Operating Systems:
Virtual Machines & Exceptions

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
6.004 So Far: Single-User Machines

- Hardware executes a single program
- This program has direct and complete access to all hardware resources in the machine
6.004 So Far: Single-User Machines

- Hardware executes a single program
- This program has direct and complete access to all hardware resources in the machine
6.004 So Far: Single-User Machines

- Hardware executes a single program
- This program has direct and complete access to all hardware resources in the machine
- The instruction set architecture (ISA) is the interface between software and hardware
6.004 So Far: Single-User Machines

- Hardware executes a single program
- This program has direct and complete access to all hardware resources in the machine
- The instruction set architecture (ISA) is the interface between software and hardware
- Most computer systems don’t work like this!
Operating Systems

- Multiple executing programs share the machine
- Each executing program does not have direct access to hardware resources
Operating Systems

- Multiple executing programs share the machine
- Each executing program does not have direct access to hardware resources
- Instead, an operating system (OS) controls these programs and how they share hardware resources
Multiple executing programs share the machine
Each executing program does not have direct access to hardware resources
Instead, an operating system (OS) controls these programs and how they share hardware resources
  Only the OS has unrestricted access to hardware
Multiple executing programs share the machine
Each executing program does not have direct access to hardware resources
Instead, an operating system (OS) controls these programs and how they share hardware resources
  Only the OS has unrestricted access to hardware
The application binary interface (ABI) is the interface between programs and the OS
Nomenclature: Process vs. Program

- A program is a collection of instructions (i.e., just the code)
Nomenclature: Process vs. Program

- A program is a collection of instructions (i.e., just the code)
- A **process** is an instance of a program that is being executed
  - Includes program code + other state (registers, memory, and other resources)
Nomenclature: Process vs. Program

- A program is a collection of instructions (i.e., just the code)
- A process is an instance of a program that is being executed
  - Includes program code + other state (registers, memory, and other resources)
- The OS Kernel is a process with special privileges
Goals of Operating Systems

- Protection and privacy: Processes cannot access each other’s data
Goals of Operating Systems

- **Protection and privacy**: Processes cannot access each other’s data
Goals of Operating Systems

- **Protection** and privacy: Processes cannot access each other’s data
- **Abstraction**: OS hides details of underlying hardware
  - e.g., processes open and access files instead of issuing raw commands to the disk
Goals of Operating Systems

- **Protection** and privacy: Processes cannot access each other’s data
- **Abstraction**: OS hides details of underlying hardware
  - e.g., processes open and access files instead of issuing raw commands to the disk
- **Resource management**: OS controls how processes share hardware (CPU, memory, disk, etc.)
The OS kernel provides a **private address space** to each process

- Each process is allocated space in physical memory by the OS
- A process is not allowed to access the memory of other processes
The OS kernel provides a **private address space** to each process.

- Each process is allocated space in physical memory by the OS.
- A process is not allowed to access the memory of other processes.
The OS kernel provides a **private address space** to each process
- Each process is allocated space in physical memory by the OS
- A process is not allowed to access the memory of other processes

The OS kernel **schedules processes** into the CPU
- Each process is given a fraction of CPU time
- A process cannot use more CPU time than allowed
The OS kernel provides a private address space to each process:
- Each process is allocated space in physical memory by the OS.
- A process is not allowed to access the memory of other processes.

The OS kernel schedules processes into the CPU:
- Each process is given a fraction of CPU time.
- A process cannot use more CPU time than allowed.

Diagram:
- Physical Memory:
  - OS Kernel memory
  - Process 1 memory
  - Process 2 memory
  - Free memory

Timeline:
- Running process: Process 1
- Process 2
- Process 1

Time:

November 13, 2018
The OS kernel provides a **private address space** to each process
- Each process is allocated space in physical memory by the OS
- A process is not allowed to access the memory of other processes

The OS kernel **schedules processes** into the CPU
- Each process is given a fraction of CPU time
- A process cannot use more CPU time than allowed

The OS kernel lets processes invoke system services (e.g., access files or network sockets) via **system calls**
Virtual Machines
A New Layer of Abstraction

- The OS gives a Virtual Machine (VM) to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines
A New Layer of Abstraction

- The OS gives a **Virtual Machine (VM)** to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines
A New Layer of Abstraction

- The OS gives a Virtual Machine (VM) to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines
A New Layer of Abstraction

- The OS gives a **Virtual Machine (VM)** to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines
A New Layer of Abstraction

- The OS gives a **Virtual Machine (VM)** to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines
A New Layer of Abstraction

- A Virtual Machine (VM) is an *emulation* of a computer system
  - Very general concept, used beyond operating systems

---

**Process1**

- Virtual Processor
- Virtual Memory
- Events
- Files
- Sockets
- Syscalls

---

**OS Kernel** (specially privileged process)

---

**Physical Hardware**

- Processor
- Memory
- Disk
- Network card
- Display
- Keyboard

---

November 13, 2018

MIT 6.004 Fall 2018
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

RISC-V process (quicksort)
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

RISC-V process (quicksort)

RISC-V ISA
Virtual Machines Are Everywhere

Example: How many VMs did you use in Lab 4?

