COMS 1001 Spring 2006

Introduction to Computers

Operating Systems

What's Ahead

- Introduction to Operating Systems
  - Shell
  - Kernel
  - Support for applications
  - System call interface
  - Files, threads, processes, memory, devices

What is an Operating System?

- A program that manages the execution of other programs
- A program that abstracts away the detail of handling hardware (*resource abstraction*)
- A layer between software applications and the underlying hardware (CPU, memory, disks, devices)
- The operating system is an *interface*.
- The “real” operating system is the **kernel**.

Properties of an OS

- The most critical piece of software in a computer system, but also “underappreciated”: the application is king.
  - Performance, Functionality, Invisibility
- Application software is designed to solve a specific problem
- System software (like an OS) is designed to provide a support environment for application software

Design Factors

- Performance cost: how much does the OS abstraction cost me?
- Protection: can the OS protect the hardware and data from the user?
- Correctness: is the OS stable and reliable? Is it accurate, predictable, and precise?
- Maintainability: is the OS easy to upgrade and fix?

Protection

- Allowing unfettered control of the hardware is an invitation for abuse
- Hardware can execute in 2 modes: supervisor and normal (in theory, there are more, the Pentium allows for 4 modes)
- This division is reflected in the OS. Certain operations are privilaged (any type of I/O)
- Protection seeks to isolate applications and users

Some Protection Mechanisms

- Supervisor/user mode (hardware)
- Base and limit registers (define memory region for program execution)
- System calls (ask OS to execute supervisor code on user behalf in a well-defined manner)

History of Operating Systems

- Programming for early computers used punch cards and other archaic techniques to directly manipulate the hardware
- Programmers had to load the program (cards) in, run the job, and remove their output. Programmers had to manage the tedious details of hardware manipulation and I/O
- First OS's were *batch operating systems*: ran a series of 'jobs' without involving the programmer manually loading in cards.

History of the OS (cont.)

- Batch operating systems were good b/c they required no interaction with the user and abstracted away some details of hardware usage and job scheduling
- Timesharing systems: these OS's provided support for multiple interactive sessions
  - Key was response time to user
  - A CPU can only do 1 task at a time, need to repeatedly switch tasks fast enough for a human to be fooled into thinking many things are happening at once

History (cont.)

- PC's or personal computers were introduced as an alternative to the timesharing mainframes. The PC provided a multi-programmed environment for one user
- Eventually, PC's were made to support multiple users

OS Responsibilities

- Resource Management
- Process Scheduling
- Memory Management
- Filesystem management
- Networking and I/O handling
- Provide an API (Application Programmer's Interface)
- Implement the API via *system calls* and *interrupts*.
- Protection!

Resource Sharing

- One of the most important jobs of the OS is managing resources
  - Space-multiplexed: the resource can be divided into two or more logical distinct units.
  - Time-multiplexed: the resource is not divided into units, but rather is dedicated during short time slices.
  - Memory is space-multiplexed
  - CPU is time-multiplexed

Processes: the basic unit of computation

- A basic OS abstraction that represents a job
- Usually defined as a program in execution
  - Program code
  - Address space (chunk of memory dedicated to the process)
  - Program Counter
  - Stack
  - All this info contained in a PCB (Process Control Block) (in Linux, the PCB is named a 'task_struct' object.)

Process (cont.)

- Processes represent user code that the OS needs to manage.
- Processes request resources (memory, disk, other I/O devices, CPU time)
- Processes are created in Unix/Linux via a special function: fork()
- Fork() clones the current process and then loads all the information into the cloned process.

Files

- Files are an abstraction that represent a collection of bytes under one name
- In Unix, everything that is not a process is a file. (for the most part)
- Files store information and can be organized or formatted according to the needs of the application using them. (ex. Java file reader)
- But files themselves need to be stored on some underlying medium and in some consistent format

The Filesystem

- Linux supports a number of filesystems
- There are many available filesystems, because there are many different ways to store information on a magnetic disk (the hard disk)
- Linux provides the VFS (virtual file system) which is an interface or abstraction layer. It represents the common operations that can be performed on files. The implementation details are up to the specific file system.

Filesystems and Hard Disks

- Hard disks are usually split up into sectors and tracks to provide some logical grouping.
- Most filesystems overlay a specific directory structure or directory tree on the underlying physical organization.
- Sectors are broken up into blocks, usually about 4096 bytes (4k). Files are stored in collections of blocks. Keeping track of which blocks belong to which files is the job of the file system part of the OS. An OS can use many filesystems at once.

