I/O and Disks

Thierry Sans

I/O

I/O management

…

- I/O devices vary greatly
  and new types of I/O devices appear frequently

- Various methods to control them
  and to manage their performances

➡ Ports, buses, device controllers connect to various devices

I/O Device Interfaces

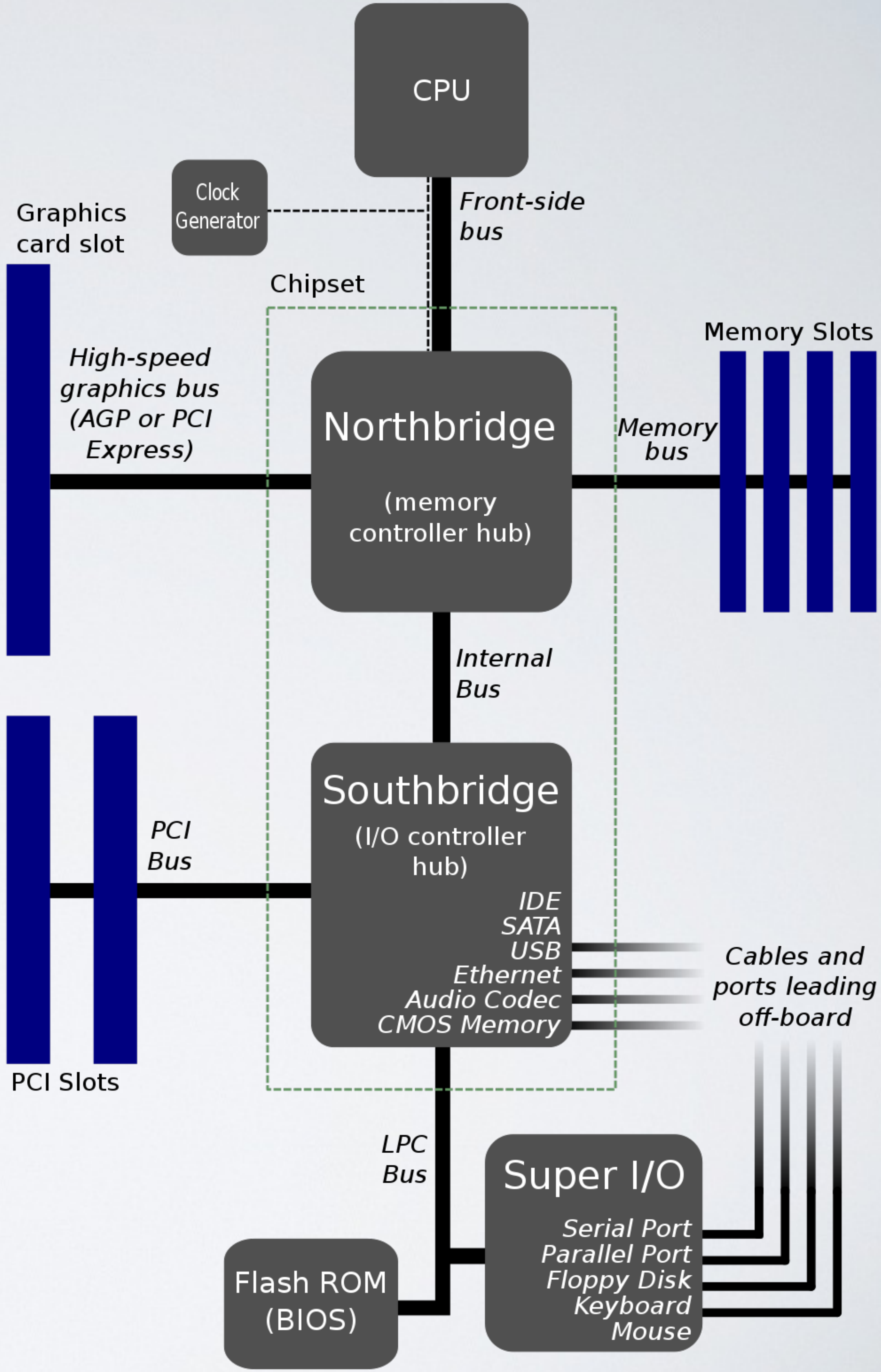
Port - connection point for device (e.g. serial port)

Bus - daisy chain or shared direct access
e.g. Peripheral Component Interconnect Bus (PCI)
e.g Universal Serial Bus (USB)

Controller (host adapter) - electronics that operate port, bus, device (e.g Northbridge, Southbridge, graphics controller, DMA, NIC, ...)

- Can be integrated or separated (host adapter)
- Contains processor, microcode, private memory, bus controller, etc

I/O architecture



How the OS communicates with the device?

➡ Each device has three types of registers
and the OS controls the device by reading or writing these registers

status register
See the current status of the device

command register (also called control register)
Tell the device to perform a certain task

data register
Pass data to the device, or get data from the device

Two ways to read/write those registers

I/O ports
`in` and `out` instructions on x86 to read and write devices registers

Memory-mapped I/O
Device registers are available as if they were memory locations and the OS can `load` (to read) or `store` (to write) to the device

I/O Ports on PC

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)

Reading/Writing to I/O ports

Pintos `threads/io.h`

```
static inline uint8_t inb (uint16_t port)
{
  uint8_t data;
  asm volatile ("inb %w1, %b0" : "=a" (data) : "Nd" (port));
  return data;
}


static inline void outb (uint16_t port, uint8_t data)
{
  asm volatile ("outb %b0, %w1" : : "a" (data), "Nd" (port));
}
```

Device driver

```
while (STATUS == BUSY)
    ; //wait until device is not busy
write data to data register
write command to command register
    Doing so starts the device and executes the command
while (STATUS == BUSY)
    ; //wait until device is done with your request
```

Example : parallel port (LPT1)

- Three registers

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

read/write data register (port 0x378)

