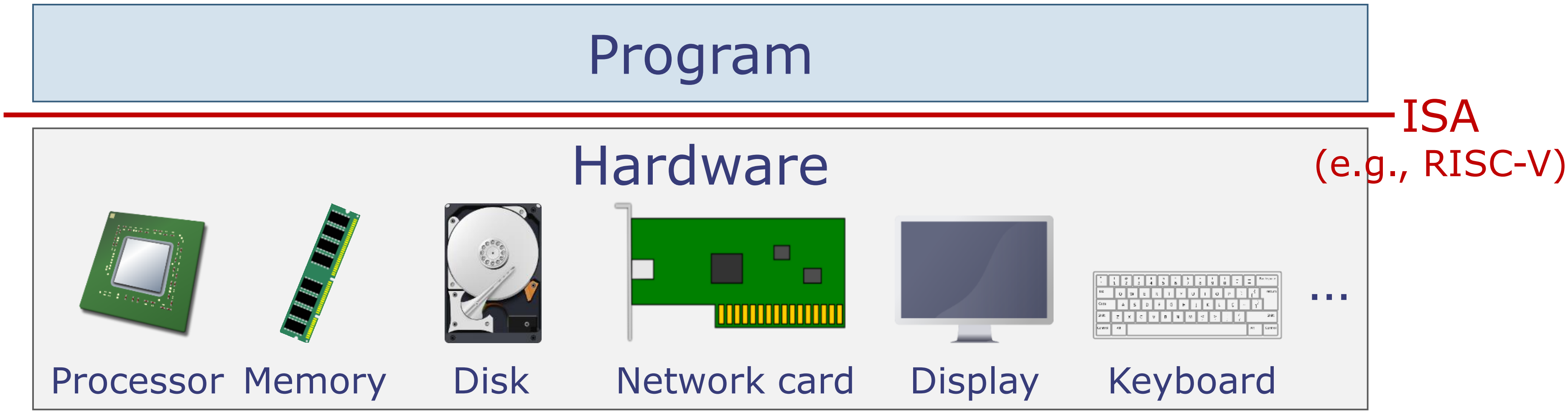
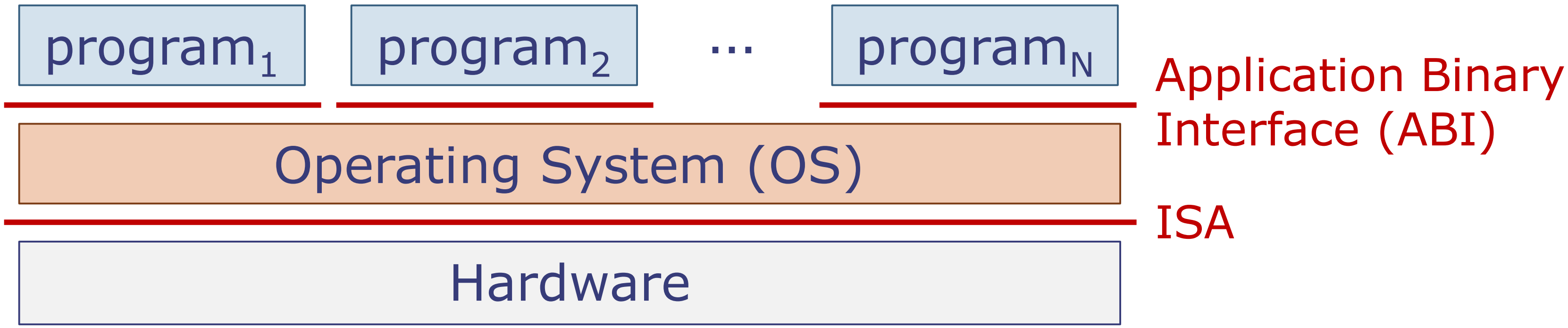
Operating Systems: Virtual Machines & Exceptions

