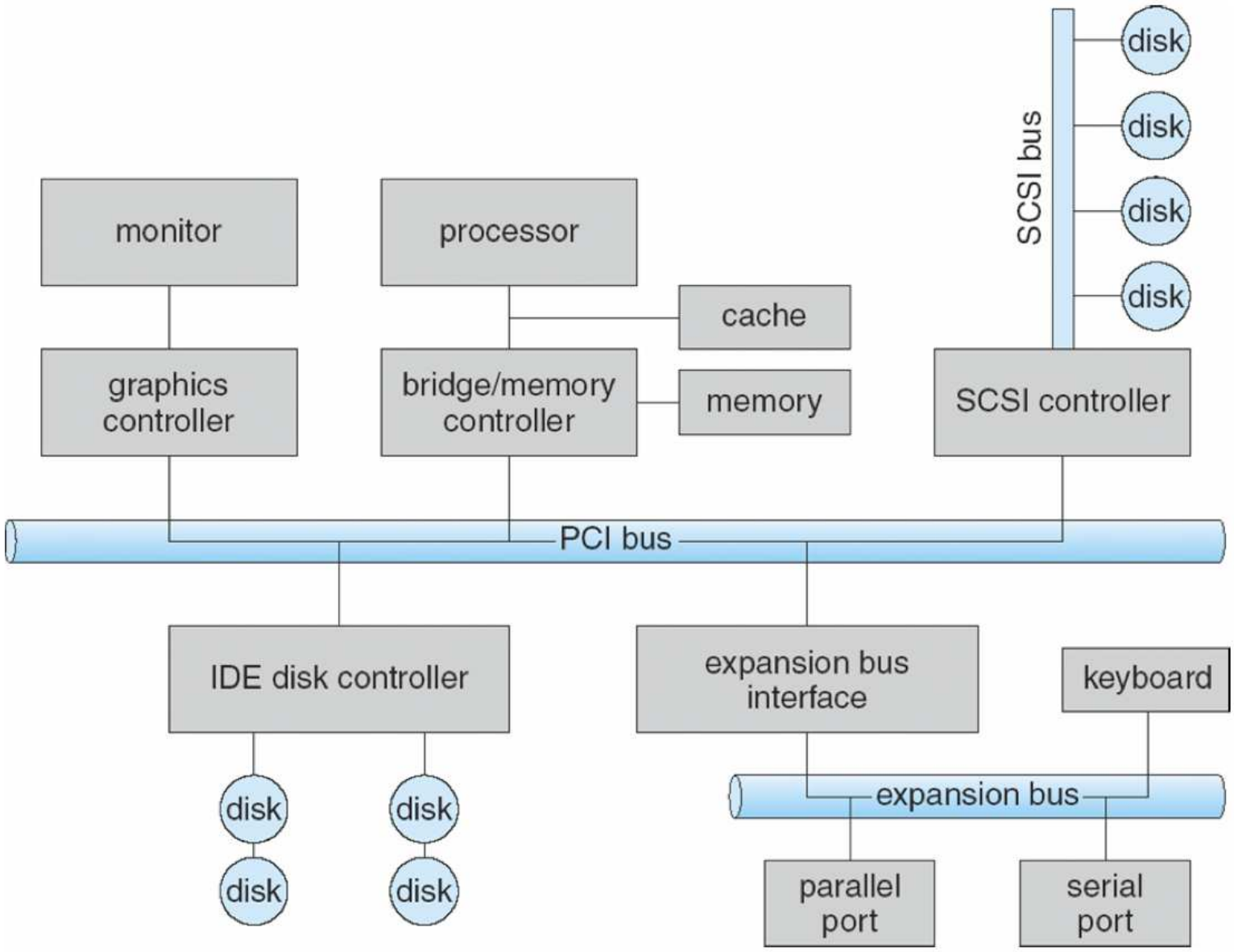
PC Architecture



PC Port Addresses

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Programmed I/O (PIO) with Polling

- Use **`OUT`** instruction to make a device request
- Loop using **`IN`** until the result comes back

Port usually has at least 4 registers:

