CS450/550
Operating Systems

Lecture 1  Introductions to OS and Unix

Dr. Xiaobo Zhou
Department of Computer Science

Chapter 1: Introduction

1.1 What is an operating system
1.2 History of operating systems
1.3 The operating system zoo
1.4 Computer hardware review
1.5 Operating system concepts
1.6 System calls
1.7 Operating system structure
10.2 UNIX
Introduction

A computer system consists of
- hardware
- system programs
- application programs

Instruction Set Architecture

Operating Systems

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
  - Protection domain

- It is a resource manager
  - Each program gets time with the resource, e.g., CPU
  - Each program gets space on the resource, e.g., MEM
Layers of a Computer System

- End User
- Programmer
- Application Programs
- Utilities
- Operating System
- Computer Hardware

Figure  Layers and Views of a Computer System

Services Provided by the OS

- Program development - editors and debuggers
- Program execution
- Access to I/O devices
- Controlled access to files
- System access
- Error detection and response
- Accounting

CS450/550 P&T.6 Adapted from MOS2E
**Kernel**

- Portion of operating system that is in main memory
- Contains most-frequently used functions
- Protected from user tampering by the hardware

---

**Are compilers and editors part of an OS? Why?**

```
user --------------- Trap (system calls) --------------- kernel
```
History of Operating Systems (OS)

- First generation 1945 - 1955
  - vacuum tubes, plug boards
- Second generation 1955 - 1965
  - transistors, batch systems
- Third generation 1965 – 1980
  - ICs and multiprogramming
- Fourth generation 1980 – present
  - personal computers

History of OS (2nd) - Batching

Early no-interaction batch systems in mainframes using job control lang.
- bring cards to 1401
- read cards to tape
- put tape on 7094 which does computing
- put tape on 1401 which prints output

What is the main purpose of job batching?
Overlap the reading/writing time with computing time
### History of OS (3rd) - Multiprogramming

- What are major disadvantages of batching systems?
  - No interaction support; bad for debugging for example
  - I/O operations make CPU idle

- What is the purpose of Multiprogramming?
  - To overlap one job's I/O time with another job's computing time

### History of OS (3rd) – Spooling and Timesharing

- Spooling (Simultaneous Peripheral Operation On Line)
  - Whenever a running job finished, the OS can load a new job from the disk into the now empty memory partition to run
  - Input spooling and output spooling

- Time-sharing
  - A variant of multiprogramming, in which each user has an online terminal
  - What is the key difference between multiprogramming and batching?

- UNIX systems
  - System V vs. BSD
  - POSIX
  - Linux
### Batch Multiprogramming vs. Time Sharing

<table>
<thead>
<tr>
<th></th>
<th>Batch Multiprogramming</th>
<th>Time Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal objective</td>
<td>Maximize processor use</td>
<td>Minimize response time</td>
</tr>
<tr>
<td>Source of directives to operating system</td>
<td>Job control language commands provided with the job</td>
<td>Commands entered at the terminal</td>
</tr>
</tbody>
</table>

### History of OS (4th) – PC and GUI

- MS-DOS
- GUI
- Windows
- X Windows
- Network OS vs. Distributed OS

What are two key different considerations between PC OS and mainframe OS?

Interactivity and protection
The Operating System Zoo

- Mainframe operating systems
- Server operating systems
- Multiprocessor operating systems
- Personal computer operating systems
- Real-time operating systems (hard and soft)
- Embedded operating systems
- Smart card operating systems

Computer Hardware Review

- Basic components of a simple personal computer

- Processors
  - General-purpose registers
  - Special-purpose registers; PC, IR, SP, PSW, etc.
Computer Hardware – Pipelining and Superscalar

(a) A three-stage pipeline
(b) A superscalar CPU

What are key issues in pipelining and superscalar?
Hazards and out-of-order execution (dependencies)

Computer Hardware Review – Memory Hierarchy

Typical access time               Typical capacity

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 nsec</td>
<td>Registers</td>
<td>&lt;1 KB</td>
</tr>
<tr>
<td>2 nsec</td>
<td>Cache</td>
<td>1 MB</td>
</tr>
<tr>
<td>10 nsec</td>
<td>Main memory</td>
<td>64-512 MB</td>
</tr>
<tr>
<td>10 nsec</td>
<td>Magnetic disk</td>
<td>5-50 GB</td>
</tr>
<tr>
<td>100 sec</td>
<td>Magnetic tape</td>
<td>20-100 GB</td>
</tr>
</tbody>
</table>

A typical memory hierarchy

° Cache line (block)
CA: Who Cares About the Memory Hierarchy?

