

# OPERATING SYSTEMS DESIGN AND IMPLEMENTATION

Third Edition

ANDREW S. TANENBAUM

ALBERT S. WOODHULL

## Chapter 1 Introduction

# The Modern Computer System

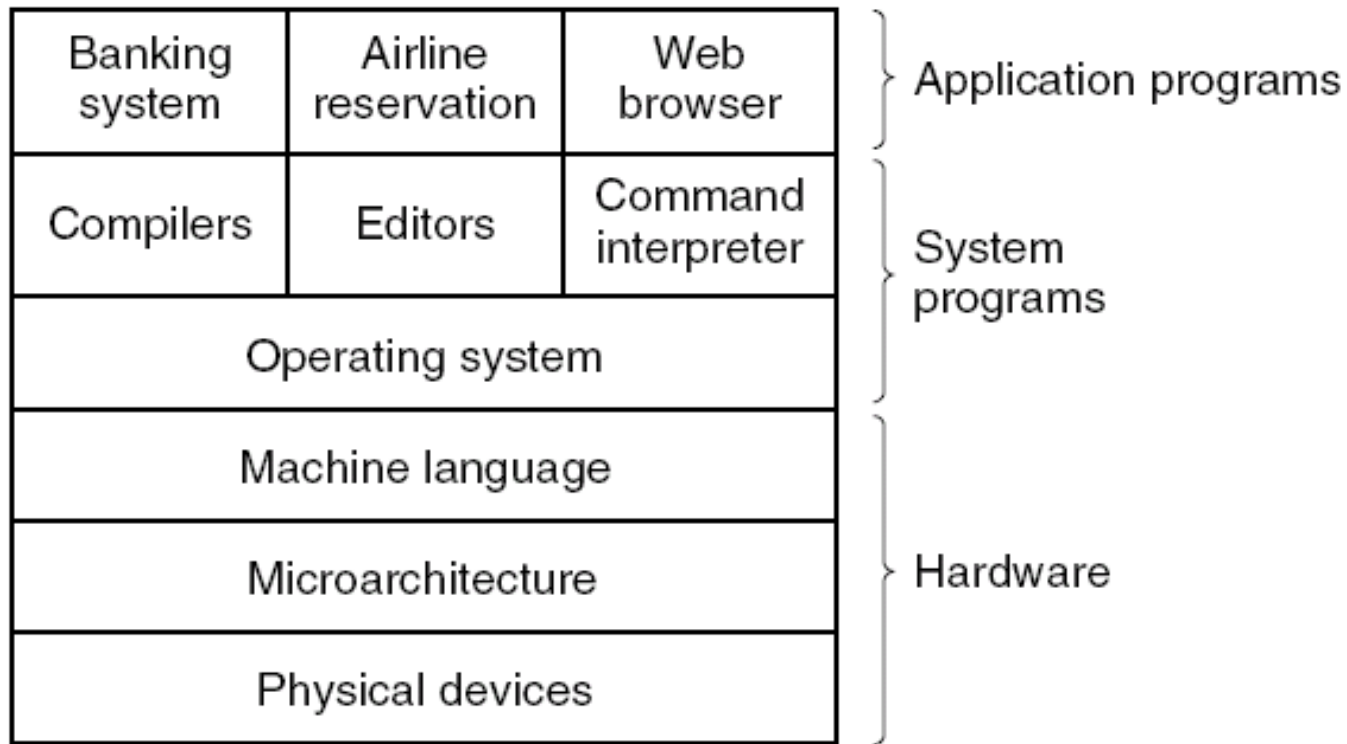


Figure 1.1 A computer system consists of hardware, system programs, and application programs.

# What Is an Operating System?

The operating system has two basic functions of the operating system

- It is an extended machine or virtual machine
  - Easier to program than the underlying hardware
- It is a resource manager
  - Shares resources in time and space

# Operating System Generations

- Generation 1 (1945 – 55)  
Vacuum tubes and plugboards
- Generation 2 (1955 – 65)  
Transistors and batch systems
- Generation 3 (1965 – 80)  
ICs and multiprogramming
- Generation 4 (1980 – Present)  
Personal computers

# Early Batch System (1)

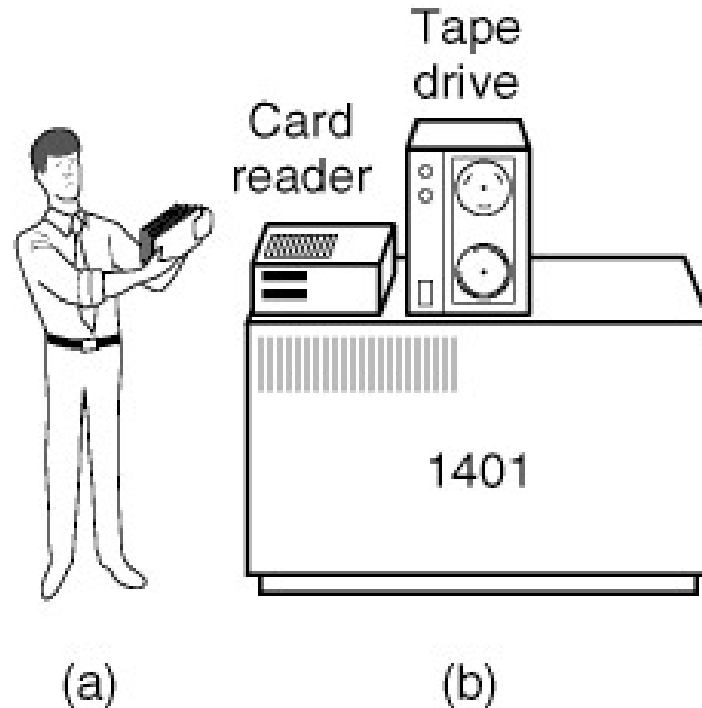


Figure 1-2. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape.

