

CS 33

Signals Part 3

Signals Occurring During System Calls

- **Either**
 - wait for system call to finish, then invoke handler
- or
- **stop system call early, then invoke handler**
 - » EINTR error if nothing had been done yet
 - » return partial results if it was underway

Interrupted System Calls: Lengthy Case

```
char buf[BSIZE];
fillbuf(buf);
long remaining = BSIZE;
char *bptr = buf;
while (1){
    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 gigabyte 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 of the 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;    long_running_procedure( ) {  
                                while (a_long_time) {  
main( ) {                      update_state(&state);  
    void handler(int);        compute_more( );  
                                }  
    signal(SIGINT, handler);  }  
  
    long_running_procedure( );  
}                               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 sig) {
    void handler(int);                    ... /* deal with signal */
    signal(SIGINT, handler);              myputs("equally important "
    ... /* complicated program */         "message\n");
                                          }
    myputs("important message\n");
    ... /* more program */
}
```

In this example, both the mainline code and the signal handler call **myputs**, 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 **myputs**. Could this be a problem?

Asynchronous Signals (4)

```
char buf[BSIZE];
int pos;

void myputs(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 **myputs**, used in the previous slide. What it does is copy the input string, one character at a time, into **buf**, which is of size **BFSIZE**. 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 **myputs** 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 **myputs** 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 had been interrupted, **pos** would be soon incremented by 1). The signal handler now calls **myputs**, 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	- getpid	- readlink	- sigemptyset	- tcgetpgrp
