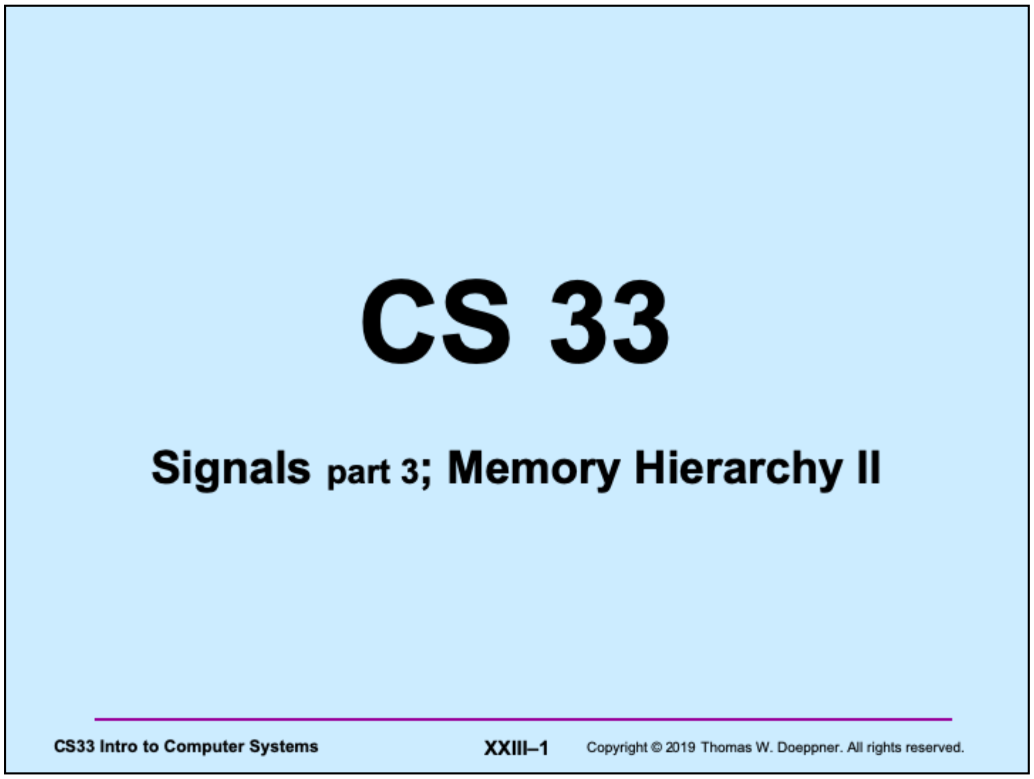
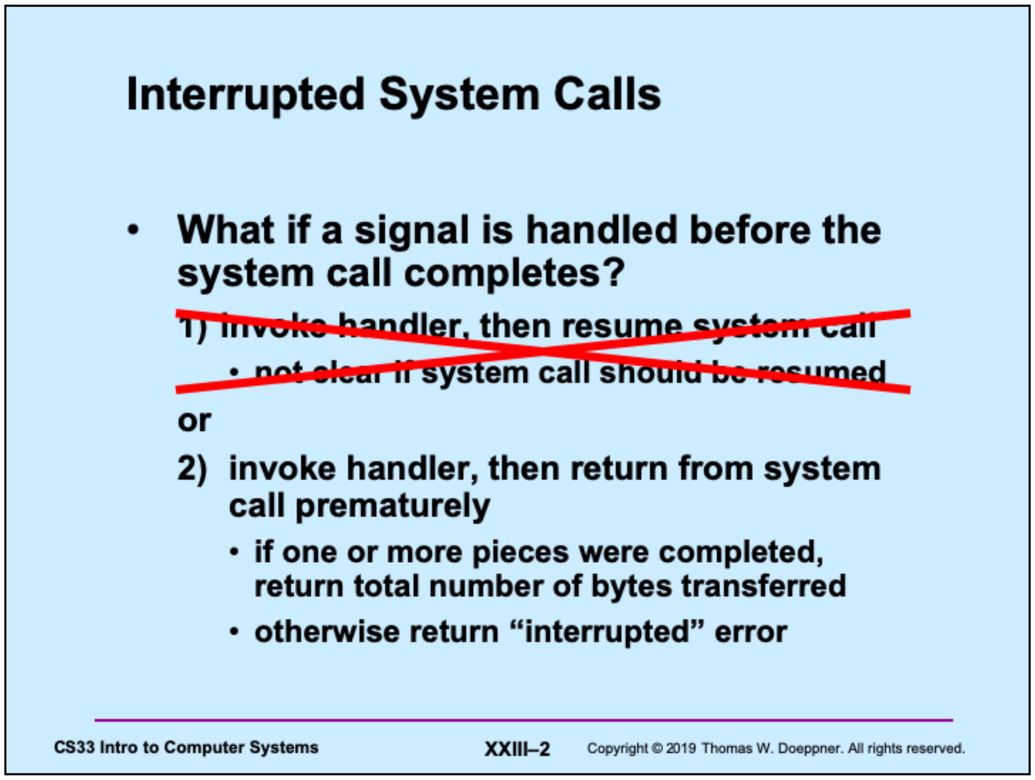


CS 33

Signals part 3; Memory Hierarchy II

Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.



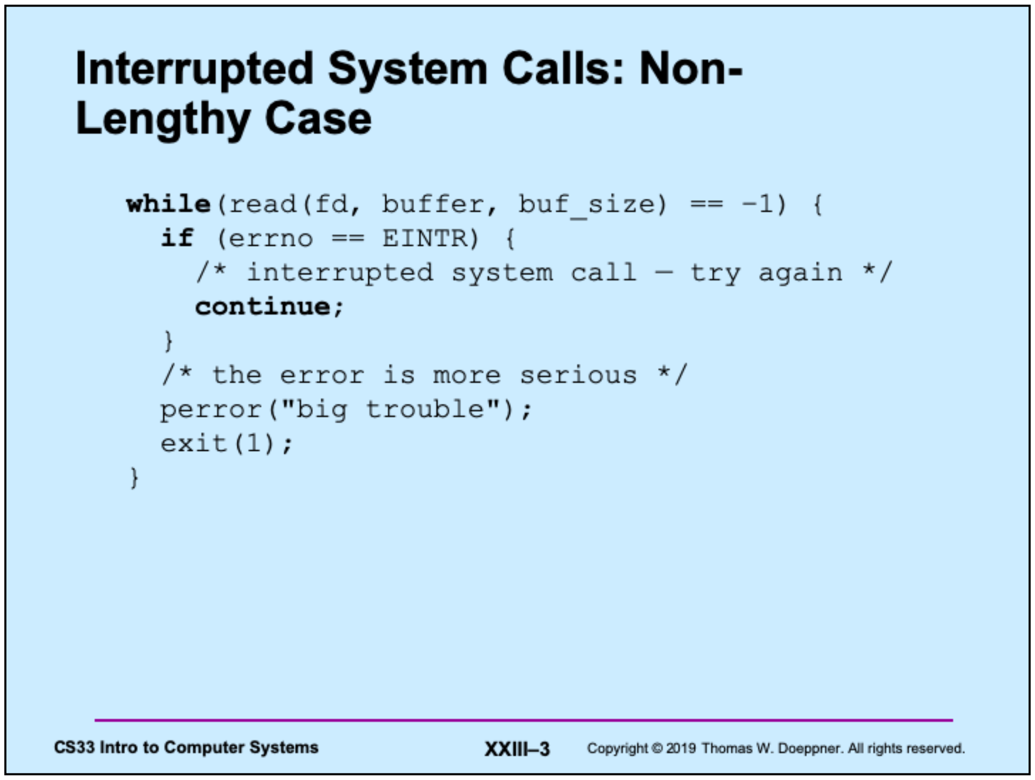
Interrupted System Calls

- What if a signal is handled before the system call completes?
  - ~~1) invoke handler, then resume system call~~
    - ~~not clear if system call should be resumed~~

  or

  2) invoke handler, then return from system call prematurely
     - if one or more pieces were completed, return total number of bytes transferred
     - otherwise return "interrupted" error

What happens to the system call after the signal handling completes (assuming that the process has not been terminated)? The fact that a signal has occurred may be an indication that the system call shouldn't be resumed. For example, the signal may have been generated in an attempt to stop whatever the system call was doing. Thus, rather than resume the system call, the system call is effectively terminated and either it returns an indication of how far it progressed before being interrupted by the signal (it would return the number of bytes actually transferred, as opposed to the number of bytes requested) or, if it was interrupted before anything actually happened, it returns an error indication and sets *errno* to EINTR (meaning "interrupted system call").



Interrupted System Calls: Non-Lengthy Case

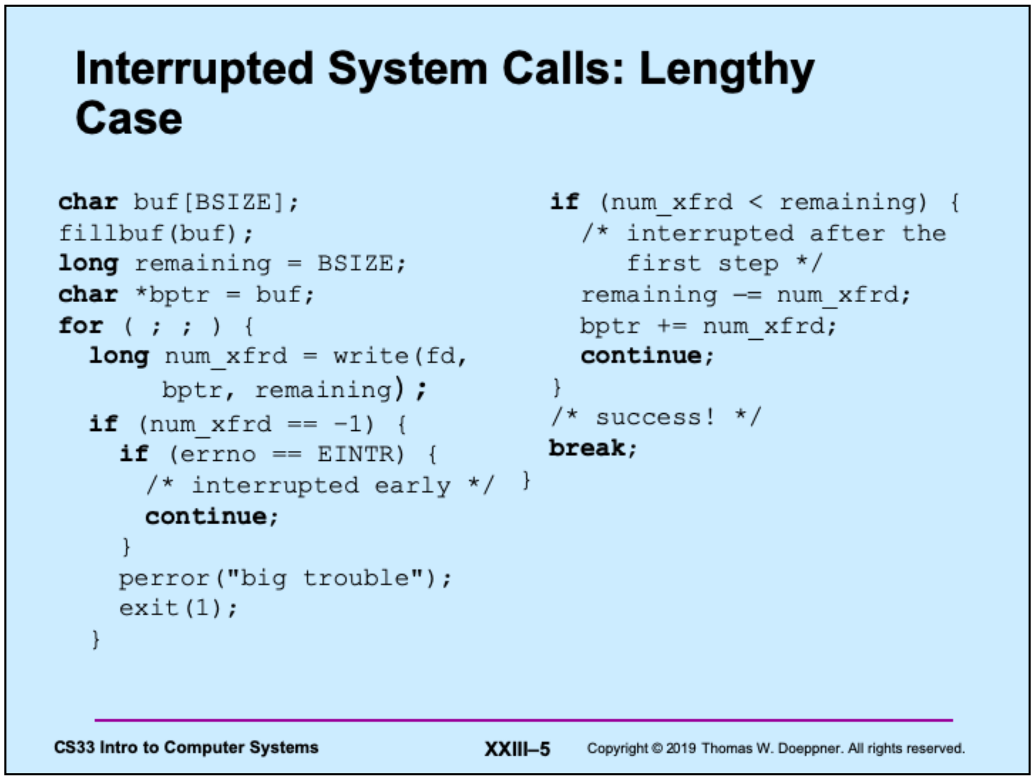
```
while(read(fd, buffer, buf_size) == -1) {
  if (errno == EINTR) {
    /* interrupted system call — try again */
    continue;
  }
  /* the error is more serious */
  perror("big trouble");
  exit(1);
}
```

If a non-lengthy system call is interrupted by a signal, the call fails and the error code EINTR is put in *errno*. The process then executes the signal handler and then returns to the point of the interrupt, which causes it to (finally) return from the system call with the error.

Quiz 1

```
int ret;
char buf[128];

fillbuf(buf);

ret = write(1, buf, 128);
```

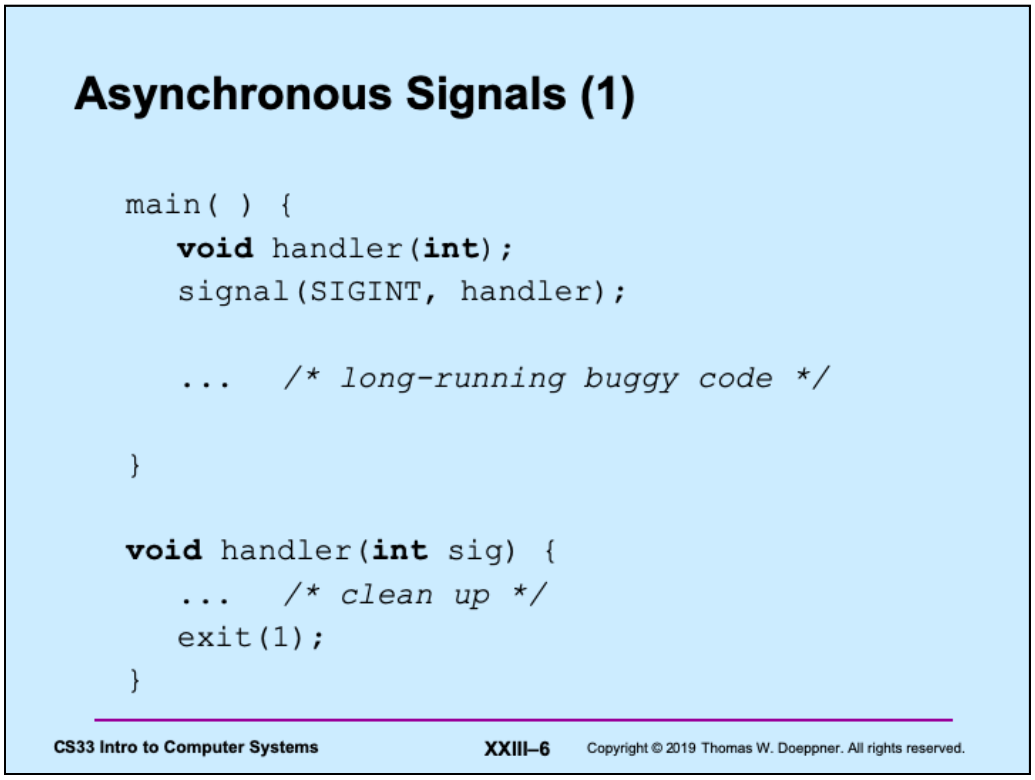
- **The value of ret is:**
  a) **either -1 or 128**
  b) **either -1, 0, or 128**
  c) **any integer in the range [-1, 128]**



Interrupted System Calls: Lengthy Case

```
char buf[BSIZE];
fillbuf(buf);
long remaining = BSIZE;
char *bptr = buf;
for ( ; ; ) {
  long num_xfrd = write(fd,
       bptr, remaining);
  if (num_xfrd == -1) {
    if (errno == EINTR) {
      /* interrupted early */
      continue;
    }
    perror("big trouble");
    exit(1);
  }
  if (num_xfrd < remaining) {
    /* interrupted after the
       first step */
    remaining -= num_xfrd;
    bptr += num_xfrd;
    continue;
  }
  /* success! */
  break;
}
```

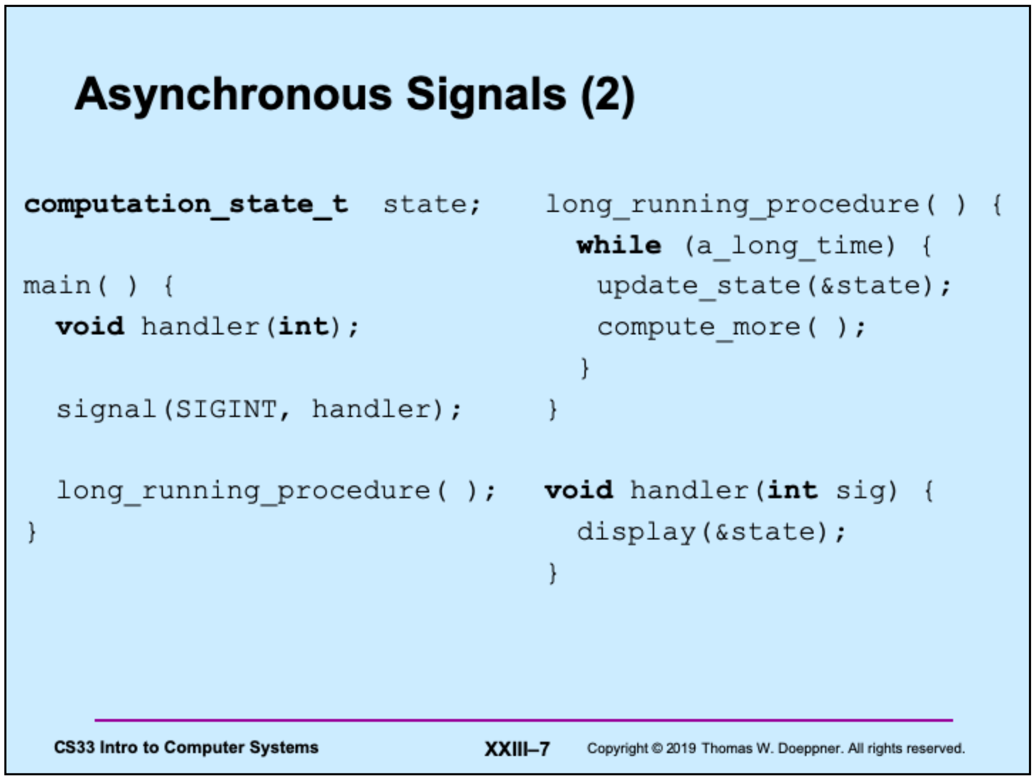
The actions of some system calls are broken up into discrete steps. For example, if one issues a system call to write a megabyte of data to a file, the write will actually be split by the kernel into a number of smaller writes. If the system call is interrupted by a signal after the first component write has completed (but while there are still more to be done), it would not make sense for the call to return an error code: such an error return would convince the program that none of the write had completed and thus all should be redone. Instead, the call completes successfully: it returns the number of bytes actually transferred, the signal handler is invoked, and, on return from the signal handler, the user program receives the successful return from the (shortened) system call.