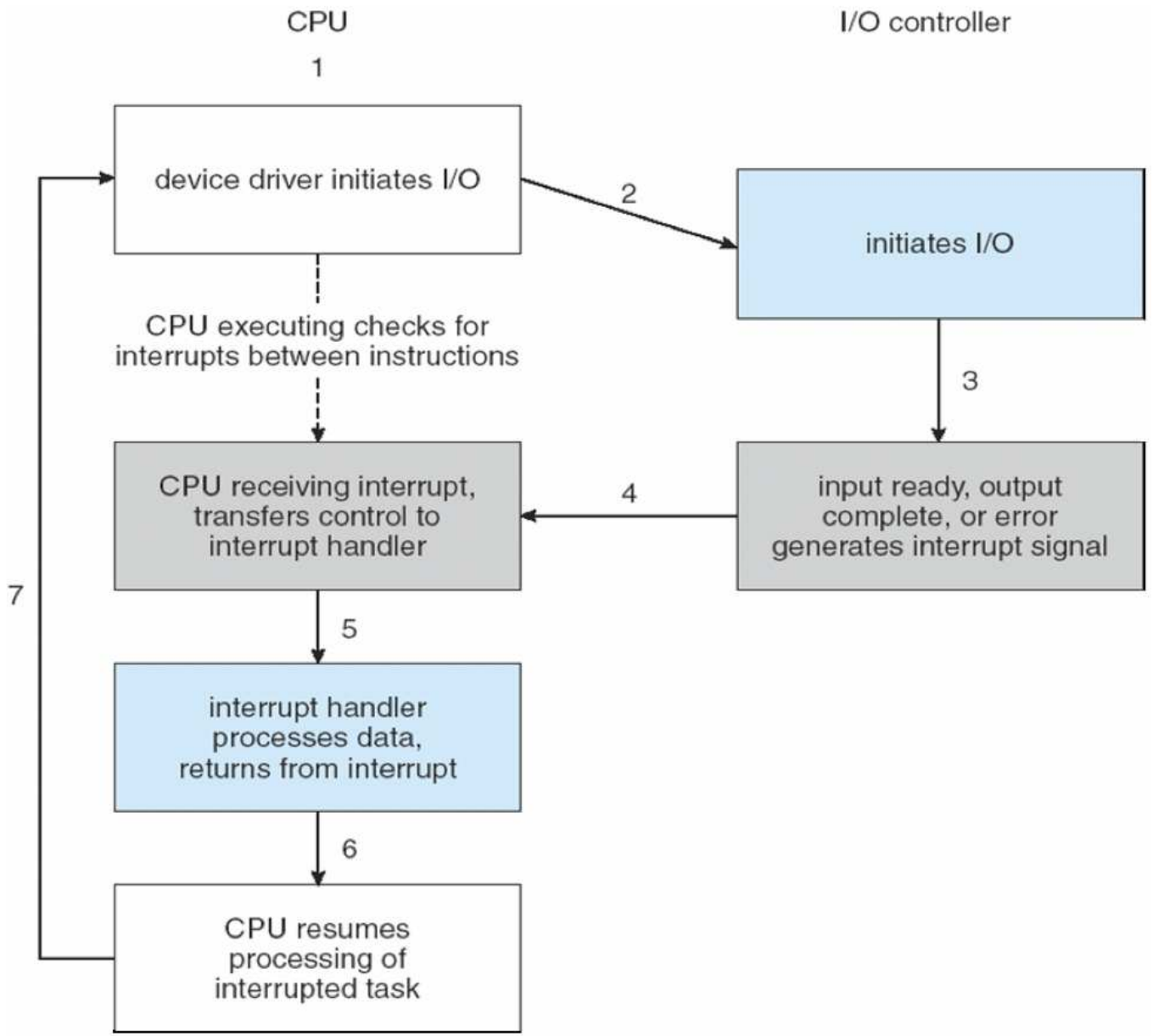
- **Status** — written by device, read by CPU
- **Control** — written by CPU, read by device
- **Data-in** — data sent from the device to the CPU
- **Data-out** — data sent from the CPU to the device

Devices

A From the docs for a PS/2 mouse interface controller:

> Read status register. If Data Pending bit is set, read data register, store the value (it may be required by handler software). Repeat until Data Pending bit is not set for more than 2ms, or more than 16 bytes have been read. Also check Active bit during this procedure: If no data is pending, check active. If active is not set, proceed with initialize, if active is set, timeout after 32ms. Always read pending data in this procedure! Quick end of this procedure: If chip is not active, data is not pending, and both PS2DAT and PS2CLK are set, the bus is idle. Proceed with initialize in this case.