The Kernel

- The Operating System *kernel* is the piece of code that implements all this low-level functionality.
- Usually written in C, some critical pieces are written in the assembly language of the target CPU
- Usually does not include graphics, GUI's, and higher level user-interaction software.
- These basic applications are often packaged with the kernel to provide a complete *distribution*.

Kernel (cont)

- The kernel is the core of any OS.
- The kernel is privilaged and can do anything to the hardware.
- OS is split into 2 basic regions:
  - Userland (said in a nice way, sometimes)
  - Kernel space (executing OS code in privileged mode)
- What are some arguments for putting user-level applications into the kernel? (like a web browser, for example)

The Linux Kernel

- A Unix clone that aims towards POSIX compliance
- Released under the GPL
- Small and compact (fits on 1 floppy)
- Monolithic (vs mico-kernel)
- Support for dynamic module loading
- Support for multiple CPUs
- Kernel threads
- Support for application level threads

What Happens When a Computer Turns On?

- BIOS (basic input-output system) runs some checks of the hardware and then loads some program from the beginning of the hard disk into memory. Tells CPU to execute this program.
- This program is often a boot-loader and is read from the Boot Sector
- The boot loader knows where to find the code for the operating system on the disk and tells the CPU to execute that code.

System Startup

- The OS creates the first process
- This process executes the OS startup code, often ending in the creation of a tty (in Unix), which is basically a command-line interface, or shell for a user to log in and enter commands.
- Commands are simply the names of programs somwhere in the filesystem
- Example: the shell
  - A simple program written in C, executed by a process

Introducing Processes Into the OS

- At any given time, a number of processes are present in the system
- The OS must manage them: giving them resources, letting them run, creating them, killing them, etc.
- The OS is responsible for **process scheduling**
- Remember, the CPU can only do 1 thing at a time (which is why a lot of high-end machines have more than one CPU)

The Process Lifecycle

- Processes can be in a number of states depending on their relationship to the CPU and resources
- PLC:
  - Processes are born or created
  - Processes are made runnable
  - A process can block or be put to sleep
  - A process can terminate and become a zombie
  - Finally, a process can be cleaned up

Process Scheduling

- The OS needs some mechanism (algorithm) for putting processes on the CPU (making them Runnable and then allowed the CPU to execute their code)
- There are many ways to accomplish this task, and each algorithm represents some of the conflicting goals
- Goals of process scheduling?
  - Fairness, priority, max resource utilization, real time

Scheduling Algorithms

- First come, first served (FCFS, FIFO)
- Shortest Job Next (SJN)
- Priority (highest priority job next)
- Round-robin (give each process one small slice of CPU time, known as a **time quantum**)
- All schemes involve a **context switch** (overhead)

Threads

- Threads are sometimes known as lightweight processes
- Processes often involve a lot of overhead to deal with (mostly due to their process address space, which includes the object code for the program and the data the program operates on)
- Threads are much like processes except they share an address space
- Threads can be built into the kernel or supplied at the user level

Java Threads

- Java has support for threads in the language.
- The JVM implements threads according to the underlying operating system
  - Linux JVM uses green threads (each thread is a separate process)
  - Native threads are implemented in a user-level library where one process maps to many executing threads
  - Windows JVM (depends on version) uses windows threads (probably)
- Example: Java threads, race

Back to Protection

- The kernel implements the interface layer via system calls
- The system calls are essentially a list of functions that encapsulate or abstract away the details of particular tasks
- Common system calls: open, read, write, exit, fork, brk/malloc
- All represent tasks that a programmer needs to write programs, but cannot be trusted to perform by themselves in supervisor mode

The System Call Mechanism

- When a system call is invoked, the kernel jumps from user mode to supervisor mode and then executes trusted code on behalf (but not under the control) of the user.
- System call bodies are carefully implemented to check their input and output for illegal use
- An interrupt is used to trap to a specific place in the OS code in memory and begin executing the system call service routine

Summary

- An OS is an abstraction or interface between application software and system hardware
- The core of an OS is the kernel
- The kernel implements the interface via system calls
- The kernel is responsible for managing hardware resources, memory, files, and processes and for providing protection between processes