

CS202 (003): Operating Systems I/O

Instructor: Jocelyn Chen

Last Time

Replacement policy

FIFO

throw out the oldest

MIN (optimal)

throw away the entry that won't
be used for the longest time

LRU

throw out the least
recently used

Pretty decent!

It approximates OPT when:
principle of temporal locality
holds strongly

Implementing LRU

In OS, it doubles the memory traffic
(since after every reference, have to move some structure to the head of some list)

In hardware, it's **a lot of work** to timestamp each reference and keep the list ordered

Implementing LRU in OS/hardware is a lot of pain!

Approximating LRU

Clock Algorithm: Not Recently Used

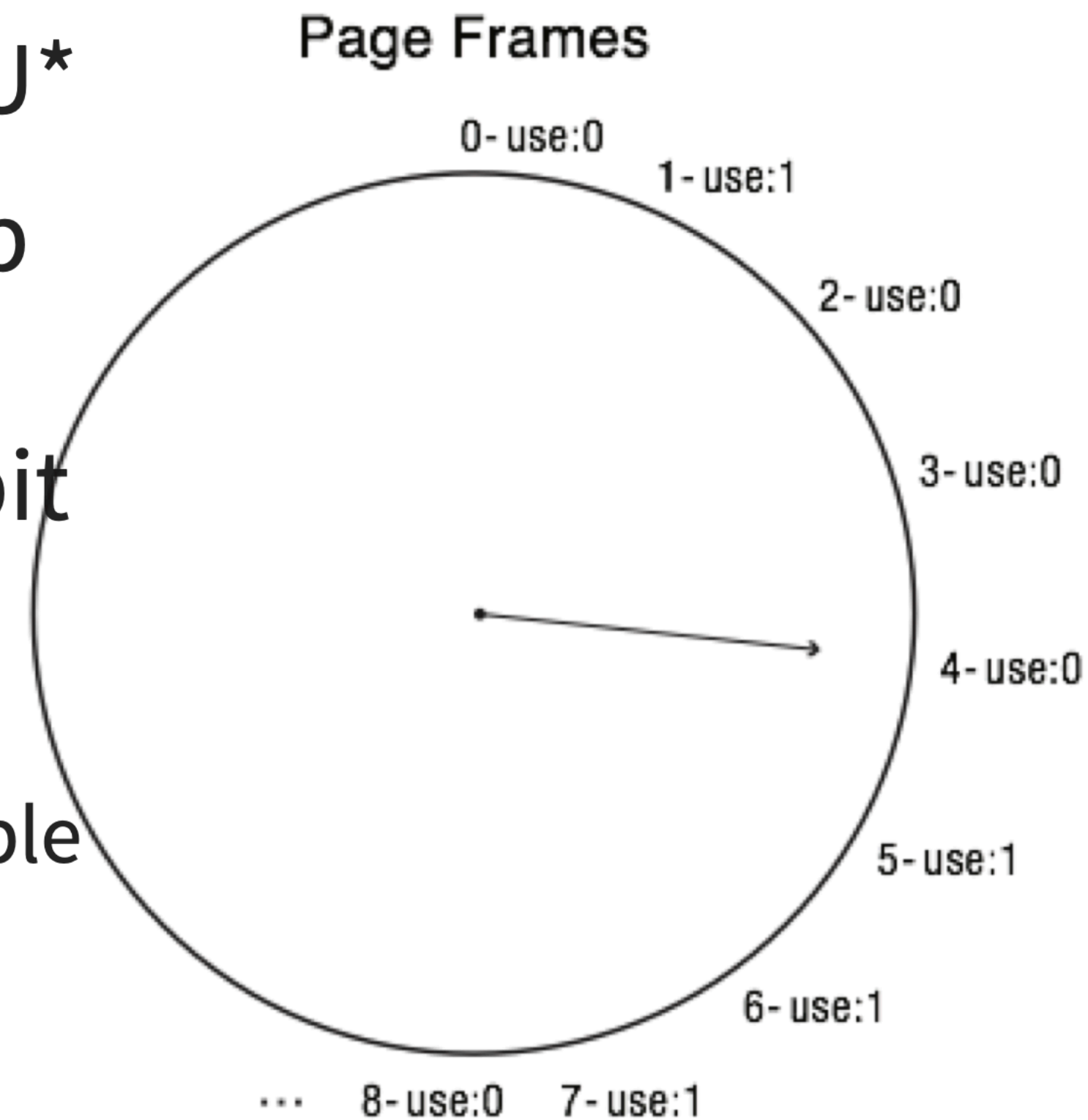
“Second-chance” algorithm

Approximating LRU*

Periodically, sweep through all pages

- Used? Clear use bit
- Unused? reclaim
 - update core map
 - invalidate page table
 - write back if dirty
 - TLB shutdown
 - add to free list