Processor-DRAM Memory Gap (latency)

Processor-Memory Performance Gap:
(grows 50% / year)

"Moore's Law"

Year

Performance


CPU

DRAM

Clock hardware: generate interrupts at known intervals

Clock Software:
- Maintaining the time of day
- support time-shared scheduling
- Accounting CPU usage
- Handling the alarm system call
- ...

Clocks / Timers
A stack of platters, a surface with a magnetic coating

Typical numbers (depending on the disk size):
- 500 to 2,000 tracks per surface
- 32 to 128 sectors per track
  - A sector is the smallest unit that can be read or written

Originally, all tracks have the same number of sectors:
- "Constant" bit density: record more sectors on the outer tracks

Disk head: each side of a platter has separate disk head

Read/write data is a three-stage process:
- Seek time: position the arm over the proper track
- Rotational latency: wait for the desired sector to rotate under the read/write head
- Transfer time: transfer a block of bits (sector) under the read-write head

Average seek time as reported by the industry:
- Typically in the range of 8 ms to 15 ms
- (Sum of the time for all possible seek) / (total # of possible seeks)

Due to locality of disk reference
- Actual average seek time may only be 25% to 33% of the advertised number
**Computer Hardware Review – Memory Management**

- Multiprogramming

1. How to protect the programs from one another and the kernel from them all?

2. How to handle relocation?

Virtual memory space/address → Physical memory space/address

---

**Computer Hardware Review - MMU**

(a) One base-limit register pair  
(b) two base-limit register pairs for code sharing
Computer Hardware Review – MMU (example)

- A program is 5,000 bytes long and is loaded at physical address 10,000. What values do the base and limit register get?

Why we prefer to use virtual address in the limit register?

The address addition and comparison can start simultaneously

Computer Hardware Review – I/O Devices

- I/O devices consist of two parts: a controller and the device itself
  - What a controller for?
    - To provide a simple interface of device control to OS
    - An example on P.28
  
- Device driver
    - The software that talks to a controller, giving it commands and accepting responses
    - How to put a device driver into the OS?
      - Three ways on P.29
The Operating System must be able to prevent:
- The user program from communicating with the I/O device directly

If user programs could perform I/O directly:
- Protection to the shared I/O resources could not be provided

Three types of communication are required:
- The OS must be able to give commands to the I/O devices
- The I/O device must be able to notify the OS when the I/O device has completed an operation or has encountered an error
- Data must be transferred between memory and an I/O device

Two methods are used to address the device:
- Special I/O instructions
- Memory-mapped I/O
- A device has registers to provide status and control information

Special I/O instructions specify:
- Both the device number and the command word
  - Device number: the processor communicates this via a set of wires normally included as part of the I/O bus
  - Command word: this is usually send on the bus’s data lines
- Examples: early Intel 80x86 and IBM 370; waning in popularity

Memory-mapped I/O:
- Portions of the address space are assigned to I/O device registers
- Read and writes to those addresses are interpreted as commands to the I/O device registers
- User programs are prevented from issuing I/O operations directly:
  - The I/O address space is protected by the address translation
CA: I/O Device Notifying the OS

° The OS needs to know when:
  • The I/O device has completed an operation
  • The I/O operation has encountered an error

° This can be accomplished in three different ways:
  • Polling:
    - The I/O device put information in a status register
    - The OS periodically check the status register
  • I/O Interrupt:
    - Whenever an I/O device needs attention from the processor, it interrupts the processor from what it is currently doing.

       In real-time systems, a hybrid approach is often used
       Use a clock to periodically interrupt the CPU, at which time the CPU polls all I/O devices
  • DMA:
    - Delegate I/O responsibility from CPU

Computer Hardware Review – Interrupts

° Interrupts
  • An interruption of the normal sequence of execution
  • Improves processing efficiency
  • Allows the processor to execute other instructions while an I/O operation is in progress
  • A suspension of a process caused by an event external to that process and performed in such a way that the process can be resumed

° Classes of interrupts
  • I/O
  • Program (exception)
    - arithmetic overflow
    - division by zero
    - reference outside user’s memory space
  • Timer, Hardware failure
Computer Hardware Review – I/O Interrupt

(a) Steps in starting an I/O device and getting interrupt

(b) How the CPU is interrupted, involving talking the interrupt, running the interrupt handler, and returning to the user program

What is the key difference between interrupts and traps (system calls)?
program-triggered vs. event-triggered; synchronous vs. asynchronous

