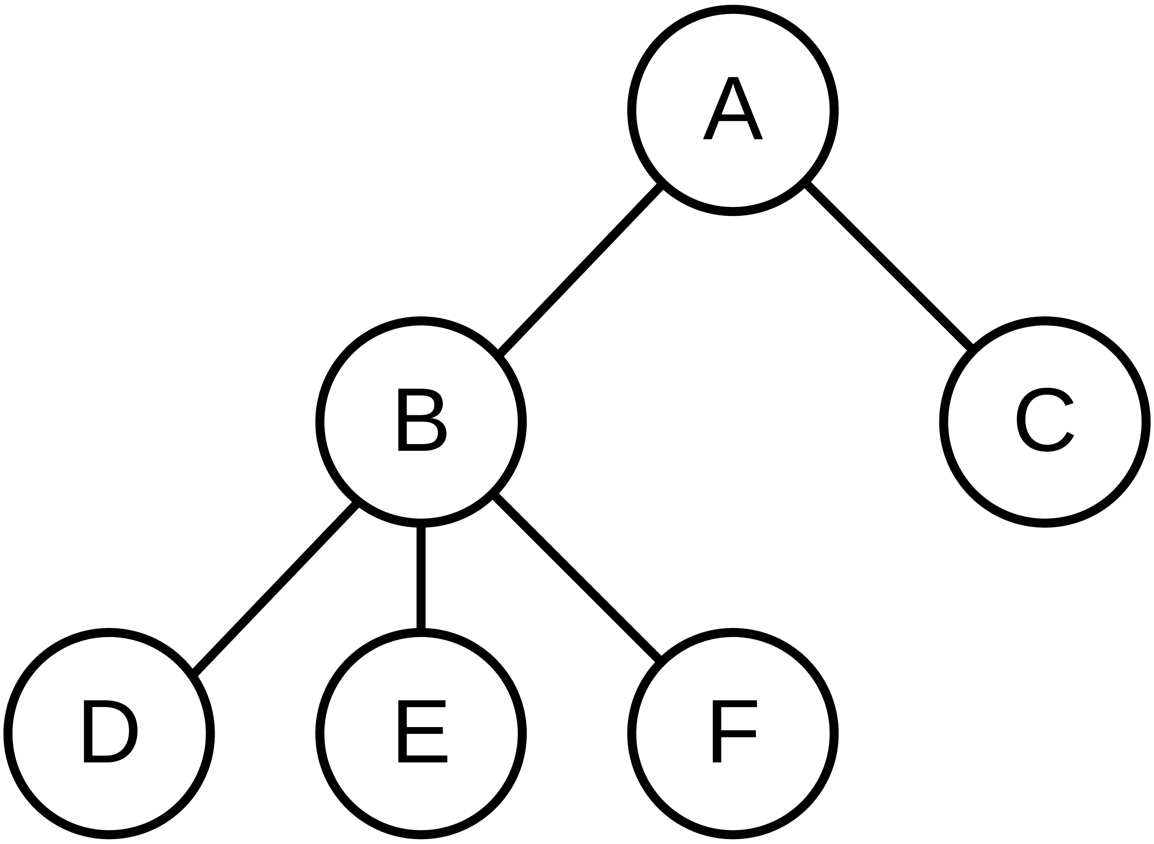

Figure 1-5. A process tree. Process *A* created two child processes, *B* and *C*. Process *B* created three child processes, *D*, *E*, and *F*.

