Chapter 1
Introduction

- What is an operating system
- History of operating systems
- The operating system zoo
- Computer hardware review
- Operating system concepts
- System calls
- Operating system structure

What Is An Operating System

A modern computer consists of:

- One or more processors
- Main memory
- Disks
- Printers
- Various input/output devices

Managing all these components requires a layer of software – the **operating system**

What Is An Operating System

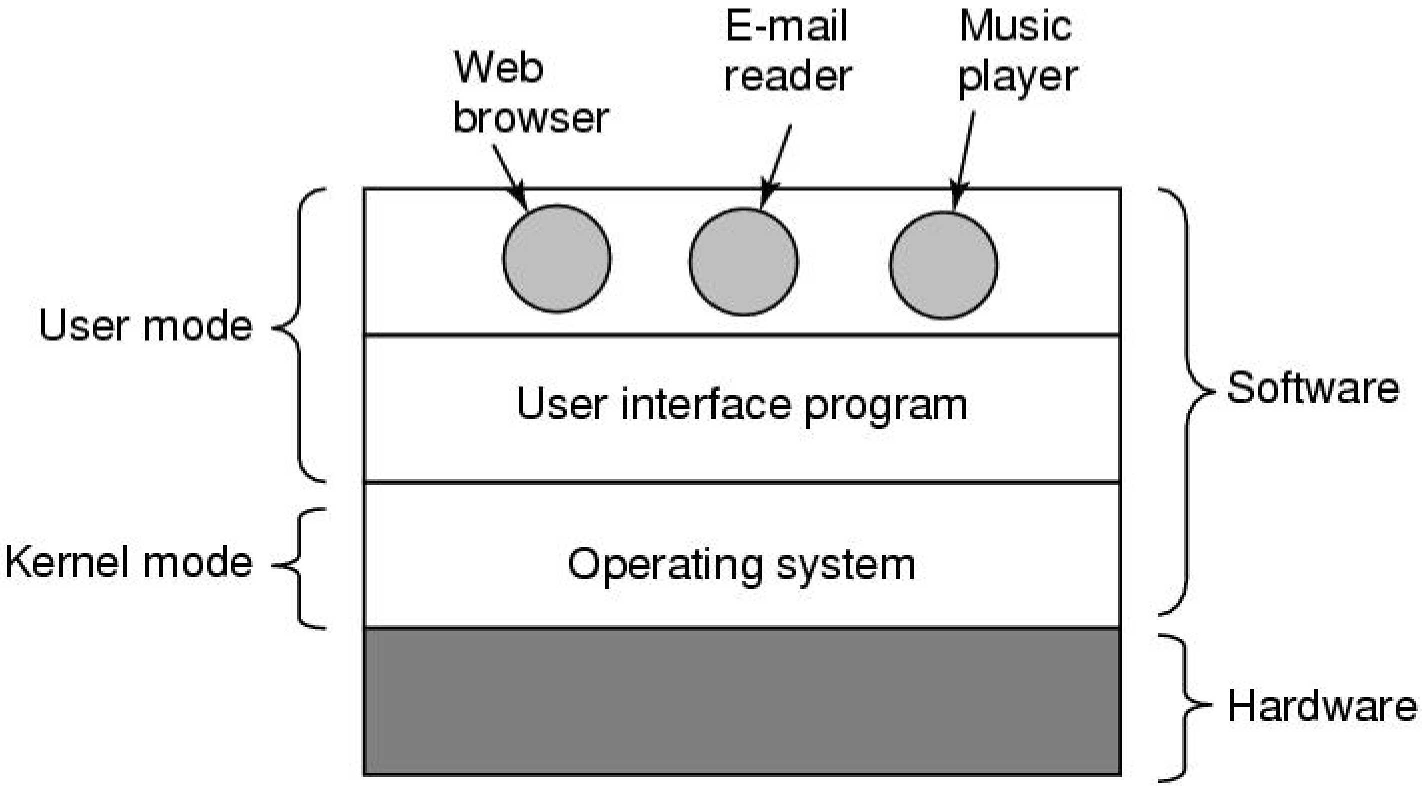
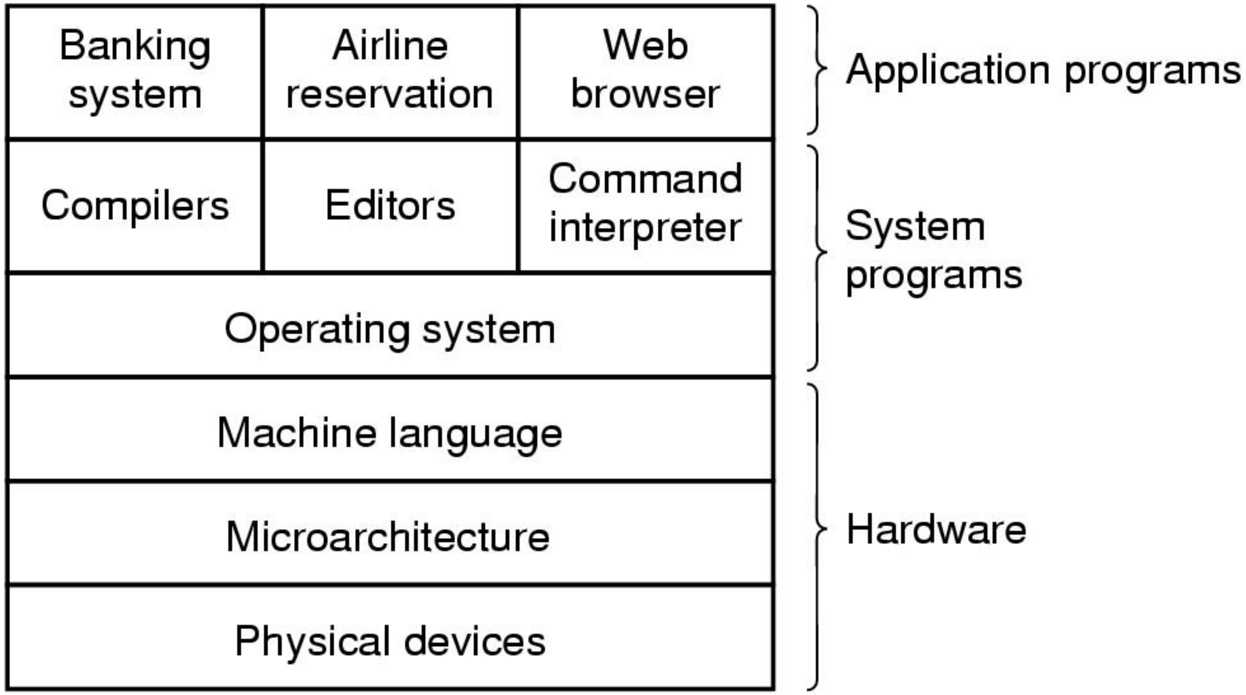


Figure 1-1. Where the operating system fits in.

What Is An Operating System



What is an Operating System

It is an extended machine

- Hides the messy details which must be performed
- Presents user with a virtual machine, easier to use

It is a resource manager

- Each program gets time with the resource
- Each program gets space on the resource