- RISC-V process (quicksort)
- RISC-V emulator (sim.py)
- RISC-V ISA
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

RISC-V process (quicksort)  
RISC-V emulator (sim.py)  
RISC-V ISA  
Implements a RISC-V VM
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

  - RISC-V process (quicksort)
  - RISC-V emulator (sim.py)

RISC-V ISA

Implements a RISC-V VM

Python Language
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

  - RISC-V process (quicksort)
  - RISC-V emulator (sim.py)
  - Python interpreter (CPython)

RISC-V ISA
Implements a RISC-V VM
Python Language
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

- RISC-V process (quicksort)
- RISC-V emulator (sim.py)
- Python interpreter (CPython)

RISC-V ISA
Implements a RISC-V VM

Python Language
Implements a Python VM
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

- RISC-V process (quicksort)
- RISC-V emulator (sim.py)
- Python interpreter (CPython)

RISC-V ISA
- Implements a RISC-V VM

Python Language
- Implements a Python VM

Linux ABI
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

- RISC-V process (quicksort)
- RISC-V emulator (sim.py)
- Python interpreter (CPython)
- Linux OS kernel

RISC-V ISA
- Implements a RISC-V VM
Python Language
- Implements a Python VM
Linux ABI
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

  - RISC-V process (quicksort)
  - RISC-V emulator (sim.py)
  - Python interpreter (CPython)
  - Linux OS kernel

RISC-V ISA
- Implements a RISC-V VM

Python Language
- Implements a Python VM

Linux ABI
- Implements a Linux-x86 VM
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

- RISC-V process (quicksort)
  - RISC-V ISA
    - Implements a RISC-V VM
  - Python Language
    - Implements a Python VM
  - Linux ABI
    - Implements a Linux-x86 VM
  - x86 ISA

- RISC-V emulator (sim.py)

- Python interpreter (CPython)
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

RISC-V process (quicksort)

RISC-V emulator (sim.py)

Python interpreter (CPython)

Linux OS kernel

VirtualBox

RISC-V ISA

Implements a RISC-V VM

Python Language

Implements a Python VM

Linux ABI

Implements a Linux-x86 VM

x86 ISA
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

- RISC-V process (quicksort)
- RISC-V emulator (sim.py)
- Python interpreter (CPython)
- Linux OS kernel
- VirtualBox

- RISC-V ISA
  Implements a RISC-V VM

- Python Language
  Implements a Python VM

- Linux ABI
  Implements a Linux-x86 VM

- x86 ISA
  Implements an x86 system VM
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

RISC-V process (quicksort)
RISC-V ISA
Implements a RISC-V VM

RISC-V emulator (sim.py)
Python Language
Implements a Python VM

Python interpreter (CPython)
Linux ABI
Implements a Linux-x86 VM

Linux OS kernel
x86 ISA
Implements an x86 system VM

VirtualBox
Win/Linux/MacOS/... ABI
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

- RISC-V process (quicksort)
- RISC-V emulator (sim.py)
- Python interpreter (CPython)
- Linux OS kernel
- VirtualBox
- OS kernel (Win/Linux/MacOS/...)

- RISC-V ISA
  - Implements a RISC-V VM

- Python Language
  - Implements a Python VM

- Linux ABI
  - Implements a Linux-x86 VM

- x86 ISA
  - Implements an x86 system VM

Win/Linux/MacOS/... ABI
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

- RISC-V process (quicksort)
- RISC-V emulator (sim.py)
- Python interpreter (CPython)
- Linux OS kernel
- VirtualBox
- OS kernel (Win/Linux/MacOS/...)

- RISC-V ISA
  - Implements a RISC-V VM

- Python Language
  - Implements a Python VM

- Linux ABI
  - Implements a Linux-x86 VM

- x86 ISA
  - Implements an x86 system VM

- Win/Linux/MacOS/... ABI
  - Implements an OS-x86 VM
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

  - RISC-V process (quicksort)
  - RISC-V emulator (sim.py)
  - Python interpreter (CPython)
  - Linux OS kernel
  - VirtualBox
  - OS kernel (Win/Linux/MacOS/...)