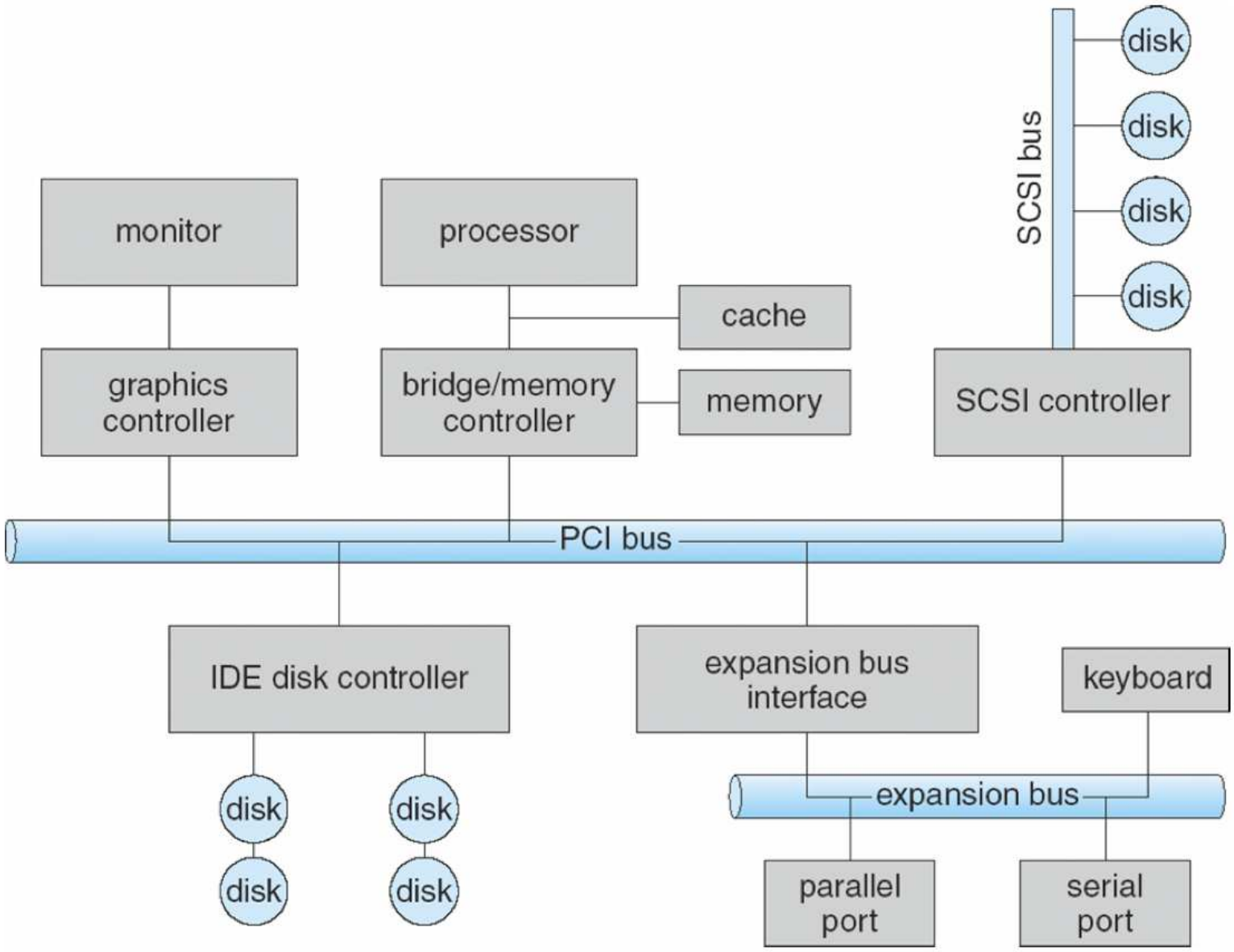
Using Interrupts



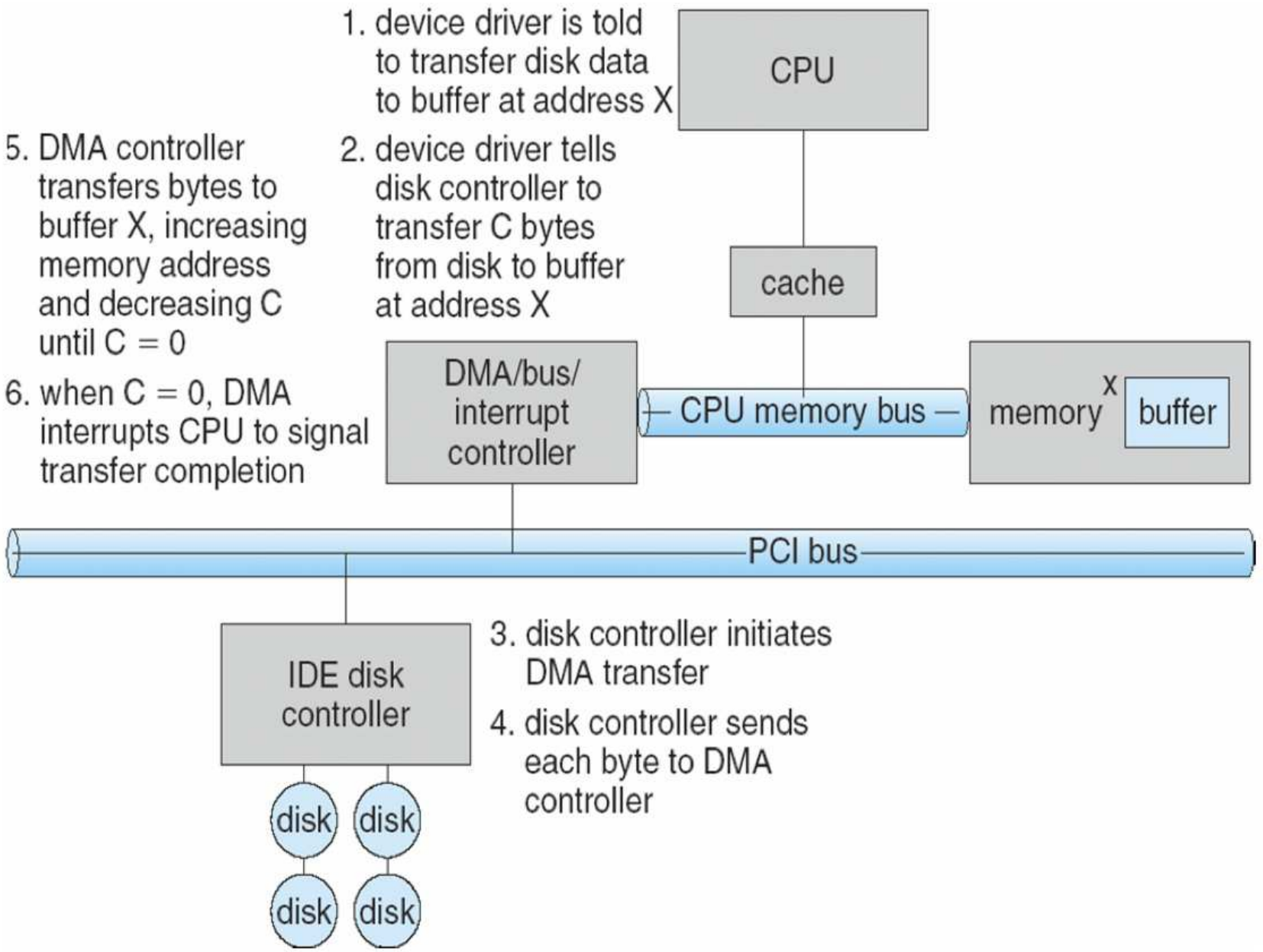
Interrupt Vector

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

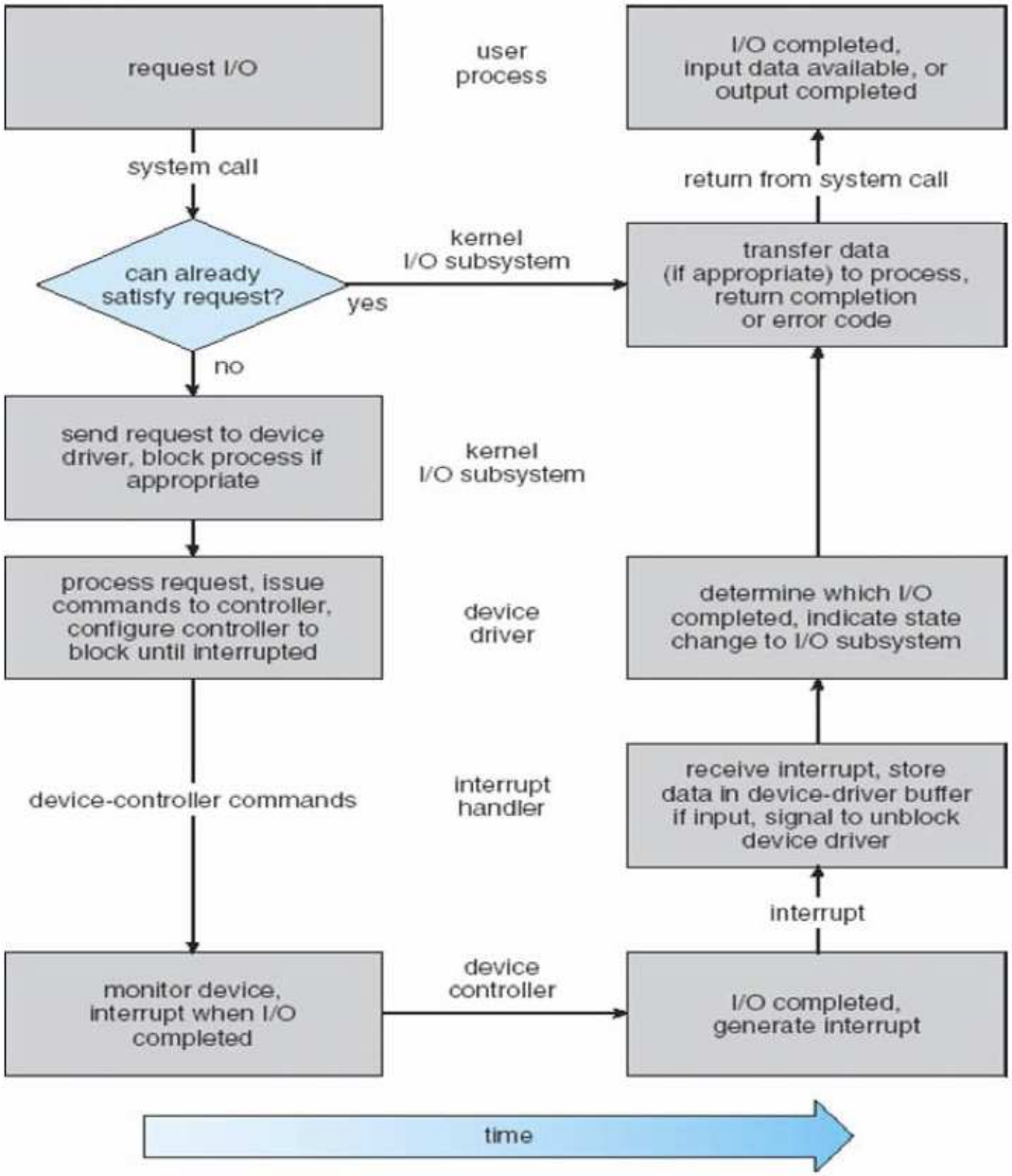
PC Architecture



Direct-Memory Access (DMA)



Implementing I/O



Application Interface

- A ***file descriptor*** can be
  - A file
  - A pipe
  - A network connection
  - Other device: a terminal, **`/dev/null`**
- **`read()`** and **`write()`** work on all of them
- **`lseek()`** works on some of them

see **`byte.c`**, **`block.c`**

Buffering vs. Interleaving

- Buffering allows more data per request
- Buffering can interfere with interactivity
  - Interactivity ≈ scheduling flexibility

see **`block2.c`**

I/O Patterns

- ***blocking*** waits until I/O is available
- ***non-blocking*** returns, maybe did I/O
- ***asynchronous*** returns, I/O done meanwhile