Asynchronous Signals (1)

```
main( ) {
   void handler(int);
   signal(SIGINT, handler);

   ...   /* long-running buggy code */

}

void handler(int sig) {
   ...   /* clean up */
   exit(1);
}
```

Let's look at some of the typical uses for asynchronous signals. Perhaps the most common is to force the termination of the process. When the user types control-C, the program should terminate. There might be a handler for the signal, so that the program can clean up and then terminate.

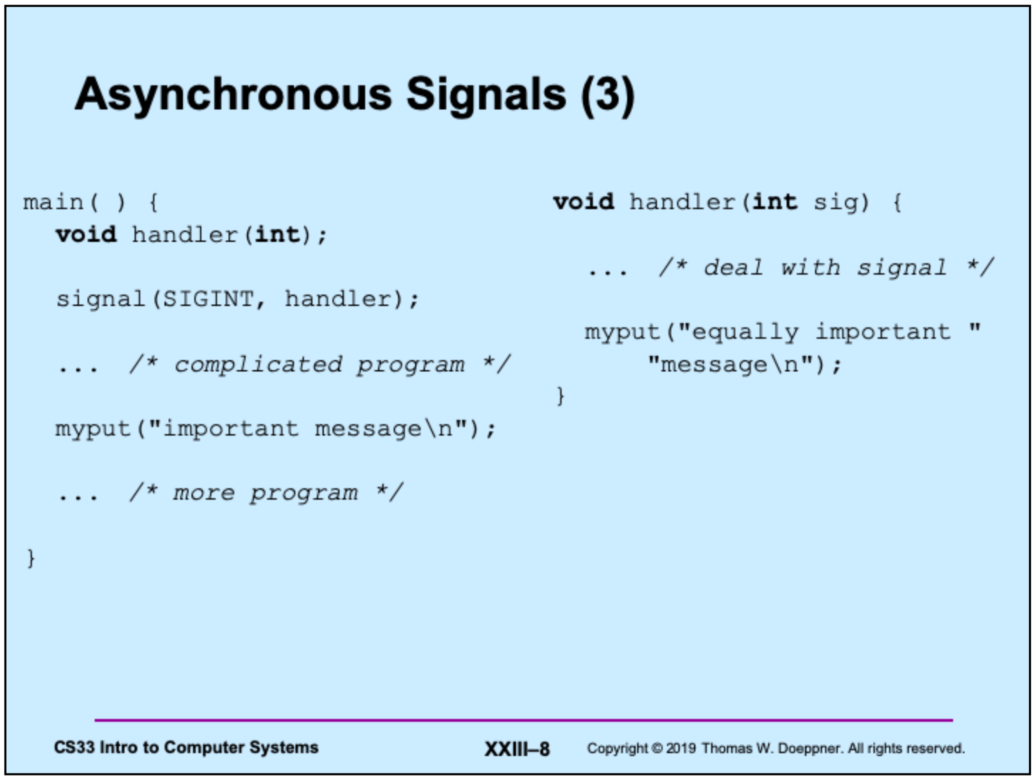


Asynchronous Signals (2)

```
computation_state_t  state;

main( ) {
  void handler(int);

  signal(SIGINT, handler);

  long_running_procedure( );
}
```

```
long_running_procedure( ) {
  while (a_long_time) {
    update_state(&state);
    compute_more( );
  }
}

void handler(int sig) {
  display(&state);
}
```

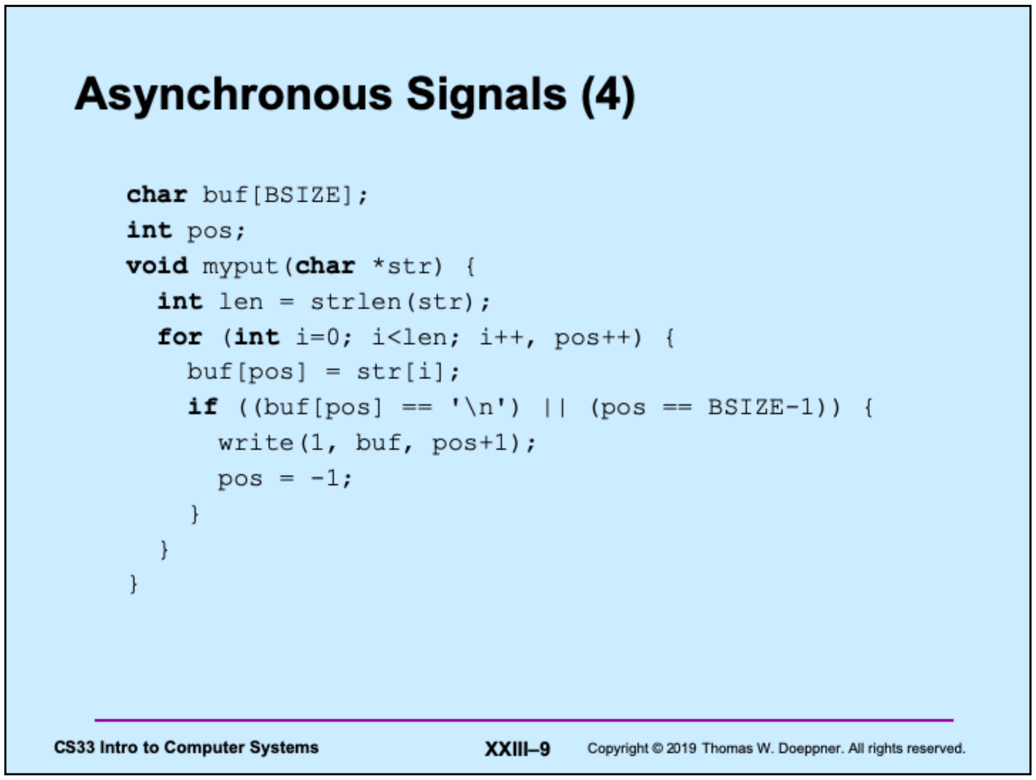
Here we are using a signal to send a request to a running program: when the user types control-C, the program prints out its current state and then continues execution. If synchronization is necessary so that the state is printed only when it is stable, it must be provided by appropriate settings of the signal mask.


Asynchronous Signals (3)

```
main( ) {
  void handler(int);

  signal(SIGINT, handler);

  ...  /* complicated program */

  myput("important message\n");

  ...  /* more program */

}
```

```
void handler(int sig) {

  ...  /* deal with signal */

  myput("equally important "
      "message\n");
}
```

In this example, both the mainline code and the signal handler call *myput*, which is similar to the standard-I/O routine *puts*. It's possible that the signal invoking the handler occurs while the mainline code is in the midst of the call to *myput*. Could this be a problem?



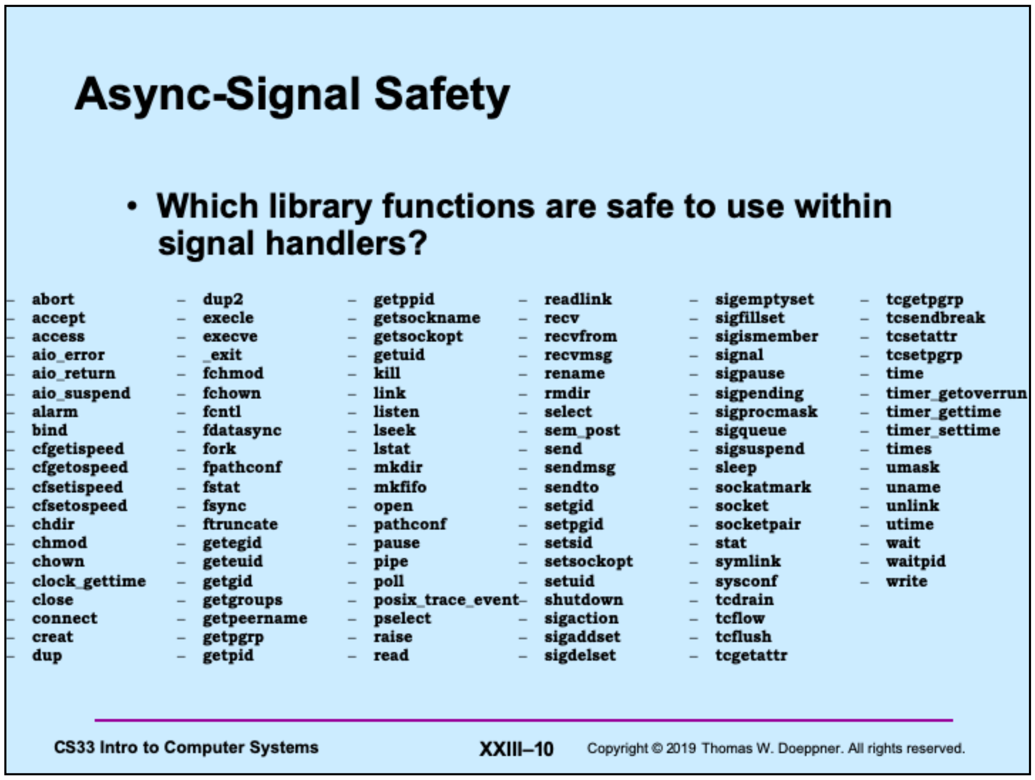
Asynchronous Signals (4)

```
char buf[BSIZE];
int pos;
void myput(char *str) {
  int len = strlen(str);
  for (int i=0; i<len; i++, pos++) {
    buf[pos] = str[i];
    if ((buf[pos] == '\n') || (pos == BSIZE-1)) {
      write(1, buf, pos+1);
      pos = -1;
    }
  }
}
```

Here's the implementation of *myput*, used in the previous slide. What it does is copy the input string, one character at a time, into *buf*, which is of size BSIZE. Whenever a newline character is encountered, the current contents of *buf* up to that point are written to standard output, then subsequent characters are copied starting at the beginning of *buf*. Similarly, if *buf* is filled, its contents are written to standard output and subsequent characters are copied starting at the beginning of *buf*. Since *buf* is global, characters not written out may be written after the next call to *myput*. Note that *printf* (and other stdio routines) buffers output in a similar way.

The point of *myput* is to minimize the number of calls to *write*, so that *write* is called only when we have a complete line of text or when its buffer is full.

However, consider what happens if execution is in the middle of *myput* when a signal occurs, as in the previous slide. Among the numerous problem cases, suppose *myput* is interrupted just after *pos* is set to -1 (if the code hadn't have been interrupted, *pos* would be soon incremented by 1). The signal handler now calls *myput*, which copies the first character of *str* into *buf[pos]*, which, in this case, is *buf[-1]*. Thus the first character "misses" the buffer. At best it simply won't be printed, but there might well be serious damage done to the program.



Async-Signal Safety

- Which library functions are safe to use within signal handlers?

– abort	– dup2	– getppid	– readlink	– sigemptyset	– tcgetpgrp
– accept	– execle	– getsockname	– recv	– sigfillset	– tcsendbreak
– access	– execve	– getsockopt	– recvfrom	– sigismember	– tcsetattr
– aio_error	– _exit	– getuid	– recvmsg	– signal	– tcsetpgrp
– aio_return	– fchmod	– kill	– rename	– sigpause	– time
– aio_suspend	– fchown	– link	– rmdir	– sigpending	– timer_getoverrun
– alarm	– fcntl	– listen	– select	– sigprocmask	– timer_gettime
– bind	– fdatasync	– lseek	– sem_post	– sigqueue	– timer_settime
– cfgetispeed	– fork	– lstat	– send	– sigsuspend	– times
– cfgetospeed	– fpathconf	– mkdir	– sendmsg	– sleep	– umask
– cfsetispeed	– fstat	– mkfifo	– sendto	– sockatmark	– uname
– cfsetospeed	– fsync	– open	– setgid	– socket	– unlink
– chdir	– ftruncate	– pathconf	– setpgid	– socketpair	– utime
– chmod	– getegid	– pause	– setsid	– stat	– wait
– chown	– geteuid	– pipe	– setsockopt	– symlink	– waitpid
– clock_gettime	– getgid	– poll	– setuid	– sysconf	– write
– close	– getgroups	– posix_trace_event	– shutdown	– tcdrain	
– connect	– getpeername	– pselect	– sigaction	– tcflow	
– creat	– getpgrp	– raise	– sigaddset	– tcflush	
– dup	– getpid	– read	– sigdelset	– tcgetattr	

To deal with the problem on the previous page, we must arrange that signal handlers cannot destructively interfere with the operations of the mainline code. Unless we are willing to work with signal masks (which can be expensive), this means we must restrict what can be done inside a signal handler. Routines that, when called from a signal handler, do not interfere with the operation of the mainline code, no matter what that code is doing, are termed *async-signal safe*. The POSIX 1003.1 spec requires the functions shown in the slide to be async-signal safe.

Note that POSIX specifies only those functions that must be async-signal safe. Implementations may make other functions async-signal safe as well.

What’s Inside A Disk Drive?