RISC-V ISA
Implements a RISC-V VM

Python Language
Implements a Python VM

Linux ABI
Implements a Linux-x86 VM

x86 ISA
Implements an x86 system VM

Win/Linux/MacOS/... ABI
Implements an OS-x86 VM

x86 ISA
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

- RISC-V process (quicksort)
- RISC-V emulator (sim.py)
- Python interpreter (CPython)
- Linux OS kernel
- VirtualBox
- OS kernel (Win/Linux/MacOS/...)
- Hardware (e.g., your laptop)

RISC-V ISA
- Implements a RISC-V VM

Python Language
- Implements a Python VM

Linux ABI
- Implements a Linux-x86 VM

x86 ISA
- Implements an x86 system VM

Win/Linux/MacOS/... ABI
- Implements an OS-x86 VM
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 4?

- RISC-V process (quicksort)
  - RISC-V ISA
  - Implements a RISC-V VM

- RISC-V emulator (sim.py)
  - Python Language
  - Implements a Python VM

- Python interpreter (CPython)
  - Linux ABI
  - Implements a Linux-x86 VM

- Linux OS kernel
  - x86 ISA
  - Implements an x86 system VM

- VirtualBox
  - Win/Linux/MacOS/… ABI
  - Implements an OS-x86 VM

- OS kernel (Win/Linux/MacOS/…)
  - x86 ISA
  - Implements an x86 physical machine

- Hardware (e.g., your laptop)
Implementing Virtual Machines

- Virtual machines can be implemented entirely in software, but at a performance cost
  - e.g., Python programs are 10-100x slower than native Linux programs due to Python interpreter overheads
Implementing Virtual Machines

- Virtual machines can be implemented entirely in software, but at a performance cost
  - e.g., Python programs are 10-100x slower than native Linux programs due to Python interpreter overheads

- We want to support operating systems with minimal overheads → need hardware support for virtual machines!
ISA Extensions to Support OS
ISA Extensions to Support OS

- Two modes of execution: user and supervisor
  - OS kernel runs in supervisor mode
  - All other processes run in user mode
ISA Extensions to Support OS

- Two modes of execution: **user** and **supervisor**
  - OS kernel runs in supervisor mode
  - All other processes run in user mode
- Privileged instructions and registers that are only available in supervisor mode
ISA Extensions to Support OS

- Two modes of execution: user and supervisor
  - OS kernel runs in supervisor mode
  - All other processes run in user mode
- Privileged instructions and registers that are only available in supervisor mode
- Interrupts and exceptions to safely transition from user to supervisor mode
ISA Extensions to Support OS

- Two modes of execution: **user** and **supervisor**
  - OS kernel runs in supervisor mode
  - All other processes run in user mode
- Privileged instructions and registers that are only available in supervisor mode
- Interrupts and exceptions to safely transition from user to supervisor mode
- Virtual memory to provide private address spaces and abstract the storage resources of the machine
ISA Extensions to Support OS

- Two modes of execution: user and supervisor
  - OS kernel runs in supervisor mode
  - All other processes run in user mode
- Privileged instructions and registers that are only available in supervisor mode
- Interrupts and exceptions to safely transition from user to supervisor mode
- Virtual memory to provide private address spaces and abstract the storage resources of the machine

Today
ISA Extensions to Support OS

- Two modes of execution: user and supervisor
  - OS kernel runs in supervisor mode
  - All other processes run in user mode
- Privileged instructions and registers that are only available in supervisor mode
- Interrupts and exceptions to safely transition from user to supervisor mode
- Virtual memory to provide private address spaces and abstract the storage resources of the machine

Today

Next lecture
ISA Extensions to Support OS

- Two modes of execution: **user** and **supervisor**
  - OS kernel runs in supervisor mode
  - All other processes run in user mode
- Privileged instructions and registers that are only available in supervisor mode
- Interrupts and exceptions to safely transition from user to supervisor mode
- Virtual memory to provide private address spaces and abstract the storage resources of the machine

**Today**

These ISA extensions work only if hardware and software (OS) agree on a common set of conventions!
Exceptions

- Exception: Event that needs to be processed by the OS kernel. The event is usually unexpected or rare.
Exceptions

- Exception: Event that needs to be processed by the OS kernel. The event is usually unexpected or rare.

```
process

I_{i-1} \rightarrow I_i \rightarrow I_{i+1}

HI_1 \rightarrow HI_2 \rightarrow \ldots \rightarrow HI_n

exception handler (in OS kernel)
```
Causes for Exceptions

- The terms exception and interrupt are often used interchangeably, with a minor distinction:
Causes for Exceptions

- The terms exception and interrupt are often used interchangeably, with a minor distinction:

- **Exceptions** usually refer to synchronous events, generated by the process itself (e.g., illegal instruction, divide-by-0, illegal memory address, system call)
Causes for Exceptions

