I/O Devices (ch. 36+37)

Operating Systems

Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

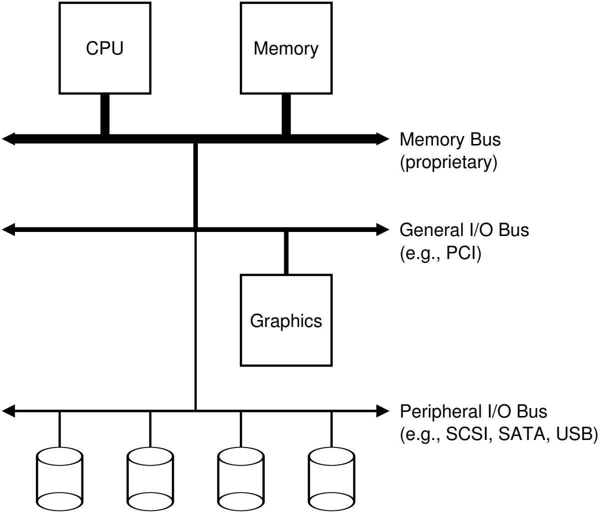
Tel-Aviv Academic College

I/O Devices

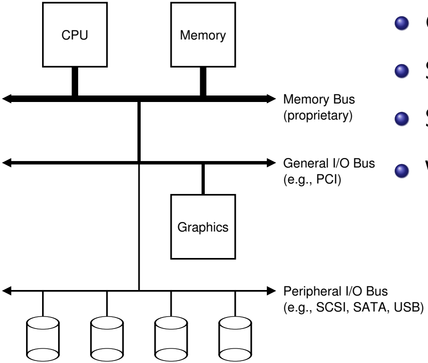
- New part: **persistence**
- But first: **input/output (I/O) devices**
  - Critical to computer systems

How should I/O be integrated into systems?

System Architecture



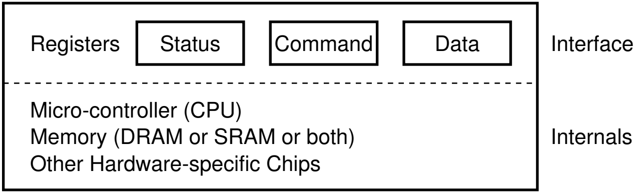
System Architecture



- CPU attached to memory via **memory bus**
- Some devices via **general I/O bus**
- Slow devices via **peripheral bus**
- Why hierarchical?
  - Physics and cost
  - Faster bus $\rightarrow$ shorter
  - Lower performance $\rightarrow$ further

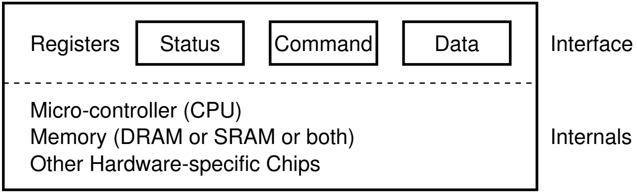
Canonical Device

- A canonical device has two important components:
  - **Hardware interface**: allows the system to control its operation
  - **Internal structure**: implementation specific to the device



Canonical Protocol

- Device interface comprised of three registers
  - **Status**: current status of device
  - **Command**: tell device to perform a task
  - **Data**: pass data to device or get data from it



- Control device behavior by reading and writing these registers

Canonical Protocol

- Typical interaction of OS with the device:

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

Canonical Protocol

1. **Polling**
   - Repeatedly reading status register
2. OS sends some data
   - Multiple writes may be needed
   - CPU involved with data movement: **programmed I/O (PIO)**
3. Write command
   - Lets device know that data is present
4. **Polling**
   - OS waits for device to finish

Canonical Protocol

- Polling is inefficient
  - Wastes CPU time waiting for device
  - Switch to another process: better utilize CPU

CPU	1	1	1	1	1	p	p	p	p	p	1	1	1	1	1
Disk						1	1	1	1	1					

How can the OS check device status without polling?

