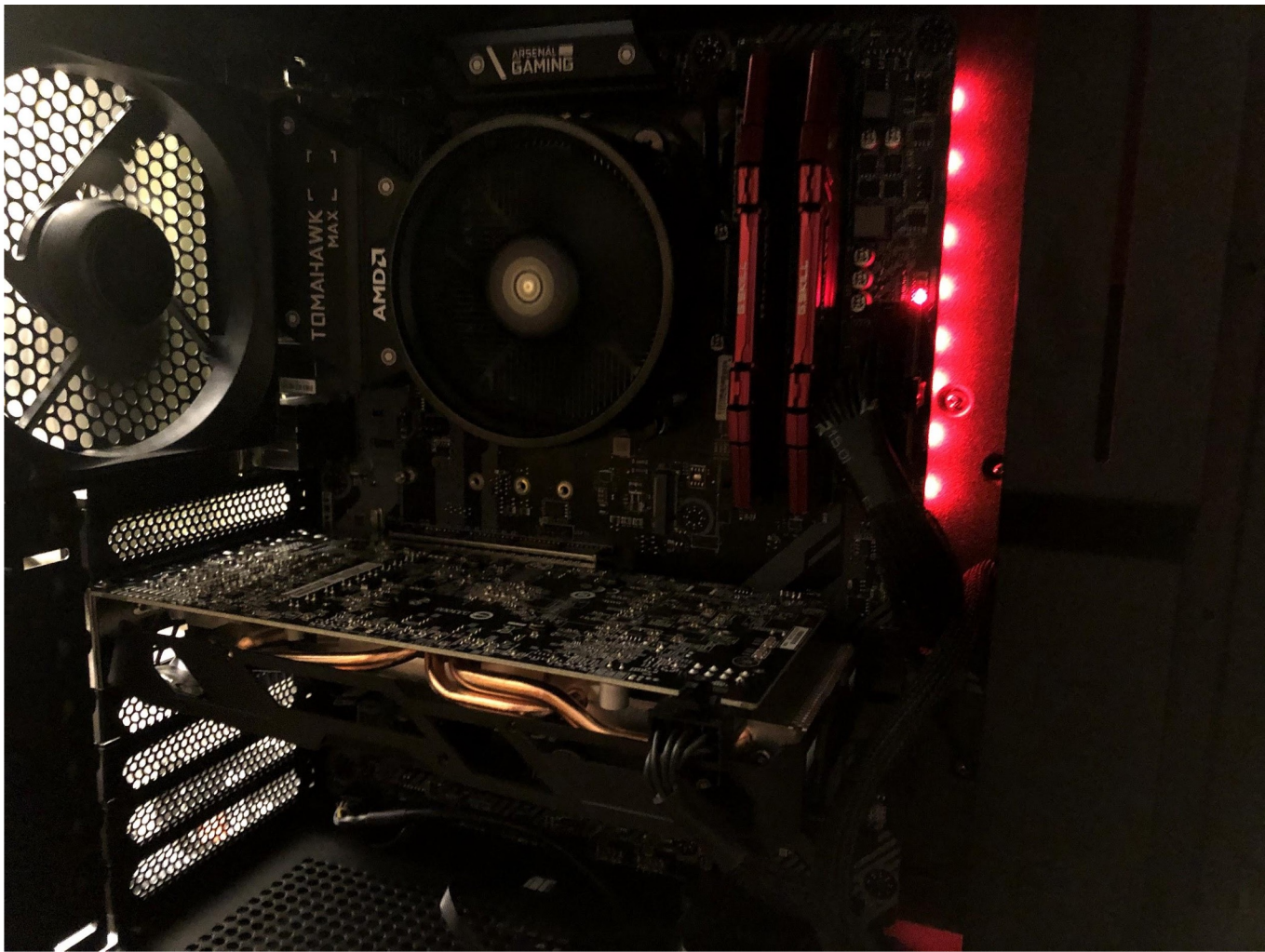




# 377 Operating Systems

I/O





# Today: I/O Systems

- How does I/O hardware influence the OS?
- What I/O services does the OS provide?
- How does the OS implement those services?
- How can the OS improve the performance of I/O?