- accept	- execl	- getsockname	- recv	- sigfillset	- tcseabreak
- 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	- fdasync	- lseek	- sem_post	- sigqueue	- timer_settime
- cfgetispeed	- fork	- lstat	- send	- sigsuspend	- times
- cfgetospeed	- fpathconf	- mkdir	- sendmsg	- sleep	- umask
- cfsetispeed	- fstat	- mkfifo	- sendto	- socketmark	- 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.

Quiz 1

**Printf is not listed as being async-signal safe.
Can it be implemented so that it is?**

- a) yes, but it would be so complicated, it's not done
- b) yes, it can be easily made async-signal safe
- c) no, it's inherently not async-signal safe

CS 33

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.

What's Inside A Disk Drive?

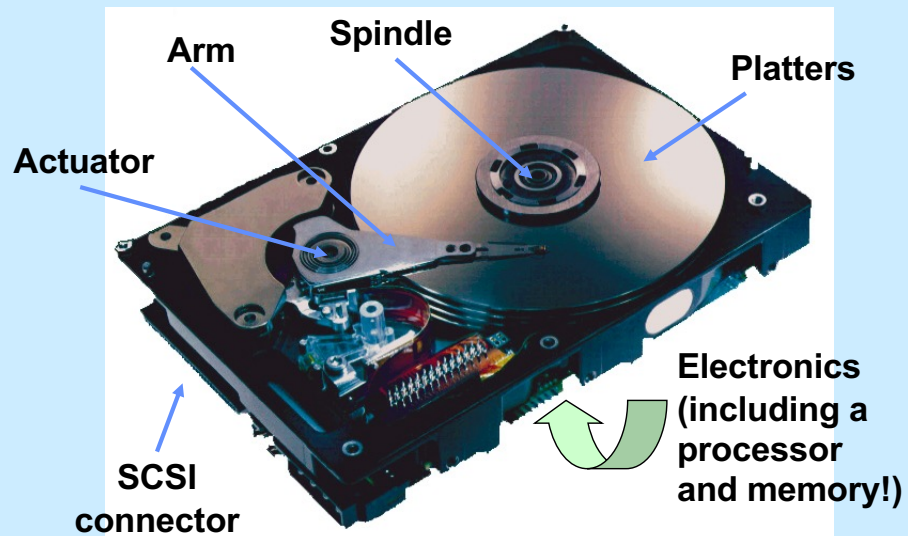
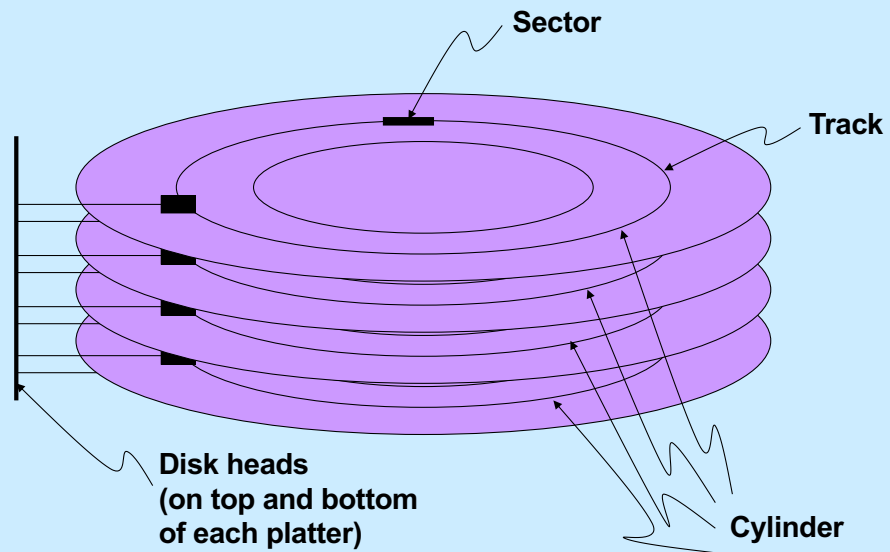


Image courtesy of Seagate Technology

Supplied by CMU.

Disk Architecture

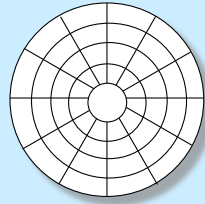


Example Disk Drive

Rotation speed	10,000 RPM
Number of surfaces	8
Sector size	512 bytes
Sectors/track	500-1000; 750 average
Tracks/surface	100,000
Storage capacity	307.2 billion bytes
Average seek time	4 milliseconds
One-track seek time	.2 milliseconds
Maximum seek time	10 milliseconds

The slide lists the characteristics of a hypothetical disk drive.

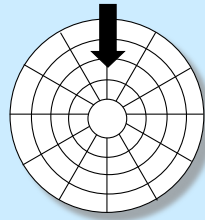
Disk Structure: Top View of Single Platter



Surface organized into tracks

Tracks divided into sectors

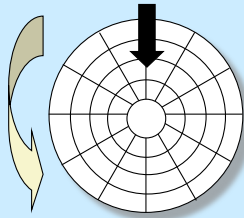
Disk Access



Head in position above a track

Supplied by CMU.

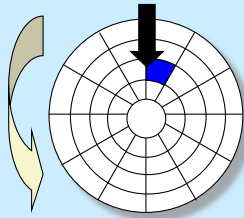
Disk Access



Rotation is counter-clockwise

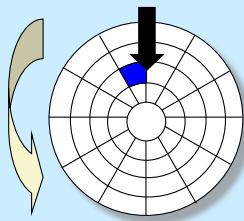
Supplied by CMU.

