Chapter 1
Introduction
What Is An Operating System (1)

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 (2)

Figure 1-1. Where the operating system fits in.
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
Figure 1-3. An early batch system.
(a) Programmers bring cards to 1401.
(b) 1401 reads batch of jobs onto tape.
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.
Figure 1-4. Structure of a typical FMS job.
ICs and Multiprogramming

Figure 1-5. A multiprogramming system with three jobs in memory.
Figure 1-6. Some of the components of a simple personal computer.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
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.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
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.
Figure 1-11. (a) The steps in starting an I/O device and getting an interrupt.
Buses

Figure 1-12. The structure of a large Pentium system
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
- The shell
- Development of computer systems
  - Large memories
  - Protection hardware
  - Disks
  - 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.
Files (1)

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

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
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.
Figure 1-16. Two processes connected by a pipe.
Figure 1-17. The 11 steps in making the system call read(fd, buffer, nbytes).

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
### System Calls for Process Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pid = fork()</code></td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td><code>pid = waitpid(pid, &amp;statloc, options)</code></td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td><code>s = execve(name, argv, environp)</code></td>
<td>Replace a process’ core image</td>
</tr>
<tr>
<td><code>exit(status)</code></td>
<td>Terminate process execution and return status</td>
</tr>
</tbody>
</table>

Figure 1-18. Some of the major POSIX system calls.
### System Calls for File Management (1)

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd = open(file, how, ...)</code></td>
<td>Open a file for reading, writing, or both</td>
</tr>
<tr>
<td><code>s = close(fd)</code></td>
<td>Close an open file</td>
</tr>
<tr>
<td><code>n = read(fd, buffer, nbytes)</code></td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td><code>n = write(fd, buffer, nbytes)</code></td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td><code>position = lseek(fd, offset, whence)</code></td>
<td>Move the file pointer</td>
</tr>
<tr>
<td><code>s = stat(name, &amp;buf)</code></td>
<td>Get a file’s status information</td>
</tr>
</tbody>
</table>

Figure 1-18. Some of the major POSIX system calls.
<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = mkdir(name, mode)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>s = rmdir(name)</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>s = link(name1, name2)</td>
<td>Create a new entry, name2, pointing to name1</td>
</tr>
<tr>
<td>s = unlink(name)</td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td>s = mount(special, name, flag)</td>
<td>Mount a file system</td>
</tr>
<tr>
<td>s = umount(special)</td>
<td>Unmount a file system</td>
</tr>
</tbody>
</table>

Figure 1-18. Some of the major POSIX system calls.
## Miscellaneous System Calls

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = chdir(dirname)</td>
<td>Change the working directory</td>
</tr>
<tr>
<td>s = chmod(name, mode)</td>
<td>Change a file’s protection bits</td>
</tr>
<tr>
<td>s = kill(pid, signal)</td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td>seconds = time(&amp;seconds)</td>
<td>Get the elapsed time since Jan. 1, 1970</td>
</tr>
</tbody>
</table>

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

```c
#define TRUE 1

while (TRUE) {
    type_prompt();
    read_command(command, parameters);
    /* display prompt on the screen */
    /* read input from terminal */

    if (fork() != 0) {
        /* 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.
Memory Layout

Figure 1-20. Processes have three segments: text, data, and stack.
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.
## Windows Win32 API

<table>
<thead>
<tr>
<th>UNIX</th>
<th>Win32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>CreateProcess</td>
<td>Create a new process</td>
</tr>
<tr>
<td>waitpid</td>
<td>WaitForSingleObject</td>
<td>Can wait for a process to exit</td>
</tr>
<tr>
<td>execve</td>
<td>(none)</td>
<td>CreateProcess = fork + execve</td>
</tr>
<tr>
<td>exit</td>
<td>ExitProcess</td>
<td>Terminate execution</td>
</tr>
<tr>
<td>open</td>
<td>CreateFile</td>
<td>Create a file or open an existing file</td>
</tr>
<tr>
<td>close</td>
<td>CloseHandle</td>
<td>Close a file</td>
</tr>
<tr>
<td>read</td>
<td>ReadFile</td>
<td>Read data from a file</td>
</tr>
<tr>
<td>write</td>
<td>WriteFile</td>
<td>Write data to a file</td>
</tr>
<tr>
<td>lseek</td>
<td>SetFilePointer</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td>stat</td>
<td>GetFileAttributesEx</td>
<td>Get various file attributes</td>
</tr>
<tr>
<td>mkdir</td>
<td>CreateDirectory</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>rmdir</td>
<td>RemoveDirectory</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>link</td>
<td>(none)</td>
<td>Win32 does not support links</td>
</tr>
<tr>
<td>unlink</td>
<td>DeleteFile</td>
<td>Destroy an existing file</td>
</tr>
<tr>
<td>mount</td>
<td>(none)</td>
<td>Win32 does not support mount</td>
</tr>
<tr>
<td>umount</td>
<td>(none)</td>
<td>Win32 does not support mount</td>
</tr>
<tr>
<td>chdir</td>
<td>SetCurrentDirectory</td>
<td>Change the current working directory</td>
</tr>
<tr>
<td>chmod</td>
<td>(none)</td>
<td>Win32 does not support security (although NT does)</td>
</tr>
<tr>
<td>kill</td>
<td>(none)</td>
<td>Win32 does not support signals</td>
</tr>
<tr>
<td>time</td>
<td>GetLocalTime</td>
<td>Get the current time</td>
</tr>
</tbody>
</table>

Figure 1-23. The Win32 API calls that roughly correspond to the UNIX calls of Fig. 1-18.
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.
Monolithic Systems

```c
#define FALSE 0
#define TRUE 1
#define N 2       /* number of processes */

int turn;       /* whose turn is it? */
int interested[N];       /* all values initially 0 (FALSE) */

void enter_region(int process);    /* process is 0 or 1 */
{
    int other;        /* number of the other process */
    other = 1 - process;    /* the opposite of process */
    interested[process] = TRUE;    /* show that you are interested */
    turn = process;        /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process)    /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```

Figure 1-24. A simple structuring model for a monolithic system.
## Layered Systems

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>The operator</td>
</tr>
<tr>
<td>4</td>
<td>User programs</td>
</tr>
<tr>
<td>3</td>
<td>Input/output management</td>
</tr>
<tr>
<td>2</td>
<td>Operator-process communication</td>
</tr>
<tr>
<td>1</td>
<td>Memory and drum management</td>
</tr>
<tr>
<td>0</td>
<td>Processor allocation and multiprogramming</td>
</tr>
</tbody>
</table>

Figure 1-25. Structure of the THE operating system.
Microkernels

Figure 1-26. Structure of the MINIX 3 system.
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.
Virtual Machines (2)

Figure 1-29. (a) A type 1 hypervisor. (b) A type 2 hypervisor.
The World According to C

- The C language
- Header files
- Large programming projects
- The model of run time
The Model of Run Time

Figure 1-30. The process of compiling C and header files to make an executable.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639