*(*yes, LRU was already an approximation...)*



Generalizing CLOCK: Nth Chance

- With each page, OS maintains a counter to indicate the number of sweeps that page has gone through.
- On page fault, OS checks accessed bit:
 - If 1, then clear it, and also clear the counter.
 - If 0, then increment the counter; if count == N, replace page.

Large N implies better approximation to LRU:

e.g., $N = 1000$ is a very good LRU approximation.

However, a large N implies more work by the OS before a page can be replaced.

Decent approximations to LRU, assuming that past is a good predictor of the future

$N = 1$ implies the default clock algorithm.

Thrashing

Process requires more memory than the system has

Each time a page is brought in, another page, whose contents will soon be referenced, is thrown out

A program touches 50 pages (each equally likely) but only have 40 physical page frames

If we have enough physical pages, 100ns/ref

If we have too few physical pages, assuming every 5th reference leads to a page fault, then:

$4 \text{ ref} * 100 \text{ ns} + 1 \text{ page fault} * 10\text{ms for disk I/O}$

This lead to 5 refs per (10ms + 400ns) $\sim 2\text{ms/ref} = \mathbf{20,000x slowdown!}$

Thrashing

Process requires more memory than the system has

Each time a page is brought in, another page, whose contents will soon be referenced, is thrown out

What we want: virtual memory the size of disk with access time the speed of physical memory

What we have: memory with access time roughly at the same magnitude as disk access

Note: this issue is not limited to page access, but we are discussing this issue in the context of page access

Thrashing - what are the causes?

What we want: virtual memory the size of disk with access time the speed of physical memory

What we have: memory with access time roughly at the same magnitude as disk access

process don't reuse memory (no temporal locality)

OR

process reuses memory but the memory that is absorbing most of the accesses doesn't fit

Each processes fit the memory individually, but too much to fit for all processes in the system!

Thrashing - What do we do?

Each processes fit the memory individually, but too much to fit for all processes in the system!

