Persistence: I/O and Disk Devices

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2024

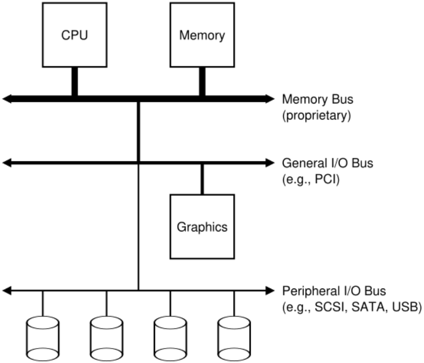
Administrivia

- Project 4 due Tue Nov 5th @ 11:59pm
- Exam 2, Thu, Nov 7th 5:45-7:15pm
  - Same format as Exam 1
  - Bring ID, #2 Pencil, and 1 sheet of notes
  - Last Name:
    - **A-K** – Van Vleck B102
    - **L-Z** – Ingraham B10
  - McBurney 5:45-8:00pm – CS 1257

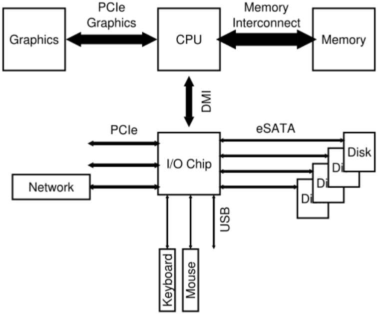
I/O Devices Agenda

- How OS interacts with I/O Devices
- How HDD is organized
- Disk Performance
- Disk Scheduling

Prototypical Systems Architecture



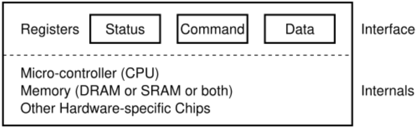
- Multiple Bus Levels
- Faster busses are shorter, more expensive



- Direct Media Interface
- Slow devices connect through an I/O chip

OS Communication with Cannonical Device

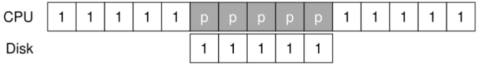
```
while (STATUS == BUSY)
  ; //wait until device is not busy
write data to DATA register
write command to COMMAND register
  (Doing so starts the device and executes the command)
while (STATUS == BUSY)
  ; //wait until device is done with request
```



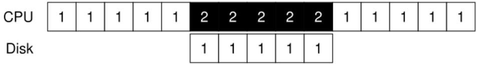
- OS uses **polling** to check status
- **Programmed I/O** (PIO) when main CPU controls data movement
- Motivates **Hardware Interrupts** for effeciency

More Efficient I/O

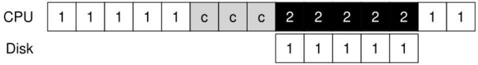
- Polling



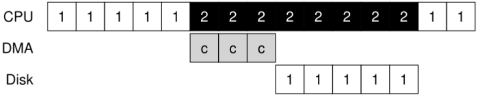
- Interrupts (allow other process to run)



- OS still copies data to device



- OS uses **Direct Memory Access (DMA)** which handles the copy portion of IO
- Just pass data location and size to DMA Controller

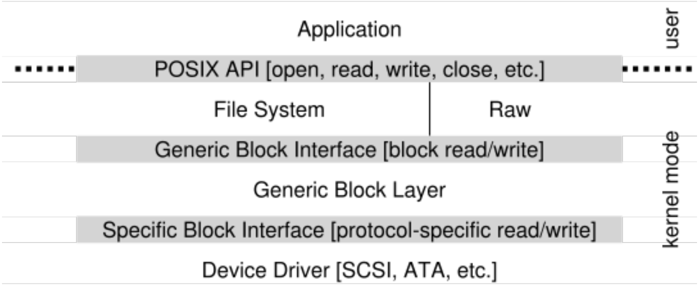


Methods of I/O Interactions

- Explicit I/O Instructions
  - on x86, the `in` and `out` instructions used to communicate with device
  - OS conrols register with data, and knows specific *port* which names the device, issues instruction.
- Memory-mapped I/O
  - Device appears as memory location
  - OS uses same load/store commands as for regular memory
  - Hardware routes the instruction to the device instead

Device Driver

- Many, many devices, each has its own protocol
- **Device driver** for each device, rest of OS just interacts with driver
- OS often has **raw interface** to directly read and write blocks
- 70% of OS code is found in device drivers



Simple IDE Disk Driver (**xv6**)

```
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);
}
```

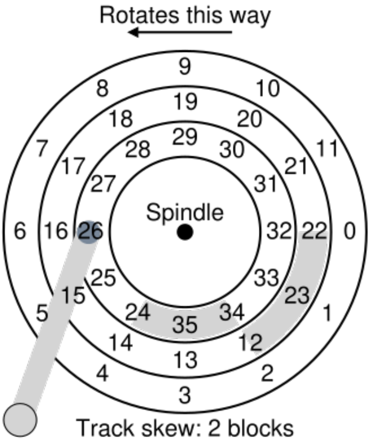
Simple IDE Disk Driver (**xv6**) (cont.)

```
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)
  }
}
```