- The terms exception and interrupt are often used interchangeably, with a minor distinction:

- **Exceptions** usually refer to synchronous events, generated by the process itself (e.g., illegal instruction, divide-by-0, illegal memory address, system call)

- **Interrupts** usually refer to asynchronous events, generated by I/O devices (e.g., timer expired, keystroke, packet received, disk transfer complete)
Causes for Exceptions

- The terms exception and interrupt are often used interchangeably, with a minor distinction:

- **Exceptions** usually refer to **synchronous events**, generated by the process itself (e.g., illegal instruction, divide-by-0, illegal memory address, system call)

- **Interrupts** usually refer to **asynchronous events**, generated by I/O devices (e.g., timer expired, keystroke, packet received, disk transfer complete)

- We use exception to encompass both types of events, and use synchronous exception for synchronous events
Handling Exceptions

- When an exception happens, the processor:
Handling Exceptions

- When an exception happens, the processor:
  - Stops the current process at instruction $I_i$, completing all the instructions up to $I_{i-1}$ (precise exceptions)
Handling Exceptions

- When an exception happens, the processor:
  - Stops the current process at instruction $I_i$, completing all the instructions up to $I_{i-1}$ (*precise exceptions*)
  - Saves the PC of instruction $I_i$ and the reason for the exception in special (privileged) registers
Handling Exceptions

- When an exception happens, the processor:
  - Stops the current process at instruction $I_i$, completing all the instructions up to $I_{i-1}$ (*precise exceptions*)
  - Saves the PC of instruction $I_i$ and the reason for the exception in special (privileged) registers
  - Enables supervisor mode, disables interrupts, and transfers control to a pre-specified exception handler PC
Handling Exceptions

- When an exception happens, the processor:
  - Stops the current process at instruction $I_i$, completing all the instructions up to $I_{i-1}$ (*precise exceptions*)
  - Saves the PC of instruction $I_i$ and the reason for the exception in special (privileged) registers
  - Enables supervisor mode, disables interrupts, and transfers control to a pre-specified exception handler PC

- After the OS kernel handles the exception, it returns control to the process at instruction $I_i$
  - Exception is transparent to the process!
Handling Exceptions

- When an exception happens, the processor:
  - Stops the current process at instruction $I_i$, completing all the instructions up to $I_{i-1}$ (*precise exceptions*)
  - Saves the PC of instruction $I_i$ and the reason for the exception in special (privileged) registers
  - Enables supervisor mode, disables interrupts, and transfers control to a pre-specified exception handler PC

- After the OS kernel handles the exception, it returns control to the process at instruction $I_i$
  - Exception is transparent to the process!

- If the exception is due to an illegal operation by the program that cannot be fixed (e.g., an illegal memory access), the OS aborts the process
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel **schedules processes** into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed

- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

<table>
<thead>
<tr>
<th>Process running in CPU</th>
<th>0</th>
<th>10</th>
<th>30</th>
<th>60</th>
<th>80</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (milliseconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel *schedules processes* into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed

- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

<table>
<thead>
<tr>
<th>Time (milliseconds)</th>
<th>0</th>
<th>10</th>
<th>30</th>
<th>60</th>
<th>80</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process running in CPU</td>
<td>Kernel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

```
Set timer to fire in 20ms
Load state (regs, pc, addr space) of process 1
Return control to process 1
```
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

```
Process running in CPU

Kernel  Process 1

Set timer to fire in 20ms
Load state (regs, pc, addr space) of process 1
Return control to process 1
```

<table>
<thead>
<tr>
<th>Process</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>110</td>
</tr>
</tbody>
</table>

November 13, 2018
Case Study 1: CPU Scheduling  
Enabled by timer interrupts

- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed

- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

Process 1

<table>
<thead>
<tr>
<th>Time (milliseconds)</th>
<th>0</th>
<th>10</th>
<th>30</th>
<th>60</th>
<th>80</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process running in CPU</td>
<td>Kernel</td>
<td>Process 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Timer interrupt → exception handler runs

Set timer to fire in 20ms
Load state (regs, pc, addr space) of process 1
Return control to process 1
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel **schedules processes** into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
- **Key enabling technology: Timer interrupts**
  - Kernel sets timer, which raises an interrupt after a specified time

Timeline:
- **0** to **10 ms**: Kernel
  - Set timer to fire in 20ms
  - Load state (regs, pc, addr space) of process 1
  - Return control to process 1
- **10** to **30 ms**: Process 1
  - Timer interrupt \(\rightarrow\) exception handler runs
  - Save state of process 1
  - Decide to schedule process 2
  - Set timer to fire in 30ms
  - Load state of process 2, return control to it

