Administrivia

• Project 0
  ♦ We are still setting things up and will make everything work soon.
  ♦ Due 4/14 11:59pm, done individually

• Homework #1
  ♦ Due 4/14 11:59pm

• Midterm
  ♦ Two votes (out of 22) NO now => we can’t do 5-7pm
  ♦ Let’s set it to May 2nd 2-4pm (assuming no one has class on weekends)
  ♦ *If you have absolute dire circumstances, email me before tomorrow!*
Administrivia

• Project groups
  ♦ We will use a Google form to collect group members
  ♦ Just need one submission per group
  ♦ Fill out even if you are working alone

• Zoom meetings: we need to prevent Zoom bombin
  ♦ Require registration (may add the requirement for authentication as well)
  ♦ Disables screen share of non-host
  ♦ Do not share meeting links outside the class!
A Typical Computer from a Hardware Point of View

CPU

Chipset

Memory

I/O bus

Network
A Typical Computer System: adding software

CPU

CPU

Memory

Programs and data

Operating System Software

OS

Apps

Data

Network
Typical OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer
Typical Unix OS Structure

Application
Libraries
Portable OS Layer
Machine-dependent layer

Written by programmer
Compiled by programmer
Uses library calls
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

Written by gurus
Provided pre-compiled
interface defined in
headers
Invoked like functions
Input to linker (compiler)
May be “resolved” when
program is loaded
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

“Guts” of system calls
All “high-level” code
Typical Unix OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

Bootstrap
System initialization
I/O device driver
Kernel/user mode switching
Interrupt and exception
Processor management

(OS ~= kernel)
Another Look: UNIX “Onion”

User and Kernel boundary

Applications

OS Service

Device

Hardware

Driver
Roadmap

- System calls
- Interrupt
What is an OS?

- Resource manager
  - Manage shared resources (CPU, mem, I/O, networking)

- Extended (abstract) machine
Dual-Mode Operation

- OS manages shared resources
- OS protects programs from other programs
- OS needs to be “privileged”

Every CPU (a CPU core actually) can run in one of the two modes:
  - Kernel mode – can run all instructions
  - User mode – can only run non-privileged instructions
  - Mode is indicated by a status bit in a protected CPU control register
Privileged Instructions

- Privileged instructions: a subset of instructions that can only run in kernel mode
  - CPU checks mode bit when privileged instructions execute
  - Attempts to execute in user mode are detected and prevented by CPU
- Privileged instructions include
  - Directly access I/O devices (disks, printers, etc.)
    » For security, fairness
  - Manipulate memory management state
    » Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    » E.g., mode bit, interrupt level
  - Halt instruction
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

User level
(run in user mode)

Kernel level
(run in kernel mode)
**Dual-Mode Operation**

- OS manages shared resources
- OS protects programs from other programs
  - OS needs to be privileged

- If OS manages shared resources, how does a user program request for accessing shared resources (e.g. hard drive)?
System calls

- Interface between a *process* and the operating system kernel
  - Kernel manages shared resources & exports interface
  - Process requests for access to shared resources

- Generally available as assembly-language instructions

- Directly invoked in many languages (C, C++, Perl)
  - Who is helping out here?
Typical Unix OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

Written by gurus
Provided pre-compiled
interface defined in
headers
Invoked like functions
Input to linker (compiler)
May be “resolved” when
program is loaded
Example

• Given the I/O instructions are privileged, how does the user program perform I/O?
  ♦ open()
  ♦ read()
  ♦ write()
  ♦ close()
System calls

• Categories
  ♦ Process management
  ♦ Memory management
  ♦ File management
  ♦ Device management
  ♦ Networking
System calls in Linux (man syscalls)

- SYSCALLS(2)  Linux Programmer’s Manual  SYSCALLS(2)

- NAME
  - none - list of all system calls

- SYNOPSIS
  - Linux 2.4 system calls.

- DESCRIPTION
  - The system call is the fundamental interface between an application and the Linux kernel. As of Linux 2.4.17, there are 1100 system calls listed in /usr/src/linux/include/asm-*/unistd.h. This man page lists those that are common to most platforms (providing hyperlinks if you read this with a browser).

  _llseek(2), _newselect(2), _sysctl(2), accept(2), access(2), acct(2), adjtimex(2), afs_syscall, alarm(2), bdflush(2), bind(2), break, brk(2), cacheflush(2), capget(2), capset(2), chdir(2), chmod(2), chown(2), chown32, chroot(2), clone(2), close(2), connect(2), creat(2), create_module(2), delete_module(2), dup(2), dup2(2), execve(2), exit(2), fchdir(2), fchmod(2), fchown(2), fchown32, fcntl(2), fcntl64, fdata- ......
Invoking system calls (man syscall)

- **NAME**
  - syscall - indirect system call

- **SYNOPSIS**
  - #define _GNU_SOURCE /* or _BSD_SOURCE or _SVID_SOURCE */
  - #include <unistd.h>
  - #include <sys/syscall.h> /* For SYS_xxx definitions */
  - int syscall(int number, ...);

- **DESCRIPTION**
  - syscall() performs the system call whose assembly language interface has the specified number with the specified arguments. Symbolic constants for system calls can be found in the header file <sys/syscall.h>.

