

# CS194-24 Advanced Operating Systems Structures and Implementation Lecture 20

## Device Drivers (Con't) Disk Modeling

April 14<sup>th</sup>, 2014

Prof. John Kubiatowicz

<http://inst.eecs.berkeley.edu/~cs194-24>

## Goals for Today

- Device Drivers (Continued)
- Disk Drives and Queueing Theory

Interactive is important!  
Ask Questions!

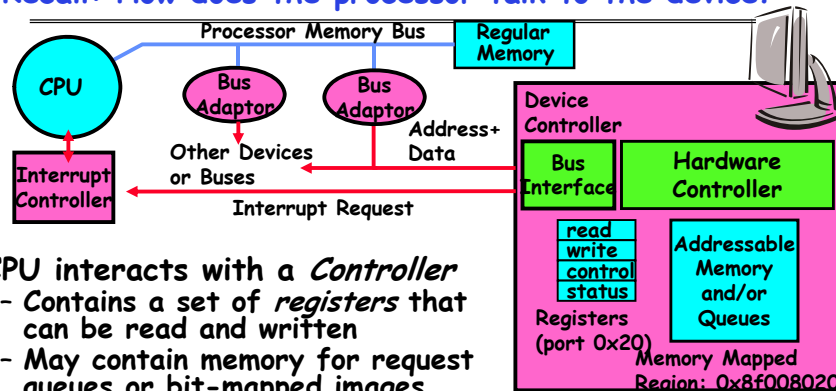
Note: Some slides and/or pictures in the following are adapted from slides ©2013

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.2

### Recall: How does the processor talk to the device?



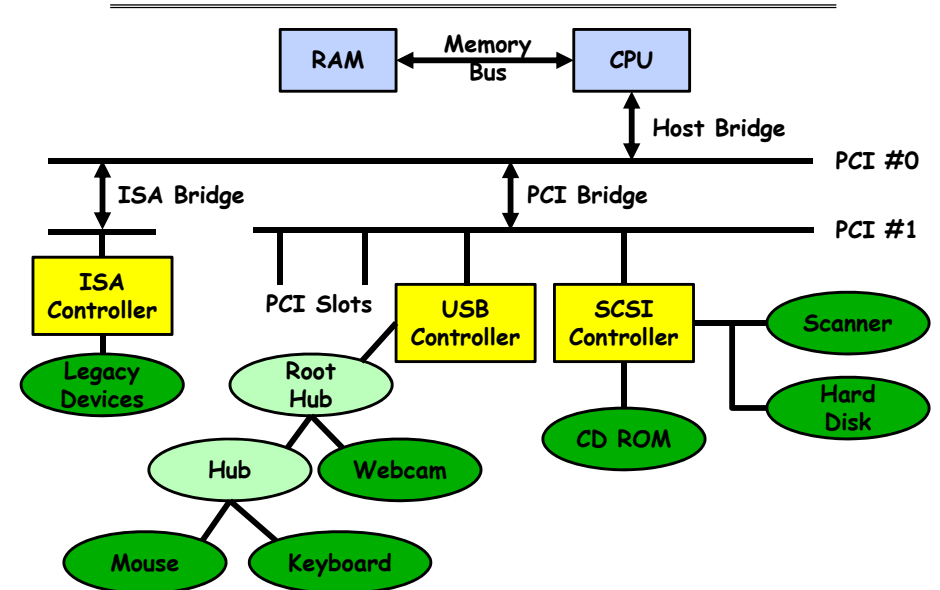
- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - **I/O instructions:** in/out instructions
    - » Example from the Intel architecture: out 0x21,AL
  - **Memory mapped I/O:** load/store instructions
    - » Registers/memory appear in physical address space
    - » I/O accomplished with load and store instructions

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.3

### Recall: PCI Architecture



4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.4

## PCI Details (con't)

- **Device identification:**
  - vendorID (16 bits): global registry of vendors
  - deviceID (16-bits): vendor-assigned device
  - class (16-bits): top 8 bits identify "base class" (i.e. network)
  - subsystem vendorID/subsystem deviceID
    - » Used to help identify bridges/interfaces