- **30** to **60 ms**: Timer interrupt
- **60** to **80 ms**: Process 2
- **80** to **110 ms**: Process 2

November 13, 2018
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

<table>
<thead>
<tr>
<th>Time (milliseconds)</th>
<th>0</th>
<th>10</th>
<th>30</th>
<th>60</th>
<th>80</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process running in CPU</td>
<td>Kernel</td>
<td>Process 1</td>
<td>Process 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Timer interrupt → exception handler runs
- Save state of process 1
- Decide to schedule process 2
- Set timer to fire in 30ms
- Load state of process 2, return control to it

- Set timer to fire in 20ms
- Load state (regs, pc, addr space) of process 1
- Return control to process 1
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel **schedules processes** into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed

- **Key enabling technology:** Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

---

**Timeline:**
- **0:** Kernel, Process 1
- **10:** Load state (regs, pc, addr space) of process 1
- **30:** Timer interrupt → exception handler runs
  - Save state of process 1
  - Decide to schedule process 2
  - Set timer to fire in 30ms
  - Load state of process 2, return control to it
- **60:** Process 2
- **80:** 30ms
- **110:** 20ms

**Steps:**
1. Set timer to fire in 20ms
2. Load state (regs, pc, addr space) of process 1
3. Return control to process 1
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel **schedules processes** into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed

- **Key enabling technology:** Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

---

**Timeline:**

<table>
<thead>
<tr>
<th>Time (milliseconds)</th>
<th>Process running in CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Kernel</td>
</tr>
<tr>
<td>10</td>
<td>Process 1</td>
</tr>
<tr>
<td>30</td>
<td>Process 2</td>
</tr>
<tr>
<td>60</td>
<td>Process 1</td>
</tr>
<tr>
<td>80</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>

**Event sequence:**

- Timer interrupt → exception handler runs
- Save state of process 1
- Decide to schedule process 2
- Set timer to fire in 30ms
- Load state of process 2, return control to it

**Initial state:**
- Set timer to fire in 20ms
- Load state (regs, pc, addr space) of process 1
- Return control to process 1
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

Process running in CPU

<table>
<thead>
<tr>
<th>Time (milliseconds)</th>
<th>0</th>
<th>10</th>
<th>30</th>
<th>60</th>
<th>80</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process running</td>
<td>Kernel</td>
<td>Process 1</td>
<td>Process 2</td>
<td>Process 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Timeline:
- Set timer to fire in 20ms
- Load state (regs, pc, addr space) of process 1
- Return control to process 1
- Timer interrupt → exception handler runs
- Save state of process 1
- Decide to schedule process 2
- Set timer to fire in 30ms
- Load state of process 2, return control to it
Case Study 1: CPU Scheduling
Enabled by timer interrupts

- The OS kernel **schedules processes** into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed

- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

**Timeline:**

- **0-10 ms:** Kernel runs
- **10-30 ms:** Process 1 runs
- **30-60 ms:** Process 2 runs
- **60-80 ms:** Process 1 runs
- **80-110 ms:** Process 2 runs

**Events:**

- **0 ms:** Set timer to fire in 20ms
- **10 ms:** Load state (regs, pc, addr space) of process 1
- **110 ms:** Return control to process 1

Timer interrupt → exception handler runs
- Save state of process 1
- Decide to schedule process 2
- Set timer to fire in 30ms
- Load state of process 2, return control to it

L19-15

November 13, 2018
Case Study 2: Emulating Instructions
Enabled by illegal instruction exceptions

- `mul x1, x2, x3` is an instruction in the RISC-V ‘M’ extension (`x1 := x2 * x3`)
  - If ‘M’ is not implemented, this is an illegal instruction
Case Study 2: Emulating Instructions
Enabled by illegal instruction exceptions

- \texttt{mul x1, x2, x3} is an instruction in the RISC-V `M` extension \((x1 := x2 \times x3)\)
  - If `M` is not implemented, this is an illegal instruction

- What happens if we run code from an RV32IM machine on an RV32I machine?
Case Study 2: Emulating Instructions
Enabled by illegal instruction exceptions

- \texttt{mul x1, x2, x3} is an instruction in the RISC-V ‘M’ extension (\(x1 := x2 \times x3\))
  - If ‘M’ is not implemented, this is an illegal instruction

- What happens if we run code from an RV32IM machine on an RV32I machine?
  - \texttt{mul} causes an illegal instruction exception
Case Study 2: Emulating Instructions
Enabled by illegal instruction exceptions

- `mul x1, x2, x3` is an instruction in the RISC-V ‘M’ extension (`x1 := x2 * x3`)
  - If ‘M’ is not implemented, this is an illegal instruction

- What happens if we run code from an RV32IM machine on an RV32I machine?
  - `mul` causes an illegal instruction exception

