CSE 513
Introduction to Operating Systems

Class 1 - History and Intro to OS-related Hardware and Software

Jonathan Walpole
Dept. of Comp. Sci. and Eng.
Oregon Health and Science University
About the Instructor & TA

- **Instructor - Jonathan Walpole**
  - Professor and Director of Systems Software Lab - OGI (1998 -)

- **TA - Chris Chambers**
  - Ph.D. Student - Systems Software Lab, (2002-)
  - Research Interests: Distributed Games
About CSE 513

- **Goals of the class**
  - core class primarily for non-systems students
  - may be your first and last exposure to OS
  - understand the basic concepts
  - gain some practical experience

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

- **Exams**
  - Mid-term - 20%
  - Final - 30%

- **Coursework**
  - project - 30%

- **Participation**
  - in-class discussions - 20%
Books

Modern Operating Systems
A. Tannenbaum
2nd Edition

Understanding the Linux Kernel
Daniel Bovet & Marco Cesati
2nd Edition
The Project

- We will be using Linux on real systems ... ugh!!
- The computer labs are equipped with Linux that can be downloaded fresh when you turn them on
  + You will be modifying a real O.S.
  + It's the real deal!
  - More complicated than emulated systems
  - Mistakes you make can cause catastrophic failure → the blue screen of death will look friendly

- TA
  - Chris Chambers will be the primary contact for the project
Administrative

- **Course web site**
  - http://www.cse.ogi.edu/class/cse513/

- **Assignment 0**
  - mail me a brief description of who you are, and a picture of yourself ... so I know who to assign credit to for your in-class discussion!

- **Assignment 1**
  - due next week! See class web site for project assignments.
Lecture 1 - Introduction
Lecture overview

- What is an Operating System?
- A review of OS-related hardware
- History of operating systems
- Types of operating systems
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

CPU Memory Network

Video Card Monitor Disk Printer

Hardware
Providing Abstraction via System Calls

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

- Application
- Operating System
- Device Mgmt
- Protection
- File System
- Network Comm.
- Process Mgmt
- Security
- CPU
- Memory
- Network
- Video Card
- Monitor
- Disk
- Printer

Hardware
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!
Lecture overview

- What is an Operating System?
- A review of OS-related hardware
- History of operating systems
- Types of operating systems
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 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
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

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

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 applications 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

Fetch unit → Decode unit → Execute unit

Fetch unit → Decode unit

Holding buffer

Execute unit

Execute unit

Execute unit
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 memory hierarchy

- **2GHz processor** → 0.5 ns
- **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 than you want, less than magnetic disk!
## Terminology review - metric units

<table>
<thead>
<tr>
<th>Exp.</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>1,000</td>
<td>Kilo</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>0.000001</td>
<td>micro</td>
<td>$10^6$</td>
<td>1,000,000</td>
<td>Mega</td>
</tr>
<tr>
<td>$10^{-9}$</td>
<td>0.00000001</td>
<td>nano</td>
<td>$10^9$</td>
<td>1,000,000,000</td>
<td>Giga</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>Tera</td>
</tr>
<tr>
<td>$10^{-15}$</td>
<td>0.000000000000001</td>
<td>femto</td>
<td>$10^{15}$</td>
<td>1,000,000,000,000,000,000</td>
<td>Peta</td>
</tr>
<tr>
<td>$10^{-18}$</td>
<td>0.000000000000000001</td>
<td>atto</td>
<td>$10^{18}$</td>
<td>1,000,000,000,000,000,000,000,000</td>
<td>Exa</td>
</tr>
<tr>
<td>$10^{-21}$</td>
<td>0.00000000000000000001</td>
<td>zepto</td>
<td>$10^{21}$</td>
<td>1,000,000,000,000,000,000,000,000,000,000</td>
<td>Zetta</td>
</tr>
<tr>
<td>$10^{-24}$</td>
<td>0.0000000000000000000001</td>
<td>yocto</td>
<td>$10^{24}$</td>
<td>1,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000</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!
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?
Memory protection and relocation …

- **Memory protection**
  - virtual vs physical addresses
    - address range in each application starts at 0
  - “base register” used to convert each virtual address to a physical address before main memory is accessed
  - address is compared to a “limit register” to keep memory references within bounds

- **Relocation**
  - by changing the base register value

- **Paged virtual memory**
  - same basic concept, but more powerful (and complex)
Base & Limit Registers (single & multiple)

(a) 0xFFFFF

Limit

User program and data

User program and data

Base

Operating System

(b) Registers when program 2 is running

Limit-2

Base-2

Limit-2

Base-2

Limit-1

Base-1

User-2 data

User-1 data

User program

Operating System
Virtual memory and MMUs

- Memory management unit (MMU)
  - hardware provided equivalent of base registers
  - at the granularity of “pages” of memory, say 2kB, i.e., lots of them!
  - supports relocation at page granularity
  - applications need not occupy contiguous physical memory

- Memory protection
  - limit registers don’t work in this context
  - per-page and per-application protection registers

- Relocation and protection occur at CPU speeds!
What about I/O devices?