Working Set

**The pages a process has touched over
some trailing window of time**

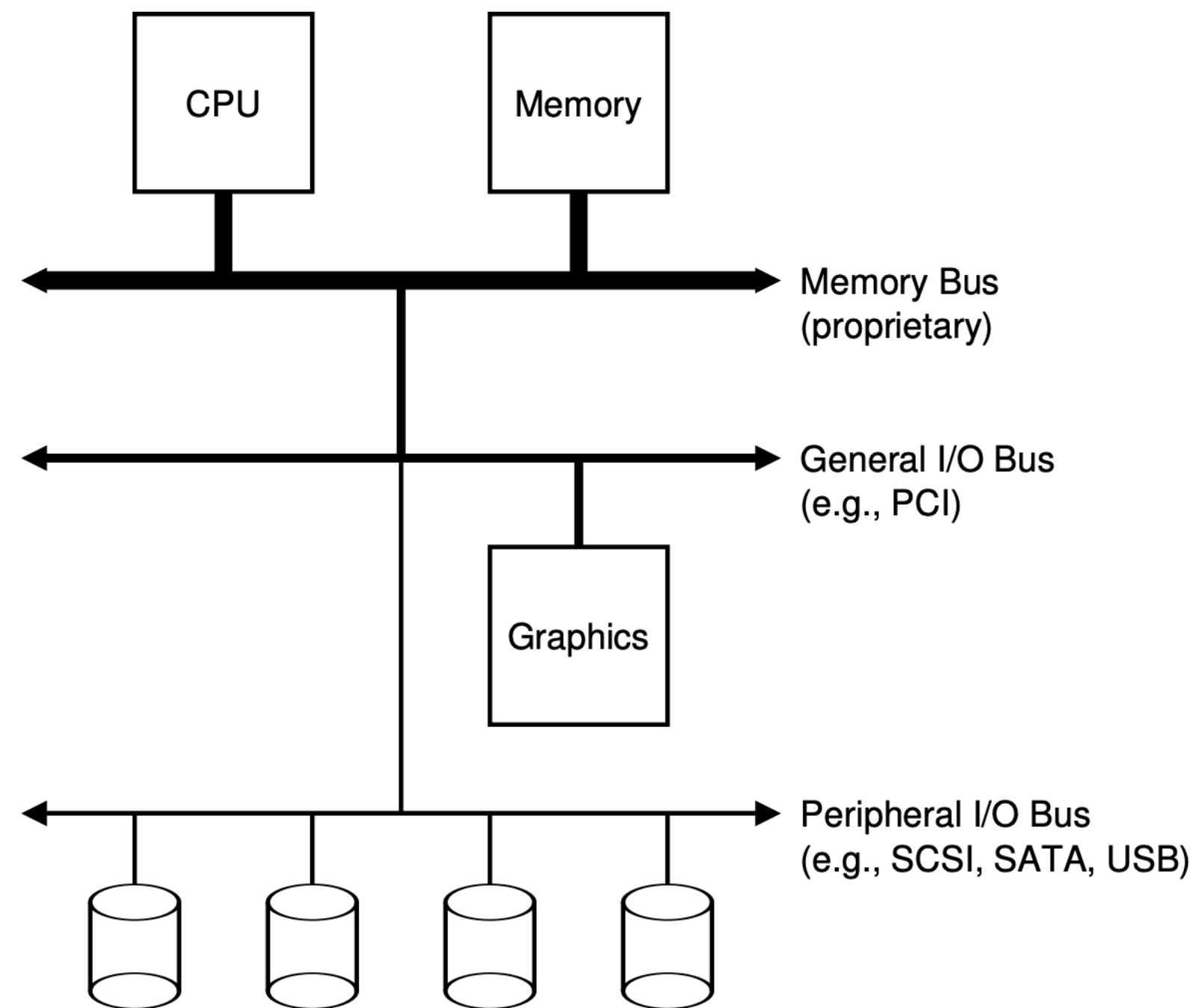
Only run a set of processes s.t. the union
of their working sets fit in memory

