Interrupt and Device Driver

Youjip Won

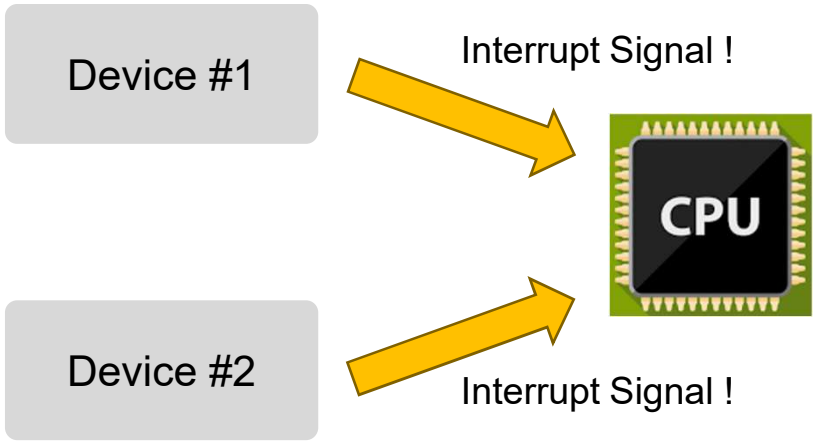
KAIST

Contents

- Interrupts
- LAPIC & IO APIC
- Drivers
- Code: Disk driver

Hardware Interrupt

- Interrupt: to tell the kernel about the hardware event
  - A device generates a signal to indicate that it needs attention from the OS.
  - Kernel handles all interrupts because in most cases, only the kernel has the required privilege and state.



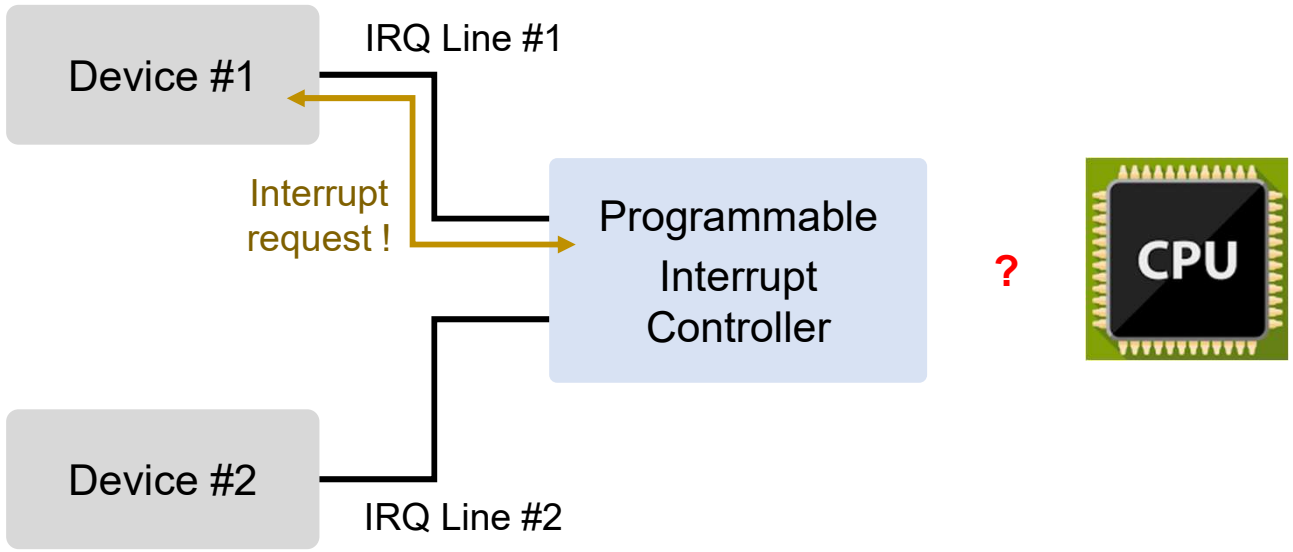
Delivering interrupt

- Interrupt Request (IRQ) Line
  - Hardware's output line that transfer the Interrupt Request.
  - **Interrupt request** is the hardware signal sent to the processor that stops a running program and allows a interrupt handler.
  - It is connected to programmable interrupt controller (PIC) not the processor.

- Programmable Interrupt Controller
  - Integrated circuit that helps a processor handles interrupt requests.

IRQ (Interrupt ReQuest) Line

- A single output line to raise an interrupt
- All IRQ lines are connected to the input pins of a hardware circuit called the ***Programmable Interrupt Controller***



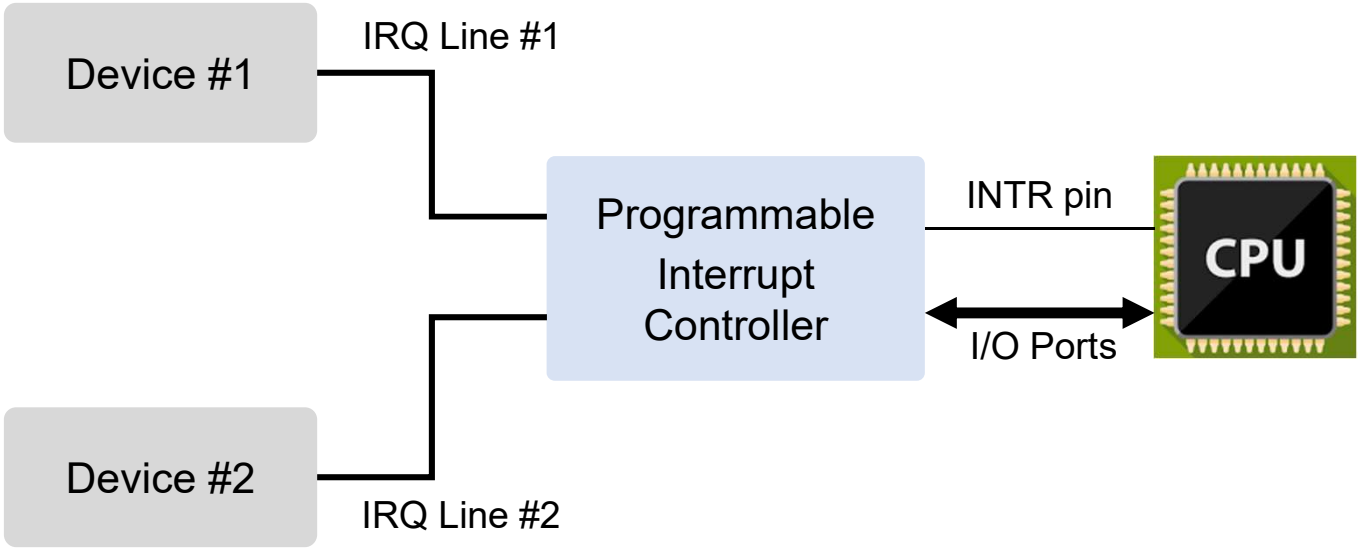
IRQ (Interrupt ReQuest) Line (Cont.)

- Each IRQ has the number and specific usage.
  - 0: Timer interrupt
  - 1: Keyboard controller interrupt

IRQ	Usage
0	system timer (cannot be changed)
1	keyboard controller (cannot be changed)
2	cascaded signals from IRQs 8–15
3	second RS-232 serial port (COM2: in Windows)
4	first RS-232 serial port (COM1: in Windows)
5	parallel port 2 and 3 or sound card
6	floppy disk controller
7	first parallel port
8	real-time clock
9	open interrupt
10	open interrupt
11	open interrupt
12	PS/2 mouse
13	math coprocessor
14	primary ATA channel
15	secondary ATA channel

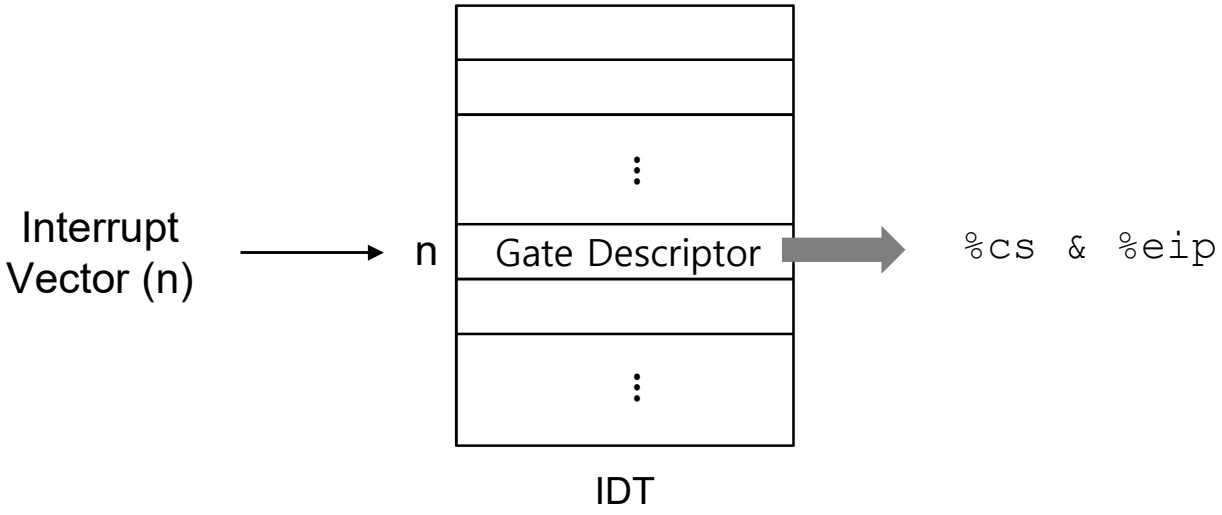
Programmable Interrupt Controller

- It is an IC that helps a processor handles interrupt requests.
- Interrupt requests can be coming from multiple different resource which may occurs simultaneously.