see **`nonblock.c`**, **`nonblock2.c`**, **`thread.c`**

see **`as_client.c`**, **`server.c`**

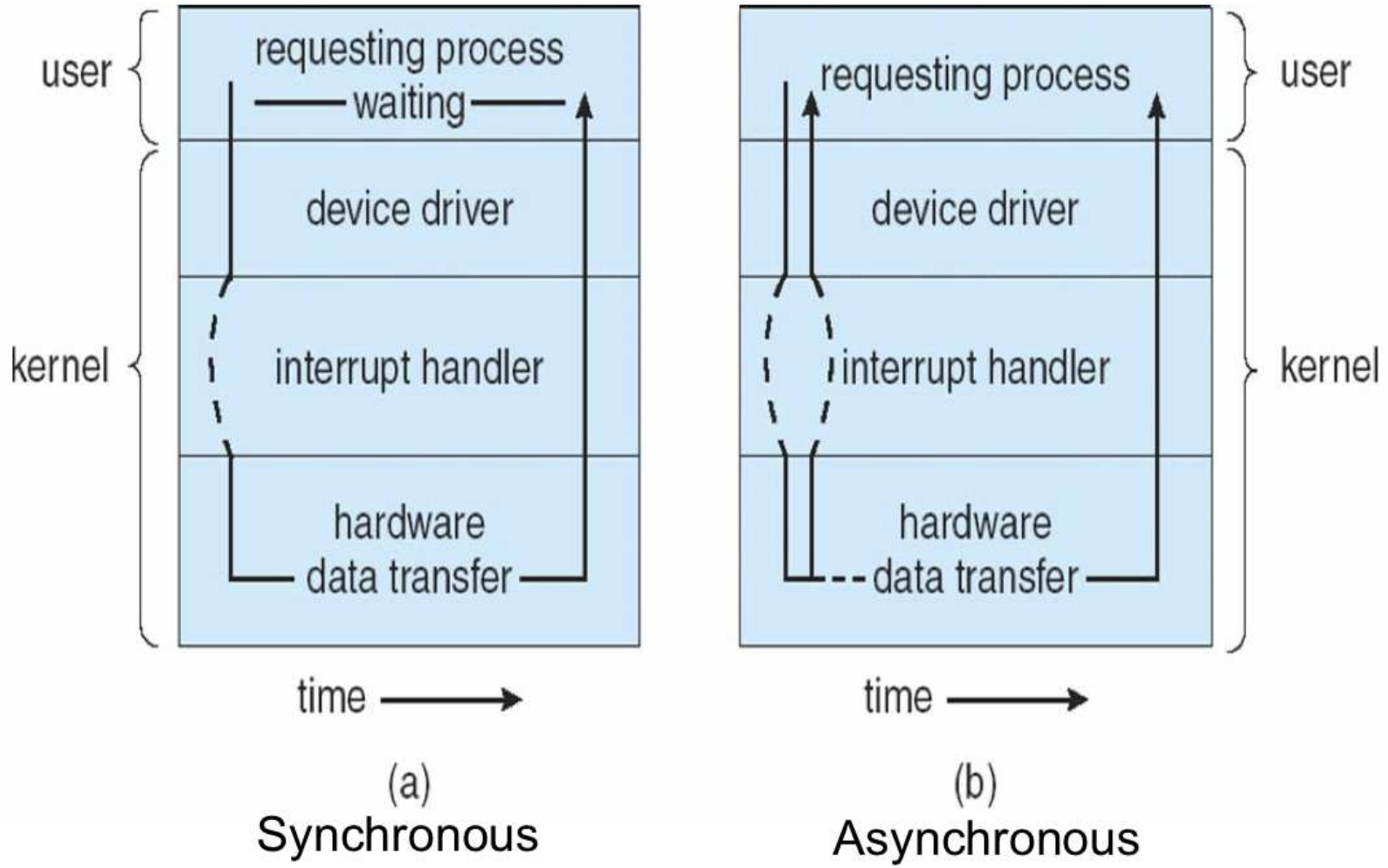
Select

For non-blocking I/O, **`select()`** waits for I/O

- No timeout: waits until I/O available from one device
- Zero timeout: polls devices

see **`nonblock.c`**, **`server2.c`**

Synchronous vs. Asynchronous



Asynchronous I/O

How do you know when asynchronous I/O has completed?

- Poll
- Callback

see **`async.c`**, **`async2.c`**

Summary

- I/O is slow
  - Need to overlap computation and I/O
  - Need to balance buffering and interactivity
- Blocking, non-blocking, and asynchronous modes
  - Blocking: use threads to overlap
  - Non-blocking: need poll/wait operation like **`select()`**
  - Asynchronous: either poll/wait or callback