Operating Systems:
Virtual Machines & Exceptions
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!
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
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
  - Only the OS has unrestricted access to hardware
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
  - 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

- process$_1$
- process$_2$
- ... 
- process$_N$

OS Kernel

Application Binary Interface (ABI)

Hardware

ISA
A program is a collection of instructions (i.e., just the code)
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 + state (registers, memory, and other resources)
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 + 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
- **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.

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

Time:
OS Kernel memory
- free
- Process 1 memory
- free
- Process 2 memory
- ...
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
A Virtual Machine (VM) is an **emulation** of a computer system

- Very general concept, used beyond operating systems
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 2?
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 2?

RISC-V process (quicksort)
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 2?

RISC-V process (quicksort)

RISC-V ISA
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 2?

- 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 2?

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 2?

  - 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 2?

  - 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 2?

- 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 2?

  - 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 2?

  - 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 2?
  - 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 2?

- 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

x86 ISA
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 2?

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

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 2?

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

- 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 2?

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

- 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

- Windows/Linux/BSD/... ABI
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 2?

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

- 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
- Windows/Linux/BSD/... ABI

November 5, 2019

MIT 6.004 Fall 2019
### Virtual Machines Are Everywhere

- **Example:** How many VMs did you use in Lab 2?

<table>
<thead>
<tr>
<th>VM Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISC-V process (quicksort)</td>
<td>Implements a RISC-V VM</td>
</tr>
<tr>
<td>RISC-V emulator (sim.py)</td>
<td>Implements a RISC-V VM</td>
</tr>
<tr>
<td>Python interpreter (CPython)</td>
<td>Implements a Python VM</td>
</tr>
<tr>
<td>Linux OS kernel</td>
<td>Implements a Linux-x86 VM</td>
</tr>
<tr>
<td>VMware</td>
<td>Implements an x86 system VM</td>
</tr>
<tr>
<td>OS kernel (Win/Linux/BSD/...)</td>
<td>Implements an OS-x86 VM</td>
</tr>
</tbody>
</table>
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 2?

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

- 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

- Windows/Linux/BSD/... ABI
- Implements an OS-x86 VM

- x86 ISA
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 2?

- RISC-V process (quicksort)
- RISC-V emulator (sim.py)
- Python interpreter (CPython)
- Linux OS kernel
- VMware
- OS kernel (Win/Linux/BSD/...)
- Hardware (Athena server)

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
- Implements an OS-x86 VM

Windows/Linux/BSD/... ABI

November 5, 2019
MIT 6.004 Fall 2019
Virtual Machines Are Everywhere

- Example: How many VMs did you use in Lab 2?

- 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

- VMware
  - Windows/Linux/BSD/... ABI
  - Implements an OS-x86 VM

- OS kernel (Win/Linux/BSD/...)
  - x86 ISA
  - Implements an x86 physical machine

- Hardware (Athena server)
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
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
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

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.

![Diagram showing exception processing]

process

$I_i$ $\rightarrow$ $HI_1$ $\rightarrow$ $HI_2$ $\rightarrow$ $HI_n$

exception handler (in OS kernel)
Exceptions

- Exception: Event that needs to be processed by the OS kernel. The event is usually unexpected or rare.
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
Exception Use #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
Exception Use #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
Exception Use #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>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>60</td>
<td>80</td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>
Exception Use #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>

November 5, 2019
MIT 6.004 Fall 2019
L17-15
Exception Use #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>

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

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

---

**Diagram:**

- Process 1
- Time (milliseconds): 0, 10, 30, 60, 80, 110
- Process running in CPU: Kernel, Process 1
- 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

November 5, 2019

MIT 6.004 Fall 2019

L17-15
Exception Use #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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>110</td>
</tr>
</tbody>
</table>

**Process running in CPU**

- Kernel
- 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

**Set timer to fire in 20ms**
- Load state (regs, pc, addr space) of process 1
- Return control to process 1
Exception Use #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>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></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>

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

- **Process 1**
  - Load state (regs, pc, addr space)
  - Return control to process 1

- **Process 2**
  - Decide to schedule process 2
  - Set timer to fire in 30ms
  - Load state of process 2, return control to it