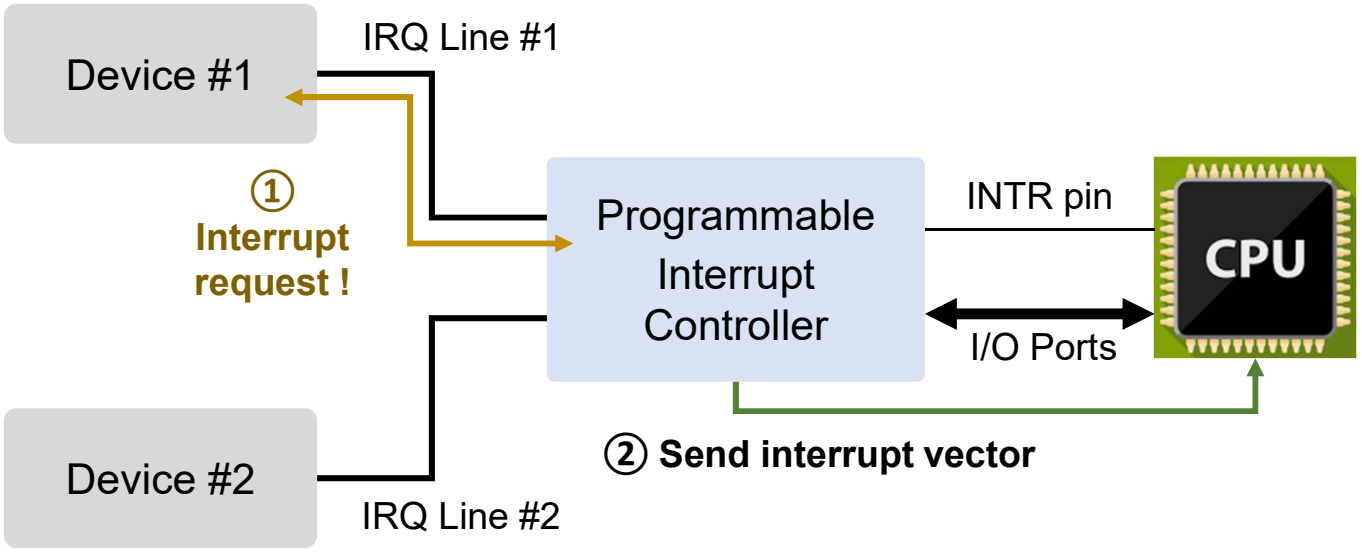
Interrupt Vector

- Index of a gate descriptor in interrupt descriptor table.
- The gate descriptor contains the information about interrupt handler.



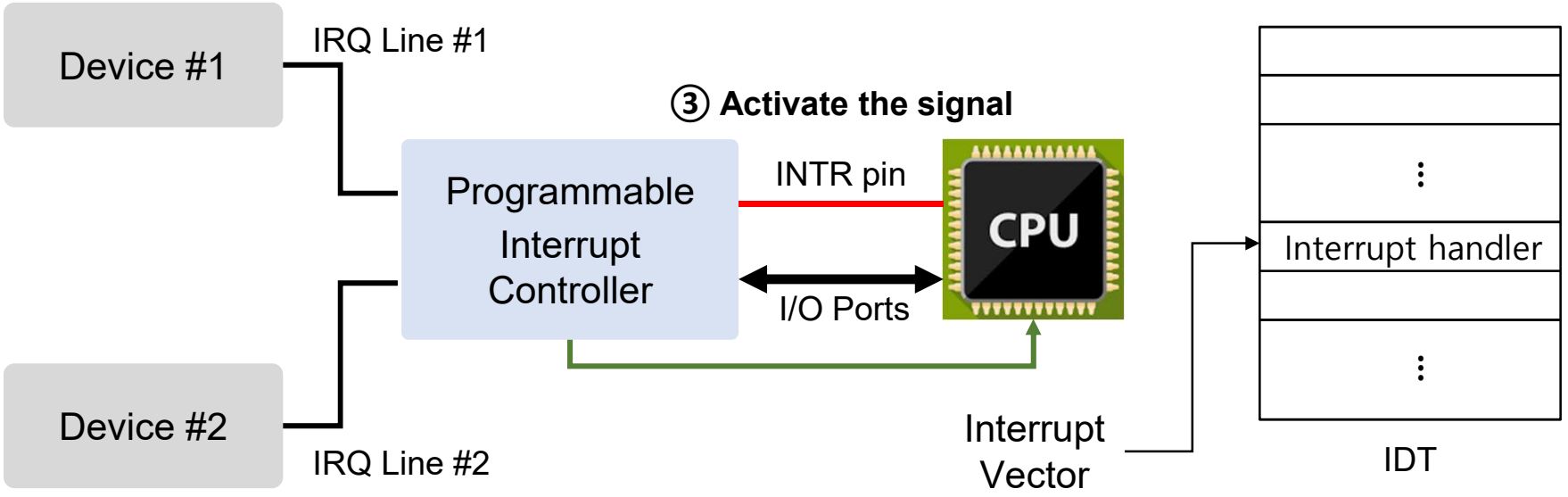
Converting IRQ into interrupt vector

- Once interrupt request is sent,
  - PIC converts the interrupt request into a corresponding interrupt vector.
  - In xv6, IRQ (n) is converted into interrupt vector (n+32).
  - PIC sends the interrupt vector to the processor via I/O ports.



`INTR` pin

- If interrupt vector is ready,
  - PIC sends the signal to processor's `INTR` pin.
  - Processor reads interrupt vector and begins to execute corresponding interrupt handler in IDT.



Programmable Interrupt Controller (Cont.)

1. Monitors the IRQ lines and keep checking the interrupt request. If two or more IRQs are received, selects the one having the lower number.
2. Once interrupt request is detected,
   1) Convert the interrupt request into a corresponding interrupt vector.
   2) Store the interrupt vector in PIC I/O port, thus allowing the CPU to read it via the data bus.
   3) Send a signal to the processor's `INTR` pin (that is, issues an interrupt).
   4) Wait until the CPU acknowledges the interrupt signal by writing into one of the *PIC* I/O ports; When this occurs, clear the INTR line.
3. Goes back to step ①.

Advanced Programmable Interrupt Controller

- Programmable Interrupt Controller (PIC): old mother boards
  - PIC from archaic Intel 8259 processor.