The Operating System as an Extended Machine

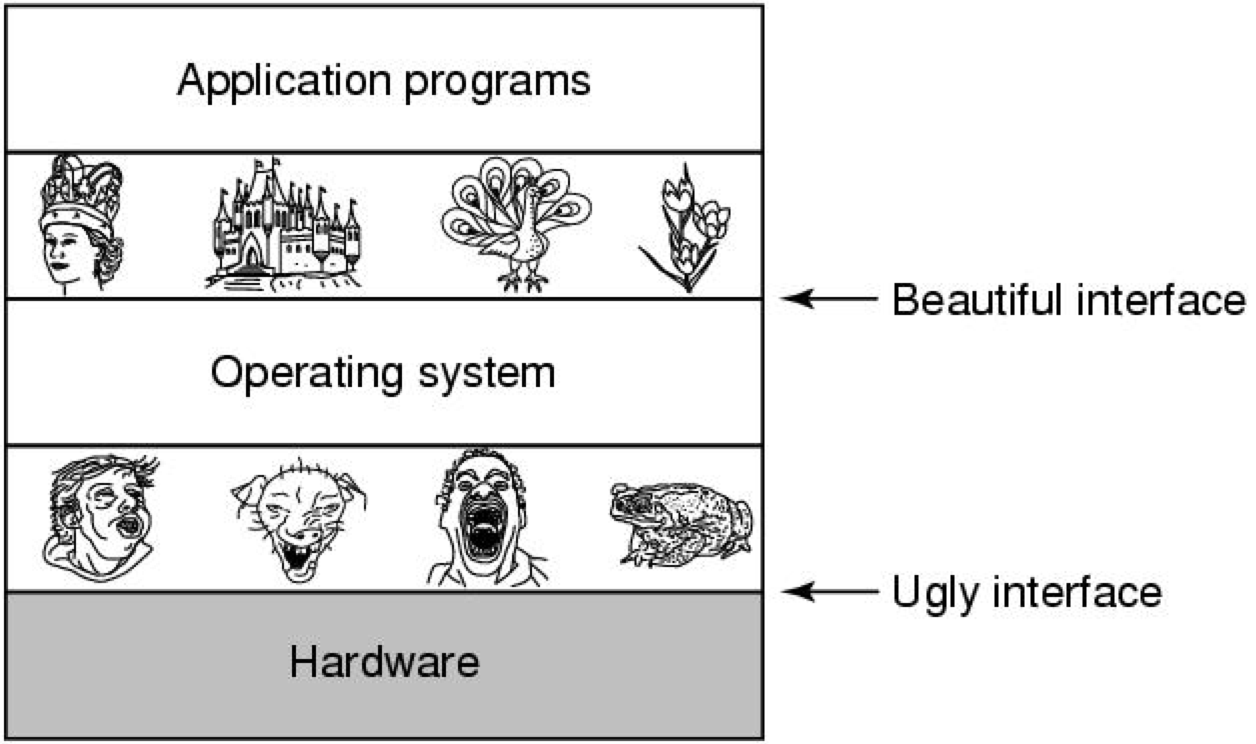


Figure 1-2. Operating systems turn ugly hardware into beautiful abstractions.

The Operating System as a Resource Manager

- Allow multiple programs to run at the same time
- Manage and protect memory, I/O devices, and other resources
- Includes multiplexing (sharing) resources in two different ways:
  - In time
  - In space

History of Operating Systems

Generations:

- (1945–55) Vacuum Tubes
- (1955–65) Transistors and Batch Systems
- (1965–1980) ICs and Multiprogramming
- (1980–Present) Personal Computers

Transistors and Batch Systems (1)

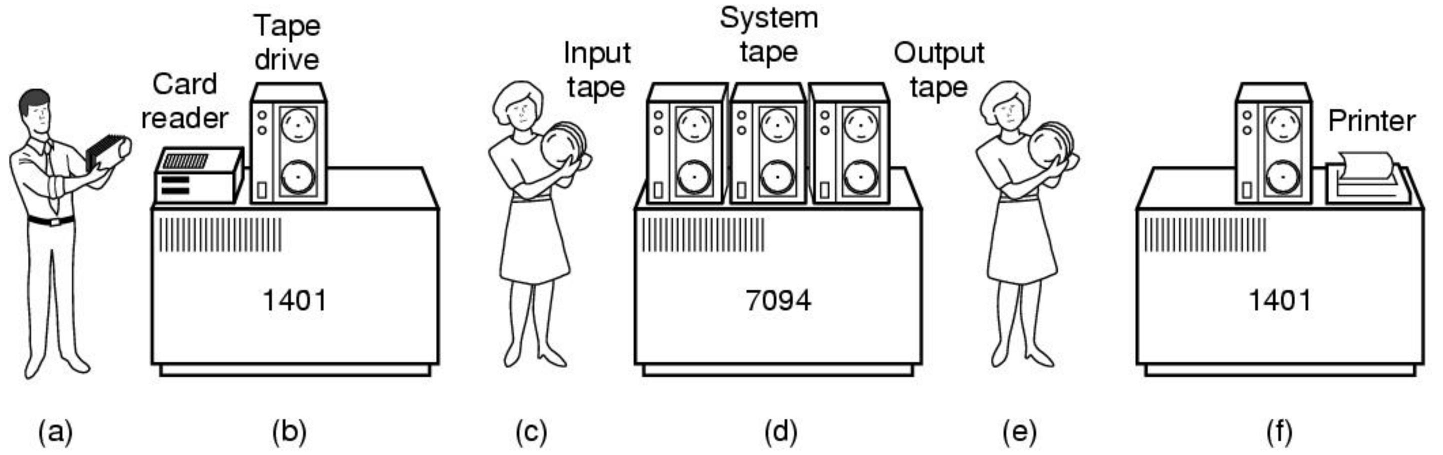


Figure 1-3. An early batch system.
(a) Programmers bring cards to 1401.
(b)1401 reads batch of jobs onto tape.

Transistors and Batch Systems (2)

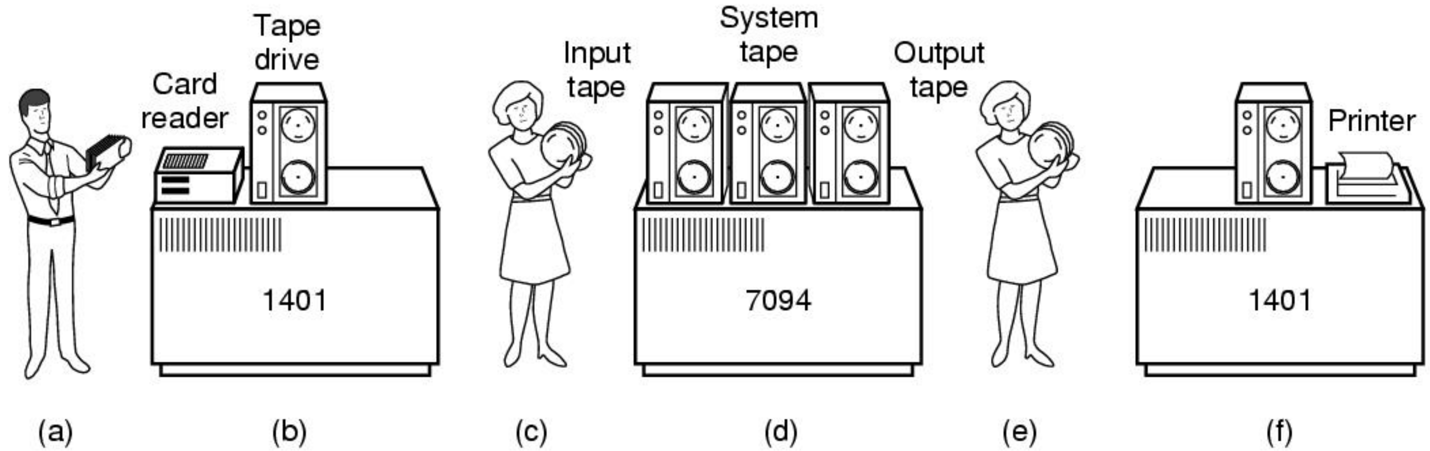


Figure 1-3. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

Transistors and Batch Systems (4)