Page fault frequency

**Track the metric
(# page faults/instructions executed)**

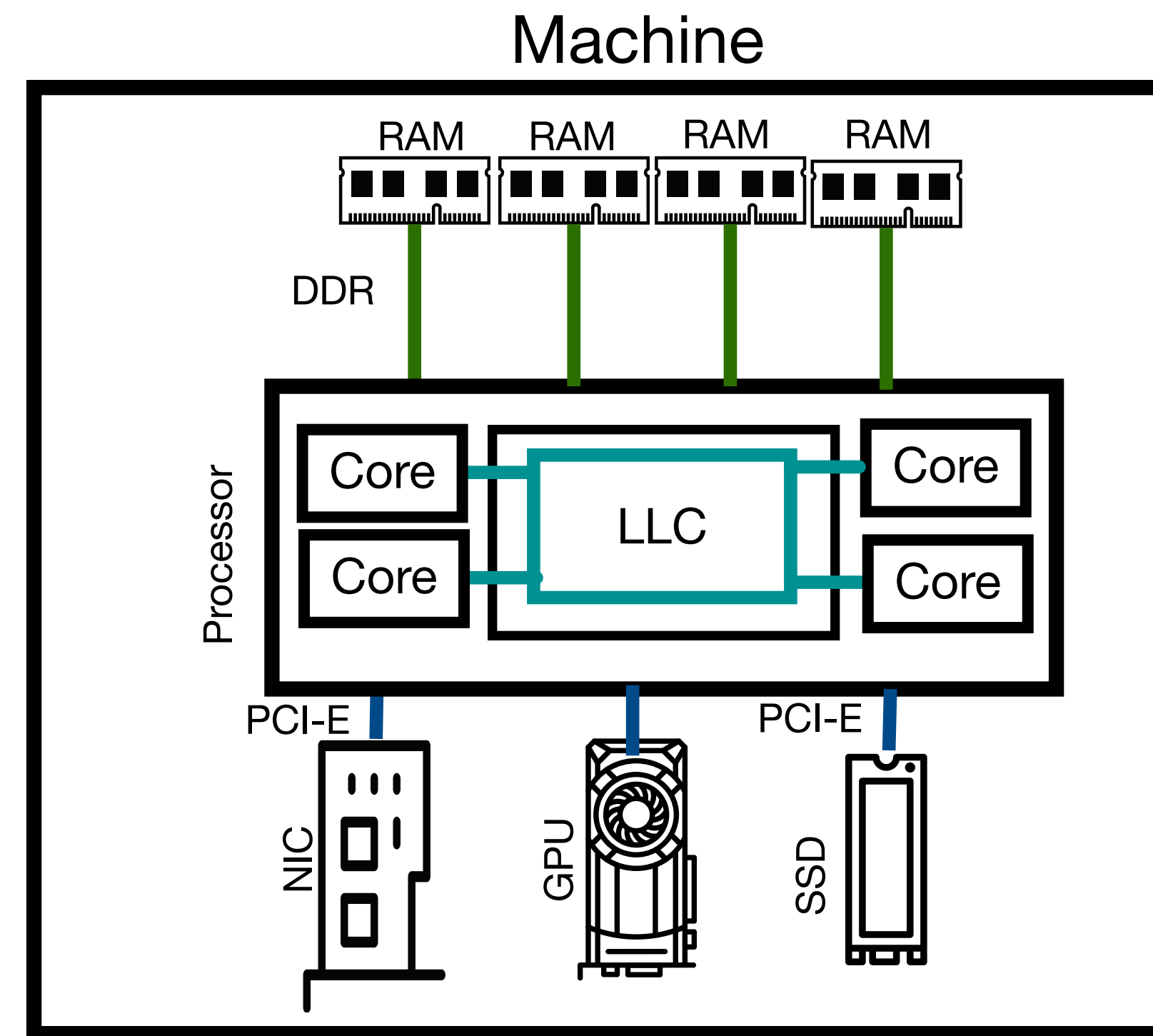
If that thing rises above a threshold, and
there is not enough memory on the
system, swap out the process

I/O Architecture at a high-level



Textbook (older)

Figure 36.1: Prototypical System Architecture



Newer

CPU_(kernel)/Device Interaction

Mechanics of Communication

Explicit I/O instructions

`outb, inb, outw, inw`

WeesyOS boot.c (handout)

Control Register:

Address 0x3F6 = 0x08 (0000 1RE0): R=reset,
E=0 means "enable interrupt"

Command Block Registers: **each register stores 8-bit data**

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 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

LBA: Linear block address (28-bit in the handout)

describes storage locations

each sector gets a unique number starting from 0

Status is read-only

Command is used to add instructions to the disk

Figure 36.5: **The IDE Interface** (Integrated Device Electronics): connection between a bus on the motherboard and disk storage

```

static int ide_wait_ready() {
    while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
        ; // loop until drive isn't busy
    // return -1 on error, or 0 otherwise
}

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() >= 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);
}

```

handle read/write request

read: data is read and valid bit is set
write: data is written and dirty bit is cleaned

Figure 36.6: The xv6 IDE Disk Driver (Simplified)

CPU_(kernel)/Device Interaction

Mechanics of Communication

Explicit I/O instructions

`outb, inb, outw, inw`

WeesyOS boot.c (handout)

`keyboard_readc()`

reading keyboard input (handout)

CPU_(kernel)/Device Interaction

Mechanics of Communication

Explicit I/O instructions

`outb, inb, outw, inw`

WeesyOS boot.c (handout)

`keyboard_readc()`

reading keyboard input (handout)

`console_show_cursor()`

setting blinking cursor (handout)

CPU_(kernel)/Device Interaction

Mechanics of Communication

Memory-mapped I/O

Most of the physical address space contains regular RAM
However, the lower memory addresses (650K-1MB) are
special - they don't refer to actual RAM

WeesyOS console printing (handout)

