1-1. Introduction
Assignment!

- Read sections 1.1 - 1.8 in Stallings

An Operating System…

enables this…

to become…

THIS!!
What **IS** an operating system?

- A program that
  - Controls the resources of a computer
  - Provides an interface to user applications and their developers.

- Advantages over working with the bare machine:
  - Programmer productivity
  - Modularity, multiprogramming
  - Reuse of functionality
  - Correctness, security, fair access (scheduling)

- Disadvantages?

---

Recall (very basic) computer architecture

```
CPU
- PC
- IR
- MAR
- MBR
- I/O AR
- I/O BR

System Bus

Main Memory
- Instructions
- Data

I/O Module
- Buffers
```

PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register
OS and the Machine Architecture

• Users typically do not interact with the OS directly
• Rather, OS and hardware work together to support applications (a.k.a. user-level programs), which the user sees
• A program in execution is called a process.
• A major responsibility of the OS is to enable application processes to execute on the processor while preserving the integrity of the system.
• Some parts of the OS are tightly integrated with the system architecture; others are totally independent

A fundamental concept…

• A modern OS does not monitor or interpret the execution of individual instructions in a user process
  – Instead, the OS actually relinquishes control of the processor to user processes
  – When your process runs, it runs directly on the hardware, “The processor is yours. Use it!”
• Potential Problem?
• Solution?
• Hence, this is called…
Another fundamental concept…

• When your process reads and writes to computer memory, the addresses it uses are **not** physical memory addresses.

• Rather, the OS and architecture collaborate to “fake out” your process and make it “think” it is referencing physical memory.

• This is called…

• **Address space**: range of addresses seen by process

• This approach has huge benefits and only minor drawbacks.

• A **LOT** of OS functionality is related to memory mgmt.

User Mode and Kernel Mode

• Processors have different modes (or rings) of execution
  – E.g., x86 has 4 modes (levels), 0 is most privileged
  – Many systems use only 2 (or sometimes 3)

• Much of the OS code executes in kernel mode (mode 0), while user processes execute in user mode (mode 3).

• Certain actions (executing sensitive instructions, accessing key registers) can be performed only if the processor is in privileged (kernel) mode.

  **What if the process is not executing in kernel mode?**
Operating System **Kernel** (note spelling, two e’s)

• Typically, parts of the OS that execute in privileged mode
  – Manage processes/threads, memory, I/O, etc.
  – When we use the term OS “internals,” we usually mean the kernel (nuts and bolts of the OS)

• What is not in the kernel?
  – Often: libraries, user interface software, middleware, system call “stubs,” linkers, and many others
  – Sometimes, even “major” parts of OS for process management, memory management, I/O, networking, etc.

Abstract view of a system
A third fundamental concept...

• In most cases, kernel code runs as an extension of a user program, by way of system calls, which look like function calls, but involve a “trap” to the kernel.

Security Issue?

• Even though it is running in kernel mode, it is still your process.
• If your application process can “run around” in the kernel, how is the OS protected against a misbehaving process?
Summary: Limited Direct Execution

• When executing, your process “owns” the processor
• The instructions comprising your process execute directly on the hardware (no middleman)
• Limits:
  – Runs in virtual memory, can’t touch other processes
  – When executing at user level
    • Clock interrupts enable OS to regain control periodically
    • Exceptions will give control back to OS if the process does something not allowed
  – System calls: enable the process to run in kernel mode to access a specific set of OS services, always relinquishing processor cleanly. (How do we know it will do so?)

Major OS Functions

• Process and thread management
  – process control: creation, termination, suspension, resumption, scheduling, resource allocation, etc.

• Memory management
  – allocation/deallocation of computer memory, implementation of virtual memory, demand paging

• Interprocess (or interthread) communication
  – signals, pipes, messages, events, semaphores, shared memory, etc.
Major OS Functions (cont.)

• I/O and device drivers
  – character and block device management, handling
    interrupts, direct memory access (DMA), etc.

• File management
  – (virtual) file system implementation, memory-resident data
    structures, buffer cache, device files, etc.

• Network management
  – sockets or equivalent, TCP/IP protocols, etc.

Big vs. Small

• One of the major historical OS issues is whether
  systems should be designed around a:

  • **Monolithic kernel**: kernel provides most OS services

  or

  • **Microkernel**: kernel provides as little as possible;
    user-level **servers** provides most OS services!
Monolithic Kernel vs. Microkernel

Monolithic Kernel

- All of the OS shares a common, protected address space, which executes in privileged mode
- Kernel performs all major functions of the OS discussed earlier
- System calls are made by trapping to kernel, which performs work on behalf of the process
- Some parts of the kernel also run asynchronously to events related to timers and peripheral devices.
- Canonical example?
Microkernel-Based Systems

• Very small kernel provides only the most basic services to support processes and threads
  – Low-level process management, memory management, interprocess communication, interrupt handling
• Servers (often running at user-level) provide all other OS functions
  – full process management, virtual memory management and paging, file service, I/O, networking, system calls and system emulation
  – user process sends message to server, which does work and returns results
• Canonical Example?

Impact on Modern Operating Systems?