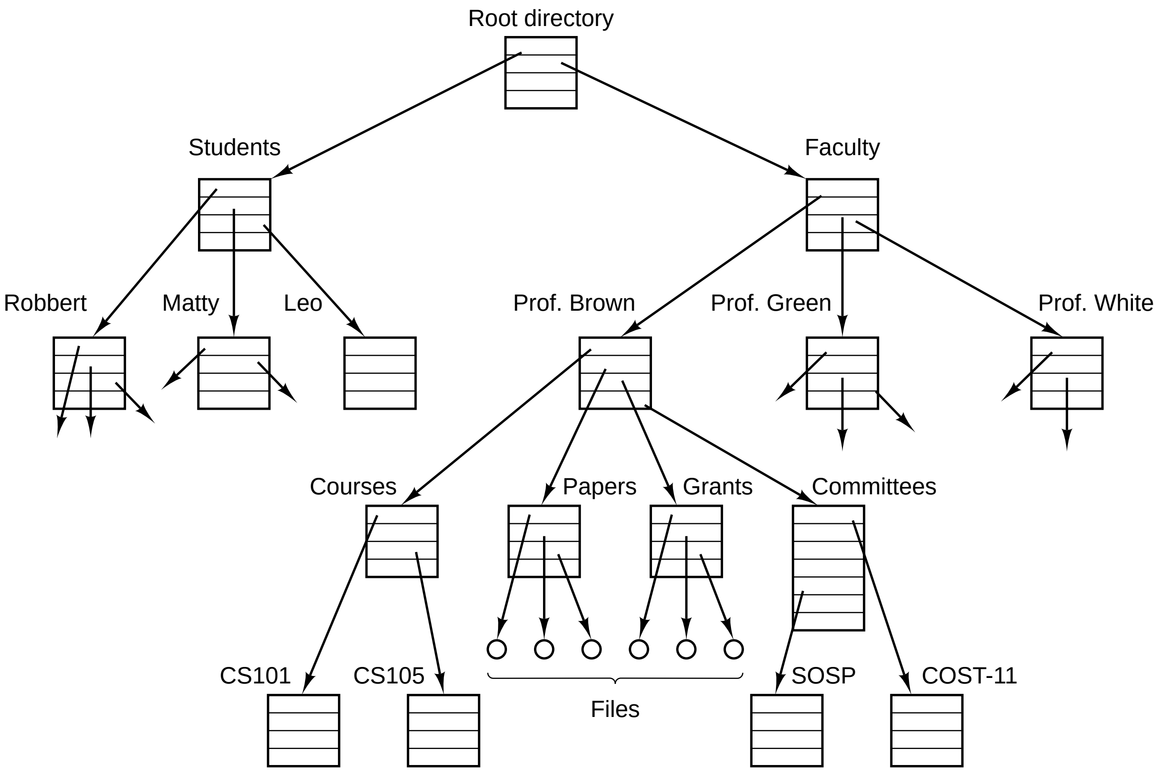

Figure 1-6. A file system for a university department.

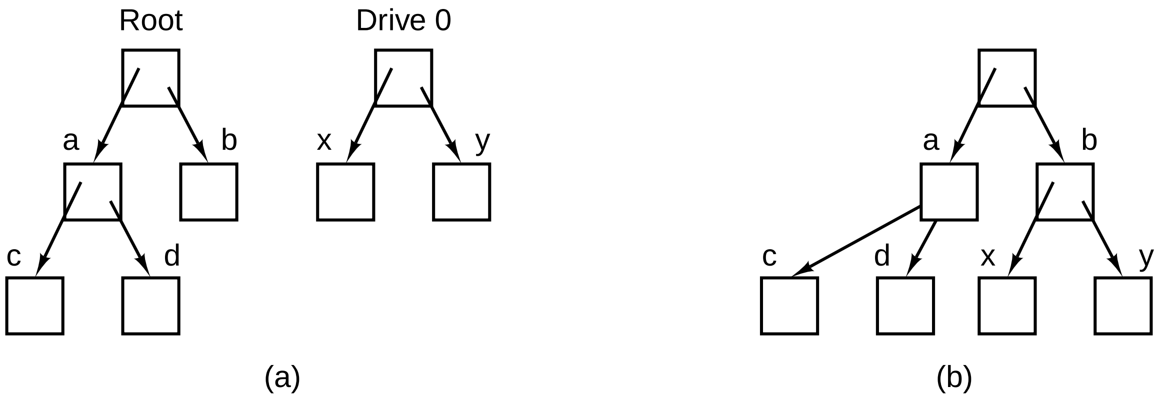

Figure 1-7. (a) Before mounting, the files on drive 0 are not accessible. (b) After mounting, they are part of the file hierarchy.

