CS 33

Signals Part 3

Reaping: Zombie Elimination

- **Shell must call `waitpid` on each child**
  - easy for a foreground child
  - what about background?

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- *pid* values:

< −1	any child process whose process group is |pid|
−1	any child process
0	any child process whose process group is that of caller
> 0	child process whose ID is equal to pid

- `wait(&status)` **is equivalent to** `waitpid(-1, &status, 0)`

A process may wait only for its children to terminate (this excludes grandchildren).

(continued)

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- *options* are some combination of the following
  - WNOHANG
    - return immediately if no child has exited (returns 0)
  - WUNTRACED
    - also return if a child has been stopped (suspended)
  - WCONTINUED
    - also return if a child has been continued (resumed)

If a process is found, **waitpid** returns the process ID of the process that has been suspended or resumed.

When to Call `waitpid`

- Shell reports status only when it is about to display its prompt
  – thus sufficient to check on background jobs just before displaying prompt

`waitpid` status

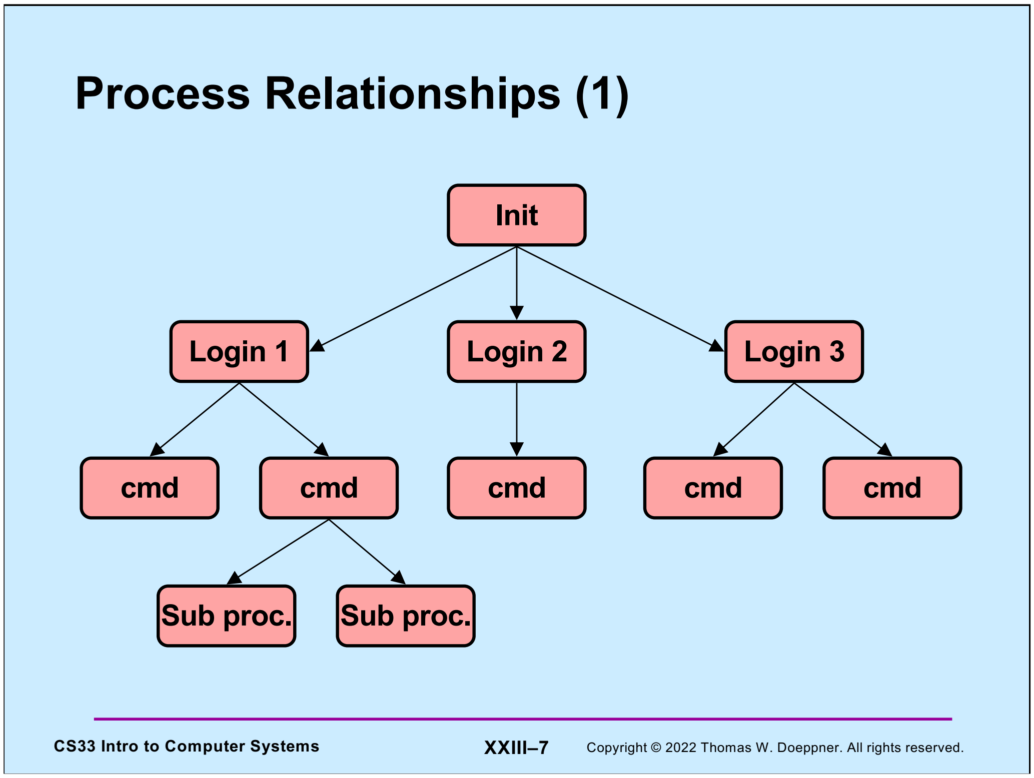
- WIFEXITED(*status): 1 if the process terminated normally and 0 otherwise
- WEXITSTATUS(*status): argument to exit
- WIFSIGNALED(*status): 1 if the process was terminated by a signal and 0 otherwise
- WTERMSIG(*status): the signal which terminated the process if it terminated by a signal
- WIFSTOPPED(*status): 1 if the process was stopped by a signal
- WSTOPSIG(*status): the signal which stopped the process if it was stopped by a signal
- WIFCONTINUED(*status): 1 if the process was resumed by SIGCONT and 0 otherwise

These are macros that can be applied to the status output argument of **waitpid**. Note that "terminated normally" means that the process terminated by calling **exit**. Otherwise, it was terminated because it received a signal, which it neither ignored nor had a handler for, whose default action was termination.

