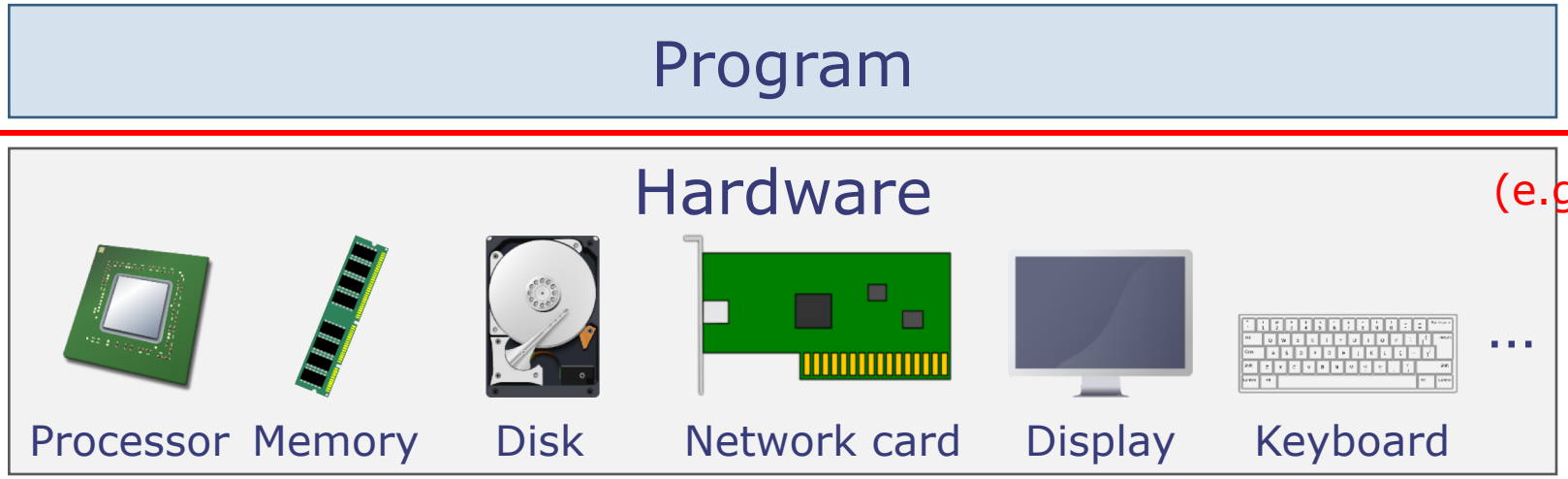
Operating Systems

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.

Quiz 2 tonight 7:30-9:30pm
Room 50-340

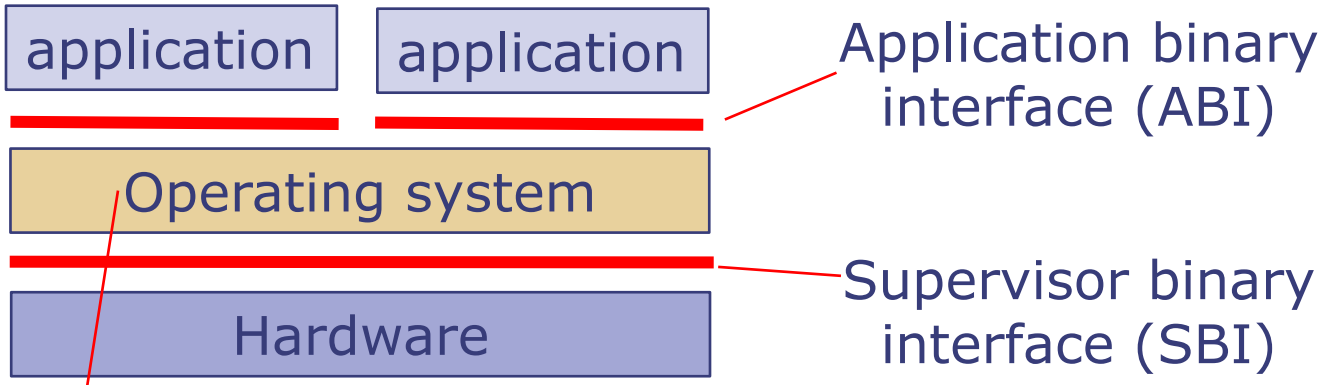
6.004 So Far: Single-User Machines



- 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

This picture is too simplistic.