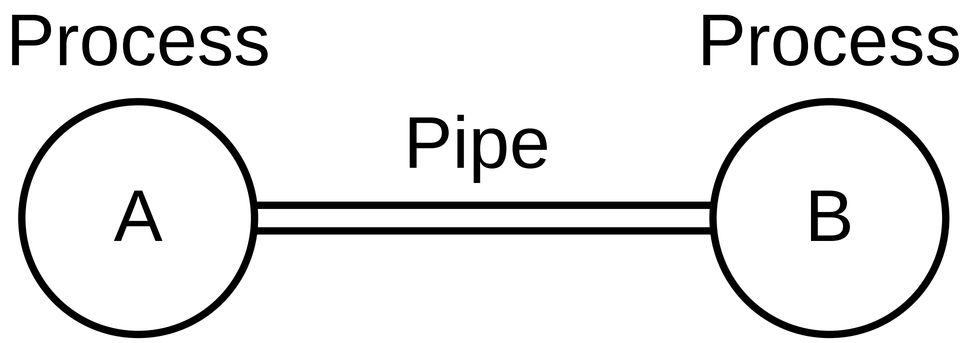

Figure 1-8. Two processes connected by a pipe.

Process management	pid = fork()	Create a child process identical to the parent
	pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
	s = wait(&status)	Old version of waitpid
	s = execve(name, argv, envp)	Replace a process core image
	exit(status)	Terminate process execution and return status
	size = brk(addr)	Set the size of the data segment
	pid = getpid()	Return the caller's process id
	pid = getpgrp()	Return the id of the caller's process group
	pid = setsid()	Create a new session and return its process group id
	l = ptrace(req, pid, addr, data)	Used for debugging
Signals	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
File Management	fd = creat(name, mode)	Obsolete way to create a new file
	fd = mknod(name, mode, addr)	Create a regular, special, or directory i-node
	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
	pos = 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
	fd = dup(fd)	Allocate a new file descriptor for an open file
	s = pipe(&fd[0])	Create a pipe
	s = ioctl(fd, request, argp)	Perform special operations on a file
	s = access(name, amode)	Check a file's accessibility
	s = rename(old, new)	Give a file a new name
	s = fcntl(fd, cmd, ...)	File locking and other operations
Directory & File System Management	s = mkdir(name, mode)	Create a new directory
	s = rmdir(name)	Remove an empty directory
	s = link(name1, name2)	Create a new entry, name2, pointing to name1
	s = unlink(name)	Remove a directory entry
	s = mount(special, name, flag)	Mount a file system
	s = umount(special)	Unmount a file system
	s = sync()	Flush all cached blocks to the disk
	s = chdir(dirname)	Change the working directory
	s = chroot(dirname)	Change the root directory
Protection	s = chmod(name, mode)	Change a file's protection bits
	uid = getuid()	Get the caller's uid
	gid = getgid()	Get the caller's gid
	s = setuid(uid)	Set the caller's uid
	s = setgid(gid)	Set the caller's gid
	s = chown(name, owner, group)	Change a file's owner and group
	oldmask = umask(complmode)	Change the mode mask
Time Management	seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970
	s = stime(tp)	Set the elapsed time since Jan. 1, 1970
	s = utime(file, timep)	Set a file's "last access" time
	s = times(buffer)	Get the user and system times used so far

Figure 1-9. The MINIX system calls.

```
while (TRUE) {                          /* repeat forever */
  read_command(command, parameters);/* read input from terminal */

  if (fork() != 0) {                    /* fork off child process */
      /* Parent code. */
      waitpid(–1, &status, 0);          /* wait for child to exit */
  } else {
      /* Child code. */
      execve(command, parameters, 0);/* execute command */
  }
}
```

Figure 1-10. A stripped-down shell. Throughout this book, *TRUE* is assumed to be defined as 1.

