Lecture 2: Architecture support for OS
Last time

- What is an OS?
- What roles does it play?
- Today: Historic evolution of Operating Systems (and computing!)
A brief history—Phase 0

- In the beginning, OS is just runtime libraries
  - A piece of code used/sharable by many programs
  - Abstraction: reuse magic to talk to physical devices
  - Avoid bugs

- User scheduled an exclusive time where they would use the machine

- User interface was switches and lights, eventually punched cards and tape
  - An interesting side effect: less bugs
Phase 1: Batch systems (1955-1970)

- Computers expensive; people cheap
  - Use computers efficiently – move people away from machine
- OS in this period became a program loader
  - Loads a job, runs it, then moves on to next
  - More efficient use of hardware but increasingly difficult to debug
    - Still less bugs 😊
Advances in OS in this period

- **SPOOLING/Multiprogramming**
  - Simultaneous Peripheral Operation On-Line (SPOOL)
    - Non-blocking tasks
    - Copy document to printer buffer so printer can work while CPU moves on to something else
  - Hardware provided memory support (protection and relocation)
  - Scheduling: let short jobs run first
  - OS must manage interactions between concurrent things

- **OS/360 from IBM**
  - first OS designed to run on a family of machines from small to large
Phase 1, problems

- Utilization is low (one job at a time)
- No protection between jobs
  - But one job at a time, so?
- Short jobs wait behind long jobs
- Coordinating concurrent activities
- People time is still being wasted
- Operating Systems didn’t really work
  - Birth of software engineering
Phase 2: 1970s – Time sharing, Unix, Persistence

- Computers and people are expensive
  - Help people be more productive

- Interactive time sharing: let many people use the same machine at the same time
  - CTSS/Multics projects at MIT
  - Corbato got Turing award for this idea

- Emergence of minicomputers
  - Terminals are cheap

- Persistence: Keep data online on fancy file systems
Unix appears

- Ken Thompson, who worked on MULTICS, wanted to use an old PDP-7 laying around in Bell labs
  - He and Dennis Richie built a system designed by programmers for programmers

- Originally in assembly. Rewritten in C
  - In their paper describing unix, they defend this decision!
  - However, this is a new and important advance: portable operating systems!

- Shared code with everyone (particularly universities)
  - Start of open source?
Unix (cont’d)

- Berkeley added support for virtual memory for the VAX
  - Unix BSD

- DARPA selected Unix as its networking platform in arpanet

- Unix became commercial
  - …which eventually lead Linus Torvald to develop Linux
Phase 3: 1980s -- PCs

- Computers are cheap, people expensive
  - Put a computer in each terminal
  - CP/M from DEC first personal computer OS (for 8080/85) processors
  - IBM needed software for their PCs, but CP/M was behind schedule
  - Approached Bill Gates to see if he can build one
  - Gates approached Seattle computer products, bought 86-DOS and created MS-DOS
  - Goal: finish quickly and run existing CP/M software
  - OS becomes subroutine library and command executive
Phase 4: Networked/distributed systems--1990s to now?

- It's all about connectivity
- Enables parallelism but performance is not goal
- Goal is communication/sharing
  - Requires high speed communication
  - We want to share data not hardware
- Networked applications drive everything
  - Web, email, messaging, social networks, …
New problems

- Large scale
  - Google file system, mapreduce, …
- Parallelism on the desktop (multicores)
- Heterogeneous systems, IoT
  - Real-time; energy efficiency
- Security and Privacy
Phase 5

- Computing evolving beyond networked systems
  - Cloud computing, IoT, Drones, Cyber-physical systems, computing everywhere

- Hardware accelerators, heterogeneous systems, end of Moore’s Law, Hardware democratization/Open source HW

- New workloads: AI, Blockchain, ...

- New generation?
  - But what is it?
    » ...and what problems will it bring?
Break – Alan Turing (1912-1954)

- Computer Scientist, mathematician, logician, cryptographer, ...
- Known for
  - Breaking Enigma
  - Turing machines
  - Turing test
  - Turing awards 😊
  - Many others…

- https://en.wikipedia.org/wiki/Alan_Turing
How is the OS structured?