# Early Batch System (2)

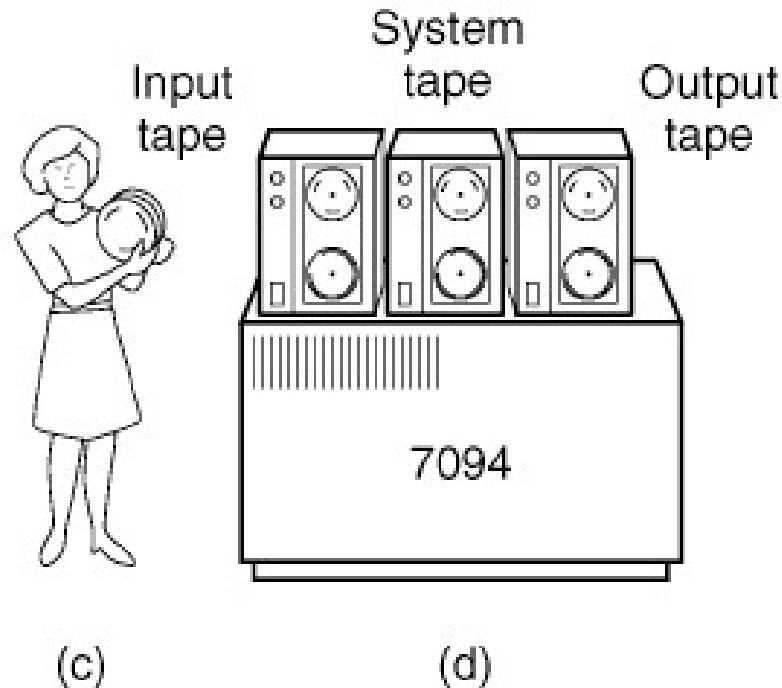


Figure 1-2. An early batch system. (c) Operator carries input tape to 7094. (d) 7094 does computing.

# Early Batch System (3)

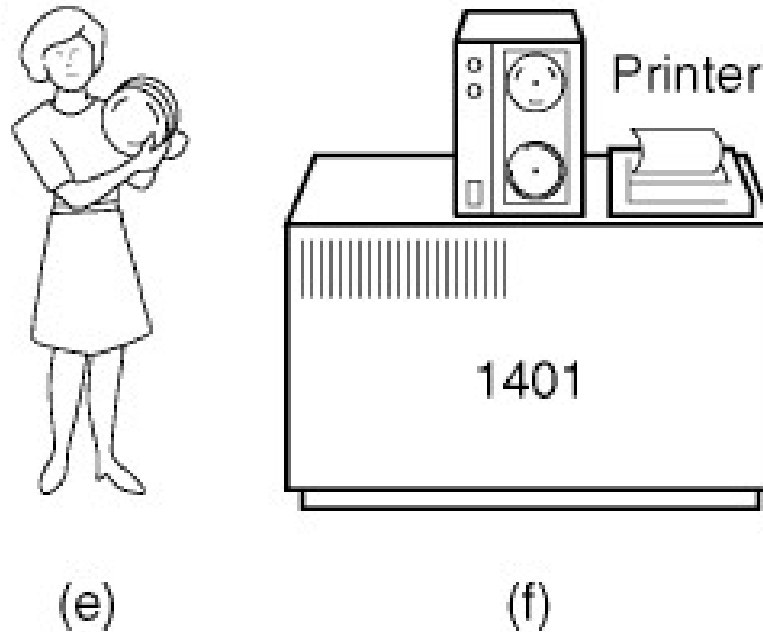


Figure 1-2. An early batch system. (e) Operator carries output tape to 1401. (f) 1401 prints output.