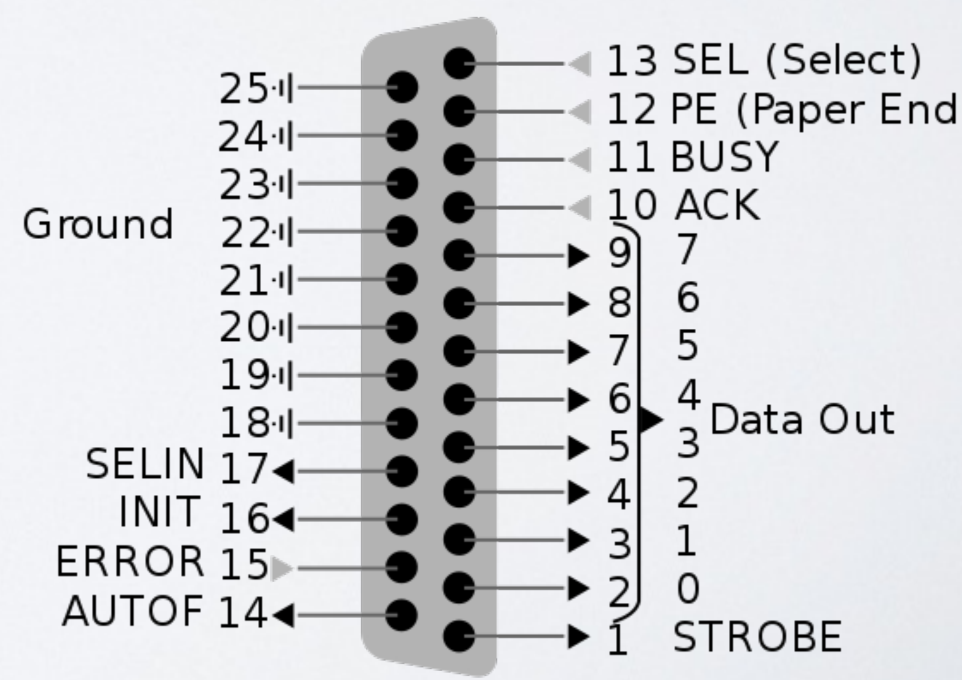
$\overline{\text{BSY}}$	$\overline{\text{ACK}}$	PAP	OFON	$\overline{\text{ERR}}$	–	–	–

read-only status register (port 0x379)

–	–	–	IRQ	DSL	$\overline{\text{INI}}$	ALF	STR

read/write control register (port 0x37a)

- Every bits (except IRQ) corresponds to a pin on 25-pin connector



D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

read/write data register (port 0x378)

$\overline{\text{BSY}}$	$\overline{\text{ACK}}$	PAP	OFON	$\overline{\text{ERR}}$	–	–	–

read-only status register (port 0x379)

–	–	–	IRQ	DSL	$\overline{\text{INI}}$	ALF	STR

read/write control register (port 0x37a)

Parallel Port Driver

```
void
sendbyte(uint8_t byte)
{
  /* Wait until BSY bit is 1. */
  while ((inb (0x379) & 0x80) == 0)
    delay ();

  /* Put the byte we wish to send on pins D7-0. */
  outb (0x378, byte);

  /* Pulse STR (strobe) line to inform the printer
   * that a byte is available */
  uint8_t ctrlval = inb (0x37a);
  outb (0x37a, ctrlval | 0x01);
  delay ();
  outb (0x37a, ctrlval);
}
```

Polling

➡ OS waits until the device is ready by repeatedly reading the status register

✓ Simple and working

◉ Wastes CPU time just waiting for the device

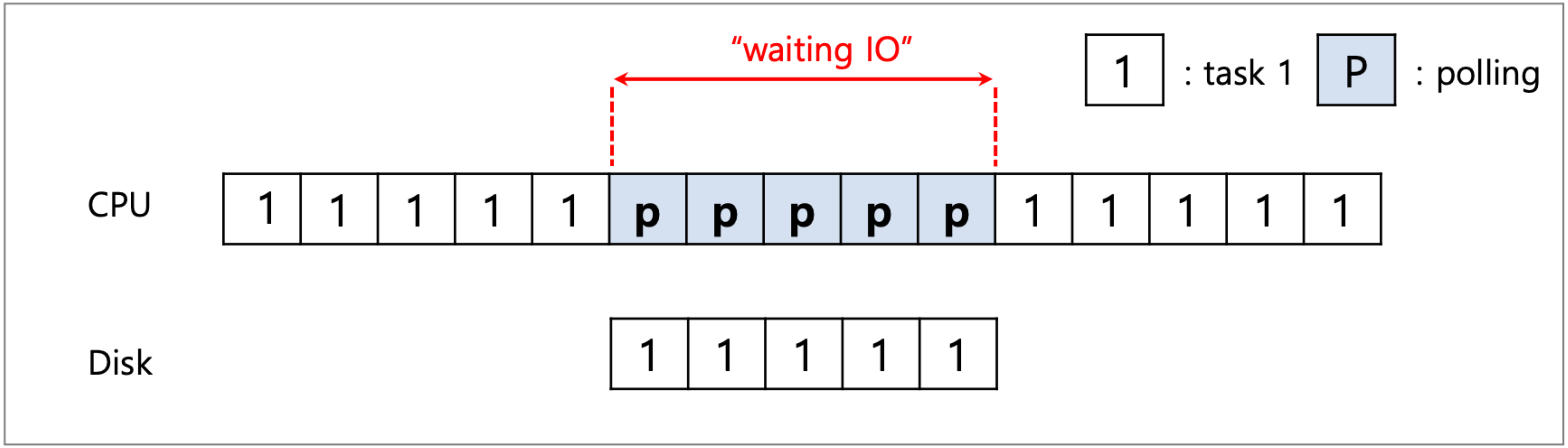


Diagram of CPU utilization by polling

Interrupts

1. Put the I/O request process to sleep and switch context
2. When the device is finished, send an interrupt to wake the process waiting for the I/O