# Bus Structure

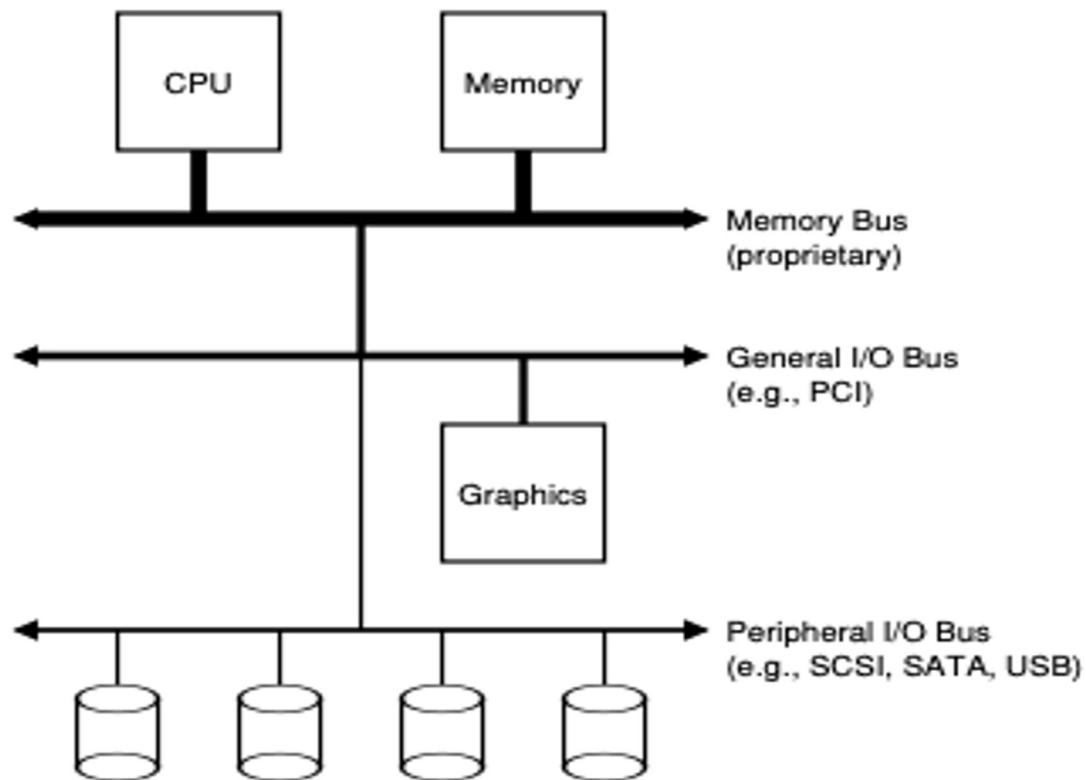


Figure 36.1: Prototypical System Architecture

# Design of a bus

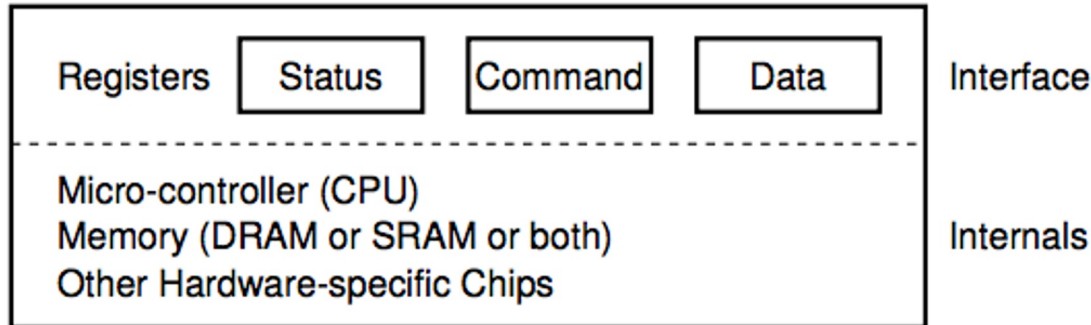
- A bus design trades off cost, complexity, performance, and the physical connection method
- USB devices are slow, but you can unplug them while the machine is running. And they are cheap.
- A memory or graphics bus is the opposite, it is expensive, complicated, and you can't unplug things...