### • Example initialization:

```
#ifndef CONFIG_PCI
#error "This driver needs PCI support to be available"
#endif

int mydev_find_all_devices(void) {
    struct pci_dev *dev = NULL;
    int found;
    if (!pci_present()) return -ENODEV;

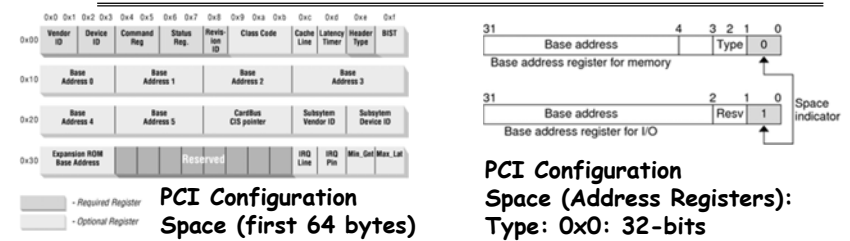
    for (found=0; found < MYDEV_MAX_DEV;) {
        dev = pci_find_device(MYDEV_VENDOR, MYDEV_ID, dev);
        if (!dev) /* no more devices are there */
            break; /* do device-specific actions and count the device */
        found += mydev_init_one(dev);
    }
    return (index == 0) ? -ENODEV : 0;
}
```

4/14/14

Kubiatowicz CS194-24 @UCB Fall 2014

Lec 20.5

## PCI Details (con't)



### • Access configuration space with special functions:

```
int pci_read_config_byte(struct pci_dev *dev, int where, u8 *ptr);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *ptr);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *ptr);
int pci_write_config_byte(struct pci_dev *dev, int where, u8 val);
int pci_write_config_word(struct pci_dev *dev, int where, u16 val);
int pci_write_config_dword(struct pci_dev *dev, int where, u32 val);
```