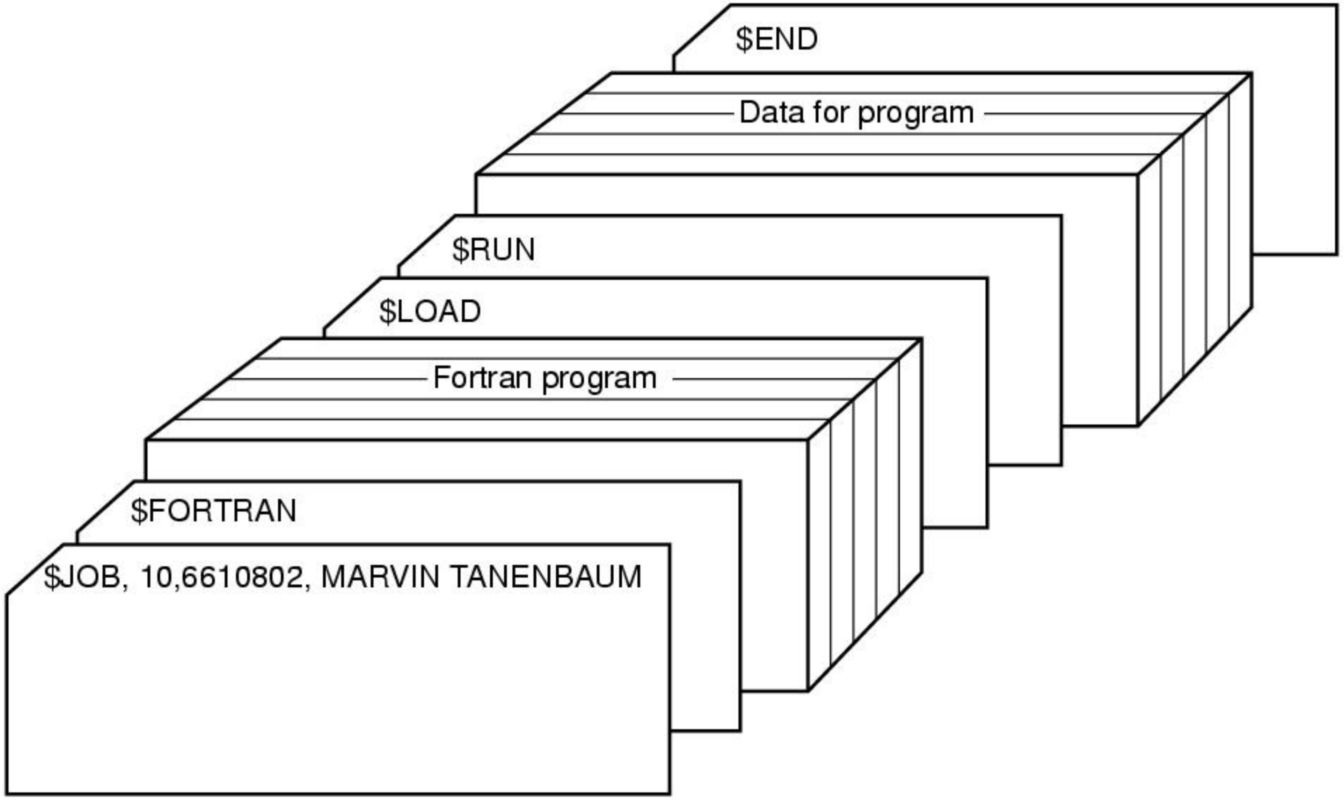


Figure 1-4. Structure of a typical FMS job.

ICs and Multiprogramming

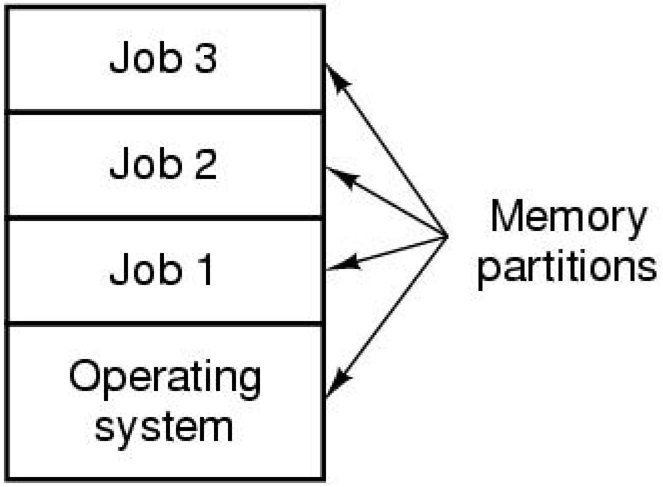


Figure 1-5. A multiprogramming system with three jobs in memory.

Computer Hardware Review

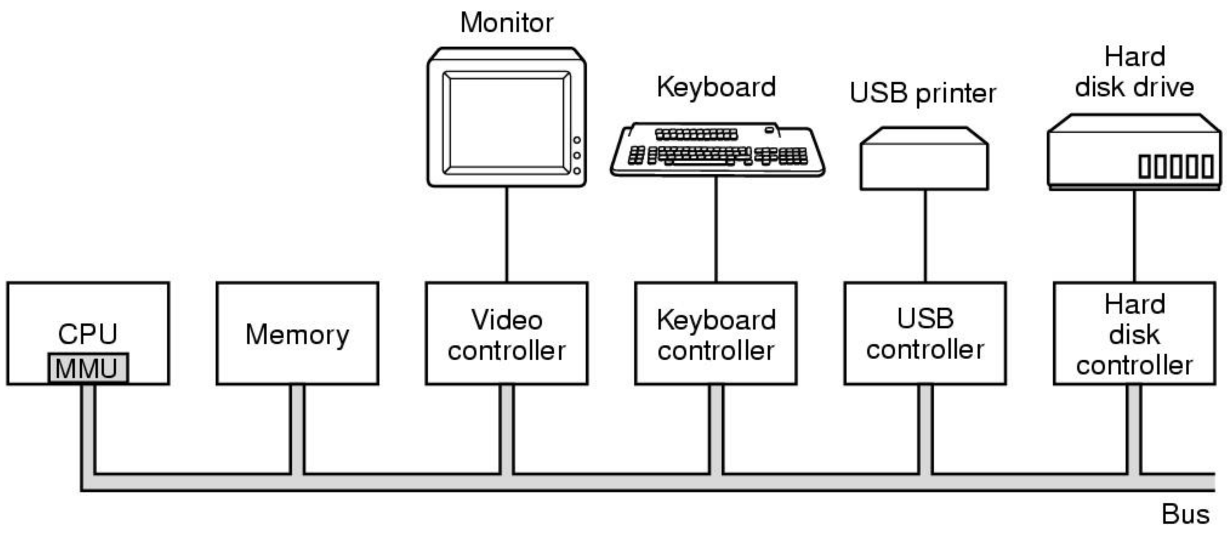


Figure 1-6. Some of the components of a simple personal computer.

CPU Pipelining

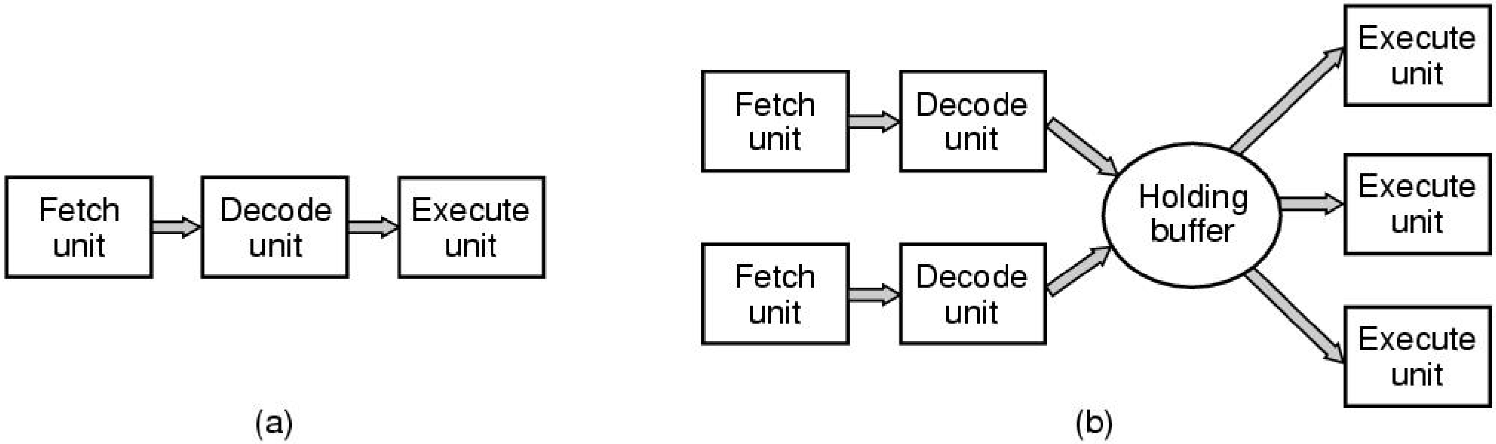


Figure 1-7. (a) A three-stage pipeline. (b) A superscalar CPU.

Multithreaded and Multicore Chips

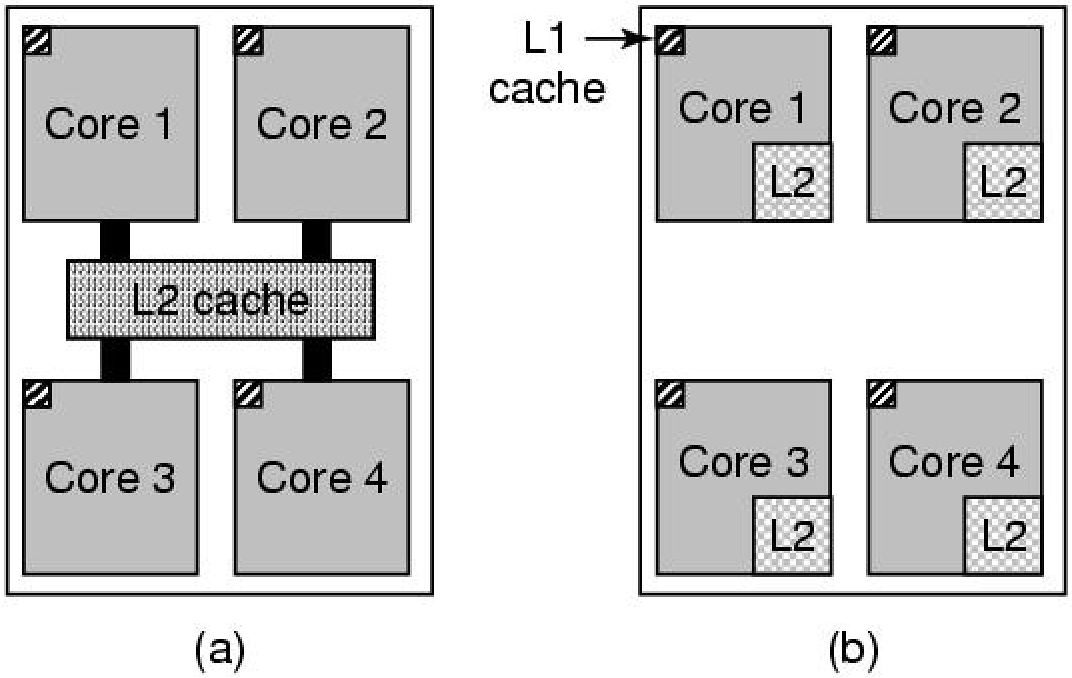


Figure 1-8. (a) A quad-core chip with a shared L2 cache.
(b) A quad-core chip with separate L2 caches.