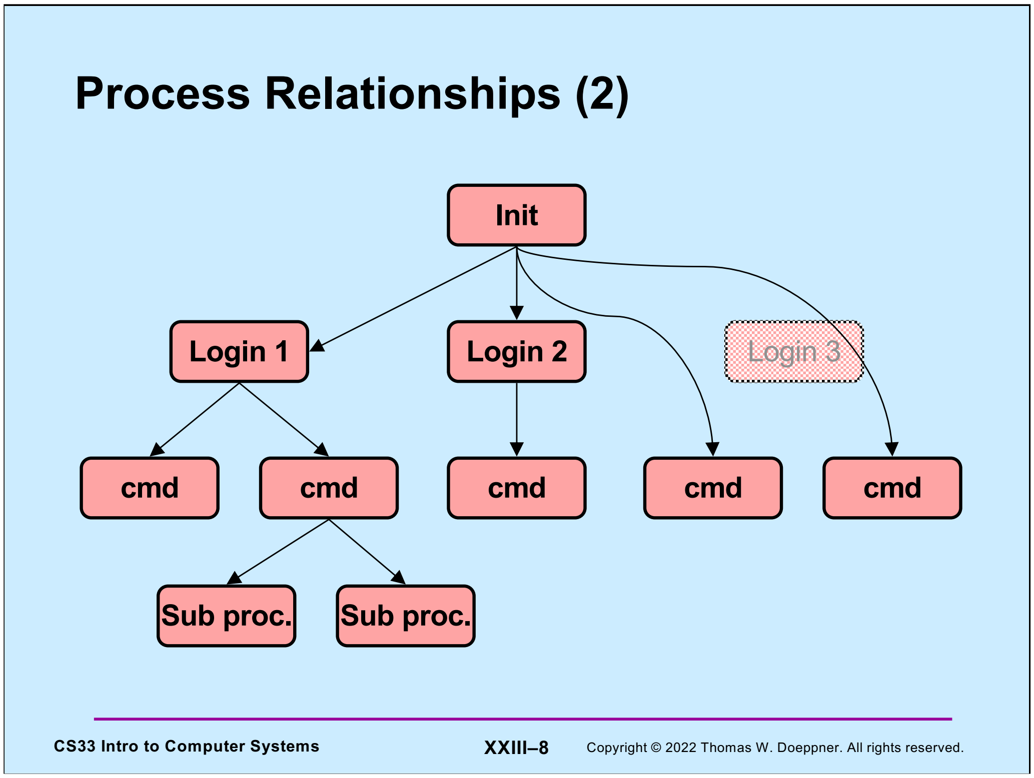
Example (in Shell)

```
int wret, wstatus;
while ((wret = waitpid(-1, &wstatus, WNOHANG|WUNTRACED)) > 0){
  // examine all children who've terminated or stopped
  if (WIFEXITED(wstatus)) {
    // terminated normally
    ...
  }
  if (WIFSIGNALED(wstatus)) {
    // terminated by a signal
    ...
  }
  if (WIFSTOPPED(wstatus)) {
    // stopped
    ...
  }
}
```


This code might be executed by a shell just before it displays its prompt. The loop iterates through all child processes that have either terminated or stopped. The WNOHANG option causes **waitpid** to return 0 (rather than waiting) if the caller has extant children, but there are no more that have either terminated or stopped. If the caller has no children, then **waitpid** returns -1.