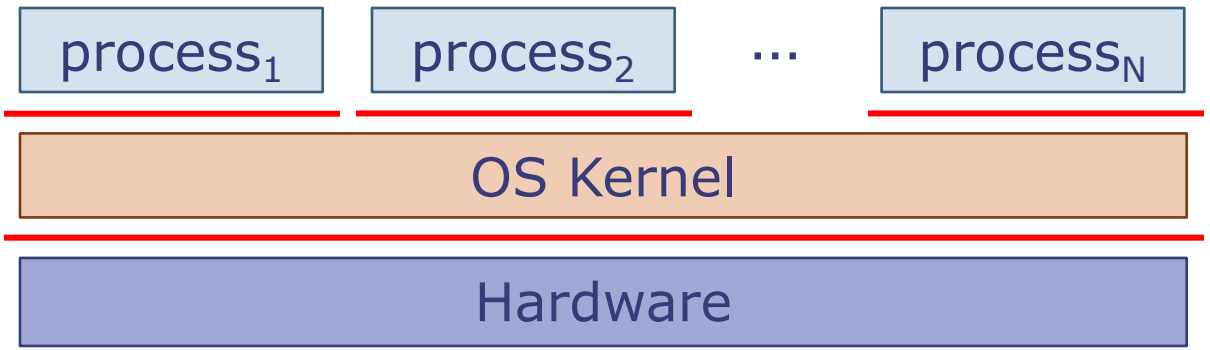
Operating Systems



- Modern computers cannot function without some resident programs ("services"), which are shared by all users
  - Services: programs for accessing I/O devices like, keyboard, mouse, display, disks, printers, network, ...
  - I/O devices have very long access latency as compared to registers and memories (milliseconds as opposed to microseconds)
  - It is common to employ "multiprogramming" and switch to another program while waiting for an I/O operation to complete

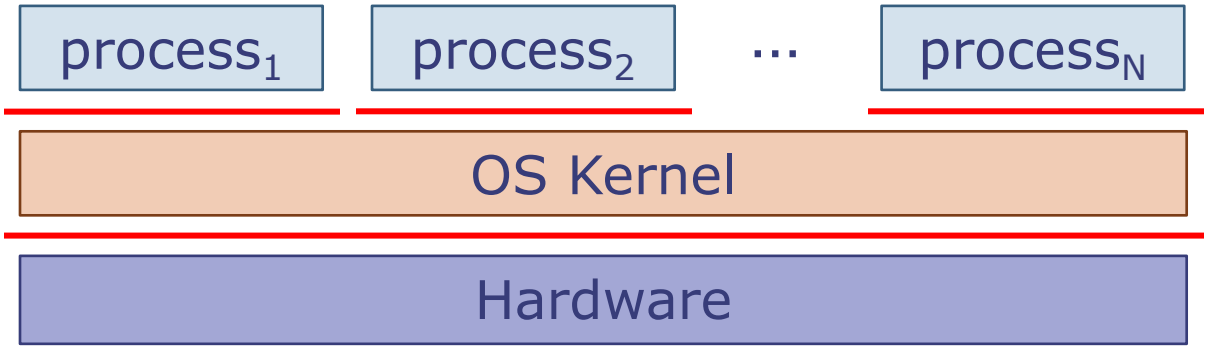
Supporting Operating Systems creates a host of new problems in processor design