Interrupts

- Instead of pulling, issue a request
  - Put calling process to sleep
  - Context switch to another task
- Device finished: hardware interrupt
  - CPU jumps into OS **interrupt handler**
  - (Also: **interrupt service routing (ISR)**)
  - Handler will finish the request and wake waiting process

CPU	1	1	1	1	1	2	2	2	2	2	1	1	1	1	1
Disk						1	1	1	1	1					

Interrupts

- Not always the best solution
  - Device performs very quickly
  - Interrupts will slow down the system
  - Switching back and forth is expensive
- **Hybrid** (**two-phased** approach)
  - Poll for a little while
  - If not finished, use interrupts

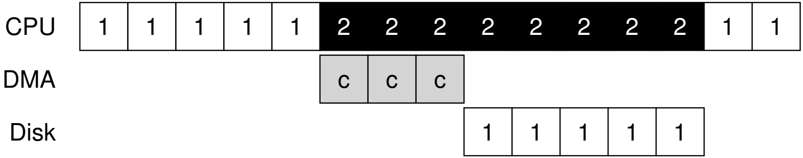
DMA

- Programmed I/O:
  - CPU transfers a large chunk of data to a device
  - CPU overburdened with a trivial task

CPU	1	1	1	1	1	c	c	c	2	2	2	2	2	1	1
Disk									1	1	1	1	1		

DMA

- Solution: **Direct Memory Access (DMA)**
  - DMA controller (a device) handles copying of data
  - OS programs DMA:
    - Where the data lives in memory
    - How much data to copy
    - Which device to send to/read from



Device Interaction

- How does the OS/CPU communicate with devices?
- **I/O instructions**
  - OS sends data to specific device registers
  - For example, `in` and `out` **privileged** instructions on x86
- **Memory-mapped I/O**
  - Device registers available as if they were memory locations
  - OS issues `load` or `store` to address
  - Hardware routes to device instead of main memory

Device Driver

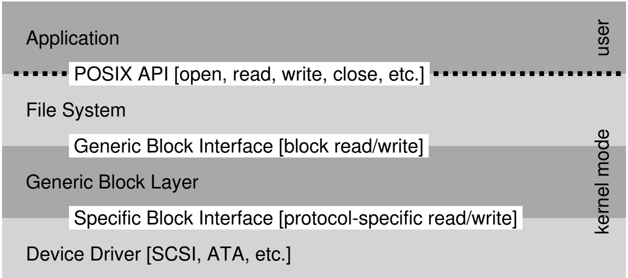
- Devices have specific interfaces
  - Keep OS as general as possible
  - e.g., build file system that works on SCSI disks, IDE disks, USB drives, etc.
- Solution?

Device Driver

- Devices have specific interfaces
  - Keep OS as general as possible
  - e.g., build file system that works on SCSI disks, IDE disks, USB drives, etc.
- Solution? **abstraction**
  - Software that knows device specifics: **device driver**
  - Over 70% of OS code in Linux
  - Primary contributor to **kernel crashes**

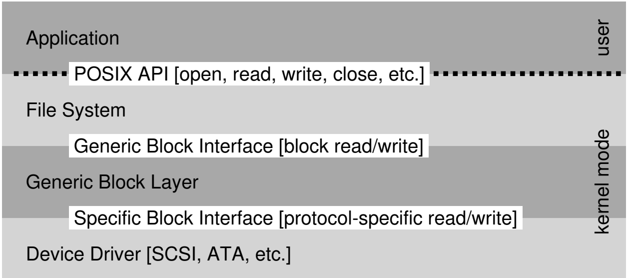
Device Driver

- The Linux file system software stack:



Device Driver

- The Linux file system software stack:



- Also available: **raw interface**
  - Enables special applications to directly read and write blocks
  - e.g., file-system checker, disk defragmentation tool

Case Study

- IDE disk
  - Four types of registers:
    - Control, command block, status, and error
  - Available at specific "I/O addresses"
    - Using `in` and `out` instructions

Case Study

- The 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 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
```

Case Study

- **Wait for device to be ready**: read Status Register (0x1F7) until READY and not BUSY
- **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 ide driver: wait

```
static int ide_wait_ready() {
    while (((int r = inb(0x1f7)) & IDE_BSY) ||
        !(r & IDE_DRDY));
}
```

xv6 ide driver: start

```
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
    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); //  WRITE
        outsl(0x1f0, b->data, 512/4); //
    } else {
        outb(0x1f7, IDE_CMD_READ); // this is a READ
            (no data)
    }
}
```

xv6 ide driver: rw

```
void ide_rw(struct buf *b) {
    acquire(&ide_lock);
    for (struct buf **pp = &ide_queue; *pp;
                                pp = &(*pp)->qnext);