Memory (1)

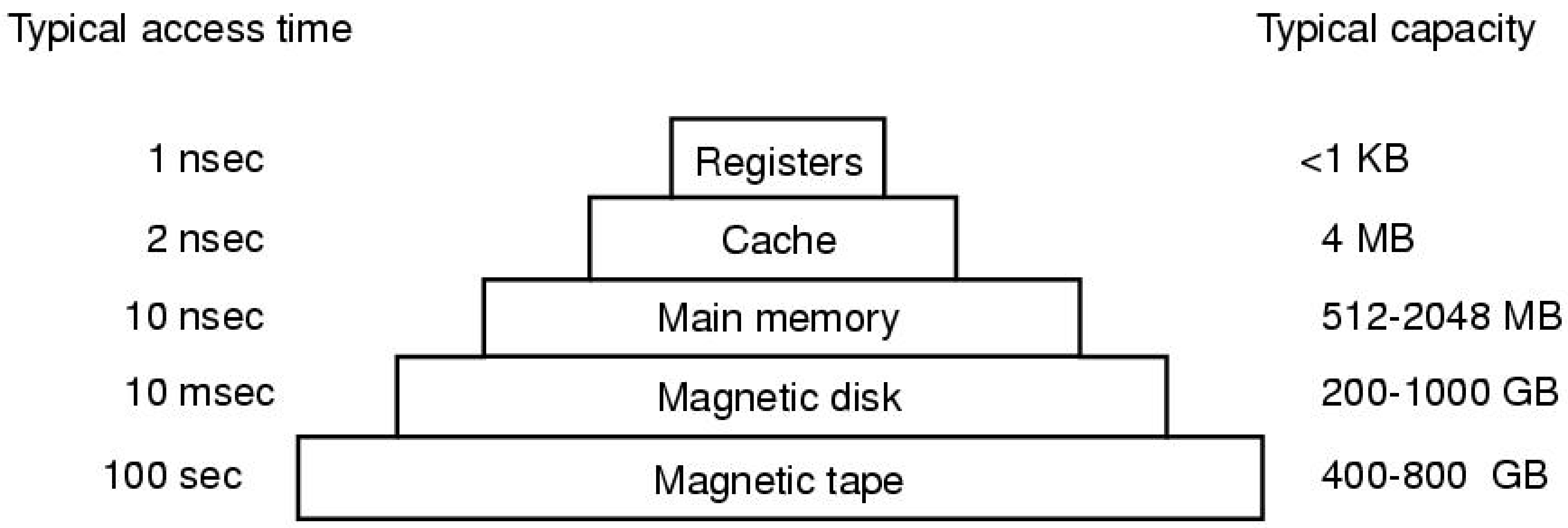


Figure 1-9. A typical memory hierarchy.
The numbers are very rough approximations.

Memory (2)

Questions when dealing with cache:

- When to put a new item into the cache.
- Which cache line to put the new item in.
- Which item to remove from the cache when a slot is needed.
- Where to put a newly evicted item in the larger memory.

Disks

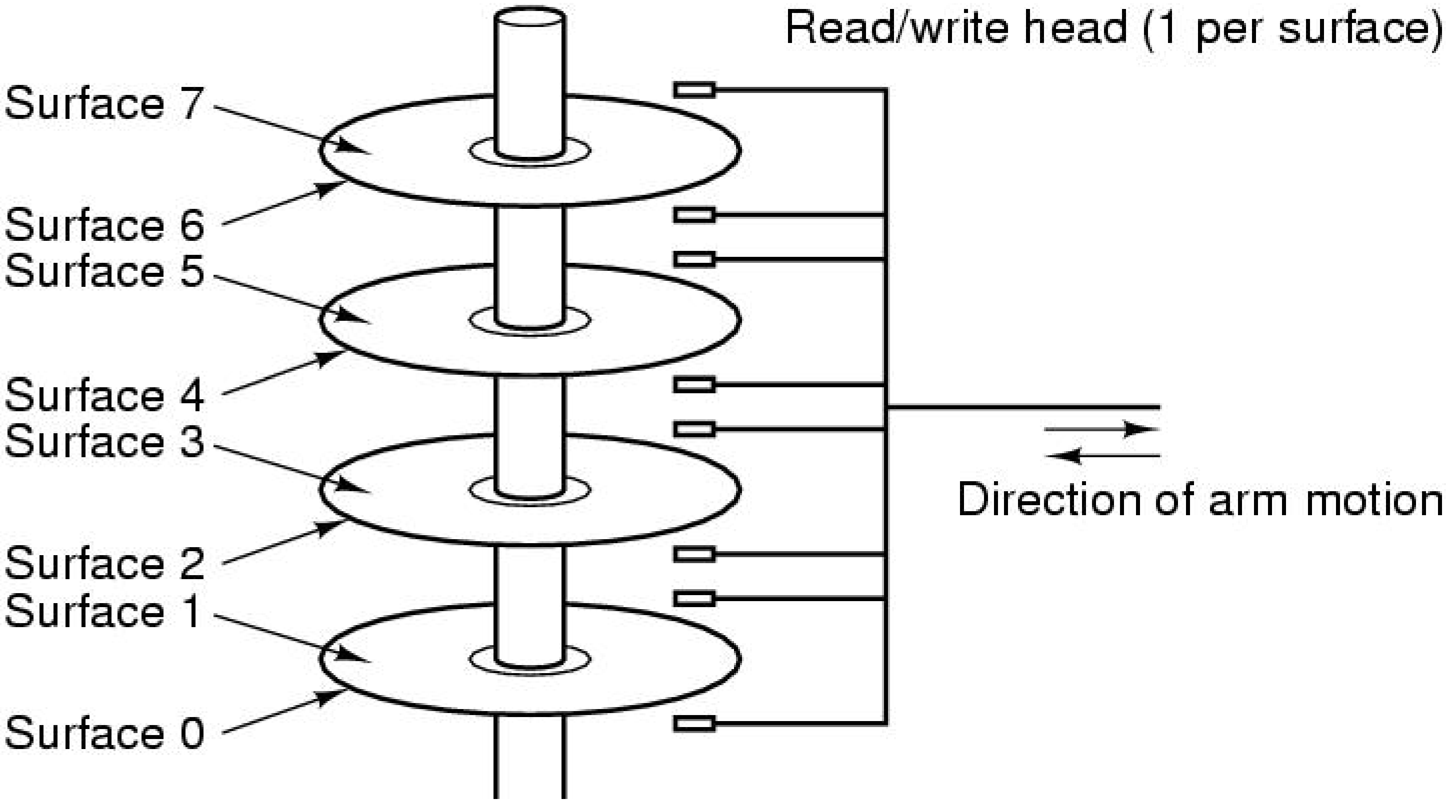


Figure 1-10. Structure of a disk drive.