### • Example: Figure out which interrupt line

```
result = pci_read_config_byte(dev, PCI_INTERRUPT_LINE, &myirq);
if (result) { /* deal with error */ }

int request_irq(myirq,
               void (*handler)(int, void *, struct pt_regs *),
               unsigned long flags,
               const char *dev_name,
               void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

4/14/14

Kubiatowicz CS194-24 @UCB Fall 2014

Lec 20.6

## Device Drivers

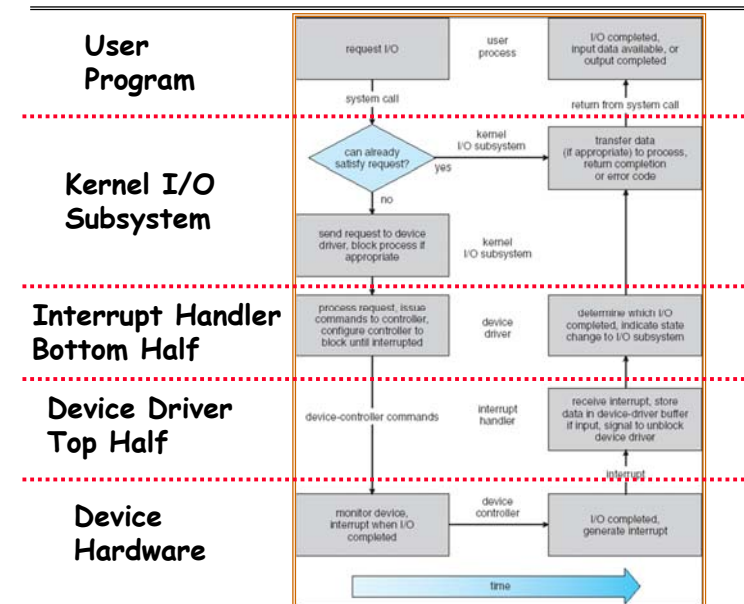
- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Linux Device drivers often installed via a Module
  - Interface for dynamically loading code into kernel space
  - Modules loaded with the "insmod" command and can contain parameters
- Driver-specific structure
  - One per driver
  - Contains a set of standard kernel interface routines
    - » Open: perform device-specific initialization
    - » Read: perform read
    - » Write: perform write
    - » Release: perform device-specific shutdown
    - » Etc.
  - These routines registered at time device registered

4/14/14

Kubiatowicz CS194-24 @UCB Fall 2014

Lec 20.7

## Life Cycle of An I/O Request



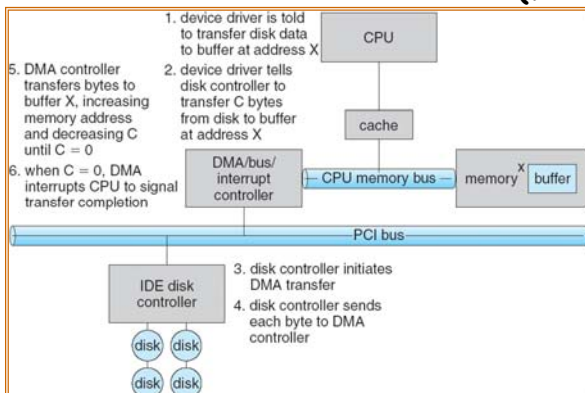
4/14/14

Kubiatowicz CS194-24 @UCB Fall 2014

Lec 20.8

## Transferring Data To/From Controller

- **Programmed I/O:**
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
- **Direct Memory Access:**
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly
- **Sample interaction with DMA controller (from book):**



4/14/14

Lec 20.9

## I/O Device Notifying the OS

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- **I/O Interrupt:**
  - Device generates an interrupt whenever it needs service
  - Handled in top half of device driver
    - » Often run on special kernel-level stack
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- **Polling:**
  - OS periodically checks a device-specific status register
    - » I/O device puts completion information in status register
    - » Could use timer to invoke lower half of drivers occasionally
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- **Actual devices combine both polling and interrupts**
  - For instance: High-bandwidth network device:
    - » Interrupt for first incoming packet
    - » Poll for following packets until hardware empty

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.10

## Administrivia

- **No Class on Wednesday**
  - Off at a DOE workshop
- **Use extra time on Wednesday to finish up Lab 3**
  - How's it going?
- **Special Topics lecture**
  - On Monday 5/5 during RRR week
  - What topics would you like me to talk about?
    - » Send me email!

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.11

## Heartbleed

- **SSL/TLS vulnerability in the OpenSSL library implementation of "heartbeats"**
- **RFC 6520: Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension**
- **What it does?**
  - Provides a keep-alive "heartbeat," and discovers how big a packet can be sent (Path Max Transfer Unit)
- **Why?**
  - Need to periodically send data on TCP/UDP connection so NAT boxes/firewalls don't close connection



4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.12

## How?

### • Message format:

```
struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    opaque payload[HeartbeatMessage.payload_length];
    opaque padding[padding_length];
} HeartbeatMessage;
```

Request or Response  
14-bit Length (16-bit field!)

- One side sends request with random data
  - Other side replies with response containing the SAME random data
  - All traffic is encrypted
  - Bug was an error in how the heartbeat receiver checks the message values
- Random Length Padding

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.13

## OpenSSL Code Analysis

```
hbtype = *p++;
n2s(p, payload);
pl = p;
```

Macro to extract TWO BYTE Length  
Payload

- The length in the SSLv3 record is not checked!

### • Later in the function:

```
unsigned char *buffer;
int r;
buffer = OPENSSL_malloc(r);
bp = buffer;
```

Allocate as much mem as requester asked for – up to ~64K bytes!

### • Then:

```
/* Enter response type,
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload_length);
```

Copies payload AND server memory beyond payload!

- Repeat to read server memory!
  - No log entries, so no forensic information

Analysis by <http://blog.existentialize.com/diagnosis-of-the-openssl-heartbleed-bug.html>

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.14

## Interrupt handling

- Interrupt routines typically divided into two pieces:
  - Top half: run as interrupt routine
    - » Gets input or transfers next block of output
    - » Handles any direct access to hardware
    - » Handles any time-sensitive aspects of handling interrupts
    - » Runs in the ATOMIC Context (cannot sleep)
  - Bottom half: accessed later to finish processing
    - » Perform any interrupt-related work not performed by the interrupt handler itself
    - » Scheduled "later" with interrupts re-enabled
    - » Some options for bottom halves can sleep
- Since you typically have two halves of code, must remember to *synchronize* shared data
  - Since interrupt handler is running in interrupt (ATOMIC) context, cannot sleep!
  - Good choice: spin lock to synchronize data structures
  - Must be careful never to hold spinlock for too long
    - » When non-interrupt code holds a spinlock, must make sure to disable interrupts!
    - » Consider "spin\_lock\_irqsave()" or "spin\_lock\_bh()" variants
  - Consider lock free queue variants as well

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.15

## More on Synchronization

- Must always be aware that interrupt handlers can interrupt running code!
  - Must come up with a synchronization methodology to deal with this issue
  - May need to deal with multiple processors
- Some possible ways of dealing with synchronization:
  - Build some sort of lock-free queue implemented as a circular buffer
  - Spinlocks
  - Lock variables that are atomically incremented and decremented
- Note about spinlocks
  - Many variants, make sure to use variants to disable interrupts as well as spin
  - Bovee Chapter 9 has lots of discussion of synchronization

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.16

## Recall: Portable Spinlock constructs in Linux

- Linux provides lots of synchronization constructs
  - We will highlight them throughout the term
- Example: Spin Lock support: Not recursive!
  - Only a lock on multiprocessors: Becomes simple preemption disable/enable on uniprocessors

```
#include <linux/spinlock.h>
DEFINE_SPINLOCK(my_lock);
```

```
spin_lock(&my_lock);
/* Critical section ... */
spin_unlock(&my_lock);
```

- Disable interrupts and grab lock (while saving and restoring state in case interrupts already disabled):

```
DEFINE_SPINLOCK(my_lock);
unsigned long flags;
```

```
spin_lock_irqsave(&my_lock, flags);
/* Critical section ... */
spin_unlock_irqrestore(&my_lock);
```

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.17

## Recall: Portable Atomic operations in Linux

- Linux provides `atomic_t` for declaring an atomic integer
  - Also, `atomic64_t` for declaring atomic 64-bit variant
  - Not necessarily same as a regular integer!
    - » Originally on SPARC, `atomic_t` ⇒ only 24 of 32 bits usable

- Example usage:

```
atomic_t v; /* define v */
atomic_t u = ATOMIC_INIT(0); /* define and init u=0 */
```

```
atomic_set(&v, 4); /* v=4 (atomically) */
atomic_add(2, &v); /* v = v + 2 (atomically) */
atomic_inc(&v); /* v = v + 1 (atomically) */
```

```
int final = atomic_read(&v); /* final == 7 */
```

- Some operations (see Love, Ch 10, Table 10.1/10.2):

```
atomic_inc()/atomic_dec() /* Atomically inc/dec */
atomic_add()/atomic_sub() /* Atomically add/sub */
int atomic_dec_and_test() /* Sub 1. True if 0 */
int atomic_inc_return() /* Add 1, return result */
```

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.18

## Recall: Portable bit operations in Linux

- Atomic bitwise operations operate on regular Ints.

- For example, to set  $n^{\text{th}}$  bit, starting from addr:

```
void set_bit(int nr, void *addr);
```

- Atomicity ensures that bit transitions are always seen atomically - regardless of competing concurrency

- » When bit is set and cleared - actually reflected as stores
- » When two different bits set - end up with two bits set, rather than one set operation erasing result of another

- Some operations (see Love, Ch 10, Table 10.3):