CPU_(kernel)/Device Interaction

Mechanics of Communication

Explicit I/O instructions

Memory-mapped I/O

Interrupts

CPU_(kernel)/Device Interaction

Mechanics of Communication

Explicit I/O instructions

Memory-mapped I/O

Interrupts

Through memory

Both CPU and the device see the same memory, so they can use shared memory to communicate. (eg. DMA)

Polling vs. Interrupt (vs. busy waiting)

Busy Waiting

```
while (!device_is_ready()) {  
    // CPU is stuck here,  
    repeatedly checking  
    // Consuming CPU cycles doing  
    nothing useful  
}
```

Simple but inefficient
CPU stuck here running at full speed

Polling vs. Interrupt (vs. busy waiting)

Busy Waiting

```
while (!device_is_ready()) {  
    // CPU is stuck here,  
    repeatedly checking  
    // Consuming CPU cycles doing  
    nothing useful  
}
```

Polling

```
void poll_device() {  
    while (1) {  
        if (device_is_ready())  
            { process_device_data(); }  
  
        // Sleep/delay for a set  
        interval  
        sleep_ms(100); // Check every  
        100ms  
    }  
}
```

System checks device status at
regular intervals
Lower CPU usage

Polling vs. Interrupt (vs. busy waiting)

Busy Waiting

```
while (!device_is_ready()) {  
    // CPU is stuck here,  
    repeatedly checking  
    // Consuming CPU cycles doing  
    nothing useful  
}
```

Interrupts

```
// CPU sets up interrupt handler  
set_interrupt_handler  
    (device_interrupt_handler);  
  
// CPU goes on to do other useful work  
do_other_work();  
  
// When device needs attention,  
it triggers interrupt  
// and the handler runs automatically  
void device_interrupt_handler() {  
    // Handle the device's needs  
    process_device_data();  
}
```

Polling

```
void poll_device() {  
    while (1) {  
        if (device_is_ready())  
            { process_device_data(); }  
  
        // Sleep/delay for a set  
        interval  
        sleep_ms(100); // Check every  
        100ms  
    }  
}
```

Most sophisticated approach
If interrupt rate is high, we can get **livelock**

Programmed I/O vs. DMA (Direct memory access)

Programmed I/O

CPU writes data directly to device, and reads data directly from device.

Programmed I/O vs. DMA (Direct memory access)

Programmed I/O

CPU writes data directly to device, and reads data directly from device.

DMA

CPU places some buffers in main memory.

Tells device where the buffers are

Then "pokes" the device by writing to register

Then device uses DMA to read or write the

The CPU can poll to see if the DMA completed (or the device can interrupt the CPU when done).

CPU don't have to constantly deal with small amount of data transfer, the device can write the contents straight into memory

Software architecture: device drivers

Device drivers act as a bridge between hardware devices and the operating system kernel

Software architecture: device drivers

Device drivers act as a bridge between hardware devices and the operating system kernel

```
reset(): Initializes or resets the device  
ioctl(): Provides device-specific controls  
read()/write(): Standard data transfer operations  
handle_interrupt(): Manages hardware interrupts
```

Software architecture: device drivers

Device drivers act as a bridge between hardware devices and the operating system kernel

`reset()`: Initializes or resets the device
`ioctl()`: Provides device-specific controls
`read()/write()`: Standard data transfer operations
`handle_interrupt()`: Manages hardware interrupts

Example interface

Advantages: don't have to worry about the specific hardware implementation

Issues:

1. Device drivers is per-OS and per-device (hard part cannot be reused)
2. Bugs in device drivers often bring down the entire machine

Synchronous vs. Asynchronous I/O

Synchronous I/O

When a process makes a system call (like `read()` or `write()`), it blocks (suspends) until the operation completes

The process enters a sleep state and cannot execute other codes

Control returns to process after operation finishes

Code is often more readable, but it is slow

Synchronous vs. Asynchronous I/O

Synchronous I/O

When a process makes a system call (like `read()` or `write()`), it blocks (suspends) until the operation completes

The process enters a sleep state and cannot execute other codes

Control returns to process after operation finishes

Async I/O

System calls return immediately, even if the operation isn't completed

Instead of blocking, the call returns a status indicating what would have happened

Check through polling or interrupt

Need to use platform-specific extensions to POSIX to do async I/O for files