CS 333
Introduction to Operating Systems

Class 1 - Introduction to OS-related Hardware and Software

Jonathan Walpole
Computer Science
Portland State University
About the instructor

- Instructor – Jonathan Walpole
  - Professor at OGI 1989 - 2004, PSU 2004 -
  - Director of Systems Software Lab at OGI
  - Research Interests: *Operating System Design, Distributed Computing Systems, Multimedia Computing and Networking*
  - http://www.cs.pdx.edu/~walpole
About CS 333

- **Goals of the class**
  - understand the basic concepts of operating systems
  - gain some practical experience so it's not just all words!

- **Expectations**
  - reading assignments should be read *before class*
  - active participation in class discussions
  - no cheating
  - be nice to the instructor!
Grading

- **Exams**
  - Mid-term - 25%
  - Final - 25%

- **Coursework**
  - project - 30%

- **Quizzes**
  - in-class quizzes and discussions - 20%
Text books

“Modern Operating Systems”  “The SPANK System”
2nd Edition  Harry Porter
A. Tannenbaum
The project

- You will read, understand and write real operating system code!
- We will be using the SPANK system, written by Harry Porter
- About SPANK
  - CPU emulator, assembler, high-level language, operating system, and debugging environment
  - Simple enough to understand “in detail” how everything works!
  - Realistic enough to understand “in detail” how everything works!
  - Runs on the departmental Sun machines
Administrative

- **Class web site**
  - www.cs.pdx.edu/~walpole/class/cs333/fall2005/home.html

- **Class mailing list**
  - https://webmail.cecs.pdx.edu/mailman/listinfo.cgi/cs333-001

- **Project 0**
  - read the class web site
  - join the class mailing list

- **Project 1**
  - due next week! See web site for project assignments.
Class 1 - Introduction to OS-related Hardware and Software
Overview

- What is an Operating System?
- A review of OS-related hardware
What is an operating system?

- Operating system --"is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware"
  - Narrow view
    - Traditional computer with applications running on it (e.g. PCs, Workstations, Servers)
  - Broad view
    - Anything that needs to manage resources (e.g. router OS, embedded devices, pagers, ...)


Two key OS functions

- **Abstract Machine**
  - Hide details of the underlying hardware
  - Provide “common” API to applications and services
  - Simplifies application writing

- **Resource Manager**
  - Controls accesses to “shared” resources
    - CPU, memory, disks, network, …
  - Allows for “global” policies to be implemented
Why is abstraction important?

- **Without OSs and abstract interfaces application writers program all device access directly**
  - load device command codes into device registers
  - handle initialization, recalibration, sensing, timing etc for physical devices
  - understand physical characteristics and layout
  - control motors
  - interpret return codes ... etc

- **Applications suffer severe code bloat!**
  - very complicated maintenance and upgrading
  - writing this code once, and sharing it, is how OS began!
Providing abstraction via system calls

Application

Operating System

Hardware
- Video Card
- Monitor
- Disk
- Printer

CPU
Memory
Network
Providing abstraction via system calls

Application

System Calls: `read()`, `open()`, `write()`, `mkdir()`, `kill()` ...

Operating System

Device Mgmt
Protection

File System

Network Comm.

Process Mgmt
Security

CPU
Memory

Video Card
Monitor
Disk
Printer

Hardware

Network

Network

Protection

File System

Process Mgmt

Device Mgmt

Operating System

Application
OS as a resource manager

- Sharing resources among applications across space and time
  - scheduling
  - allocation
- Making efficient use of limited resources
  - improving utilization
  - minimizing overhead
  - improving throughput/good put
- Protecting applications from each other
  - enforcement of boundaries
Problems an OS must solve

- Time sharing the CPU among applications
- Space sharing the memory among applications
- Space sharing the disk among users
- Time sharing access to the disk
- Time sharing access to the network
More problems an OS must solve

- **Protection**
  - of applications from each other
  - of user data from other users
  - of hardware/devices
  - of the OS itself!

