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

Understand the basic concepts of operating systems

*designing & building* operating systems, not *using* them!

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

Active participation in class discussions
No cheating!
The contract:
  You try to learn!
  I try to teach you!
Grading

Exams
  Mid-term - 25%
  Final - 25%

Coursework
  Project – 50%
Text books

“The BLITZ System” by Harry Porter

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

You will read, understand and write operating system code!
We will use the BLITZ system, written by Harry Porter
  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 (cs.pdx.edu), plus Macs
  and x86/Linux
Useful Links

Class web site
www.cs.pdx.edu/~walpole/class/cs399/winter2016/home.html
Find my website from the faculty listing on the department website. Follow teaching link to current classes

Class mailing list
?
What to do Next?

Project 0
Read the class web site

Project 1
Due next week!
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 program ... that controls the execution of application programs and implements an interface between the user of a computer and the computer hardware”

Runs on PCs, workstations, servers, smart phones, routers, embedded systems, etc
Operating System Roles

Abstract Machine
- Hides complex details of the underlying hardware
- Provides 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
The Abstract Machine Role

Without operating systems, application writers would have 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

... Application programming would be complicated
Applications would be difficult to maintain, upgrade and port
  This OS code could be written just once and then shared!
Applications

Operating System

CPUs  Memory  Networks

Video Card  Monitor  Disks  Printers
Applications

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

Operating System

Device Mgmt

File System

Protection

Network Comm.

Process Mgmt

Security

CPUs

Memory

Networks

Video Card

Monitor

Disks

Printers
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

Protecting applications from each other
Resources to Allocate

Time sharing the CPU
Space sharing the memory
Space sharing the disk
Time sharing the network
Time sharing the disk?
Space sharing the CPU?
Time sharing the memory?
Problems Solved by OS

Time sharing the CPU among applications
Space sharing the memory among applications
Space sharing the disk among users
Time sharing access to the network
What about space sharing CPU, time sharing disk, etc?
More Problems Solved by OS

Protection of applications from each other, of user data from other users and of I/O devices

Protection of the OS itself!

Prevention of direct access to hardware, where this would cause problems

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

The OS is just a program!
When it is not running, it can’t do anything!
Its goal is to run applications, not itself!

The OS needs help from the hardware in order to detect and prevent certain activities, and to maintain control.
Brief Review of Hardware

Instruction sets define all that a CPU can do, and differ among CPU architectures. 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, which is similar to a SUN SPARC instruction set, for a concrete example.
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 (IR)
Basic Anatomy of a CPU

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

Instruction Register (IR)
   Holds the instruction currently being executed
Basic Anatomy of a CPU

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

Instruction Register (IR)
  holds the instruction currently being executed

General Registers (Reg. 1..n)
Basic Anatomy of a CPU

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

Instruction Register (IR)
   holds the instruction currently being executed

General Registers (Reg. 1..n)
   hold variables and temporary results
Basic Anatomy of a CPU

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

Instruction Register (IR)
   holds the instruction currently being executed

General Registers (Reg. 1..n)
   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 (IR)
   holds the instruction currently being executed
General Registers (Reg. 1..n)
   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 with a frame for each active procedure’s parameters & local variables
Basic Anatomy of a CPU

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

Processor Status Word (PSW)
Basic Anatomy of a CPU

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

Processor Status Word (PSW)
- contains various control bits including the mode bit which determines whether privileged instructions can be executed at this time
Basic Anatomy of a CPU

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

Processor Status Word (PSW)
contains various control bits including the mode bit which determines whether privileged instructions can be executed
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, regardless of which program it is executing
Fetch/Decode/Execute Cycle

CPU

PC  IR

Reg. 1  ...

ALU

Reg. n

Memory
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
}
While (1) {
    Fetch instruction from memory
    Execute instruction
    (Get other operands if necessary)
    Store result
}
Fetch/Decode/Execute Cycle

CPU

PC
IR

ALU

Reg. 1
...
Reg. n

Memory

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

41
Fetch/Decode/Execute Cycle

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

CPU

Memory

ALU

PC  IR

Reg. 1

...  ...

Reg. n
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
}

CPU

Memory

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
}

CPU

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?

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
The OS is Just a Program!

How does the OS ever get to run again?
How can the OS switch the CPU to run a new application (and later resume the first one)?
How can the OS maintain control of what the application does when the OS is not running?
In what ways can application try to seize control indefinitely (ie. cheat)?
How can the OS prevent such cheating?
How Can the OS Regain Control?

What if an application doesn’t call the OS and instead just hogs the CPU?
- OS needs *interrupts* 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
- The interrupt handler is part of the OS program
- The address of the interrupt handler’s first instruction is placed in the PC by the h/w
Can the Application Cheat?

Can the application disable the future timer interrupt so that the OS can not take control back from it?
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 PSW.
Privileged instructions can only be executed when the mode bit is set
- eg. disabling interrupts, setting the mode bit!
- attempted execution in non-privileged mode generally causes an interrupt (trap) to occur
Are There Other Ways to Cheat?

What stops the running application from modifying the OS?
- eg. modifying the timer interrupt handler to jump control back to the application?
What Stops Applications From Modifying the OS?

Memory protection!
Memory protection instructions must be privileged
- i.e., they can only be executed with the mode bit set ...

Why must the OS clear the mode bit before it hands control to an application?
How Can Applications Invoke the OS?

Why not just set PC to an OS instruction address and transfer control that way?
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. I.e., traps are caused by the executing program rather than a device external to the CPU.
Switching to a New Application?

To suspend execution of an application the OS must run!

After that, simply
- capture the application’s memory state and processor state
- preserve all the memory values of this application
- copy values of all CPU registers into a data structure which is saved in memory
- restarting the application from the same point just requires reloading the register values
Recap

Why do we need a timer device?
Why do we need an interrupt mechanism?
Why do we need privileged instructions?
Why are system calls different to procedure calls?
How are system calls different to interrupts?
Why is memory protection necessary?
How can the OS switch among applications?
What to do before next class

Reading for today’s class
Reading for class 2
Assignment 0 – read class web page and join class email list
Start project 1 – Introduction to BLITZ