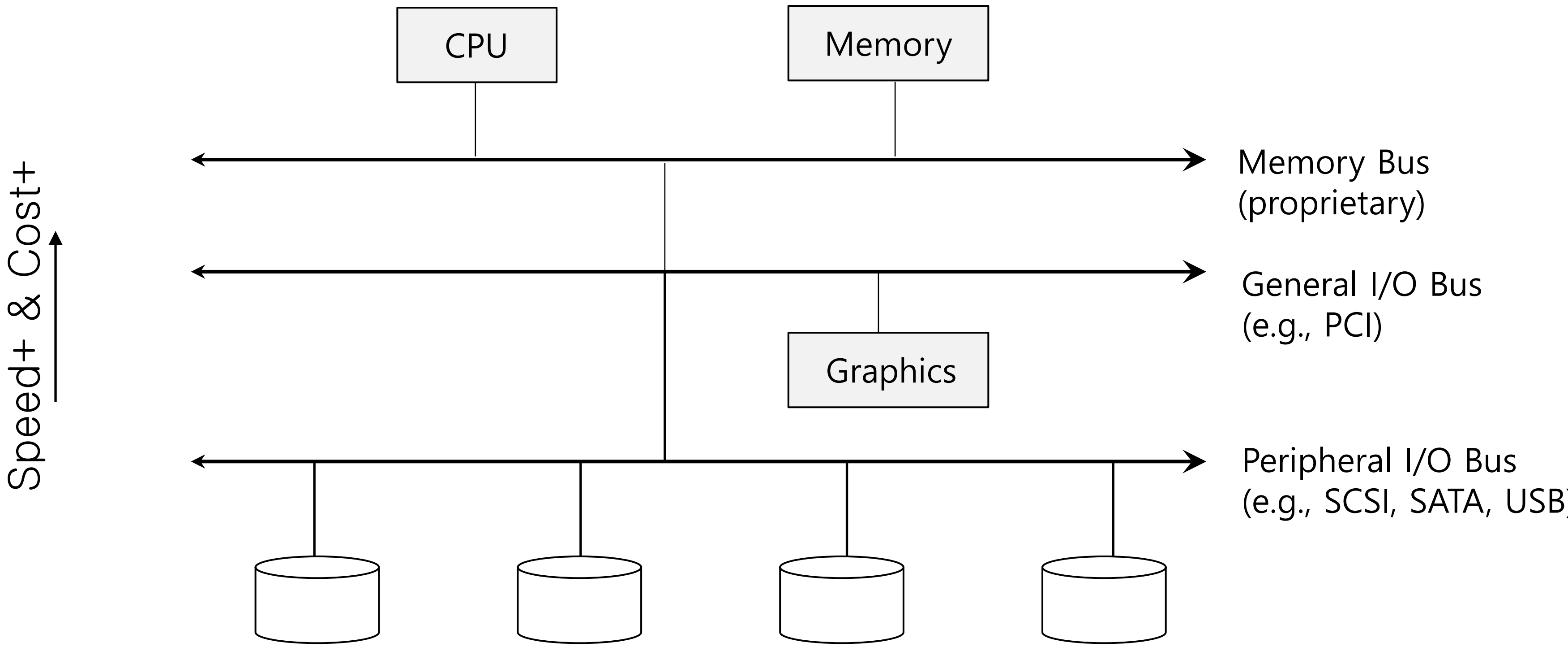
36. I/O Devices

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.

Why not a flat design? (like in the early days)