# Early Batch System (4)

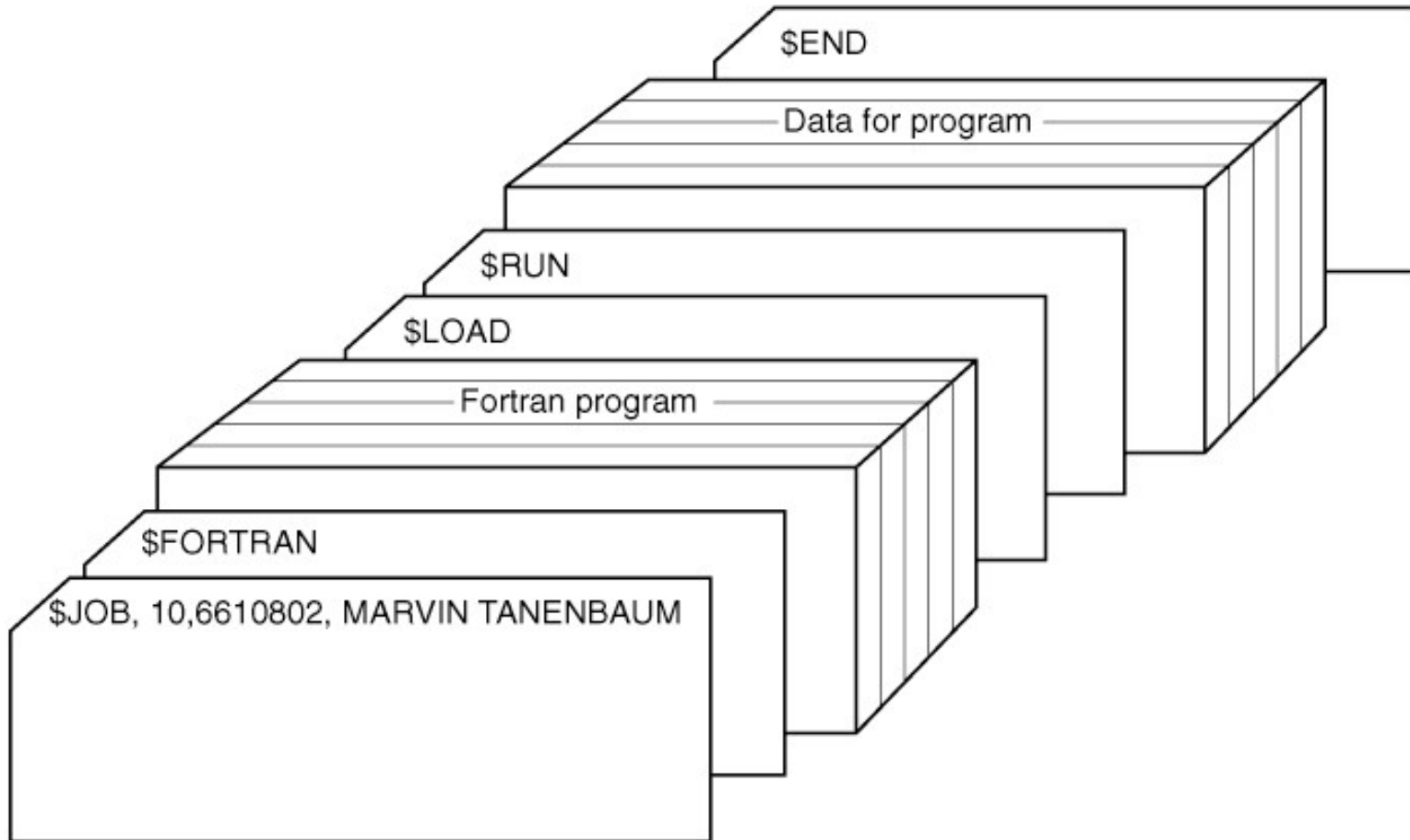


Figure 1-3. Structure of a typical FMS job.



# Multiprogramming

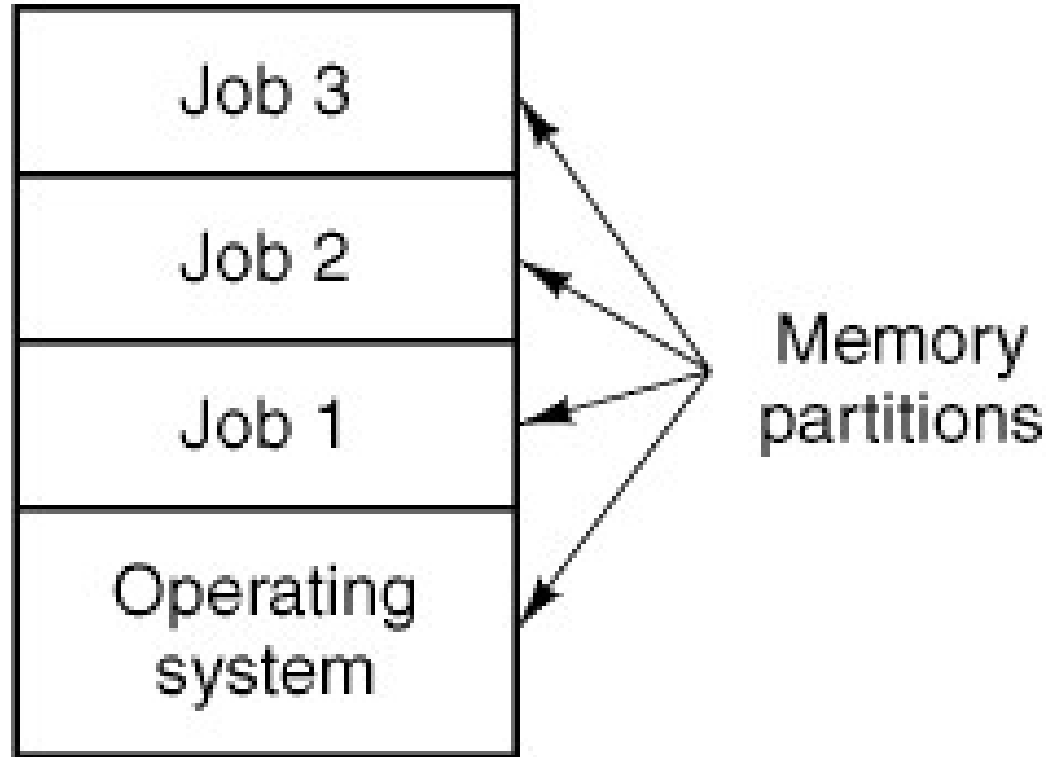


Figure 1-4. A multiprogramming system with three jobs in memory.

# Processes

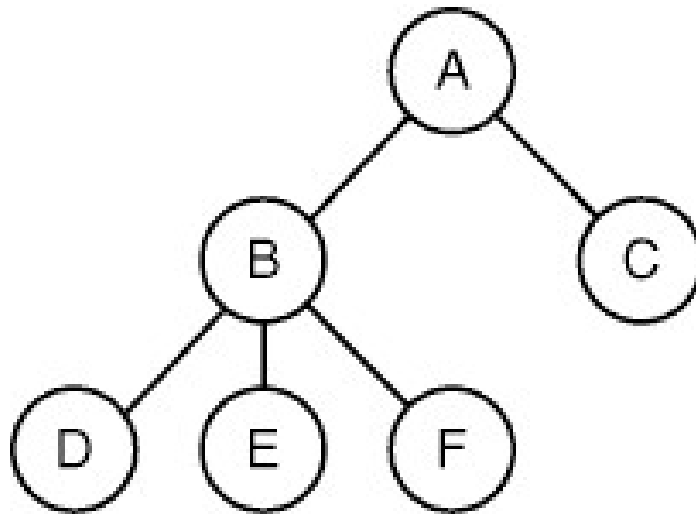


Figure 1-5. A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F.