✓ CPU is properly utilized

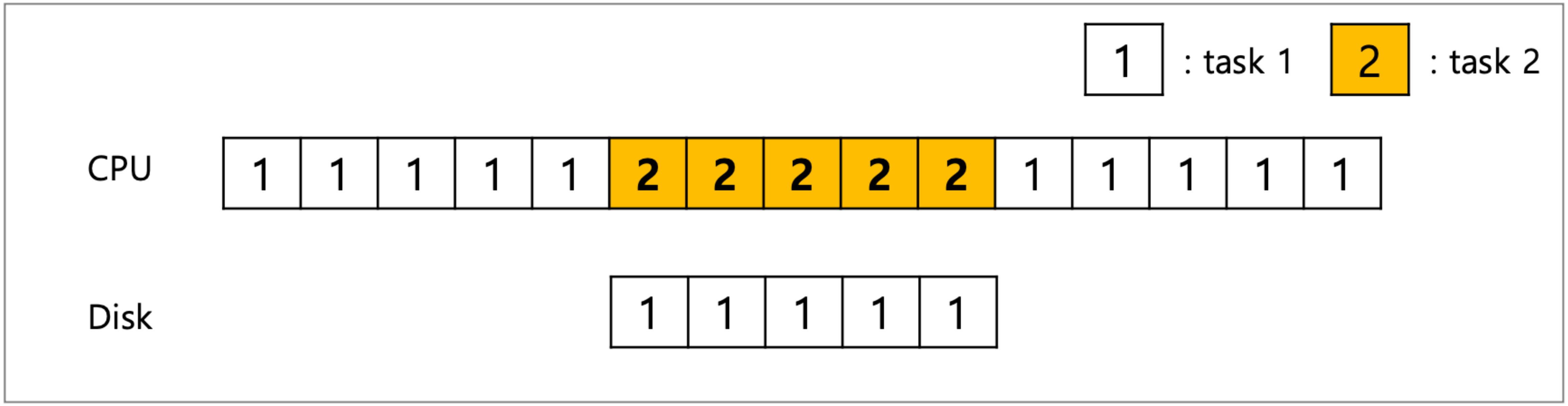


Diagram of CPU utilization by interrupt

Polling vs Interrupts

➡ **Interrupts is not always the best solution**
If, device performs very quickly, interrupt will slow down the system

E.g. high network packet arrival rate

- Packets can arrive faster than OS can process them
- Interrupts are very expensive (context switch)
- Interrupt handlers have high priority
- In worst case, can spend 100% of time in interrupt handler and never make any progress a.k.a receive livelock

✓ Best - adaptive switching between interrupts and polling

One More Problem : Data Copying

- CPU wastes a lot of time in copying a large chunk of data from memory to the device

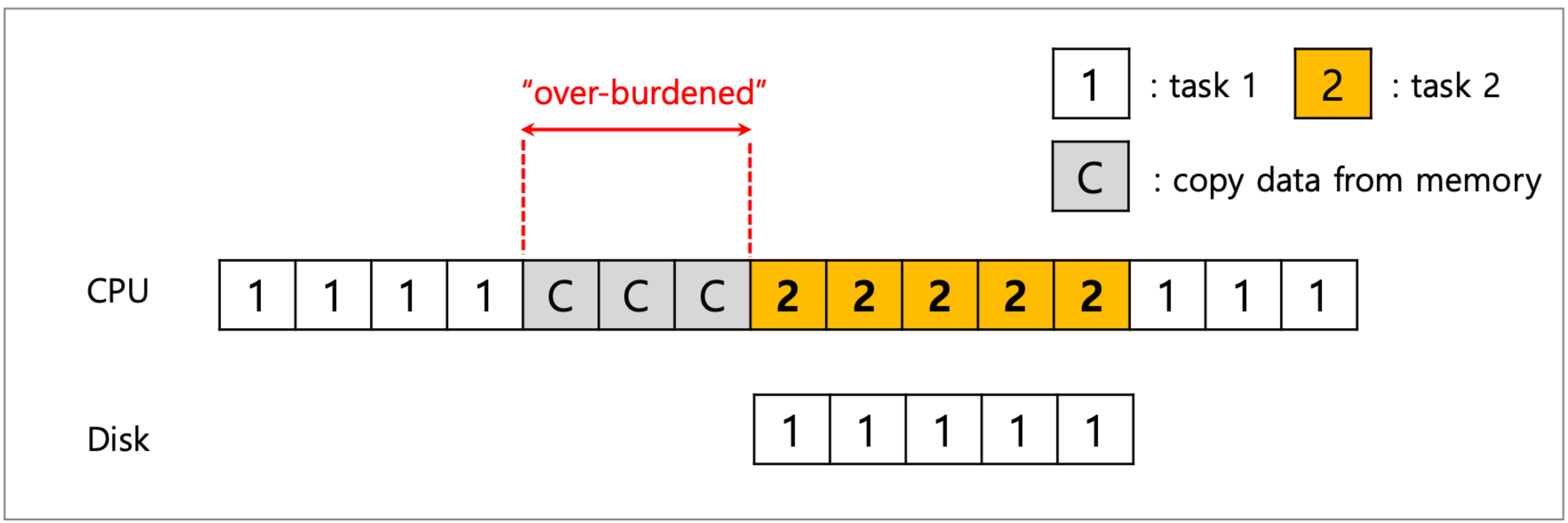
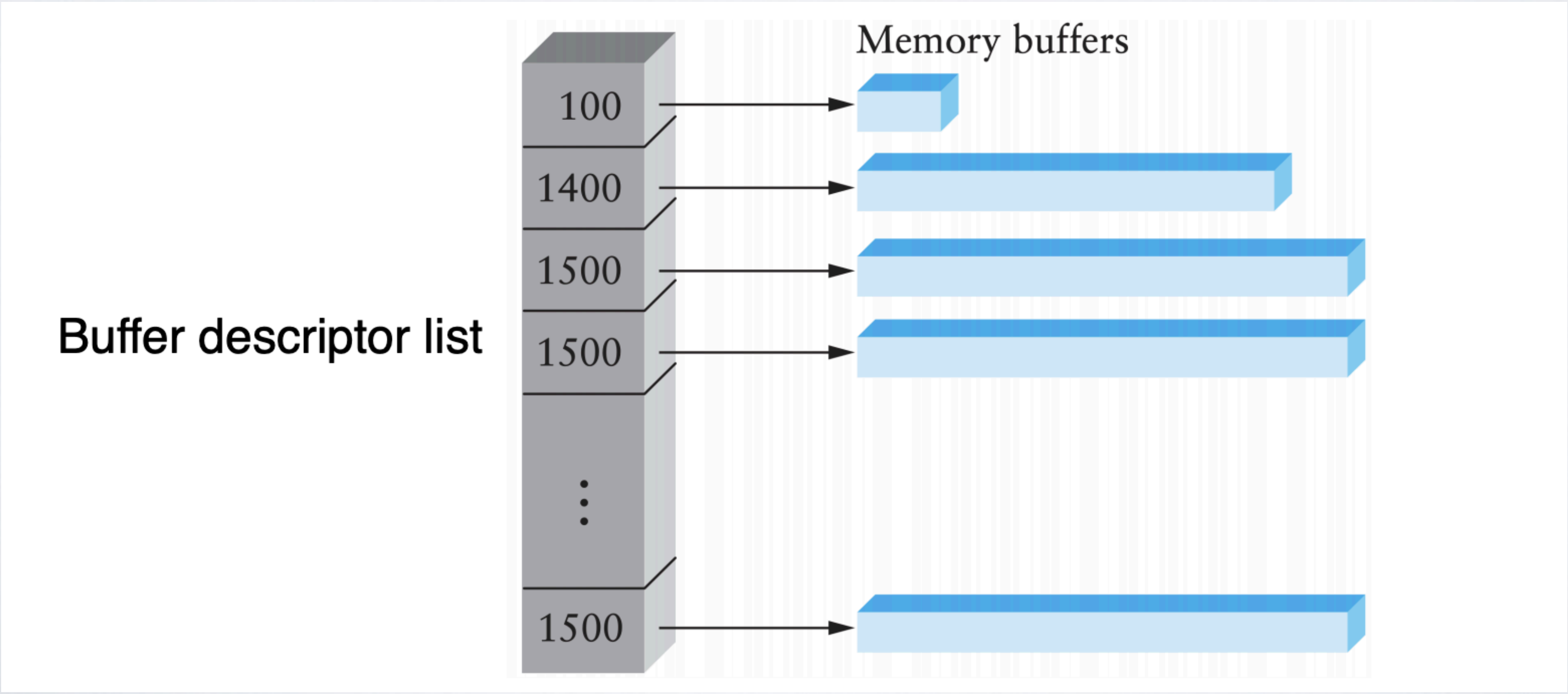