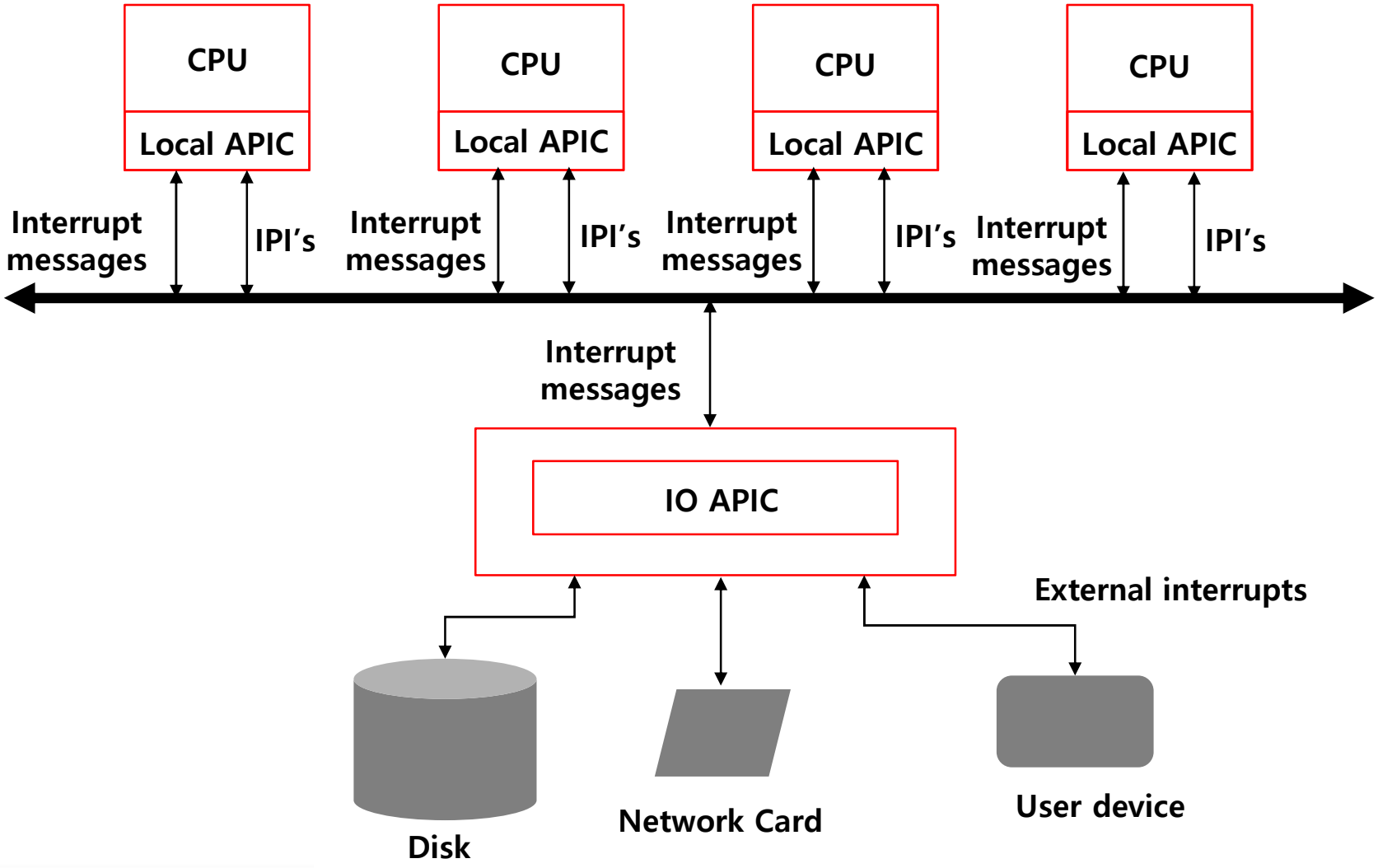
- Advanced Programmable Interrupt Controller (APIC)
  - APIC from 82489DX(80486 and early Pentium)
  - Split architecture of Local APIC and IO APIC
    - IO APIC (in IO device) + local APIC (attached to CPU):

IOAPIC and LAPIC

- LAPIC (Local APIC)
  - It is integrated into the CPU itself.
  - It handles all external interrupts.

- IO APIC
  - It is integrated into system bus.
  - It contains a redirection table to route the interrupt it receives from peripheral buses to one or more local APICs.
  - CPU can program the entries in the table through memory mapped IO (MMIO).

IOAPIC and LAPIC



Setting IOAPIC

- Operating system can modify the mapping between the IRQ line and interrupt vector by using memory mapped I/O
- Xv6 establishes so that IRQ (n) is mapped in interrupt vector (n+32).

- Memory Mapped I/O
  - A method of assigning peripherals with unique addresses.
  - The peripheral has the register map than maintains its registers.
  - Each register has an index.

Structure of IO APIC MMIO

- IO APIC provides the structure for memory mapped I/O.
- `reg`: Index of register.
- `data` : OS can access the value of register via this variable.

```
27 // IO APIC MMIO structure: write reg, then read or write data.
28 struct ioapic {
29   uint reg;
30   uint pad[3];
31   uint data;
32 };
```

Programming IO APIC

- After writing reg value, OS can read/write the data from/to register through data variable.

```
34 static uint
35 ioapicread(int reg)
36 {
37   ioapic->reg = reg;
38   return ioapic->data;
39 }
40
41 static void
42 ioapicwrite(int reg, uint data)
43 {
44   ioapic->reg = reg;
45   ioapic->data = data;
46 }
```

Read the value from register (ID == reg)

Write the value to register (ID == reg)

Initializing IO APIC: main.c

- The IO APIC is initialized when the system is booting up.
  - The main function for kernel calls the function ioapicinit().
  - The ioapicinit() initializes the mapping between IRQ and interrupt vector.

```
17 int
18 main(void)
19 {
                                    …
26   ioapicinit();    // another interrupt controller
                                    …
35   kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
36   userinit();      // first user process
37   mpmain();        // finish this processor's setup
38 }
```

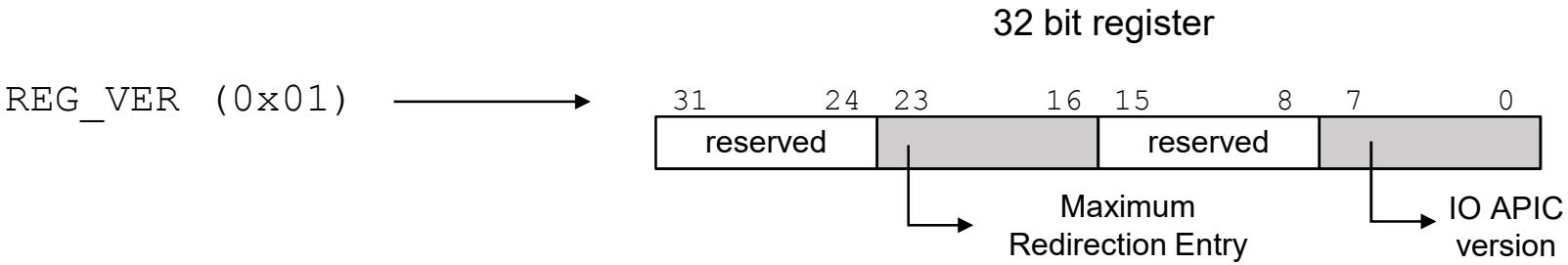
Initializing IO APIC: ioapic.c

- Checking the status of IO APIC
  - Read the ID of IO APIC and compare it with the processor's configuration.
  - Read the maximum entry number of redirection table (Line 54).

```
9 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
                                        …
48 void
49 ioapicinit(void)
50 {
51   int i, id, maxintr;
52
53   ioapic = (volatile struct ioapic*)IOAPIC; // IO device to memory
54   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
55   id = ioapicread(REG_ID) >> 24;
56   if(id != ioapicid)
57     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
                                        …
65 }
```

Read maximum entry number: `ioapic.c`

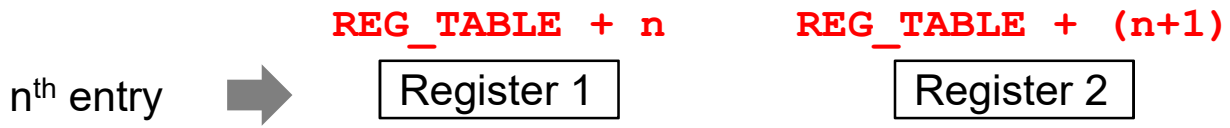
- Read the maximum entry number of redirection table.




```
9 #define IOAPIC  0xFEC00000   // Default physical address of IO APIC
                                        …
48 void
49 ioapicinit(void)
50 {
51   int i, id, maxintr;
52
53   ioapic = (volatile struct ioapic*)IOAPIC; // IO device to memory
54   maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
55   id = ioapicread(REG_ID) >> 24;
56   if(id != ioapicid)
57     cprintf("ioapicinit: id isn't equal to ioapicid; not a MP\n");
                                        …
65 }
```

Setting a redirection entry

- Each entry has two register.
  - Register 1: The configuration for IRQ .
    - (i) Disabled or not / (ii) Interrupt vector.
  - Register 2: Target CPU that the signal will be sent.
- The index of the first redirection entry is `REG_TABLE (== 0x10)`.
- The indexes of n^{th} entry registers are
  - `REG_TABLE + n`
  - `REG_TABLE + n + 1`