- **Timer interrupt** → exception handler runs
  - Save state of process 1
  - Set timer to fire in 30ms
  - Load state of process 2, return control to it
Exception Use #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:
- 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
Exception Use #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>

- 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
Exception Use #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>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>
```

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
Exception Use #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>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>0</td>
</tr>
<tr>
<td>Process 1</td>
<td>10</td>
</tr>
<tr>
<td>Process 2</td>
<td>30</td>
</tr>
<tr>
<td>Process 1</td>
<td>60</td>
</tr>
<tr>
<td>Process 2</td>
<td>80</td>
</tr>
<tr>
<td>Process 2</td>
<td>110</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
Exception Use #2: Emulating Instructions
Enabled by illegal instruction exceptions

- `mul x1, x2, x3` is an instruction in the RISC-V ‘M’ extension (\(x1 \leftarrow x2 \times x3\))
  - If ‘M’ is not implemented, this is an illegal instruction
Exception Use #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?
Exception Use #2: Emulating Instructions
Enabled by illegal instruction exceptions

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

- What happens if we run code from an RV32IM machine on an RV32I machine?
  - mul causes an illegal instruction exception
Exception Use #2: Emulating Instructions
Enabled by illegal instruction exceptions

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

- What happens if we run code from an RV32IM machine on an RV32I machine?
  - `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

Process 1

Illegal instruction exception

Process 1 code:

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

November 5, 2019  
MIT 6.004 Fall 2019  
L17-17
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
...

November 5, 2019
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

Kernel

Illegal instruction exception

Save state of process 1
Emulate a multiply instruction in software (e.g., by repeated addition)

Process 1 code:
...
add a3, a2, a1
mul a4, a3, a2
xor a5, a4, a3
...

November 5, 2019
MIT 6.004 Fall 2019
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

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
Emulating Unsupported Instructions

Process running in CPU

Process 1 code:

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

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
Emulating Unsupported Instructions

Process running in CPU

Process 1 code:
...
add a3, a2, a1
mul a4, a3, a2
xor a5, a4, a3
...

Time

Process 1
Kernel
Process 1

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

Illegal instruction exception
Emulating Unsupported Instructions

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

Time

Process 1 Kernel Process 1

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

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

- Result: Program believes it is executing in a RV32IM processor, when it’s actually running in a RV32I
  - Any drawback? Much slower than a hardware multiply
Exception Use #3: System Calls

- The OS kernel lets processes invoke system services (e.g., access files) via **system calls**.
Exception Use #3: System Calls

- The OS kernel lets processes invoke system services (e.g., access files) via system calls.

- Processes invoke system calls by executing a special instruction that causes an exception (e.g., ecall in RISC-V).
Typical System Calls
Typical System Calls

- Accessing files (sys_open/close/read/write/...)

November 5, 2019

MIT 6.004 Fall 2019
Typical System Calls

- Accessing files (sys_open/close/read/write/...)
- Using network connections (sys_bind/listen/accept/...)

November 5, 2019
Typical System Calls

- Accessing files (sys_open/close/read/write/...)
- Using network connections (sys_bind/listen/accept/...)
- Managing memory (sys_mmap/munmap/mprotect/...)

November 5, 2019
Typical System Calls

- Accessing files (sys_open/close/read/write/...)
- Using network connections (sys_bind/listen/accept/...)
- Managing memory (sys_mmap/munmap/mprotect/...)

November 5, 2019  MIT 6.004 Fall 2019
Typical System Calls

- Accessing files (sys_open/close/read/write/...)
- Using network connections (sys_bind/listen/accept/...)
- Managing memory (sys_mmap/munmap/mprotect/...)
- Getting information about the system or process (sys_gettime/getpid/getuid/...)

November 5, 2019
MIT 6.004 Fall 2019
Typical System Calls

- Accessing files (sys_open/close/read/write/...)
- Using network connections (sys_bind/listen/accept/...)
- Managing memory (sys_mmap/munmap/mprotect/...)
- Getting information about the system or process (sys_gettime/getpid/getuid/...)