# File Systems (1)

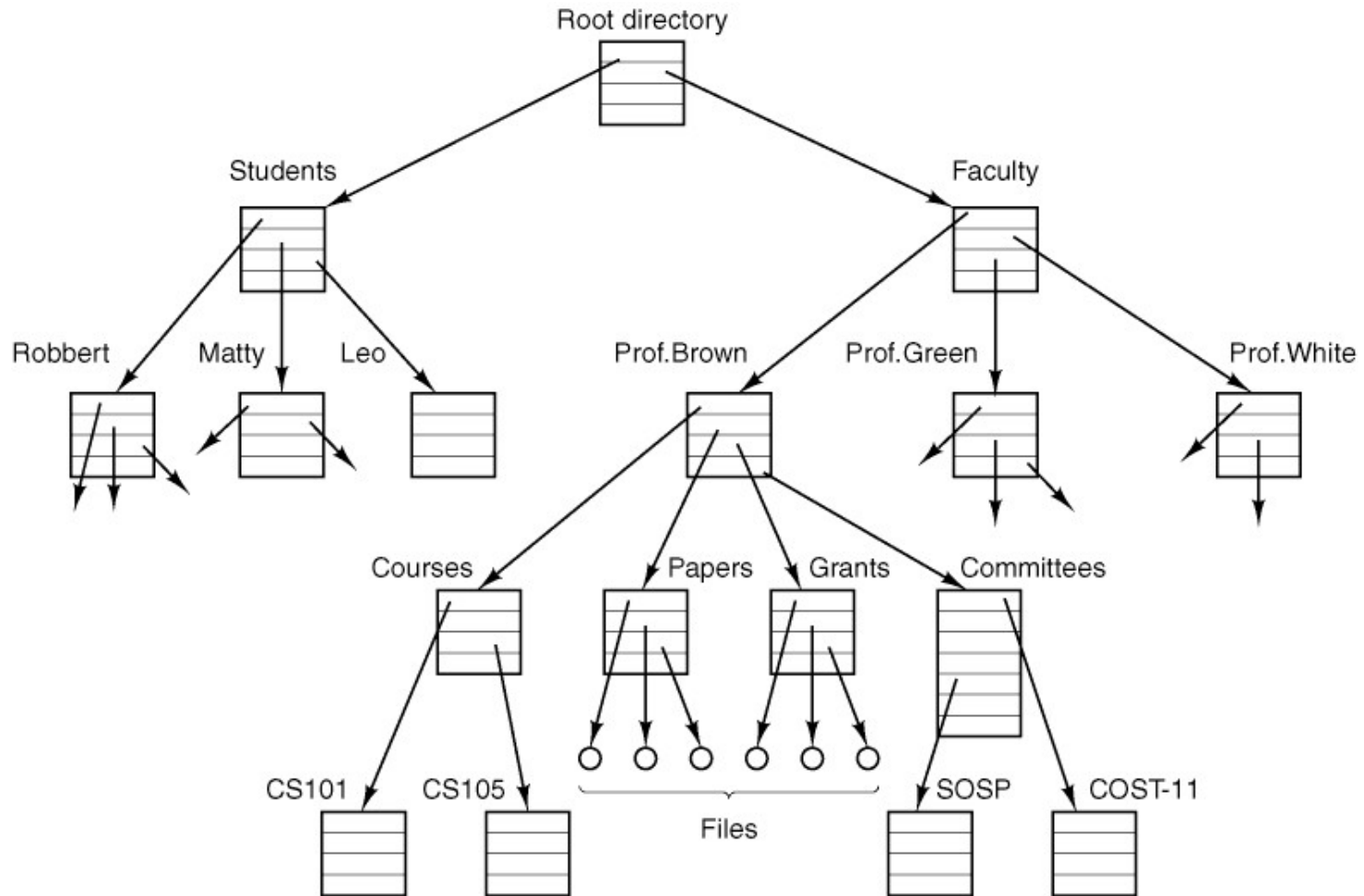


Figure 1-6. A file system for a university department.

# File Systems (2)

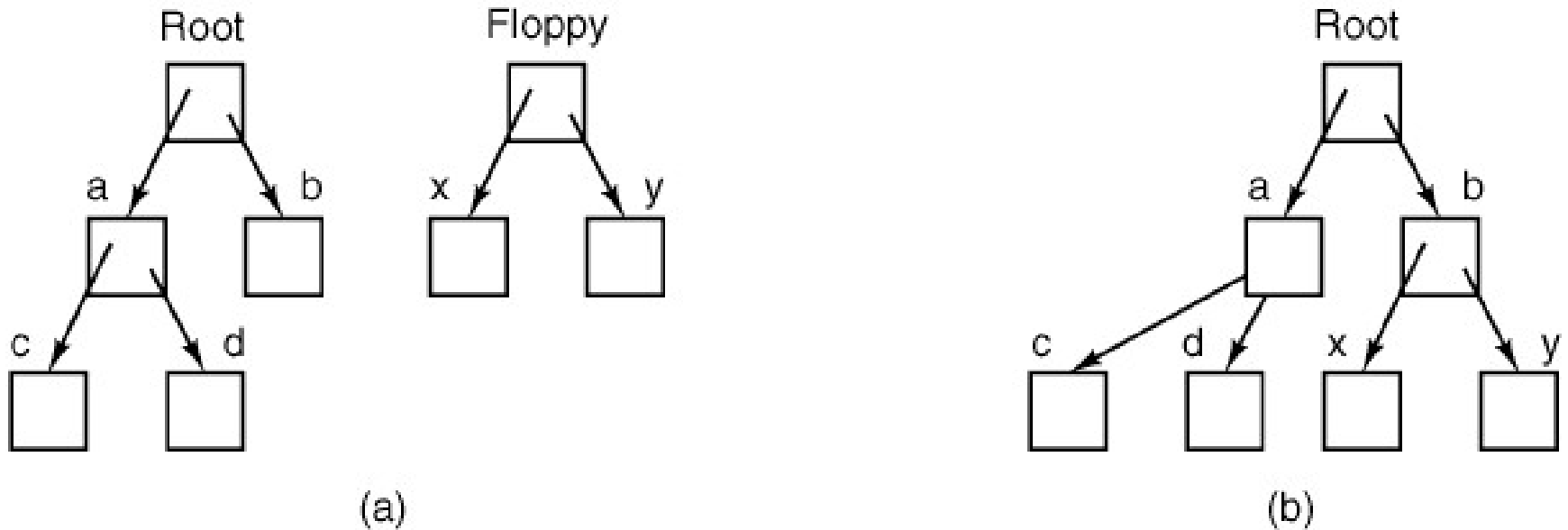


Figure 1-7. (a) Before mounting, the files on drive 0 are not accessible. (b) After mounting, they are part of the file hierarchy.