I/O Architecture

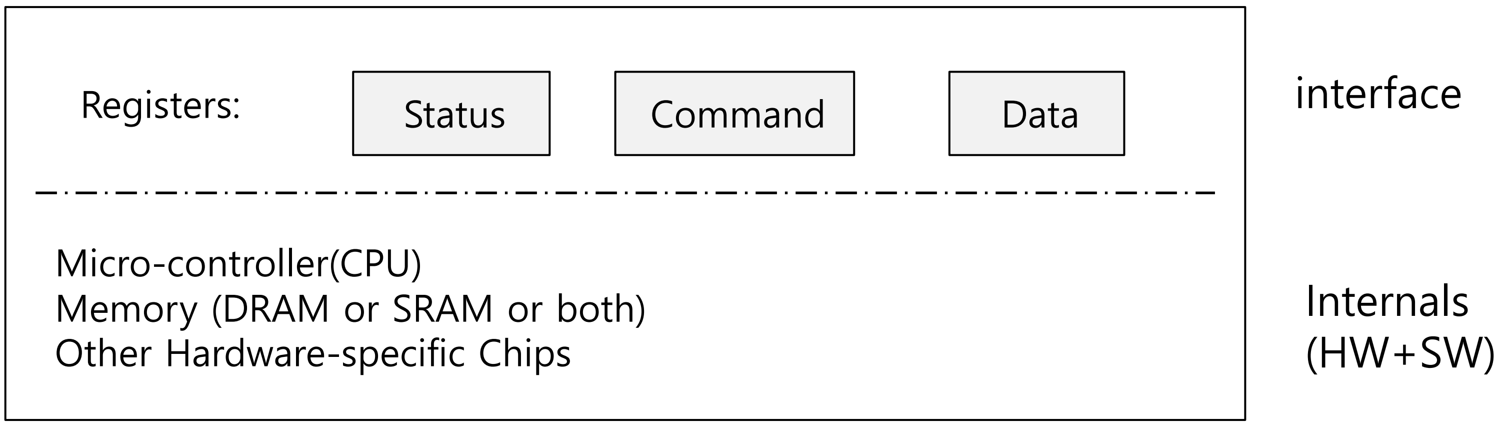
- **Buses**
  - Data paths that provided to enable information between CPU(s), RAM, and I/O devices.

- **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 ports, interfaces and device controllers.

- In current system (due to scalability limitations of the buses) some high-speed buses have migrated to point-to-point networks

Canonical Device

- Canonical Devices has two important components.
  - **Hardware interface** allows the system software to control its operation.
  - **Internals** which is implementation specific.



Canonical Device

Hardware interface of Canonical Device

- **status register**
  - See the current status of the device
- **command register**
  - Tell the device to perform a certain task
- **data register**
  - Pass data to the device, or get data from the device

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

Hardware interface of Canonical Device (Cont.)