# Abstraction

- An I/O device presents some kind of abstraction to the system
- Internally it may be very complex with lots of firmware and chipsets. (RAID controllers, disk controllers, etc).

# A Canonical Device

- Consider a generic device
- It presents three registers as its interface to the system: a read-only **status** register, and two writable registers: **command** and **data**



# Example Interaction

```
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 Interaction



```
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
```


Poll the device until it isn't busy

# Example Interaction

```
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
```


Put data into the register (such as data to write to a disk block)

# Example Interaction

```
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
```

## Tell the device to write

# Example Interaction

```
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
```

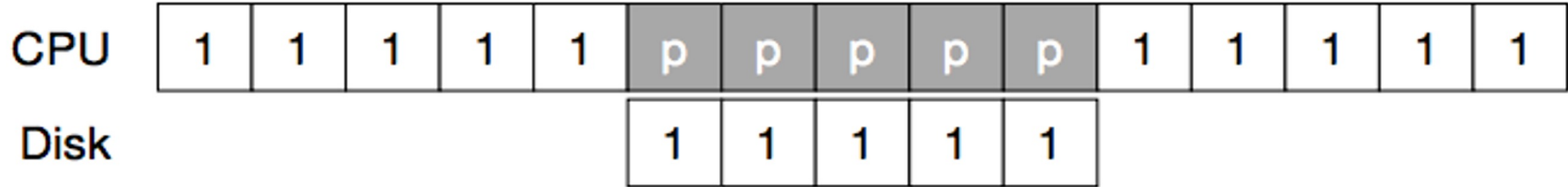
## Wait until it is done

# Programmed IO (PIO)

- This is called Programmed Input Output (PIO)
- Simple, but inefficient
- Polling is a form of busy-waiting. This chews up processor cycles (and probably bus bandwidth)

## Example: Programmed IO (PIO)

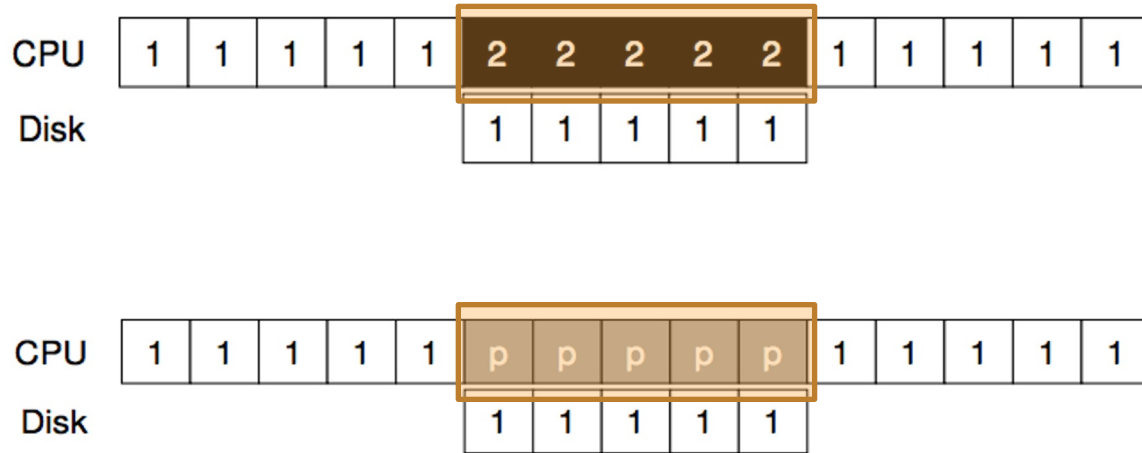
- Process 1 has to poll while doing I/O



# Interrupts

- Interrupts: The OS tells the I/O device to do something and can go off and do something else
- The process can put itself to sleep if it is blocking I/O
- The hardware device then interrupts the processor, which invokes the OS
- The code that runs is called an Interrupt Service Routine

# Interrupts allow us to overlap computation with I/O





# How often to interrupt

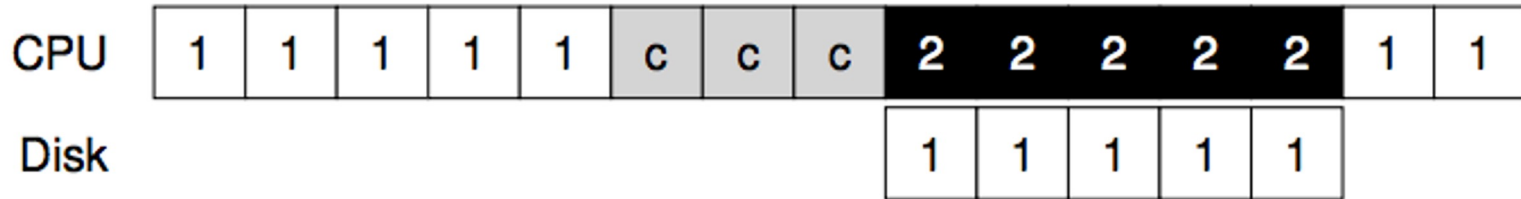
- Lots of interrupts may take a lot of time to handle
- I/O devices may buffer a bunch of data and then interrupt (called **coalescing**)

# Moving Data Efficiently

Unfortunately, there is one other aspect of our canonical protocol that requires our attention. In particular, when using programmed I/O (PIO) to transfer a large chunk of data to a device

