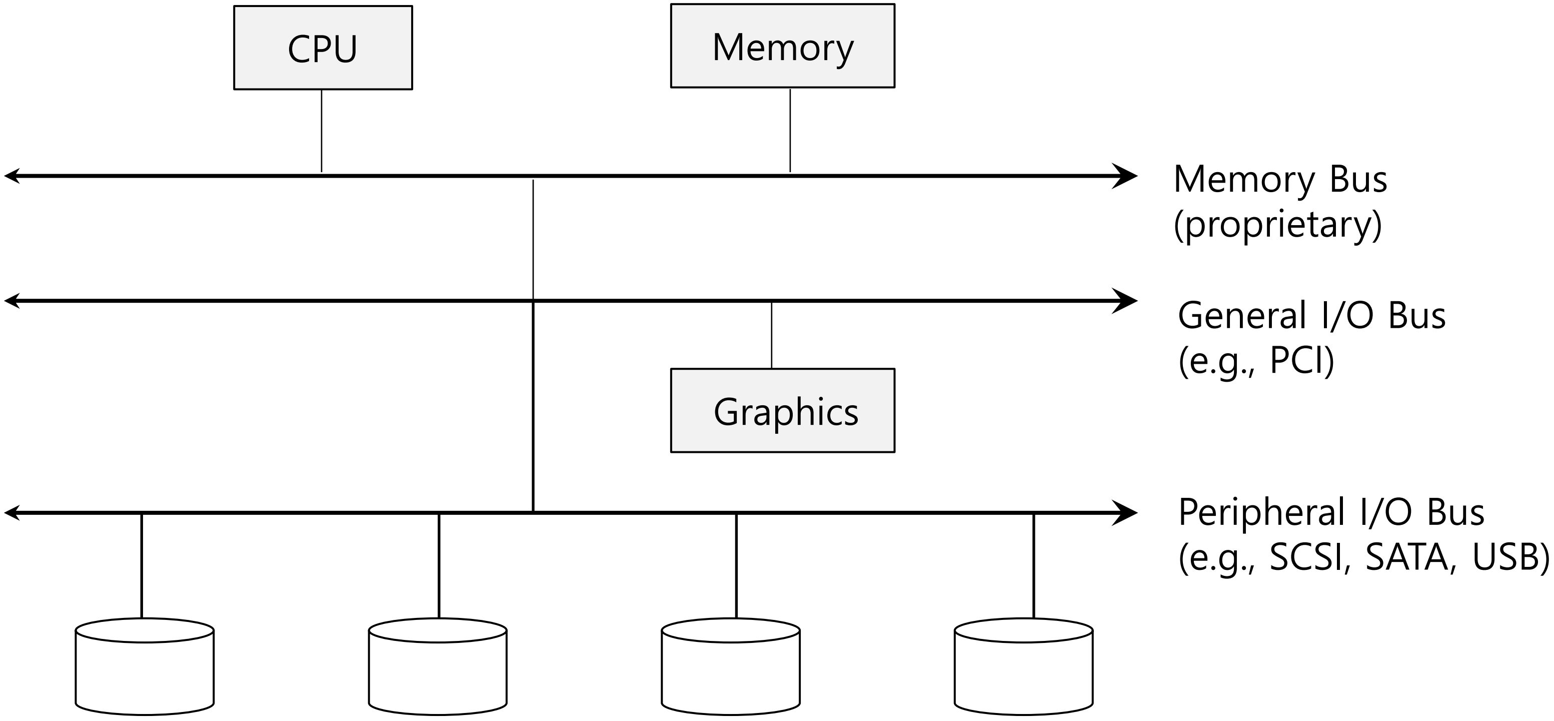
36. I/O Devices

Operating System: Three Easy Pieces

I/O Devices

- I/O is **critical** to computer system to **interact with systems.**
- Issue :
  - How should I/O be integrated into systems?
  - What are the general mechanisms?
  - How can we make the efficiently?