```
void set_bit() /* Atomically set bit */
void clear_bit() /* Atomically clear bit */
void change__bit() /* Atomically toggle bit */
int test_and_set_bit() /* set bit, return previous*/
int test_and_clear_bit() /* clear bit, return prev */
int test_and_change_bit() /* toggle bit, return prev */
int test_bit() /* Return value of bit*/
```

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.19

## Options for Bottom Half

- Bottom Half used for handling work after interrupt is re-enabled (i.e. deferred work):

- Perform any interrupt-related work not performed by the interrupt handler
- Ideally most of the work
- What to minimize amount of work done in an interrupt handler because they run with interrupts disabled

- Many different mechanisms for handling bottom halves

- Original "Bottom Half" (deprecated)
- Task Queues
  - » Put work on a task queue for later execution
- Softirqs are statically defined bottom halves that can run simultaneously on any processor
- Tasklets: dynamically created bottom halves built on top of softirq mechanism
  - » Only one of each type of tasklet can run at given time
  - » Simplifies synchronization

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.20

## Recall: Completion Patterns

- One use pattern that does not fit mutex pattern:
  - Start operation in another thread/hardware container
  - Sleep until woken by completion of event
- Can be implemented with semaphores
  - Start semaphore with count of 0
  - Immediate down() - puts parent to sleep
  - Woken with up()
- More efficient: use "completions":
 

```
DEFINED_COMPLETION();          /* Static definition */
struct completion my_comp;
init_completion(&my_comp);    /* Dynamic comp init */
```
- One or more threads to sleep on event:
 