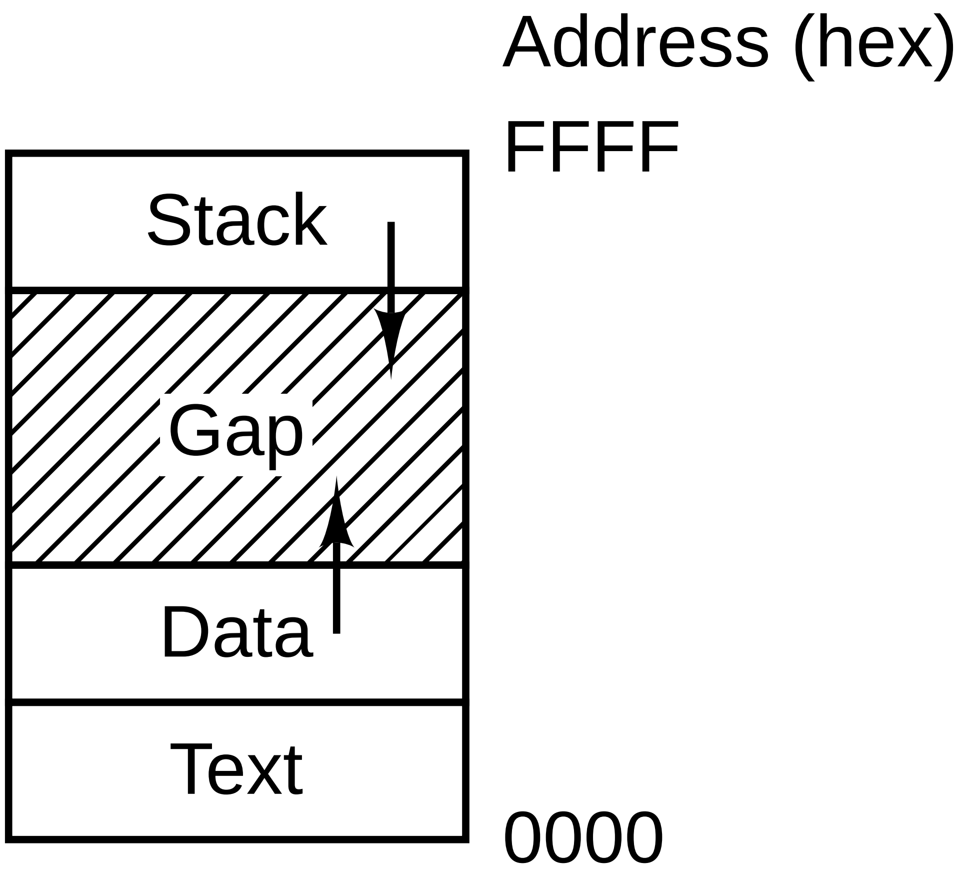

Figure 1-11. Processes have three segments: text, data, and stack. In this example, all three are in one address space, but separate instruction and data space is also supported.

```
struct stat {
    short st_dev;                 /* device where i-node belongs */
    unsigned short st_ino;   /* i-node number */
    unsigned short st_mode;   /* mode word */
    short st_nlink;               /* number of links */
    short st_uid;                 /* user id */
    short st_gid;                 /* group id */
    short st_rdev;                /* major/minor device for special files */
    long st_size;                 /* file size */
    long st_atime;                /* time of last access */
    long st_mtime;                /* time of last modification */
    long st_ctime;                /* time of last change to i-node */
};
```

Figure 1-12. The structure used to return information for the STAT and FSTAT system calls. In the actual code, symbolic names are used for some of the types.

```
#define STD_INPUT 0          /* file descriptor for standard input */
#define STD_OUTPUT 1         /* file descriptor for standard output */

pipeline(process1, process2)
char *process1, *process2;   /* pointers to program names */
{
  int fd[2];

  pipe(&fd[0]);                       /* create a pipe */
  if (fork() != 0) {
       /* The parent process executes these statements. */
       close(fd[0]);                  /* process 1 does not need to read from pipe */
       close(STD_OUTPUT);   /* prepare for new standard output */
       dup(fd[1]);                    /* set standard output to fd[1] */
       close(fd[1]);                  /* this file descriptor not needed any more */
       execl(process1, process1, 0);
  } else {
       /* The child process executes these statements. */
       close(fd[1]);                  /* process 2 does not need to write to pipe */
       close(STD_INPUT);        /* prepare for new standard input */
       dup(fd[0]);                    /* set standard input to fd[0] */
       close(fd[0]);                  /* this file descriptor not needed any more */
       execl(process2, process2, 0);
  }
}
```

