CS250P: Computer Systems Architecture
Operating System Support
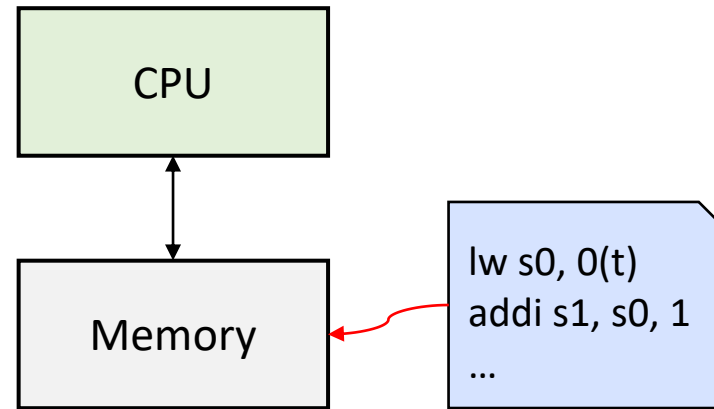
Sang-Woo Jun

Fall 2022

UCI

Computer architecture so far



```
lw s0, 0(t)
addi s1, s0, 1
...
```

What do we have to add to our processor to support a modern operating system?

Computer architecture so far
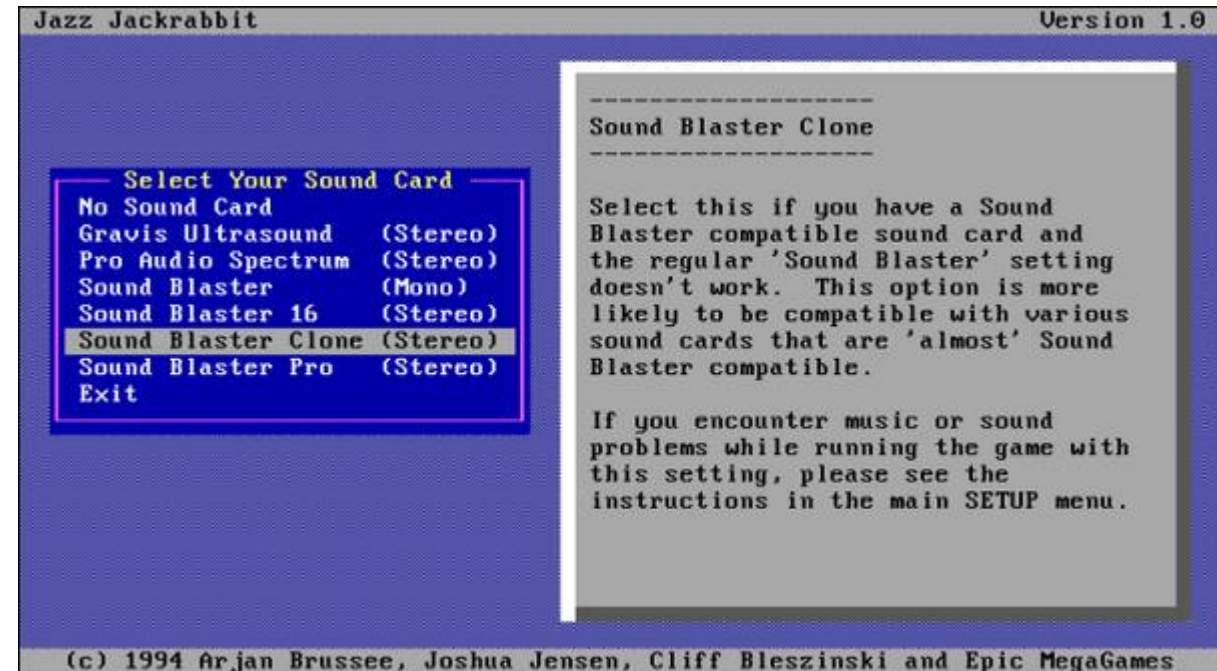
❑ Single program working with local memory

❑ What do we have to add to support a modern operating system?
- Isolation between processes
- System abstraction – Hide details about underlying hardware
- Resource management – CPU, memory, disk, network, …