6.004 So Far: Single-User Machines



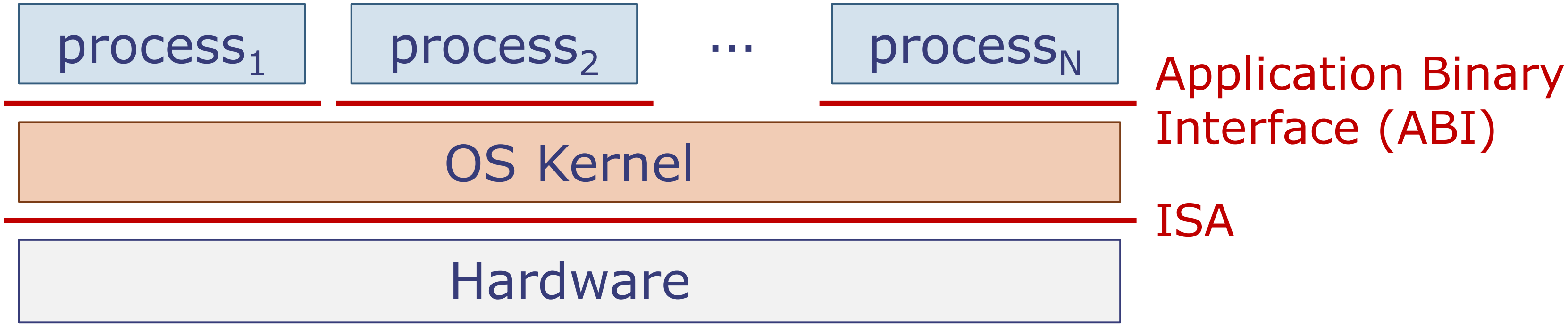
- Hardware executes a single program
- This program has direct and complete access to all hardware resources in the machine
- The instruction set architecture (ISA) is the interface between software and hardware
- Most computer systems don’t work like this!

Operating Systems



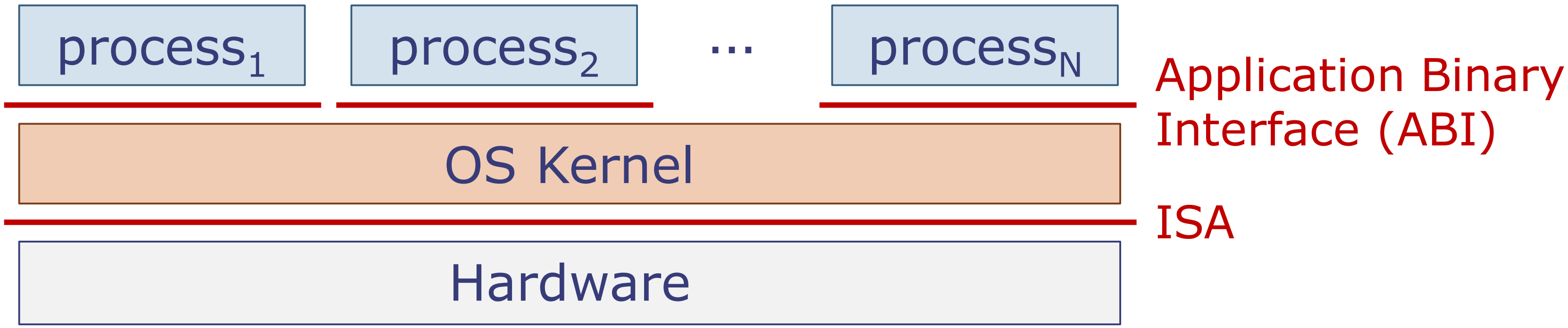
- Multiple executing programs share the machine
- Each executing program does not have direct access to hardware resources
- Instead, an operating system (OS) controls these programs and how they share hardware resources
  - Only the OS has unrestricted access to hardware
- The application binary interface (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 + state (registers, memory, and other resources)
- The OS Kernel is a process with special privileges