Setting a redirection entry: `ioapic.c`

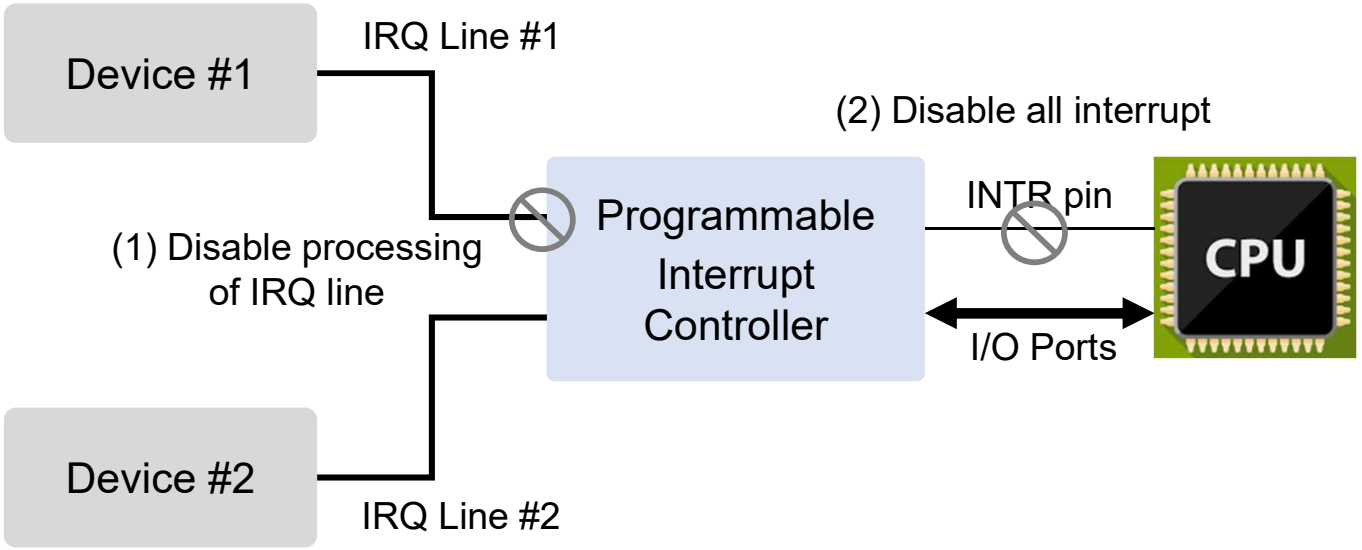
- Register 1 (Line 62)
  - IRQ is disabled and mapped to interrupt vector (32 + n)
- Register 2 (Line 63)
  - Set the target CPU that the signal will be sent.

```
48 void ioapicinit(void) {
                                        …
58
59   // Mark all interrupts edge-triggered, active high, disabled,
60   // and not routed to any CPUs.
61   for(i = 0; i <= maxintr; i++){
62     ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
63     ioapicwrite(REG_TABLE+2*i+1, 0);
64   }
65 }
```

T_IRQ0 == 32

Disabling the interrupt

- (1) Disable processing of interrupt from a specific IRQ line.
  - Set the disable flag (0x1000) at n^{th} IRQ line by using memory mapped I/O.
  - Disabled interrupts are not lost; PIC sends them to the CPU as soon as IRQ lines are enabled again.
- (2) Disable all interrupt.
  - `cli` instruction: Clear the IF flag to the eflags register. ↔ `sti` instruction
  - Ignore all interrupt signal from INTR pin.



Enabling IRQ line: `ioapic.c`

- Enable the IRQ line and set the target CPU.
- If operating system is ready to handle interrupt, calls this function to enable interrupt.
  - (i) Disk driver is ready or (ii) Console is ready.

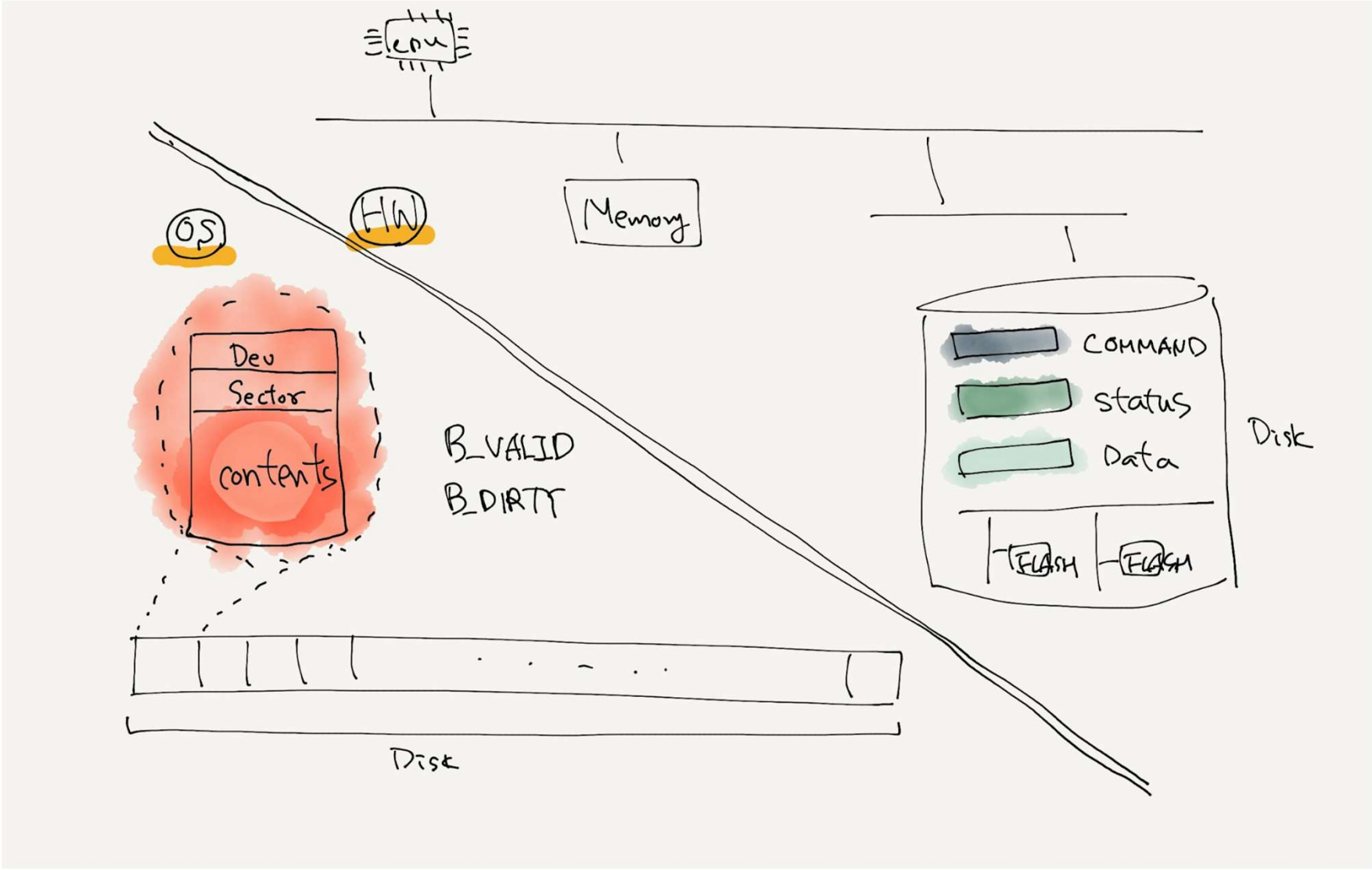
```
67 void
68 ioapicenable(int irq, int cpunum)
69 {
70   // Mark interrupt edge-triggered, active high,
71   // enabled, and routed to the given cpunum,
72   // which happens to be that cpu's APIC ID.
73   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
74   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
75 }
```

Drivers

- The code in an OS that manages a device.
  - Tells the device to do something.
  - Configure the device.
  - Handle the interrupt from the device.

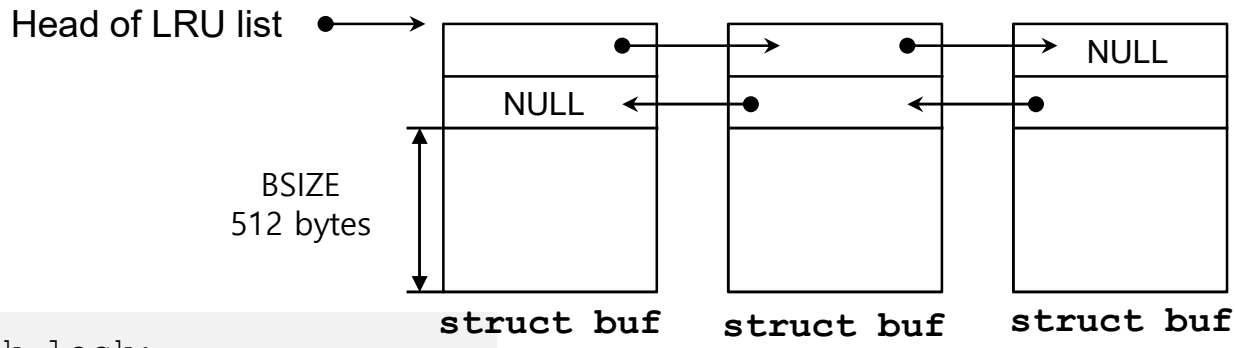
- Device driver for disk
  - Copies data to and from the disk.
  - Unit of transfer: 512 byte (sector)
  - Host side data structure for sector: `struct buf`

Disk Driver


CPU
Memory
OS
HW
Dev
Sector
contents
B_VALID
B_DIRTY
COMMAND
status
Data
Disk
FLASH
FLASH
Disk

struct buf

- It has two roles; (i) buffer cache & (ii) hold disk command.
- Buffer cache
  - All buffer caches are managed by doubly linked list.
  - It contains the data and block number as an ID.