```
wait_for_completion(&my_comp); /* put thread to sleep */
```
- Wake up threads (can be in interrupt handler!)
 

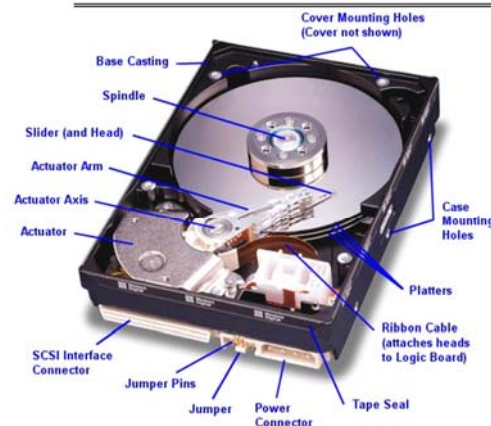
```
complete(&my_comp);
```

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.21

## Hard Disk Drives



Read/Write Head Side View



IBM/Hitachi Microdrive

Western Digital Drive  
<http://www.storagereview.com/guide/>

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.22

## Example: Seagate Barracuda (2014)

- 6TB! 1000 Gb/in<sup>2</sup>
- 6 (3.5") platters?, 2 heads each
- Perpendicular recording
- 7200 RPM, 4.16ms latency
- 4KB sectors (512 emulation?)
- 216MB/sec sustained transfer speed
- 128MB cache
- Error Characteristics:
  - MBTF: 1.4M hours
  - Bit error rate: 10<sup>-15</sup>
- Special considerations:
  - Normally need special "bios" (EFI): Bigger than easily handled by 32-bit OSes.
  - Seagate provides special "Disk Wizard" software that virtualizes drive into multiple chunks that makes it bootable on these OSes.

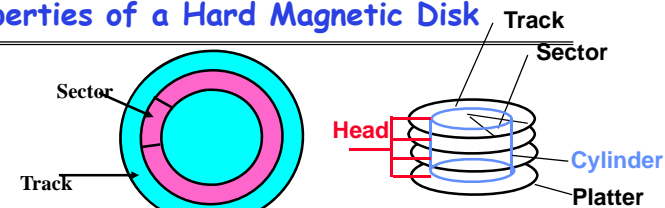


4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.23

## Properties of a Hard Magnetic Disk



- Properties
  - Head moves in to address circular **track** of information
  - Independently addressable element: **sector**
    - » OS always transfers groups of sectors together—"blocks"
  - Items addressable without moving head: **cylinder**
  - A disk can be rewritten in place: it is possible to read/modify/write a block from the disk
- Typical numbers (depending on the disk size):
  - 500 to more than 20,000 tracks per surface
  - 32 to 800 sectors per track
- Zoned bit recording
  - Constant bit density: more sectors on outer tracks
  - Speed varies with track location

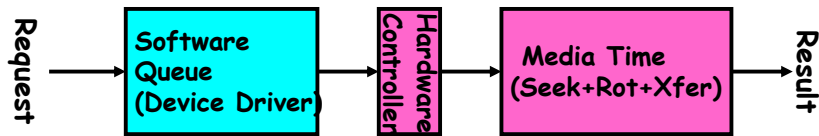
4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.24

## Performance Model

- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- **Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time**



- **Highest Bandwidth:**
  - Transfer large group of blocks sequentially from one track

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.25

## Typical Numbers of a Magnetic Disk

- Average seek time as reported by the industry:
  - Typically in the range of 4 ms to 12 ms
  - Locality of reference may only be 25% to 33% of the advertised number
- Rotational Latency:
  - Most disks rotate at 3,600 to 7200 RPM (Up to 15,000RPM or more)
  - Approximately 16 ms to 8 ms per revolution, respectively
  - An average latency to the desired information is halfway around the disk: 8 ms at 3600 RPM, 4 ms at 7200 RPM
- Transfer Time is a function of:
  - Transfer size (usually a sector): 512B - 1KB per sector
  - Rotation speed: 3600 RPM to 15000 RPM
  - Recording density: bits per inch on a track
  - Diameter: ranges from 1 in to 5.25 in
  - Typical values: up to 216 MB per second (sustained)
- Controller time depends on controller hardware

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.26

## Example: Disk Performance

- Question: How long does it take to fetch 1 Kbyte sector?
- Assumptions:
  - Ignoring queuing and controller times for now
  - Avg seek time of 5ms, avg rotational delay of 4ms
  - Transfer rate of 4MByte/s, sector size of 1 KByte
- Random place on disk:
  - Seek (5ms) + Rot. Delay (4ms) + Transfer (0.25ms)
  - Roughly 10ms to fetch/put data: 100 KByte/sec
- Random place in same cylinder:
  - Rot. Delay (4ms) + Transfer (0.25ms)