- The exception handler can take over and abort the process... but it can also emulate the instruction!
Emulating Unsupported Instructions

Process running in CPU

Time

Process 1

Process 1 code:

...  
add a3, a2, a1  
mul a4, a3, a2  
xor a5, a4, a3  
...
Emulating Unsupported Instructions

Process running in CPU

Process 1

Illegal instruction exception

Process 1 code:

... 
add a3, a2, a1 
mul a4, a3, a2 
xor a5, a4, a3 
...
Emulating Unsupported Instructions

Process running in CPU

Time

Process 1

Kernel

Illegal instruction exception

Process 1 code:

...  
add a3, a2, a1
mul a4, a3, a2
xor a5, a4, a3
...
Emulating Unsupported Instructions

Process running in CPU

Process 1 code:

... add a3, a2, a1
mul a4, a3, a2
xor a5, a4, a3
...

Illegal instruction exception
Emulating Unsupported Instructions

Process running in CPU

Time

Process 1

Kernel

Illegal instruction exception

Save state of process 1

Emulate a multiply instruction in software (e.g., by repeated addition)

Process 1 code:

```plaintext
... add a3, a2, a1
mul a4, a3, a2
xor a5, a4, a3
...
```
Emulating Unsupported Instructions

Process running in CPU

Time

Process 1

Kernel

Illegal instruction exception

Save state of process 1

Emulate a multiply instruction in software (e.g., by repeated addition)

Load state of process 1

Return control to process 1 at instruction following the multiply

Process 1 code:

... add a3, a2, a1
mul a4, a3, a2
xor a5, a4, a3
...

November 13, 2018
Emulating Unsupported Instructions

Process running in CPU

Time

Process 1
Kernel
Process 1

Illegal instruction exception

Save state of process 1
Emulate a multiply instruction in software (e.g., by repeated addition)
Load state of process 1
Return control to process 1 at instruction following the multiply

Process 1 code:

...  
add a3, a2, a1
mul a4, a3, a2
xor a5, a4, a3
...

November 13, 2018
Emulating Unsupported Instructions

- Result: Program believes it is executing in a RV32IM processor, when it’s actually running in a RV32I
Emulating Unsupported Instructions

Result: Program believes it is executing in a RV32IM processor, when it’s actually running in a RV32I

- Any problem?
Emulating Unsupported Instructions

- Result: Program believes it is executing in a RV32IM processor, when it’s actually running in a RV32I
  - Any problem? Much slower than a hardware multiply
RISC-V Exception Handling

- RISC-V provides several privileged registers, called control and status registers (CSRs), e.g.,
  - mepc: exception PC
  - mcause: cause of the exception (interrupt, illegal instr, etc.)
  - mtvec: address of the exception handler
  - mstatus: status bits (privilege mode, interrupts enabled, etc.)
RISC-V Exception Handling

- RISC-V provides several privileged registers, called control and status registers (CSRs), e.g.,
  - mepc: exception PC
  - mcause: cause of the exception (interrupt, illegal instr, etc.)
  - mtvec: address of the exception handler
  - mstatus: status bits (privilege mode, interrupts enabled, etc.)

- RISC-V also provides privileged instructions, e.g.,
  - csrr and csrw to read/write CSRs
  - mret to return from the exception handler to the process
RISC-V Exception Handling

- RISC-V provides several privileged registers, called control and status registers (CSRs), e.g.,
  - mepc: exception PC
  - mcause: cause of the exception (interrupt, illegal instr, etc.)
  - mtvec: address of the exception handler
  - mstatus: status bits (privilege mode, interrupts enabled, etc.)

- RISC-V also provides privileged instructions, e.g.,
  - csrr and csrw to read/write CSRs
  - mret to return from the exception handler to the process
  - Trying to execute these instructions from user mode causes an exception → normal processes cannot take over the system
Typical Exception Handler Structure

- A small common handler (CH) written in assembly + many exception handlers (EHs), one for each cause (typically written in normal C code)
Typical Exception Handler Structure

- A small common handler (CH) written in assembly + many exception handlers (EHs), one for each cause (typically written in normal C code)
- Common handler is invoked on every exception:
Typical Exception Handler Structure

- A small common handler (CH) written in assembly + many exception handlers (EHs), one for each cause (typically written in normal C code)
- Common handler is invoked on every exception:
  1. Saves registers x1-x31, mepc into known memory locations
Typical Exception Handler Structure

- A small common handler (CH) written in assembly + many exception handlers (EHs), one for each cause (typically written in normal C code)

- Common handler is invoked on every exception:
  1. Saves registers x1-x31, mepc into known memory locations
  2. Passes mcause, process state to the right EH to handle the specific exception/interrupt
Typical Exception Handler Structure