I/O Devices

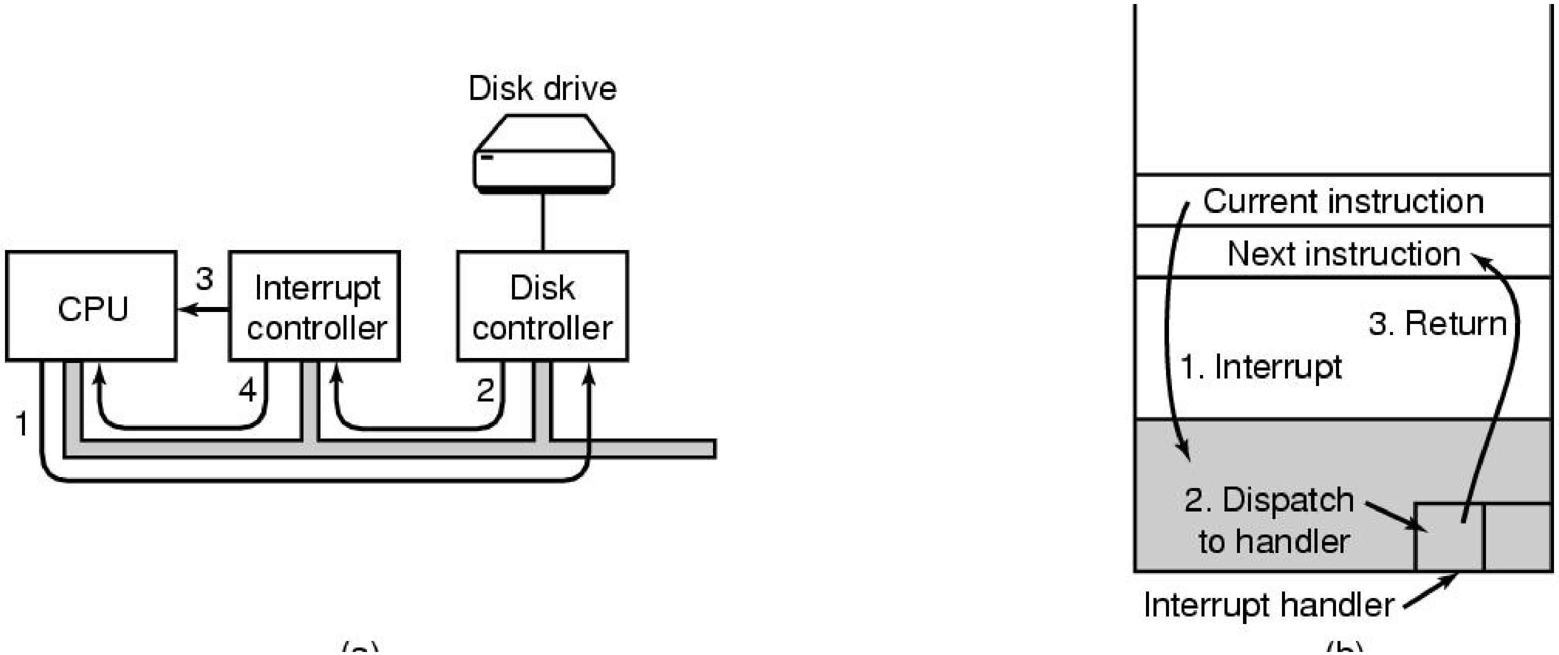


Figure 1-11. (a) The steps in starting an I/O device and getting an interrupt.

The Operating System Zoo

- Mainframe operating systems
- Server operating systems
- Multiprocessor operating systems
- Personal computer operating systems
- Handheld operating systems
- Embedded operating systems
- Sensor node operating systems
- Real-time operating systems
- Smart card operating systems

Operating System Concepts

- Processes
- Address spaces
- Files
- Input/Output
- Protection
- Shell
- Virtual memory

Processes

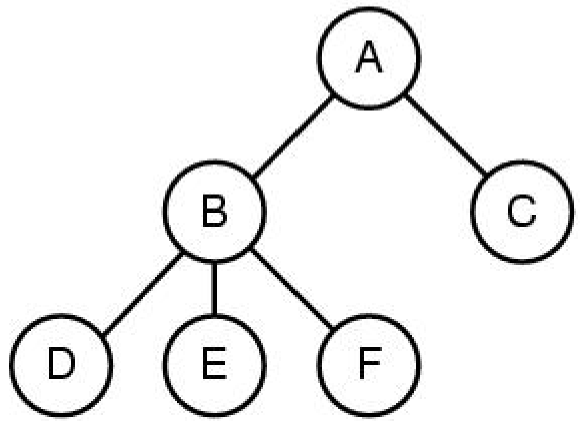
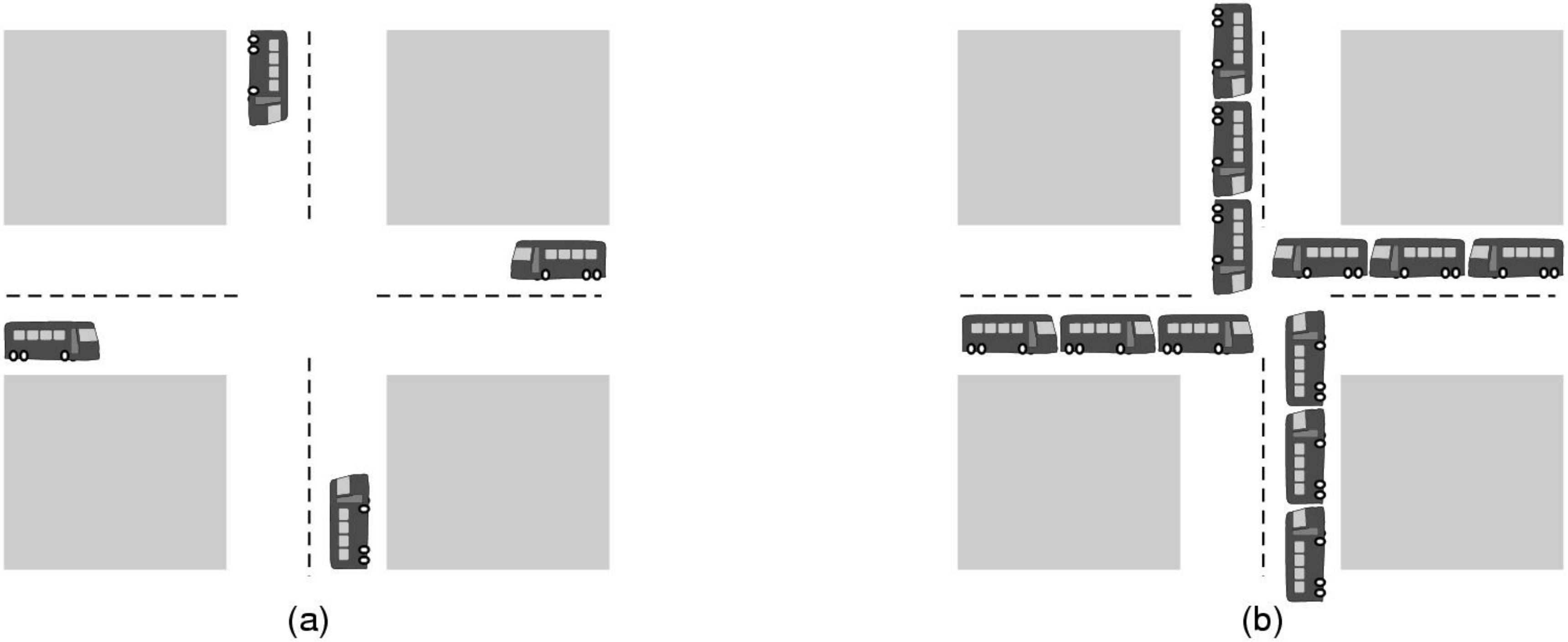


Figure 1-13. A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F.

Deadlock



(a) A potential deadlock. (b) an actual deadlock.

Files (1)