- Typical interaction example (Programmed I/O or PIO)

```
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

- Operating system waits until the device is ready by **repeatedly** reading the status register.
  - Positive aspect is simple and working.
  - **However, it wastes CPU time just waiting for the device**.
    - Switching to another ready process is better utilizing the CPU.

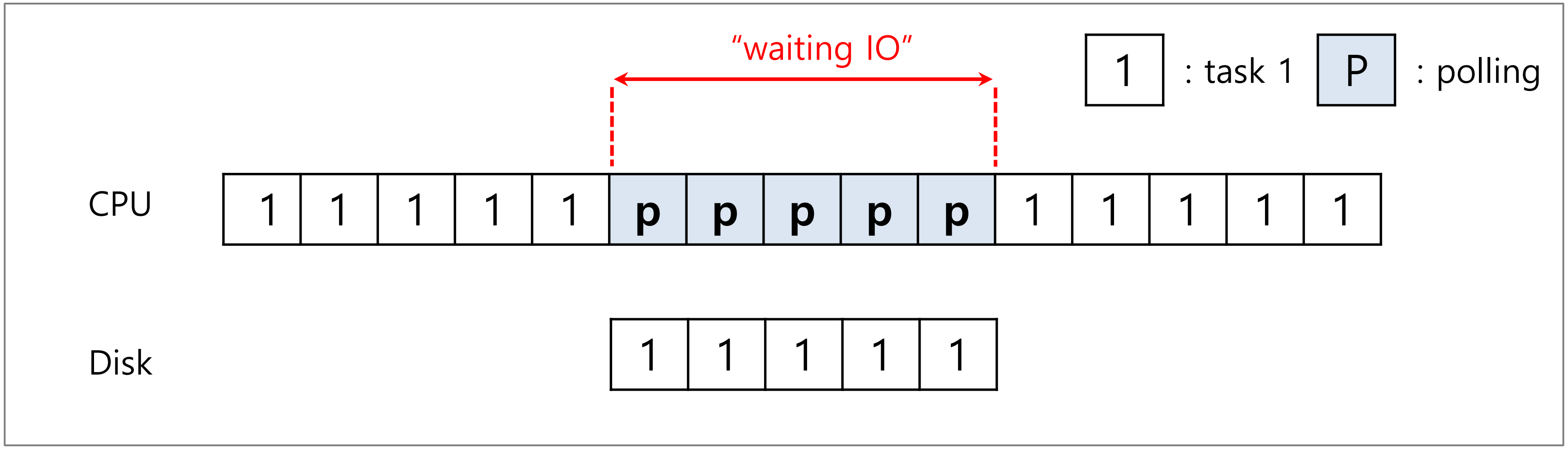


Diagram of CPU utilization by polling

interrupts

- **Put the I/O request process to sleep** and context switch to another.
- When the device is finished, wake the process waiting for the I/O by **interrupt** (via interrupt handler or *Interrupt Service Routine ISR*)
  - Positive aspect is allow to **CPU and the disk are properly utilized.**

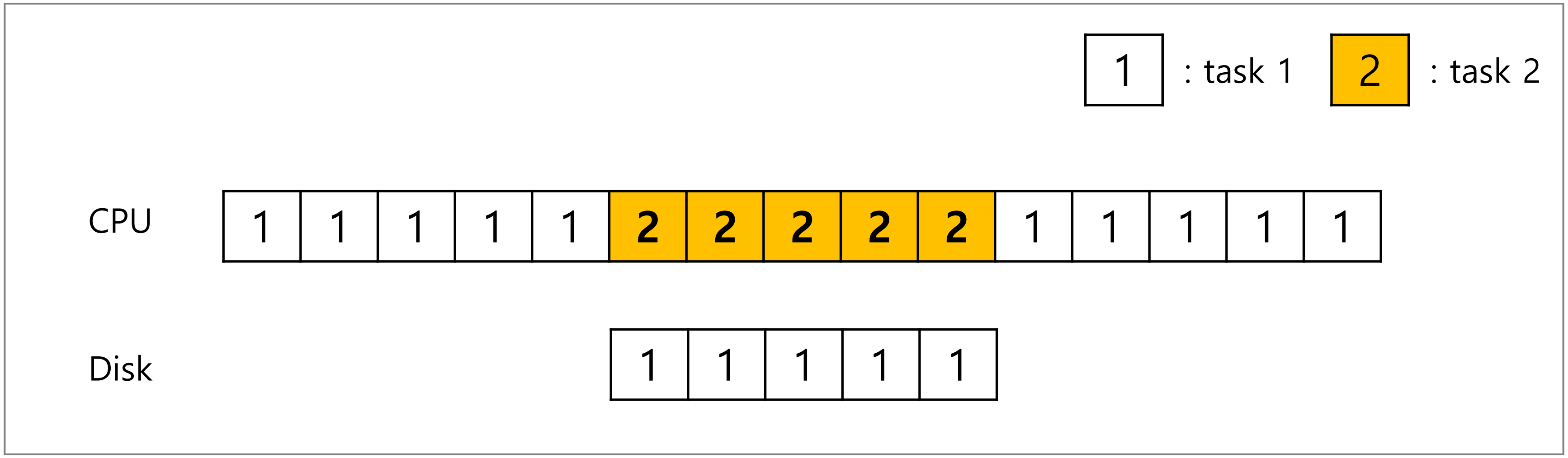


Diagram of CPU utilization by interrupt

Polling vs interrupts

- *However,* **"interrupts is not always the best solution"**
  - If, device performs very quickly (for example, at first poll the operation is done), interrupt will "slow down" the system.
