Today’s Plan

Upcoming:
- Quiz #1 today!
- Assignment 1

Last time:
- Operating system duties

Today’s topics:
- From last time: Management duties, Protection/security, & computing environments
- Operating System Services
- User Operating System Interface
- System Calls
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
Operating System Services

- User Interface (UI)
- Program execution:
- I/O operations
  - User programs cannot execute I/O operations directly, so the operating system must provide means to perform I/O
- File-system manipulation
- Communications – exchange of information between processes executing either on the same computer or on different systems tied together by a network.
Operating System Services

- Error detection – ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs.

Additional functions exist not for helping the user, but rather for ensuring efficient system operations:

- Resource allocation – allocating resources to multiple users or multiple jobs running at the same time.

- Accounting – keep track of and record which users use how much and what kinds of computer resources
A View of Operating System Services
Many commands are given to the operating system by control statements typed at the keyboard (for example).

The program that reads and interprets control statements is called variously:

Its function is to get and execute the next command statement.
User Operating System Interface –
Graphical User Interface (GUI)

- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Invented at Xerox PARC

Many systems now include both CLI and GUI interfaces

- Microsoft Windows is GUI with CLI “command” shell
- Apple Mac OS X has “Aqua” GUI interface with UNIX kernel underneath and shells available
- Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)
Bourne Shell CLI vs. Mac OS/X GUI
Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands
System Calls

- System calls provide the interface between a running program and the operating system.
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use.
- Three most common APIs are:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)
System Calls

Why use APIs rather than system calls?

Types of system calls:
- Process control
- File management
- Device management
- Information Maintenance
- Communications
- Protection
Example of System Calls

System call sequence to copy the contents of one file to another file:

- Acquire input file name
- Write prompt to screen
- Accept input
- Acquire output file name
- Write prompt to screen
- Accept input
- Open the input file
  - if file doesn't exist, abort
- Create output file
  - if file exists, abort
- Loop
  - Read from input file
  - Write to output file
  - Until read fails
- Close output file
- Write completion message to screen
- Terminate normally
# Examples of Windows and Unix System Calls

<table>
<thead>
<tr>
<th>Process Control</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CreateProcess()</td>
<td>fork()</td>
</tr>
<tr>
<td></td>
<td>ExitProcess()</td>
<td>exit()</td>
</tr>
<tr>
<td></td>
<td>WaitForSingleObject()</td>
<td>wait()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File Manipulation</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CreateFile()</td>
<td>open()</td>
</tr>
<tr>
<td></td>
<td>ReadFile()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>WriteFile()</td>
<td>write()</td>
</tr>
<tr>
<td></td>
<td>CloseHandle()</td>
<td>close()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device Manipulation</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SetConsoleMode()</td>
<td>ioctl()</td>
</tr>
<tr>
<td></td>
<td>ReadConsole()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>WriteConsole()</td>
<td>write()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Information Maintenance</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GetCurrentProcessID()</td>
<td>getpid()</td>
</tr>
<tr>
<td></td>
<td>SetTimer()</td>
<td>alarm()</td>
</tr>
<tr>
<td></td>
<td>Sleep()</td>
<td>sleep()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Communication</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CreatePipe()</td>
<td>pipe()</td>
</tr>
<tr>
<td></td>
<td>CreateFileMapping()</td>
<td>shmget()</td>
</tr>
<tr>
<td></td>
<td>MapViewOfFile()</td>
<td>mmap()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protection</th>
<th>Windows</th>
<th>Unix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SetFileSecurity()</td>
<td>chmod()</td>
</tr>
<tr>
<td></td>
<td>InitializeSecurityDescriptor()</td>
<td>umask()</td>
</tr>
<tr>
<td></td>
<td>SetSecurityDescriptorGroup()</td>
<td>chown()</td>
</tr>
</tbody>
</table>
System Call Implementation

- Typically, a number associated with each system call
- System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
API – System Call – OS Relationship
Standard C Library Example

C program invoking `printf()` library call, which calls `write()` system call

```c
#include <stdio.h>
int main ()
{
    ...
    ...
    printf("Greetings");
    ...
    return 0;
}
```
Three general methods are used to pass parameters between a running program and the operating system:

- Pass parameters in **registers**
- Store the parameters in a **table** in memory
- Use a **stack**

Table and stack methods do not limit the number of parameters
Parameter Passing via Table
System Programs

System programs provide a convenient environment for program development and execution. Examples:

- File manipulation
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications
- Application programs

Most users’ view of the operating system is defined by system programs, not the actual system calls.
Example: The Linker and Loader

- **Source Program**
  - main.c
  - gcc -c main.c
    - generates
    - main.o
  - gcc -o main main.o -lm
    - generates
    - main
  - .main

- **Linker**
  - Other object files
  - linker
  - executable file
  - loader
  - program in memory

- **Loader**
  - Dynamically linked libraries
Operating System Design & Implementation

- **User goals** – operating system should be:

- **System goals** – operating system should be:
Important principle to separate

Policy:

Mechanism:

Why have this separation?
Operating System Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++

- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts

- High-level languages easier to port to other hardware
  - But slower

- Emulation can allow an OS to run on non-native hardware
MS-DOS – written to provide the most functionality in the least space

Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

With modularity, layers are selected such that each uses functions and services of only lower-level layers.
UNIX – limited by hardware functionality, the original UNIX operating system was a *monolithic kernel*.

The UNIX OS consists of two separable parts:

- Systems programs
- The kernel
  - Consists of everything below the system-call interface and above the physical hardware
  - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
Traditional UNIX Structure

- **Kernel interface to the hardware**
  - terminal controllers
  - device controllers
  - memory controllers

- **System-call interface to the kernel**
  - signals terminal handling
  - character I/O system
  - terminal drivers
  - file system swapping block I/O system
  - disk and tape drivers
  - CPU scheduling
  - page replacement
  - demand paging
  - virtual memory

- **(the users)**
  - shells and commands
  - compilers and interpreters
  - system libraries
Microkernel System Structure

- Moves as much from the kernel into "user" space

- Benefits:
  - 
  - 
  - 
  - 

- Detriments:
  - Performance overhead of user space to kernel space communication
Monolithic vs. Microkernel Structure

- Most popular modern OSes are actually hybrids of the monolithic and microkernel structures:
  - Many functions are moved into “user” space
  - Some are kept in the kernel for performance reasons
  - E.g. Mac OS/X structure:
A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.

The operating system host creates the illusion that a process has its own processor and memory.

Each guest is provided with a (virtual) copy of underlying computer.
Virtual Machines

(a) Nonvirtual machine (b) virtual machine
VMWare Architecture

- Host operating system (Linux)
- Hardware:
  - CPU
  - Memory
  - I/O devices
- Virtualization layer
- Guest operating systems:
  - guest operating system (free BSD)
  - guest operating system (Windows NT)
  - guest operating system (Windows XP)
- Virtual devices:
  - virtual CPU
  - virtual memory
Java Virtual Machine

Diagram:
- Java program .class files
- class loader
- Java interpreter
- host system (Windows, Linux, etc.)
- Java API .class files
Operating-System Debugging

- **Debugging** is finding and fixing errors, or bugs
- OSes generate log files containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Kernighan’s Law: “**Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.**”