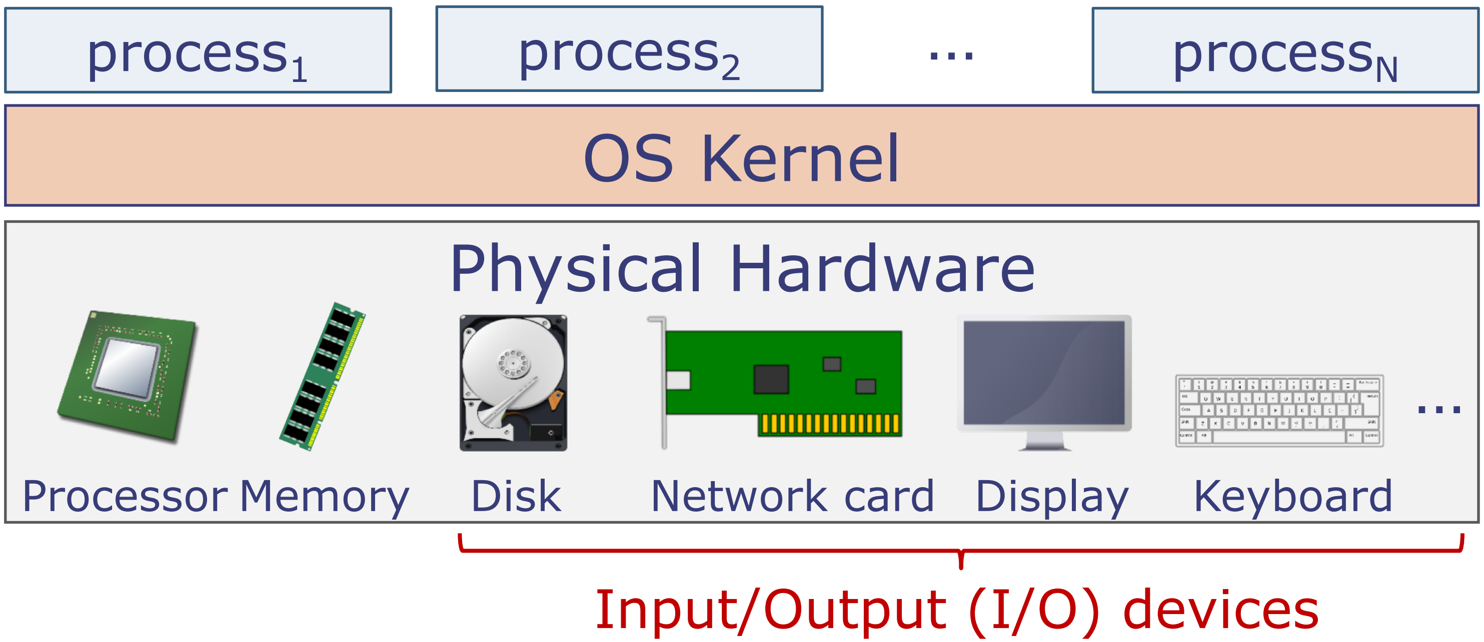
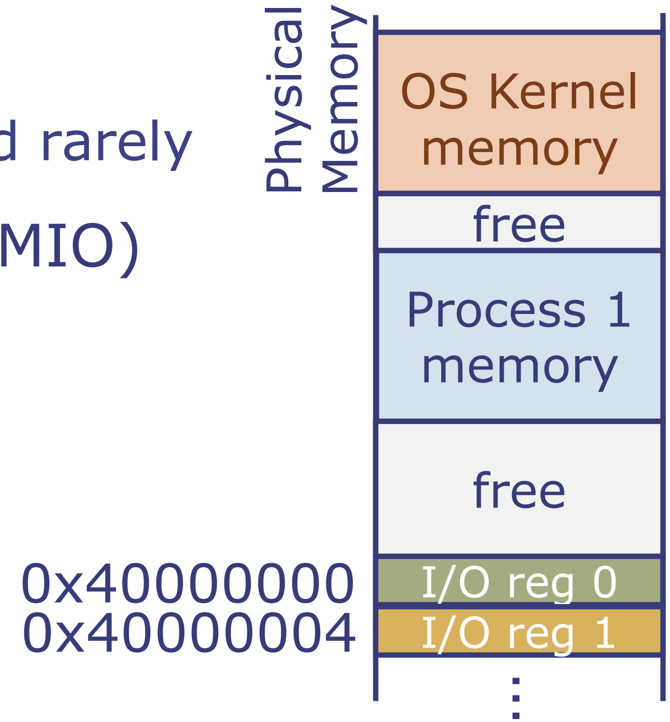
Communicating with I/O devices