• Unix/Linux - monolithic kernel
• Mach – microkernel OS developed at Carnegie Mellon University in 1980s
• Windows (NT) – basis of current Windows versions
  – OS developed “from scratch” in early 1990s
  – Heavily influenced initially by microkernel concepts
  – Has evolved toward monolithic kernel over the years
• Mac OS X
  – Combines Mach microkernel with BSD Unix, all in single address space!
### Key Operating System Topics…

<table>
<thead>
<tr>
<th>kernel</th>
<th>interrupt</th>
<th>supervisor mode</th>
<th>kernel mode</th>
<th>user-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>user mode</td>
<td>privileged mode</td>
<td>monolithic kernel</td>
<td>user interface</td>
<td>microkernel</td>
</tr>
<tr>
<td>middleware</td>
<td>program</td>
<td>process</td>
<td>thread</td>
<td>virtual memory</td>
</tr>
<tr>
<td>page table</td>
<td>system call</td>
<td>scheduling</td>
<td>limited direct execution</td>
<td></td>
</tr>
</tbody>
</table>
### Key Operating Systems Topics…

<table>
<thead>
<tr>
<th>kernel</th>
<th>interrupt</th>
<th>supervisor mode</th>
<th>context switching</th>
<th>kernel-level</th>
<th>virtualization</th>
<th>device driver</th>
<th>character device</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel mode</td>
<td>user-level</td>
<td>kernel-level</td>
<td>virtualization</td>
<td>device driver</td>
<td>character device</td>
<td>device driver</td>
<td></td>
</tr>
<tr>
<td>user mode</td>
<td>privileged mode</td>
<td>block device</td>
<td>plain file</td>
<td>device file</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>monolithic kernel</td>
<td>middleware</td>
<td>file system</td>
<td>address space</td>
<td>virtual file system</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>microkernel</td>
<td>program</td>
<td>virtual address</td>
<td>file system</td>
<td>address space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>thread</td>
<td>virtual address</td>
<td>processor context</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>virtual memory</td>
<td>page table</td>
<td>virtual address</td>
<td>processor context</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>system call</td>
<td>scheduling</td>
<td>page</td>
<td>physical memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>limited direct execution</td>
<td></td>
<td>demand paging</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>page replacement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>page frame</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CONFUSED YET?!?**

---

CSE 410  1.1 Introduction
Timeout!

• A convenient feature of Computer Science:
  – Since we made up the whole field, … (Literally. It’s virtual!)
  – We can make up names for anything we like.
  – We can call the same thing by different names…
  – Or, apply the same name to different things.
  – This helps keep us one step ahead of “normal” people.

Basically, we’re geeks.

The Sad Fact…

People who understand operating system internals are really geeks!

Welcome to core computer science!
In CSE 410…

- We will look “under the hood”
- Along the way, not only learn what all those terms mean, but see how they are implemented.
- It will not always be pretty, but we will…
- Discover how real operating systems work!

Main Components of the Course

(Note: subject to alterations…)

- Part 1: Introduction (Stallings, Ch. 1,2)
  - Review of computer architecture
  - Overview of operating systems, some history

- Part 2: Processes and Threads (Ch. 3, 4, 5, 9,10)
  - OS process management, context switching, system calls
  - Process address space implementation
  - User- vs. kernel-level threads, multi-threaded kernel
  - Brief overview of process/thread scheduling
  - Concurrency and synchronization
Main Components of the Course

• Part 3: Memory Management (Stallings, Ch. 7,8)
  – Virtual memory and demand paging (and a little history)
  – Address translation: page tables, TLBs
  – 32-bit vs. 64-bit addresses
  – Page replacement algorithms, optimizations

• Part 4: I/O and File systems (Stallings, Ch 11, 12)
  – Device files, drivers, kernel data structures
  – Kernel networking software
  – File system layout, buffer cache, virtual file system

• Part 5. Boot, loading and linking (Stallings, Ch 7A)
  – Linux boot process evolution, loadable modules
  – Executable formats, loader operation, DLLs

• Part 6. OS and limited footprints (Ch 13, handouts)
  – Embedded operating systems
  – Android internals

• Part 7. Virtualization & containers (Stallings, Ch 14)
  – Virtual Machines: history, challenges, approaches, impact
  – A brief look at how VMware did “the impossible”
  – Containers, Docker, etc.
  – Impact of the “rediscovery” of virtualization (The Cloud!)
Time permitting…

• Aspects of OS security (Stallings, Ch 15, handouts)
  – Traditional OS security
  – OS vulnerabilities
  – Recent countermeasures
  – Details of selected high-profile attacks

What is in…

• Understanding how real operating systems work requires looking at code.
• So, to better understand OS internals, we will
  – look at real OS code, using an old (and therefore relatively simple) version of Linux
  – dig into a few other resources that provide low-level details of OS functionality
• I will also draw on my own experience in industry and academic research
What is out...

- What this course is **not** about:
  - How to configure the Windows Registry (or anything else)
  - Walk-through of modern Linux, Windows, MacOS X code
  - Low-level user commands and shell scripts (CSE 325)
  - Programming using low-level system calls (CSE 325)
  - Theoretical concurrency control, deadlock, etc (Ch 6)
  - Theoretical comparisons of scheduling algorithms (much of Ch 9 and 10)
  - Database files and records (some of Ch 12)
  - Distributed systems and networking (other than distributed file systems and OS network I/O)

A note about writing

- Effective writing is critical to success:
  - in school, in work, in life.
- This course is as much about concepts as it is about code: eventually, you will **think** like an OS!
- On quizzes, homework sets, and exams, you will be expected to write concise, coherent essays that describe complex OS concepts.