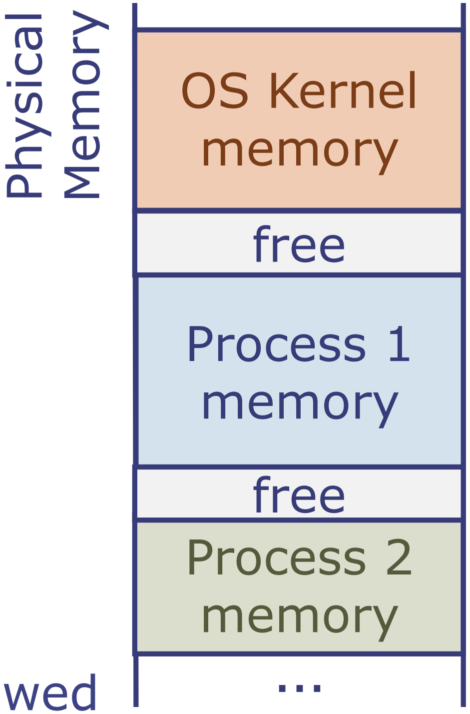
Goals of Operating Systems

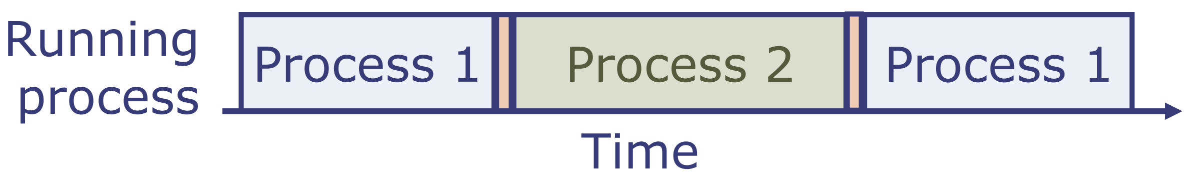


- Protection and privacy: Processes cannot access each other's data
- Abstraction: OS hides details of underlying hardware
  - e.g., processes open and access files instead of issuing raw commands to the disk
- Resource management: OS controls how processes share hardware (CPU, memory, disk, etc.)