    *pp = b;

    if (ide_queue == 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
    release(&ide_lock)  ;
}
```

xv6 ide driver: isr

```
void ide_intr() {
    struct buf *b;
    acquire(&ide_lock);
    if (!(b->flags & B_DIRTY) &&
        ide_wait_ready() >= 0)
            insl(0x1f0, b->data, 512/4);
    b->flags |= B_VALID;
    b->flags &= ~B_DIRTY;
    wakeup(b); // wake waiting process
    if ((ide_queue = b->qnext) != 0) //
        ide_start_request(ide_queue);
    release(&ide_lock);
}
```

Summary (I/O Devices)

- Canonical device: registers, HW interface, internal structure
- Canonical protocol: polling, data, command, polling
- Interrupts: instead of polling, issue a request
  - Device finished: hardware interrupt
  - **Hybrid** approach: poll for a little while, then use interrupts
- **DMA** controller: handles copying of data
- Device interaction: **I/O instructions** or **memory-mapped I/O**
- **Device driver**: software abstraction that knows device specifics

Hard Disk Drives

- Main form of persistent data storage
- File system technology: predicated on their behavior

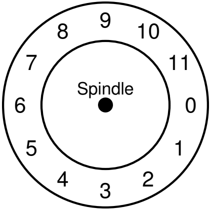
How do modern hard-disk drives store data?
What is the interface?

The Interface

- Consists of sectors (512-byte blocks)
  - Numbered 0 to $n-1$ (the drive **address space**)
  - Each can be read or written
- Multi-sector operations are possible
  - Many file systems read or write 4KB at a time
  - Only guarantee: single 512-byte block write is **atomic**
  - i.e., will completely entirely or not at all
  - **Torn write**: only portion of a larger write complete
- Common assumption: sequential access is the fastest

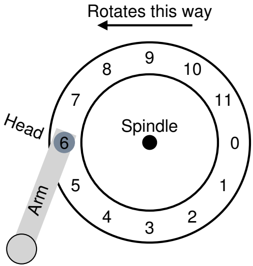
Basic Geometry

- A **platter**
  - Circular surface on which data is stored
  - Two sides, each called a **surface**
- A disk has one or more platters
  - Bound together around the **spindle**
    - Connected to a motor that spins the platters
    - Fixed rate of **rotations per minute (RPM)**

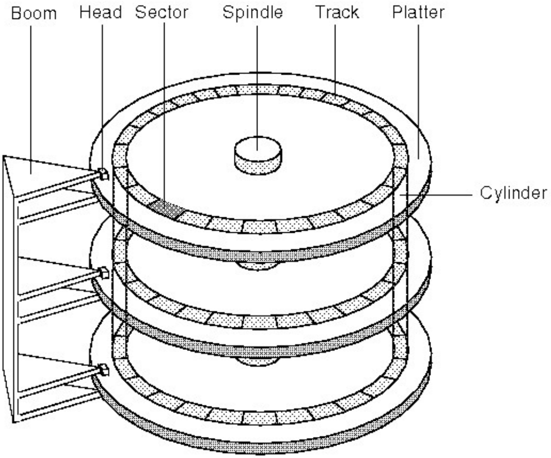


Basic Geometry

- Data is encoded in **tracks**
  - Concentric circles of sectors
  - Single surface contains thousands of tracks
- Read and write accomplished by **disk head**
  - One per surface
  - Attached to a **disk arm**
    - Moves across surface

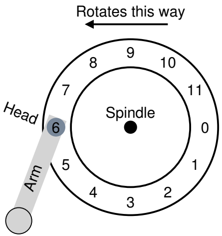


Basic Geometry



Single-track Latency

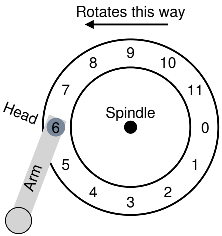
- Single track, with 12 sectors
- **Rotational delay**: wait for desired sector to reach disk head:



- Full rotational delay is R
  - Wait for sector 0?

Single-track Latency

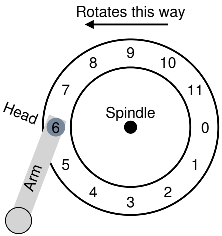
- Single track, with 12 sectors
- **Rotational delay**: wait for desired sector to reach disk head:



- Full rotational delay is R
  - Wait for sector 0? $\frac{R}{2}$
  - Worst-case request?

Single-track Latency

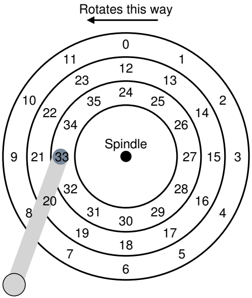
- Single track, with 12 sectors
- **Rotational delay**: wait for desired sector to reach disk head:

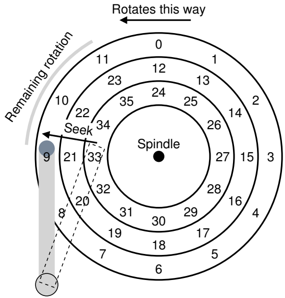