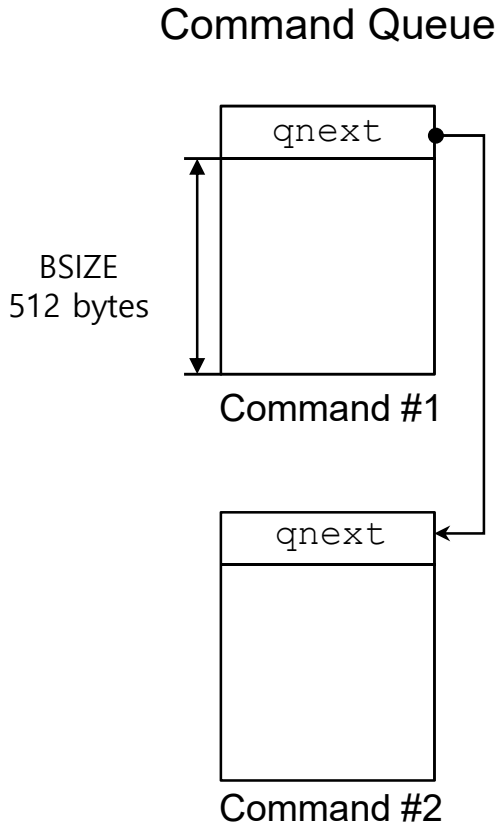



```
 1 struct buf {
 2   int flags;
 3   uint dev;
 4   uint blockno;
 5   struct sleeplock lock;
 6   uint refcnt;
 7   struct buf *prev; // LRU cache list
 8   struct buf *next;
 9   struct buf *qnext; // disk queue
10   uchar data[BSIZE];
11 };
```

struct buf (Cont.)

- Hold disk command
  - `data` is the area that contains the data to be written or empty for reading data.
  - The pending commands are maintained by queue.

```
 1 struct buf {
 2   int flags;
 3   uint dev;
 4   uint blockno;
 5   struct sleeplock lock;
 6   uint refcnt;
 7   struct buf *prev; // LRU cache list
 8   struct buf *next;
 9   struct buf *qnext; // disk queue
10   uchar data[BSIZE];
11 };
```



Flag of `struct buf`

- `B_VALID`
  - This flag is set when data read is done.
  - Before sending the read command, OS has to clear this flag.
- `B_DIRTY`
  - This flag is set when data is updated.
  - Before sending the write command, OS has to set this flag.

```
 1 struct buf {
 2   int flags;
 3   uint dev;
 4   uint blockno;
 5   struct sleeplock lock;
 6   uint refcnt;
 7   struct buf *prev; // LRU cache list
 8   struct buf *next;
 9   struct buf *qnext; // disk queue
10   uchar data[BSIZE];
11 };
```

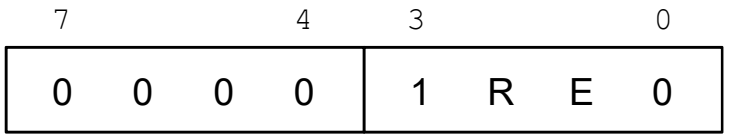
IDE Device Driver

- The layer for controlling the IDE interface device.
- Xv6 uses IDE device drive to control the disk.

- There are the registers that are used to communicate with disk.
  - Control register (8 bit)
  - Command block registers
    - Data register (16 bit)
    - 7 registers (7 * 8 bit)

Control Register

- Address `0x3F6`
- `0000 1RE0` (1 Byte)
  - The bits with the numerical value (0 or 1) must be set to the same value.
  - Control register contains only two valid bits.
  - `R` is the software reset bit, which causes a drive reset when being set to 1.
  - `E` bit 1 is the interrupt enable flag. (Active when set to 0).



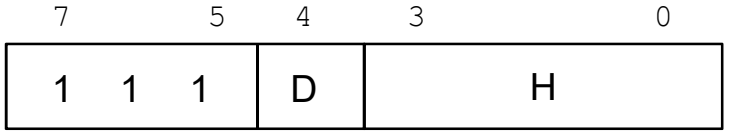
Command Block Registers

- Address `0x1F0` = Data Port (2 Byte)
  - Reading or Writing the data via this port.

- Address `0x1F1` = Error (1 Byte)
- Address `0x1F2` = Sector Count (1 Byte)
- Address `0x1F3` = LBA low byte (1 Byte)
- Address `0x1F4` = LBA mid byte (1 Byte)
- Address `0x1F5` = LBA hi byte (1 Byte)
- Address `0x1F6` = SDH register (1 Byte)
- Address `0x1F7` = Command / Status (1 Byte)

SDH register

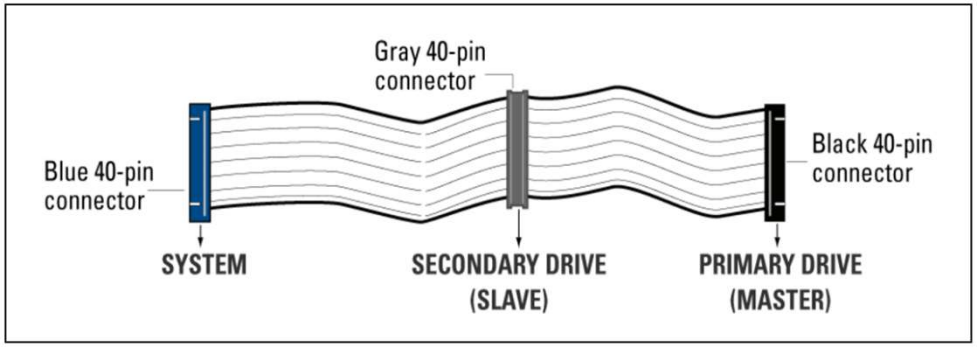
- Address `0x1F6` = SDH register (1 byte)
  - "**S**ector size", "**D**rive" and "**H**ead" Register.



  - Sector size is unchangeable by the user. It must be fixed.
  - The first 3 bits have to be set to 1.

SDH register (Cont.)

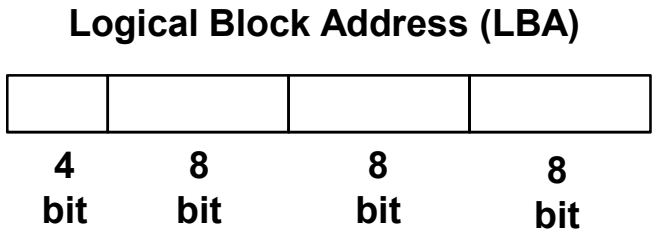
- Drive ID (1 bit)
  - This bit distinguishes between the two connected drives when using the master-slave chain (Master = 0 / Slave = 1).
  - OS can change the controlled disk by changing the ID to SDH register.



- LBA Top 4 bit (4 bit)
  - The most significant four bit for LBA.

LBA registers

- Address `0x1F2` = Sector Count (1 Byte)
  - How many sector you read or write.
- Address `0x1F3` = LBA low byte (1 Byte)
- Address `0x1F4` = LBA mid byte (1 Byte)
- Address `0x1F5` = LBA hi byte (1 Byte)
- Address `0x1F6` = SDH □ LBA top (4 bit)



LBA is 28 bit.

IDE interface can cover up to 16 GB, 512 Byte * 2^28.

Command / Status Register

- Address `0x1F7` = Command / Status (1 Byte)
  - Command register when OS writes the value. READ or WRITE
  - Status register when OS reads the value.
- Status Register (8 bit)
  - Each bit represents different status.

7	Drive busy	Cannot access registers due to internal operations.
6	Drive ready	Drive is ready to accept a command.
5	Drive write fault	Write is faulted.
4	Drive seek complete	When actuator of the drive's head is on track.
3	Data request	Drive is ready for a data transfer.
2	Corrected data flag	Set when there was a correctable data error.
1	Index	This bit is active once per disk revolution.
0	Error flag	This bit is set whenever an error occurs.