Structure of input/output (I/O) device



Prototypical System Architecture

CPU is attached to the main memory of the system via some kind of memory bus.

Some devices are connected to the system via a general I/O bus.

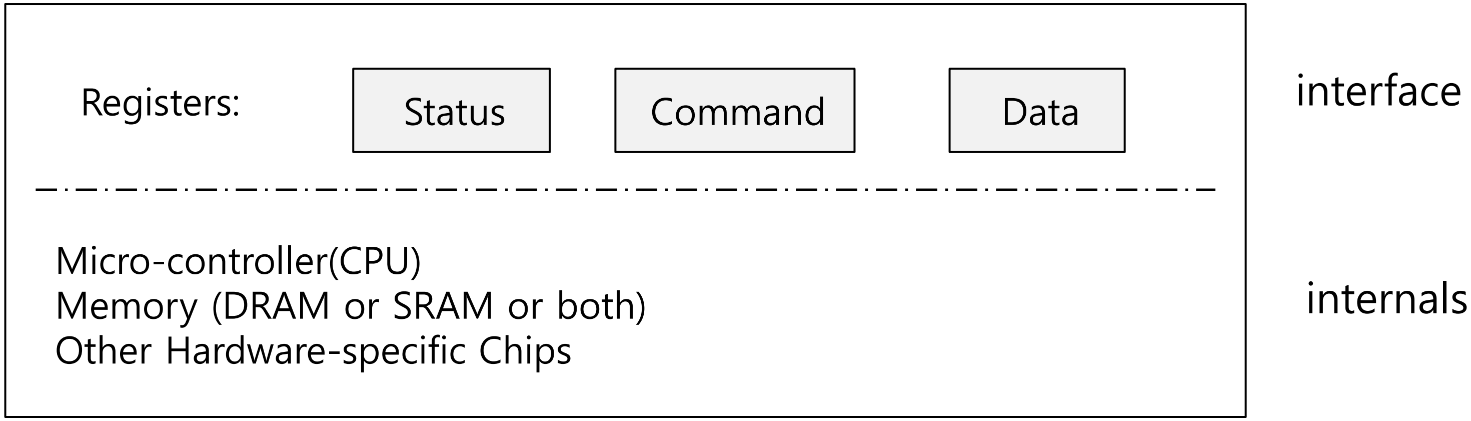
I/O Architecture

- Buses
  - Data paths that provided to enable information between CPU(s), RAM, and I/O devices.

- I/O bus
  - Data path that connects a CPU to an I/O device.
  - I/O bus is connected to I/O device by three hardware components: I/O po rts, interfaces and device controllers.

Canonical Device

- A typical device has two important components.
  - **Hardware interface** allows the system software to control its operation.
  - **Internals** which are implementation specific.



Canonical Device

Hardware interface of Canonical Device

- **status register**
  - The current status of the device
- **command register**
  - Tells the device to perform a certain task
- **data register**
  - Used to pass data to the device, or get data from the device

By reading and writing these three registers,
the operating system can control device behavior.

Hardware interface of Canonical Device (Cont.)

- Typical interaction example

```
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
```

Polling

- The operating system waits until the device is ready by **repeatedly** reading the status register (polling?).
  - Its simple to do and works OK if there is frequently available data.
  - **However, it wastes CPU time waiting for the device**.
    - Switching to another ready process makes for better CPU utilization.