  - Because **context switch is expensive (switching to another process)**

If a device is fast → poll is best.
If it is slow → interrupts is better.

- Hybrid approach
  - If poll too slow go to interrupts
- Coalescing interrupts

CPU is once again over-burdened

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

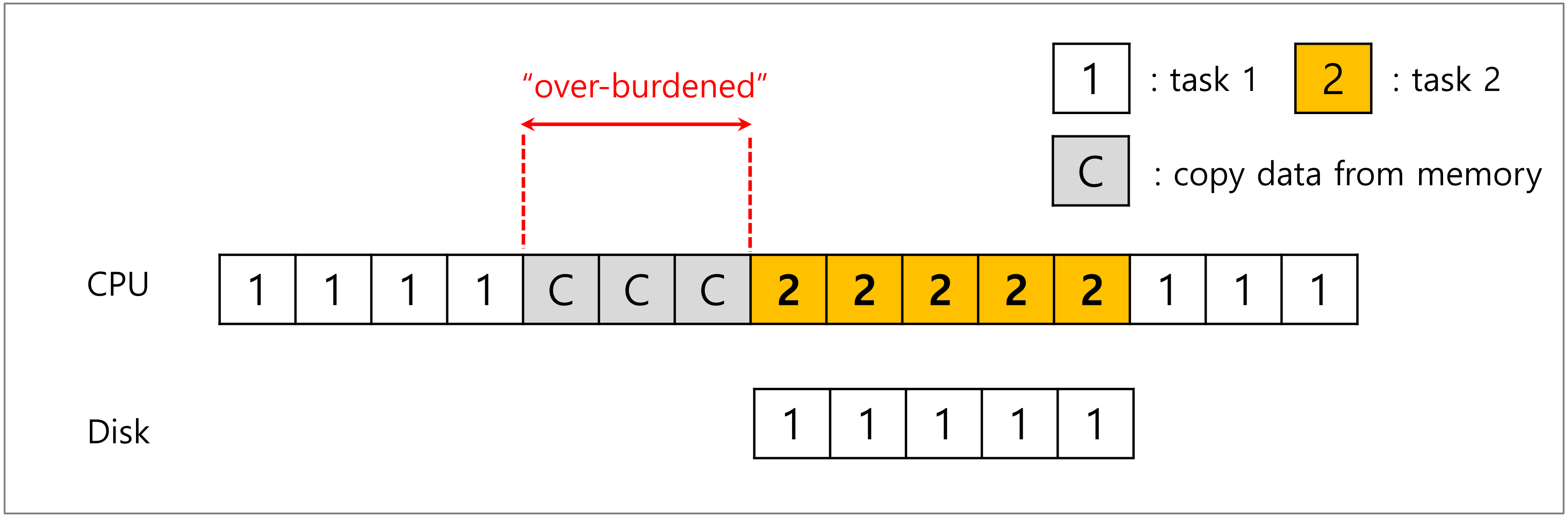


Diagram of CPU utilization

DMA (Direct Memory Access)

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

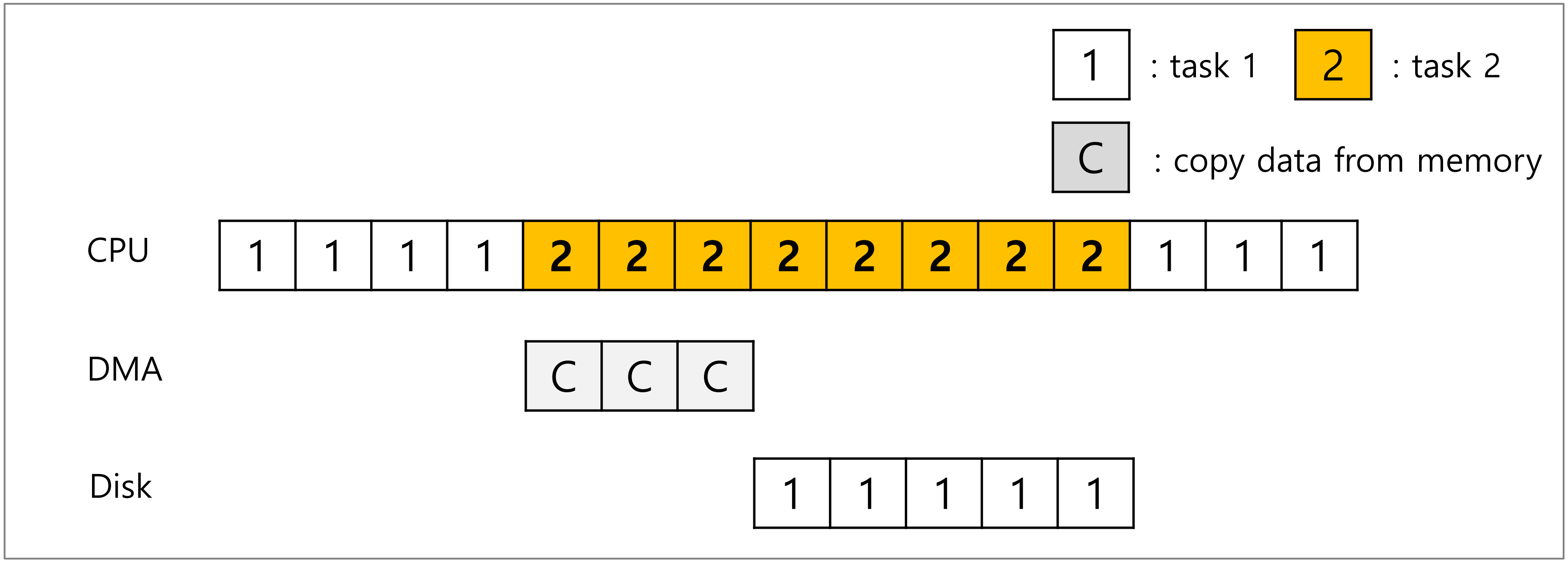


Diagram of CPU utilization by DMA

Device interaction

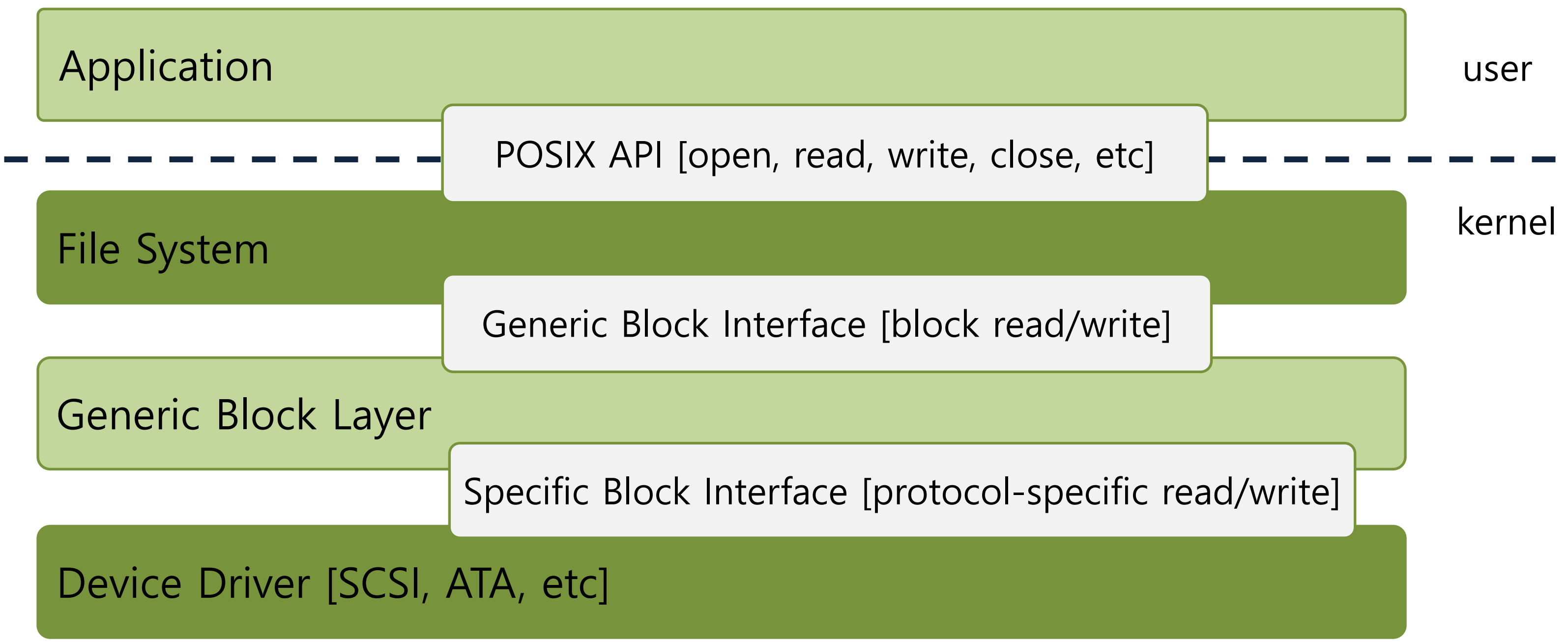
- How the CPU communicates with the **device**?
- Approaches
  - **I/O instructions**: a way for the OS to send data to specific device registers.
    - Ex) `in` and `out` instructions on x86
    - Separate I/O and memory buses in early days
  - **memory-mapped I/O**
    - Device registers available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

Fitting Into The OS: The Device Driver

- How the OS interact 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.
- **Solution**: Abstraction
  - Abstraction encapsulate any specifics of device interaction.
  - Only the lowest level should be aware of the specifics: called Device driver

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.



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 devices have a rich error reporting that are mostly unused in Linux because IDE/ATA had very limited capabilities

- Over 70% of OS code is found in device drivers.
  - Any device drivers are needed because you might plug it to your system.
  - They are primary contributor to **kernel crashes**, making **more bugs**.
  - Device signing in current windows system has improved its resiliency greatly

Case Study: A Simple IDE Disk Driver (xv6 uses QEMU IDE)

- Four types of register
  - Control, command block, status and error
  - Mapped to I/O addresses
  - `in` and `out` I/O instruction

- Book code doesn't not match with current version

- 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 (Logical Block Address)

    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.

xv6: I/O buffer (node struct)