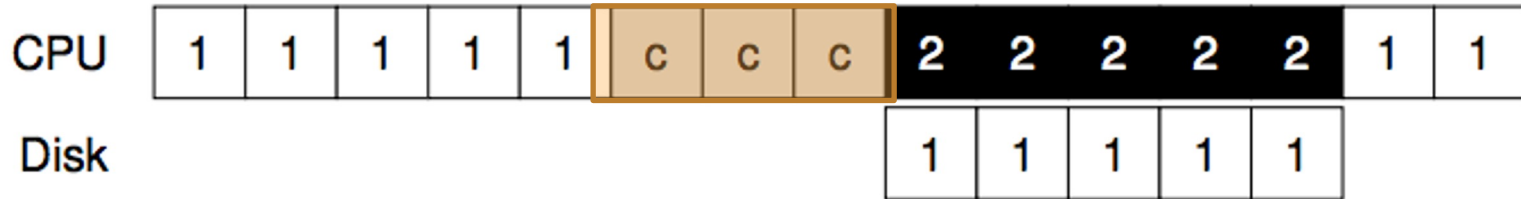
# Moving Data Efficiently

the CPU is once again overburdened with a rather trivial task, and thus wastes a lot of time and effort that could better be spent running other processes. This timeline illustrates the problem:



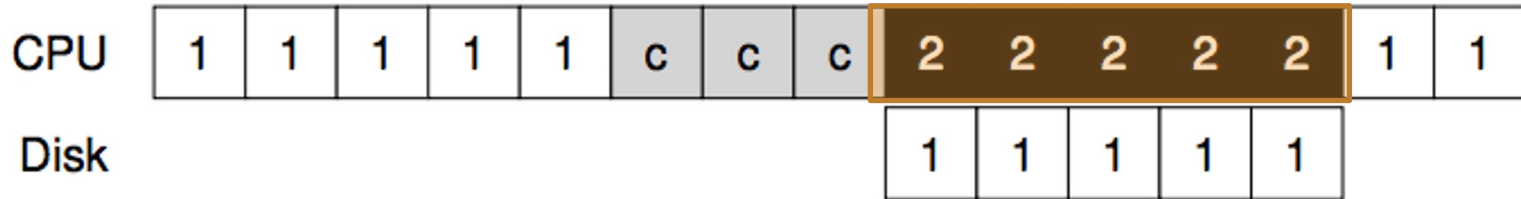
# Moving Data Efficiently

In the timeline, Process 1 is running and then wishes to write some data to the disk. It then initiates the I/O, which must copy the data from memory to the device explicitly, one word at a time (marked c in the diagram).



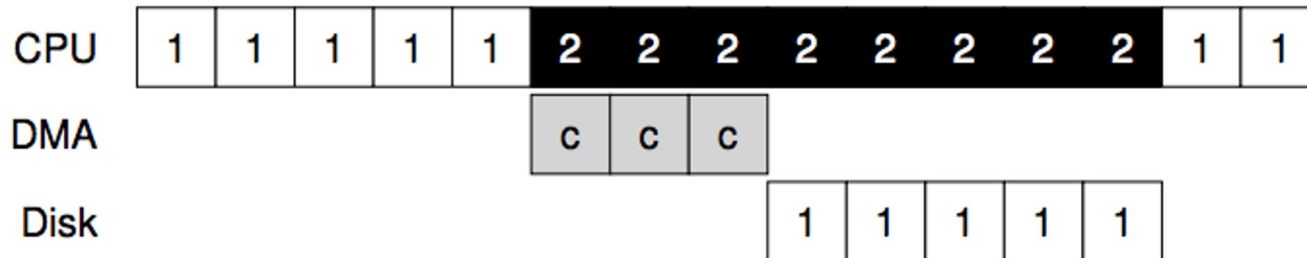
# Moving Data Efficiently

When the copy is complete, the I/O begins on the disk and the CPU can finally be used for something else.



# Direct Memory Access

- DMA is special hardware that moves data from memory, over the bus, to the I/O device
- The OS just tells the DMA where the data is and how big it is
- When done the DMA unit raises an interrupt



# Accessing the Device

- Sending commands to the device is a different issue...
- There are two methods:
  - Port-mapped I/O
  - Memory mapped I/O

# Port Mapped I/O

- Also called Isolated I/O
- Explicit instructions are used to communicate with the device
- On x86: in and out
- in and out write to specific addresses that are distinct from the physical memory addresses of RAM



## Question?

**in** and **out** can write to devices, such as disks. These instructions should or should not require kernel mode?

(A) Should

(B) Should not

# Memory Mapped I/O

- Load and Store to addresses just like physical memory
- But the bus knows these addresses are actually going to a particular device that isn't memory
- Takes up part of your physical address space (not a big deal with 64 bit machines!)

# Access vs Transfer

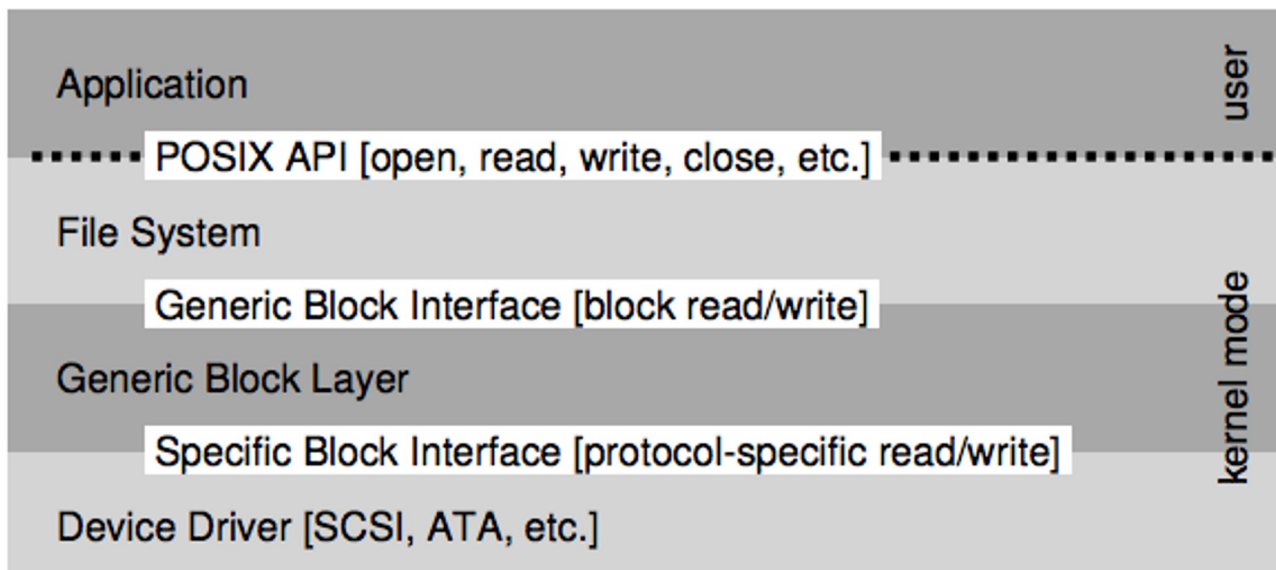
- It is tricky to see, but the **access method** (port mapped, memory mapped) is separate from the **movement of data** (PIO vs DMA)
- Awesome Stack Overflow post for more: [Difference between Memory Mapped I/O and Programmed I/O](#)

# Where does this fit in?

- Consider a file system like NTFS, XFS, FAT, etc.