Error Register

- Address `0x1F1` = Error (1 Byte)
  - The register value is valid only when the status's ERROR bit is 1.
  - Each bit represents different error.

7	Bad block
6	Uncorrectable data error
5	Unused
4	Sector ID not found (Wrong sector number.)
3	Unused
2	Command Aborted
1	Track 0 has not been found when recalibrating
0	Unused

Initializing device driver: `main.c`

- The device driver is initialized when the system is booting up.
  - The `ideinit()` enables the interrupt for IDE interface and waits until the disk is ready.

```
17 int
18 main(void)
19 {
                                    …
26   ideinit();       // disk
                                    …
35   kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
36   userinit();      // first user process
37   mpmain();        // finish this processor's setup
38 }
```

Setting redirection entry for IDE IRQ: `ide.c`

- IRQ is enabled and mapped to interrupt vector (32 + n)
- The interrupt is routed to the highest numbed CPU.

```
50 void
51 ideinit(void)
52 {
53   int i;
54
55   initlock(&idelock, "ide");
56   ioapicenable(IRQ_IDE, ncpu - 1);
                    …
70 }
```

```
67 void
68 ioapicenable(int irq, int cpunum)
69 {
                    …
73   ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
74   ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
75 }
```

Wait for process of disk-0: `ide.c`

- Master disk (ID = 0) → disk-0 / Slave disk (ID = 1) → disk-1.
- Before enabling the disk-1, xv6 checks the status of disk-0 and waits until the disk-0 is ready.

```
 38 static int idewait(int checkerr){
                                        …
 43   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
;
                                        …
 47   return 0;
 48 }
```

```
50 void
51 ideinit(void)
52 {
53   int i;
54
55   initlock(&idelock, "ide");
56   ioapicenable(IRQ_IDE, ncpu - 1);
57   idewait(0);
                          …
70 }
```

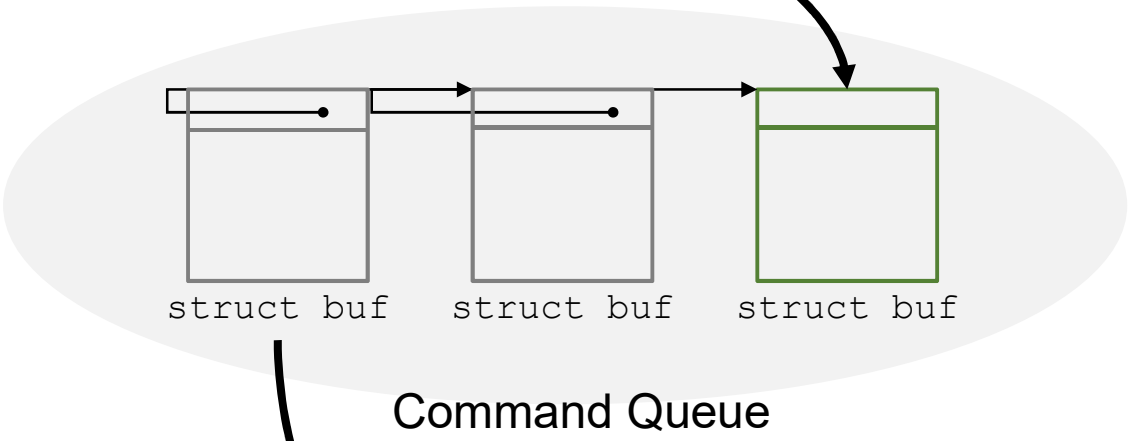
Check the disk-1: `ide.c`

- Then, switch the controlled disk to disk-1 (Line 60).
- Xv6 checks if disk-1 is present. If it becomes ready in the 1,000 iterations, set the havedisk1 to 1 (Line 61~66).
- Then, switch back to disk-0 (Line 69).

```
50 void ideinit(void) {
                                        …
59   // Check if disk 1 is present
60   outb(0x1f6, 0xe0 | (1<<4));
61   for(i=0; i<1000; i++){
62     if(inb(0x1f7) != 0){
63       havedisk1 = 1;
64       break;
65     }
66   }
67
68   // Switch back to disk 0.
69   outb(0x1f6, 0xe0 | (0<<4));
70 }
```

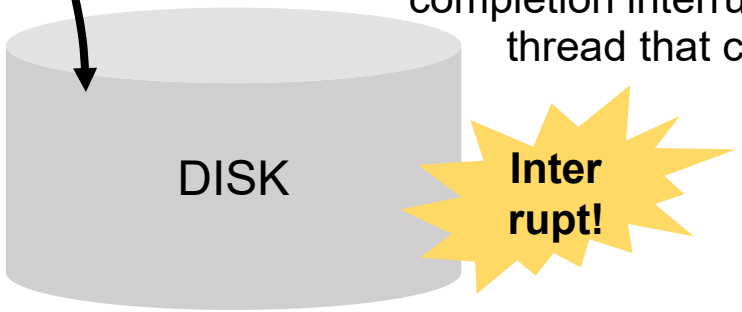
IDE device driver

① **`iderw()`:** Enqueue the command and waits for completion.



② **`idestart()`:** Transfer the command to disk.

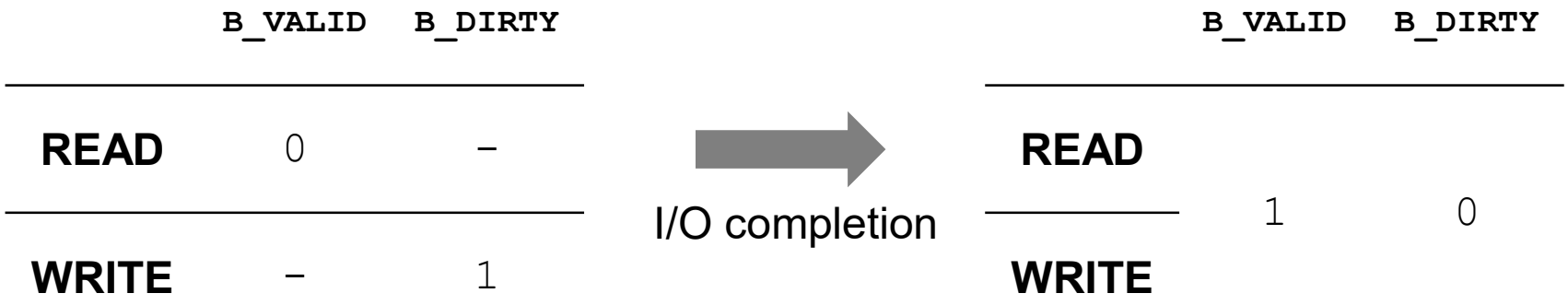
③ **`ideintr()`:** Handle I/O completion interrupt and wake up the thread that calls `iderw()`.



Flag of disk command

- The flag have to be set appropriately according to the type of I/O.
  - When reading the data, the valid bit have to be set to 0.
  - When writing the data, the dirty bit have to be set to 1.

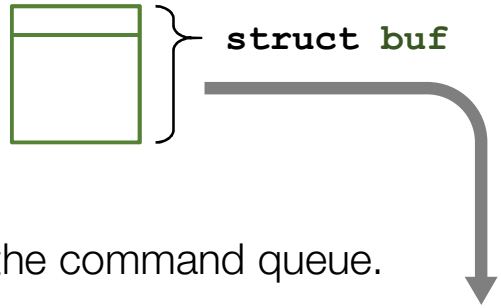
- The flag is used to detect I/O completion.
  - The interrupt handler for I/O completion sets the valid bit and clears the dirty bit.



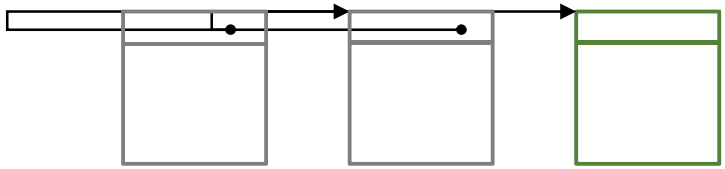
iderw()

When reading or writing the data, xv6 creates the command and calls the `iderw()`.

Where, What?
Read vs. Write



1. Enqueue the command to the command queue.



Command Queue

1. If the enqueued command is the only entry in the queue, call `idestart()`.
2. Wait for I/O completion.

Checking disk command: `iderw()`

(i) In case of write: `B_DIRTY` should be set.

(ii) In case of read: `B_VALID` should be not set.