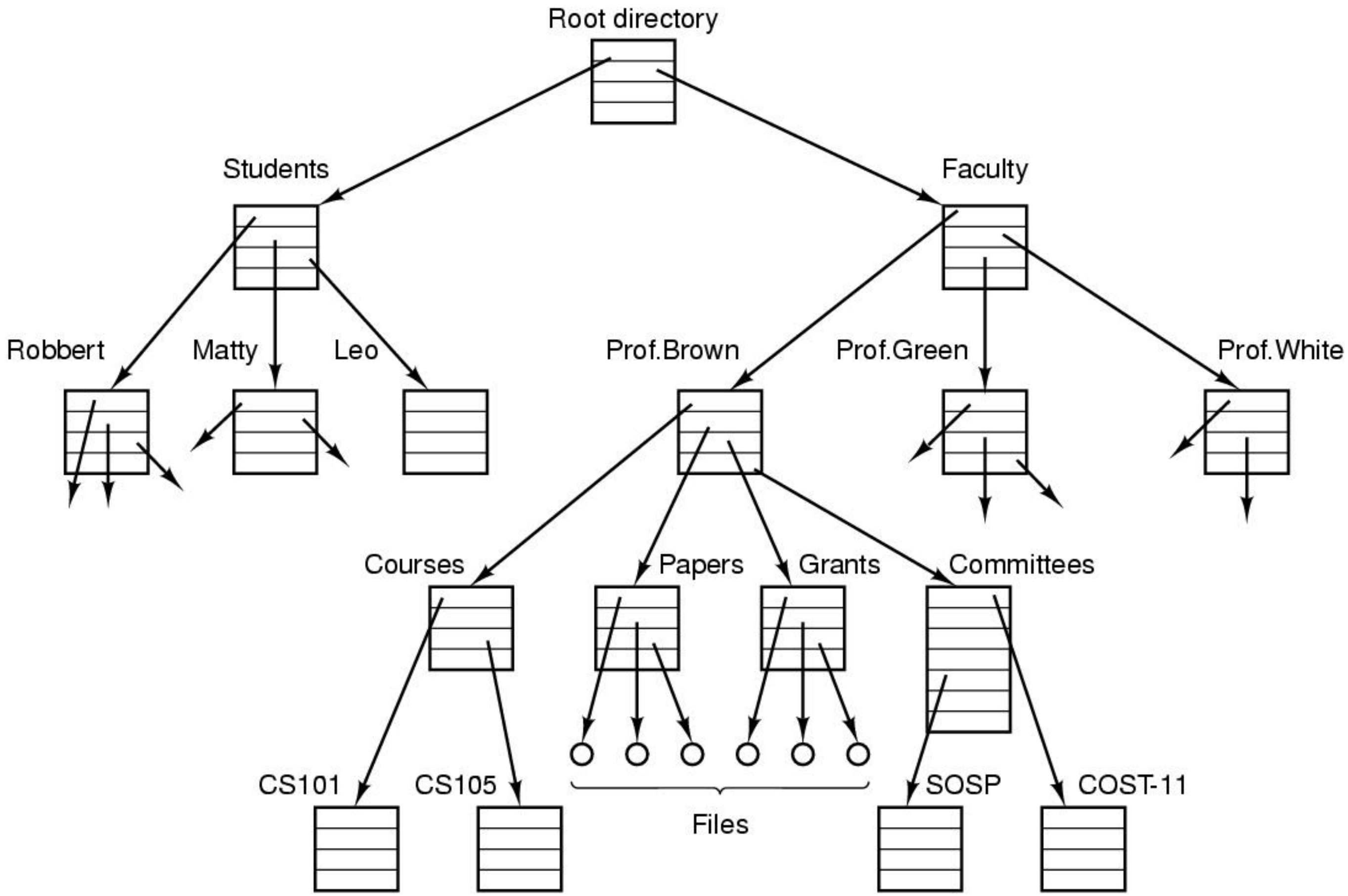


Figure 1-14. A file system for a university department.

Files (2)

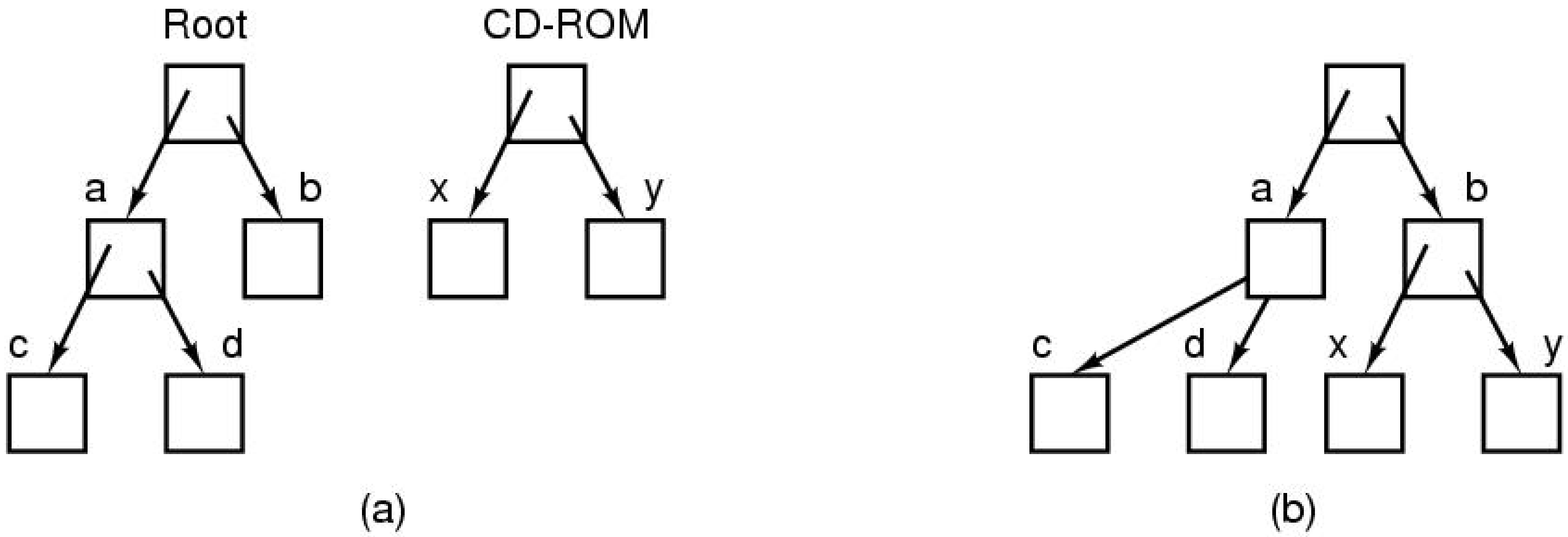


Figure 1-15. (a) Before mounting, the files on the CD-ROM are not accessible. (b) After mounting, they are part of the file hierarchy.

Files (3)

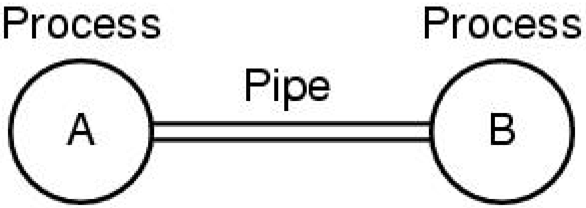


Figure 1-16. Two processes connected by a pipe.

System Calls

System calls: a set of “extended instructions" provided by O.S., providing the interface between a process and the O.S.

Example: Read a certain number of bytes from a file
count = read(fd, buffer, nbytes)

System Calls

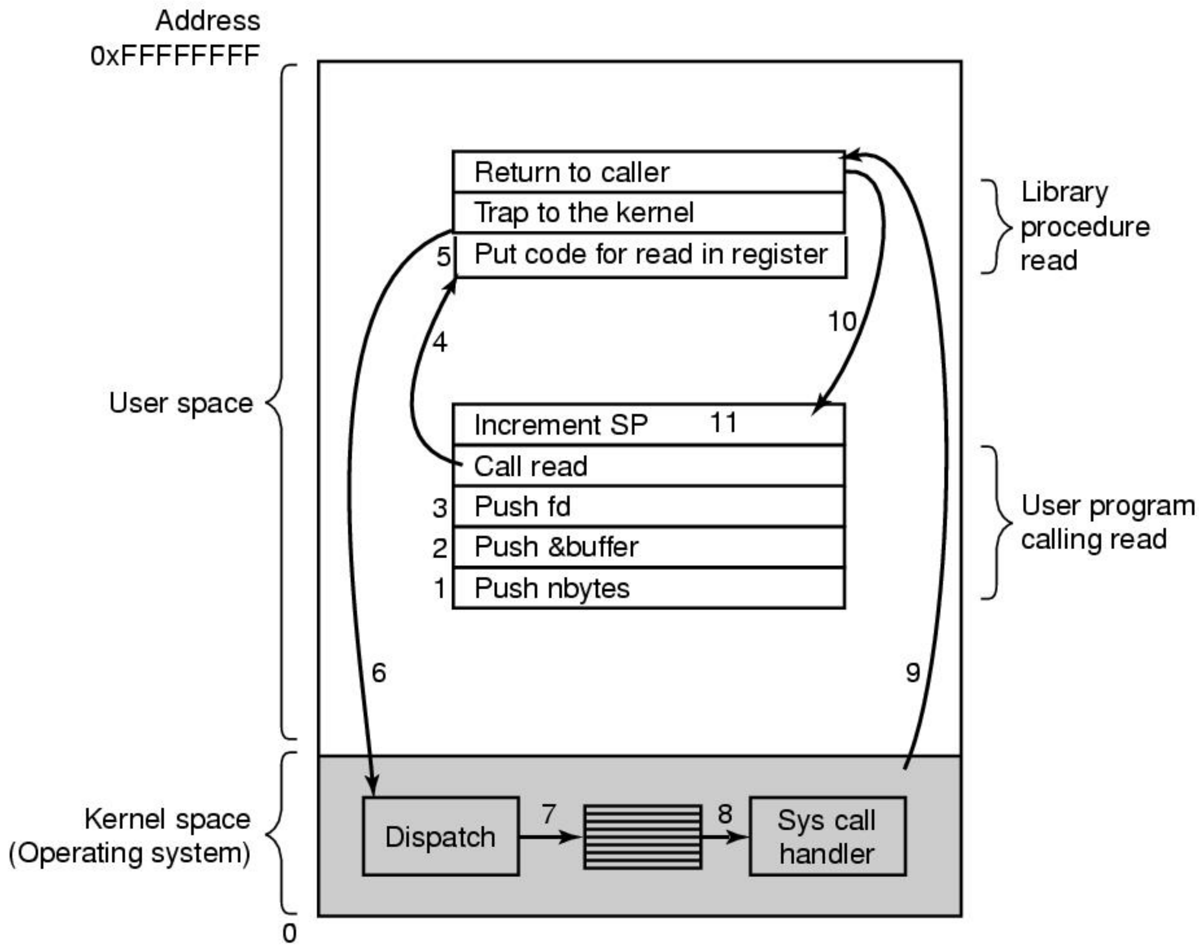


Figure 1-17. The 11 steps in making the system call read(fd, buffer, nbytes).