- We want our file system to work over lots of I/O devices: IDE drives, RAID arrays, USB thumb drives, SSD, etc.
- To solve these problems we use: **abstraction**
- In the case of file systems there are several layers

# Block vs Character

- Block devices include disk drives
  - Commands include read, write, seek
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character devices include keyboards, mice, serial ports
  - Commands include get, put
  - Libraries layered on top allow line editing



Devices drivers are a huge % of kernel code.  
70% of Linux is device drivers!

<https://github.com/torvalds/linux/blob/5a81e6a171cdbc1fa8bc1fdd80c23d3d71816fac/CREDITS#L739>

## Example: IDE (Integrated Drive Electronics)

- This example is taken from XV6 (a teaching OS from MIT)
- <https://github.com/ecros/xv6-vm/blob/master/ide.c>
- Port-mapped I/O, Programmed I/O (non DMA)

### Control Register:

Address 0x3F6 = 0x08 (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

The IDE disk presents a simple interface to the system consisting of 4 types of registers.

## IDE Interface



### Control Register:

Address 0x3F6 = 0x08 (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

The IDE disk presents a simple interface to the system consisting of 4 types of registers.

#### 1. Control Register

## IDE Interface

### Control Register:

Address 0x3F6 = 0x08 (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

The IDE disk presents a simple interface to the system consisting of 4 types of registers.

1. Control Register
2. Command Block Registers

## IDE Interface

### Control Register:

Address 0x3F6 = 0x08 (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

The IDE disk presents a simple interface to the system consisting of 4 types of registers.

1. Control Register
2. Command Block Registers
3. Status Register

## IDE Interface

### Control Register:

Address 0x3F6 = 0x08 (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

The IDE disk presents a simple interface to the system consisting of 4 types of registers.

1. Control Register
2. Command Block Registers
3. Status Register
4. Error Register

## IDE Interface

### Control Register:

Address 0x3F6 = 0x08 (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

The IDE disk presents a simple interface to the system consisting of 4 types of registers.

1. Control Register
2. Command Block Registers
3. Status Register
4. Error Register

These registers are available by reading/writing to specific I/O addresses using special instructions such as **in** and **out** on the x86.

## IDE Interface



```

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

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)
    }
}

```

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

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

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)
    }
}
```



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

static void ide_start_request(struct buf *b) {
    ide_wait_ready();
    outb(0x1f6, 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)
    }
}
```

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

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)
    }
}
```

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

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)
    }
}
```

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

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)
    }
}
```

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

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)
    }
}
```



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

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)
    }
}
```

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

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)
    }
}
```

# Notes on Example

```
static inline void
outsl(int port, const void *addr, int cnt)
{
    asm volatile("cld; rep outsl" :
                  "=S" (addr), "=c" (cnt) :
                  "d" (port), "0" (addr), "1" (cnt) :
                  "cc");
}
```

- outsl in xv6 is actually a macro that does “rep outsl”. So it repeats (one per byte) an outsl



*The End*