Diagram of CPU utilization

DMA (Direct Memory Access)

- ➡ Only use CPU to transfer control requests, not data, by passing buffer locations in memory
  - Device reads list and accesses buffers through DMA
  - Descriptions sometimes allow for scatter/gather I/O



DMA (Direct Memory Access)

1. OS writes DMA command block into memory
2. DMA bypasses CPU to transfer data directly between I/O device and memory
3. When completed, DMA raises an interrupt

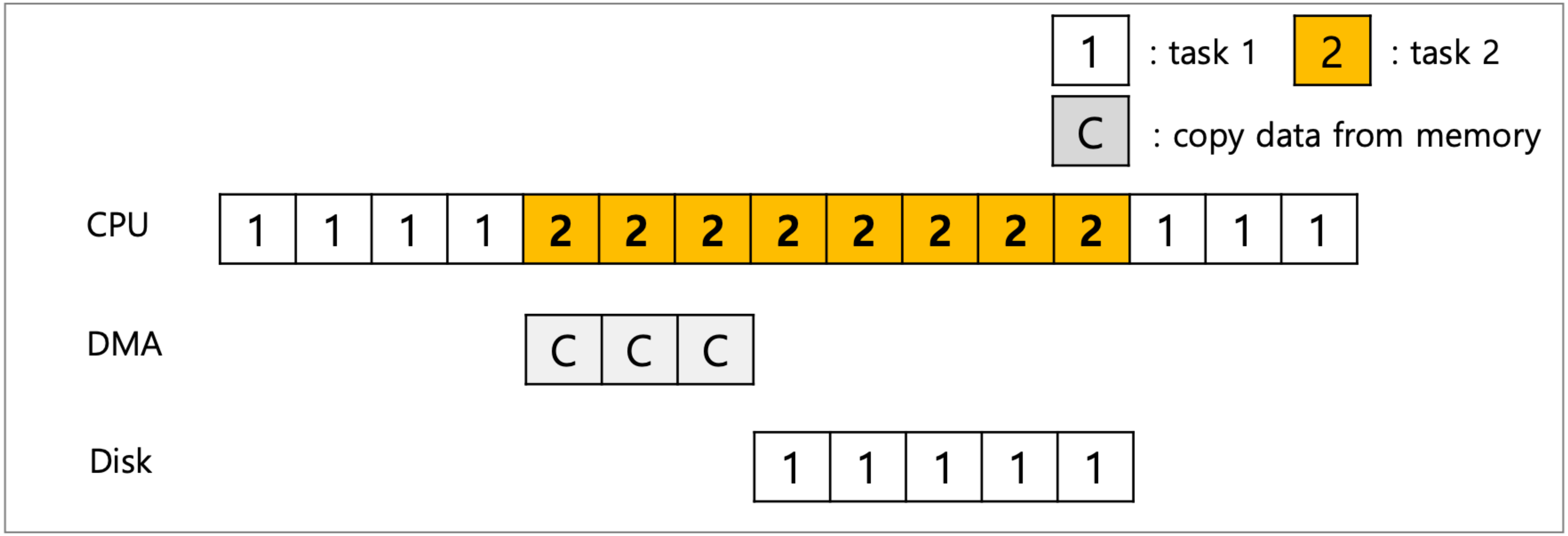
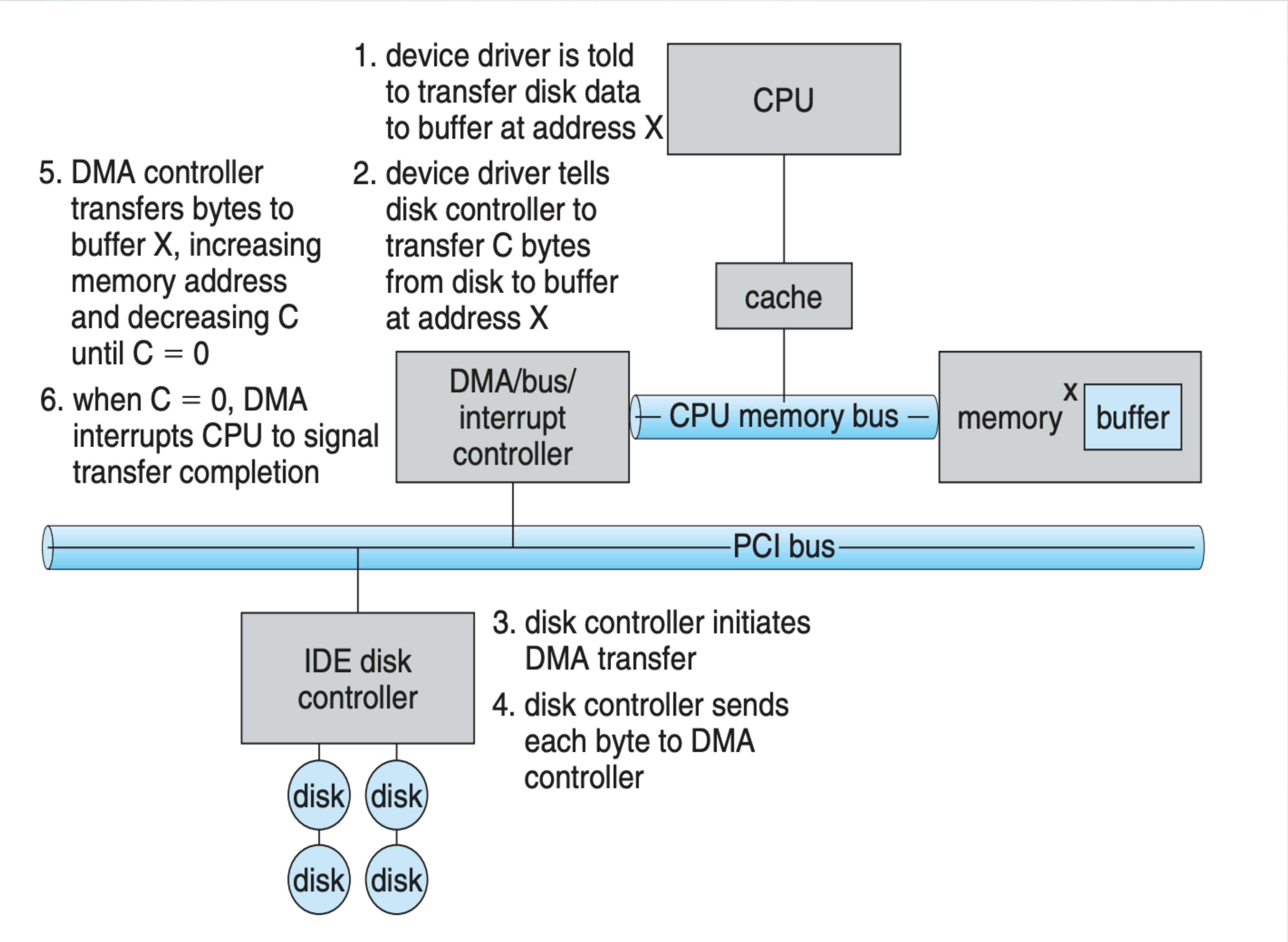