System Calls for Process Management

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Figure 1-18. Some of the major POSIX system calls.

System Calls for Process Management

ork(): the only way to create a new

fork()

The only way to create a new process in Unix.

Create a copy of the process executing it.

fork returns 0 in the child, and returns child's pid in the parent. Returns -1 for error.

exit(status)

A process terminates by calling exit system call.

status: 0-255, 0: normal, others: abnormal terminations.

waitpid(pid, status, opts)

pid: specific child, -1: first child.

status: child exit status.

opts: block or not.

System Calls for Process Management

execve
The only way a program is executed in Unix.
s = execve(file, argv, envp)

Example: A simplified shell.

Shell: Unix command interpreter.
Examples of shell commands:
date
date > file (output redirection)
sort < file (input redirection)
sort < file1 > file2 (input + output redirection)
cat file1 file2 | sort > file3 (pipe + output redirection)

A Simple 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 1-19. A stripped-down shell.

System Calls for File Management (1)

File management

Call	Description
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

Figure 1-18. Some of the major POSIX system calls.

System Calls for File Management

Read, write, create, open and close a file:
fd = creat(filename, mode)
fd = open(file, how)
close(fd)
Random access a file:
pos = lseek(fd, offset, whence)
Duplicate the file descriptor:
fd2 = dup(fd)
fd2 = dup2(fd, fd2)
Create a pipe:
pipe(&fd[0])
returns two file descriptors:
fd[0] : for reading
fd[1] : for writing
Example for using pipe system call

Example of Creating a Pipe

```
#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]);              /* pipe 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]);              /* pipe not needed any more */
        execl(process2, process2, 0);
  }
}
```

Fig. 1-14. A skeleton for setting up a two-process pipeline.

System Calls for File Management (2)

Call	Description
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

Figure 1-18. Some of the major POSIX system calls.

Miscellaneous System Calls

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure 1-18. Some of the major POSIX system calls.

Memory Layout

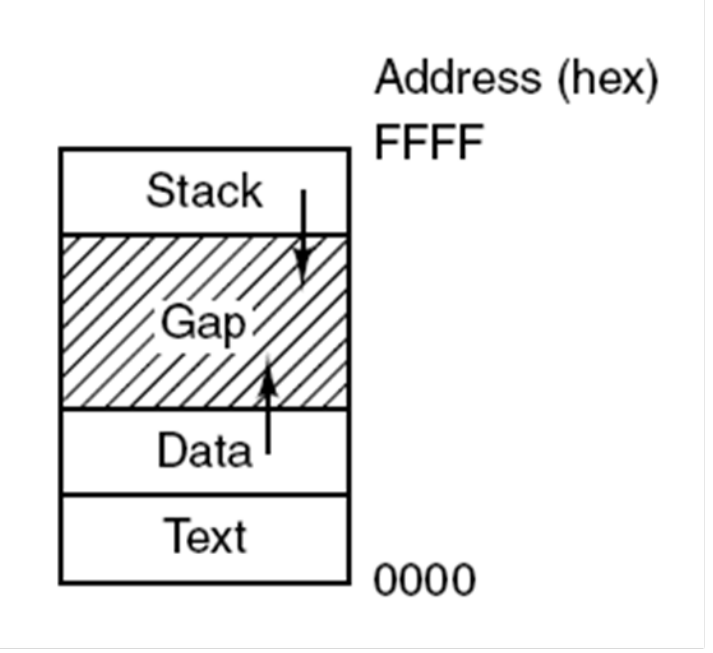


Figure 1-20. Processes have three segments: text, data, and stack.

Linking

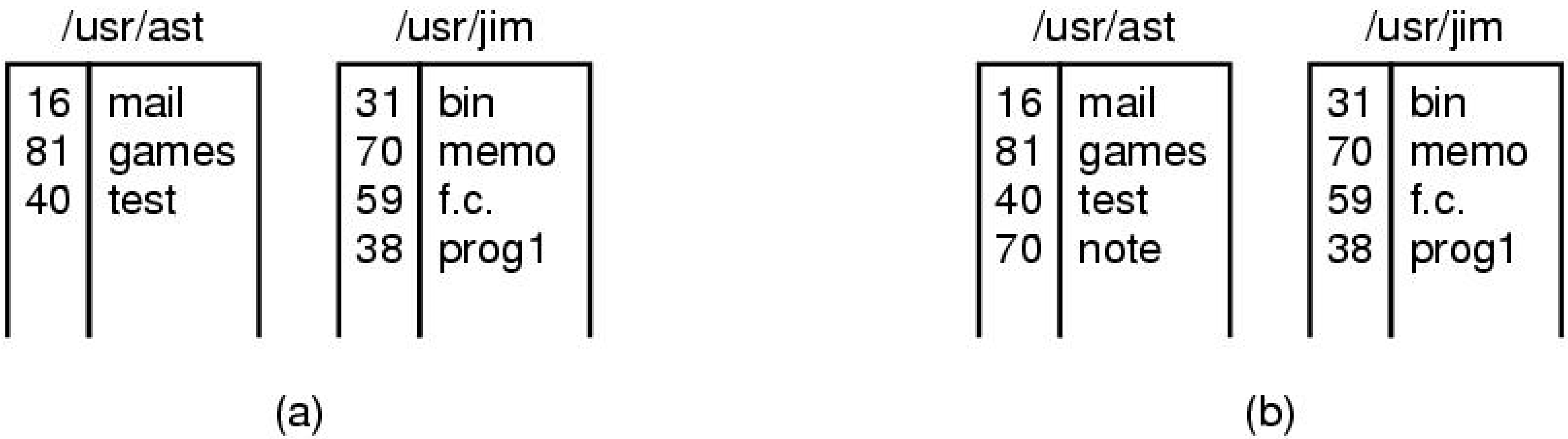


Figure 1-21. (a) Two directories before linking */usr/jim/memo* to ast's directory. (b) The same directories after linking.