- The OS needs help from the hardware to accomplish these tasks!
Overview

- What is an Operating System?
- A review of OS-related hardware
Overview of computer system layers

Hardware - CPU, memory, I/O devices - disk, network ...
Basic anatomy on a CPU (1)

- **Some key CPU Components**
  - Program Counter (PC)
    - holds memory address of next instruction
  - Instruction Register (IR)
    - holds instruction currently being executed
  - Registers (Reg. 1..n)
    - hold variables and temporary results
  - Arithmetic and Logic Unit (ALU)
    - performs arithmetic functions and logic operations
Basic anatomy on a CPU (2)

- **Some key CPU Components**
  - Memory Address Register (MAR)
    - contains address of memory to be read/written
  - Memory Data Register (MDR)
    - contains memory data read or to be written
  - Stack Pointer (SP)
    - holds memory address of a stack with a frame for each procedure's local variables & parameters
  - Processor Status Word (PSW)
    - contains the mode bit and various control bits
Program execution

- **Instruction sets**
  - different for different machines
  - all have load and store instructions for moving items between memory and registers
  - many instructions for comparing and combining values in registers and putting result in a register

- **Fetch/Decode/Execute cycle**
  - fetch next instruction pointed to by PC
  - decode it to find its type and operands
  - execute it
  - repeat
Fetch/decode/execute cycle

CPU
- PC
- IR
- Reg. 1
- ... (Reg. n)
- ALU
- MAR
- MDR

Memory
Fetch/decode/execute cycle

While (1) {
    Fetch instruction from memory
    Execute instruction
    (Get other operands if necessary)
    Store result
}
Fetch/decode/execute cycle

While (1) {
  Fetch instruction from memory
  Execute instruction
    (Get other operands if necessary)
  Store result
}
While (1) {
    Fetch instruction from memory
    Execute instruction
    (Get other operands if necessary)
    Store result
}
Fetch/decode/execute cycle