- Full rotational delay is R
  - Wait for sector 0? $\frac{R}{2}$
  - Worst-case request? sector 5 ($\frac{11R}{12}$)

Multiple Tracks

- **Seek**: move disk arm to the correct track
  - Costly disk operation, along with rotation
  - Acceleration: disk arm gets moving
  - Coasting: moving at full speed
  - Deceleration: arm slows down
  - Settling: head carefully positioned over correct track
  - **Settling time**: often quite significant, e.g., 0.5 to 2 ms



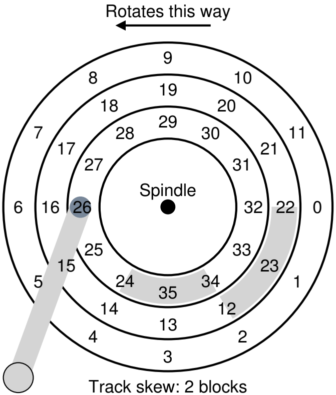


I/O Time

- Seek
- Wait for rotational delay
- **Transfer**: data is read from or written to surface

Other Details

- **Track skew**
  - Switching tracks → time to reposition the head
  - Without skew, desired next block already rotated → have to wait almost entire rotational delay



Track skew: 2 blocks

Other Details

- **Multi-zoned** disk drives
  - Outer tracks have more sectors than inner tracks
  - Disk is organized into multiple zones
  - Each zone has the same number of sectors per track
  - Outer zones have more sectors than inner zones

Other Details

- **Cache**
  - Hold data read from or written to disk (8 to 64 MB)
  - Quickly respond to requests
  - e.g., read all sectors on a track and cache in memory
  - **Write-through**
    - Acknowledge write when it's written to disk
  - **Writeback**
    - Acknowledge write when data is in cache
    - Faster but dangerous: consistency issues (order not guaranteed)

I/O Time

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

- Rate of I/O: $R_{I/O} = \frac{Size_{transfer}}{T_{I/O}}$

- **Random** workload
  - Small 4KB reads to random locations
- **Sequential** workload
  - Read 100MB of consecutive sectors

I/O Time

- Two example modern disks (Seagate):

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

I/O Time

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

- On the Cheetah:
  - $T_{seek} =$
  - $T_{rotation} =$
  - Random $T_{transfer} =$
  - Seq. $T_{transfer} =$
- On the Barracuda:
  - $T_{seek} =$
  - $T_{rotation} =$
  - Random $T_{transfer} =$
  - Seq. $T_{transfer} =$

I/O Time

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

- On the Cheetah:
  - $T_{seek} = 4ms$
  - $T_{rotation} =$
  - Random $T_{transfer} =$
  - Seq. $T_{transfer} =$
- On the Barracuda:
  - $T_{seek} = 9ms$
  - $T_{rotation} =$
  - Random $T_{transfer} =$
  - Seq. $T_{transfer} =$

I/O Time

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

- On the Cheetah:
  - $T_{seek} = 4ms$
  - $T_{rotation} = 2ms$
  - Random $T_{transfer} =$
  - Seq. $T_{transfer} =$
- On the Barracuda:
  - $T_{seek} = 9ms$
  - $T_{rotation} = 4.2ms$
  - Random $T_{transfer} =$
  - Seq. $T_{transfer} =$

I/O Time

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

- On the Cheetah:
  - $T_{seek} = 4ms$
  - $T_{rotation} = 2ms$
  - Random $T_{transfer} = 30\mu s$
  - Seq. $T_{transfer} =$
- On the Barracuda:
  - $T_{seek} = 9ms$
  - $T_{rotation} = 4.2ms$
  - Random $T_{transfer} = 38\mu s$
  - Seq. $T_{transfer} =$

I/O Time

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