  - Roughly 5ms to fetch/put data: 200 KByte/sec
- Next sector on same track:
  - Transfer (0.25ms): 4 MByte/sec
- Key to using disk effectively (esp. for filesystems) is to minimize seek and rotational delays

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.27

## What about other non-volatile options?

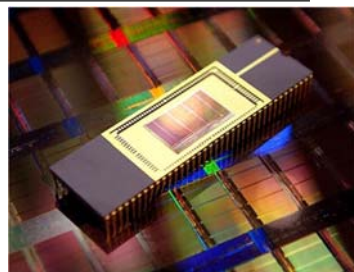
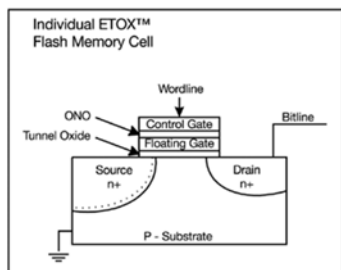
- There are a number of non-mechanical options for non-volatile storage
  - FLASH, MRAM, PCM
- Form Factors:
  - SSD (same form factor and interface as disk)
    - » Complex on-board controllers perform maintenance operations
  - SIMMs/DIMMs
    - » May need to have device driver perform wear-leveling or other operations
- Current SSD densities
  - 1TB no problem (< \$500)
  - Seems to be pretty competitive

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.28

## FLASH Memory



Samsung 2007:  
16GB, NAND Flash

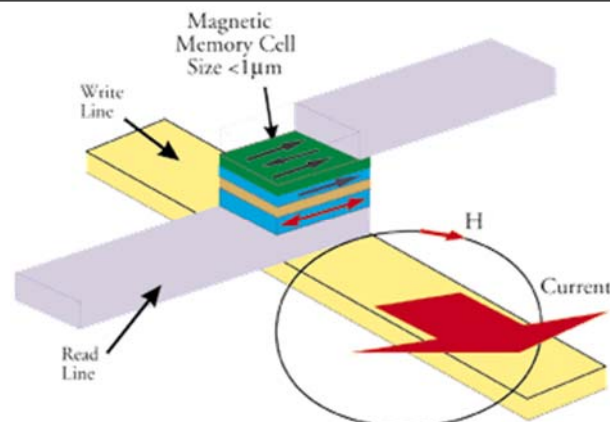
- Like a normal transistor but:
  - Has a floating gate that can hold charge
  - To write: raise or lower wordline high enough to cause charges to tunnel
  - To read: turn on wordline as if normal transistor
    - » presence of charge changes threshold and thus measured current
- Two varieties:
  - NAND: denser, must be read and written in blocks
  - NOR: much less dense, fast to read and write

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.29

## Tunneling Magnetic Junction (MRAM)



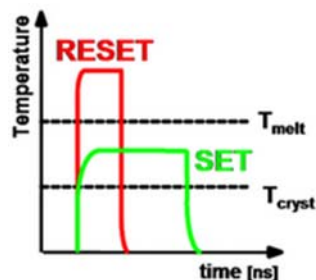
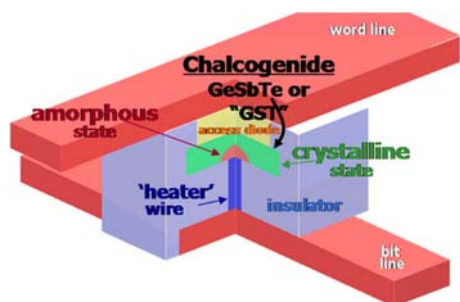
- Tunneling Magnetic Junction RAM (TMJ-RAM)
  - Speed of SRAM, density of DRAM, non-volatile (no refresh)
  - "Spintronics": combination quantum spin and electronics
  - Same technology used in high-density disk-drives

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.30

## Phase Change memory (IBM, Samsung, Intel)



- Phase Change Memory (called PRAM or PCM)
  - Chalcogenide material can change from amorphous to crystalline state with application of heat
  - Two states have very different resistive properties
  - Similar to material used in CD-RW process
- Exciting alternative to FLASH
  - Higher speed
  - May be easy to integrate with CMOS processes

4/14/14

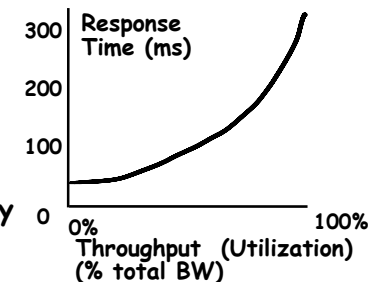
Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.31

## Properties of Magnetic Disk (Con't)

- Performance of disk drive/file system

- Metrics: Response Time, Throughput
- Contributing factors to latency:
  - » Software paths (can be loosely modeled by a queue)
  - » Hardware controller
  - » Physical disk media



- Queuing behavior:

- Leads to big increases of latency as utilization approaches 100%

4/14/14

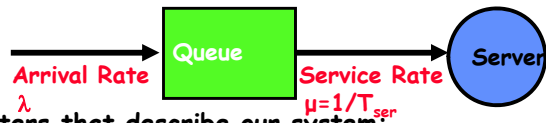
Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.32



## A Little Queuing Theory: Some Results

- Assumptions:
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless



- Parameters that describe our system:
  - $\lambda$ : mean number of arriving customers/second
  - $T_{ser}$ : mean time to service a customer ("m1")
  - $C$ : squared coefficient of variance =  $\sigma^2/m1^2$
  - $\mu$ : service rate =  $1/T_{ser}$
  - $u$ : server utilization ( $0 \leq u \leq 1$ ):  $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - $T_q$ : Time spent in queue
  - $L_q$ : Length of queue =  $\lambda \times T_q$  (by Little's law)
- Results:
  - Memoryless service distribution ( $C = 1$ ):
    - » Called **M/M/1 queue**:  $T_q = T_{ser} \times u/(1 - u)$