Goal: support consistent abstraction to software
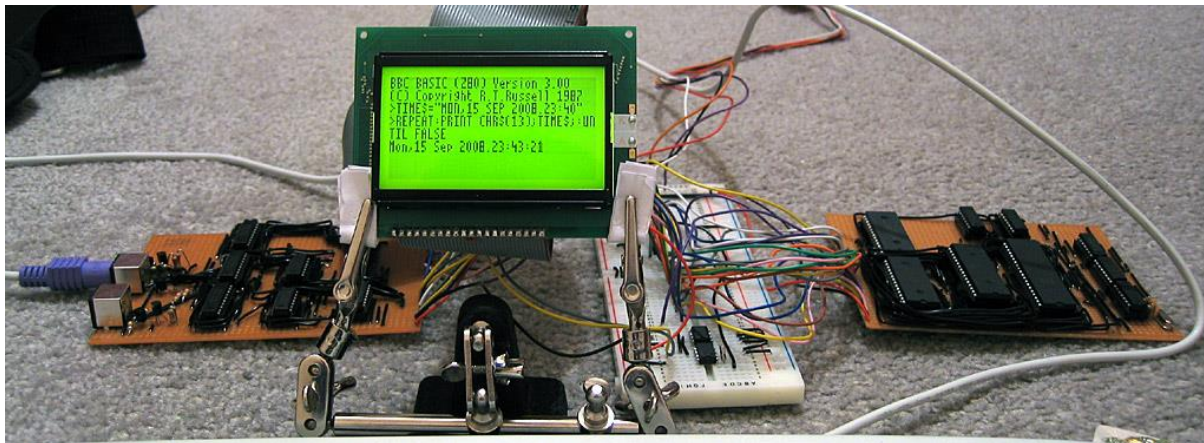Even with changing hardware, drivers, etc!

CPU

Memory

lw s0, 0(t)
addi s1, s0, 1
…
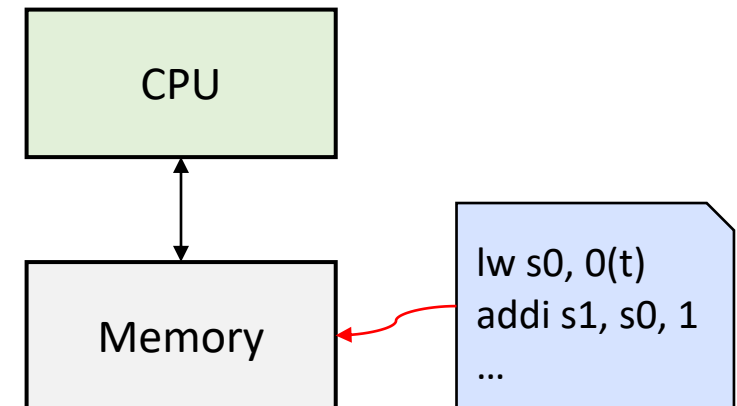
Aside: The old days

❑ Old personal operating systems (MS-DOS, CP/M, …) were very basic
- o The division between OS and user software was not strong
- o OS basically "jalr" into the user software, and "ret" out
- o User software had all access to hardware, including OS files on disk
- o Only one software running at a time!
- o Software failure -> System crash!

❑ Not much hardware abstraction
- o Each software had to handle each possible video, sound, etc hardware

```
Jazz Jackrabbit                              Version 1.0

  ┌── Select Your Sound Card ──┐    ┌─────────────────────────┐
  │ No Sound Card              │    │ ───────────────────     │
  │ Gravis Ultrasound  (Stereo)│    │ Sound Blaster Clone     │
  │ Pro Audio Spectrum (Stereo)│    │ ───────────────────     │
  │ Sound Blaster      (Mono)  │    │                         │
  │ Sound Blaster 16   (Stereo)│    │ Select this if you have a Sound │
  │ Sound Blaster Clone (Stereo)│   │ Blaster compatible sound card and │
  │ Sound Blaster Pro  (Stereo)│    │ the regular 'Sound Blaster' setting │
  │ Exit                       │    │ doesn't work.  This option is more │
  └────────────────────────────┘    │ likely to be compatible with various │
                                     │ sound cards that are 'almost' Sound │
                                     │ Blaster compatible. │
                                     │ │
                                     │ If you encounter music or sound │
                                     │ problems while running the game with │
                                     │ this setting, please see the │
                                     │ instructions in the main SETUP menu. │
                                     └─────────────────────────┘

  (c) 1994 Arjan Brussee, Joshua Jensen, Cliff Bleszinski and Epic MegaGames
```

Aside: The CP/M operating system (1974)

❑ Control Program/Monitor, created by Digital Research, Inc.
  - o Designed for Intel 8080, with less than 64 KiB of memory
  - o Massive popularity, massive influence to MS-DOS (1981)
    - A: B: C: device naming, "BIOS", AAAAAAAA.EXT naming scheme, etc survives until now

❑ Extremely simple O/S
  - o Still used/modified by hobbyists!