Operating Systems: The Big Picture

- 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

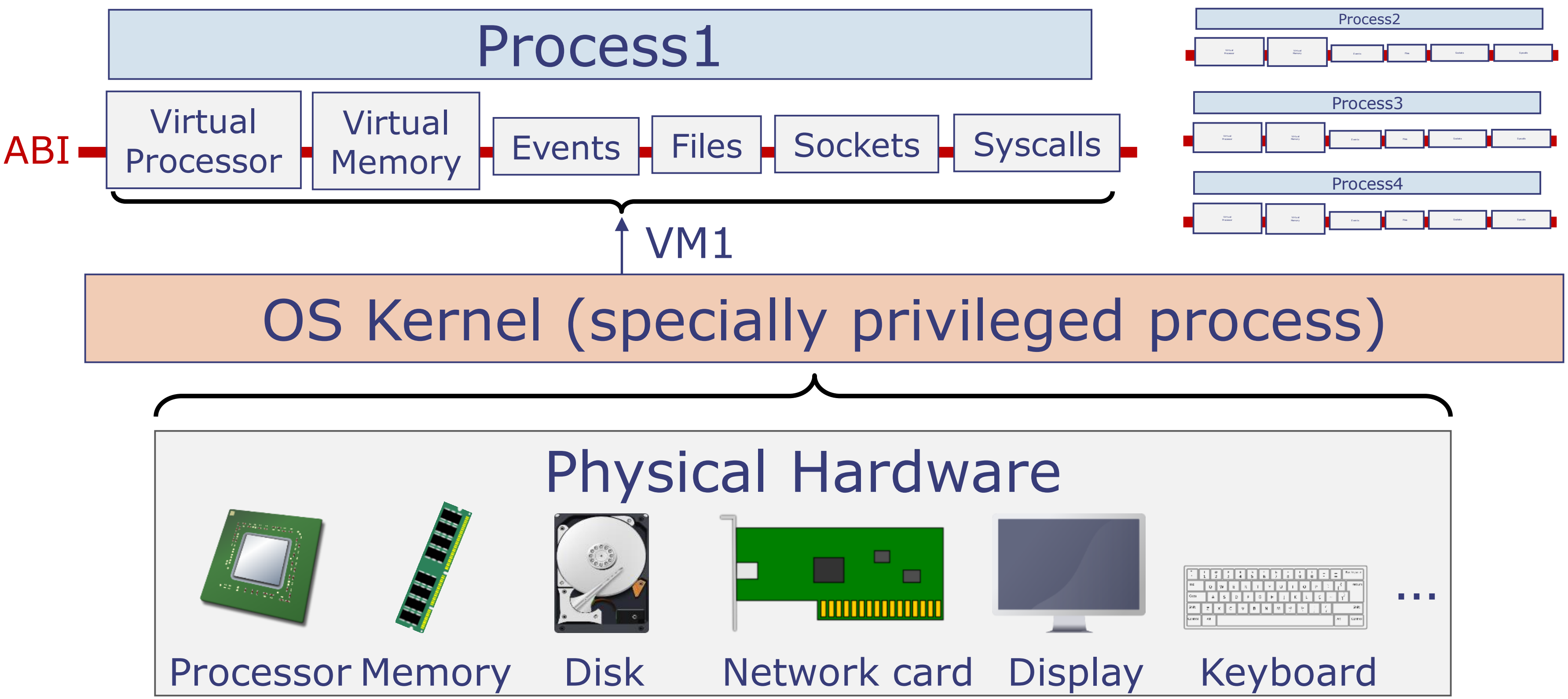




- The OS kernel lets processes invoke system services (e.g., access files or network sockets) via system calls

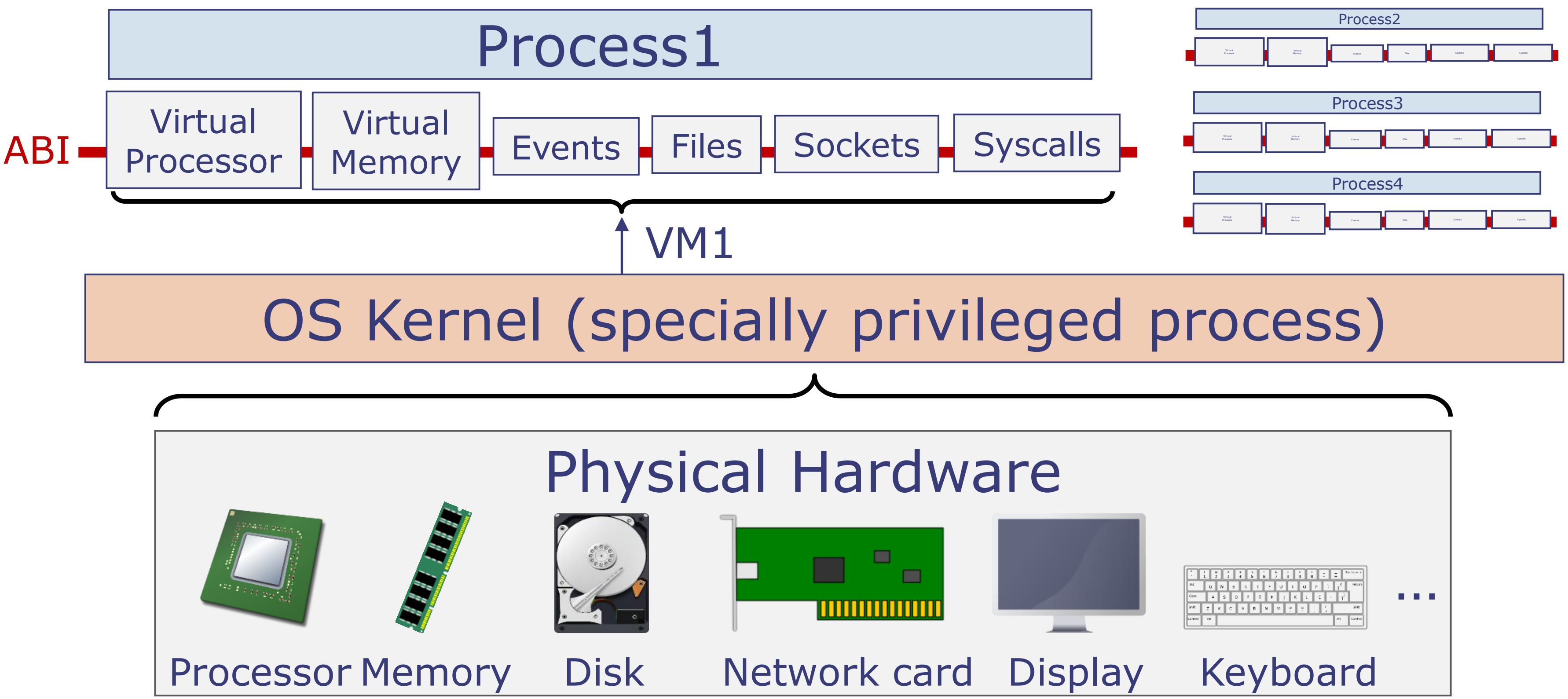
Virtual Machines
A New Layer of Abstraction