- **RETURN VALUE**
  - The return value is defined by the system call being invoked. In general, a 0 return value indicates success. A -1 return value indicates an error, and an error code is stored in errno.

- **EXAMPLE**
  - #define _GNU_SOURCE
  - #include <unistd.h>
  - #include <sys/syscall.h>
  - #include <sys/types.h>

  ```c
  int main(int argc, char *argv[]) {
      pid_t tid;
      tid = syscall(SYS_gettid);
  }
  ```
Transition from user to kernel mode
System Calls

- CPU ISA provides a system call instruction that:
  - Causes a trap to kernel
  - Passes a syscall # to determine which syscall handler to invoke
  - Saves caller state (PC, regs, mode) so it can be restored
  - Returning from system call restores this state

- Requires architectural support to:
  - Restore saved state, reset mode, resume execution

What would happen if the kernel did not save state?
System Call

- Firefox: read()
  - Trap to kernel mode, save state
  - Trap handler
    - Find the handler for read() in syscall table
    - read() kernel routine
  - Restore state, return to user level, resume execution
Roadmap

• System calls

• Interrupt
A Typical Computer from a Hardware Point of View

CPU

Chipset

Memory

I/O bus

Network

CPU

...
Concurrency & Unexpected Events

• How do human handle unexpected events?
  ♦ Assume
    » mail delivered to my mailbox continuously
    » I have a secretary
  ♦ Do I have mail now? (need to be processed quickly)

• Poll vs. interrupt

• Which one is more efficient?
  » Assume 1 interrupt more costly than 1 poll
  » If I have one mail per day?
  » If I have lots of mail per delivery?
Interrupt

- A mechanism for
  - coordination between concurrently operating units of a computer system (e.g. CPU and I/O devices)
  - for responding to specific conditions within a computer

- Results in transfer of flow of control (to interrupt handler in the OS), forced by hardware
Two types of Interrupts

- **Hardware interrupts**
  - Timer expires
  - I/O device events: keyboard strokes, receiving a network packet, etc.

- **Software interrupts** (aka. trap, exception)
  - an error (floating point exception)
  - a system call requesting OS for special services

- The kernel defines a handler for each interrupt type
  - Interrupt handlers always execute in kernel mode
  - The specific types of interrupts are defined by the hardware
Handling interrupts

- Incoming interrupts are disabled (at this and lower priority levels) while the interrupt is being processed to prevent a lost interrupt.
- Interrupt architecture must save the address of the interrupted instruction.
- Interrupt transfers control to the interrupt service routine.
  - Generally, through the interrupt vector, which contains the addresses of all the service routines.
- If interrupt routine modifies process state (register values):
  - Save the current state of the CPU (registers and the program counter) on the system stack.
  - Restore the state before returning.
- Interrupts are re-enabled after servicing current interrupt.
- Resume the interrupted instruction.
## X86 Interrupt and Exceptions

<table>
<thead>
<tr>
<th>Vector #</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide error (by zero)</td>
<td>Fault</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug</td>
<td>Fault/trap</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Non-Maskable interrupt</td>
<td>Interrupt</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>breakpoint</td>
<td>Trap</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>Trap</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND range exceeded</td>
<td>Trap</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid opcode</td>
<td>Fault</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device not available</td>
<td>Fault</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double fault</td>
<td>Abort</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Coprocessor segment overrun</td>
<td>Fault</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td></td>
</tr>
</tbody>
</table>
## X86 Interrupt and Exceptions (2)

<table>
<thead>
<tr>
<th>Vector #</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment not present</td>
<td>Fault</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack-segment fault</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General protection</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td></td>
<td>Fault</td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>Floating-point error (math fault)</td>
<td>Fault</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment check</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>19-31</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td></td>
<td>User defined</td>
<td>Interrupt</td>
</tr>
</tbody>
</table>

Vector 128 is for system calls
Interrupt time line for a single process doing I/O
I/O Completion

- 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
- If you have ever installed early versions of Windows, you now know what IRQs are for
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 descriptors to find packet in memory)
6. Upon completion, restores saved state from stack
The timer is critical for an operating system. It is the fallback mechanism by which the OS reclaims control over the machine. 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 kernel, which controls what runs next. Basis for OS scheduler (more later…)

Prevents infinite loops. OS can always regain control from erroneous or malicious programs that try to hog CPU.

Also used for time-based functions (e.g., sleep).
[lec1] What is an OS?

• Resource manager

• Extended (abstract) machine

• (will have a 3rd def based on pragmatics next time)


“Modern OSes are interrupt driven”

• “An OS is a giant interrupt handler!” (Def 3)
• Once the system is booted, all entry to the kernel occurs as the result of an interrupt
  ♦ Timer interrupt → Context switches in multiprogramming
  ♦ (unexpected) I/O interrupts
  ♦ System calls to switch from user to kernel mode

• At the lowest level an OS is just a bunch of interrupt service routines
  ♦ Each routine simply returns to whatever was executing before it was interrupted
    » A user process
    » An OS process
    » Another interrupt routine
  ♦ Else infinite wait loop
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 IRQ 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?
Questions?

- We will dive in process management next week
- Read Chapters 4-5