Aside: The CP/M operating system (1974)

❑ Once booted, the CCP command line is presented.

❑ When executing software, binary is loaded to low part of free memory, and OS simply jumps to that region
   - Always only one execution context (process)

"Basic Disk Operating System"

❑ User software interfaces with OS via BDOS
   - BDOS location is stored as a pointer in "Low storage"
   - Scheme allows contiguous memory for software regardless of memory capacity

❑ When done execution, simply returns to OS

Simple! Software has exclusive access to machine
OS is effectively just like a library – DOS was very similar

CP/M Memory Map
A typical 64K RAM system

CP/M System	
BIOS — Hardware Drivers for BDOS	FFFFh: Top of 64K memory
	F200h
BDOS — Operating System Functions	
	E400h
CCP — Command Line Interpreter	
Programs	DC00h: Start of CCP / End of free memory
TPA — Free Memory for Programs	
	0100h: Start of TPA
'Low Storage' — System Buffers & Parameters	
	0000h: Start of memory

https://obsolescence.wixsite.com/obsolescence/cpm-internals

Aside: Something new – multitasking

❑ Multiple tasks (processes) executing concurrently

o Multi-user systems, servers with multiple parallel workloads, services, GUI, …

❑ Memory usage becomes complicated with multitasking

o Two binaries cannot be loaded to same memory location, software can be loaded to arbitrary, possibly non-contiguous, locations

o Will have contention between processes for data memory locations

o We cannot use absolute addressing any more for jumps and data referencing!

o No longer simple address model with assumed exclusive access to memory

```
00000340 <main>:
 340:   fd010113            addi    sp,sp,-48
 344:   02112623            sw  ra,44(sp)
 348:   02812423            sw  s0,40(sp)
 34c:   03010413            addi    s0,sp,48
 350:   fe042623            sw  zero,-20(s0)
 354:   06c0006f            jal zero,3c0 <main+0x80>
```

Address "0x3c0" is encoded as literal.
Needs exclusive access guarantee
(At compile time?!)

Modern operating systems

❑ Modern operating systems support user process isolation

❑ The OS kernel provides a private address space to each process

1. Each process thinks it has exclusive access to contiguous memory
2. A process is not allowed to access the memory of other processes
3. No user process can access OS memory

❑ The OS kernel schedules processes into the CPU

o Each process is given a fraction of CPU time
o 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

Familiar concepts from OS classes!

Architectural support for operating systems

❑ Operating system must have different capabilities from user processes
   o Typical ISA defines two or more "privilege levels" (e.g., "user", and "supervisor")
   o Some instructions and registers that are only accessible for a process executing in supervisor mode
   o Typically, the very first process to execute is given supervisor privilege, and is responsible for spawning future user processes

❑ Interrupts and exceptions to transition from user to supervisor mode

❑ Virtual memory to provide private address spaces and abstract the storage resources of the machine
   o User processes executing LW/SW/etc access memory through a hardware virtual memory manager

Topics

❑ Privilege levels

❑ Interrupts and exceptions

❑ Virtual memory

Privilege levels in modern architectures

❑ RISC-V has three (or more) formally defined levels
  o Machine level, full access to all hardware after initial boot
  o Hypervisor level – For virtualization. Recently formally defined! (2022)
  o Supervisor level – For operating systems
  o User level – For applications

❑ x86 has "protection rings"
  o Typically only ring 0 and 3 are used
  o Additional ring -1 for hypervisors

❑ Each process/thread belongs on one level

Less privileged levels have more restrictions
- Cannot access some registers
- Can only access memory via virtual memory, not raw hardware
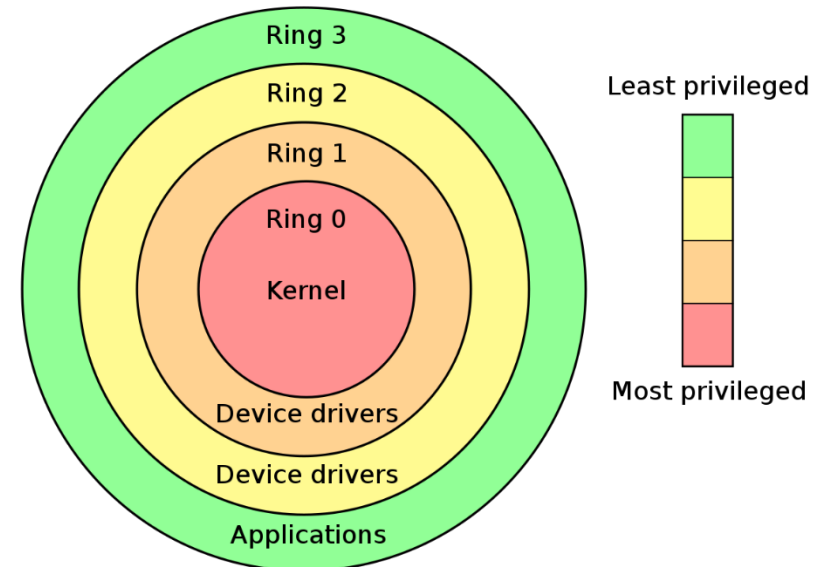
Example: RISC-V