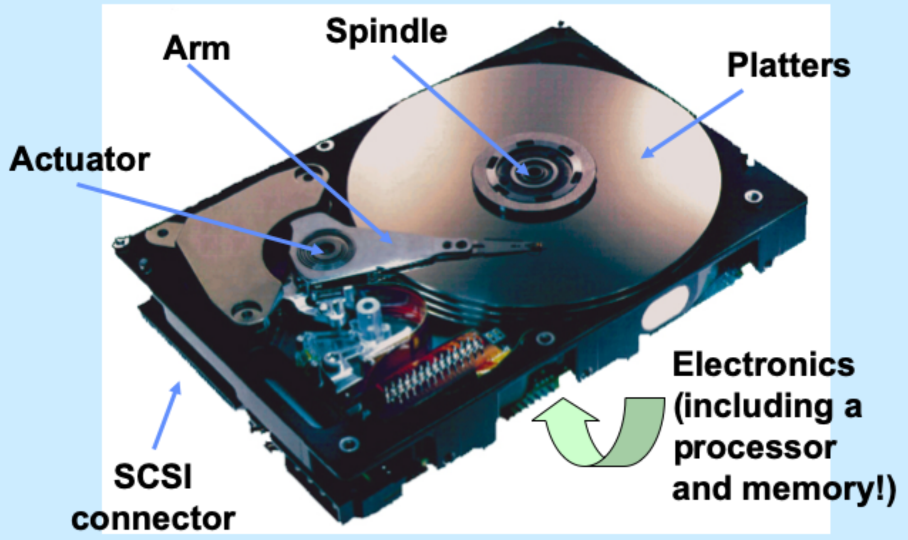
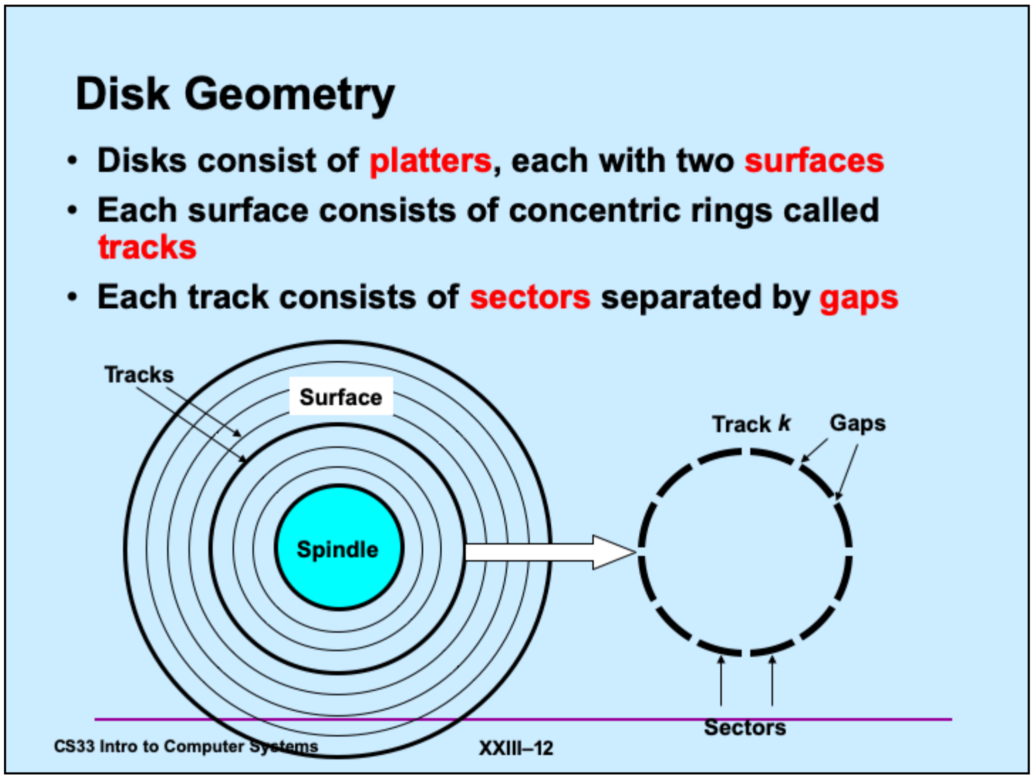


Image courtesy of Seagate Technology

Supplied by CMU.



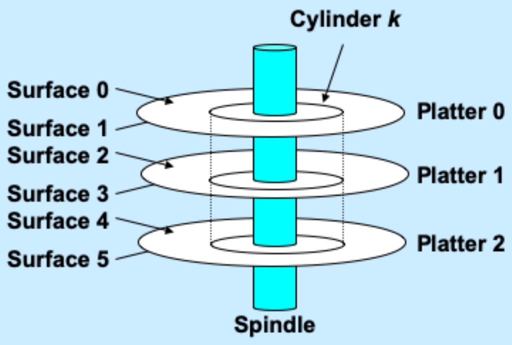
Disk Geometry

- Disks consist of **platters**, each with two **surfaces**
- Each surface consists of concentric rings called **tracks**
- Each track consists of **sectors** separated by **gaps**

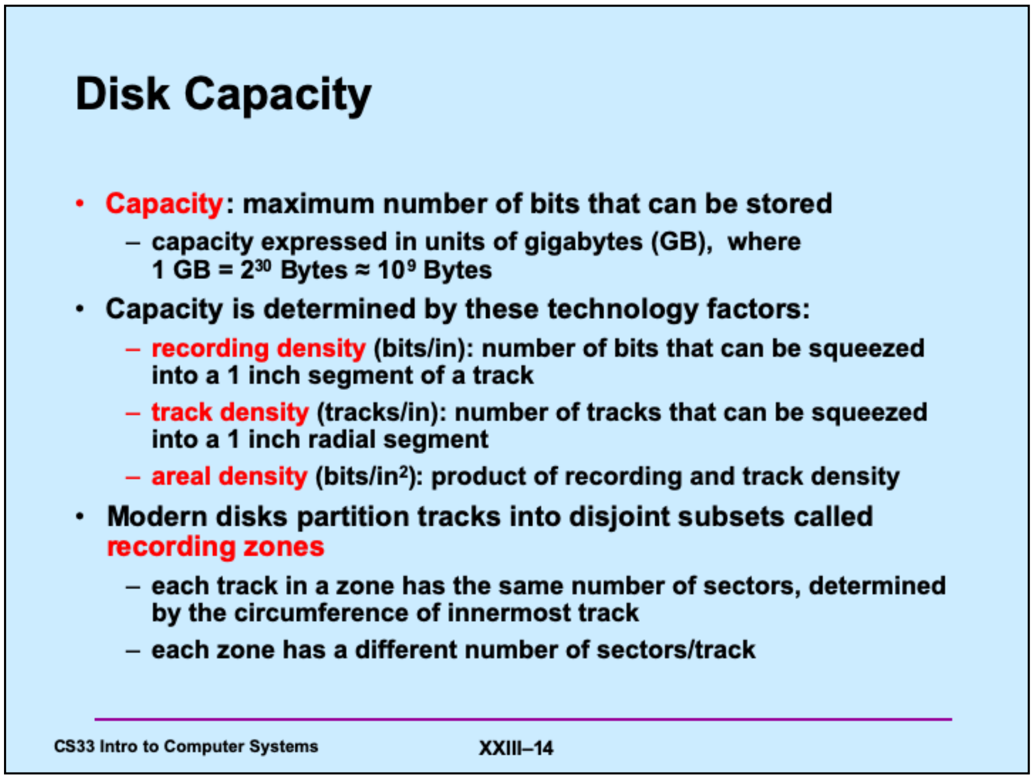
Supplied by CMU.

Disk Geometry (Multiple-Platter View)

- Aligned tracks form a cylinder



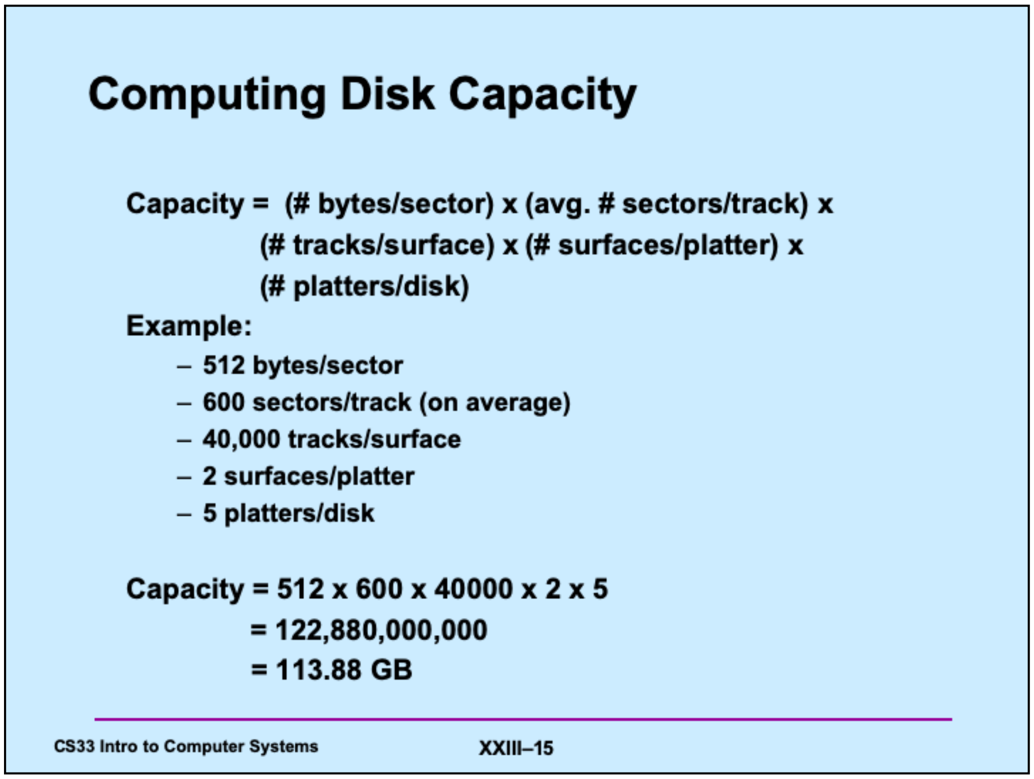
Supplied by CMU.



Disk Capacity

- **Capacity**: maximum number of bits that can be stored
  - capacity expressed in units of gigabytes (GB), where 1 GB = 2^{30} Bytes ≈ 10^9 Bytes
- Capacity is determined by these technology factors:
  - **recording density** (bits/in): number of bits that can be squeezed into a 1 inch segment of a track
  - **track density** (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment
  - **areal density** (bits/in^2): product of recording and track density
- Modern disks partition tracks into disjoint subsets called **recording zones**
  - each track in a zone has the same number of sectors, determined by the circumference of innermost track
  - each zone has a different number of sectors/track

Supplied by CMU.



Computing Disk Capacity

Capacity = (# bytes/sector) x (avg. # sectors/track) x
(# tracks/surface) x (# surfaces/platter) x
(# platters/disk)

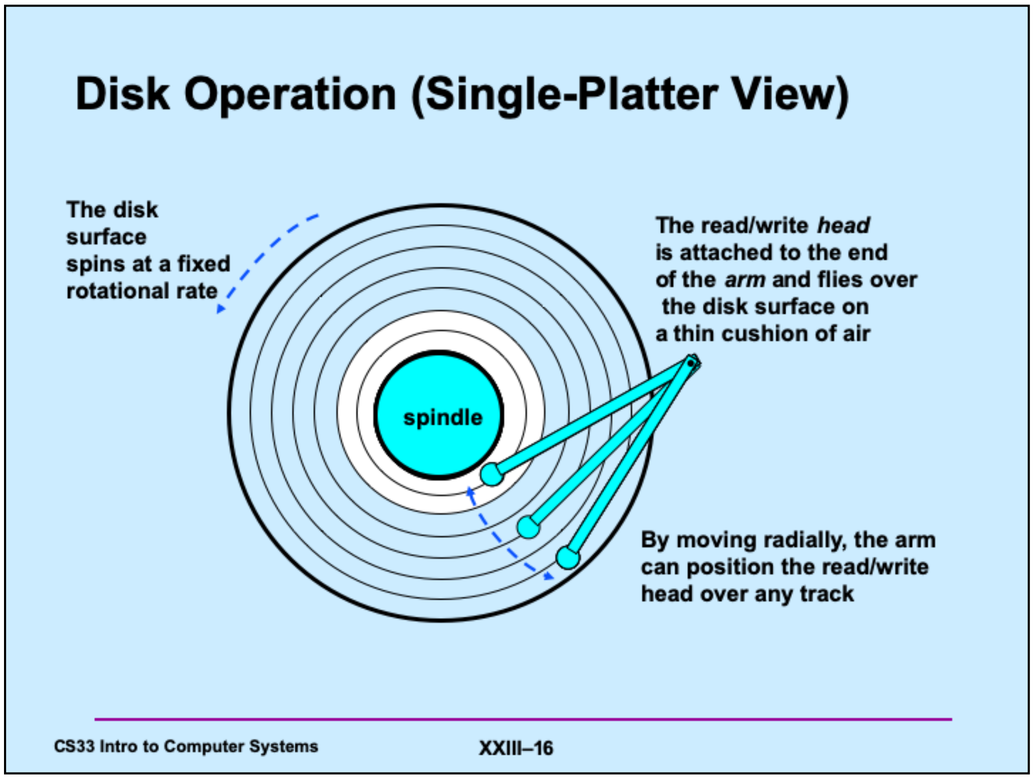
Example:

- 512 bytes/sector
- 600 sectors/track (on average)
- 40,000 tracks/surface
- 2 surfaces/platter
- 5 platters/disk

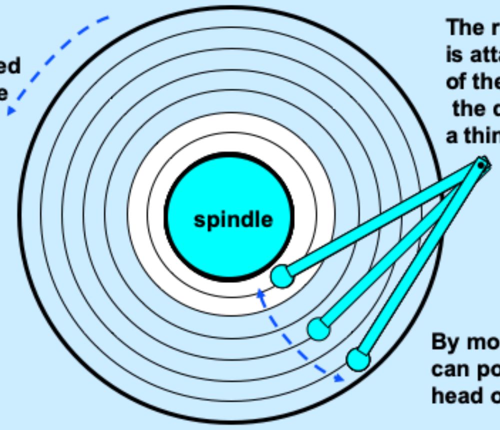
Capacity = 512 x 600 x 40000 x 2 x 5
= 122,880,000,000
= 113.88 GB

Supplied by CMU.

Note that 1GB = 2^{30} bytes.



Disk Operation (Single-Platter View)

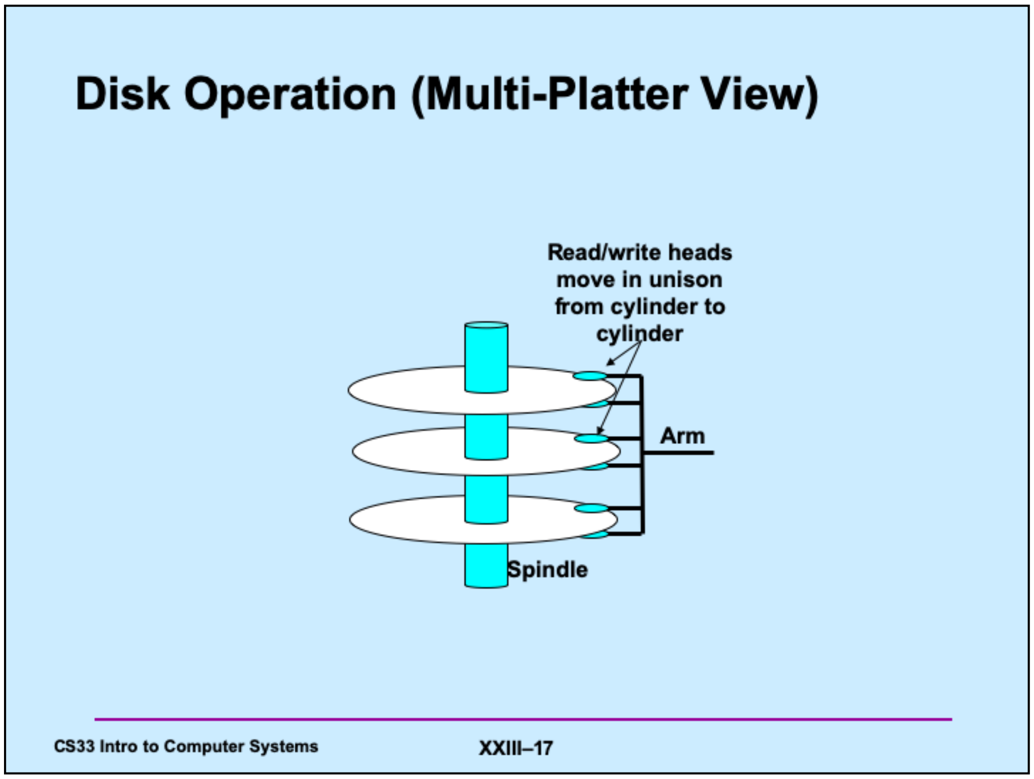


The disk surface spins at a fixed rotational rate

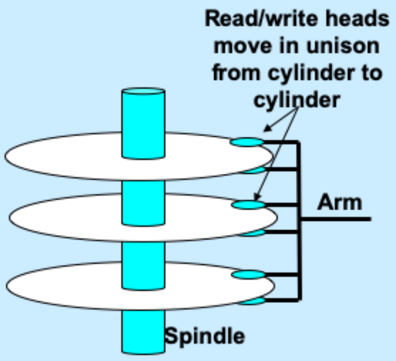
The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air

By moving radially, the arm can position the read/write head over any track

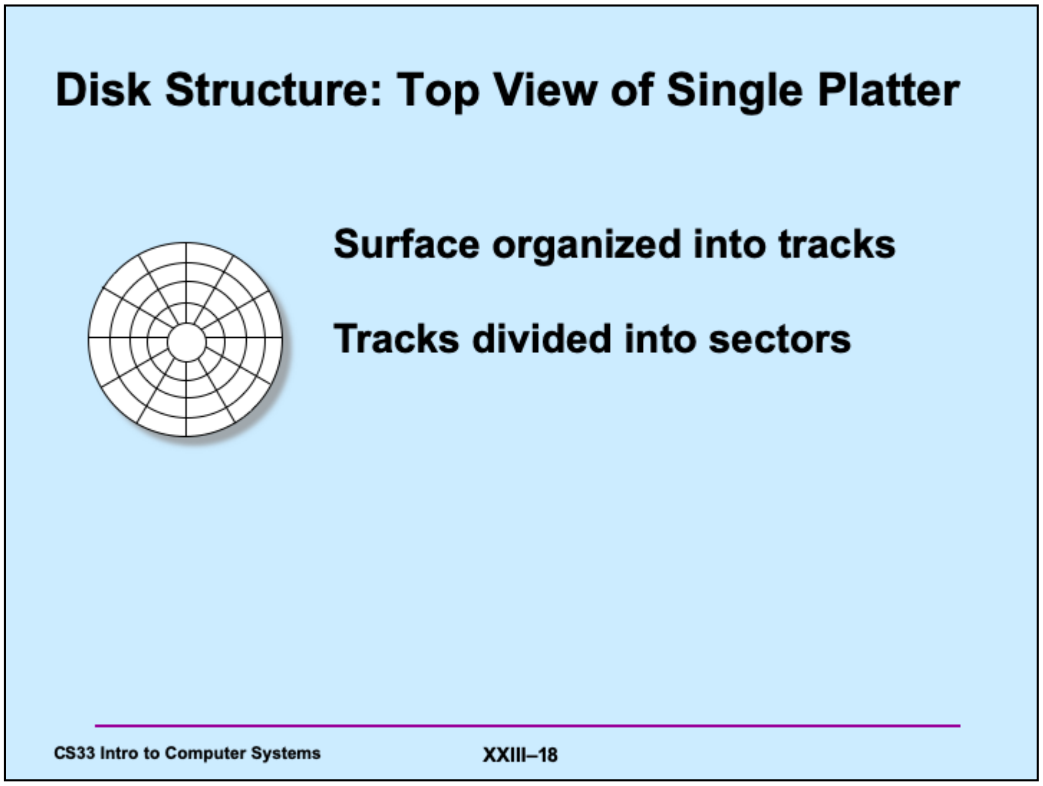
Supplied by CMU.