While (1) {
    Fetch instruction from memory
    **Execute instruction** (Get other operands if necessary)
    Store result
}
Fetch/decode/execute cycle

```c
While (1) {
    Fetch instruction from memory
    Execute instruction
    (Get other operands if necessary)
    Store result
}```
Fetch/decode/execute cycle

While (1) {
    Fetch instruction from memory
    Execute instruction
        (Get other operands if necessary)
    Store result
}
The OS is just a program!

- How can the OS cause application programs to run?
- How can application programs cause the OS to run?
- How can the OS switch the CPU to run a different application and later resume the first one?
- How can the OS maintain control?
- In what ways can application code try to cheat?
- And how can the OS stop the cheating?
How can the OS invoke an application?

- The computer boots and begins running the OS
  - fetch/decode/execute OS instructions

- OS can request user input to identify an application to run
  - OS loads the address of the application’s starting instruction into the PC
  - CPU fetches/decodes/executes the application’s instructions
How can applications invoke the OS?

- Trap instruction changes PC to point to an OS entry point instruction
  - application calls a library procedure that includes the appropriate trap instruction
  - fetch/decode/execute cycle begins at a specified OS entry point called a system call
How can the OS run a new application?

- To suspend execution of an application simply capture its memory state and processor state
  - copy values of all registers into a data structure and save it to memory
  - preserve the memory values of this application so it can be restarted later
How can OS guarantee to regain control?

- What if a running application doesn’t make a system call and hence hogs the CPU?
  - timer interrupts!
  - OS must register a future timer interrupt before it hands control of the CPU over to an application

- How can the OS avoid trampling on the processor state it wants to save?
  - carefully written interrupt handlers!
What if the application tries to cheat?

- What stops the running application from disabling the future timer interrupt so that the OS can not regain control?
  - the mode bit (in the PSW)!

- Certain instructions can only be executed when the mode bit is set
  - manipulating timer interrupts
  - setting the mode bit!
  - ...
What other ways are there to cheat?

- What stops the running application from modifying the OS?
  - Memory protection!
  - Can only be set with mode bit set ....

- The OS must clear the mode bit before it hands control to an application!
  - Interrupts and trap instructions set the mode bit and transfer control to specific locations (in the OS)
Why it’s not quite that simple …

- Pipelined CPUs
- Superscalar CPUs
- Multi-level memory hierarchies
- Virtual memory
- Complexity of devices and buses
- Heterogeneity of hardware
Pipelined CPUs

Execution of current instruction performed in parallel with decode of next instruction and fetch of the one after that
Superscalar CPUs
What does this mean for the OS?

- **Pipelined CPUs**
  - more complexity in capturing state of a running application
  - more expensive to suspend and resume applications

- **Superscalar CPUs**
  - even more complexity in capturing state of a running application
  - even more expensive to suspend and resume applications

- More details, but fundamentally the same task
- The SPANK CPU is not pipelined or superscalar
The memory hierarchy

- **2GHz processor** → 0.5 ns

- **Data/instruction cache** → 0.5 ns – 10 ns, 64 kB – 1 MB
  *(this is where the CPU looks first!)*

- **Main memory** → 60 ns, 512 MB – 1 GB

- **Magnetic disk** →
  10 ms, 160 Gbytes

- **Tape** → Longer than you want, less than magnetic disk!

*Figure 1.14 The Memory Hierarchy*
## Terminology review - metric units

<table>
<thead>
<tr>
<th>Exa,</th>
<th>Explicit</th>
<th>Prefix</th>
<th>Exp.</th>
<th>Explicit</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-3}$</td>
<td>0.001</td>
<td>milli</td>
<td>$10^3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>0.000001</td>
<td>micro</td>
<td>$10^6$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^{-9}$</td>
<td>0.000000001</td>
<td>nano</td>
<td>$10^9$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^{-12}$</td>
<td>0.000000000001</td>
<td>pico</td>
<td>$10^{12}$</td>
<td>1,000,000,000,000</td>
<td>Giga</td>
</tr>
<tr>
<td>$10^{-15}$</td>
<td>0.00000000000001</td>
<td>femto</td>
<td>$10^{15}$</td>
<td>1,000,000,000,000,000</td>
<td>Tera</td>
</tr>
<tr>
<td>$10^{-18}$</td>
<td>0.0000000000000001</td>
<td>atto</td>
<td>$10^{18}$</td>
<td>1,000,000,000,000,000,000</td>
<td>Exa</td>
</tr>
<tr>
<td>$10^{-21}$</td>
<td>0.00000000000000001</td>
<td>zepto</td>
<td>$10^{21}$</td>
<td>1,000,000,000,000,000,000,000</td>
<td>Zetta</td>
</tr>
<tr>
<td>$10^{-24}$</td>
<td>0.000000000000000001</td>
<td>yocto</td>
<td>$10^{24}$</td>
<td>1,000,000,000,000,000,000,000,000,000</td>
<td>Yotta</td>
</tr>
</tbody>
</table>

The metric prefixes
The memory hierarchy

- **2GHz processor** → 0.5 ns for access to a few 10s of registers

- **Data/inst. cache** → 0.5ns – 10 ns, 64 kB–1MB  
  *(this is where the CPU looks first!)*

- **Main memory** → 60 ns,  
  512 MB – 1GB

- **Magnetic disk** →  
  10 ms, 160 Gbytes

- **Tape** → Longer wait than you want, costs less than magnetic disk!

---

*Figure 1.14 The Memory Hierarchy*
Who manages the memory hierarchy?

- **Movement of data from main memory to cache** is under hardware control
  - cache lines loaded on demand automatically
  - replacement policy fixed by hardware
- **Movement of data from cache to main memory** can be affected by OS
  - instructions for “flushing” the cache
  - can be used to maintain consistency of main memory
- **Movement of data among lower levels of the memory hierarchy** is under direct control of the OS
  - virtual memory page faults
  - file system calls
OS implications of a memory hierarchy?

- How do you keep the contents of memory consistent across layers of the hierarchy?

- How do you allocate space at layers of the memory hierarchy “fairly” across different applications?

- How do you hide the latency of the slower subsystems?
  - Main memory... yikes!
  - Disk
  - Tape

- How do you protect one application’s area of memory from other applications?

- How do you relocate an application in memory?
Quiz

How does the OS solve these problems:

- Time sharing the CPU among applications?
- Space sharing the memory among applications?
- Space sharing the disk among users?
- Time sharing access to the disk?
- Time sharing access to the network?
- Protection of applications from each other?
- Protection of user data from other users?
- Protection of hardware/devices?
- Protection of the OS itself?
What to do before next class

- Reading for today’s class - pages 1-70
- Reading for next week’s class - pages 71-100
- Assignment 0 - read class web page and join class email list
- Start project 1 - Introduction to SPANK