Mounting

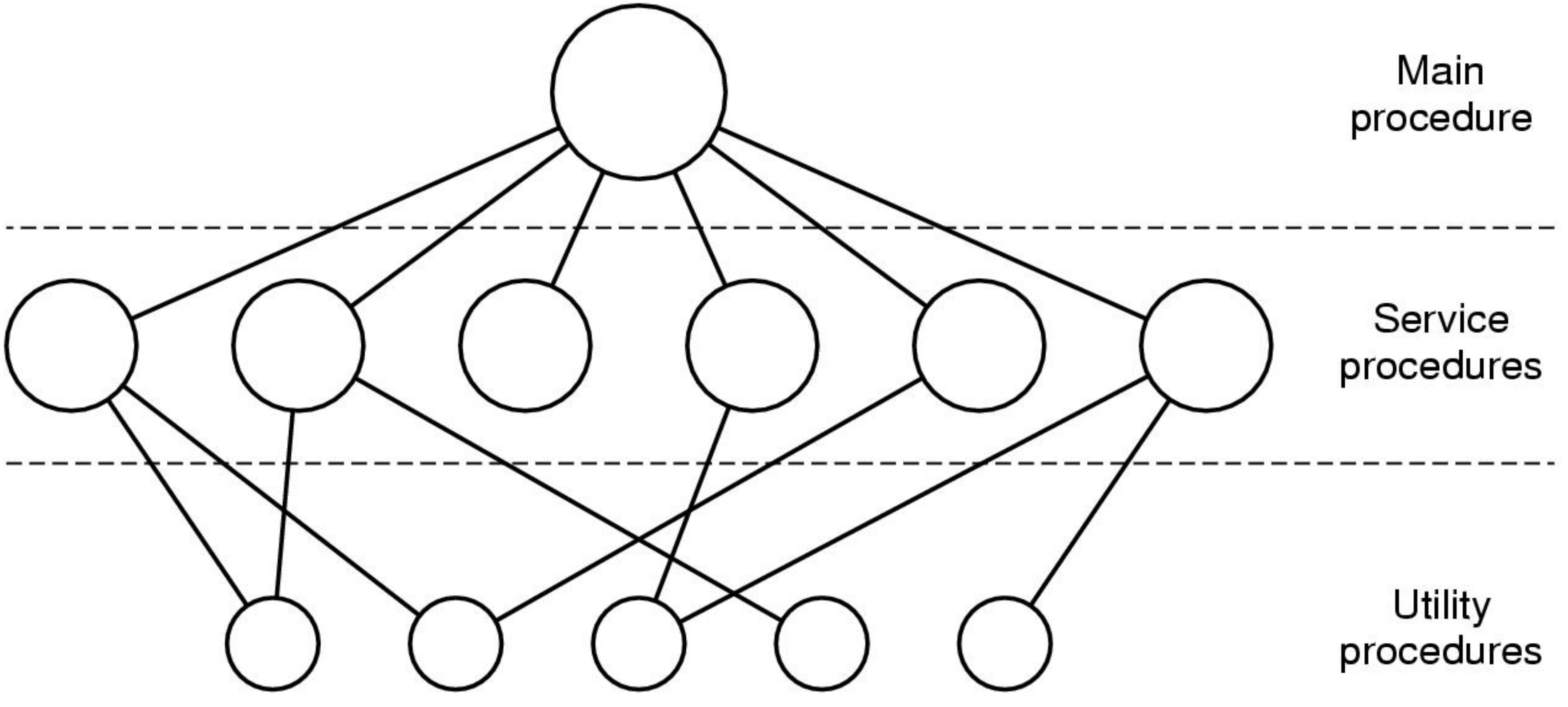
Figure 1-22. (a) File system before the mount.
(b) File system after the mount.

Operating Systems Structure

Monolithic systems – basic structure:

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

Operating System Structure



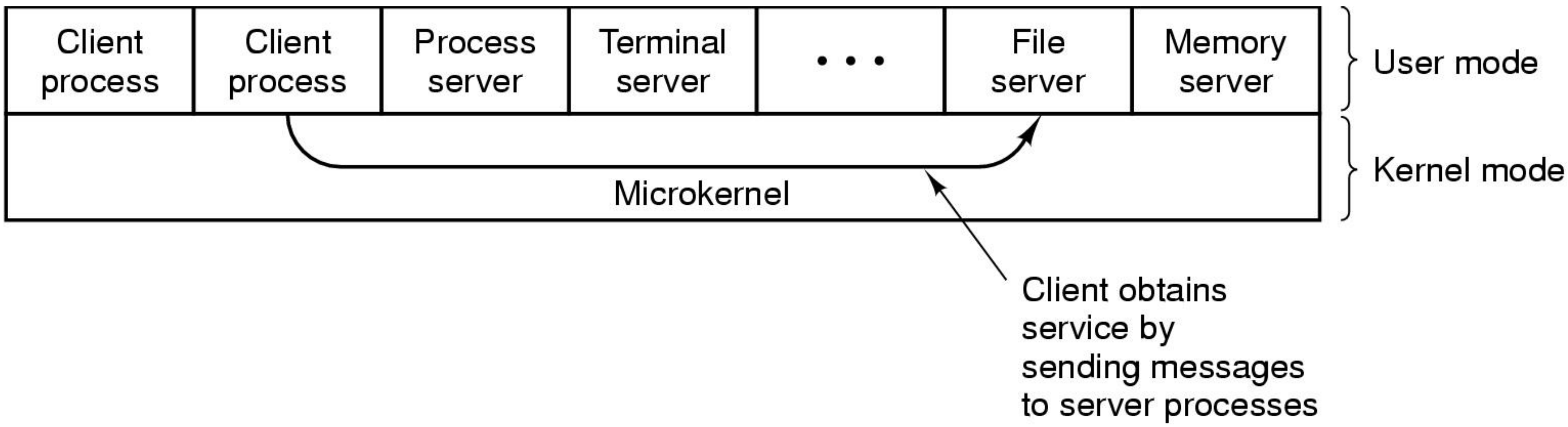
Simple structuring model for a monolithic system

Layered Systems

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 1-25. Structure of the THE operating system.

Client-Server Model



The client-server model

Client-Server Model

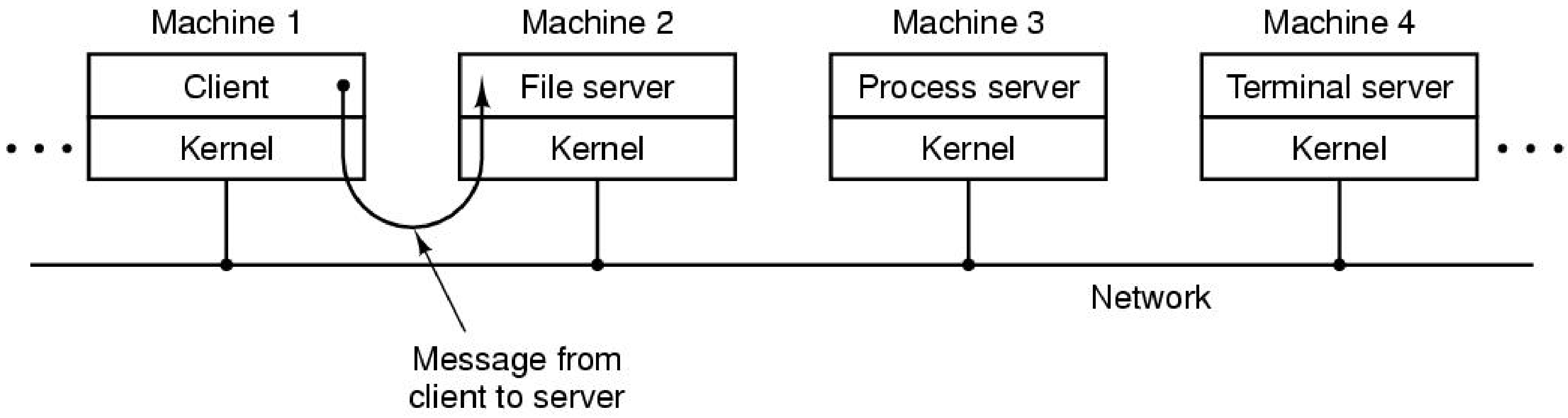


Figure 1-27. The client-server model over a network.

Virtual Machines (1)

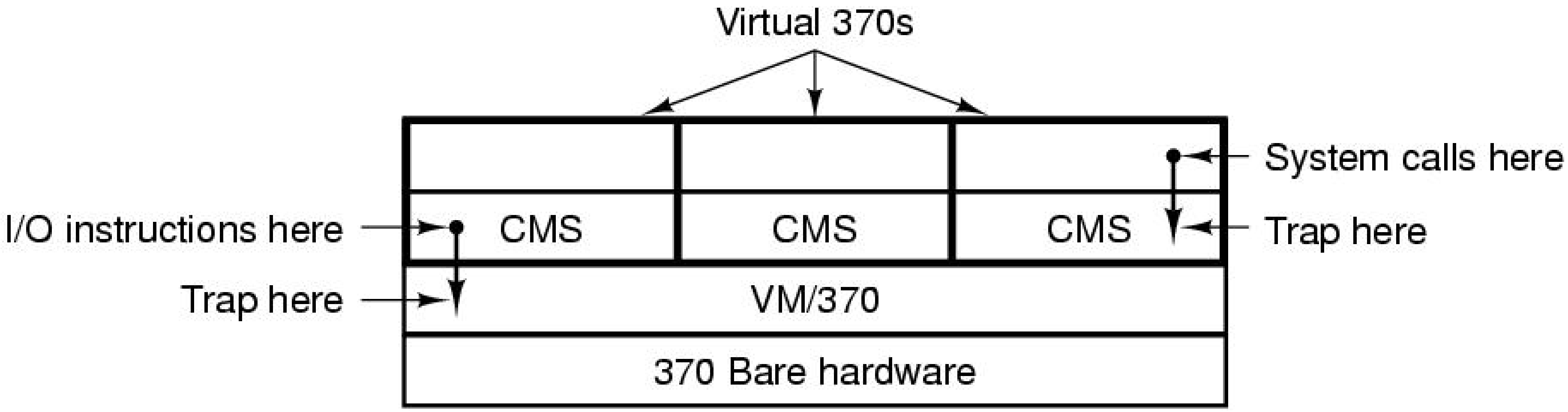


Figure 1-28. The structure of VM/370 with CMS.