```
struct buf {
    int flags;
    uint dev;
    uint sector;
    struct buf *prev; // LRU cache list
    struct buf *next;
    struct buf *qnext; // disk queue
    uchar data[512];
};
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

xv6 code: Queues request (if IDE not avail) or issue the req.

```
void ide_rw(struct buf *b) {
   acquire(&ide_lock);
   for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
   ;                              // walk queue (beware 2nd term)
   *pp = b;                       // add request to end
   if (ide_queue == b)            // if q was empty (only has b)
      ide_start_request(b);       // send req to disk
   while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
      sleep(b, &ide_lock); // wait for completion and rel. lock
   release(&ide_lock);
}
```

xv6 code: intercedes with the driver

```
static void ide_start_request(struct buf *b) {
    ide_wait_ready();
    outb(0x3f6, 0);                          // generate interrupt
    outb(0x1f2, 1);                          // how many sectors to read/write?
    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)); //M or S?
    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)
    }
}
```

xv6 code: just check the device is ready and not busy

```
static int ide_wait_ready() {
        while (((int r = inb(0x1f7)) & IDE_BSY) ||
                !(r & IDE_DRDY))
        ; // loop until drive isn't busy
}
```

Device should be initialized somewhere else (at boot)

xv6 code: interrupt handler

```
void ide_intr() {
    struct buf *b;
    acquire(&ide_lock);
    //take b as the first element in the ide_queue (not shown)
    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 (equivalent to signal)
    if ((ide_queue = b->qnext) != 0) // start next request
        ide_start_request(ide_queue); // (if one exists)
    release(&ide_lock);
}
```

Current xv6 code (kernel/ide.c)

```
static void
idestart(struct buf *b)
{
  if(b == 0)
    panic("idestart");

  idewait(0);
  outb(0x3f6, 0);  // generate interrupt
  outb(0x1f2, 1);  // number of sectors
  outb(0x1f3, b->sector & 0xff);
  outb(0x1f4, (b->sector >> 8) & 0xff);
  outb(0x1f5, (b->sector >> 16) & 0xff);
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);
    outsl(0x1f0, b->data, 512/4);
  } else {
    outb(0x1f7, IDE_CMD_READ);
  }
}
```

```
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void
iderw(struct buf *b)
{
  struct buf **pp;

  if(!(b->flags & B_BUSY))
    panic("iderw: buf not busy");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);

  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  // Assuming will not sleep too long: ignore proc->killed.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }

  release(&idelock);
}
```

```c
static int
idewait(int checkerr)
{
  int r;

  while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
    ;
  if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
    return -1;
  return 0;
}

void
ideinit(void)
{
  int i;

  initlock(&idelock, "ide");
  picenable(IRQ_IDE);
  ioapicenable(IRQ_IDE, ncpu - 1);
  idewait(0);

  // Check if disk 1 is present
  outb(0x1f6, 0xe0 | (1<<4));
  for(i=0; i<1000; i++){
    if(inb(0x1f7) != 0){
      havedisk1 = 1;
      break;
    }
  }

  // Switch back to disk 0.
  outb(0x1f6, 0xe0 | (0<<4));
}
```

```c
void
ideintr(void)
{
  struct buf *b;

  // Take first buffer off queue.
  acquire(&idelock);
  if((b = idequeue) == 0){
    release(&idelock);
    // cprintf("spurious IDE interrupt\n");
    return;
  }
  idequeue = b->qnext;

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, 512/4);

  // Wake process waiting for this buf.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);

  // Start disk on next buf in queue.
  if(idequeue != 0)
    idestart(idequeue);

  release(&idelock);
}
```

- Disclaimer: This lecture slide set has been used in AOS course at University of Cantabria by V.Puente. 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 Arpaci-Dusseau (at University of Wisconsin)