Disk Access – Read



About to read blue sector

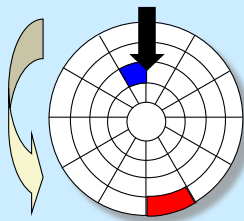
Disk Access – Read



After **BLUE**
read

After reading blue sector

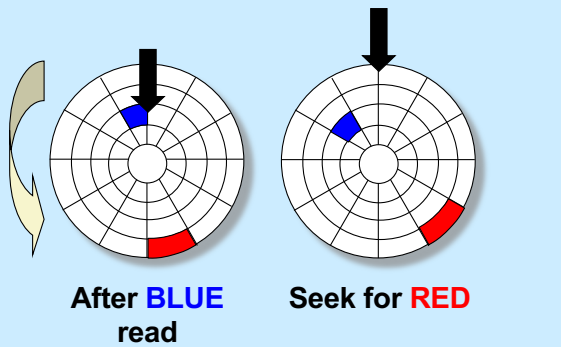
Disk Access – Read



After **BLUE**
read

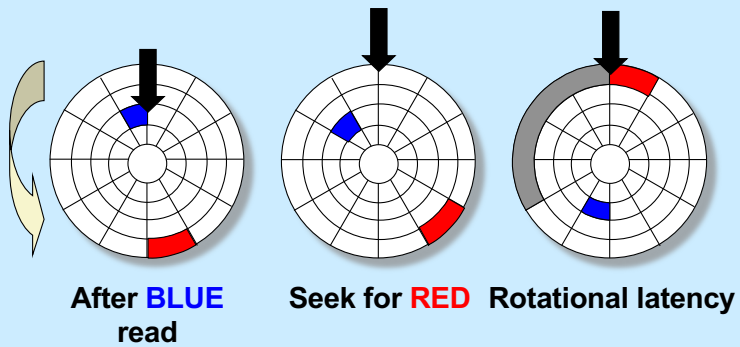
Red request scheduled next

Disk Access – Seek

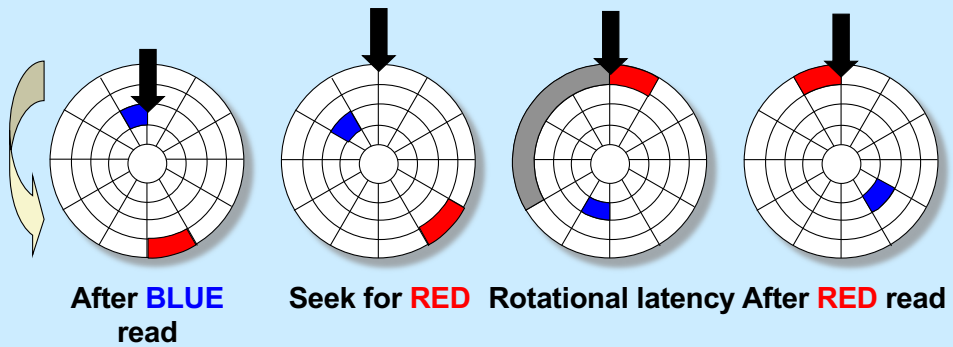


Seek to red's track

Disk Access – Rotational Latency

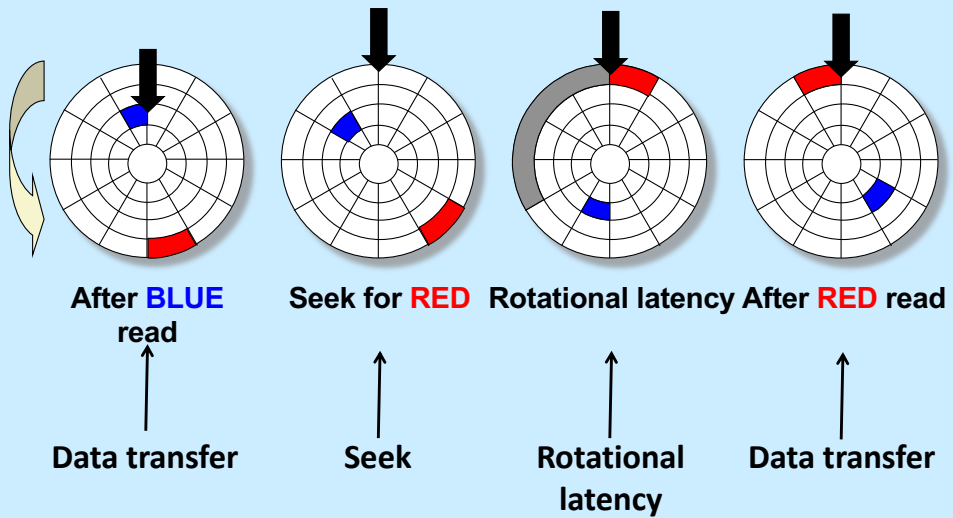


Disk Access – Read



Complete read of red

Disk Access – Service Time Components



Supplied by CMU.

Disk Access Time

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

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 = $1/2 \times (60 \text{ secs}/7200 \text{ RPM}) \times 1000 \text{ ms/sec} = 4 \text{ ms}$
 - Tavg transfer = $60/7200 \text{ RPM} \times 1/600 \text{ sects/track} \times 1000 \text{ ms/sec} = 0.014 \text{ ms}$
 - 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

Supplied by CMU.