Figure 1-13. A skeleton for setting up a two-process pipeline.

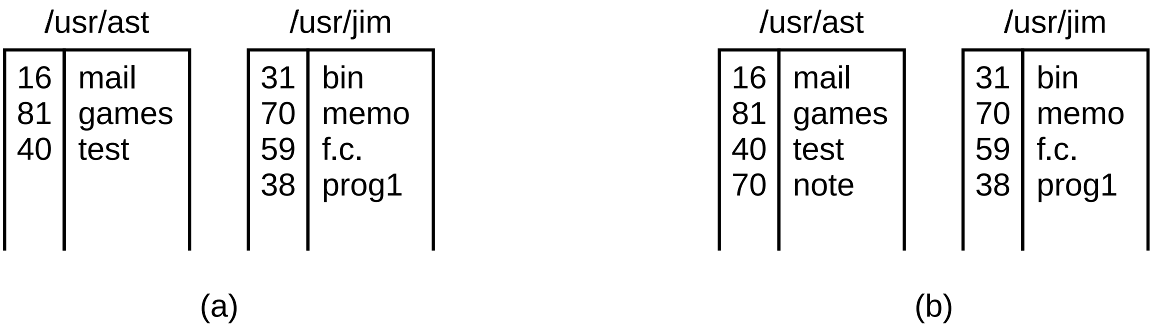

Figure 1-14. (a) Two directories before linking */usr/jim/memo* to ast’s directory. (b) The same directories after linking.

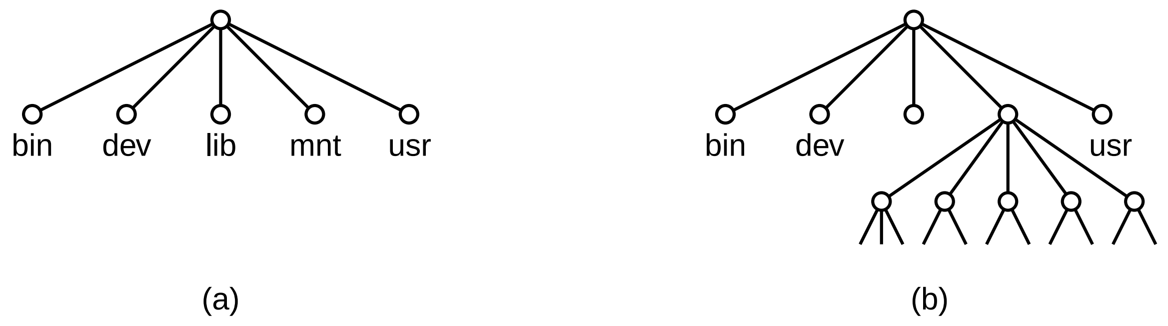

Figure 1-15. (a) File system before the mount. (b) File system after the mount.