- A small common handler (CH) written in assembly + many exception handlers (EHs), one for each cause (typically written in normal C code)

- Common handler is invoked on every exception:
  1. Saves registers x1-x31, mepc into known memory locations
  2. Passes mcause, process state to the right EH to handle the specific exception/interrupt
  3. EH returns which process should run next (could be the same or a different one)
Typical Exception Handler Structure

- A small common handler (CH) written in assembly + many exception handlers (EHs), one for each cause (typically written in normal C code)

- Common handler is invoked on every exception:
  1. Saves registers x1-x31, mepc into known memory locations
  2. Passes mcause, process state to the right EH to handle the specific exception/interrupt
  3. EH returns which process should run next (could be the same or a different one)
  4. CH loads x1-x31, mepc from memory for the right process
Typical Exception Handler Structure

- A small common handler (CH) written in assembly + many exception handlers (EHs), one for each cause (typically written in normal C code)

- Common handler is invoked on every exception:
  1. Saves registers x1-x31, mepc into known memory locations
  2. Passes mcause, process state to the right EH to handle the specific exception/interrupt
  3. EH returns which process should run next (could be the same or a different one)
  4. CH loads x1-x31, mepc from memory for the right process
  5. CH executes mret, which sets pc to mepc, disables supervisor mode, and re-enables interrupts
common_handler: // entry point for exception handler
   // save x1 to mscratch to free up a register
   csrw mscratch, x1 // write x1 to mscratch CSR
Common Exception Handler
RISC-V Assembly code

common_handler: // entry point for exception handler
    // save x1 to mscratch to free up a register
    csrw mscratch, x1 // write x1 to mscratch CSR
    // get the pointer for current process’s state
    lw x1, curProcState
common_handler: // entry point for exception handler
// save x1 to mscratch to free up a register
csrw mscratch, x1 // write x1 to mscratch CSR
// get the pointer for current process’s state
lw x1, curProcState
// save registers x2-x31
sw x2, 8(x1)
sw x3, 12(x1)
...
sw x31, 124(x1)
common_handler: // entry point for exception handler
  // save x1 to mscratch to free up a register
  csrw mscratch, x1 // write x1 to mscratch CSR
  // get the pointer for current process’s state
  lw x1, curProcState
  // save registers x2-x31
  sw x2, 8(x1)
  sw x3, 12(x1)
  ...
  sw x31, 124(x1)
  // now registers x2-x31 are free for the kernel
Common Exception Handler
RISC-V Assembly code

```assembly
common_handler: // entry point for exception handler
    // save x1 to mscratch to free up a register
    csrw mscratch, x1 // write x1 to mscratch CSR
    // get the pointer for current process’s state
    lw x1, curProcState
    // save registers x2-x31
    sw x2, 8(x1)
    sw x3, 12(x1)
    ...
    sw x31, 124(x1)
    // now registers x2-x31 are free for the kernel
    // save original x1 (now in mscratch)
    csrr t0, mscratch
    sw t0, 4(x1)
```
Common Exception Handler
RISC-V Assembly code

```assembly
common_handler: // entry point for exception handler
  // save x1 to mscratch to free up a register
  csrw mscratch, x1 // write x1 to mscratch CSR
  // get the pointer for current process’s state
  lw x1, curProcState
  // save registers x2-x31
  sw x2, 8(x1)
  sw x3, 12(x1)
  ...
  sw x31, 124(x1)
  // now registers x2-x31 are free for the kernel
  // save original x1 (now in mscratch)
  csrr t0, mscratch
  sw t0, 4(x1)
  // finally, save mepc
  csrr t1, mscratch
  sw t1, 0(x1)
```

November 13, 2018
Common Exception Handler cont.
Calling EH_Dispatcher and returning

```c
common_handler:
   ...  // we have saved the state of the process
```
Common Exception Handler cont.
Calling EH_Dispatcher and returning

common_handler:
    ...  // we have saved the state of the process
    // call function to handle exception
    lw sp, kernelSp  // use the kernel’s stack
Common Exception Handler cont.
Calling EH_Dispatcher and returning