Diagram of CPU utilization by DMA

Example : IDE disk read with DMA



I/O instruction using DMA

Pintos `threads/io.h`

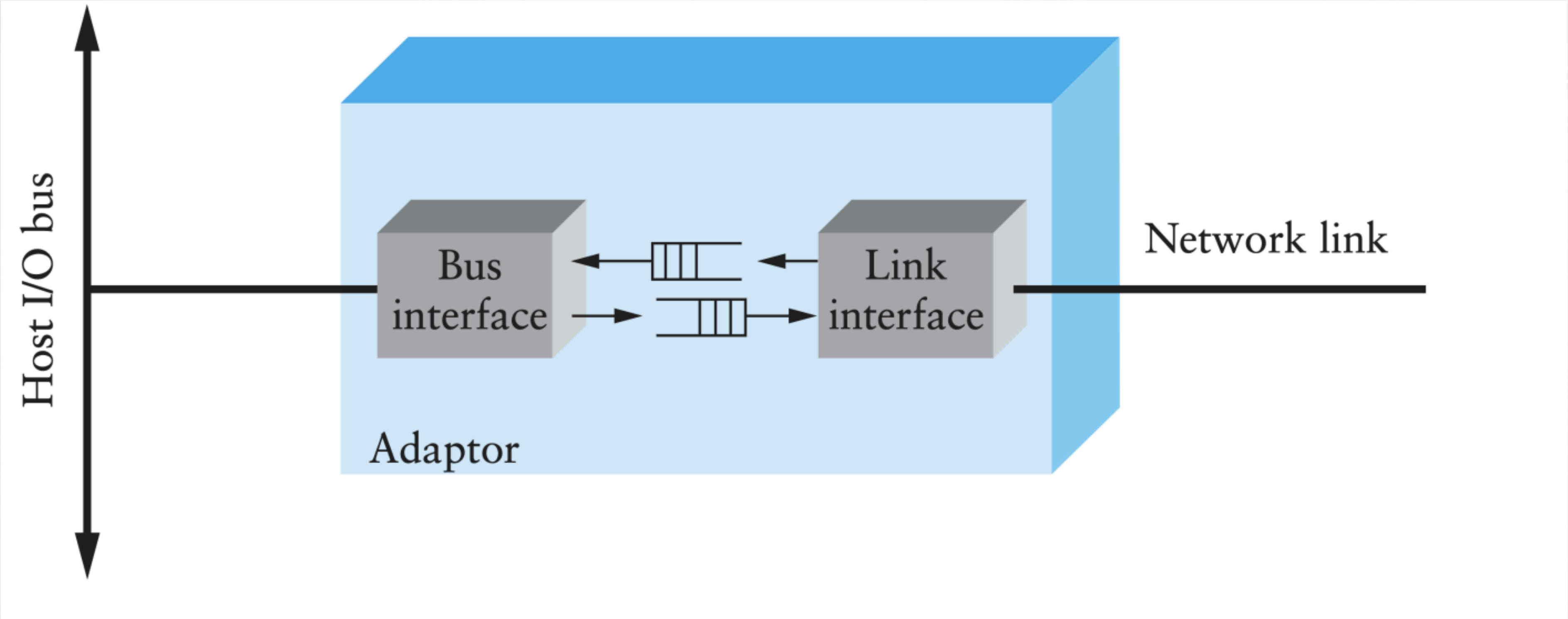
```
static inline void insw (uint16_t port, void *addr, size_t cnt)
{
  asm volatile ("rep insw" : "+D" (addr), "+c" (cnt)
                : "d" (port) : "memory");
}
```