- 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



Virtual Machines Are Everywhere

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

RISC-V process (quicksort)

RISC-V ISA

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

Python Language

Python interpreter (CPython) — Implements a Python VM

Linux ABI

Linux OS kernel — Implements a Linux-x86 VM

x86 ISA

VMware — Implements an x86 system VM

Windows/Linux/BSD/… ABI

OS kernel (Win/Linux/BSD/…) — Implements an OS-x86 VM

x86 ISA

Hardware (Athena server) — 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

Today

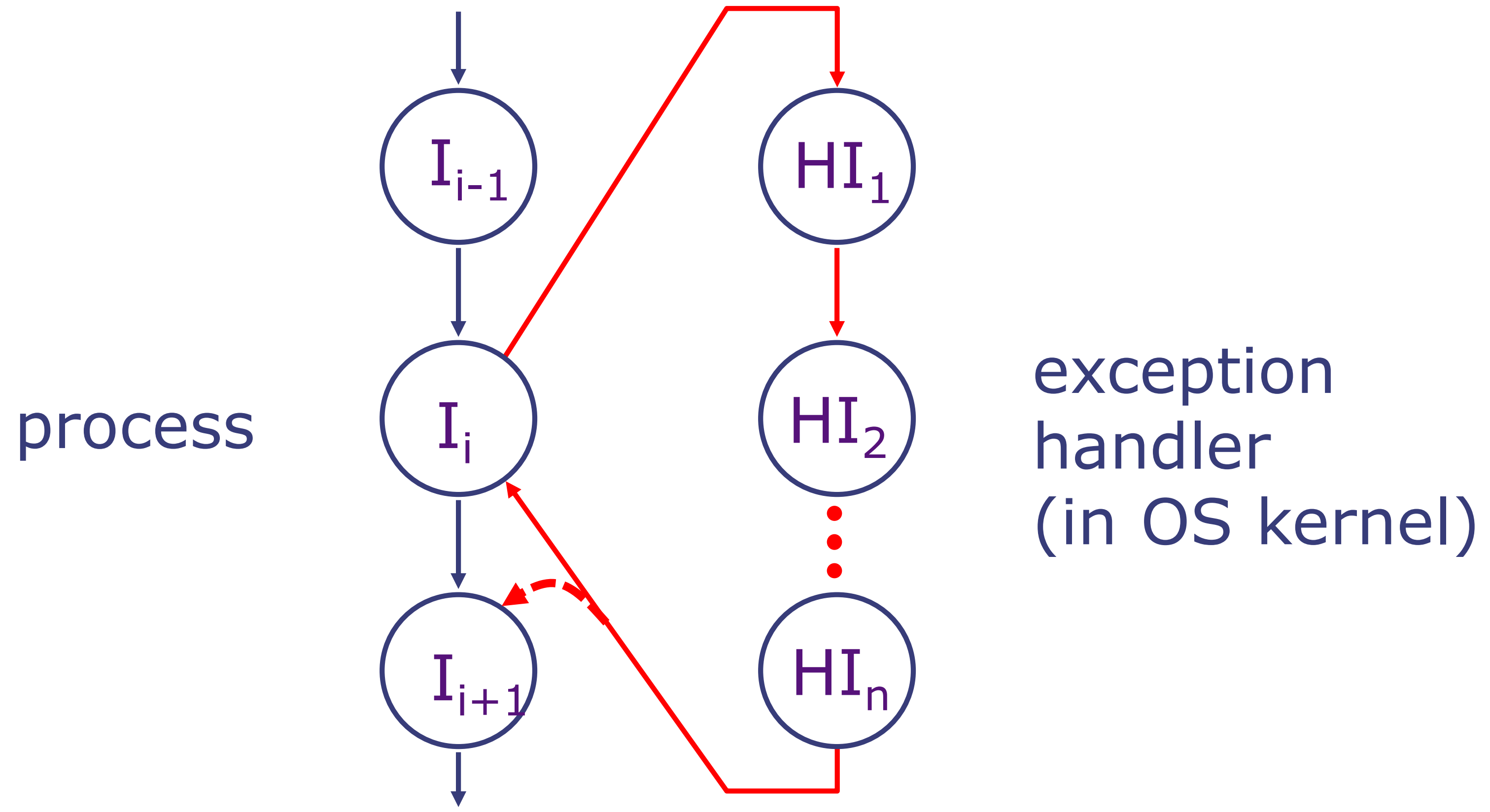
- Virtual memory to provide private address spaces and abstract the storage resources of the machine