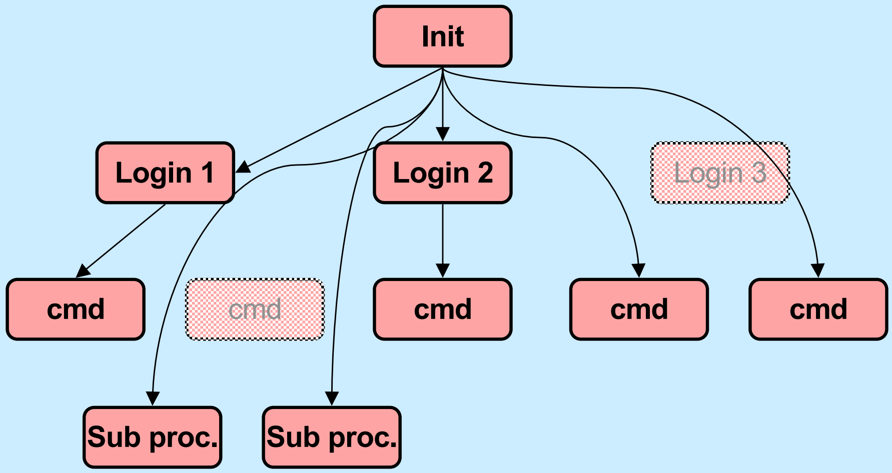


The **init** process is the common ancestor of all other processes in the system. It continues to exist while the system is running. It starts things going soon after the system is booted by forking child processes that exec the login code. These login processes then exec the shell. Note that, since only the parent may wait for a child's termination, only parent-child relationships are maintained between processes.



When a process terminates, all of its children are inherited by the **init** process, process number 1.

Process Relationships (3)



Signals, Fork, and Exec

```
// set up signal handlers ...
if (fork() == 0) {
   // what happens if child gets signal?
   ...
   signal(SIGINT, SIG_IGN);
   signal(SIGFPE, handler);
   signal(SIGQUIT, SIG_DFL);
   execv("new prog", argv, NULL);
   // what happens if SIGINT, SIGFPE,
   // or SIGQUIT occur?
}
```


As makes sense, the signal-handling state of the parent is reproduced in the child.

What also makes sense is that, if a signal has been given a handler, then, after an **exec**, since the handler no longer exists, the signal reverts to default actions.

What at first glance makes less sense is that ignored signals stay ignored after an **exec** (of course, signals with default action stay that way after the **exec**). The intent is that this allows one to run a program protected from certain signals.

Signals and System Calls

- What happens if a signal occurs while a process is doing a system call?
  - handler not invoked until just before system call returns to user
    - » system call might terminate early because of signal
  - system call completes
  - signal handler is invoked
  - user code resumed as if the system call has just returned

It's generally unsafe to interrupt the execution of a process while it's in the midst of doing a system call. Thus, if a signal is sent to a process while it's in a system call, it's usually not acted upon until just before the process returns from the system call back to the user code. At this point the handler (if any) is executed. When the handler returns, normal execution of the the user process resumes and it returns from the system call.

Signals and Lengthy System Calls

- Some system calls take a long time
  - large I/O transfer
    - » multi-gigabyte read or write request probably done as a sequence of smaller pieces
  - a long wait is required
    - » a read from the keyboard requires waiting for someone to type something
- If signal arrives in the midst of lengthy system call, handler invoked:
  - after current piece is completed
  - after cancelling wait

Some system calls take a long time to execute. Such calls might be broken up into a sequence of discrete steps, where it's safe to check for and handle signals after each step. For example, if a process is writing multiple gigabytes of data to a file in a single call to **write**, the kernel code it executes will probably break this up into a number of smaller writes, done one at a time. After each write completes, it checks to see if any unmasked signals are pending.

Interrupted System Calls

- **What if a signal is handled before the system call completes?**
  - **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 system call effectively terminated when the handler was called. When the handler returns, the system call either 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").

Summary: 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;

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 */

  myputs("important message\n");

  ...  /* more program */

}
```

```
void handler(int sig) {

  ...  /* deal with signal */

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


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

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?