Nomenclature: Process, Program, Processor



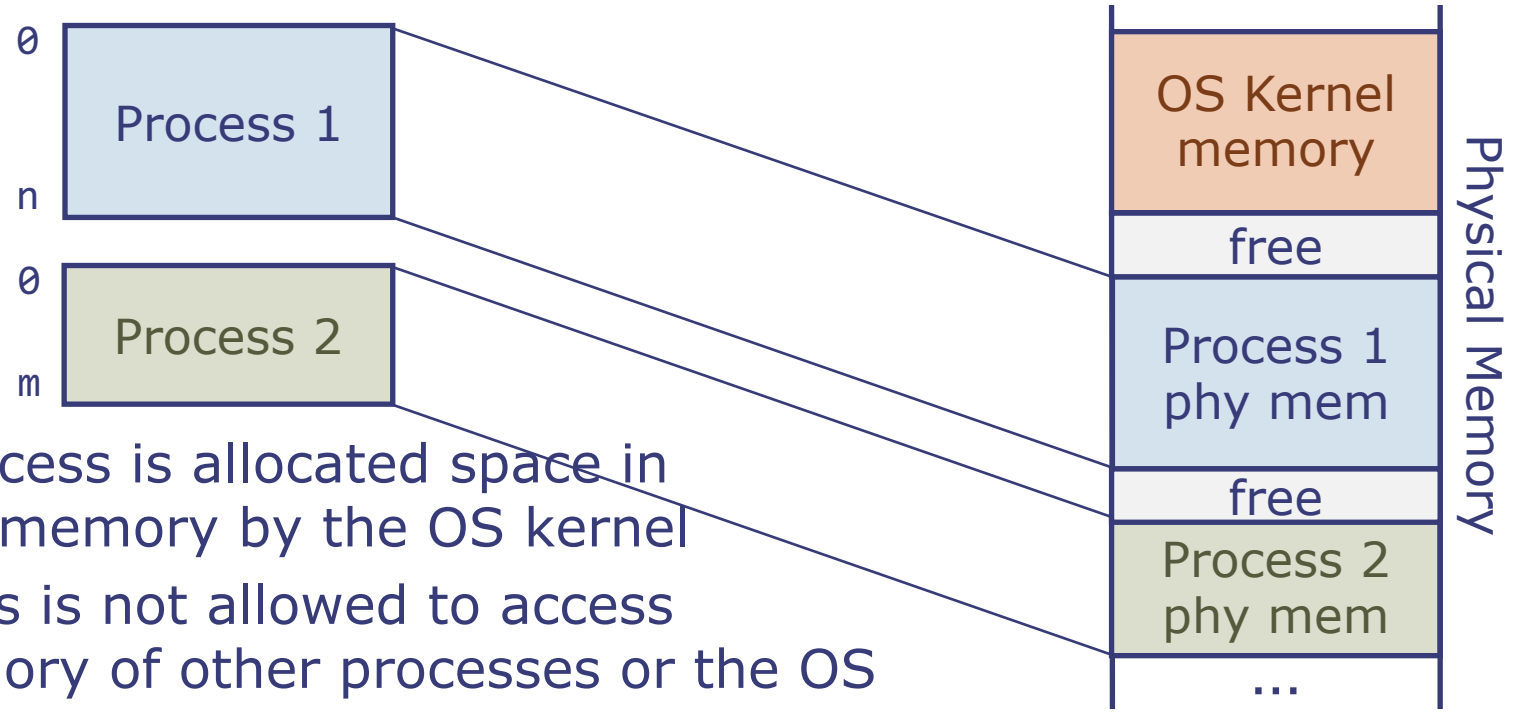
- A program is a collection of instructions and data
- A processor is the hardware to execute programs
- A process is an instance of a program that is being executed
  - Includes program, processor state (registers, memory), and I/O device state
- The OS Kernel is a special process with extra privileges