Simple Interrupt Processing

Hardware
- Device controller or other system hardware issues an interrupt
- Processor finishes execution of current instruction
- Processor signals acknowledgment of interrupt
- Processor pushes PSW and PC onto control stack
- Processor loads new PC value based on interrupt

Software
- Save remainder of process state information
- Process interrupt
- Restore process state information
- Restore old PSW and PC
Multiple Interrupts

- **Sequential Order**
  - Disable interrupts so processor can complete task, and processor ignores any new interrupt request signals
  - Interrupts remain pending until the processor enables interrupts
  - After interrupt handler routine completes, the processor checks for additional interrupts

- **Priorities**
  - Higher priority interrupts cause lower-priority interrupts to wait
  - Causes a lower-priority interrupt handler to be interrupted
  - Example when input arrives from communication line, it needs to be absorbed quickly to make room for more input

---

Computer Hardware Review - Buses

Structure of a large Pentium system
Operating System Concepts - Processes

° Process: a fundamental OS concept
  • Memory address space
  • Some set of registers
  • Protection domain
  • Resource allocation unit

° Process table in OS

° A process tree in UNIX
  • A created two child processes, B and C
  • B created three child processes, D, E, and F
  • IPC

Operating System Concepts - Deadlocks

° Deadlocks: multiple processes are competing shared resources but no resolution

(a) A potential deadlock. (b) an actual deadlock.
Operating System Concepts – Memory Management

Logical program in its virtual address space

Address translation

Actual locations of the pages in physical memory

Operating System Concepts – File Systems

A hierarchical file system for a university department
System Calls

- System calls: interface between the OS and the user programs

There are 11 steps in making the system call `read(fd, buffer, nbytes)`

- What is the main difference between a system call and a procedure call?

Some POSIX System Calls For Process Management

<table>
<thead>
<tr>
<th>Process management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call</strong></td>
</tr>
<tr>
<td><code>pid = fork()</code></td>
</tr>
<tr>
<td><code>pid = waitpid(pid, &amp;statloc, options)</code></td>
</tr>
<tr>
<td><code>s = execve(name, argv, environ)</code></td>
</tr>
<tr>
<td><code>exit(status)</code></td>
</tr>
</tbody>
</table>

When `fork()` can cause a failure?
### Some POSIX System Calls For File Management

<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>

### Some POSIX System Calls For Directory Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = mkdir(name, mode)</code></td>
<td>Create a new directory</td>
</tr>
<tr>
<td><code>s = rmdir(name)</code></td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td><code>s = link(name1, name2)</code></td>
<td>Create a new entry, name2, pointing to name1</td>
</tr>
<tr>
<td><code>s = unlink(name)</code></td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td><code>s = mount(special, name, flag)</code></td>
<td>Mount a file system</td>
</tr>
<tr>
<td><code>s = umount(special)</code></td>
<td>Unmount a file system</td>
</tr>
</tbody>
</table>
Some POSIX System Calls For Miscellaneous Tasks

<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>

Shell

Shell: a commander interpreter

- A user application program that uses the system call interface to implement an operator’s console, thus, not part of OS
### UNIX Shell

- The shell starts with a prompt, waits for standard input
  - A child process is created for a command, which runs the program
  - Shell waits for the child process to terminate
  - When the child process finishes, the shell types the prompt again and tries to read the next input line
- Examples:
  - `$ date`
  - `$ data > file`; *redirection* from standard output
  - `$ cat file1 file2 | sort &`; *(1-stage) pipelining with background*

Why creating a child process to execute a command, instead of by itself?

- Protect itself from any fatal errors that might arise during execution

---

### The Shell Strategy

- The shell isolates itself from program failures by creating a child process to execute each command/program

- Printing a prompt
- Getting the command line
- Parsing the command
- Finding the file
- Executing the program
System Calls for Shell Design

° A stripped down shell:

```c
while (TRUE) { /* repeat forever */
    type_prompt( ); /* display prompt */
    read_command (command, parameters) /* input from terminal */
    if (fork() != 0) { /* fork child process */
        /* Parent code here ......*/
        waitpid( -1, &status, 0); /* wait for child to exit */
    } else {
        /* Child code here */
        execve (command, parameters, 0); /* code for the child ...*/
        /* exec commands*/
    }
} /* char *path, char *argv[], char *envp[] */
```

Concurrent Processes

° How to support background execution ‘&’ in a Shell?

- The shell creates the child process, start it executing on the designated command, but not have the parent wait for the child to terminate
- The parent and the child are executing concurrently
- The parent prints another prompt to stdout and waits for the user to enter another command line
- All processes share the same stdin (the keyboard) and stdout (the terminal display)
How processes in Unix can communicate with each other?