❑ Special register, "mstatus" (for "machine status")
- o Among other information, stores the privilege level of the current process
- o Writing a new value to it can change the privilege level, but only machine mode is allowed to write to it
- o OS runs in machine mode, when user process must be spawned, it first spawns a kernel process which downgrades itself to user mode before jumping to actual user software

❑ Special ISA instructions to access the special registers
- o One of many "Control Status Register"
- o csrr, csrw instructions, only allowed in machine mode
- o There are many CSRs! Will mention more soon.

x86 typically has separate instructions for each privileged operation

Topics

❑ Privilege levels

❑ Interrupts and exceptions

❑ Virtual memory

Exceptions?

❑ Event that needs to be processed by the OS kernel.
  The event is usually unexpected or rare
    o Exceptions cause an exception handler in OS, in higher privilege

process

I_{i-1}

I_i

I_{i+1}

HI_1

HI_2

HI_n

exception
handler
(in OS kernel)
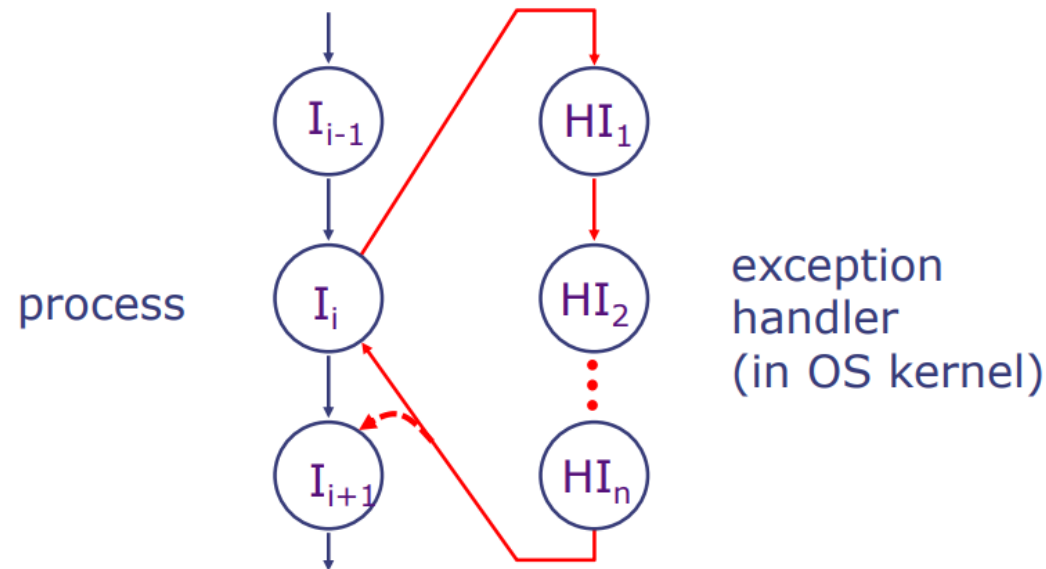
Typical terminology

❑ Exceptions: Usually events caused by the running process itself
  o Illegal memory access (SEGFAULT), divide-by-zero, system call, etc

❑ Interrupts: Usually events caused by the outside world
  o Timer, I/O completion, keystroke, etc

❑ Terminology is often used interchangeably…

Handling exceptions

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

❑ After the exception handler finishes, the processor:
- o Returns control to the user process at instruction I_i
- o User process is oblivious to the interrupt