(iii) The device should be ready.

```
138 void iderw(struct buf *b)
139 {
140   struct buf **pp;
141
142   if(!holdingsleep(&b->lock))
143     panic("iderw: buf not locked");
144   if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
145     panic("iderw: nothing to do");
146   if(b->dev != 0 && !havedisk1)
147     panic("iderw: ide disk 1 not present");
                                  …
```

Enqueueing the command: `iderw()`

1. Acquire the lock to prevent race condition for command queue.
2. Enqueue the command at the end of command queue.

```
138 void iderw(struct buf *b){
                                        …
149 1 acquire(&idelock);  //DOC:acquire-lock
150
151   // Append b to idequeue.
152   b->qnext = 0;
153 2 for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC:insert-queue
154     ;
155   *pp = b;
156
157   // Start disk if necessary.
158   if(idequeue == b)
159     idestart(b);
                                        …
```

Transfer the command : `iderw()`

3. If the command is the only entry in the queue, call `idestart()` to transfer the command to disk.

   If not, interrupt handler transfers this command to disk (more detail later).

```
138 void iderw(struct buf *b){
                                        …
149   acquire(&idelock);  //DOC:acquire-lock
150
151   // Append b to idequeue.
152   b->qnext = 0;
153   for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC:insert-queue
154     ;
155   *pp = b;
156
157   // Start disk if necessary.
158 3 if(idequeue == b)
159     idestart(b);
                                        …
```

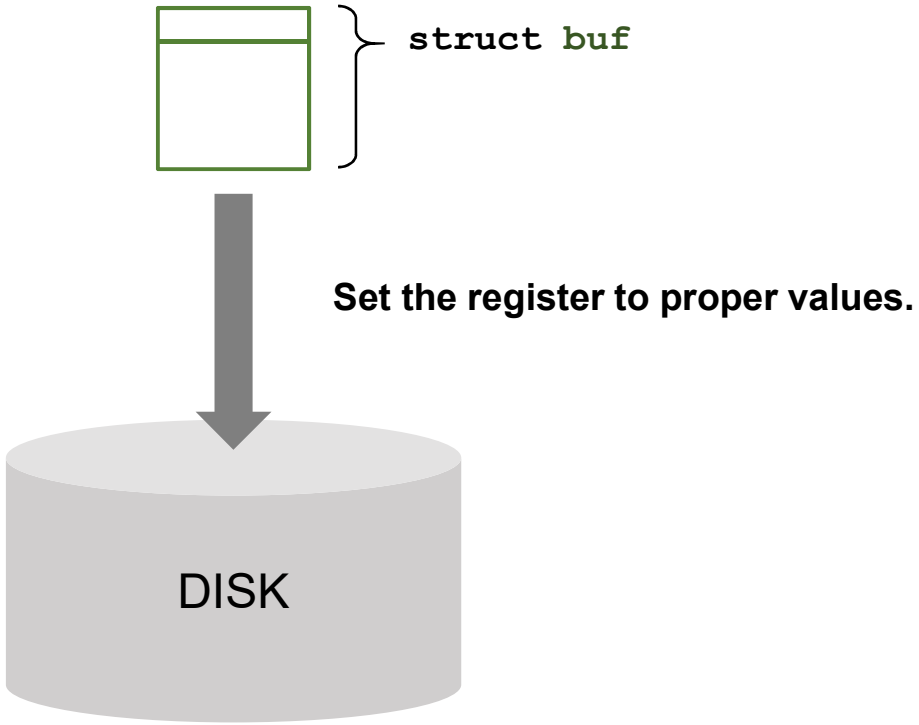
Waiting for I/O completion : `iderw()`

4. Wait for the completion of the I/O.
   1) Check the valid bit and dirty bit and goes to sleep.
   2) Wait until **the valid bit is set to 1 and dirty bit is set to 0.**

```
138 void iderw(struct buf *b){
                                        …
161   // Wait for request to finish.
162   while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
163 4   sleep(b, &idelock);
164   }
165
166
167   release(&idelock);
168 }
```

idestart()

- Transfer the command to the disk.
- Write the proper value to control block registers of disk.



Waiting until disk is ready: `idestart()`

1. Wait until the disk is ready.

   This is for waiting the end of internal operation before updating the register.

```
73 static void idestart(struct buf *b) {
                    …
87   idewait(0);
```

```
 38 static int idewait(int checkerr) {
                                    …
 43   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) !=
IDE_DRDY)
 44     ;
 45   if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
 46     return -1;
 47   return 0;
 48 }
```

Enabling interrupt / Setting sector count: `idestart()`

2. Enable interrupt for the device

   ➔ Clears the E bit at the Control Register (`0x3F6`).

3. Set the number of sectors for an I/O

   ➔ Write the sector count to the sector count register (`0x1F2`).

```
73 static void idestart(struct buf *b) {
                                    …
87   idewait(0);
88   outb(0x3f6, 0); // generate interrupt
89   outb(0x1f2, 1); // number of sectors
                                    …
```

Setting LBA / Setting drive ID: `idestart()`

4. Set the 28 bit logical block address

   ➔ Write the LBA to the four registers (`0x1F3 ~ 0x1F6`).

5. Set the drive ID.

   - Write the drive ID at the upper 4 bit to the SDH register (`0x1F6`).
   - First 3 bits should be 1 so do bitwise AND 0xe0 and drive ID << 4.

```
73 static void idestart(struct buf *b) {
                                    …
87   idewait(0);
88   outb(0x3f6, 0);  // generate interrupt
89   outb(0x1f2, 1);  // number of sectors
90   outb(0x1f3, sector & 0xff); // 1
91   outb(0x1f4, (sector >> 8) & 0xff); // 2
92   outb(0x1f5, (sector >> 16) & 0xff); // 3
93   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f)); // 4
                                    …
```

Transferring the write command: `idestart()`

6. If the DIRTY bit is 1, transferring the write command.
   - Write the write command to command register (`0x1F7`).
   - Call `outsl(port, address, cnt)`.

```
73 static void idestart(struct buf *b) {
                                    …
90   outb(0x1f3, sector & 0xff);
91   outb(0x1f4, (sector >> 8) & 0xff);
92   outb(0x1f5, (sector >> 16) & 0xff);
93   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
94   if(b->flags & B_DIRTY){
95     outb(0x1f7, write_cmd);            // WRITE
96     outsl(0x1f0, b->data, BSIZE/4);    // transfer data
97   } else {
                                    …
```

outsl(port, address, cnt)

- asm
  - Inline assembly code for C.
  - With extended asm, you can read and write C variables when executing assembly code.
  - Each string means the operation of this asm.
- volatile
  - Prevent the optimization by compiler.

```
33 static inline void
34 outsl(int port, const void *addr, int cnt)
35 {
36   asm volatile("cld; rep outsl" :
37                "=S" (addr), "=c" (cnt) :
38                "d" (port), "0" (addr), "1" (cnt) :
39                "cc");
40 }
```

Instructions: `outsl(port, address, cnt)`

- There are three instructions (Line 36).
- `(i) cld` instruction
  - It clears the direction flag, so that the following instruction increments `%esi` or `%edi`.
  - When the flag is set, following instruction decrements the register instead.
- `(ii) rep` instruction
  - It executes the following instruction `%ecx` times, decrementing `%ecx` after each iteration.

```
33 static inline void
34 outsl(int port, const void *addr, int cnt)
35 {
36   asm volatile("cld; rep outsl" :
37                "=S" (addr), "=c" (cnt) :
38                "d" (port), "0" (addr), "1" (cnt) :
39                "cc");
40 }
```

Instructions: `outsl(port, address, cnt)` (Cont.)

- There are three instructions (Line 36).
- `(iii) outsl` instruction
  - It writes a 32-bit value to port `%dx` from the data at address `%esi` and then increments `%esi` by 4.

```
33 static inline void
34 outsl(int port, const void *addr, int cnt)
35 {
36   asm volatile("cld; rep outsl" :
37                "=S" (addr), "=c" (cnt) :
38                "d" (port), "0" (addr), "1" (cnt) :
39                "cc");
40 }
```

Output: `outsl(port, address, cnt)`

- The operations starting with "=" are for saving the value of register to variable.
  - `"=S"`: Write the value of `%esi` to the `addr` variable.
  - `"=c"`: Write the value of `%ecx` to the `cnt` variable.

```
33 static inline void
34 outsl(int port, const void *addr, int cnt)
35 {
36   asm volatile("cld; rep outsl" :
37                "=S" (addr), "=c" (cnt) :
38                "d" (port), "0" (addr), "1" (cnt) :
39                "cc");
40 }
```

Input: `outsl(port, address, cnt)`

- The next operations are for setting the register.
  - `"d"`: Write the value of `port` variable to `%dx` register.
  - `"0"` and `"1"`: Using same registers with outputs.
    - Write the value of `addr` variable to `%esi` register.
    - Write the value of `cnt` variable to `%ecx` register.

```
33 static inline void
34 outsl(int port, const void *addr, int cnt)
35 {
36   asm volatile("cld; rep outsl" :
37                "=S" (addr), "=c" (cnt) :
38                "d" (port), "0" (addr), "1" (cnt) :
39                "cc");
40 }
```

Summary: `outsl(port, address, cnt)`

- `rep outsl`
  - Repeat the following operations `%ecx` times.
  - Write 4 bytes data at `%esi` to port (`%dx`). Then, increase `%esi` by 4.