common_handler:
... // we have saved the state of the process
// call function to handle exception
lw sp, kernelSp // use the kernel’s stack
mv a0, x1 // arg 0: address of process state
csrr a1, mcause // arg 1: exception cause
jal eh_dispatcher // calls the appropriate handler
Common Exception Handler \textit{cont.}
Calling EH\_Dispatcher and returning

```c
common_handler:
... // we have saved the state of the process
// call function to handle exception
lw sp, kernelSp // use the kernel’s stack
mv a0, x1 // arg 0: address of process state
csrr a1, mcause // arg 1: exception cause
jal eh\_dispatcher // calls the appropriate handler
// returns address of state of process to schedule
```
Common Exception Handler cont.
Calling EH_Dispatcher and returning

```c
common_handler:
    ... // we have saved the state of the process
    // call function to handle exception
    lw sp, kernelSp // use the kernel’s stack
    mv a0, x1 // arg 0: address of process state
    csrr a1, mcause // arg 1: exception cause
    jal eh_dispatcher // calls the appropriate handler
    // returns address of state of process to schedule
    // restore return PC in mepc
    lw t0, 0(a0)
    csrw mepc, t0
```
Common Exception Handler cont.
Calling EH_Dispatcher and returning

custom_handler:
  ... // we have saved the state of the process
  // call function to handle exception
  lw sp, kernelSp // use the kernel’s stack
  mv a0, x1 // arg 0: address of process state
  csrr a1, mcause // arg 1: exception cause
  jal eh_dispatcher // calls the appropriate handler
  // returns address of state of process to schedule
  // restore return PC in mepc
  lw t0, 0(a0)
  csrw mepc, t0
  // restore x1-x31
  mv x1, a0
  lw x2, 8(x1); lw x3, 12(x1); ...; lw x31, 124(x1)
  lw x1, 4(x1) // restore x1 last
Common Exception Handler cont.
Calling EH_Dispatcher and returning

common_handler:
... // we have saved the state of the process
// call function to handle exception
lw sp, kernelSp // use the kernel’s stack
mv a0, x1 // arg 0: address of process state
csrr a1, mcause // arg 1: exception cause
jal eh_dispatcher // calls the appropriate handler
// returns address of state of process to schedule
// restore return PC in mepc
lw t0, 0(a0)
csrw mepc, t0
// restore x1-x31
mv x1, a0
lw x2, 8(x1); lw x3, 12(x1); ...; lw x31, 124(x1)
lw x1, 4(x1) // restore x1 last
mret // return control to program
**EH Dispatcher (in C)**

Dispatches to a specific handler based on “cause”

```c
typedef struct {
    int pc;
    int regs[31];
    ...
} ProcState;

ProcState* eh_dispatcher(ProcState* curProc, int cause) {

    ...
}
```
EH Dispatcher (in C)
Dispatches to a specific handler based on “cause”

```c
typedef struct {
    int pc;
    int regs[31];
    ...
} ProcState;

ProcState* eh_dispatcher(ProcState* curProc, int cause) {
    if(cause == 0x02) // illegal instruction
        return illegal_eh(curProc, cause);

    ...
}
```
typedef struct {
    int pc;
    int regs[31];
    ...
} ProcState;

ProcState* eh_dispatcher(ProcState* curProc, int cause) {
    if (cause == 0x02) // illegal instruction
        return illegal_eh(curProc, cause);
    else if (cause == 0x08)
        ...
}
**EH Dispatcher (in C)**
Dispatches to a specific handler based on “cause”

```c
typedef struct {
    int pc;
    int regs[31];
    ...
} ProcState;

ProcState* eh_dispatcher(ProcState* curProc, int cause) {
    if(cause == 0x02) // illegal instruction
        return illegal_eh(curProc, cause);
    else if(cause == 0x08)
        // system call, e.g, OS service “write” to file
        return syscall_ih(curProc, cause);
    else if (cause < 0) // external interrupt
        ...
}
```
Summary

- Operating System goals:
  - Protection and privacy: Processes cannot access each other’s data
  - Abstraction: OS hides details of underlying hardware
    - e.g., processes open and access files instead of issuing raw commands to disk
  - Resource management: OS controls how processes share hardware resources (CPU, memory, disk, etc.)
Summary

- **Operating System goals:**
  - Protection and privacy: Processes cannot access each other’s data
  - Abstraction: OS hides details of underlying hardware
    - e.g., processes open and access files instead of issuing raw commands to disk
  - Resource management: OS controls how processes share hardware resources (CPU, memory, disk, etc.)

- **Key enabling technologies:**
  - User mode + supervisor mode w/ privileged instructions
  - Exceptions to safely transition into supervisor mode
Summary

- **Operating System goals:**
  - Protection and privacy: Processes cannot access each other’s data
  - Abstraction: OS hides details of underlying hardware
    - e.g., processes open and access files instead of issuing raw commands to disk
  - Resource management: OS controls how processes share hardware resources (CPU, memory, disk, etc.)

- **Key enabling technologies:**
  - User mode + supervisor mode w/ privileged instructions
  - Exceptions to safely transition into supervisor mode
  - Virtual memory to provide private address spaces and abstract the machine’s storage resources (*next lecture*)
Thank you!

Next lecture: Virtual memory