❑ If an interrupt is due to an illegal operation, the OS aborts the process
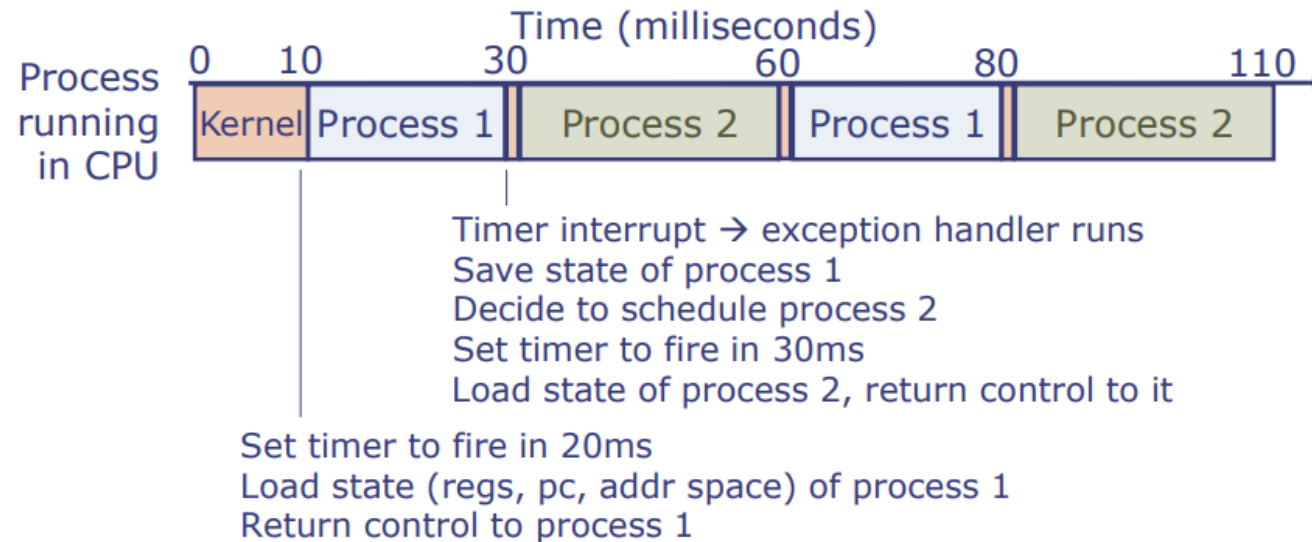- o e.g., SEGFAULT

Handling exceptions

❑ The operating system is responsible for telling the processor how to handle each type of exception

    o Typically via a table of pointers in main memory, each corresponding to a particular exception type

                                                         "Machine Trap Vector"

    o A special register is set with a pointer to the table in memory ("mtvec" for RISC-V, "IDTR" for x86)

        "Interrupt Descriptor Table Register"

❑ For each exception, the CPU transparently consults this register, reads the table, and jumps to the correct handler

       No software involved in this process. Hardware!

INT_NUM	Short Description PM [clarification needed]
0x00	Division by zero
0x01	Single-step interrupt (see trap flag)
0x02	NMI
0x03	Breakpoint (callable by the special 1-byte instruction 0xCC, used by debuggers)
0x04	Overflow
0x05	Bounds
0x06	Invalid Opcode
0x07	Coprocessor not available
0x08	Double fault
0x09	Coprocessor Segment Overrun (386 or earlier only)