- (1) Message passing: create a channel between two processes
- A pipe is a kernel buffer that can be read and written, even when there is no shared address space; the kernel creates the pipe as a kernel FIFO structure with two file descriptors

```c
int pipeID[2];
...
pipe(pipeID);
```

A Parent-Child Pipe

For two processes to share an anonymous pipe for IPC, an ancestor of the processes must create the pipe prior to creating the processes, and let child processes to inherit (and share) the pipe

- Pipes are FIFO, asynchronous `send()` and blocking `receive()`

```c
... pipe(pipeID);
if(fork()) == 0) { /* The child process */
  ...
  read(pipeID[0], childBuf, len);
  /* process the message in childBuf */
  ...
} else { /* The parent process */
  ...
  /* Send a message to the child */
  write(pipeID[1], msgToChild, len);
  ...
}
```

A parent-child pipe
Two-way Pipes

- How to have bi-directional communication?

```c
int A_to_B[2], B_to_A[2];
min(){
pipes(A_to_B);
pipes(B_to_A);
if (fork()==0) { /* This is the first child process */
    execute("prog_A.out", ...);
    exit(1); /* Error-terminate the child */
}
if (fork()==0) { /* This is the second child process */
    execute("prog_B.out", ...);
    exit(1); /* Error-terminate the child */
}
/* This is the parent process code */
wait( ...);
wait( ...);
}
```

Two processes connected by two pipes for 2-way communication

Two-way Pipes (cont.)

```c
proc A(){
    while (TRUE) {
        <compute A1>;
        write(A_to_B[1], x, sizeof(int));
        /* Use this pipe to send info */
        <compute A2>;
        read(B_to_A[0], y, sizeof(int));
        /* Use this pipe to get info */
    }
}

proc B(){
    while (TRUE) {
        read(A_to_B[0], x, sizeof(int));
        /* Use this pipe to get info */
        <compute B1>;
        write(B_to_A[1], y, sizeof(int));
        /* Use this pipe to send info */
        <compute B2>;
    }
}
```

Two processes connected by two pipes for 2-way communication
POSIX Signals

How processes in Unix can communicate with each other?

- (2) Signals: processes can send signals to its process group and also tell the system what they want to happen when a signal arrives

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

The signals required by POSIX.

Pipelining and I/O Redirection in Shell

How to support ‘<’, ‘>’, and ‘|’ in a Shell?

- $ cat file1 file2 | sort > foo
- A created process has three default file descriptors, stdin, stdout, and stderr
- The shell can redirect I/O by manipulating the child processes’ file descriptors
  - Each child process has its own file descriptor table
    - fileDescriptor[0] : stdin (mapped to keyboard)
    - fileDescriptor[1] : stdout (mapped to terminal display)
    - fileDescriptor[2] : stderr (mapped to terminal display)
  - A code fragment does output redirection
    - fid = open (foo, O_WRONLY | O_CREAT);
    - close(1); // close the fileDescriptor[1] - stdout
    - dup(fid); // re-use the earliest available file descriptors
    - … ; the one just closed - stdout
    - close(fid)
System Calls - Segments

- Processes have three segments: text, data, stack

System Calls – File and Directory Management

- UNIX i-number, an integer number for file descriptors
  - The index into a table of i-nodes, one per file, telling who owns the file, where its blocks are, and so on.

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

(a) Two directories before linking

(b) The same directories after linking `usr/jim/memo` to ast's directory
System Calls – UNIX Mount System

- UNIX Mount System: to integrate removable media into a single integrated file systems, without having to worry about which device a file is on

```bash
mount("/dev/fd0", "/mnt", 0);
```

(a) File system before the mount           (b) File system after the mount

(a) File system before the mount           (b) File system after the mount

System Calls – The 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>GetFileAttributes</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>

Some Win32 API calls
Operating System Structures

- Monolithic systems
- Layered systems
- Virtual machines
- Exokernels
- Client-server systems

UNIX Layers

The layers in a UNIX system
Modular Structure

- Dynamic linking
- Stackable modules

Summary of Lecture 1

- Two major OS functionalities:
  - machine extension and resource management

- History of OS:
  - Batching, multiprogramming, time-sharing, PC

- Computer Architecture Reviews

- Fundamental OS concepts:
  - Process, memory management, deadlocks, file & directory management

- System calls and Unix Shell

- More reading: textbook 1.1 - 1.7, Ch10: 671 - 696