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
- 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
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)
- 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
- **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 **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

- 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
Virtual Machines Are Everywhere

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

- RISC-V process (quicksort) (RISC-V ISA)
- RISC-V emulator (sim.py) (Python Language)
- Python interpreter (CPython) (Linux ABI)
- Linux OS kernel (Implements a Linux-x86 VM)
- VirtualBox (Implements an x86 system VM)
- OS kernel (Win/Linux/MacOS/...) (Implements an OS-x86 VM)
- Hardware (e.g., your laptop) (Implements an x86 physical machine)
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

- 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.
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:
  - 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

- 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 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>Process 2</td>
<td></td>
<td></td>
<td></td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process 1</td>
<td>30</td>
<td></td>
<td></td>
<td>80</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>Process 2</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Exception Handler:**
- 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 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

- 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**

Process running in CPU

Process 1 code:

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

Time

- 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
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)
- 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
  // 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)
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) {
    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.)

- **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