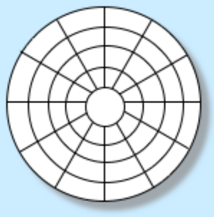
Disk Operation (Multi-Platter View)



Supplied by CMU.



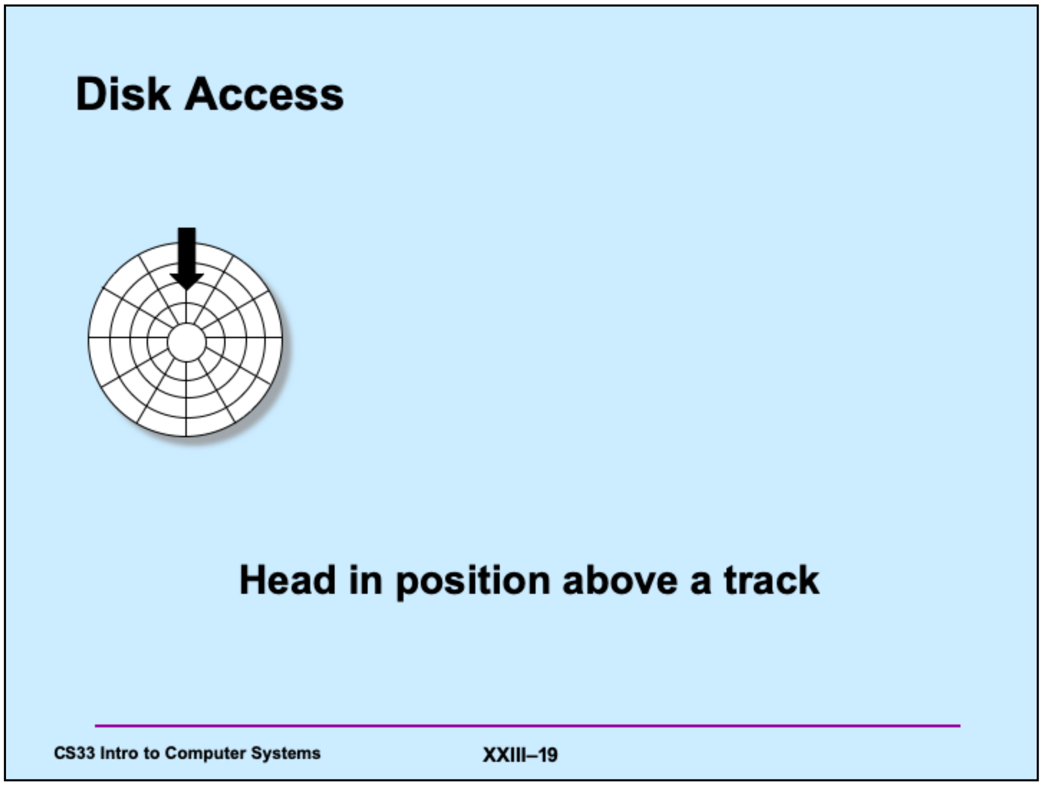
Disk Structure: Top View of Single Platter



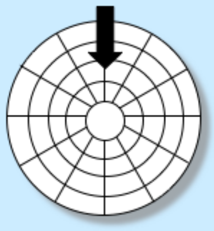
Surface organized into tracks

Tracks divided into sectors

Supplied by CMU.

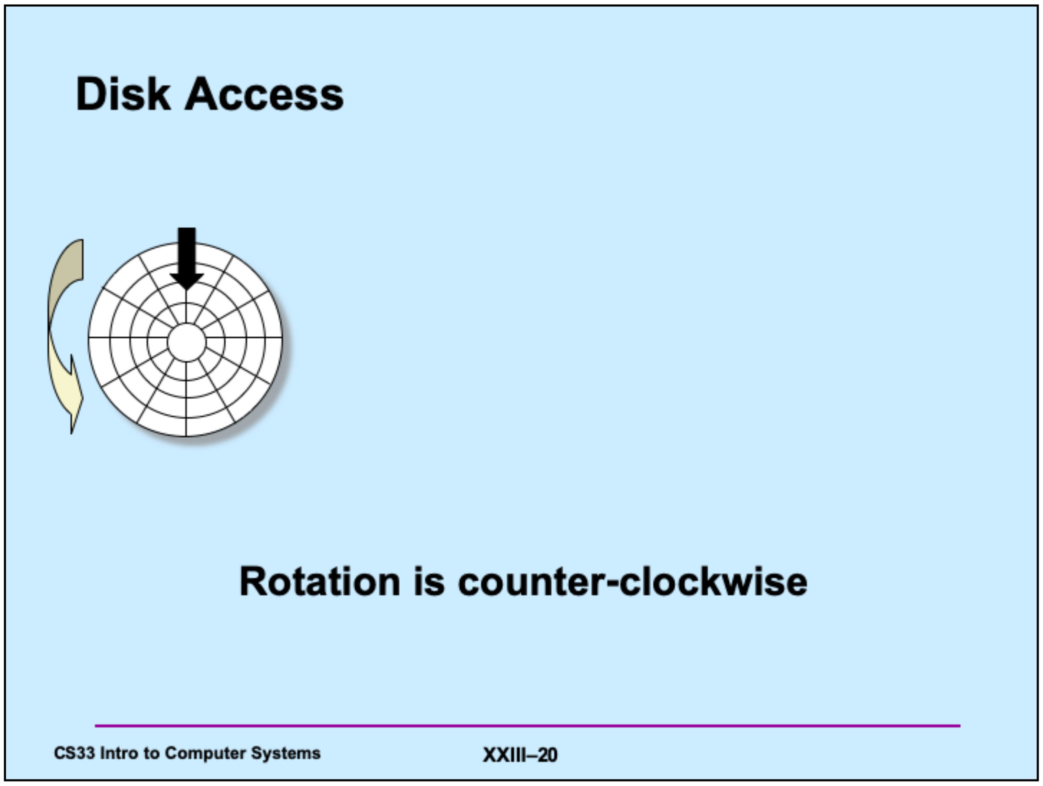


Disk Access

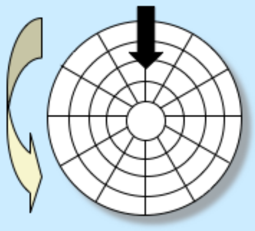


Head in position above a track

Supplied by CMU.

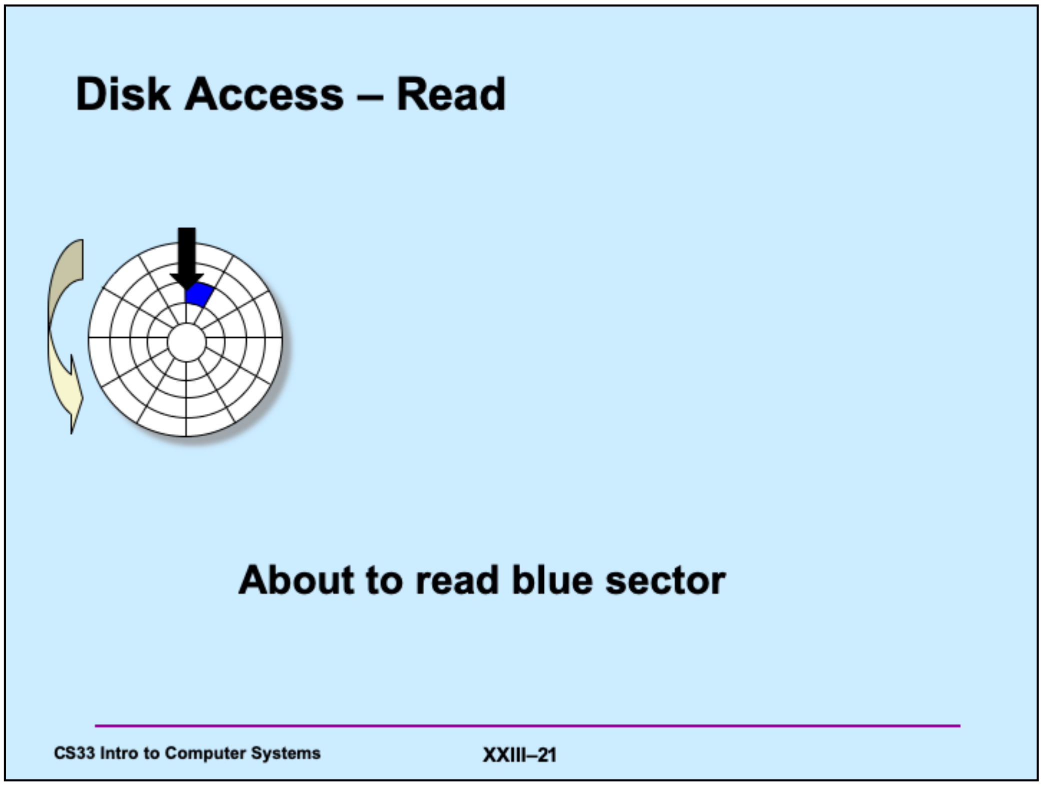


Disk Access

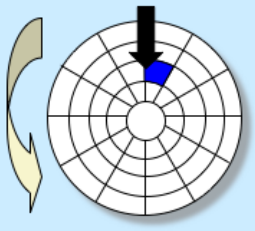


Rotation is counter-clockwise

Supplied by CMU.

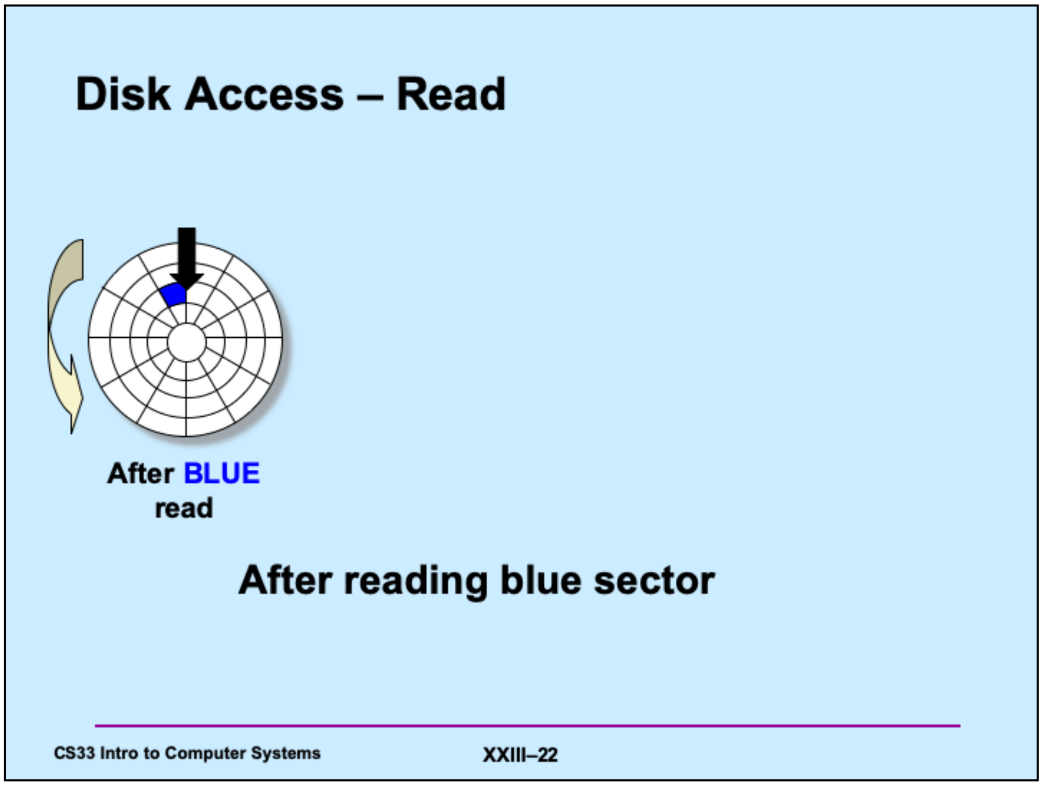


Disk Access – Read

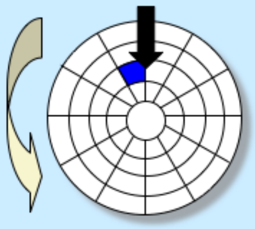


About to read blue sector

Supplied by CMU.

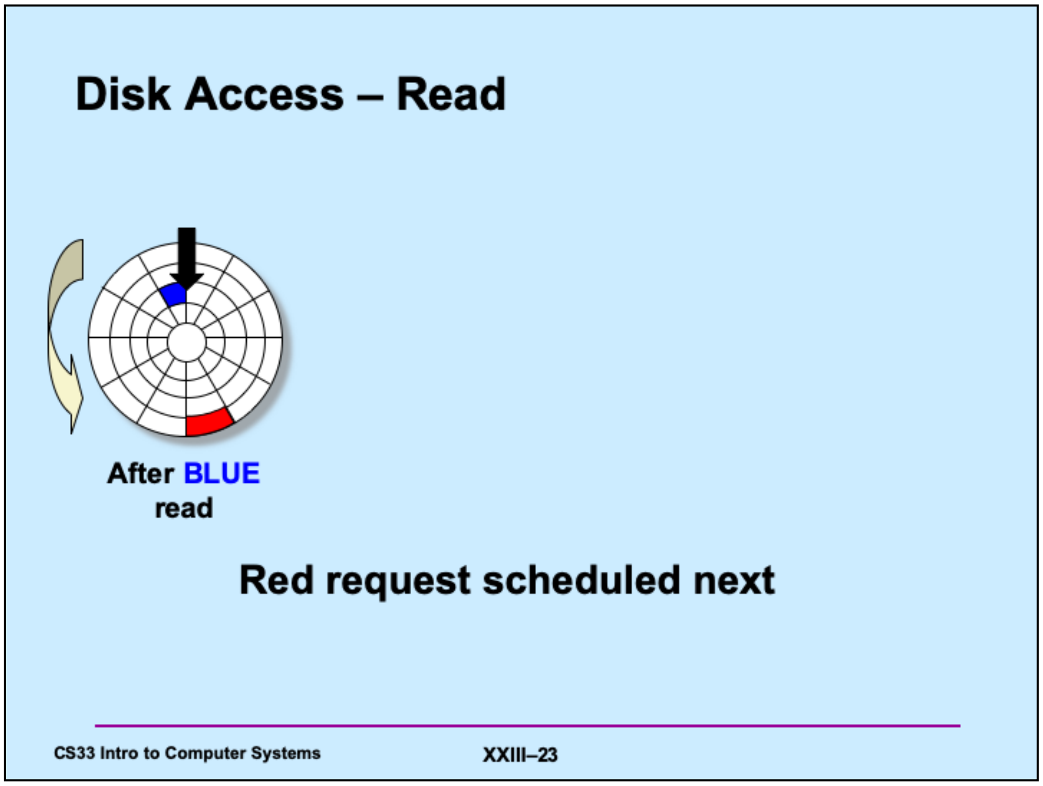


Disk Access – Read

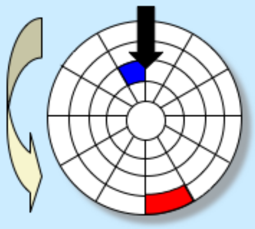


After reading blue sector

Supplied by CMU.