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

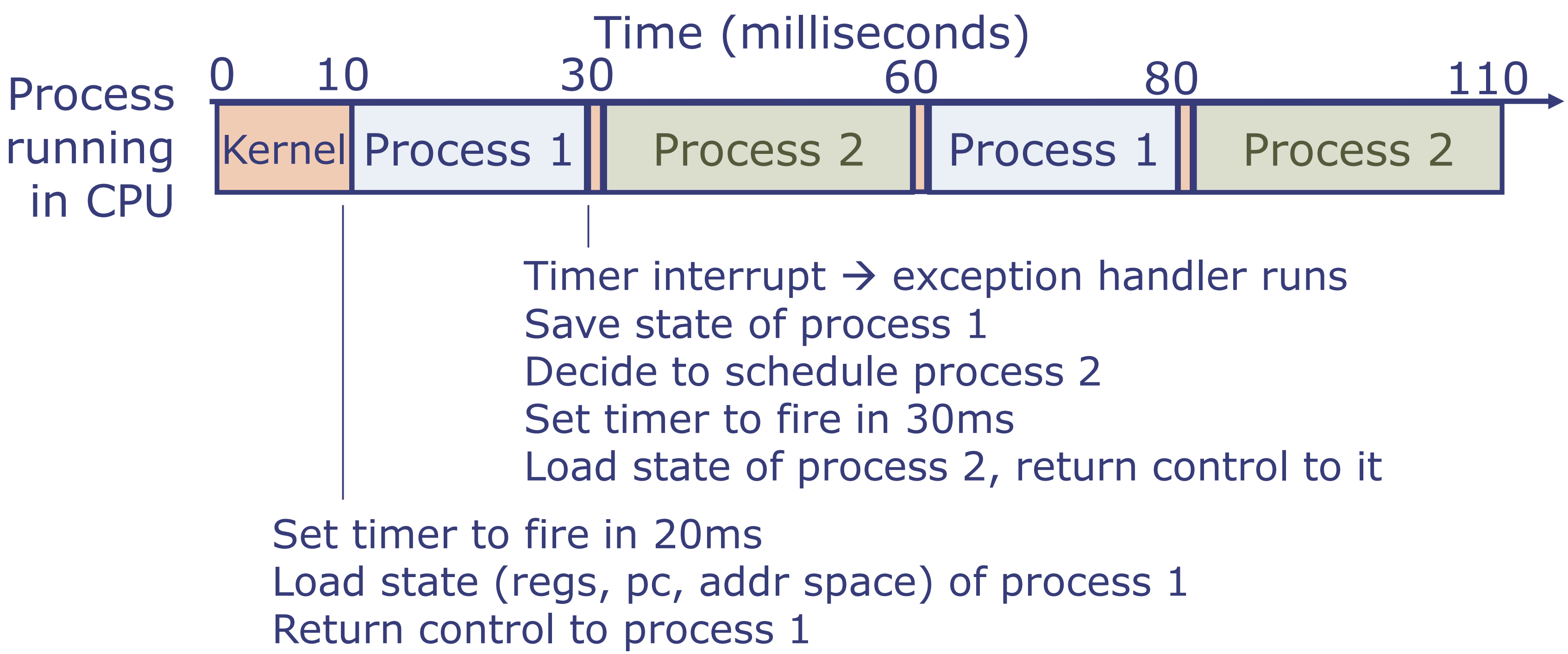
- The terms exception and interrupt are often used interchangeably, with a minor distinction:
- Exceptions usually refer to synchronous events, generated by the process itself (e.g., illegal instruction, divide-by-0, illegal memory address, system call)
- Interrupts usually refer to asynchronous events, 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