Example : IDE Disk Driver

```
void IDE_ReadSector(int disk, int off, void *buf)
{
  outb(0x1F6, disk == 0 ? 0xE0 : 0xF0); // Select Drive
  IDEWait();
  outb(0x1F2, 1);            // Read length (1 sector = 512 B)
  outb(0x1F3, off);          // LBA low
  outb(0x1F4, off >> 8);     // LBA mid
  outb(0x1F5, off >> 16);    // LBA high
  outb(0x1F7, 0x20);         // Read command
  insw(0x1F0, buf, 256);     // Read 256 words
}

void IDEWait()
{
  // Discard status 4 times
  inb(0x1F7); inb(0x1F7);
  inb(0x1F7); inb(0x1F7);
  // Wait for status BUSY flag to clear
  while ((inb(0x1F7) & 0x80) != 0)
    ;
}
```

Example : Network Interface Card



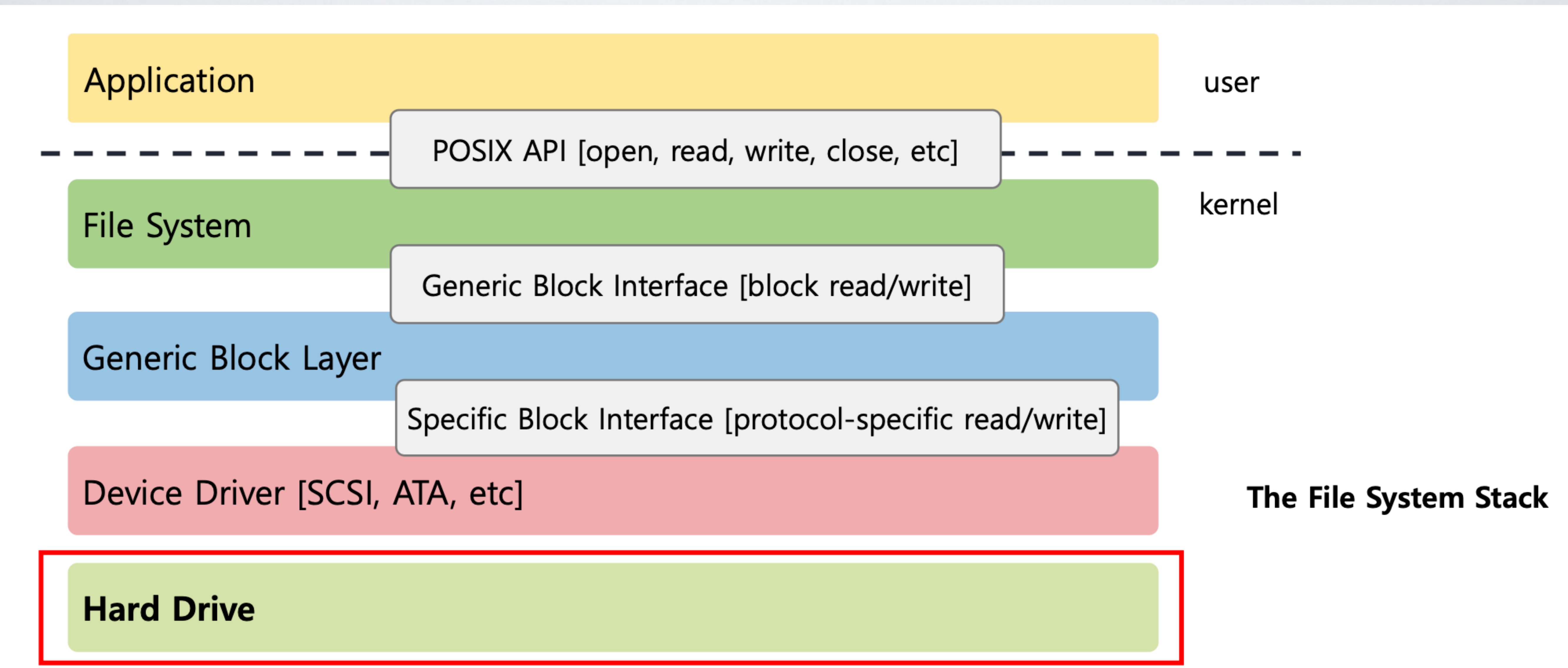
- Link interface talks to wire/fiber/antenna
- FIFOs on card provide small amount of buffering
- Bus interface logic uses DMA to move packets to and from buffers in main memory

Variety is a challenge

- Problem : there are many devices and each has its own protocol
  - Some devices are accessed by I/O ports or memory mapping or both
  - Some devices can interact by polling or interrupt or both
  - Some device can transfer data by programmed I/O or DMA or both
- ✓ Solution : abstraction
  - Build a common interface
  - Write device driver for each device
- ➡ Drivers are 70% of Linux source code

File System Abstraction

- File system specifics of which disk class it is using
  It issues block read and write request to the generic block layer



Disks

Hard Disk Drive (HDD)

Platter (aluminum coated with a thin magnetic layer)

- A circular hard surface
- Data is stored persistently by inducing magnetic changes to it
- Each platter has 2 sides, each of which is called a surface