- On the Cheetah:
  - $T_{seek} = 4ms$
  - $T_{rotation} = 2ms$
  - Random $T_{transfer} = 30\mu s$
  - Seq. $T_{transfer} = 800ms$
- On the Barracuda:
  - $T_{seek} = 9ms$
  - $T_{rotation} = 4.2ms$
  - Random $T_{transfer} = 38\mu s$
  - Seq. $T_{transfer} = 950ms$

I/O Time

- On the Cheetah:
  - $T_{seek} = 4ms$
  - $T_{rotation} = 2ms$
  - Random $T_{transfer} = 30\mu s$
  - Seq. $T_{transfer} = 800ms$
- On the Barracuda:
  - $T_{seek} = 9ms$
  - $T_{rotation} = 4.2ms$
  - Random $T_{transfer} = 38\mu s$
  - Seq. $T_{transfer} = 950ms$

	Cheetah 15K.5	Barracuda
$T_{I/O}$ Random		
$R_{I/O}$ Random		
$T_{I/O}$ Sequential		
$R_{I/O}$ Sequential		

I/O Time

- On the Cheetah:
  - $T_{seek} = 4ms$
  - $T_{rotation} = 2ms$
  - Random $T_{transfer} = 30\mu s$
  - Seq. $T_{transfer} = 800ms$
- On the Barracuda:
  - $T_{seek} = 9ms$
  - $T_{rotation} = 4.2ms$
  - Random $T_{transfer} = 38\mu s$
  - Seq. $T_{transfer} = 950ms$

	Cheetah 15K.5	Barracuda
$T_{I/O}$ Random	6 ms	13.2 ms
$R_{I/O}$ Random		
$T_{I/O}$ Sequential		
$R_{I/O}$ Sequential		

I/O Time

- On the Cheetah:
  - $T_{seek} = 4ms$
  - $T_{rotation} = 2ms$
  - Random $T_{transfer} = 30\mu s$
  - Seq. $T_{transfer} = 800ms$
- On the Barracuda:
  - $T_{seek} = 9ms$
  - $T_{rotation} = 4.2ms$
  - Random $T_{transfer} = 38\mu s$
  - Seq. $T_{transfer} = 950ms$

	Cheetah 15K.5	Barracuda
$T_{I/O}$ Random	6 ms	13.2 ms
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$T_{I/O}$ Sequential		
$R_{I/O}$ Sequential		

I/O Time

- On the Cheetah:
  - $T_{seek} = 4ms$
  - $T_{rotation} = 2ms$
  - Random $T_{transfer} = 30\mu s$
  - Seq. $T_{transfer} = 800ms$
- On the Barracuda:
  - $T_{seek} = 9ms$
  - $T_{rotation} = 4.2ms$
  - Random $T_{transfer} = 38\mu s$
  - Seq. $T_{transfer} = 950ms$

	Cheetah 15K.5	Barracuda
$T_{I/O}$ Random	6 ms	13.2 ms
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$T_{I/O}$ Sequential	806 ms	963 ms
$R_{I/O}$ Sequential		

I/O Time

- On the Cheetah:
  - $T_{seek} = 4ms$
  - $T_{rotation} = 2ms$
  - Random $T_{transfer} = 30\mu s$
  - Seq. $T_{transfer} = 800ms$
- On the Barracuda:
  - $T_{seek} = 9ms$
  - $T_{rotation} = 4.2ms$
  - Random $T_{transfer} = 38\mu s$
  - Seq. $T_{transfer} = 950ms$

	Cheetah 15K.5	Barracuda
$T_{I/O}$ Random	6 ms	13.2 ms
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$T_{I/O}$ Sequential	806 ms	963 ms
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

I/O Time

- On the Cheetah:
  - $T_{seek} = 4ms$
  - $T_{rotation} = 2ms$
  - Random $T_{transfer} = 30\mu s$
  - Seq. $T_{transfer} = 800ms$
- On the Barracuda:
  - $T_{seek} = 9ms$
  - $T_{rotation} = 4.2ms$
  - Random $T_{transfer} = 38\mu s$
  - Seq. $T_{transfer} = 950ms$

	Cheetah 15K.5	Barracuda
$T_{I/O}$ Random	6 ms	13.2 ms
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$T_{I/O}$ Sequential	806 ms	963 ms
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

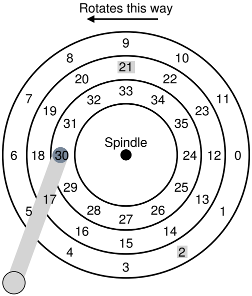
- **Use disks sequentially!**

Disk Scheduling

- **Disk scheduler**
  - OS examines requests and decides which to schedule next