- Or to rephrase, what is the operating system?
  - Some special program?
  - If so, is it running all the time?
    - But what if we have only one CPU?
  - How would it interact with the other programs? With the hardware?
Sleeping Beauty Model

- Answer: Sleeping beauty model
  - Technically known as *controlled direct execution*
  - OS runs in response to “events”; we support the switch in hardware
  - Only the OS can manipulate hardware or critical system state

- Most of the time the OS is sleeping
  - Good! Less overhead
  - Good! Applications are running directly on the hardware
Need some architecture Support for this model

- **Manipulating privileged machine state**
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access

- **Generating and handling “events”**
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap

- **Other stuff**
  - Synchronization, memory protection, …
Protected Instructions

- OS must have exclusive access to hardware and critical data structures

- Only the operating system can
  - Directly access I/O devices (disks, printers, etc.)
    - Security, fairness (why?)
  - Manipulate memory management state
    - Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - Kernel mode, interrupt level
  - Halt instruction (why?)
Privilege mode

- Hardware restricts privileged instructions to OS
  - HW must support (at least) two execution modes: OS (kernel) mode and user mode

- Mode kept in a status bit in a protected control register
  - User programs execute in user mode
  - OS executes in kernel mode (OS == “kernel”)
  - CPU checks mode bit when protected instruction executes
  - Attempts to execute in user mode trap to OS

How do we make sure OS gets privileged mode but not programs?
Protocol for Secure Switching

- When the machine boots, OS is running
  - OS is mapped into part of memory of every process

- Going from higher privilege to lower privilege
  - Easy: can directly modify the mode register to drop privilege

- But how do we escalate privilege?
  - Special instructions to change mode and switch to the OS
    - System calls (int 0x80, syscall, svc)
    - Saves context and invokes designated handler
      - You jump to the privileged code; you cannot execute your own
    - OS checks your syscall request and honors it only if safe
  - Or, some kind of event happens in the system
Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access

- Generating and handling “events”
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap

- Other stuff
  - Synchronization, memory protection, …
Review: Computer Organization

- Program Counter
- Instructions
- Fetch
- Execute
- Branch Address
- Select PC
- New PC
- CPU Instructions
  - Fetch
  - Exec
- Opcode
Events

- An event is an “unnatural” change in control flow
  - Events immediately stop current execution
  - Changes mode, context (machine state), or both

- The kernel defines a handler for each event type
  - Event handlers always execute in kernel mode
  - The specific types of events are defined by the machine

- Once the system is booted, OS is one big event handler
  - all entry to the kernel occurs as the result of an event
Handling events – Interrupt vector table

handleTimerInterrupt() {
  ...
}

handleDivideByZero() {
  ...
}

handleSystemCall() {
  ...
}
Categorizing Events

- Two kinds of events: synchronous and asynchronous
- Sync events are caused by executing instructions
  - Example?
- Async events are caused by an external event
  - Example?

CPU ticks

Asynchronous events

Synchronous events
Categorizing Events

- Two kinds of events: **synchronous** and **asynchronous**
  - Sync events are caused by executing instructions
  - Async events are caused by an external event

- Two reasons for events: **unexpected** and **deliberate**
  - Unexpected events are, well, unexpected
    - Example?
  - Deliberate events are scheduled by OS or application
    - Why would this be useful?
Categorizing Events

This gives us a convenient table:

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>interrupt</td>
<td>signal</td>
</tr>
</tbody>
</table>

Terms may be slightly different by OS and architecture
  - E.g., POSIX signals, asynch system traps, async or deferred procedure calls
Faults

- Hardware detects and reports “exceptional” conditions
  - Page fault, memory access violation (unaligned, permission, not mapped, bounds…), illegal instruction, divide by zero

- Upon exception, hardware “faults” (verb)
  - Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted
  - Invokes registered handler
Handling Faults

- Some faults are handled by “fixing” the exceptional condition and returning to the faulting context
  - Page faults cause the OS to place the missing page into memory
  - Fault handler resets PC of faulting context to re-execute instruction that caused the page fault
Handling Faults

- The kernel may handle unrecoverable faults by killing the user process
  - Program fault with no registered handler
  - Halt process, write process state to file, destroy process
  - In Unix, the default action for many signals (e.g., SIGSEGV)

- What about faults in the kernel?
  - Dereference NULL, divide by zero, undefined instruction
  - These faults considered fatal, operating system crashes
  - Unix panic, Windows “Blue screen of death”
    - Kernel is halted, state dumped to a core file, machine locked up
# Categorizing Events

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>interrupt</td>
<td>signal</td>
</tr>
</tbody>
</table>
System Calls