Goals of Operating Systems



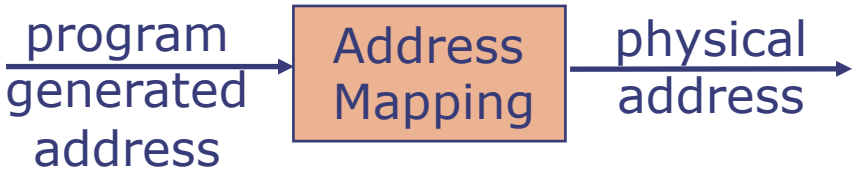
- Protection and privacy: Processes cannot access each other's data
- Abstraction: OS hides details of underlying hardware
  - e.g., processes open and access files instead of issuing raw commands to the disk
- Resource management: OS controls, i.e., allocates and schedules, how processes share hardware (CPU, memory, disk, etc.)

Private Address Space per Process



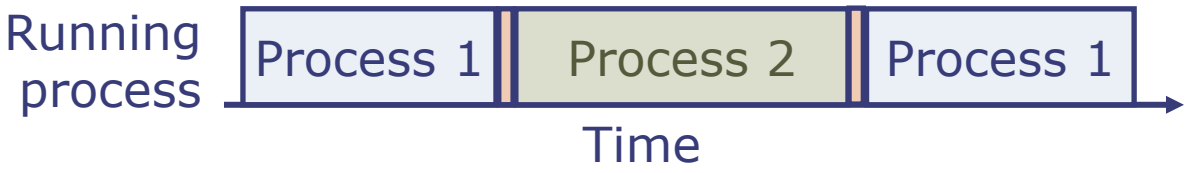
- Each process is allocated space in physical memory by the OS kernel
- A process is not allowed to access the memory of other processes or the OS
- It is convenient if the program and data addresses don't depend upon where the process is allocated in the memory
  - requires dynamic address translation

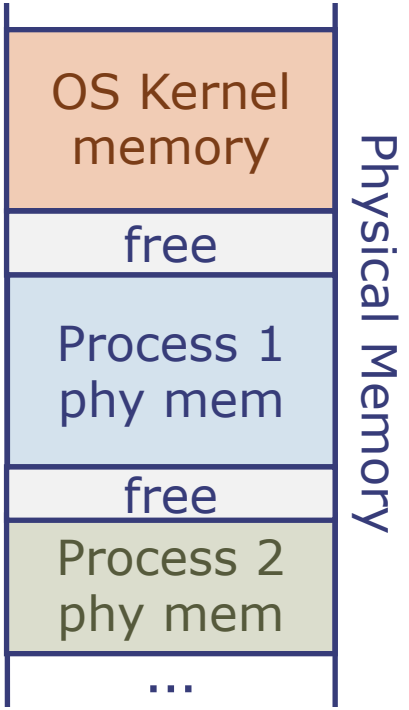
next lecture



Scheduling

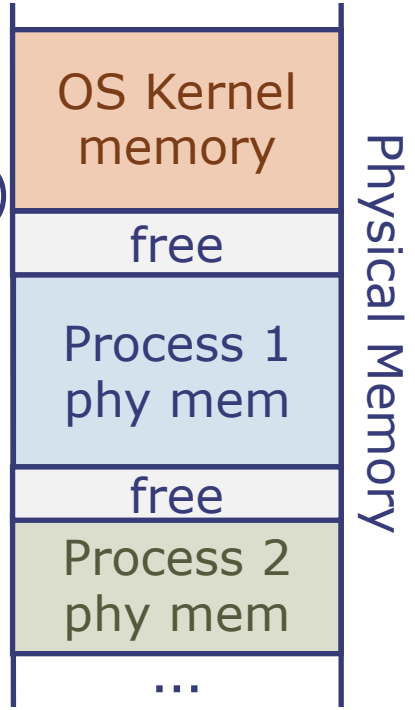
- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed