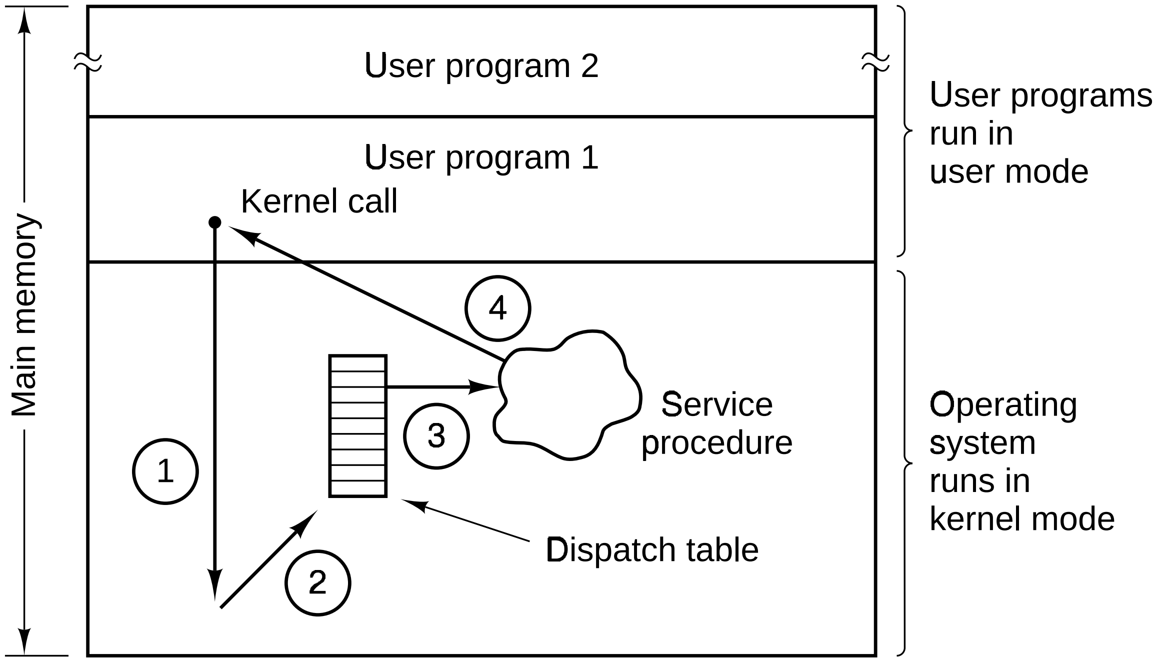

Figure 1-16. How a system call can be made: (1) User program traps to the kernel. (2) Operating system determines service number required. (3) Operating system calls service procedure. (4) Control is returned to user program.

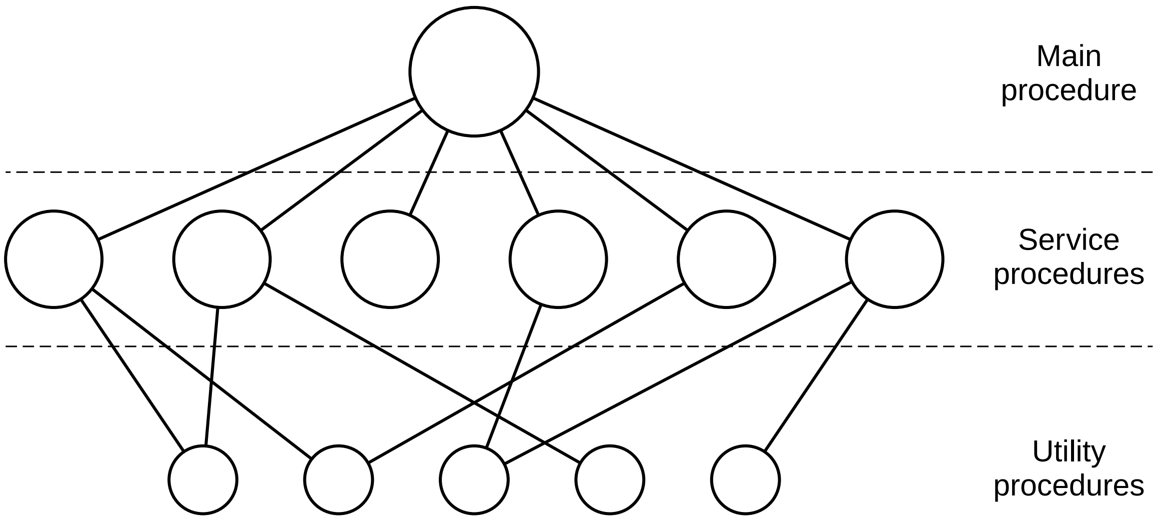

Figure 1-17. A simple structuring model for a monolithic system.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 1-18. Structure of the THE operating system.