# File Systems (3)

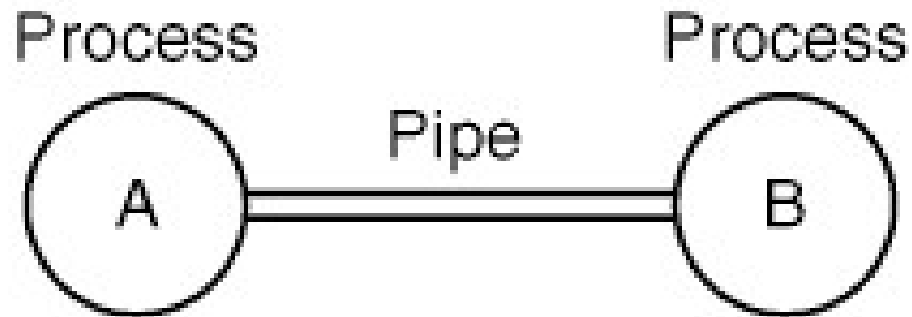


Figure 1-8. Two processes connected by a pipe.

# System Calls (1)

## Process Management

<code>pid = fork()</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &amp;statloc, opts)</code>	Wait for a child to terminate
<code>s = wait(&amp;status)</code>	Old version of <code>waitpid</code>
<code>s = execve(name, argv, envp)</code>	Replace a process core image
<code>exit(status)</code>	Terminate process execution and return status
<code>size = brk(addr)</code>	Set the size of the data segment
<code>pid = getpid()</code>	Return the caller's process id
<code>pid = getpgrp()</code>	Return the id of the caller's process group
<code>pid = setsid()</code>	Create a new session and return its process group id
<code>l = ptrace(req, pid, addr, data)</code>	Used for debugging

Figure 1-9. The MINIX system calls. `fd` is a file descriptor; and `n` is a byte count.

# System Calls (2)

## Signals

<code>s = sigaction(sig, &amp;act, &amp;oldact)</code>	Define action to take on signals
<code>s = sigreturn(&amp;context)</code>	Return from a signal
<code>s = sigprocmask(how, &amp;set, &amp;old)</code>	Examine or change the signal mask
<code>s = sigpending(set)</code>	Get the set of blocked signals
<code>s = sigsuspend(sigmask)</code>	Replace the signal mask and suspend the process
<code>s = kill(pid, sig)</code>	Send a signal to a process
<code>residual = alarm(seconds)</code>	Set the alarm clock
<code>s = pause()</code>	Suspend the caller until the next signal

Figure 1-9. The MINIX system calls. `fd` is a file descriptor; and `n` is a byte count.

# System Calls (3)

## File Management

<code>fd = creat(name, mode)</code>	Obsolete way to create a new file
<code>fd = mknod(name, mode, addr)</code>	Create a regular, special, or directory i-node
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>pos = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &amp;buf)</code>	Get a file's status information
<code>s = fstat(fd, &amp;buf)</code>	Get a file's status information
<code>fd = dup(fd)</code>	Allocate a new file descriptor for an open file
<code>s = pipe(&amp;fd[0])</code>	Create a pipe
<code>s = ioctl(fd, request, argp)</code>	Perform special operations on a file
<code>s = access(name, amode)</code>	Check a file's accessibility
<code>s = rename(old, new)</code>	Give a file a new name
<code>s = fcntl(fd, cmd, ...)</code>	File locking and other operations

Figure 1-9. The MINIX system calls. `fd` is a file descriptor; and `n` is a byte count.



# System Calls (4)

## Dir. & File System Mgmt.

<code>s = mkdir(name, mode)</code>	Create a new directory
<code>s = rmdir(name)</code>	Remove an empty directory
<code>s = link(name1, name2)</code>	Create a new entry, name2, pointing to name1
<code>s = unlink(name)</code>	Remove a directory entry
<code>s = mount(special, name, flag)</code>	Mount a file system
<code>s = umount(special)</code>	Unmount a file system
<code>s = sync()</code>	Flush all cached blocks to the disk
<code>s = chdir(dirname)</code>	Change the working directory
<code>s = chroot(dirname)</code>	Change the root directory

Figure 1-9. The MINIX system calls. fd is a file descriptor; and n is a byte count.

# System Calls (5)

## Protection

<code>s = chmod(name, mode)</code>	Change a file's protection bits
<code>uid = getuid()</code>	Get the caller's uid
<code>gid = getgid()</code>	Get the caller's gid
<code>s = setuid(uid)</code>	Set the caller's uid
<code>s = setgid(gid)</code>	Set the caller's gid
<code>s = chown(name, owner, group)</code>	Change a file's owner and group
<code>oldmask = umask(complmode)</code>	Change the mode mask

Figure 1-9. The MINIX system calls. `fd` is a file descriptor; and `n` is a byte count.

# System Calls (6)

## Time Management

`seconds = time(&seconds)`

`s = stime(tp)`

`s = utime(file, timep)`

`s = times(buffer)`

Get the elapsed time since Jan. 1, 1970

Set the elapsed time since Jan. 1, 1970

Set a file's "last access" time

Get the user and system times used so far

Figure 1-9. The MINIX system calls. `fd` is a file descriptor; and `n` is a byte count.

# The fork Call in the Shell

```
#define TRUE 1

while (TRUE) {                               /* repeat forever */
    type_prompt( );                          /* display prompt on the screen */
    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.

# Processes

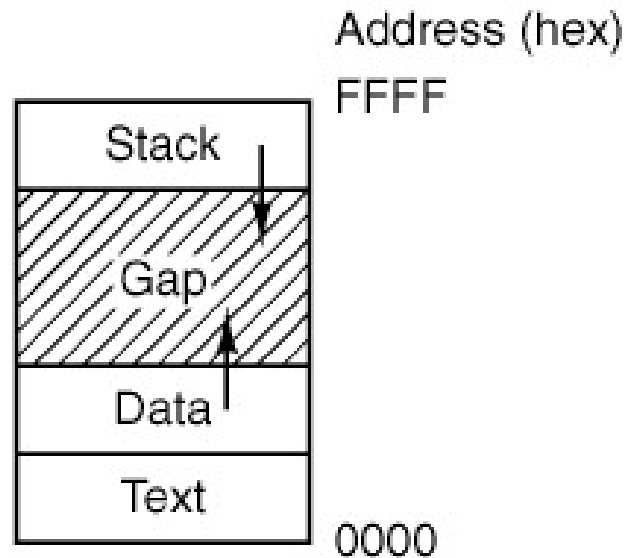


Figure 1-11. Processes have three segments: text, data, and stack. In this example, all three are in one address space, but separate instruction and data space is also supported.

# System Calls for File Management (1)

```
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.