Spindle

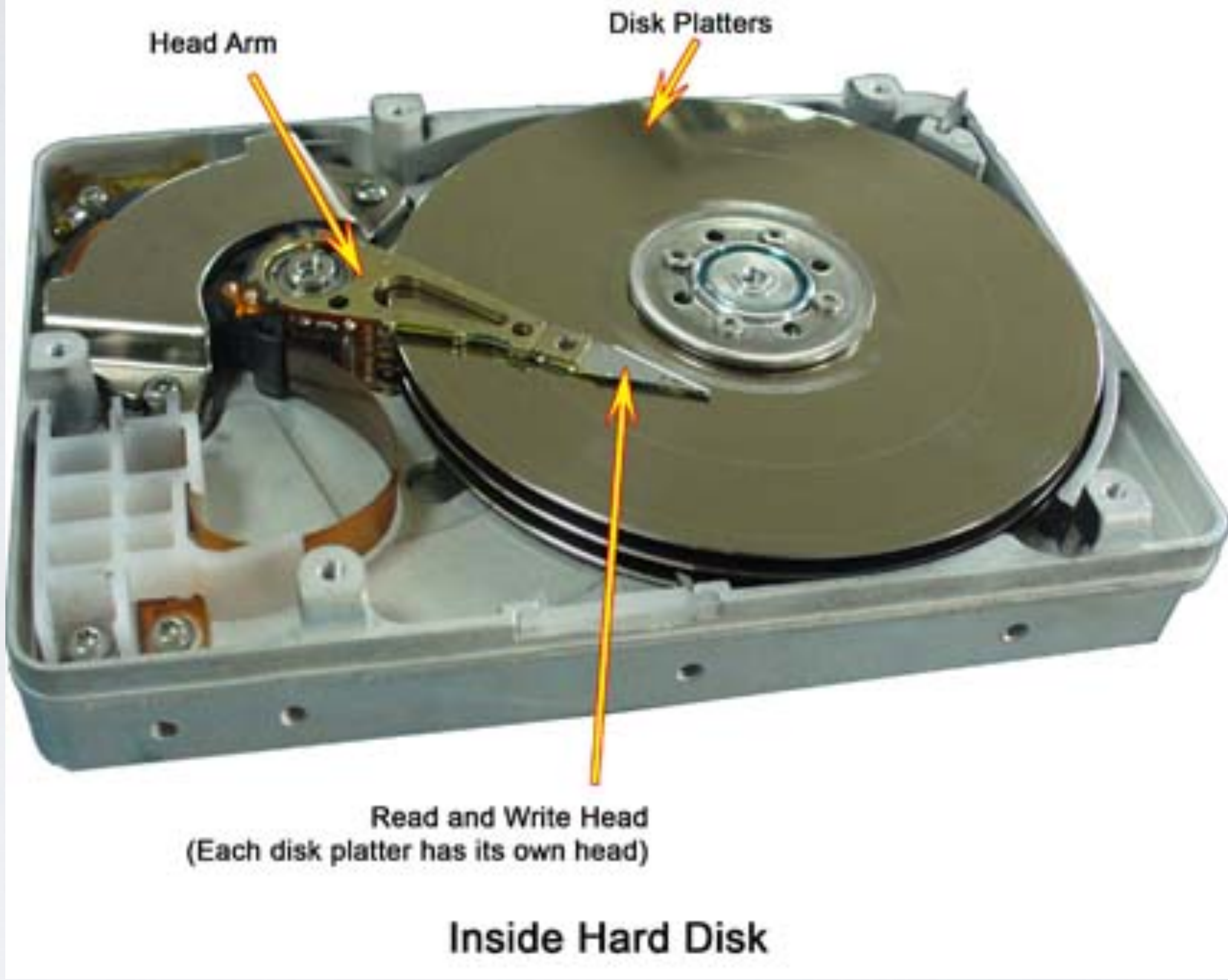
- Spindle is connected to a motor that spins the platters around
- The rate of rotations is measured in RPM (Rotations Per Minute)
  Typical modern values : 7,200 RPM to 15,000 RPM

Track

- Concentric circles of sectors
- Data is encoded on each surface in a track
- A single surface contains many thousands and thousands of tracks

Cylinder

- A stack of tracks of fixed radius
- Heads record and sense data along cylinders
- Generally only one head active at a time