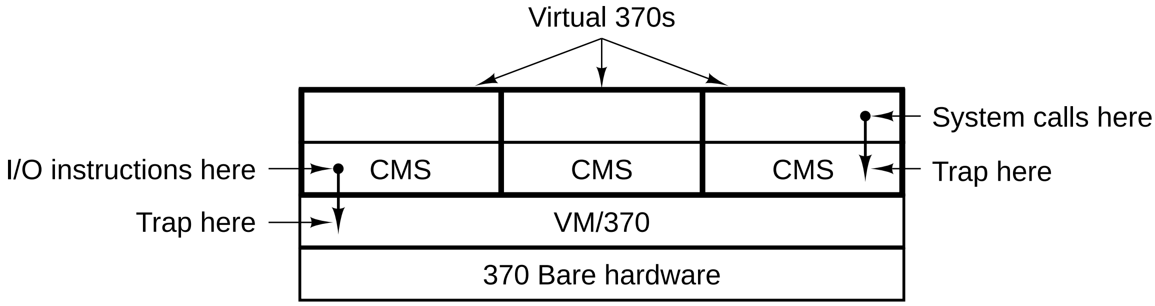

Figure 1-19. The structure of VM/370 with CMS.

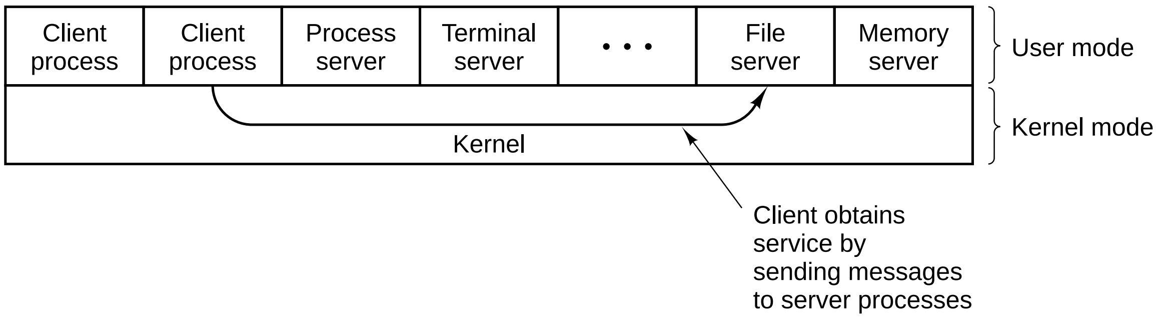

Figure 1-20. The client-server model.

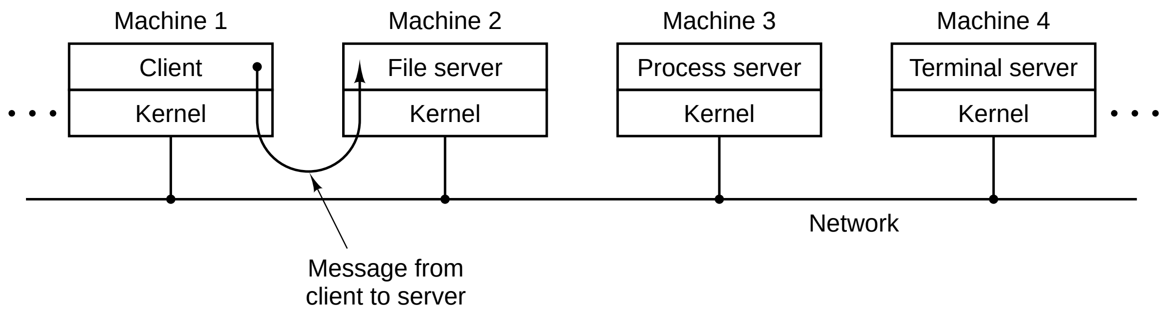

Figure 1-21. The client-server model in a distributed system.