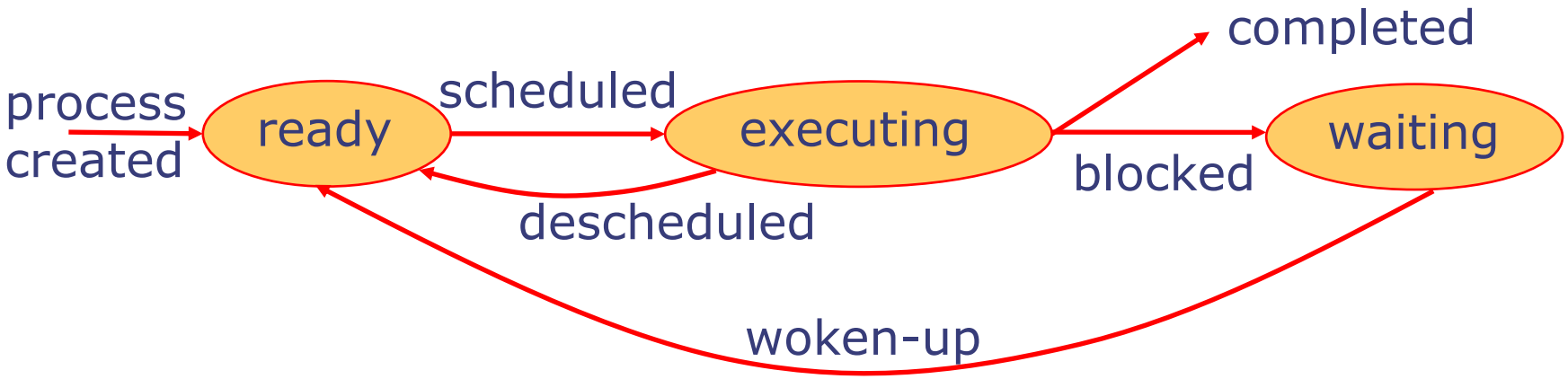
System Calls

- The OS kernel lets processes invoke system services (e.g., access files or network sockets) via system calls (special instruction)
  - Key Board input
  - Mouse
  - Display
  - File Access
- 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