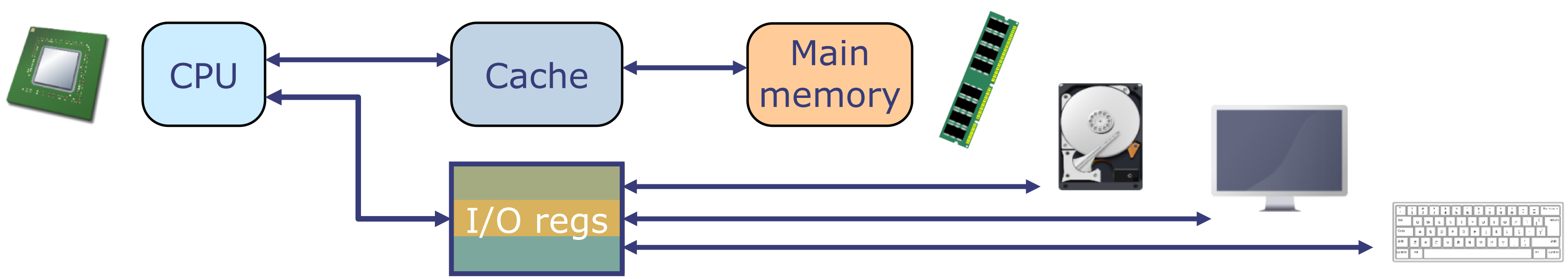


- The system has shared I/O registers that both the processor and the I/O devices can read and write
- Key questions:
  - How to access I/O registers?
  - How do processor and device coordinate on each transfer?

Accessing I/O registers

- Option 1: Use special instructions
  - e.g., `in` and `out` instructions in x86
  - Inflexible, adds instructions → used rarely
- Option 2: Memory-mapped I/O (MMIO)
  - I/O registers are mapped to physical memory locations
  - Processor accesses them with loads and stores
  - These loads and stores should not be cached!





Coordinating I/O Transfers

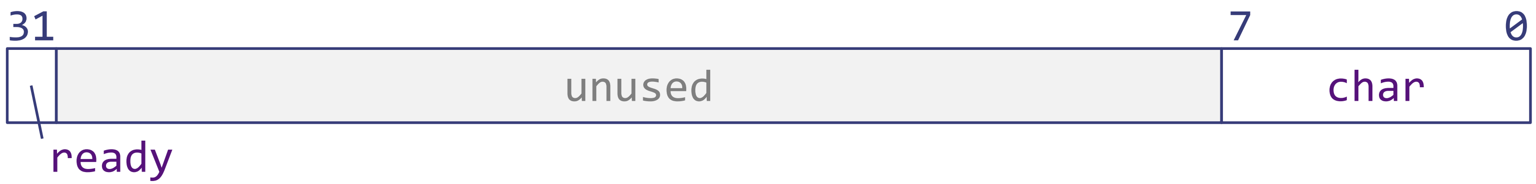
- Option 1: Polling (synchronous)
  - Processor periodically reads the register associated with a specific I/O device
- Option 2: Interrupts (asynchronous)
  - Processor initiates a request, then moves to other work
  - When the request is serviced, the I/O device interrupts the processor
- *Pros of each approach?*

Polling is simple

Interrupts let the processor do useful computation while request is serviced

Example 1: Polling-based I/O

- Consider a simple character-based display
- Uses one I/O register with the following format:

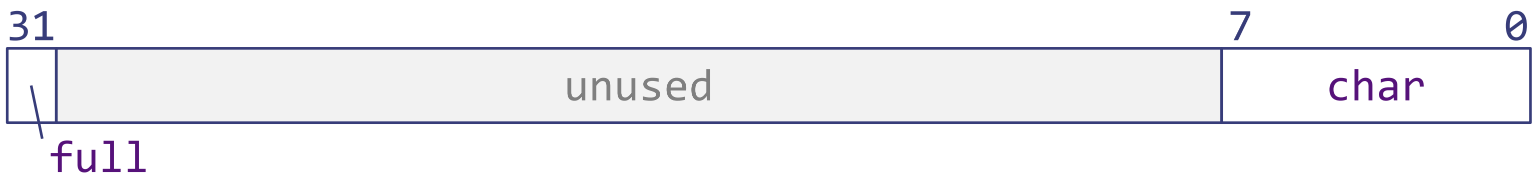


  - The ready bit (bit 31) is set to 1 only when the display is ready to print a character
  - When the processor wants to display an 8-bit character, it writes it to char (bits 0-7) and sets the ready bit to 0
  - After the display has processed the character, it sets the ready bit to 1

Let’s see a demo!

Example 2: Interrupt-based I/O

- Consider a simple keyboard that uses a single I/O register with a similar format:

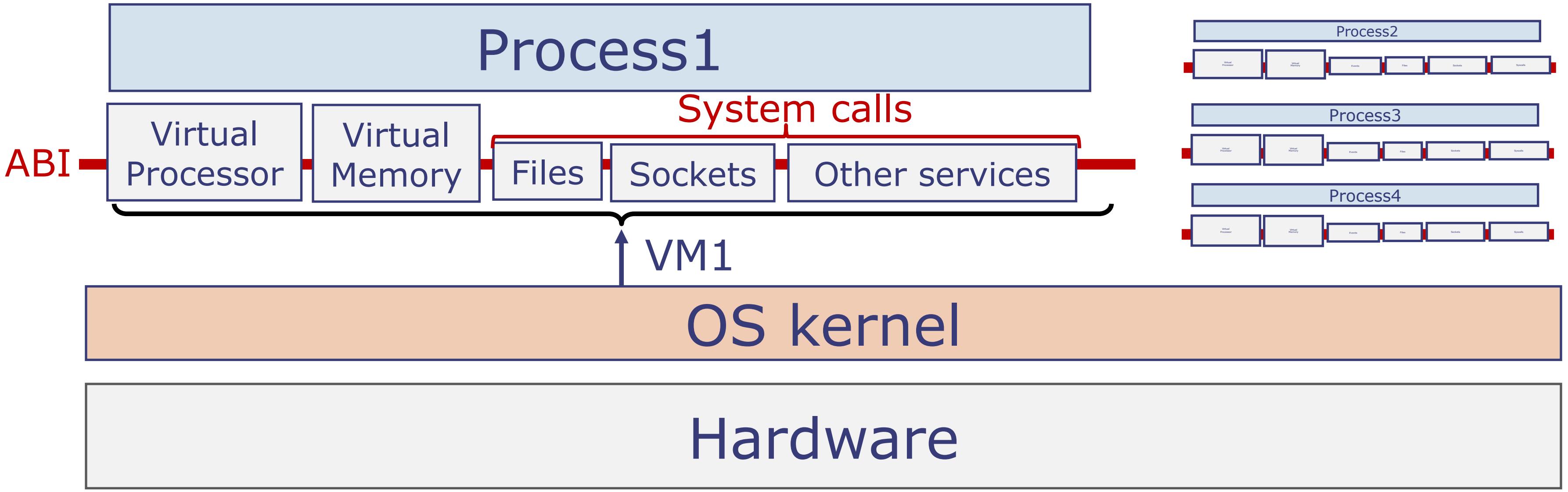


  - On a keystroke, the keyboard writes the typed character in char, sets full bit to 1, and raises a keyboard interrupt
  - Interrupt handler reads char and sets full bit to 0, so the keyboard can deliver future keystrokes

  - This works fine because keyboard is very slow compared to CPU. Faster devices use more sophisticated mechanisms (e.g., network cards write packets to main memory and use I/O registers only to indicate status of transfer)

Communicating with the OS

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



- Processes invoke system calls by executing an instruction that causes an exception
  - Same mechanism as before!