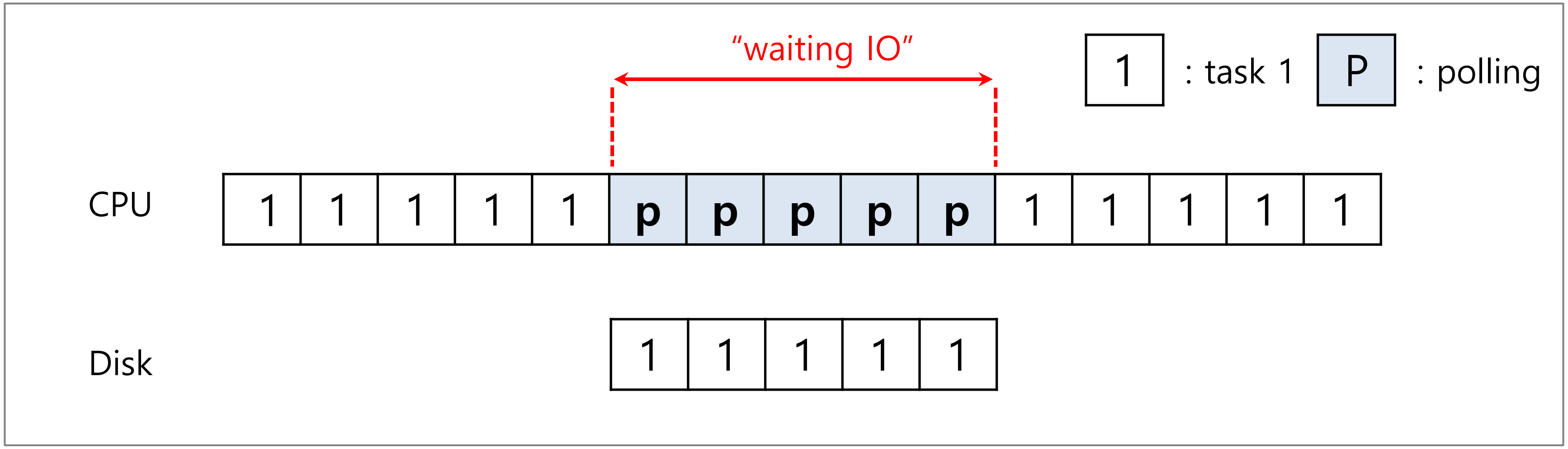
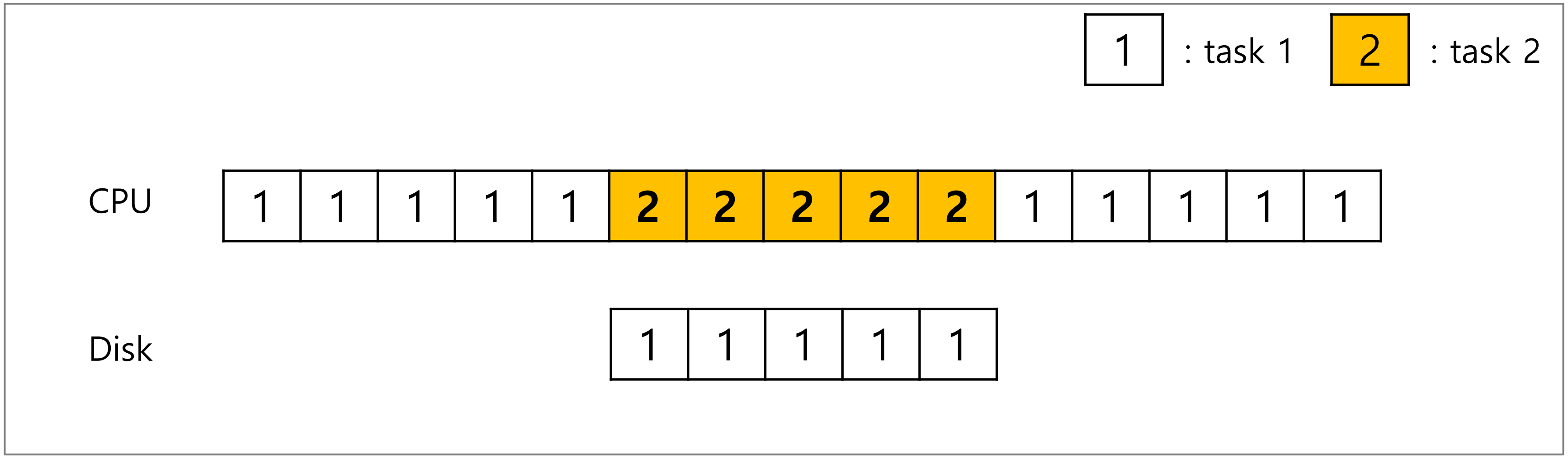