Lab 8

Process Life Cycle



- 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

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

Today

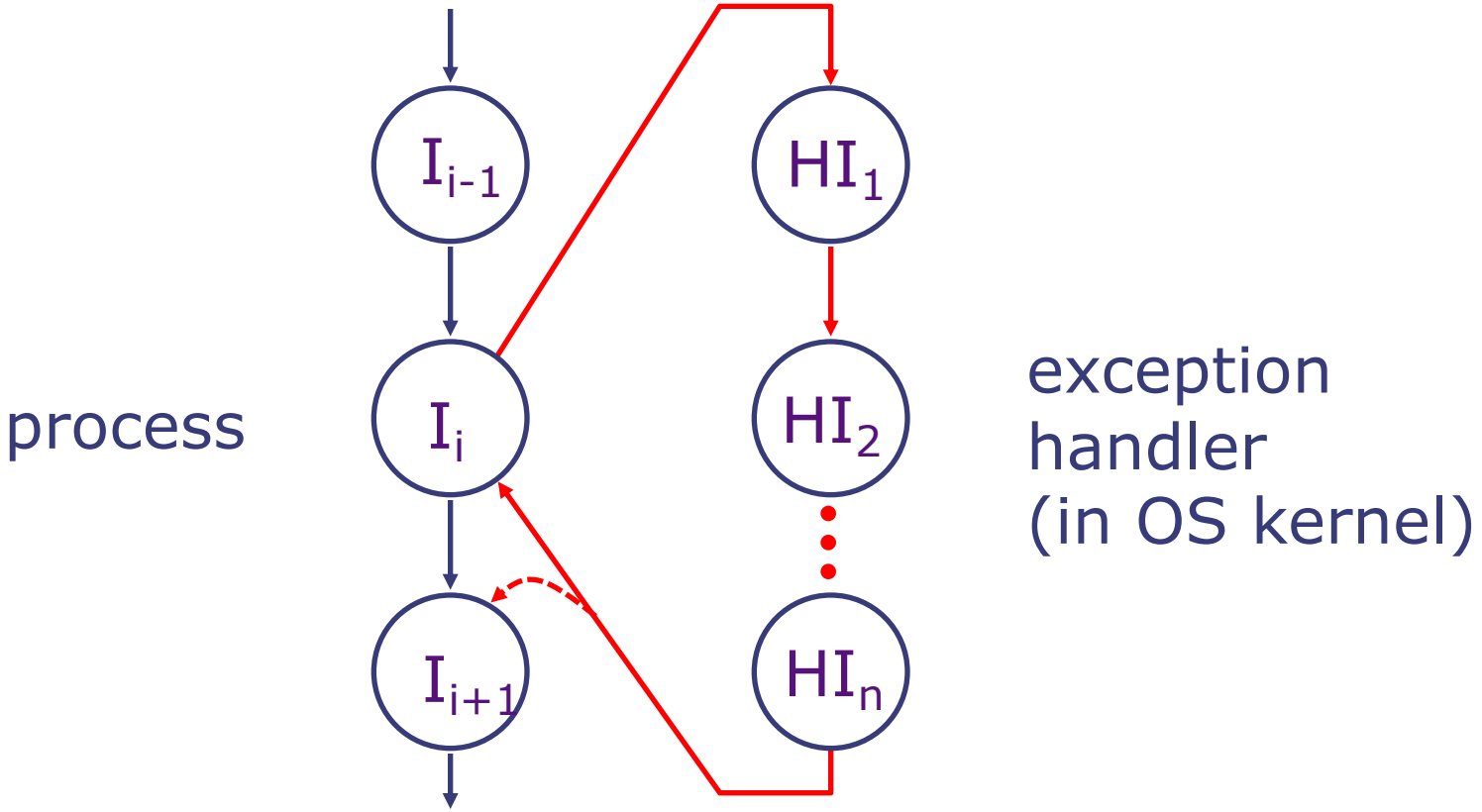
- Virtual memory to provide private address spaces and abstract the storage resources of the machine

Next lecture

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)

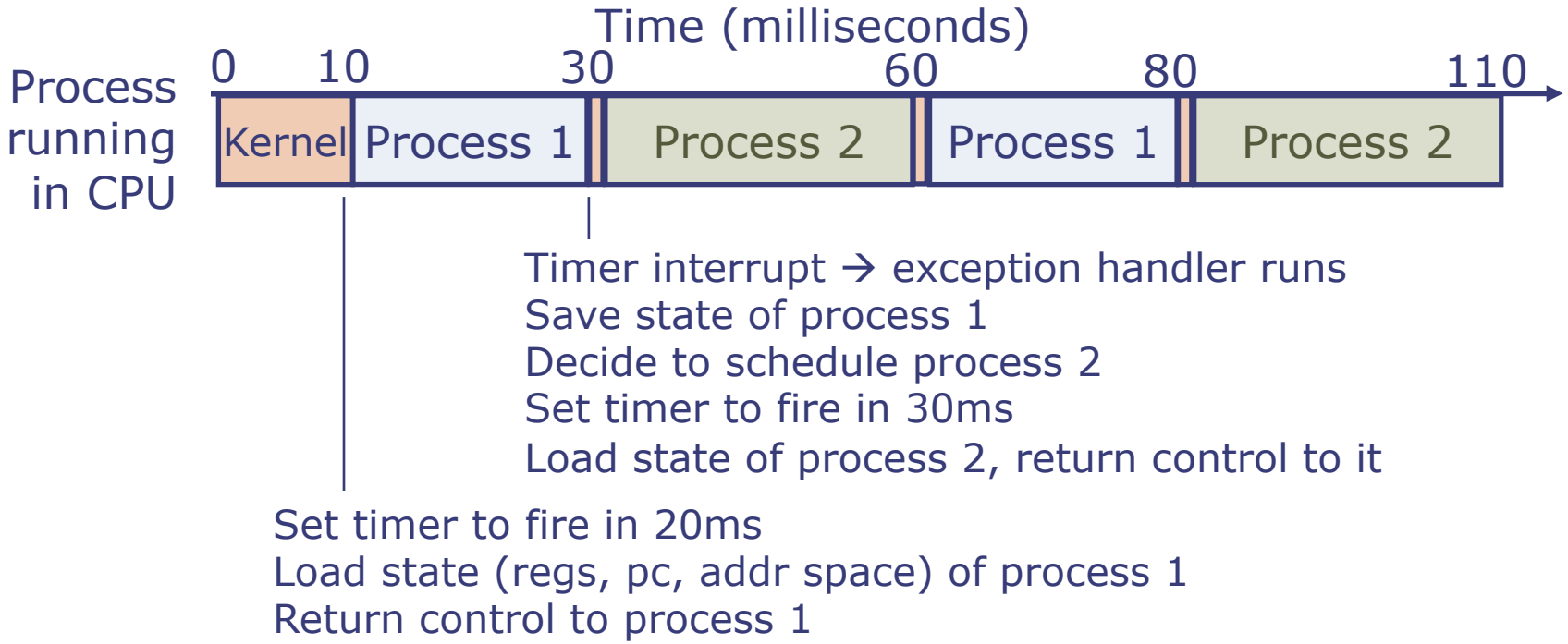
- We use exception to encompass both types of events, and use synchronous exception for synchronous events