Variable	port	addr	cnt
Register	%dx	%esi	%ecx

```
33 static inline void
34 outsl(int port, const void *addr, int cnt)
35 {
36   asm volatile("cld; rep outsl" :
37                "=S" (addr), "=c" (cnt) :
38                "d" (port), "0" (addr), "1" (cnt) :
39                "cc");
40 }
```

Transferring the write command: `idestart()` (Cont.)

6. If the DIRTY bit is 1, transfer the write command.
   - Write the write command to command register (`0x1F7`).
   - Call `outsl(port, address, cnt)`.
     - Write 4 bytes 128 times. (`BSIZE/4 == 128`)

```
73 static void idestart(struct buf *b) {
                                    …
90   outb(0x1f3, sector & 0xff);
91   outb(0x1f4, (sector >> 8) & 0xff);
92   outb(0x1f5, (sector >> 16) & 0xff);
93   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
94   if(b->flags & B_DIRTY){
95     outb(0x1f7, write_cmd);              // WRITE
96     outsl(0x1f0, b->data, BSIZE/4);      // transfer data
97   } else {
                                    …
```

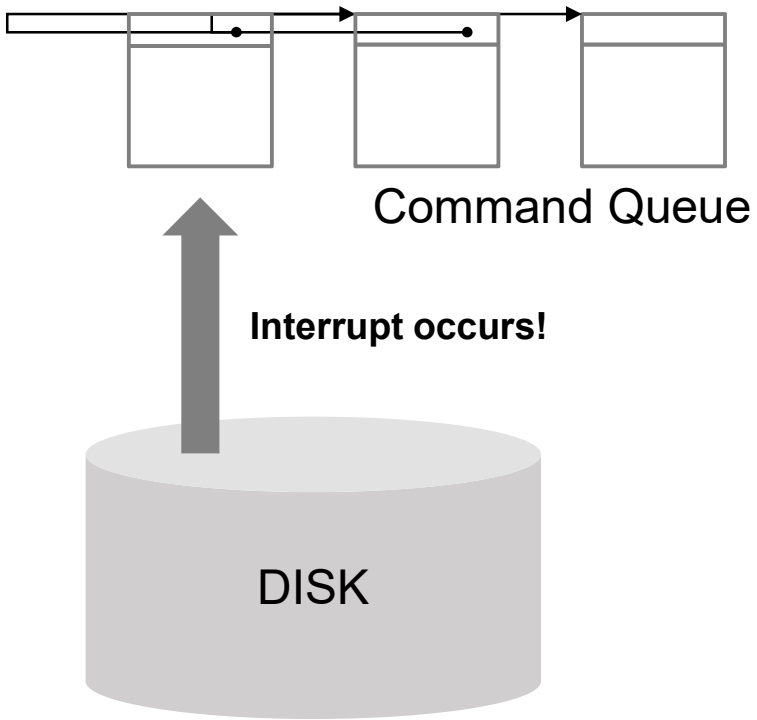
Transfer the read command: `idestart()` (Cont.)

6. If the DIRTY bit is 0, transferring the read command.
   - Write the read command to command register (`0x1F7`).

```
 73 static void idestart(struct buf *b) {
                                    …
 93   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) |
((sector>>24)&0x0f));
 94   if(b->flags & B_DIRTY){
 95     outb(0x1f7, write_cmd);            // WRITE
 96     outsl(0x1f0, b->data, BSIZE/4);    // trnasfer
 97   } else {
 98     outb(0x1f7, read_cmd);             // READ
 99   }
100 }
```

ideintr()

- Handle the I/O completion interrupt.
- Dequeue an entry at the head of queue and transfer next command.



Dequeue the command: `ideintr()`

This function is invoked once the I/O completion interrupt occurs.

1. Dequeue the command at the head of the queue.

```
103 void
104 ideintr(void)
105 {
106   struct buf *b;
107
108   // First queued buffer is the active request.
109   acquire(&idelock);
110
111   if((b = idequeue) == 0){
112     release(&idelock);
113     return;
114   }
115   idequeue = b->qnext;
```

Read the data: `ideintr()`

2. If the valid bit is 0, read the data from the disk.
   - Wait until the disk is ready.
   - Call `insl(port, addr, cnt)`.

```
103 void
104 ideintr(void)
105 {
106   struct buf *b;
107
108   // First queued buffer is the active request.
109   acquire(&idelock);
110
                                    …
116
117   // Read data if needed.
118   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
119     insl(0x1f0, b->data, BSIZE/4);
```

insl(port, address, cnt)

- `rep insl`
  - Repeat the following operations `%ecx` times.
  - Read 4 bytes from port (`%dx`) to `%edi`. Then, increase `%edi` by 4.

Variable	port	addr	cnt
Register	%dx	%edi	%ecx

```
12 static inline void
13 insl(int port, void *addr, int cnt)
14 {
15   asm volatile("cld; rep insl" :
16                "=D" (addr), "=c" (cnt) :
17                "d" (port), "0" (addr), "1" (cnt) :
18                "memory", "cc");
19 }
```

%edi

Read the data: `ideintr()`

2. If the valid bit is 0, read the data from the disk.
   - Wait until the disk is ready.
   - Call `insl(port, addr, cnt)`.
     - Read 4 byte 128 times (`BSIZE/4 == 128`).

```
103 void
104 ideintr(void)
105 {
106   struct buf *b;
107
108   // First queued buffer is the active request.
109   acquire(&idelock);
110
                                    …
116
117   // Read data if needed.
118   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
119     insl(0x1f0, b->data, BSIZE/4);
```

Waking up sleeping thread: `ideintr()`

3. Set the valid bit and clear the dirty bit.
4. Wake up the thread that calls `iderw()`.

```
103 void
104 ideintr(void)
105 {
                                        …
117   // Read data if needed.
118   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
119     insl(0x1f0, b->data, BSIZE/4);
120
121   // Wake process waiting for this buf.
122   b->flags |= B_VALID;
123   b->flags &= ~B_DIRTY;
124   wakeup(b);
```

Transferring next command: `ideintr()`

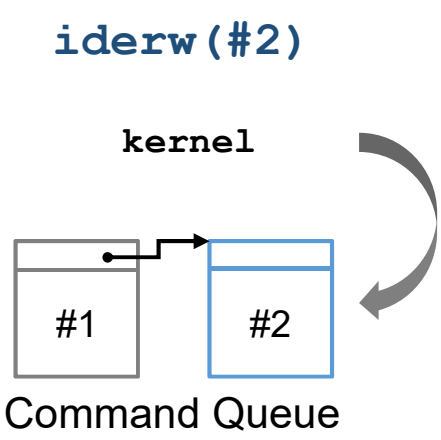
5. Transfer the next command if there is command in the queue.

   ➔ The pending commands at `iderw()` are transferred to the disk by interrupt handler.

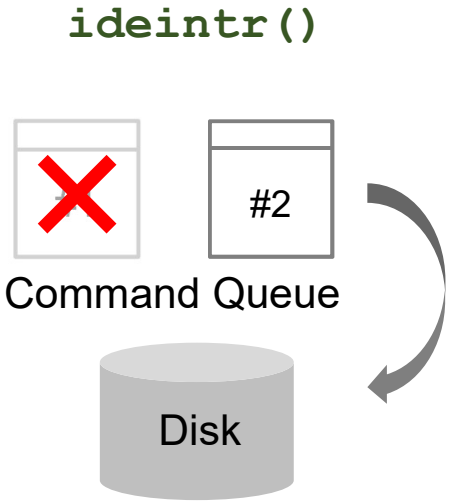
```
103 void
104 ideintr(void)
105 {
                                        …
125
126   // Start disk on next buf in queue.
127   if(idequeue != 0)
128     idestart(idequeue);
129
130   release(&idelock);
131 }
```

Handling I/O in xv6.

- Xv6 transfers the command to disk only when previous command is done.
  - (i) At `iderw()`, transferring the command is blocked if there is another entry in the command queue.
  - (ii) Interrupt handler dequeues the command at the head of queue and transfers the next command blocked by an another I/O command.



(i) The command #2 is not transferred to the disk.



(ii) The command #2 is transferred only when the command #1 is done.

Summary

- Interrupt, exception and system call
- Protection mode
- Trapframe
- System call
- Interrupt and Device driver