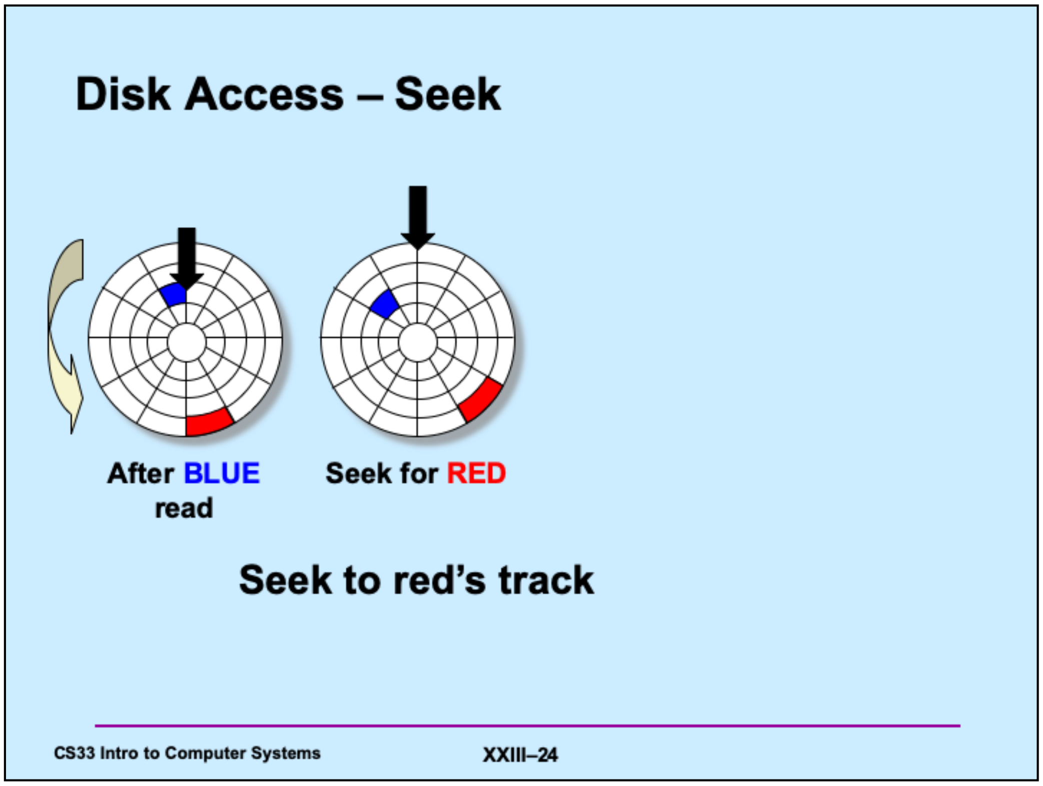
Disk Access – Read



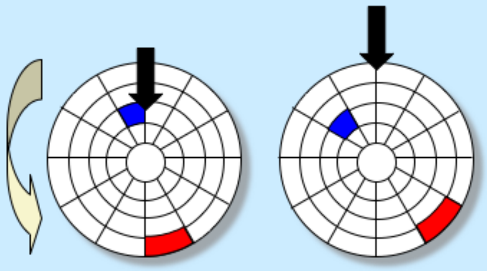
After **BLUE** read

Red request scheduled next

Supplied by CMU.

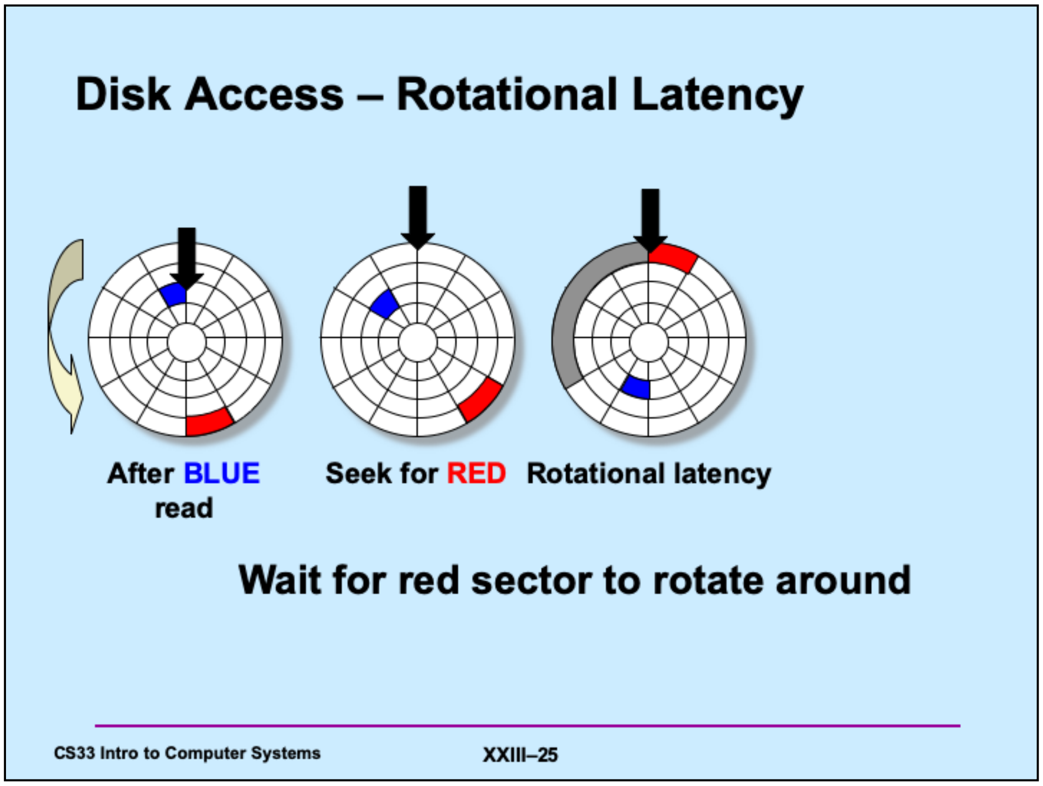


Disk Access – Seek

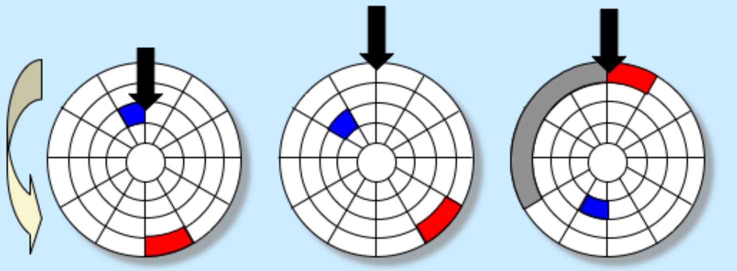


Seek to red's track

Supplied by CMU.

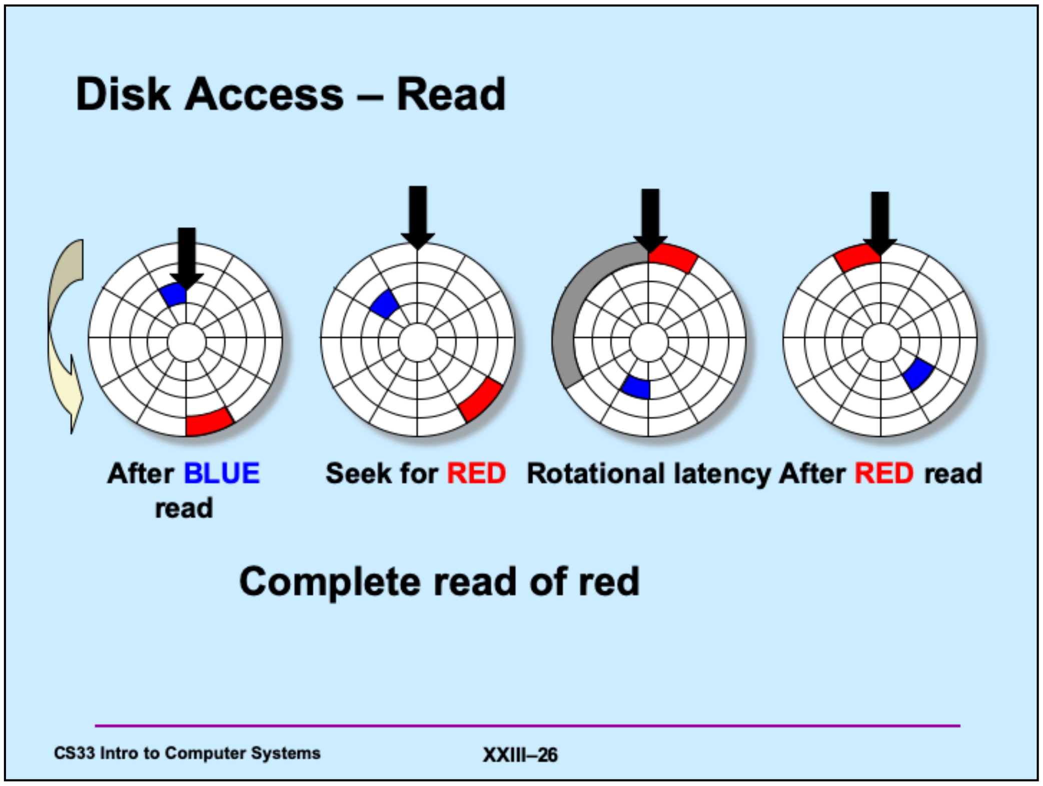


Disk Access – Rotational Latency

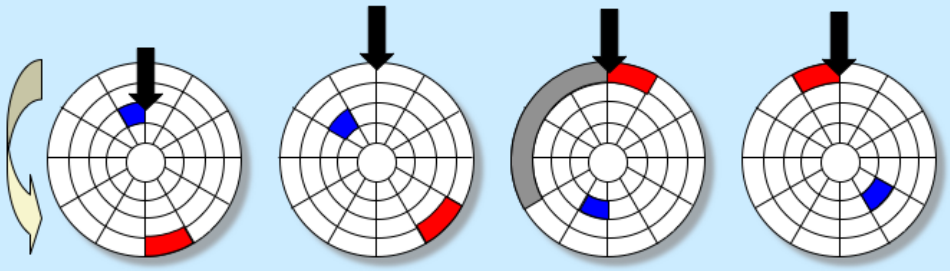


Wait for red sector to rotate around

Supplied by CMU.

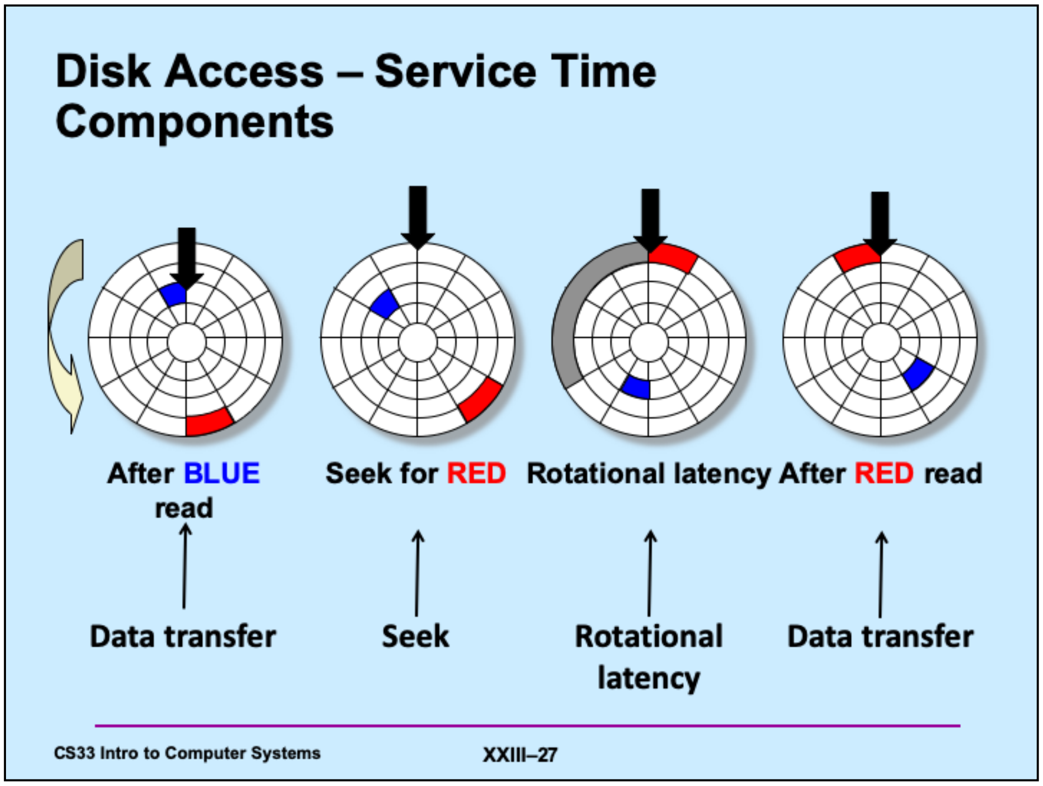


Disk Access – Read



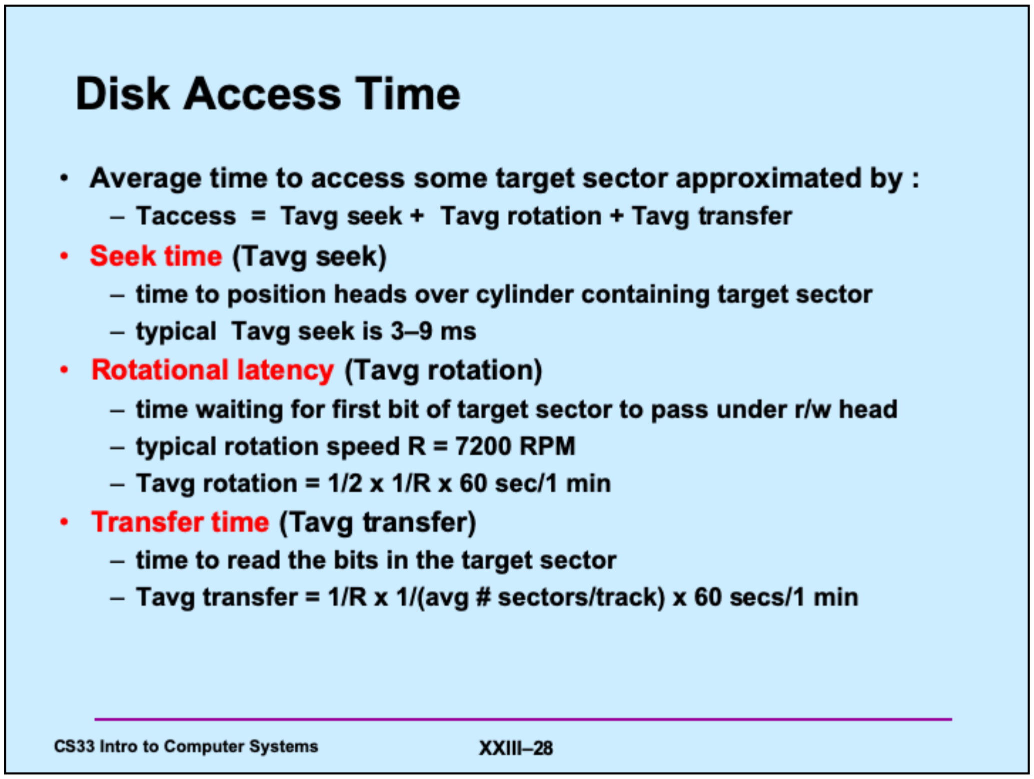
Complete read of red

Supplied by CMU.