Diagram of CPU utilization by polling

interrupts

- **So lets put the I/O requesting process to sleep** and context switch to another process.
- When the I/O device is finished, wake the process waiting for the I/O by **interrupt**.
  - Ensures efficient utilization of both the **CPU and the disk.**



CPU	1	1	1	1	1	2	2	2	2	2	1	1	1	1	1
Disk						1	1	1	1	1					

Diagram of CPU utilization by interrupt

Polling vs interrupts

- *However,* **"interrupts are not always the best solution"**
  - If, device performs very quickly, an interrupt will "slow down" the system.
  - Because **a context switch is expensive (switching to another process)**

If an I/O device is fast → polling is best.
If it is slow → interrupts are better.

CPU is once again over-burdened

- CPU **wastes a lot of time** to copy a *large chunk of data* from memory to the device.

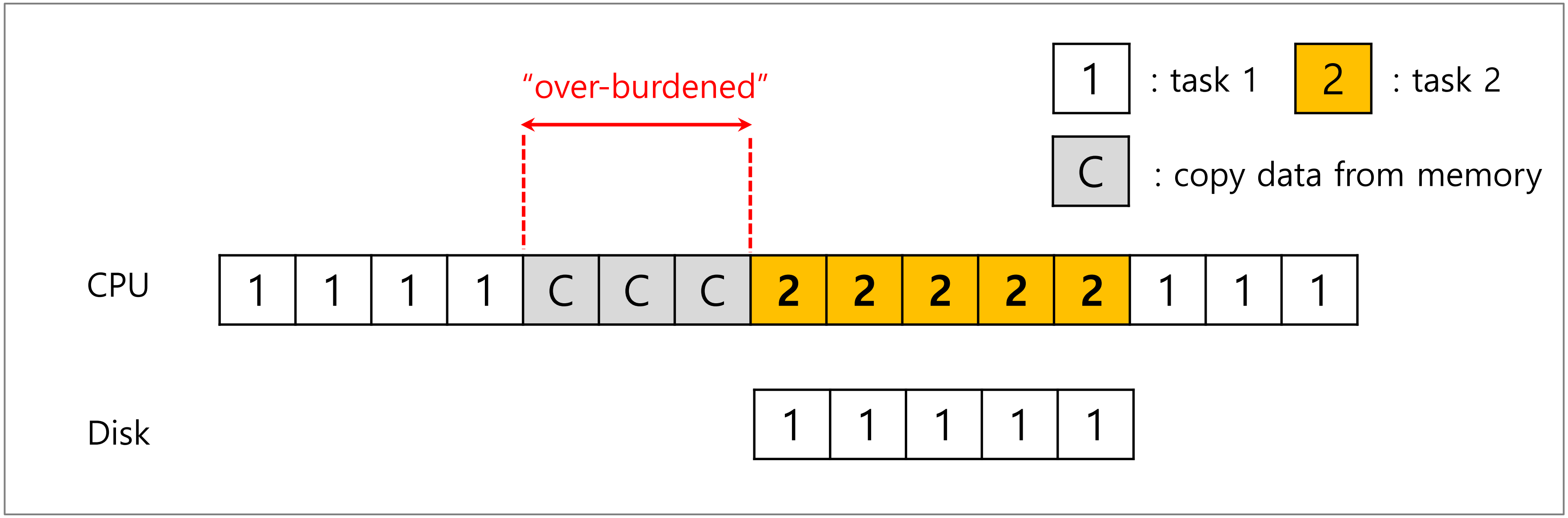


Diagram of CPU utilization (programmed I/O- CPU involved in transferring memory)

DMA (Direct Memory Access)

- **Copy data** in memory by knowing
  - where the data lives in memory
  - how much data to copy
- When completed, DMA raises an interrupt, I/O begins on Disk.