- For a user program to do something “privileged” (e.g., I/O) it must call an OS procedure
  - Known as crossing the protection boundary, or a protected procedure call

- Hardware provides a system call instruction that:
  - Causes an exception, which invokes a kernel handler
    - Passes a parameter determining the system routine to call
  - Saves caller state (PC, regs, mode) so it can be restored
    - Why save mode?
  - Returning from system call restores this state
System Call

User mode

Kernel mode

emacs: read()

Trap to kernel mode, save state

Trap handler

Find read handler

read() kernel routine

Restore state, return to user level, resume execution
Another view

Address Space

Kernel Stack

Kernel Code

User Stack

User Code

SP2

PC2

SP1

PC1

0xFFFFF

0xC0000000

0x00000000

1G

3G

0xFFFFFFFF

User Code

Kernel Code

Kernel Stack

User Stack

Address Space

SP2

PC2

SP1

PC1

0xFFFFF

0xC0000000

0x00000000

1G

3G

Another view

Address Space

Kernel Stack

Kernel Code

User Stack

User Code

SP2

PC2

SP1

PC1

0xFFFFF

0xC0000000

0x00000000

1G

3G

Another view

Address Space

Kernel Stack

Kernel Code

User Stack

User Code

SP2

PC2

SP1

PC1

0xFFFFF

0xC0000000

0x00000000

1G

3G

Another view

Address Space

Kernel Stack

Kernel Code

User Stack

User Code

SP2

PC2

SP1

PC1

0xFFFFF

0xC0000000

0x00000000

1G

3G

Another view

Address Space

Kernel Stack

Kernel Code

User Stack

User Code

SP2

PC2

SP1

PC1

0xFFFFF

0xC0000000

0x00000000

1G

3G

Another view

Address Space

Kernel Stack

Kernel Code

User Stack

User Code

SP2

PC2

SP1

PC1

0xFFFFF

0xC0000000

0x00000000

1G

3G
System Call Questions

- There are hundreds of syscalls. How do we let the kernel know which one we intend to invoke?
  - Before issuing `int $0x80` or `sysenter`, set `%eax/%rax` with the syscall number

- System calls are like function calls, but how to pass parameters?
  - Just like calling convention in syscalls, typically passed through `%ebx, %ecx, %edx, %esi, %edi, %ebp`
More questions

- How to reference kernel objects (e.g., files, sockets)?
  - Naming problem – an integer mapped to a unique object
    - int fd = open(“file”); read(fd, buffer);
  - Why can’t we reference the kernel objects by memory address?
**System calls in xv6**

- Look at trap.h and trap.c
  - Interrupt handlers are initialized in two arrays (idt and vectors)
    - Tvinit() function does the initialization
  - Syscalls have a single trap handler (T_SYSCALL, 64)
  - Trap() handles all exceptions, including system calls
    - If the exception is a system call, it calls syscall()

- Keep digging from there to understand how system calls are supported
  - You will be adding a new system call in Lab 1
## Categorizing Events

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>interrupt</td>
<td>software interrupt</td>
</tr>
</tbody>
</table>

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers
Timer

- The key to a timesharing OS
- The fallback mechanism by which the OS reclaims control
  - Timer is set to generate an interrupt after a period of time
    - Setting timer is a privileged instruction
    - When timer expires, generates an interrupt
      - Handled by the OS, forcing a switch from the user program
    - Basis for OS scheduler (*more later…*)

- Also used for time-based functions (e.g., `sleep()`)
I/O using Interrupts

- Interrupts are the basis for asynchronous I/O
  - OS initiates I/O
  - Device operates independently of rest of machine
  - Device sends an interrupt signal to CPU when done
  - OS maintains a vector table containing a list of addresses of kernel routines to handle various events
  - CPU looks up kernel address indexed by interrupt number, context switches to routine
**I/O Example**

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet (reads device registers to find packet in memory)
6. Upon completion, restores saved state from stack
Interrupt Questions

- Interrupts halt the execution of a process and transfer control (execution) to the operating system
  - Can the OS be interrupted? (Consider why there might be different interrupt levels)

- Interrupts are used by devices to have the OS do stuff
  - What is an alternative approach to using interrupts?
  - What are the drawbacks of that approach?
Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access

- Generating and handling “events”
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap

- Other stuff
  - Synchronization, memory protection, …