Disk Access – Service Time Components

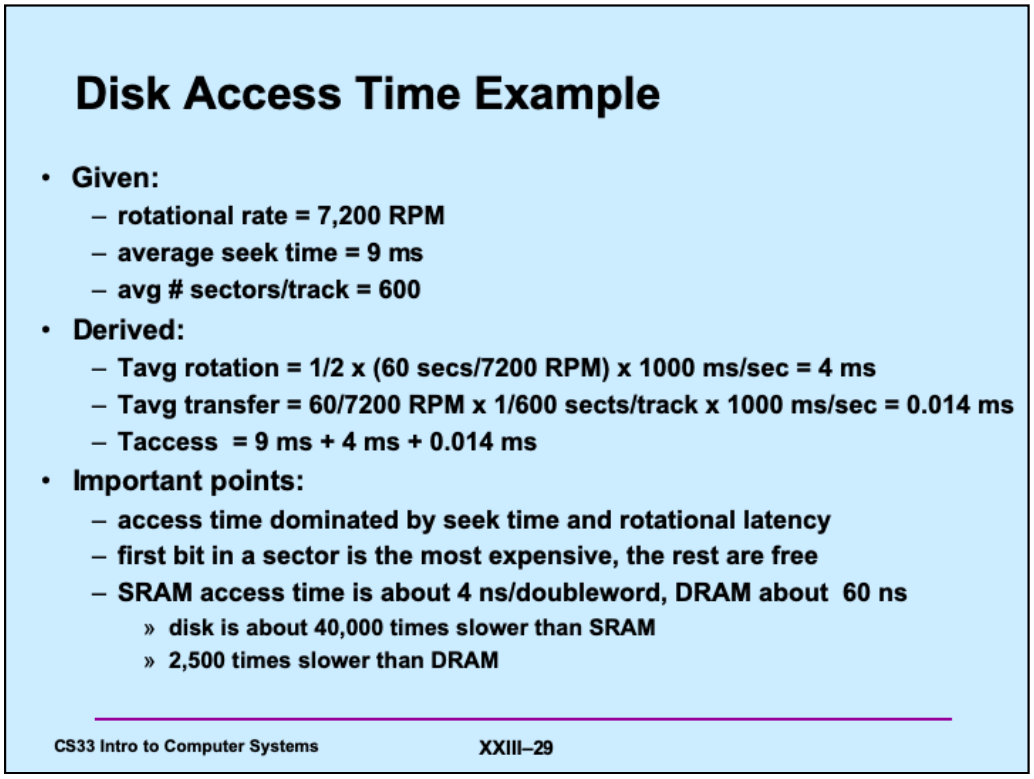
Supplied by CMU.



Disk Access Time

- Average time to access some target sector approximated by :
  - Taccess = Tavg seek + Tavg rotation + Tavg transfer
- Seek time (Tavg seek)
  - time to position heads over cylinder containing target sector
  - typical Tavg seek is 3–9 ms
- Rotational latency (Tavg rotation)
  - time waiting for first bit of target sector to pass under r/w head
  - typical rotation speed R = 7200 RPM
  - Tavg rotation = 1/2 x 1/R x 60 sec/1 min
- Transfer time (Tavg transfer)
  - time to read the bits in the target sector
  - Tavg transfer = 1/R x 1/(avg # sectors/track) x 60 secs/1 min

Supplied by CMU.

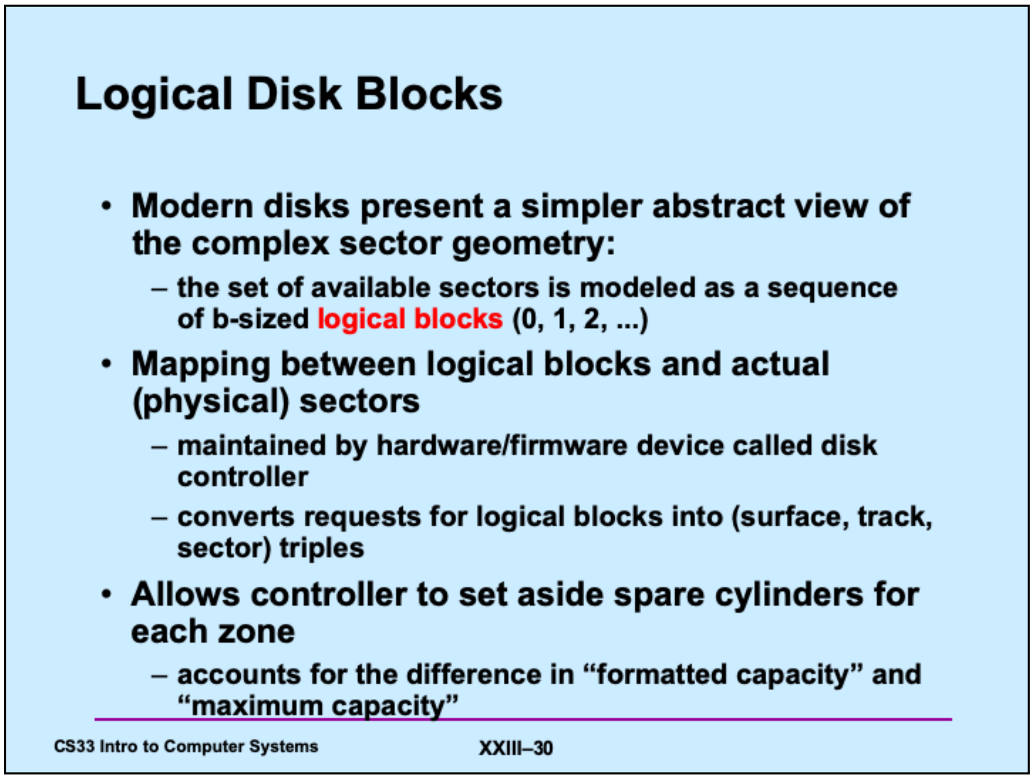


Disk Access Time Example

- Given:
  - rotational rate = 7,200 RPM
  - average seek time = 9 ms
  - avg # sectors/track = 600
- Derived:
  - Tavg rotation = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms
  - Tavg transfer = 60/7200 RPM x 1/600 sects/track x 1000 ms/sec = 0.014 ms
  - Taccess = 9 ms + 4 ms + 0.014 ms
- Important points:
  - access time dominated by seek time and rotational latency
  - first bit in a sector is the most expensive, the rest are free
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - » disk is about 40,000 times slower than SRAM
    - » 2,500 times slower than DRAM

CS33 Intro to Computer Systems XXIII–29

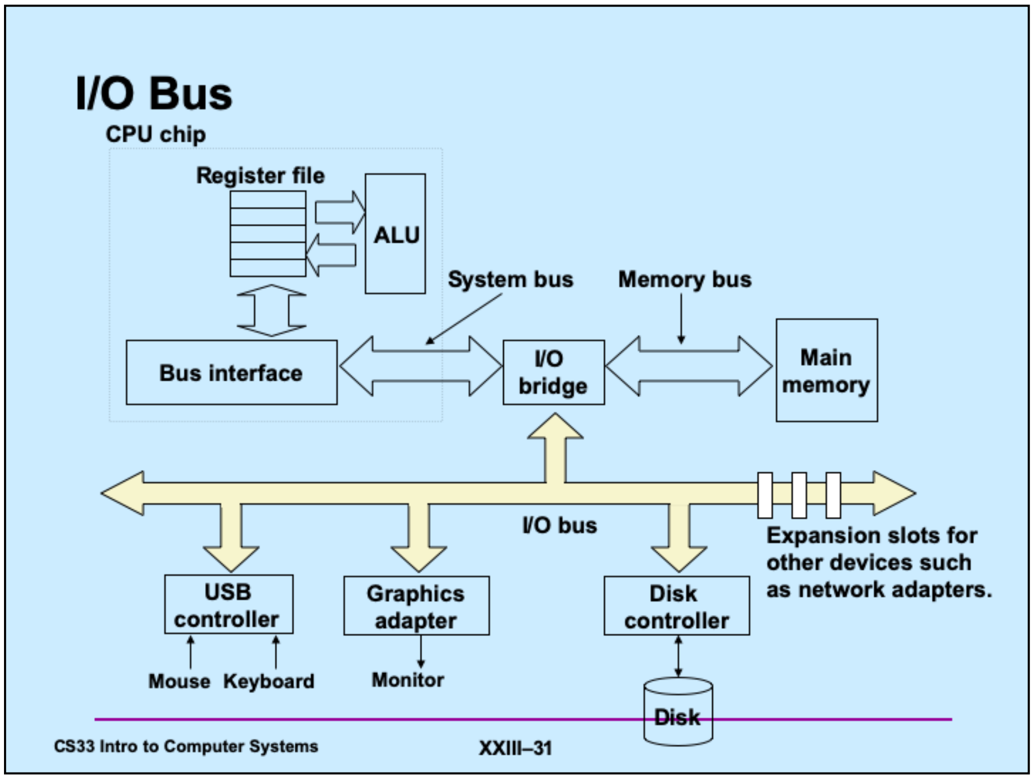
Supplied by CMU.



Logical Disk Blocks

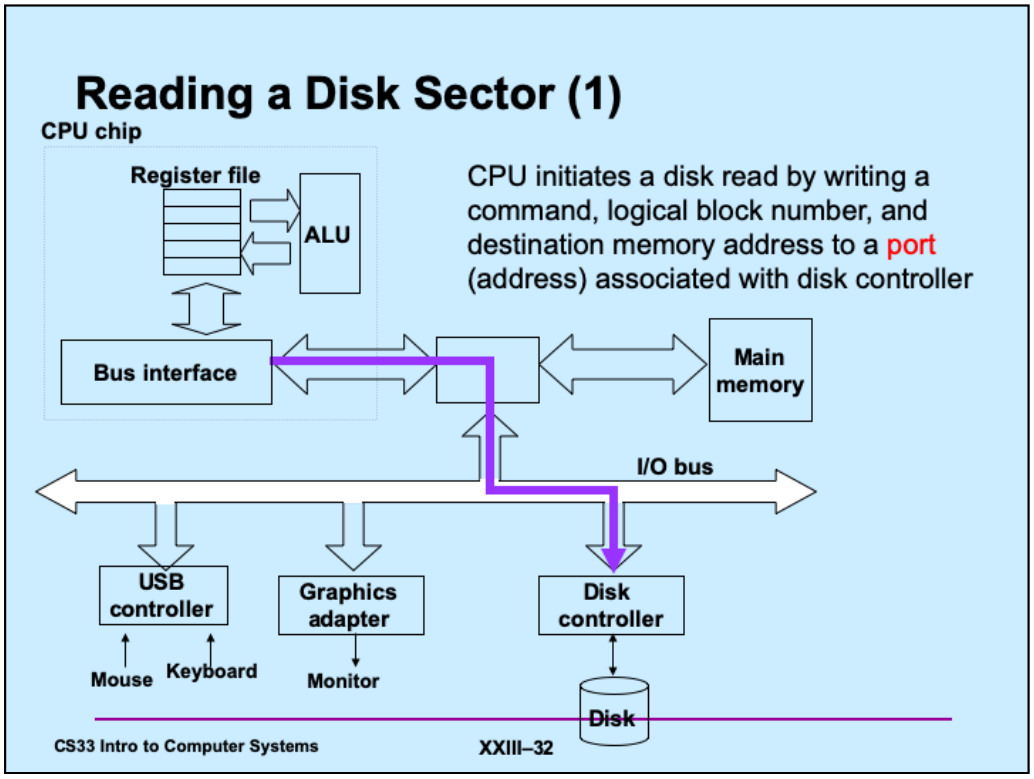
- Modern disks present a simpler abstract view of the complex sector geometry:
  - the set of available sectors is modeled as a sequence of b-sized **logical blocks** (0, 1, 2, ...)
- Mapping between logical blocks and actual (physical) sectors
  - maintained by hardware/firmware device called disk controller
  - converts requests for logical blocks into (surface, track, sector) triples
- Allows controller to set aside spare cylinders for each zone
  - accounts for the difference in "formatted capacity" and "maximum capacity"

Supplied by CMU.



I/O Bus

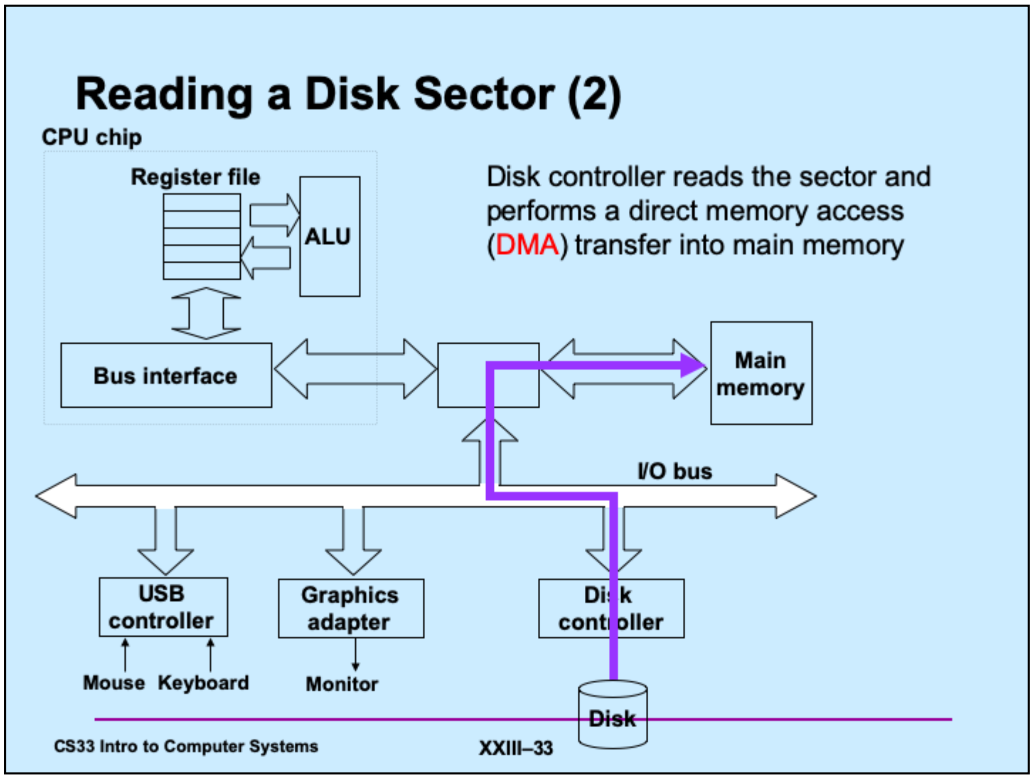
Supplied by CMU.