# System Calls for File Management (2)

```
#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 { ...
```

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

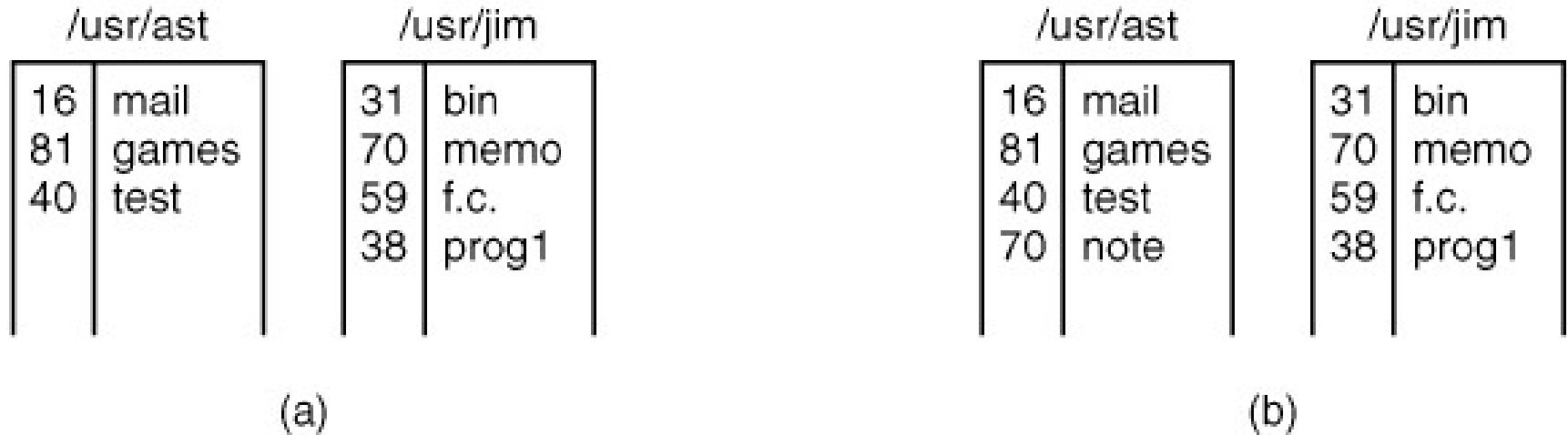
# System Calls for File Management (3)

```
...
/* 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.



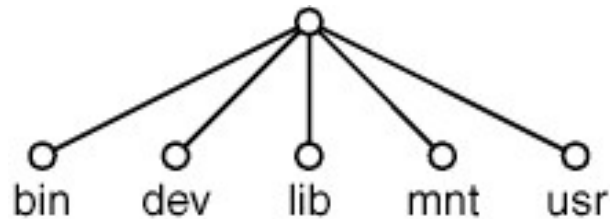
# System Calls for Directory Management (1)



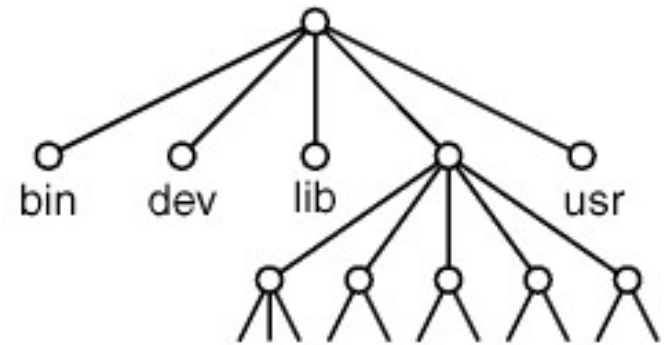
*link("/usr/jim/memo", "/usr/ast/note");*

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