- Waiting for a certain event (sys_wait/sleep/yield/...
Typical System Calls

- Accessing files (sys_open/close/read/write/...)
- Using network connections (sys_bind/listen/accept/...)
- Managing memory (sys_mmap/munmap/mprotect/...)
- Getting information about the system or process (sys_gettime/getpid/getuid/...)
- Waiting for a certain event (sys_wait/sleep/yield...)
- Creating and interrupting other processes (sys_fork/exec/kill/...)

November 5, 2019
MIT 6.004 Fall 2019
Typical System Calls

- Accessing files (sys_open/close/read/write/...)
- Using network connections (sys_bind/listen/accept/...)
- Managing memory (sys_mmap/munmap/mprotect/...)
- Getting information about the system or process (sys_gettime/getpid/getuid/...)
- Waiting for a certain event (sys_wait/sleep/yield...)
- Creating and interrupting other processes (sys_fork/exec/kill/...)
- ... and many more!
Typical System Calls

- Accessing files (sys_open/close/read/write/...)
- Using network connections (sys_bind/listen/accept/...)
- Managing memory (sys_mmap/munmap/mprotect/...)
- Getting information about the system or process (sys_gettime/getpid/getuid/...)
- Waiting for a certain event (sys_wait/sleep/yield/...)
- Creating and interrupting other processes (sys_fork/exec/kill/...)
- ... and many more!
Typical System Calls

- Accessing files (sys_open/close/read/write/...)
- Using network connections (sys_bind/listen/accept/...)
- Managing memory (sys_mmap/munmap/mprotect/...)
- Getting information about the system or process (sys_gettime/getpid/getuid/...)
- Waiting for a certain event (sys_wait/sleep/yield/...)
- Creating and interrupting other processes (sys_fork/exec/kill/...)
- ... and many more!

Programs rarely invoke system calls directly. Instead, they are used by library/language routines.
Typical System Calls

- Accessing files (sys_open/close/read/write/...)
- Using network connections (sys_bind/listen/accept/...)
- Managing memory (sys_mmap/munmap/mprotect/...)
- Getting information about the system or process (sys_gettime/getpid/getuid/...)
- Waiting for a certain event (sys_wait/sleep/yield/...)
- Creating and interrupting other processes (sys_fork/exec/kill/...)
- ... and many more!

- Programs rarely invoke system calls directly. Instead, they are used by library/language routines
- Some of these system calls may block the process!
OS maintains a list of all processes and their status \{ready, executing, waiting\}
Process Life Cycle: The Full Picture

- OS maintains a list of all processes and their status \{ready, executing, waiting\}
  - A process is scheduled to run for a specified amount of CPU time or until completion
OS maintains a list of all processes and their status \{ready, executing, waiting\}

- A process is scheduled to run for a specified amount of CPU time or until completion
- If a process invokes a system call that cannot be satisfied immediately (e.g., a file read that needs to access disk), it is blocked and put in the waiting state
OS maintains a list of all processes and their status \{ready, executing, waiting\}

- A process is scheduled to run for a specified amount of CPU time or until completion
- If a process invokes a system call that cannot be satisfied immediately (e.g., a file read that needs to access disk), it is *blocked* and put in the *waiting* state
- When the waiting condition has been satisfied, the waiting process is woken up and put in the ready list
Exceptions in RISC-V

- 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.)
Exceptions in RISC-V

- 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
Exceptions in RISC-V

- 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
System Calls in RISC-V

- `ecall` instruction causes an exception, sets `mcause` CSR to a particular value
System Calls in RISC-V

- `ecall` instruction causes an exception, sets `mcause` CSR to a particular value
- ABI defines how process and kernel pass arguments and results
System Calls in RISC-V

- **ecall** instruction causes an exception, sets mcause CSR to a particular value

- ABI defines how process and kernel pass arguments and results

- Typically, similar conventions as a function call:
  - System call number in a7
  - Other arguments in a0-a6
  - Results in a0-a1 (or in memory)
  - All registers are preserved (treated as callee-saved)
System Calls in RISC-V

- `ecall` instruction causes an exception, sets `mcause` CSR to a particular value

- ABI defines how process and kernel pass arguments and results

- Typically, similar conventions as a function call:
  - System call number in `a7`
  - Other arguments in `a0-a6`
  - Results in `a0-a1` (or in memory)
  - All registers are preserved (treated as callee-saved)

More details in tomorrow’s recitation (demo of a tiny RISC-V OS!)
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
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