Reading a Disk Sector (1)

CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller

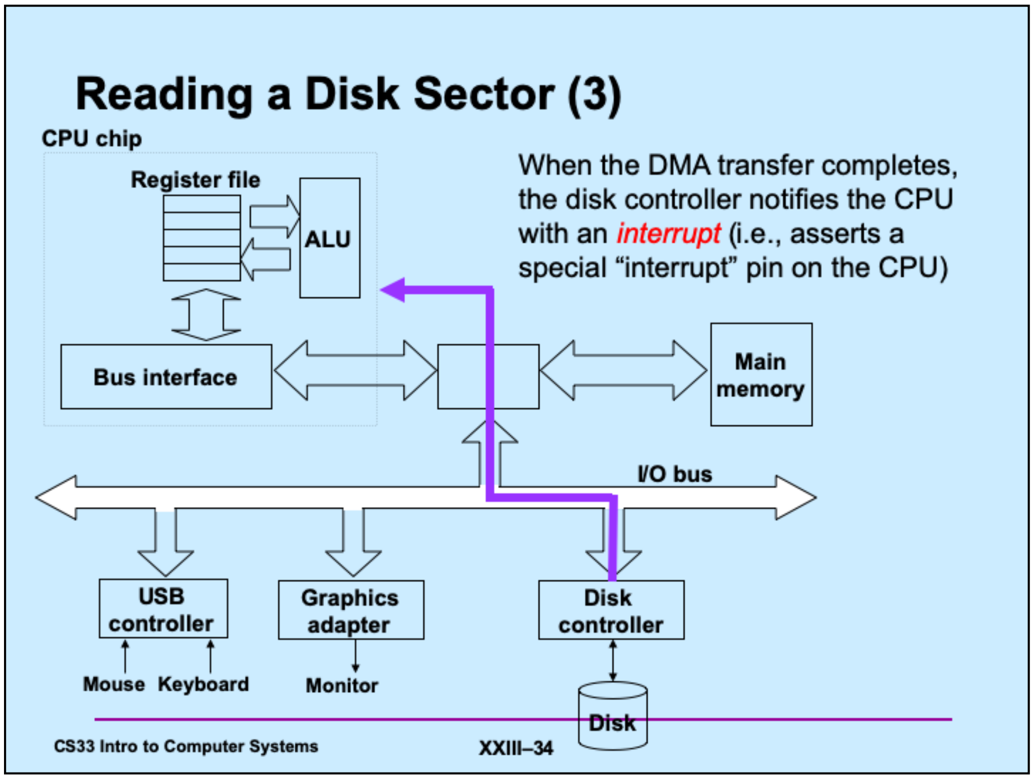
Supplied by CMU.



Reading a Disk Sector (2)

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory

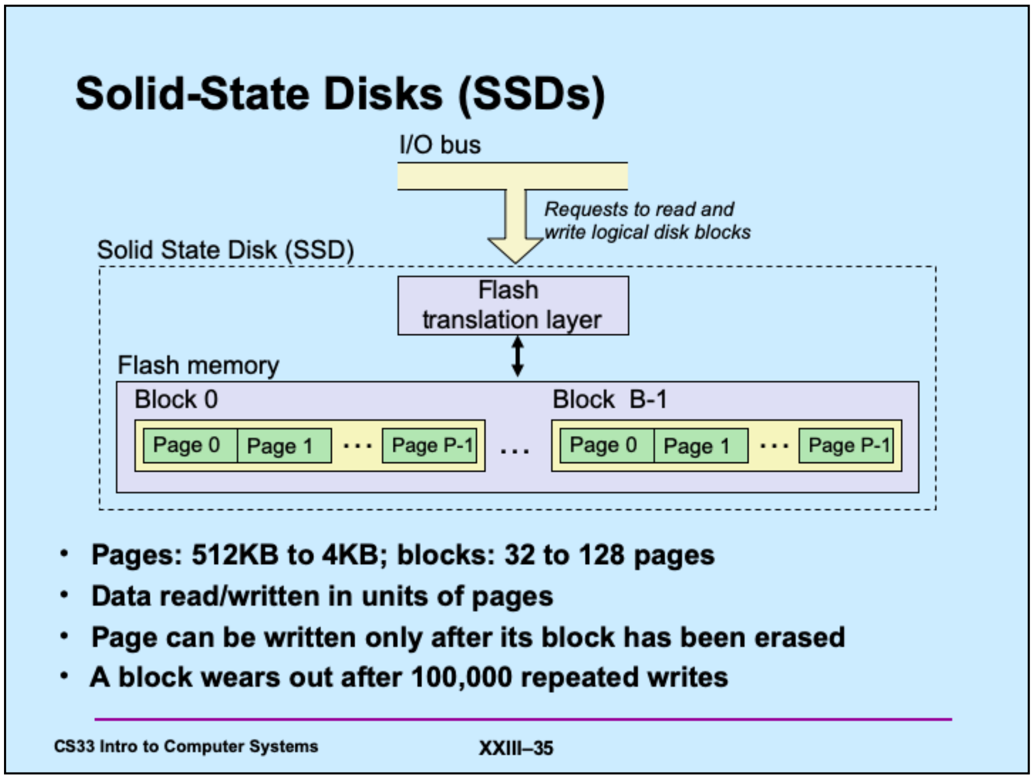
Supplied by CMU.



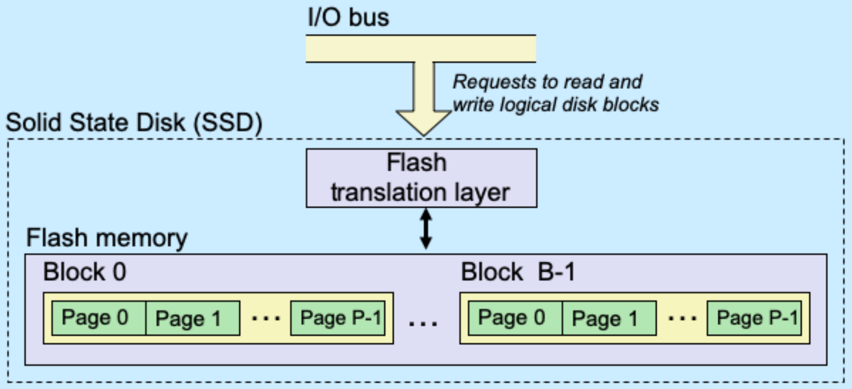
Reading a Disk Sector (3)

When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special "interrupt" pin on the CPU)

Supplied by CMU.

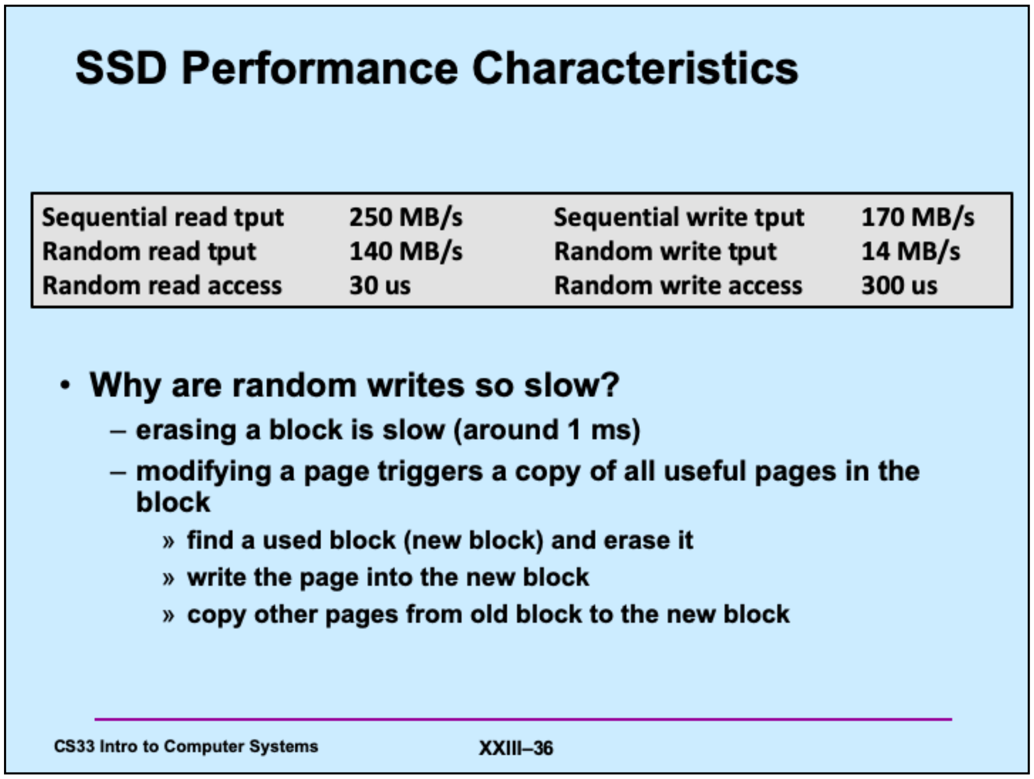


Solid-State Disks (SSDs)



- Pages: 512KB to 4KB; blocks: 32 to 128 pages
- Data read/written in units of pages
- Page can be written only after its block has been erased
- A block wears out after 100,000 repeated writes

Supplied by CMU.

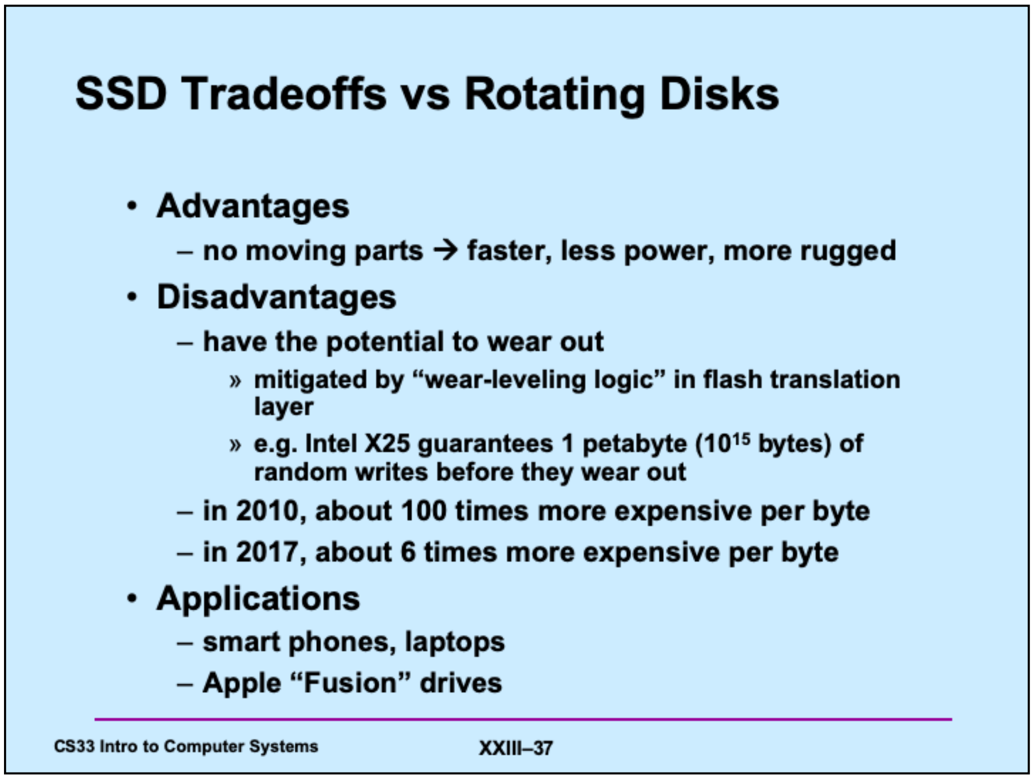


SSD Performance Characteristics

Sequential read tput	250 MB/s	Sequential write tput	170 MB/s
Random read tput	140 MB/s	Random write tput	14 MB/s
Random read access	30 us	Random write access	300 us

- Why are random writes so slow?
  - erasing a block is slow (around 1 ms)
  - modifying a page triggers a copy of all useful pages in the block
    - » find a used block (new block) and erase it
    - » write the page into the new block
    - » copy other pages from old block to the new block

Supplied by CMU.



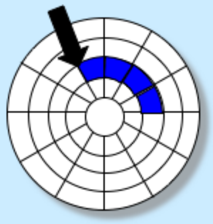
SSD Tradeoffs vs Rotating Disks

- Advantages
  - no moving parts → faster, less power, more rugged
- Disadvantages
  - have the potential to wear out
    - » mitigated by “wear-leveling logic” in flash translation layer
    - » e.g. Intel X25 guarantees 1 petabyte (10^{15} bytes) of random writes before they wear out
  - in 2010, about 100 times more expensive per byte
  - in 2017, about 6 times more expensive per byte
- Applications
  - smart phones, laptops
  - Apple “Fusion” drives

Supplied by CMU.

Reading a File on a Rotating Disk

- Suppose the data of a file are stored on consecutive disk sectors on one track
  - this is the best possible scenario for reading data quickly
    - » single seek required
    - » single rotational delay
    - » all sectors read in a single scan



Quiz 2

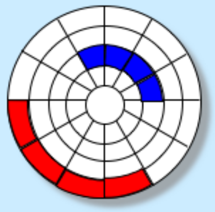
We have two files on the same (rotating) disk. The first file's data resides in consecutive sectors on one track, the second in consecutive sectors on another track. It takes a total of *t* seconds to read all of the first file then all of the second file.

Now suppose the files are read concurrently, perhaps a sector of the first, then a sector of the second, then the first, then the second, etc. Compared to reading them sequentially, this will take

a) less time

b) about the same amount of time

c) more time



Quiz 3

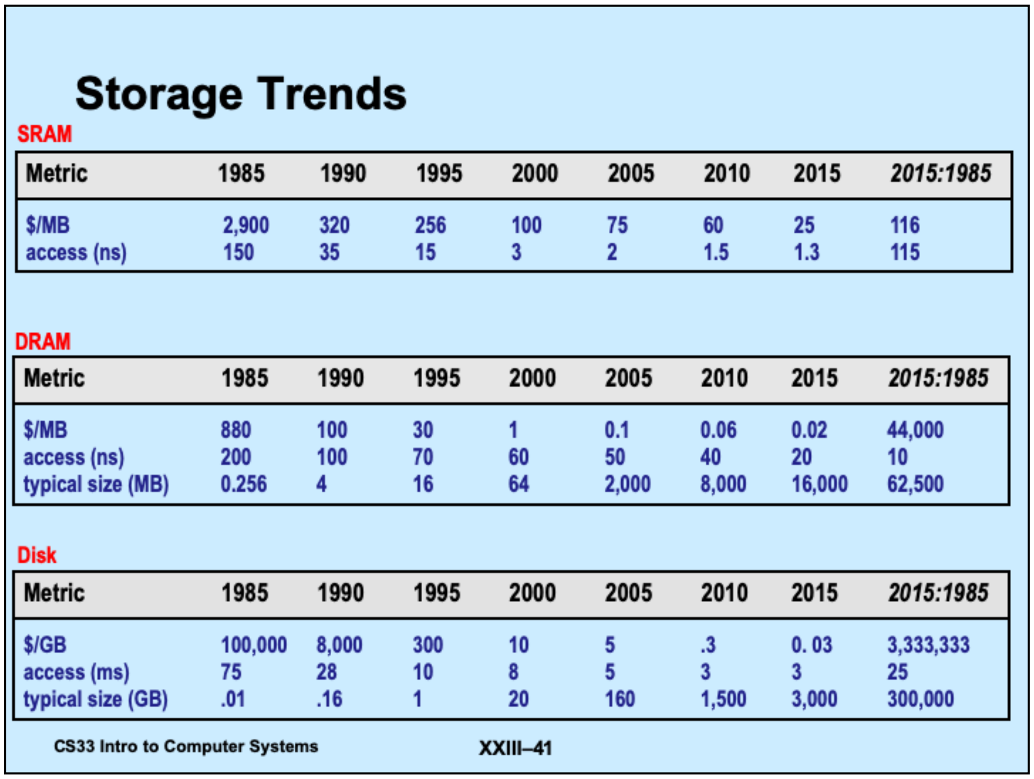
We have two files on the same solid-state disk. Each file's data resides in consecutive blocks. It takes a total of *t* seconds to read all of the first file then all of the second file.