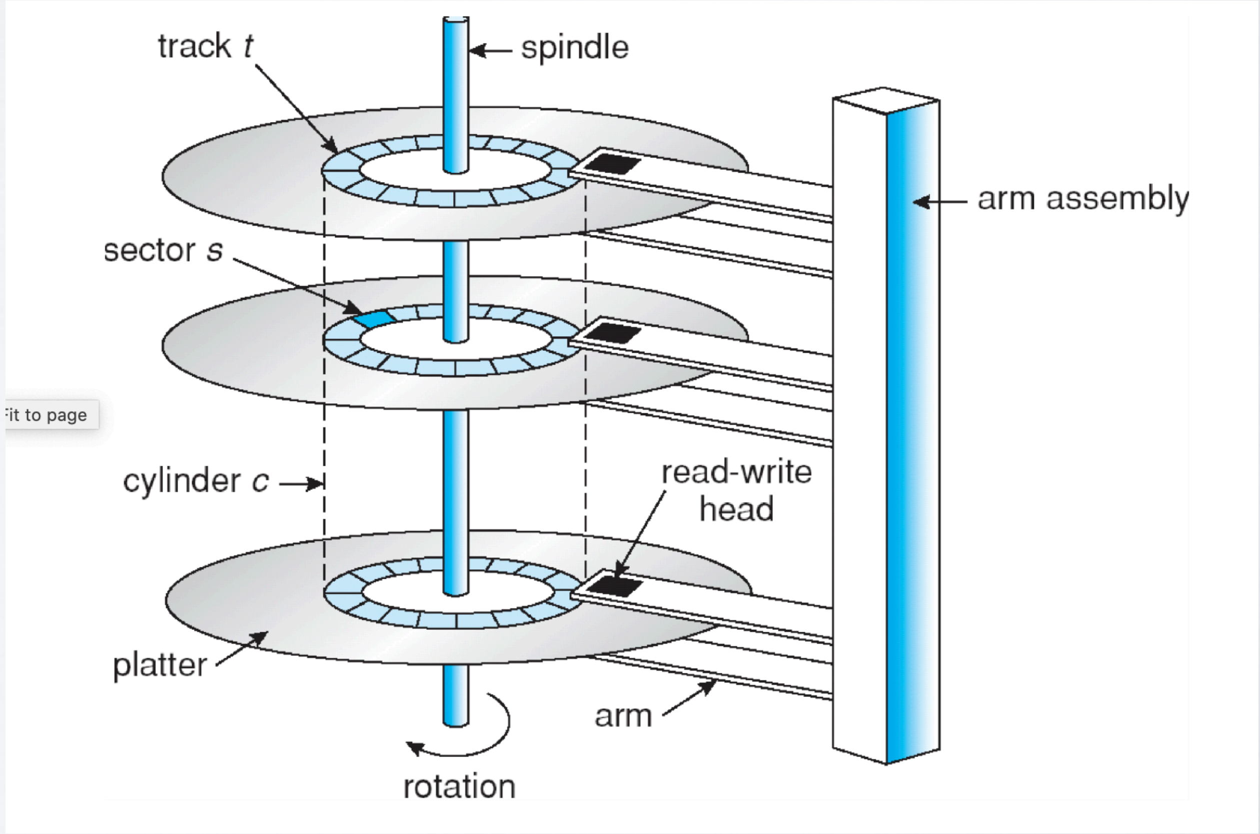
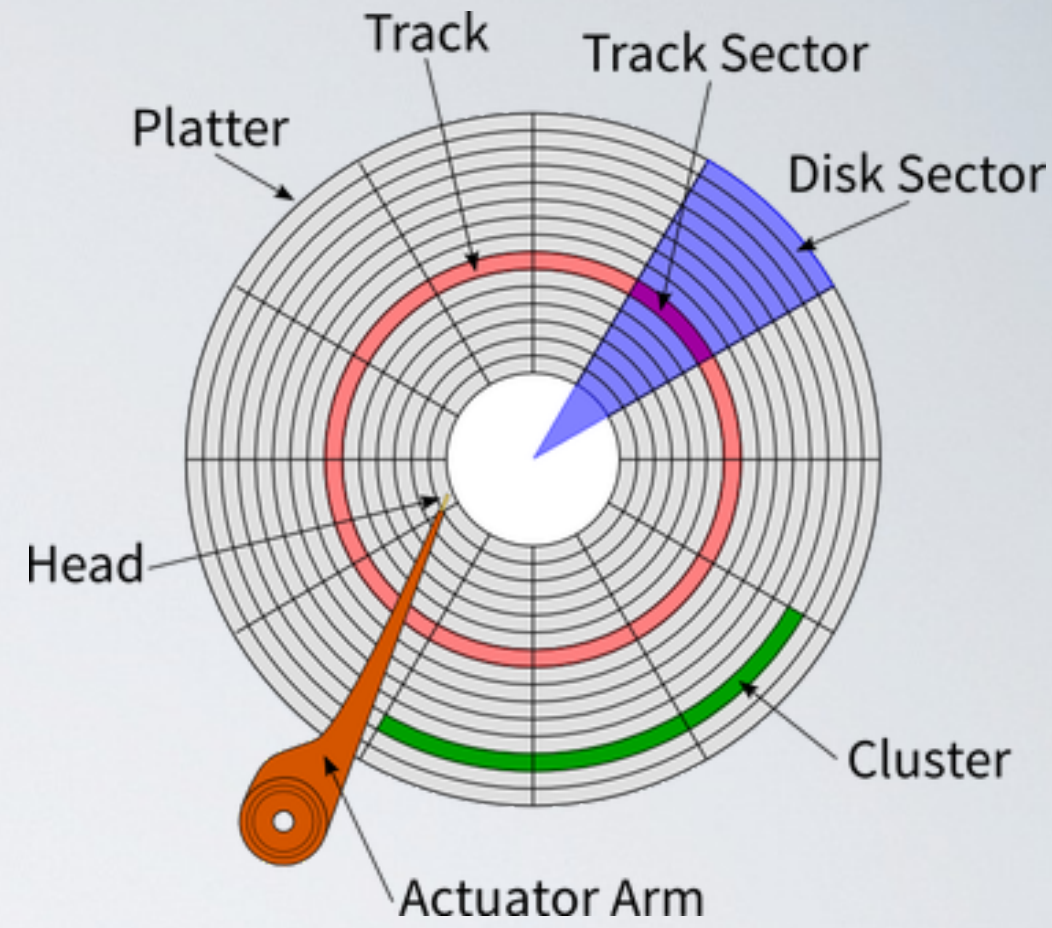


image source: https://blogs.umass.edu/Techbytes/2017/04/04/hard-drives-how-do-they-work/

HDD Interface



➡ Disk interface presents linear array of sectors

- Historically 512 Bytes but 4 KiB in "advanced format" disks
- Written atomically (even if there is a power failure)

✓ Disk maps logical sector #s to physical sectors

✓ OS doesn't know logical to physical sector mapping

image source: https://blogs.umass.edu/Techbytes/2017/04/04/hard-drives-how-do-they-work/

Seek, Rotate, Transfer

Seek - move head to above specific track

1. speedup – accelerate arm to max speed
2. coast – at max speed (for long seeks)
3. slowdown – stops arm near destination
4. settle – adjusts head to actual desired track

- Seeks is slow
  - settling alone can take 0.5 to 2ms
  - entire seek often takes 4 - 10 ms

Seek, **Rotate**, Transfer

Rotate disk until the head is above the right sector

- ➡ Depends on rotations per minute (RPM)
  With typical 7200 RPM it takes 8.3 ms / rotation

- Average rotation is slow (4.15 ms)

Seek, Rotate, **Transfer**

Data is either read from or written to the surface.

➡ Depends on RPM and sector density
With typical 100+ MB/s it takes 5μs / sector (512 bytes)

✓ Pretty Fast

Workload

So ...

- seeks are slow
- rotations are slow
- transfers are fast

What kind of workload is fastest for disks?

- Sequential : access sectors in order (transfer dominated)
- Random : access sectors arbitrarily (seek+rotation dominated)

➡ Disk Scheduler decides which I/O request to schedule next

- First Come First Served (FCFS)
- Shortest Seek Time First (SSTF)
- Elevator Scheduling (SCAN) commonly used on Unix

Solid State Drive (SSD)

- ➡ Completely solid state (no moving parts), remembers data by storing charge (like RAM)
- ✓ Same interface as HDD (linear array of sectors)
- ✓ No mechanical seek and rotation times to worry about (SSD are way faster than HDD)
- ✓ Lower power consumption and heat (better for mobile devices)
- ◉ More expensive than HDD yet (but getting cheaper)
- ◉ Limited durability as charge wears out over time (but improving)
- ◉ Limited # overwrites possible
  - Blocks wear out after 10,000 (MLC) – 100,000 (SLC) erases
  - Requires Flash Translation Layer (FTL) to provide wear levelling, so repeated writes to logical block don't wear out physical block
  - FTL can seriously impact performance