# System Calls for Directory Management (2)



(a)



(b)

*mount("/dev/cdrom0", "/mnt", 0);*

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

# Operating System Structure

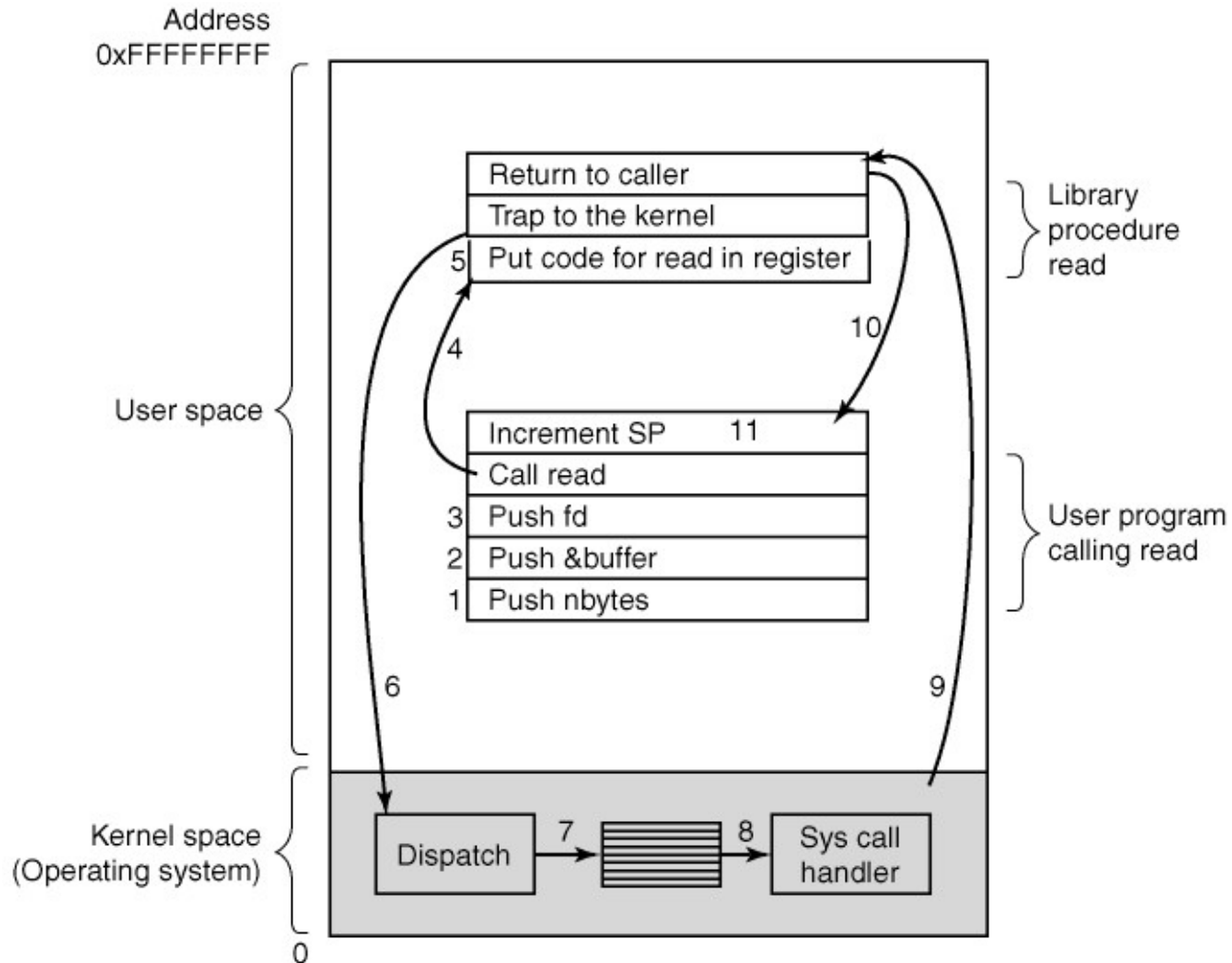


Figure 1-16. The 11 steps in making the system call `read(fd, buffer, nbytes)`.

# Basic Structure for Operating System

1. A main program that invokes the requested service procedure
2. A set of service procedures that carry out the system calls
3. A set of utility procedures that help the service procedures