Exception use #1: CPU scheduling

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

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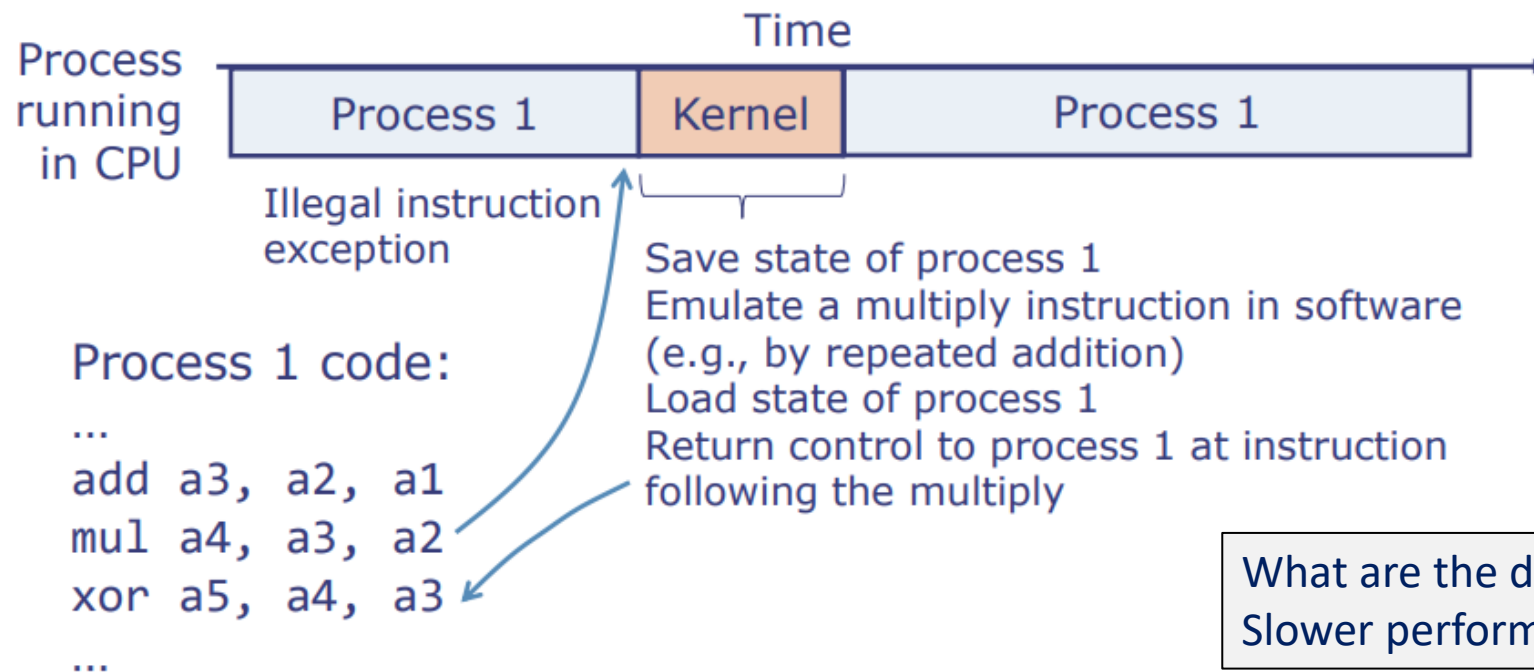 o Kernel sets timer, which raises an interrupt after a specified time

Time (milliseconds)

| 0 | 10 | | 30 | 60 | 80 | 110 |

Process running in CPU

| Kernel | Process 1 | Process 2 | Process 1 | Process 2 |

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

❑ mul x1, x2, x3 is an instruction in the RISC-V 'M' extension (x1 = x2 * x3)
   o If 'M' is not implemented, this is an illegal instruction

❑ What happens if we run code for an RV32IM ISA on an RV32I machine?
   o mul causes an illegal instruction exception
   o The exception handler can take over and abort the process... but it can also emulate the instruction!

Emulating Unsupported Instructions

❑ Program believes it is executing in a RV32IM processor, when it's actually running in a RV32I

❑ The IBM System/360 line of machines used this method to build cheap machines that adhere to ISA

Time

Process running in CPU

| 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
...
```

What are the downsides?
Slower performance compared to HW implementation!

Exception Use #3: System Calls

❑ User process has no access to raw hardware resources (not even the keyboard)
   - o User process communicates with the OS via system calls (and other methods)
   - o The syscall instruction (SYSCALL in x86, ecall in RISC-V) results in a machine-mode exception that can handle the request
     - Arguments and return values following familiar function call conventions
   - o Aside: x86 used to assign a special number in the interrupt table (0x80) to handle syscalls. This is still technically supported, but discouraged
     - "int 0x80" vs. "syscall"

Exception details in RISC-V

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

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

System call details for RISC-V

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

❑ Application Binary Interface (ABI) convention defines how process and kernel pass arguments and results
  - o Typically, similar conventions as a function call:
  - o System call number in a7
  - o Other arguments in a0-a6
  - o Results in a0-a1 (or in memory)
  - o **All** registers are preserved (treated as callee-saved) Why is this?

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!

Hello world using x86 system calls

❑ Old example using using int 0x80

```
section .data
    msg db     "hello, world!" ; defining the message

section .text
    global _start    ; this is for the linker

_start:
    mov    rax, 4       ; Select system call: 4 = sys_write
    mov    rbx, 1       ; First argument:  1 = stdout
    mov    rcx, msg     ; Second argument: pointer to message
    mov    rdx, 13      ; Third argument:  number of bytes to be written

    int 0x80            ; perform the chosen system call (pass variables
                        ; inside registers to the kernel and it will do
                        ; the rest)

