Course Overview
Who am I?

Jonathan Walpole
Professor at PSU since 2004, OGI 1989 – 2004
Research Interests: *Operating System Design, Parallel and Distributed Computing Systems*
http://www.cs.pdx.edu/~walpole
Class Goals

An OS foundations course for graduate students with little or no OS background

Understand the basic concepts of operating systems
  *designing & building* operating systems, not just *using* them!

Gain some practical experience so that it is not just words!
Expectations

Our contract:
- You try to understand the material
- I try to help you understand the material
- We all participate enthusiastically in the class
- No cheating!

It's all about education, not earning credits
Resources

Class mailing list
https://mailhost.cecs.pdx.edu/mailman/listinfo/cs510walpole

Teaching Assistant
- Amit Joshi

Learn from me, Amit and from each other!
Text Book

“Operating Systems: Three Easy Pieces” by R. Arpaci-Dusseau and A. Arpaci-Dusseau

“The BLITZ System” by Harry Porter

All are available freely on line.
Programming Project

You will read, understand and write operating system code!

We will use the BLITZ system
   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 x86/Linux, Mac, Sun etc

You will implement locks, condition variables, concurrent resource managers, a threads pool, memory manager, system calls including fork(), and print “hello world” the hard way.
Yes, you do need to come to class and do all of the programming assignments!

Your grade will be based equally on:
- Project (programming assignments)
- Mid-term exam
- Final exam

Don’t waste these opportunities to convince me that you have *understood* the material!
Useful Links

Class web site
www.cs.pdx.edu/~walpole/class/cs510/fall2017/home.html
Find my website from the faculty listing on the department website. Follow teaching link to current classes
What to do Next?

Assignment 0
   Read the class web site
   Help me learn who you are:
      - send me an email with your photo

Assignment 1
   Download, install and experiment with Blitz
   Read the Blitz documents
   Due one week from today!
   See class web site for project assignments
Introduction to Operating Systems

1. What is an Operating System?
2. Review of OS-Related Hardware
What is an Operating System?

A concurrent *program* that controls the execution of application programs and implements an interface between them and the computer hardware.
Operating System Roles

Abstract machine

- Hide complex details of the underlying hardware
- Provide common API to applications and services
- Simplify application writing

The challenges:

- Deciding what to hide and what to expose
- Defining suitable abstractions
- Efficient mapping to hardware
Operating System Roles

Resource Manager
   Control access to shared resources
   Implement global allocation policies

The challenges:
   concurrency
   security
   performance
Early Operating Systems

Before operating systems, application writers had to program all device access directly:

- Load device command codes into device registers
- Handle initialization, recalibration, sensing, timing
- Understand physical characteristics and data layout
- Control motors
- Interpret return codes

I/O complicated application programs

Application programs were difficult to maintain and port
The device code could be written just once and then shared!
How to Access the Shared OS

 Applications
 Software in high and low level languages

 Operating System
 Software in high and low level languages

 Hardware
 CPU, Memory, Disk, Network, Display, Mouse etc
Operating System Structure

Applications

System Call Interface
Calls: fork(), exec(), read(), write(), kill(), getpid() etc

Operating System Kernel
Abstractions: Processes, Threads, Files, Sockets, etc

Hardware
CPU, Memory, Disk, Network, Display, Mouse etc
The Resource Manager Role

Allocating resources to applications
  time sharing resources
  space sharing resources

Making efficient use of limited resources
  improving utilization
  minimizing overhead
  improving throughput/good put

Protection via enforcement of allocations
## Resources to Allocate

<table>
<thead>
<tr>
<th></th>
<th>Time Share</th>
<th>Space Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Memory</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Disk</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Network</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Enforcement by OS

Can the OS protect itself from applications? Can it protect applications from each other? Can it prevent direct access to hardware? Should it prevent direct access to hardware?

The OS is just a program!
How can it do all this?
OS Needs Help from Hardware

The OS dilemma:

The OS is just a program!

*When it is not running, it can’t do anything!*

The OS’s goal is to run applications, not itself!

The OS needs help from the hardware in order to detect and prevent certain activities, and to enforce allocations.
Instruction sets define all that a CPU can do. They differ among CPU architectures, but all have load and store instructions to move data between memory and registers. Many instructions for comparing and combining values in registers.

Examine the Blitz instruction set. Similar to SUN SPARC instruction set.
Basic Anatomy of a CPU

Program Counter (PC)
Basic Anatomy of a CPU

Program Counter (PC)
Holds the memory address of the next instruction
Basic Anatomy of a CPU

Program Counter (PC)
  Holds the memory address of the next instruction

Instruction Register
Basic Anatomy of a CPU

Program Counter (PC)
- Holds the memory address of the next instruction

Instruction Register
- Holds the instruction currently being executed
Basic Anatomy of a CPU

Program Counter (PC)
   Holds the memory address of the next instruction

Instruction Register
   holds the instruction currently being executed

General Purpose Registers
Basic Anatomy of a CPU

Program Counter (PC)
Holds the memory address of the next instruction

Instruction Register
holds the instruction currently being executed

General Purpose Registers
hold variables and temporary results
Basic Anatomy of a CPU

Program Counter (PC)
  Holds the memory address of the next instruction

Instruction Register
  holds the instruction currently being executed

General Purpose Registers
  hold variables and temporary results

Arithmetic and Logic Unit (ALU)
Basic Anatomy of a CPU

Program Counter (PC)
Holds the memory address of the next instruction

Instruction Register
holds the instruction currently being executed

General Purpose Registers
hold variables and temporary results

Arithmetic and Logic Unit (ALU)
performs arithmetic functions and logic operations
Basic Anatomy of a CPU