Now suppose the files are read concurrently, perhaps a block of the first, then a block of the second, then the first, then the second, etc. Compared to reading them sequentially, this will take

a) less time

b) about the same amount of time

c) more time



Storage Trends

SRAM

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/MB	2,900	320	256	100	75	60	25	116
access (ns)	150	35	15	3	2	1.5	1.3	115

DRAM

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/MB	880	100	30	1	0.1	0.06	0.02	44,000
access (ns)	200	100	70	60	50	40	20	10
typical size (MB)	0.256	4	16	64	2,000	8,000	16,000	62,500

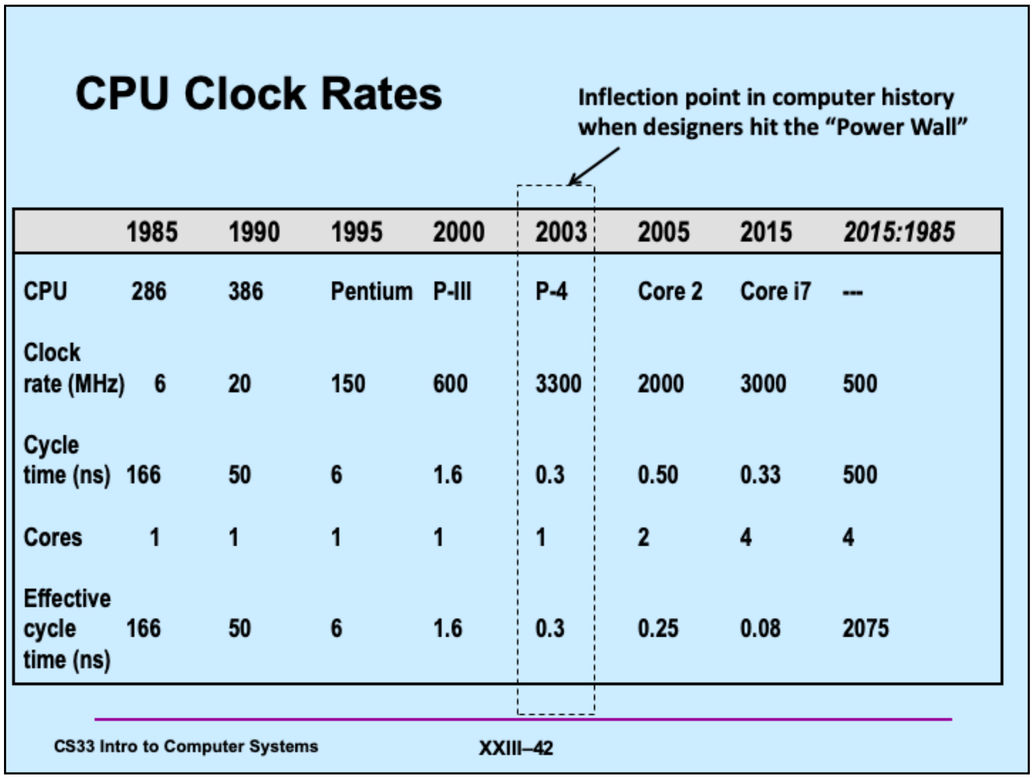
Disk

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/GB	100,000	8,000	300	10	5	.3	0. 03	3,333,333
access (ms)	75	28	10	8	5	3	3	25
typical size (GB)	.01	.16	1	20	160	1,500	3,000	300,000

Supplied by CMU.

Current (2019) prices for SRAM vary a fair amount. As of 10/11, it can be had for around $9/MB, if you buy in quantities of 1000 or more.

Current DRAM prices are as low as $.00075/MB, if bought in sufficient quantity.

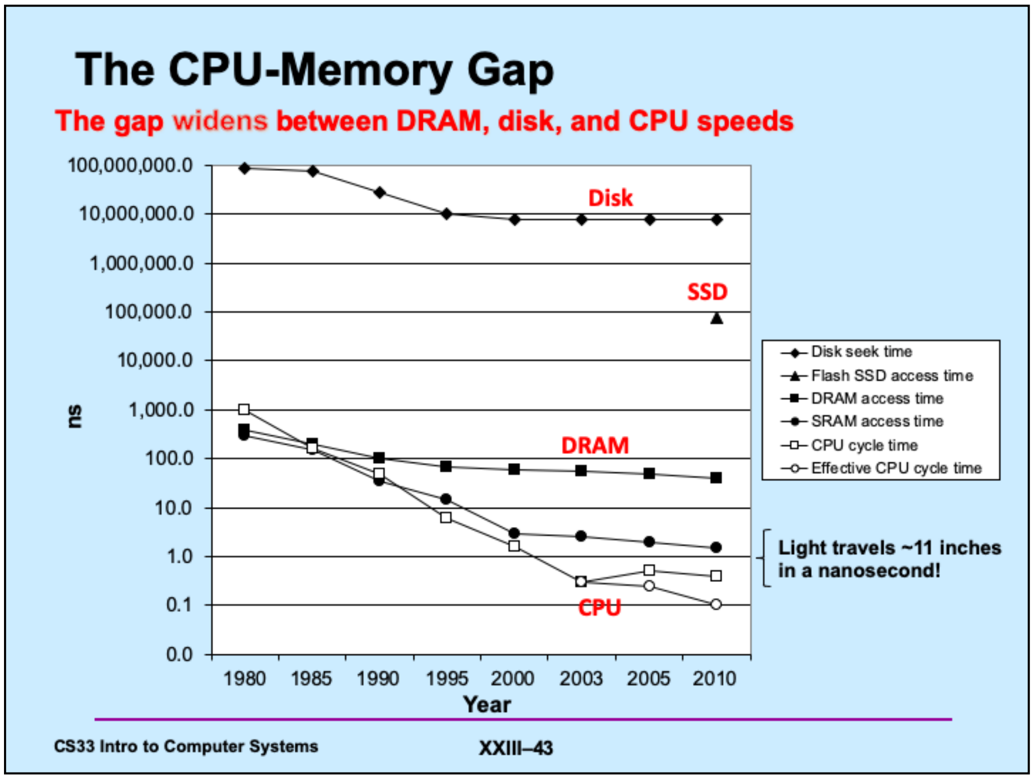


CPU Clock Rates

Inflection point in computer history when designers hit the "Power Wall"

	1985	1990	1995	2000	2003	2005	2015	*2015:1985*
CPU	286	386	Pentium	P-III	P-4	Core 2	Core i7	---
Clock rate (MHz)	6	20	150	600	3300	2000	3000	500
Cycle time (ns)	166	50	6	1.6	0.3	0.50	0.33	500
Cores	1	1	1	1	1	2	4	4
Effective cycle time (ns)	166	50	6	1.6	0.3	0.25	0.08	2075

Supplied by CMU.

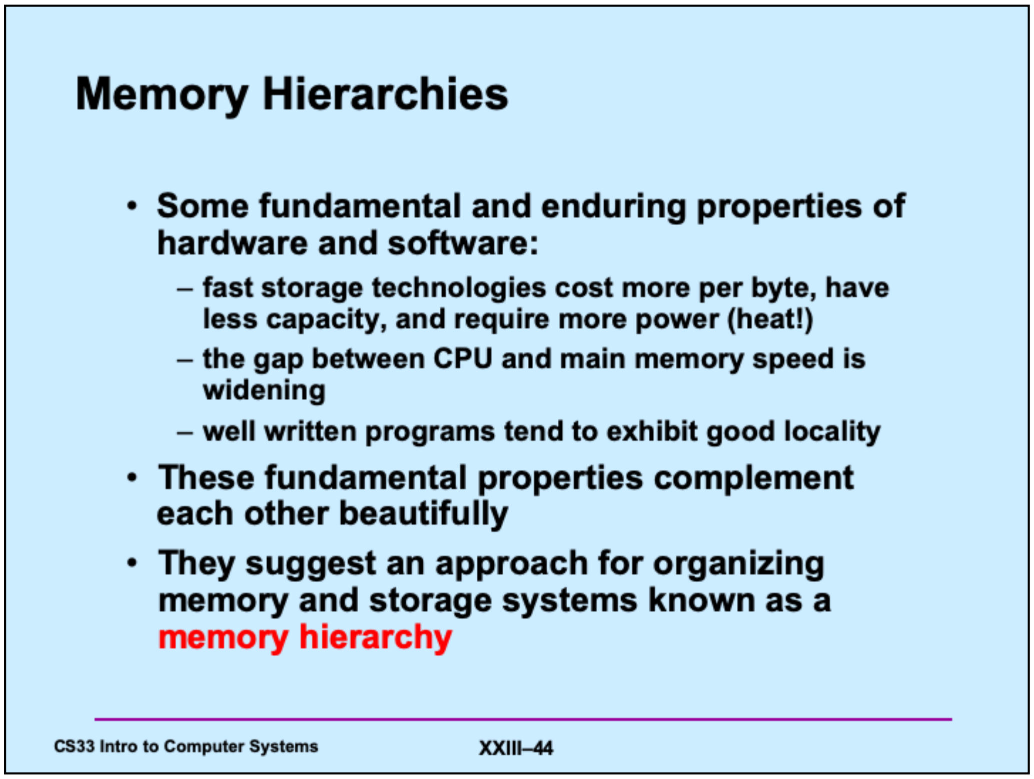


The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds

CS33 Intro to Computer Systems XXIII–43

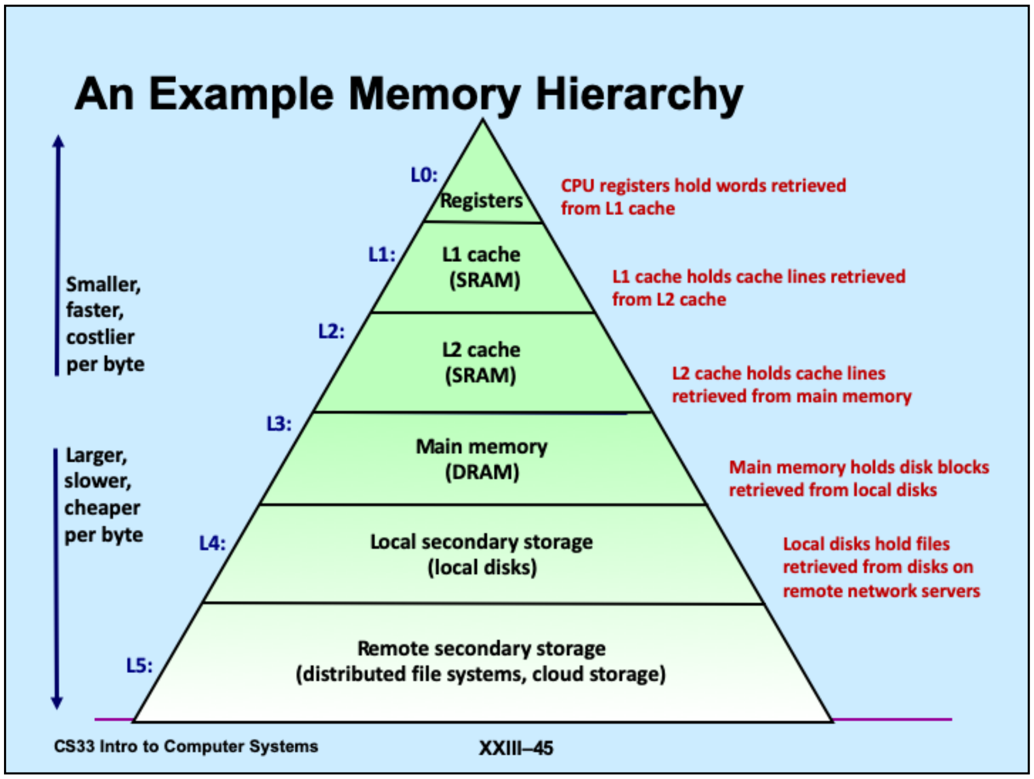
Supplied by CMU.



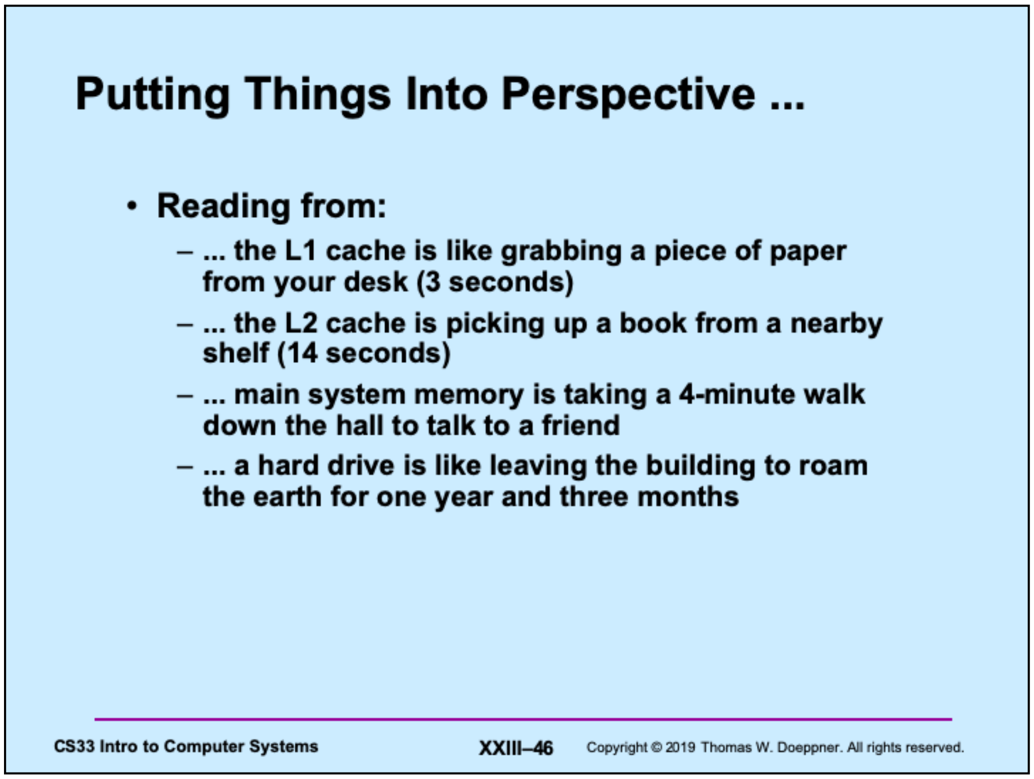
Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - fast storage technologies cost more per byte, have less capacity, and require more power (heat!)
  - the gap between CPU and main memory speed is widening
  - well written programs tend to exhibit good locality
- These fundamental properties complement each other beautifully
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**

Supplied by CMU.


An Example Memory Hierarchy
L0:
Registers
CPU registers hold words retrieved from L1 cache
L1:
L1 cache (SRAM)
L1 cache holds cache lines retrieved from L2 cache
L2:
L2 cache (SRAM)
L2 cache holds cache lines retrieved from main memory
L3:
Main memory (DRAM)
Main memory holds disk blocks retrieved from local disks
L4:
Local secondary storage (local disks)
Local disks hold files retrieved from disks on remote network servers
L5:
Remote secondary storage (distributed file systems, cloud storage)
Smaller, faster, costlier per byte
Larger, slower, cheaper per byte
CS33 Intro to Computer Systems
XXIII–45

Supplied by CMU.



Putting Things Into Perspective ...

- Reading from:
  - ... the L1 cache is like grabbing a piece of paper from your desk (3 seconds)
  - ... the L2 cache is picking up a book from a nearby shelf (14 seconds)
  - ... main system memory is taking a 4-minute walk down the hall to talk to a friend
  - ... a hard drive is like leaving the building to roam the earth for one year and three months

This analogy is from http://duartes.org/gustavo/blog/post/what-your-computer-does-while-you-wait (definitely worth reading!).

Disks Are Important

- Cheap
  - cost/byte much less than SSDs
- (fairly) Reliable
  - data written to a disk is likely to be there next year
- Sometimes fast
  - data in consecutive sectors on a track can be read quickly
- Sometimes slow
  - data in randomly scattered sectors takes a long time to read

Abstraction to the Rescue

- Programs don't deal with sectors, tracks, and cylinders
- Programs deal with *files*
  - maze.c rather than an ordered collection of sectors
  - OS provides the implementation

Implementation Problems

- Speed
  - use the hierarchy
    - » copy files into RAM, copy back when done
  - optimize layout
    - » put sectors of a file in consecutive locations
  - use parallelism
    - » spread file over multiple disks
    - » read multiple sectors at once

Implementation Problems

- **Reliability**
  - computer crashes
    - » what you thought was safely written to the file never made it to the disk — it's still in RAM, which is lost
    - » worse yet, some parts made it back to disk, some didn't
      - you don't know which is which
      - on-disk data structures might be totally trashed
  - disk crashes
    - » you had backed it up … yesterday
  - you screw up
    - » you accidentally delete the entire directory containing your strings/performance solution

Implementation Problems

- Reliability solutions
  - computer crashes
    - » transaction-oriented file systems
    - » on-disk data structures always in well defined states
  - disk crashes
    - » files stored redundantly on multiple disks
  - you screw up
    - » file system automatically keeps "snapshots" of previous versions of files