A simplified view of a computer system
Structure of a large Pentium system
What about I/O devices?

A simplified view of a computer system
How do programs interact with devices?

- **Devices vs device controllers vs device drivers**
  - device drivers are part of the OS
  - programs call the OS which calls the device driver

- **Device drivers interact with device controllers**
  - either using special IO instructions
  - or by reading/writing controller registers that appear as memory locations

- **Why protect access to devices by accessing them indirectly via the OS?**
How do devices interact with programs?

- **Interrupts**

![Diagram of device interaction with programs](image)

1. CPU
2. Disk controller
3. Interrupt controller
4. Disk drive

1. **Current instruction**
2. **Next instruction**
3. **Return**

1. **Interrupt**
2. **Dispatch to handler**
3. **Return**
Different types of interrupts

- **Timer interrupts**
  - Allows OS to maintain control
  - One way to keep track of time

- **I/O interrupts**
  - Keyboard, mouse, disks, etc...

- **Hardware failures**

- **Program generated (traps)**
  - Programming errors: seg. faults, divide by zero, etc.
  - System calls like read(), write(), gettimeofday()
Timer interrupts

- **OS can ask timer device to interrupt after a specified time period has elapse**

- **Interrupt invokes timer interrupt handler which invokes OS “scheduler”**

- **OS can take the opportunity to save the current application and restore a different one**
  - context switch
Why use Traps for system calls?

- The Operating System is just a program!
- It must have the privilege to manipulate the hardware
  - set base and limit registers for memory protection
  - access devices
  - set and clear mode bit to enable privilege
- If user programs execute with the mode bit clear, and do not have privilege to set it, how can they invoke the OS so that it can run with the mode bit set?
  - That’s what traps do ... set the mode bit and begin execution at a specific point in memory (in the OS!)
System calls

- System calls are the mechanism by which programs communicate with the O.S.
- Implemented via a TRAP instruction
- Example UNIX system calls:
  open(), read(), write(), close()
  kill(), signal()
  fork(), wait(), exec(), getpid()
  link(), unlink(), mount(), chdir()
  setuid(), getuid(), chown()
The inner workings of a system call

User-level code

```c
Process usercode
{
    ...
    read (file, buffer, n);
    ...
}
```

Library code

```c
Procedure read(file, buff, n)
{
    ...
    read(file, buff, n)
    ...
}

_read:
    LOAD  r1, @SP+2
    LOAD  r2, @SP+4
    LOAD  r3, @SP+6
    TRAP  Read_Call
```
Steps in making a system call

1. Increment SP
2. Push &buffer
3. Push fd
4. Call read
5. Put code for read in register
6. Dispatch
0. Address 0xFFFFFFFF

User space

Kernel space (Operating system)

Library procedure read

User program calling read

Return to caller

Sys call handler

OGI SCHOOL OF SCIENCE & ENGINEERING
OREGON HEALTH & SCIENCE UNIVERSITY
What about disks and file storage?

Structure of a disk drive

Read/write head (1 per surface)

Direction of arm motion

Surface 7
Surface 6
Surface 5
Surface 4
Surface 3
Surface 2
Surface 1
Surface 0

Structure of a disk drive
Disks and file storage

- **Manipulating the disk device is complicated**
  - hide some of the complexity behind disk controller, disk device driver

- **Disk blocks are not a very user-friendly abstraction for storage**
  - contiguous allocation may be difficult for large data items
  - how do you manage administrative information?

- **One application should not (automatically) be able to access another application’s storage**
  - OS needs to provide a “file system”
File Systems

File system - an abstraction above disk blocks
What about networks?