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

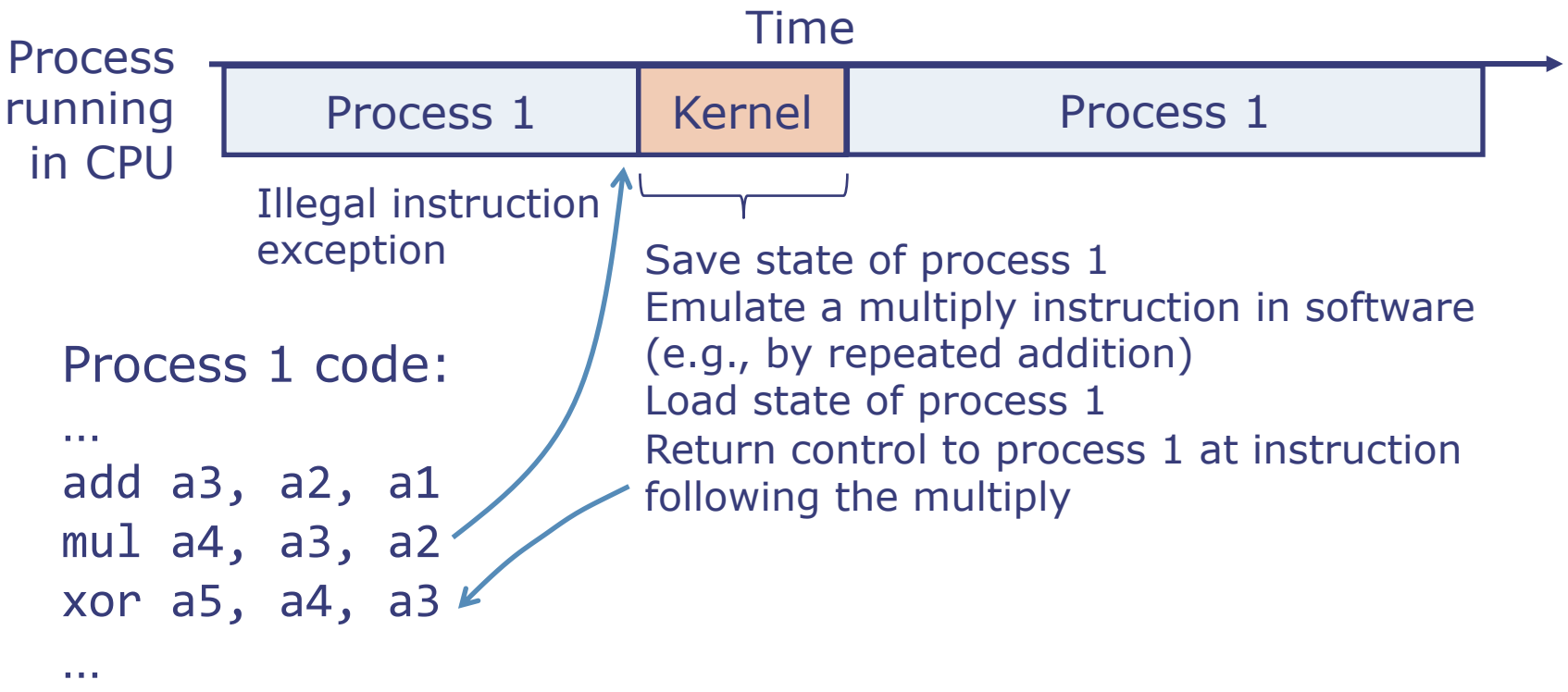
- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time