  - General service distribution (no restrictions), 1 server:
    - » Called **M/G/1 queue**:  $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u)$

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.33

## A Little Queuing Theory: An Example

- Example Usage Statistics:
  - User requests  $10 \times 8\text{KB}$  disk I/Os per second
  - Requests & service exponentially distributed ( $C=1.0$ )
  - Avg. service = 20 ms (controller+seek+rot+Xfertime)
- Questions:
  - How utilized is the disk?
    - » Ans: server utilization,  $u = \lambda T_{ser}$
  - What is the average time spent in the queue?
    - » Ans:  $T_q$
  - What is the number of requests in the queue?
    - » Ans:  $L_q = \lambda T_q$
  - What is the avg response time for disk request?
    - » Ans:  $T_{sys} = T_q + T_{ser}$  (Wait in queue, then get served)
- Computation:
  - $\lambda$  (avg # arriving customers/s) = 10/s
  - $T_{ser}$  (avg time to service customer) = 20 ms (0.02s)
  - $u$  (server utilization) =  $\lambda \times T_{ser} = 10/\text{s} \times .02\text{s} = 0.2$
  - $T_q$  (avg time/customer in queue) =  $T_{ser} \times u/(1 - u) = 20 \times 0.2/(1-0.2) = 20 \times 0.25 = 5 \text{ ms}$  (0.005s)
  - $L_q$  (avg length of queue) =  $\lambda \times T_q = 10/\text{s} \times .005\text{s} = 0.05$
  - $T_{sys}$  (avg time/customer in system) =  $T_q + T_{ser} = 25 \text{ ms}$

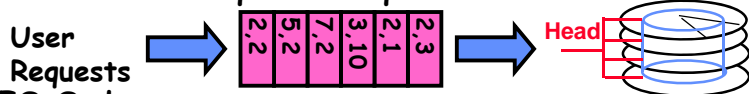
4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

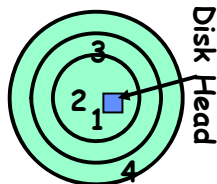
Lec 20.34

## Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queued requests?



- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk  $\Rightarrow$  Very long seeks
- SSTF: Shortest seek time first
  - Pick the request that's closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
  - Con: SSTF good at reducing seeks, but may lead to starvation
- SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
  - No starvation, but retains flavor of SSTF
- C-SCAN: Circular-Scan: only goes in one direction
  - Skips any requests on the way back
  - Fairer than SCAN, not biased towards pages in middle



4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.35

## Summary (1/2)

- I/O Devices Types:
  - Many different speeds (0.1 bytes/sec to GBytes/sec)
  - Different Access Patterns:
    - » Block Devices, Character Devices, Network Devices
  - Different Access Timing:
    - » Blocking, Non-blocking, Asynchronous
- I/O Controllers: Hardware that controls actual device
  - Processor Accesses through I/O instructions, load/store to special physical memory
  - Report their results through either interrupts or a status register that processor looks at occasionally (polling)
- Notification mechanisms
  - Interrupts
  - Polling: Report results through status register that processor looks at periodically
- Device Driver: Code specific to device which handles unique aspects of device

4/14/14

Kubiatowicz CS194-24 ©UCB Fall 2014

Lec 20.36

## Summary (2/2)

---

- **Disk Storage: Cylinders, Tracks, Sectors**
  - Access Time: 4-12ms
  - Rotational Velocity: 3600—15000
  - Transfer Speed: Up to 200MB/sec
- **Disk Time =**  
queue + controller + seek + rotate + transfer
- **Advertised average seek time benchmark much greater than average seek time in practice**
- **Queueing theory:**  $w = \left( \frac{1}{2} \frac{(1+c)\bar{x}u}{1-u} \right)$  for (c=1):  $w = \left( \frac{\bar{x}u}{1-u} \right)$