- Network interfaces are just another kind of shared device/resource
- Need to hide complexity
  - send and receive primitives, packets, interrupts etc
  - protocol layers
- Need to protect the device
  - access via the OS
- Need to allocate resources fairly
  - packet scheduling
Lecture overview

- What is an Operating System?
- A review of OS-related hardware
- History of operating systems
- Types of operating systems
Brief history of OSs

- Vacuum tubes and plugboards (pre-1955)
  - “Man programs machine”
  - Machine → calculating hardware capable of only number crunching
  - Running programs
    - Reserve machine time
    - Program it in machine language by physically connecting wires!
    - Results printed on or punched in paper
  - What was the OS?
Brief history of OSs

- Transistors and batch systems (1955–1965)
  - Human time really slow compared to processing capability, hence staffs of professional operators
  - Batch systems → take a large number of jobs together, run them sequentially, and output data to a tape for printing
Brief history of OSs

- Transistors and batch systems (1955-1965)
  - A closer look at the batch system “OS”
    - Load the next program into memory
    - Run it to its completion
    - Output to tape when writes occur
    - When done, remove job

- What’s good?

- What’s bad?

- What is the OS?
Brief history of OSs

- ICs and multiprogramming – the beginnings of computing as we know it (1965-1980)
  - CPU “speed” vs. tape access tremendously different
  - Great idea → Multiprogramming
    - load all jobs in memory
    - run one job until it had to do I/O
    - switch to another job
    - repeat forever
    - use software to do the switching

- Another great idea → Time sharing systems
  - build terminals connected directly to the mainframe
  - run each job for a short time slice, rapidly switching among jobs
  - supports interactive jobs doing lots of I/O
Brief history of OSs

- ICs and multiprogramming – the beginnings of computing as we know it (1965-1980)

  - Requirements to support multiprogramming and time-sharing
    - Ability to “save” a running task (MT)
    - Ability to manage devices
    - Ability to partition memory among processes
    - Ability to protect memory
    - Ability to prioritize and select tasks to run (in non FIFO order)

- What is the OS?
Brief history of OSs

- Personal computers – (1980-present)
  - No sharing
  - Lots of different devices and configurations
  - GUIs and interactivity

- Operating Systems
  - More user-centric
    - Non-preemptable operating systems (e.g. Macintosh, Windows)
  - Still required techniques from multiprogramming and multitasking
Brief history of OSs

- Personal computers - (1980-present)
  - Reintroduced sharing - of data and peripheral devices
    - Networking and Distributed Systems enable world wide sharing of information and other resources
Computing technologies come full circle...

- **Distributed heterogeneous systems**
  - Lots of heterogeneity (hardware/data/programs)
  - Massively scalably
  - Maintenance nightmare!

- **Current trends**
  - IT out-sourcing
  - Centralized compute servers
  - Professional system management staff
Lecture overview

- What is an Operating System?
- A review of OS-related hardware
- History of operating systems
- Types of operating systems
The Operating System Zoo

- Mainframe operating systems
- Server operating systems
- Multiprocessor operating systems
- Personal computer operating systems
- Real-time operating systems
- Embedded operating systems
- Smart card operating systems
Ways of structuring operating systems

- **Monolithic**
  - All operating system functions are combined into a single entity (address space & protection domain)

- **Microkernel**
  - Only key functionality in kernel. All other functionality in user space modules that communicate by message passing (no shared address space or protection domain)

- **Virtual machines**
  - Give each user the illusion that they have their own machine
“Structure” in a monolithic system
Structure in a strictly layered system

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>The operator</td>
</tr>
<tr>
<td>4</td>
<td>User programs</td>
</tr>
<tr>
<td>3</td>
<td>Input/output management</td>
</tr>
<tr>
<td>2</td>
<td>Operator-process communication</td>
</tr>
<tr>
<td>1</td>
<td>Memory and drum management</td>
</tr>
<tr>
<td>0</td>
<td>Processor allocation and multiprogramming</td>
</tr>
</tbody>
</table>

Structure of the THE operating system
Structure with virtual machines

Virtual 370s

I/O instructions here

Trap here

CMS

CMS

CMS

System calls here

Trap here

VM/370

370 Bare hardware

Structure of VM/370 with CMS
Client-server structure in a micro-kernel OS

The client-server model

Client process | Client process | Process server | Terminal server | ... | File server | Memory server

Microkernel

User mode

Kernel mode

Client obtains service by sending messages to server processes
Distributed client-server structure

The client-server model in a distributed system
Summary

- Introduction
- OS related hardware
- History
- Operating system architectures
What to do before next week's class

- Reading for today's class - pages 1-70
- Reading for next week's class - pages 71-97 and pages 132-152
- Assignment 0 - send me a photo and short bio
- Project assignment 1 - compile and run Linux OS
Review
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
How does the OS solve these problems?

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