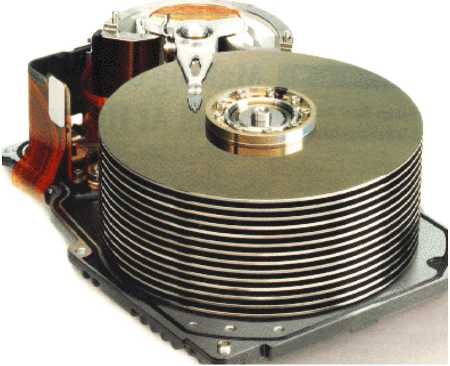
Hard Disk Interface

- Consists of sectors (512 byte blocks)
- Sectors numbered from 0 to $n-1$, **address space**
- Many file systems read/write 4KB at a time
- Sectors written along tracks
- Arm moves head as disk rotates
- Sectors have a *skew* from one track to another
- In **multi-zoned** disk, tracks in different zone have more sectors



Hard Disk Mechanics

- **Platters** has two surfaces and rotate around spindle
- Head and arm on each side of platter
- Rate of Rotation: RPM
- Time to read/write divided into three components:
  - Seek time
  - Rotation time
  - Transfer time



$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

SEEK, ROTATE, TRANSFER

Seek cost: Function of cylinder distance

- Not purely linear cost
- Must accelerate, coast, decelerate, settle
- Settling alone can take 0.5 - 2 ms

Entire seeks often takes 4 - 10 ms

Average seek = 1/3 of max seek

Depends on rotations per minute (RPM)

- 7200 RPM is common, 15000 RPM is high end

Average rotation: Half of time for 1 rotation

Pretty fast: depends on RPM and sector density.

100+ MB/s is typical for maximum transfer rate

Total time = seek + rotation + transfer time

WORKLOAD PERFORMANCE

So…

- seeks are slow
- rotations are slow
- transfers are fast

How does the kind of workload affect performance?

Sequential: access sectors in order

Random: access sectors arbitrarily

DISK SPEC

	Cheetah	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Avg Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	32 MB

Sequential read 100MB: what is throughput for each?

I/O SCHEDULERS

I/O SCHEDULERS

Given a stream of I/O requests, in what order should they be served?

Much different than CPU scheduling

Position of disk head relative to request position matters more than length of job

FCFS (FIRST-COME-FIRST-SERVE)

Assume seek+rotate = 10 ms for random request

How long (roughly) does the below workload take?Requests are given in sector numbers

300001, 700001, 300002, 700002, 300003, 700003

300001, 300002, 300003, 700001, 700002, 700003

SSTF (SHORTEST SEEK TIME FIRST)

Strategy always choose request that requires least seek time (approximate total time with seek time)

Greedy algorithm (just looks for best NEXT decision)

How to implement in OS?

Disadvantages?

SCAN

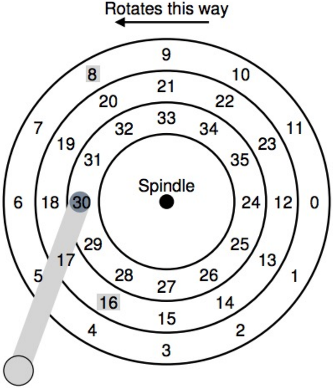
SCAN or Elevator Algorithm:

- Sweep back and forth, from one end of disk other, serving requests as pass that cylinder
- Sorts by cylinder number; ignores rotation delays

C-SCAN (circular scan): Only sweep in one direction

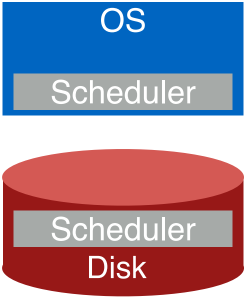
Pros/Cons?

SPTF (SHORTEST POSITIONING TIME FIRST)



SATF
(SHORTEST ACCESS TIME FIRST)

SCHEDULERS



Where should the scheduler go?

WHAT HAPPENS?

Assume 2 processes each calling read() with C-SCAN

```
void reader(int fd) {
    char buf[1024];
    int rv;
    while((rv = read(fd, buf)) != 0) {
        assert(rv);
        // takes short time, e.g., 1ms
        process(buf, rv);
    }
}
```

WORK CONSERVATION

Work conserving schedulers always try to do work if there's work to be done

Sometimes, it's better to wait instead if system anticipates another request will arrive

Possible improvements from I/O Merging

SUMMARY

Disks: Specific geometry with platters, spindle, tracks, sector

I/O Time: rotation_time + seek_time + transfer_time
Sequential throughput vs. random throughput

Scheduling approaches: SSTF, SCAN, C-SCAN
Benefits of violating work conservation

Persistence Unit:

- Intro / Disks
- File System API
- File Systems Implementation / FFS
- Journaling
- Log Structured FS
- SSDs