Stack Pointer (SP)
Basic Anatomy of a CPU

Stack Pointer (SP)

holds memory address of a stack top
one frame for parameters & local variables of each active procedure
Basic Anatomy of a CPU

Stack Pointer (SP)
holds memory address of a stack top
one frame for parameters & local variables of each active procedure

Status Register
Basic Anatomy of a CPU

Stack Pointer (SP)
- holds memory address of a stack top
- one frame for parameters & local variables of each active procedure

Status Register
- A word full of control flags/bits
- Includes the mode bit to determine whether the CPU will execute privileged instructions
Program Execution

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

At a fundamental level, this is all a CPU does -
- It does not know which program it is executing!
Fetch/Decode/Execute Cycle

CPU

- PC
- IR
- Reg. 1
- ... (other registers)
- ALU

Memory
The OS is Just a Program!

The OS is just a sequence of instructions that the CPU will fetch/decode/execute.

How can the OS cause application programs to run?

How can applications cause the OS to run?
How Can an OS Run Applications?
How Can an OS Run Applications?

The OS must load the address of the application’s starting instruction into the PC

Example:
- computer boots and begins running the OS
- OS code must get into memory somehow
- fetch/decode/execute OS instructions
- OS requests user input to identify application program/file
- OS loads application (executable file) into memory
- OS loads the address of the app’s first instruction into the PC
- CPU fetches/decodes/executes the application’s instructions
Will the OS Ever Run Again?

How can the OS guarantee that it will run again?
How Can the OS Regain Control?

What if an application doesn’t call the OS and instead just hogs the CPU?

- OS needs something to interrupt the CPU and load OS instructions again
- Interrupts can be generated from a timer device
- OS must register a future timer interrupt before handing control of the CPU over to an application
- When the timer interrupt goes off the hardware starts running the OS at a pre-specified location called an interrupt handler
What is an Interrupt?

Interrupt:
Hardware event that loads an address of an instruction into the PC

Interrupt handler:
An instruction sequence that has been associated with an interrupt
Address of first instruction is registered with the interrupt hardware by filling in an interrupt vector
Interrupts

Interrupts are a form of event-based programming
    Also known as reactive programming

Interrupt handlers are invoked in response to hardware events
    Invocation order is not known a priori
    Their execution is interleaved with existing execution sequences, unless prevented by disabling interrupts
Can Applications Cheat the Timer?

Can the application disable the future timer interrupt so that the OS can not take control back from it?
Can Applications Cheat the Timer?

Disabling interrupts must be a privileged instruction that is not executable by applications.

The CPU knows whether or not to execute privileged instructions based on the value of the mode bit in the status register.

Privileged instructions are only executed if the mode bit is set - attempted execution in non-privileged mode generally causes an interrupt (trap) to occur.
Can Applications Set the Mode Bit?

How is the mode bit set?
   Via a privileged instruction?
   If so, is this instruction privileged?
Can Applications Set the Mode Bit?

The mode bit is set via interrupts, traps, faults, and exceptions.

These are all hardware events that cause the address of a previously registered instruction to be loaded into the PC. This instruction is the start of an interrupt/trap/fault handler that is part of the OS!
Are There Other Ways to Cheat?

What stops the running application from modifying the interrupt vector, or the handlers associated with various interrupts, traps and faults?

- eg. modifying the timer interrupt handler to jump control back to the application?
What Stops Applications From Modifying the Interrupt Vector?
What Stops Applications From Modifying the Interrupt Vector?

Doing so requires privileged instructions!
What Stops Applications From Modifying the OS?

i.e., modifying the interrupt, trap, fault handler instructions in memory?
What Stops Applications From Modifying the OS?

Memory protection!

Certain areas of memory are off-limits unless the correct values are loaded into memory protection registers

... and accessing those registers requires use of privileged instructions

... they can only be changed if the mode bit is set
How Can the OS Maintain Control?

Why must the OS fill in the interrupt vectors before handing control over to applications?

Why must it register a future timer interrupt?

Why must it set memory protections?

Why must it clear the mode bit?

What if it forgets to do one of the above?
How Can Applications Invoke the OS?

Why not just set PC to an OS instruction address and transfer control that way?

How would the mode bit get set?
How Can Applications Invoke the OS?

Special trap instruction causes a kind of interrupt
- changes PC to point to a predetermined OS entry point instruction
- simultaneously sets the mode bit
- CPU is now running in privileged mode

Application calls a library procedure that includes the appropriate trap instruction
fetch/decode/execute cycle begins at a pre-specified OS entry point called a system call handler
Are Traps Interrupts?

Traps, like interrupts, are hardware events.
But traps are synchronous whereas interrupts are asynchronous.

Traps are caused by the executing program rather than by a device that is external to the CPU.
## Interrupts, Traps, Faults and Exceptions

<table>
<thead>
<tr>
<th></th>
<th>Synchronous?</th>
<th>Intentional?</th>
<th>Error?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Trap</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Fault</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Exception</td>
<td>Yes</td>
<td>Maybe</td>
<td>Sometimes</td>
</tr>
</tbody>
</table>
Review Questions

Why do we need a timer device?
Why do we need an interrupt mechanism?
Why do we need two instruction types and a mode bit?
Why are system calls different to procedure calls?
How are system calls different to interrupts?
Why is memory protection necessary?
What to do before next class

Assignment 0 – read class web page
Email a picture of yourself to me
Start project 1 – Introduction to BLITZ