    mov    rax, 1       ; 1 = sys_exit
    mov    rbx, 0       ; exit status = 0

    int 0x80              ; again, perform system call, this time sys_exit
```
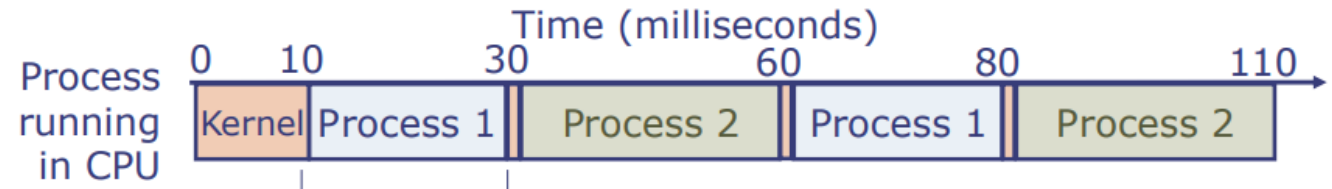
So far…

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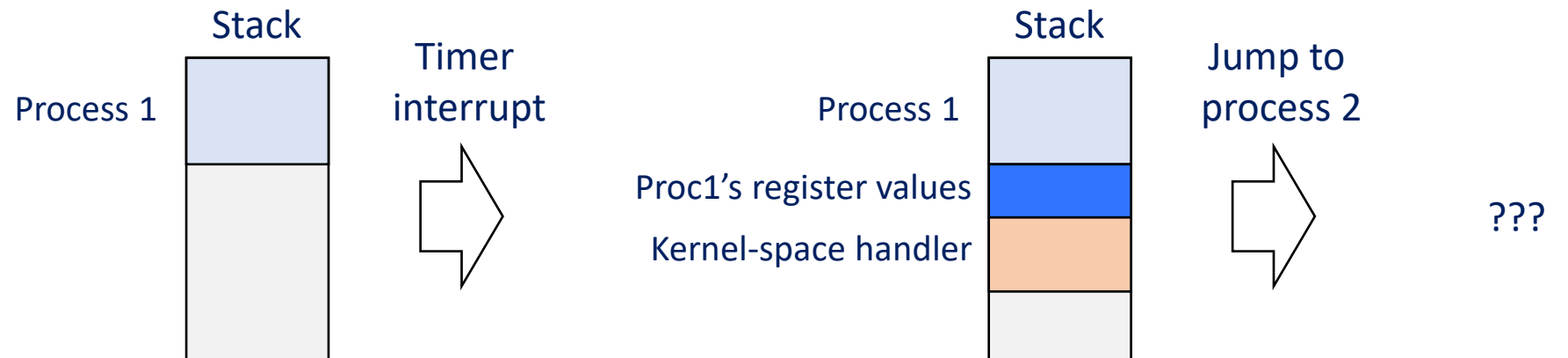 o Resource management: OS controls how processes share hardware resources (CPU, memory, disk, etc.)

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

Context switching

Time (milliseconds)

Process running in CPU: Kernel | Process 1 | Process 2 | Process 1 | Process 2
(0, 10, 30, 60, 80, 110)

❑ On a multitasked system, a processor cycles over multiple process, executing them in small increments

❑ Simply jumping between where we left off does not ensure correctness!
  o When we jumped into the kernel-space interrupt handler, the register values are stored in the stack, so they can be reclaimed after exiting the interrupt handler
    • Remember, all registers are callee-saved in this situation because user process is unaware
  o How do we know where to get the next register values? e.g., stack pointer?

Stack

Process 1

Timer interrupt

Stack

Process 1

Proc1's register values

Kernel-space handler

Jump to process 2

???

Context switching

❑ Context: The state of the process or thread which must be saved and restored for seamless multiprocessing
  o So far: PC, entirety of the register file (including the stack pointer, x2)
  o In reality, a lot more information including virtual memory state

❑ Context switching: Storing the context of the current process and loading the context of a new process
  o The processor is (conceptually) oblivious to processes
    • The concept of processes does not exist at the processor level, it's just executing instructions
  o Like loading the same body (processor) with a different soul (context)

Context switching – Process Control Block

❑ Context information is managed in the OS via a construct called the Process Control Block (PCB)

- o Again, the processor is completely unaware of this
- o Stores information including the process ID, context state (register values, etc), meta-information for scheduling control (when was it last scheduled? etc)
- o An array of PCBs, one element per process/thread
- o Operating system topic! Only introduced here to connect the dots between architecture and OS

❑ In Linux, PCB is "struct task_struct"

Context switching – Process Control Block

❑ The OS software (not the processor hardware) is responsible for context switching, including

- o Storing the current context to the appropriate PCB
- o Deciding which process to execute (and for how long)