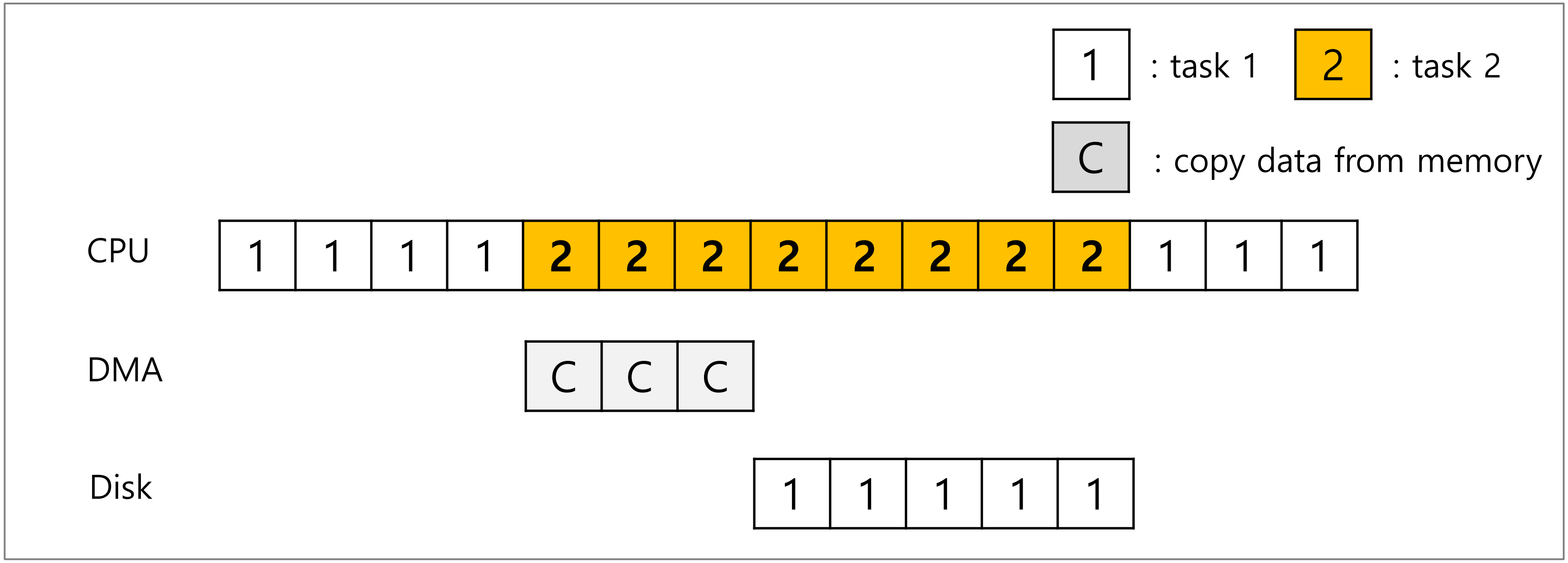


Diagram of CPU utilization by DMA

Device interaction

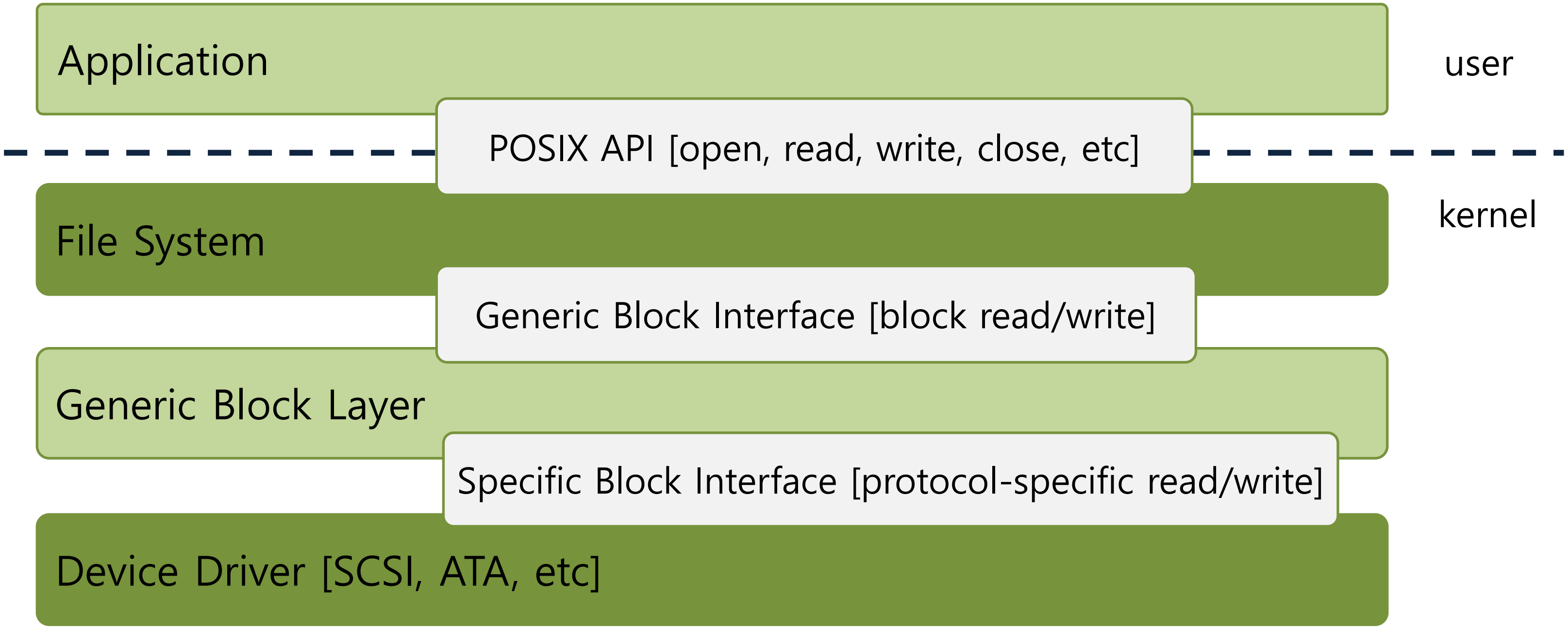
- How the OS communicates with the **device**?
- Solutions
  - I/O instructions: a way for the OS to send data to specific device registers.
    - Ex) `in` and `out` instructions on x86
  - memory-mapped I/O
    - Device registers available as if they were memory locations.
    - The OS `load` (to read) or `store` (to write) to the device instead of main memory.

Device interaction (Cont.)

- How the OS interacts with **different specific interfaces**?
  - Ex) We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drivers, and so on.
- Solutions: Abstraction
  - Abstraction encapsulate any specifics of device interaction.

File system Abstraction

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


Application
user
POSIX API [open, read, write, close, etc]
kernel
File System
Generic Block Interface [block read/write]
Generic Block Layer
Specific Block Interface [protocol-specific read/write]
Device Driver [SCSI, ATA, etc]

The File System Stack

Problem of File system Abstraction

- If there is a device having many special capabilities, these capabilities will go unused in the generic interface layer.
  - Ex. SCSI disk have rich error reporting but IDE disks do not. Program to lowest common denominator so OS cannot see rich SCSI errors.
- Over 70% of OS code is found in device drivers.
  - Device drivers are specialty programs that are written to communicate with specific (or general) devices. They are needed because different devices have different protocols (SCSI verses IDE verse eIDE verse PCI etc) Drivers are middleware that handles communication between the OS (via Generic Block Layer) and any device you plug into your system.
  - Sometimes written by non-experts. So they are a primary contributor to **kernel crashes**.