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)

Let’s see a demo!

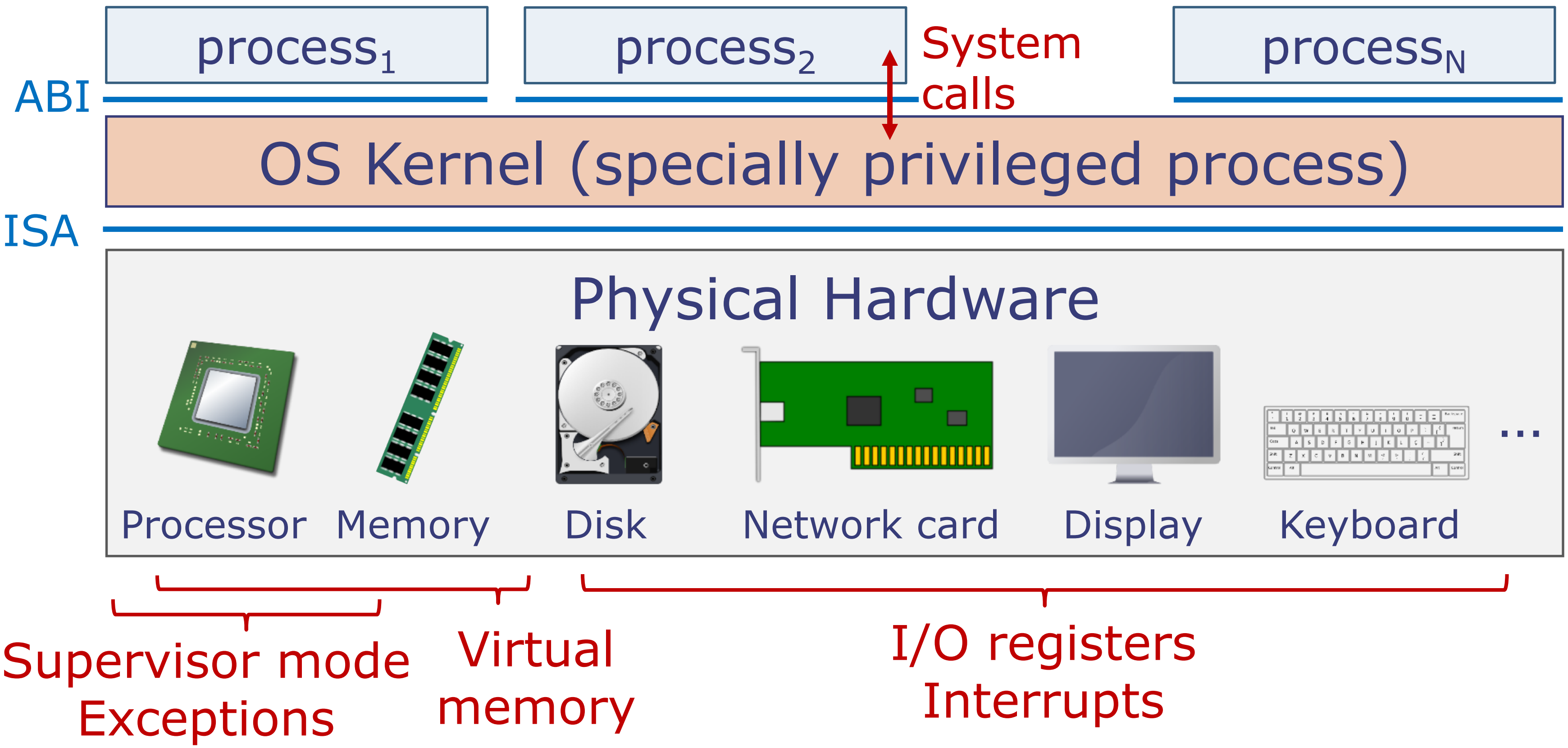
Typical System Calls

- 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!

Summary



- Want to learn more? Check out 6.033 or 6.828!