  - Can make a good guess how long a job will take
    - By estimating seek and rotation delay
  - Greedily pick least time to service first

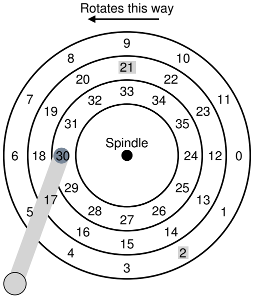
Disk Scheduling

- **Shortest Seek Time First (SSTF)**
  - Order I/O requests by track
  - Pick request on nearest track



Disk Scheduling

- **Shortest Seek Time First (SSTF)**
  - Order I/O requests by track
  - Pick request on nearest track



- Issue request to 21, then issue request to 2

Disk Scheduling

- Drive geometry not available to host OS
  - Sees an array of blocks
  - Solution?

Disk Scheduling

- Drive geometry not available to host OS
  - Sees an array of blocks
  - Solution? implement **nearest-block-first (NBF)**
- More fundamental problem?

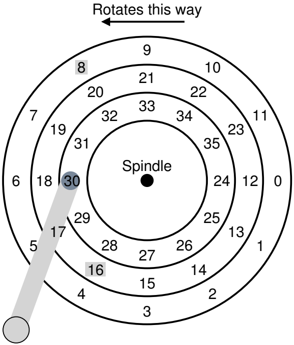
Disk Scheduling

- Drive geometry not available to host OS
    - Sees an array of blocks
    - Solution? implement **nearest-block-first (NBF)**
- More fundamental problem?
- **Starvation**
    - Steady stream of requests to inner track
    - Other tracks ignored completely

Disk Scheduling

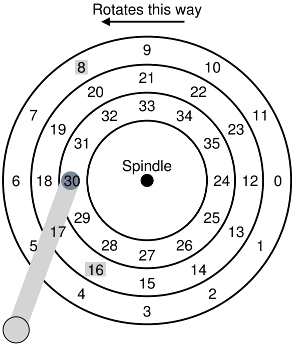
- **Elevator**
  - Service requests in order across the tracks (back and forth)
  - **Sweep**: single pass across the disk
    - Request of already-serviced track is queued until the next sweep
  - **F-SCAN**: freeze queue when doing a sweep
    - Prevents starvation of far-away requests
  - **C-SCAN**: sweep from outer-to-inner
    - Resets at outer track to begin again
    - Instead of both directions (favors middle tracks)
- Still problematic: ignores rotation

Disk Scheduling



- Schedule sector 16 or sector 8 next?

Disk Scheduling



- Schedule sector 16 or sector 8 next? it depends
  - Seek time much higher than rotational delay → SSTF
  - Seek faster than rotation → service request 8

Disk Scheduling

- Modern drives: seek and rotation times roughly equivalent
- **Shortest Positioning Time First (SPTF)**
  - Difficult to implement in OS
  - Usually performed inside a drive
  - OS picks best few requests and issues all to disk
- **I/O merging**
  - Series of requests - sectors 33, 8, then 34
  - OS merges 33 and 34 into a single two-block request
- **Work-conserving**
  - Wait before issuing I/O to disk
  - New and "better" request may arrive

Summary (Hard Disk Drives)

- 512-byte sectors
  - **Platter** with two **surfaces**, bound around the **spindle**
  - Fixed rate of **RPM**
  - Data encoded in **tracks**, read and write by **disk head**
- **Rotational delay**: wait for sector to reach head
- **Seek**: move disk arm to correct track
  - Acceleration → coasting → deceleration → settling
- **I/O time**: seek → wait for rotational delay → transfer
- **Cache** holds read/write data
  - **Write-through**: acknowledge on write to disk
  - **Writeback**: acknowledge when data is in cache
- Disk scheduling
  - **SSTF**, **NBF**, **Elevator** (**sweep**, **F-SCAN**, **C-SCAN**), **SPTF**
  - **I/O merging**: merge requests for consecutive sectors
  - **Work-conserving**: wait before issuing I/O to disk