- o Loading the next context from the PCB to the hardware registers
- o Resuming the next process
  - • "Resuming" because it is currently suspended while the current process was executing
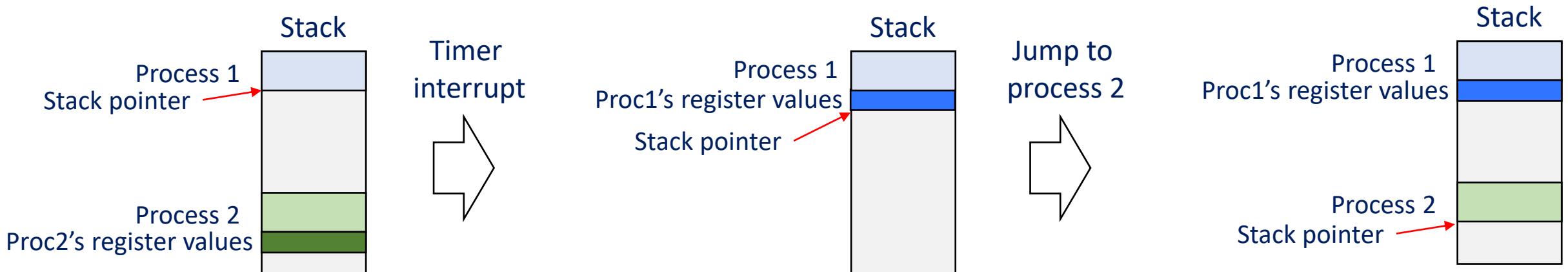
Aside:
Hardware vs. software context switching

❑ Some processor designs support hardware handling of context switching operations (e.g., x86)

- o CALL or JMP under special circumstances evoke hardware handling of context switching

- o Processor hardware automatically read/writes the PCB if it is in a specific format

❑ Unfortunately, most mainstream OSs don't use it

- o High overhead as some of the hardware-defined context includes some values that are no longer useful in modern OSs
  - e.g., segment registers, will introduce soon

- o Some newer registers are not automatically restored
 - e.g, floating point

Modern processors often omit this feature in 64-bit mode

Aside: x86 way of creating user-level processes

❑ x86 doesn't provide a way to explicitly switch to user level
  o Instead, we write code that pretends to return from an interrupt, back into user level
  o Allocate stack space in memory, and populate it with a return address, stack pointer, thread information, … pretending to be a user level process whose interrupt is being handled
  o Call "IRET" which reads the stack, and "returns" to user level operation

System boot process

❑ Our RV32I processor, when powered on, starts executing from address 0
- o When powered on, memory is blank… How does OS get there?
- o Short answer: Firmware (e.g., BIOS, UEFI)

❑ Firmware is usually located in address 0
- o Special ROM/EEPROM/etc hardwired to map to address zero
- o On power on, CPU executes the firmware to load a small "bootloader" from storage and loads it to a special address, and transfers control
- o Bootloader loads the actual OS kernel from storage to memory and transfers control

RAM

Kernel

ROM

CPU

addr 0

Firmware
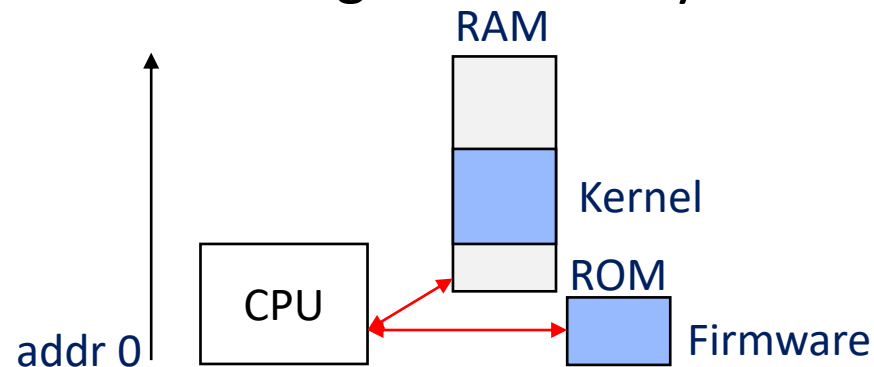
Why bootloader?

❑ BIOS (Basic Input/Output System) treated the first sector (512 Bytes) of a storage medium specially (MBR, "Master Boot Record")

- o BIOS loaded the MBR of the first HDD to memory and executed it
- o Bootloader had to fit in 512 Bytes, and is responsible for finding/loading the OS kernel and executing it
- o Due to complexities of file systems, etc, sometimes two-level bootloaders were used (e.g., GRUB on Linux)
  - Bootloader loads the second bootloader and executes it, which in turn loads the whole kernel

❑ UEFI (Unified Extensible Firmware Interface) doesn't use MBR, instead stores bootloaders in a special UEFI partition

- o Still not the whole kernel!