Quiz 2

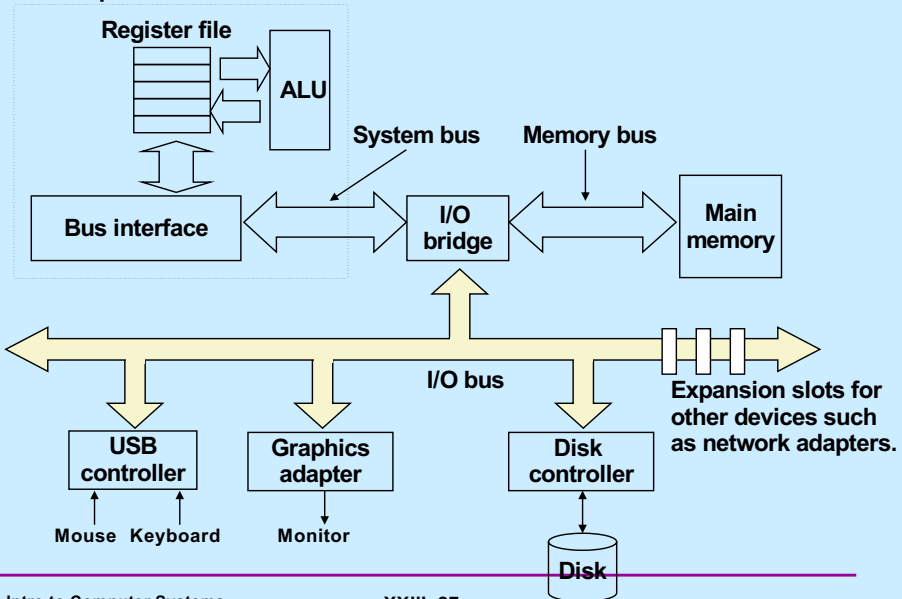
Assuming a 5-inch diameter disk spinning at 10,000 RPM, what is the approximate speed at which the outermost track is moving?

- a) faster than a speeding bullet (i.e., supersonic)**
- b) roughly the speed of a pretty fast car (150 mph)**
- c) roughly the speed of a pretty slow car (50 mph)**
- d) roughly the speed of a world-class marathoner (13.1 mph)**

For the purposes of this quiz, you may assume π is 3 and thus the circumference of the disk is 15 inches or 1.25 feet. There are 5280 feet in a mile. Thus, if the disk were a wheel, it would have to rotate a bit over 4200 times to traverse a mile.

I/O Bus

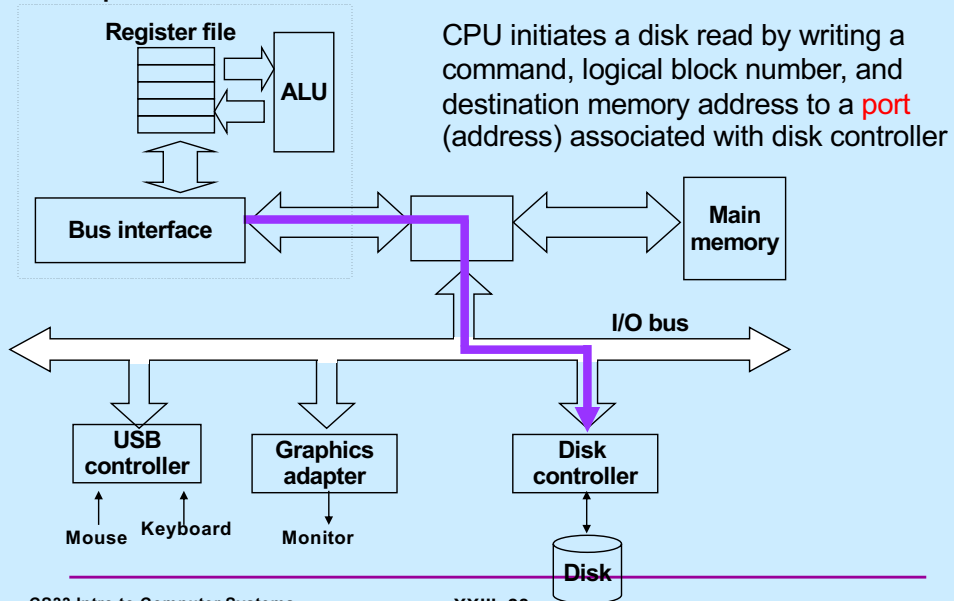
CPU chip



Supplied by CMU.