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 #2: Emulating Instructions

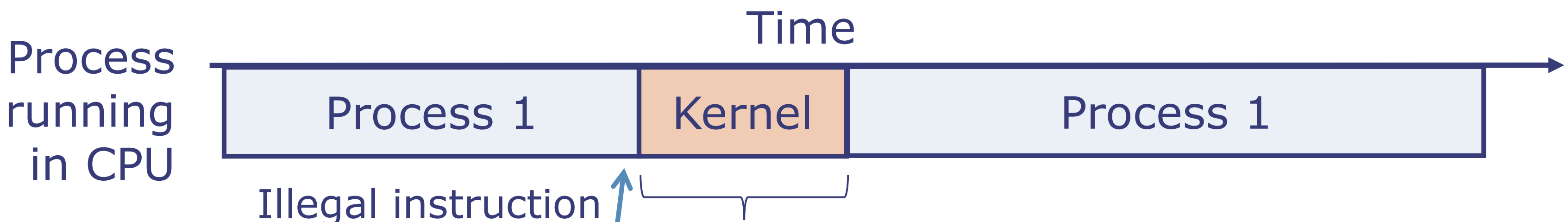
Enabled by illegal instruction exceptions

- 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



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

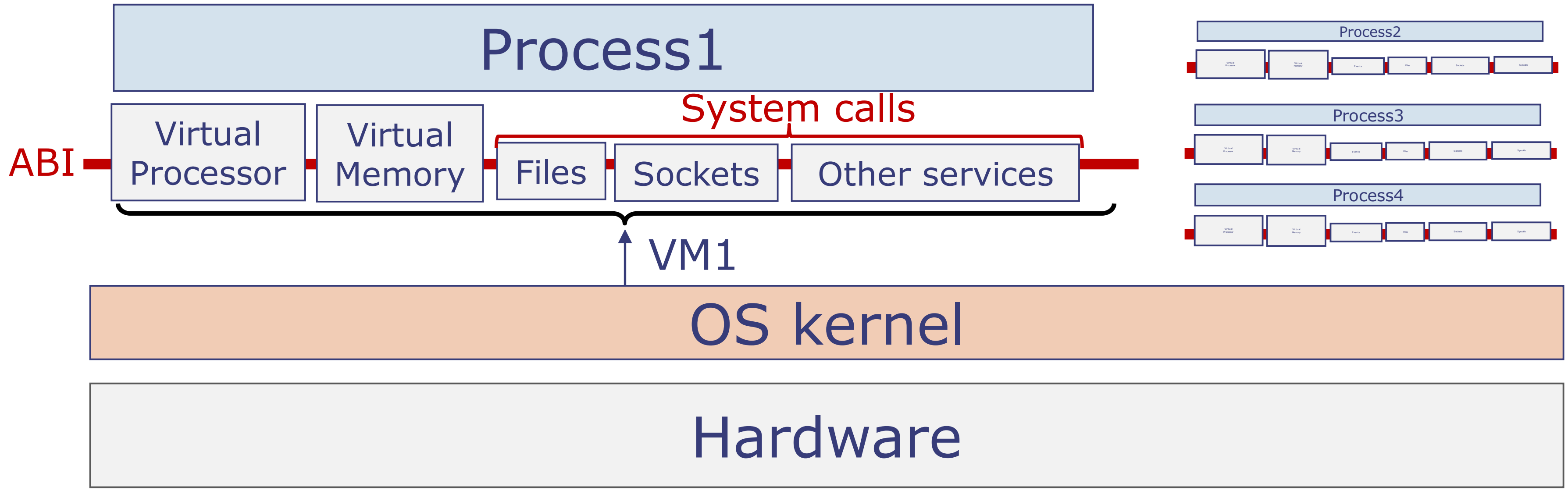
Process 1 code:

```
…
add a3, a2, a1
mul a4, a3, a2
xor a5, a4, a3
…
```