# Layered Systems (1)

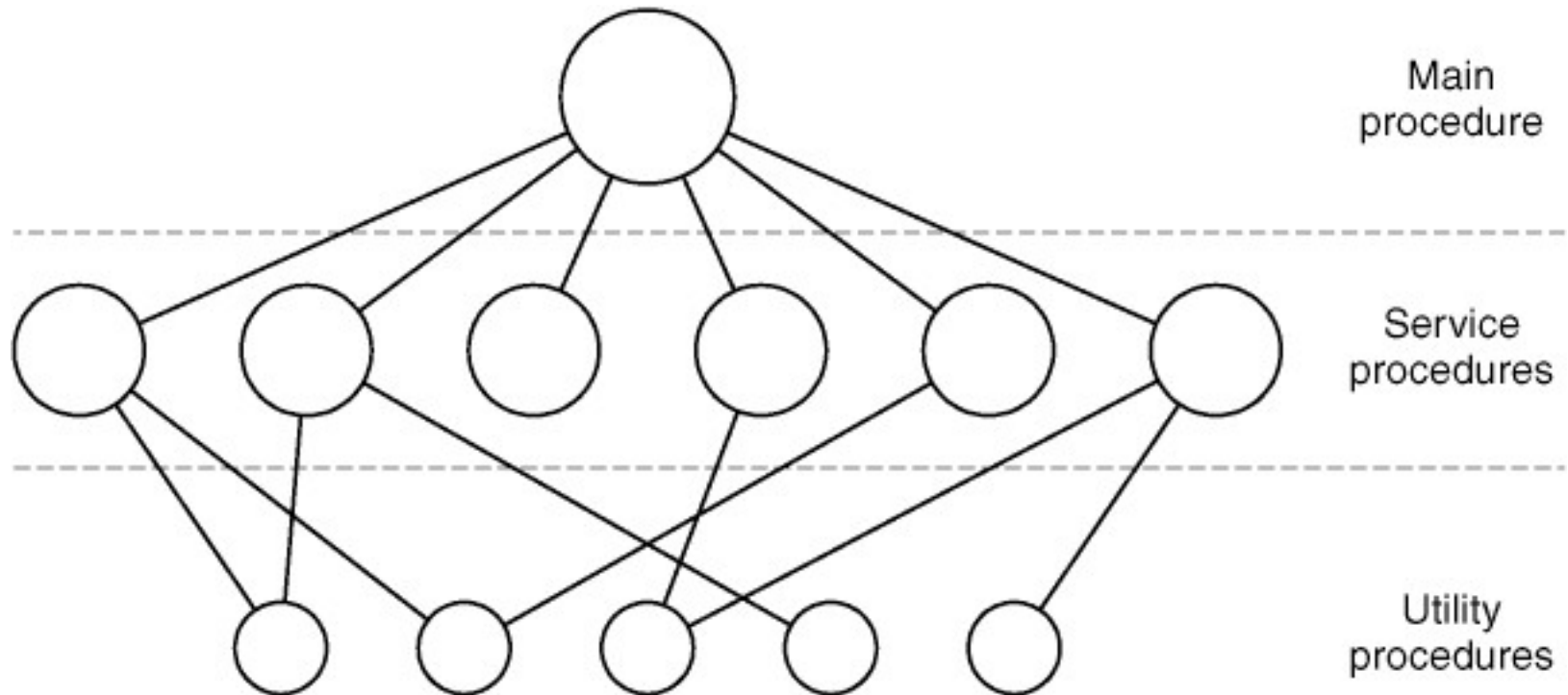


Figure 1-17. A simple structuring model for a monolithic system.

# Layered Systems (2)

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.

# Virtual Machines

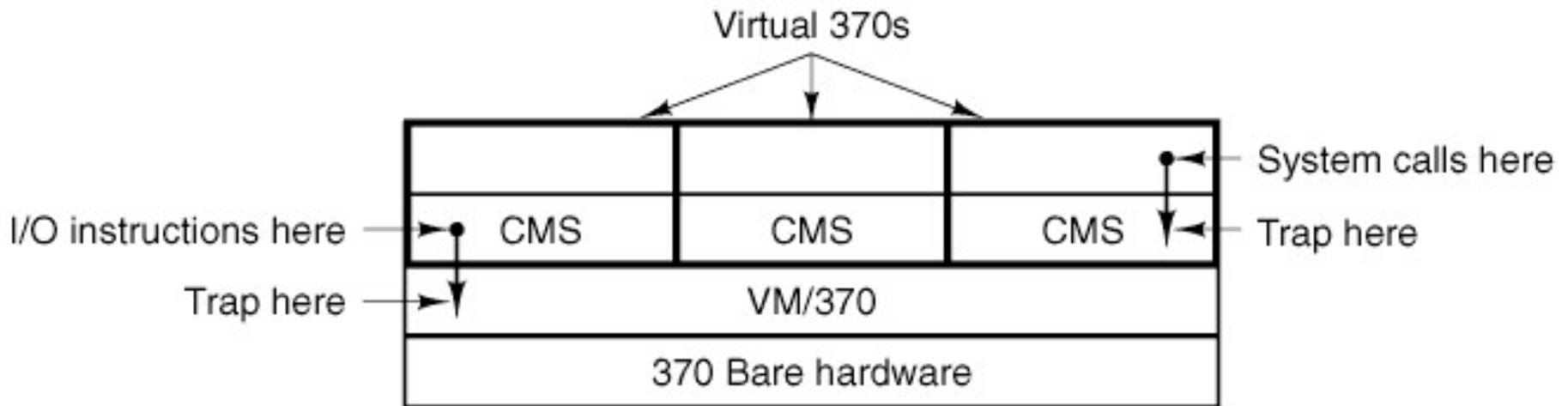


Figure 1-19. The structure of VM/370 with CMS.

# Client-Server Model (1)

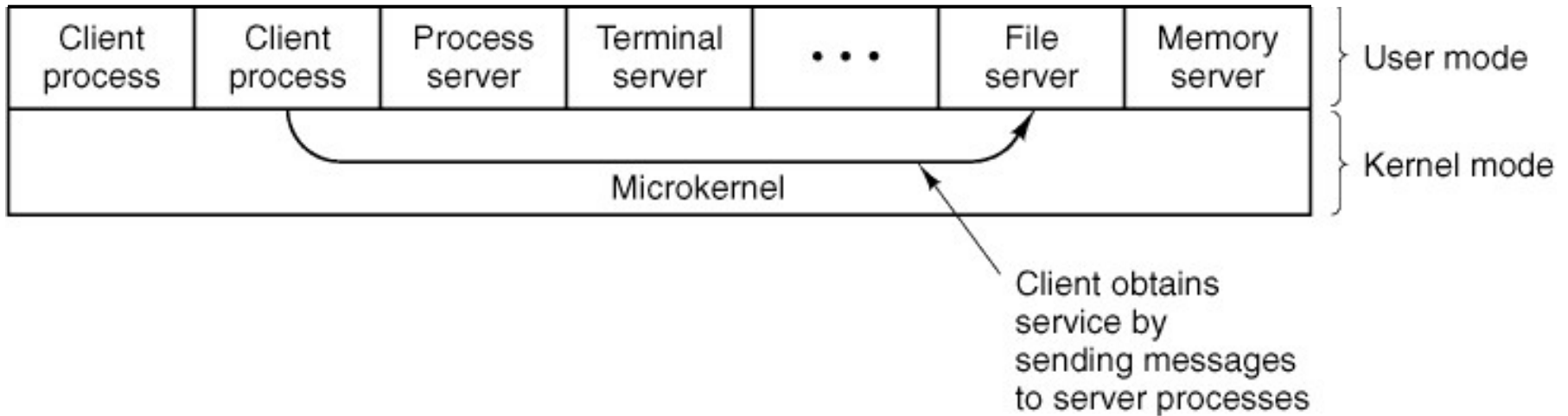


Figure 1-20. The client-server model.



# Client-Server Model (2)

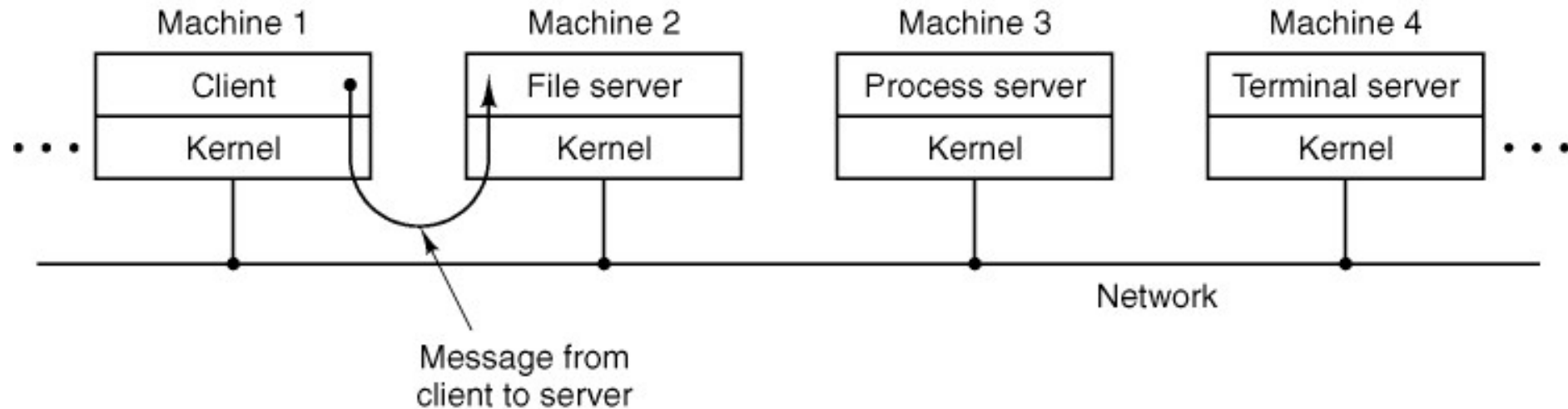


Figure 1-21. The client-server model in a distributed system.