Reading a Disk Sector (1)

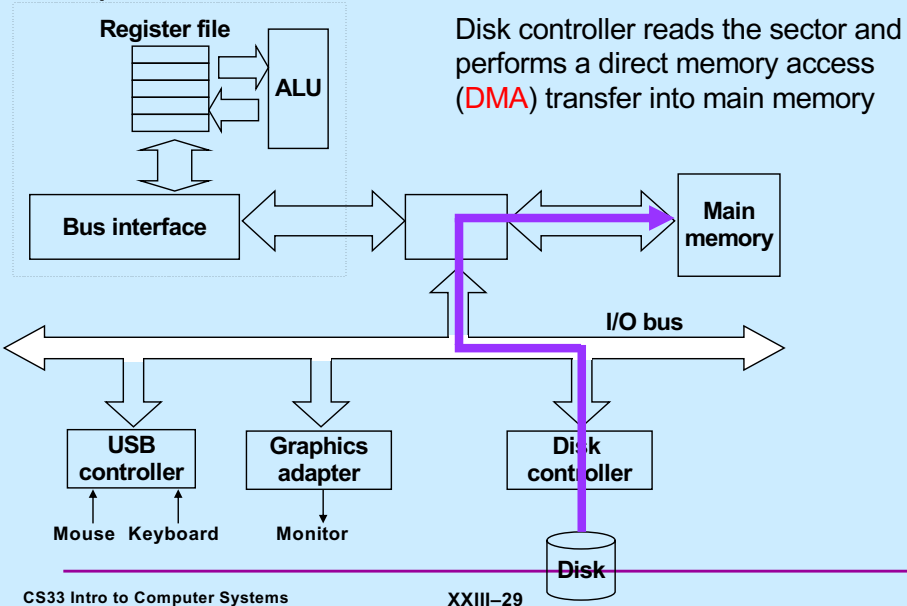
CPU chip



Supplied by CMU.

Reading a Disk Sector (2)

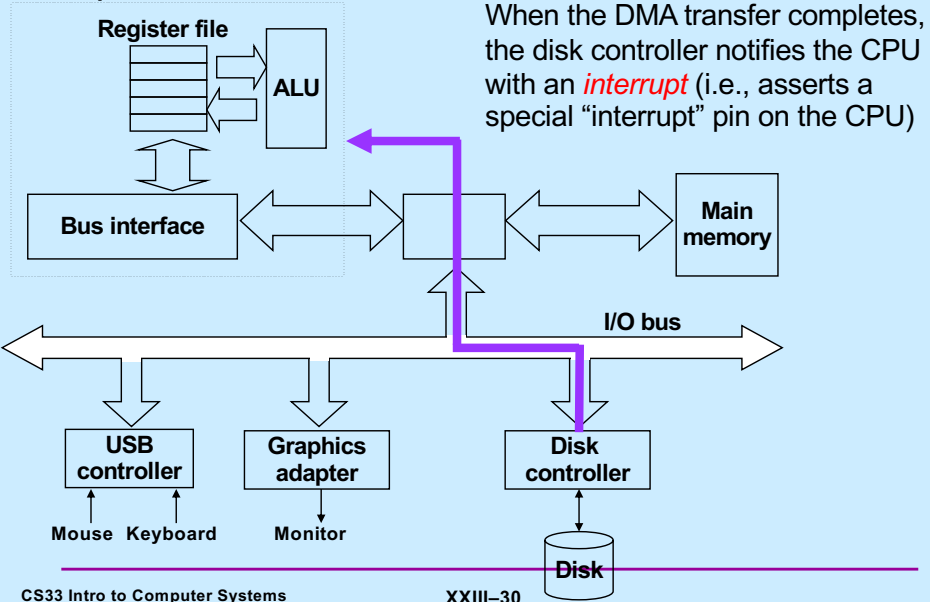
CPU chip



Supplied by CMU.