- 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



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

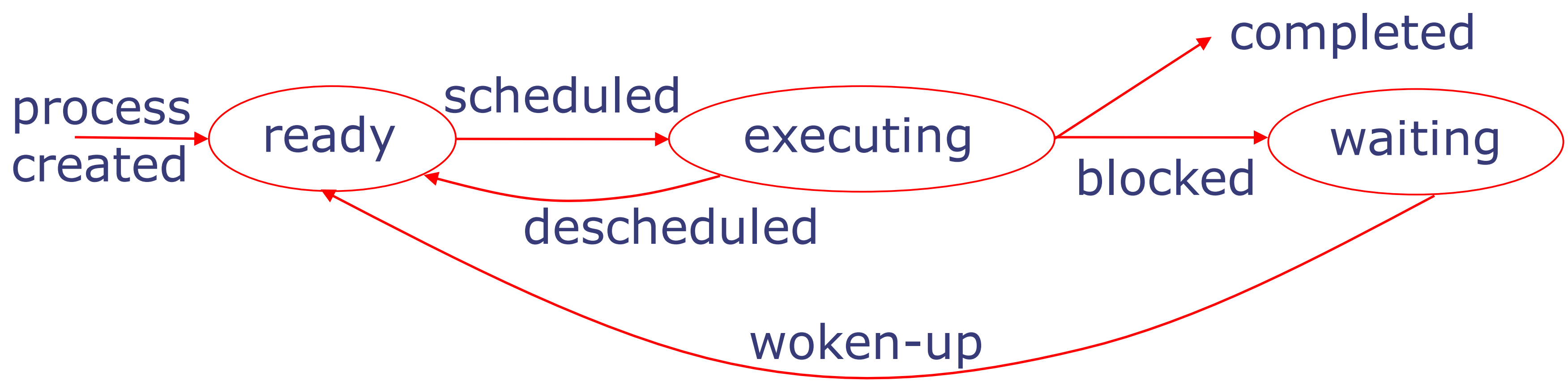
Typical System Calls

- 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!

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
  - 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.)

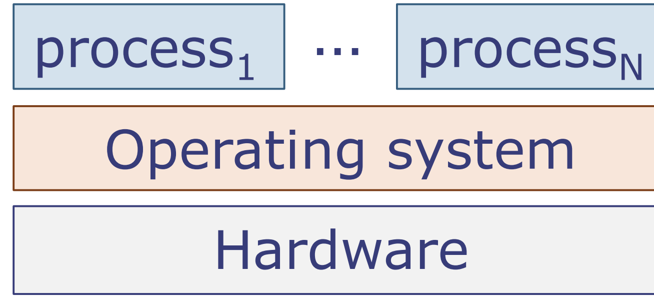
- RISC-V also provides privileged instructions, e.g.,
  - `csrr` and `csrw` to read/write CSRs
  - `mret` to return from the exception handler to the process
  - Trying to execute these instructions from user mode causes an exception → normal processes cannot take over the system

System Calls in RISC-V

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