A Simple IDE Disk Driver

- Four types of register
  - Control, command block, status and error
  - Memory mapped IO
  - `in` and `out` I/O instruction

- Control Register:

  Address 0x3F6 = 0x80 (0000 1RE0): R=reset, E=0 means "enable interrupt”

- Command Block Registers:

  Address 0x1F0 = Data Port

  Address 0x1F1 = Error

  Address 0x1F2 = Sector Count

  Address 0x1F3 = LBA low byte

  Address 0x1F4 = LBA mid byte

  Address 0x1F5 = LBA hi byte

  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive

  Address 0x1F7 = Command/status

- Status Register (Address 0x1F7):

7	6	5	4	3	2	1	0
BUSY	READY	FAULT	SEEK	DRQ	CORR	IDDEX	ERROR

- Error Register (Address 0x1F1): (check when Status ERROR==1)

7	6	5	4	3	2	1	0
BBK	UNC	MC	IDNF	MCR	ABRT	T0NF	AMNF

  - BBK = Bad Block
  - UNC = Uncorrectable data error
  - MC = Media Changed
  - IDNF = ID mark Not Found
  - MCR = Media Change Requested
  - ABRT = Command aborted
  - T0NF = Track 0 Not Found
  - AMNF = Address Mark Not Found

- **Wait for drive to be ready**. Read Status Register (0x1F7) until drive is not busy and READY.
- **Write parameters to command registers**. Write the sector count, logical block address (LBA) of the sectors to be accessed, and drive number (master=0x00 or slave=0x10, as IDE permits just two drives) to command registers (0x1F2-0x1F6).
- **Start the I/O**. by issuing read/write to command register. Write READ—WRITE command to command register (0x1F7).
- **Data transfer (for writes)**: Wait until drive status is READY and DRQ (drive request for data); write data to data port.
- **Handle interrupts**. In the simplest case, handle an interrupt for each sector transferred; more complex approaches allow batching and thus one final interrupt when the entire transfer is complete.
- Error handling. After each operation, read the status register. If the ERROR bit is on, read the error register for details.

```
static int ide_wait_ready() {
   while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
        {}; // loop while drive is busy or not ready
}
```

Only interested in if the DRIVE is busy, read its status word from 0x1f7 and mask out the busy bit (The BSY bit is bit 7, if busy it’s 1xxxxxxx, if not it’s 0xxxxxxx)

```
Disk status =  xxxxxxxx     //from inb(0x1f7)
BSY mask    =& 10000000     // logical and with IDE_BSY
Result      =  x0000000
```

if drive busy then result = 10000000 and first boolean expression above is True

if drive NOT busy then result = 00000000 and first boolean expression above is False

```
static void ide_start_request(struct buf *b) {
    ide_wait_ready();
    outb(0x3f6, 0); // generate interrupt
    outb(0x1f2, 1); // how many sectors?
    outb(0x1f3, b->sector & 0xff); // LBA goes here ...
    outb(0x1f4, (b->sector >> 8) & 0xff); // ... and here
    outb(0x1f5, (b->sector >> 16) & 0xff); // ... and here!
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
    if(b->flags & B_DIRTY){
        outb(0x1f7, IDE_CMD_WRITE); // this is a WRITE
        outsl(0x1f0, b->data, 512/4); // transfer data too!
    } else {
        outb(0x1f7, IDE_CMD_READ); // this is a READ (no data)
    }
}
```

```
void ide_rw(struct buf *b) {
    acquire(&ide_lock);
    for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
    ; // walk queue
    *pp = b; // add request to end
    if (ide_queue == b) // if q is empty
        ide_start_request(b); // send req to disk
    while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
        sleep(b, &ide_lock); // wait for completion
    release(&ide_lock);
}
```

```
void ide_intr() {
   struct buf *b;
   acquire(&ide_lock);
   if (!(b->flags & B_DIRTY) && ide_wait_ready(1) >= 0)
      insl(0x1f0, b->data, 512/4); // if READ: get data
   b->flags |= B_VALID;
   b->flags &= ~B_DIRTY;
   wakeup(b); // wake waiting process
   if ((ide_queue = b->qnext) != 0) // start next request
      ide_start_request(ide_queue); // (if one exists)
   release(&ide_lock);
}
```

- Disclaimer: This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.