Reading a Disk Sector (3)

CPU chip

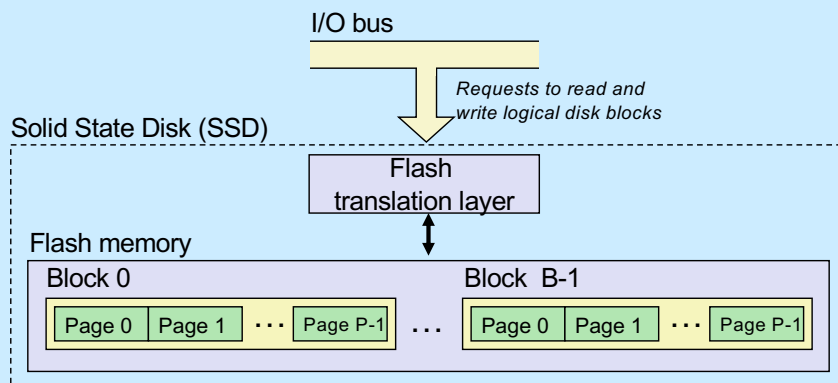


CS33 Intro to Computer Systems

XXIII-30

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

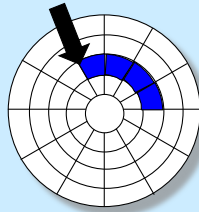
- **Advantages**
 - no moving parts → faster, less power, more rugged
- **Disadvantages**
 - have the potential to wear out
 - » mitigated by “wear-leveling logic” in flash translation layer
 - » e.g. Intel X25 guarantees 1 petabyte (10^{15} bytes) of random writes before they wear out
 - in 2010, about 100 times more expensive per byte
 - in 2017, about 6 times more expensive per byte
 - in 2021, about 2 times more expensive per byte
- **Applications**
 - smart phones, laptops, desktops

Adapted from a slide supplied by CMU.

SSDs are on their way to supplanting disks.

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

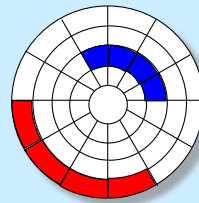


Quiz 3

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 (within a factor of 2)
- c) much more time



Quiz 4

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

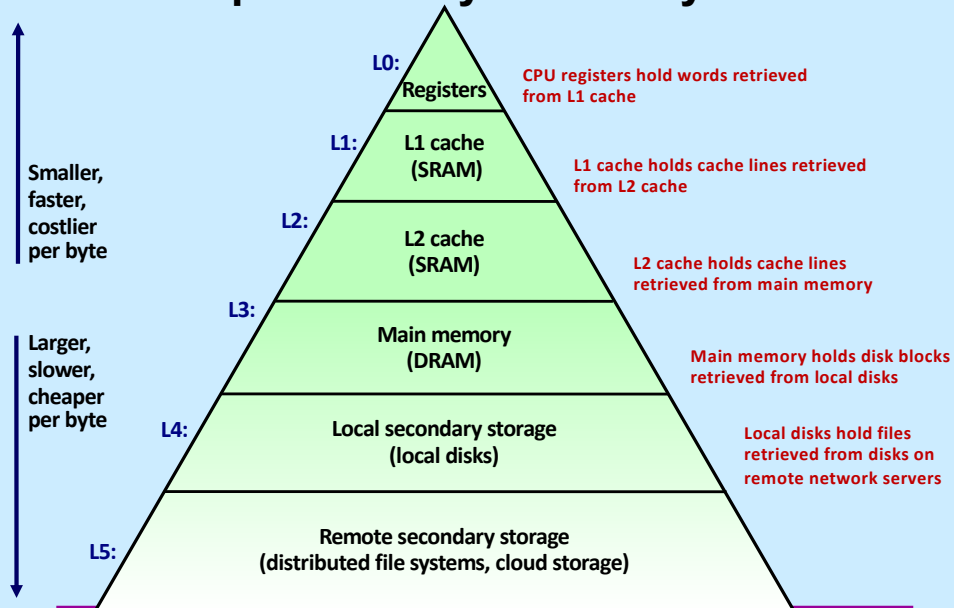
- a) less time
- b) about the same amount of time (within a factor of 2)
- c) much more time

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



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 (DRAM) 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 Still Important

- **Cheap**
 - cost/byte 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 shell 1 implementation

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