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

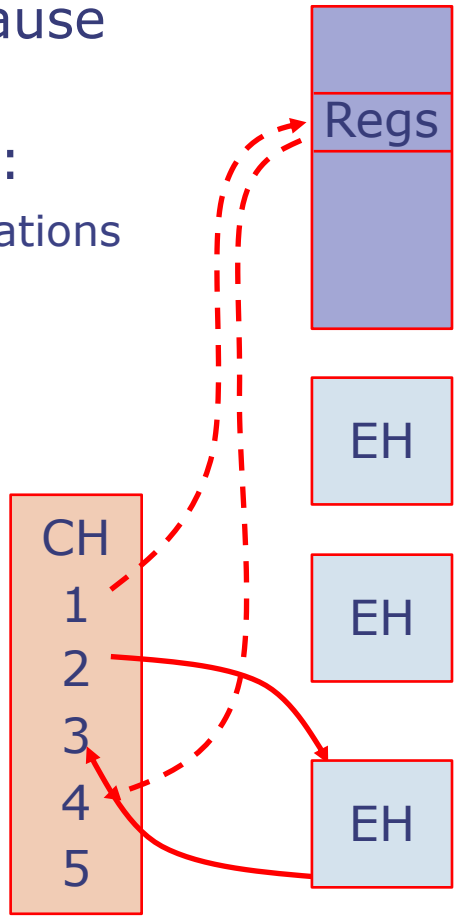
RISC-V Exception Handling

- 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, ...)
  - `mtvec`: address of the exception handler
  - `mstatus`: status bits (privilege mode, interrupts enabled, ...)
  - ...
- 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

Executing a privileged instruction in user mode causes an exception; a user process 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 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
  // 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, mepc
  sw t1, 0(x1)
```

Common Exception Handler *cont.*
Calling EH_Dispatcher and returning

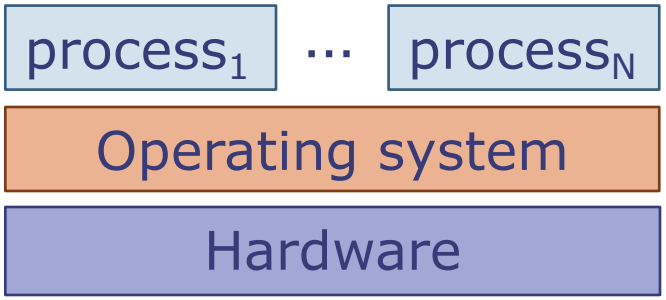
```
common_handler:
  ... // we have saved the state of the process
  //pass interrupted process state and cause to eh_dispather
  mv a0, x1 // arg 0: pointer interrupted process state
  csrr a1, mcause // arg 1: exception cause
  lw sp, kernelSp // use the kernel's stack
  // calls the appropriate handler
  jal eh_dispatcher
  // 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"

```
typedef struct {
  int pc;
  int regs[31]; // reg 0 is not passed
  …
} 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