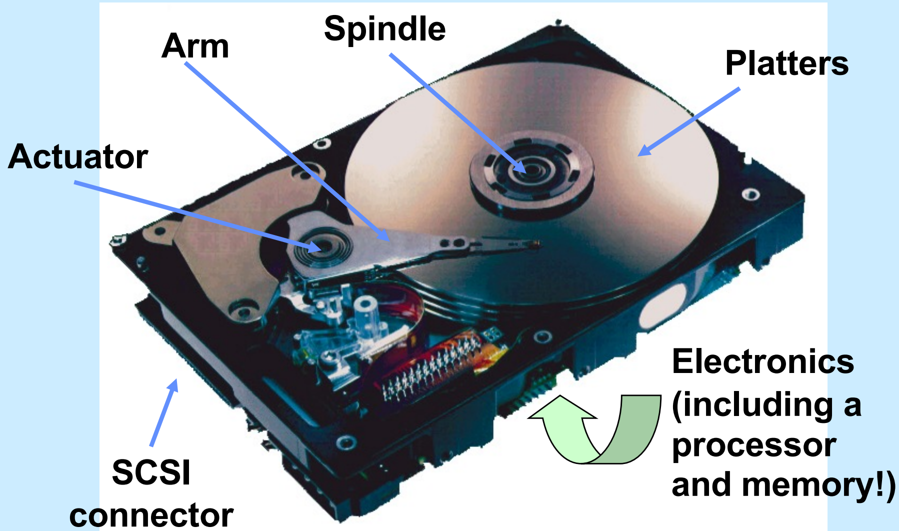
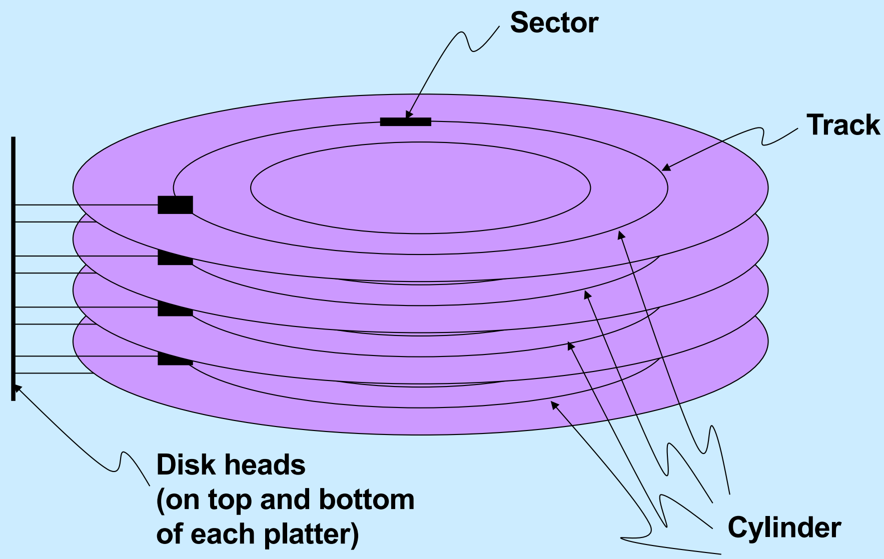


Image courtesy of Seagate Technology

Supplied by CMU.

Disk Architecture

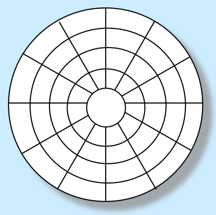


Example Disk Drive

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

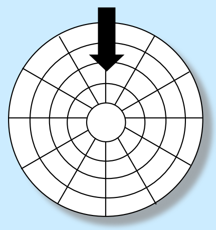


Surface organized into tracks

Tracks divided into sectors

Supplied by CMU.

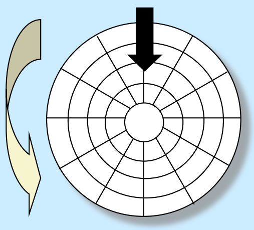
Disk Access



Head in position above a track

Supplied by CMU.

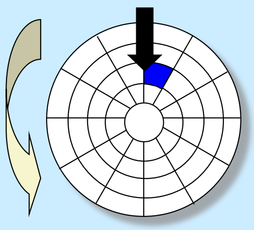
Disk Access



Rotation is counter-clockwise

Supplied by CMU.

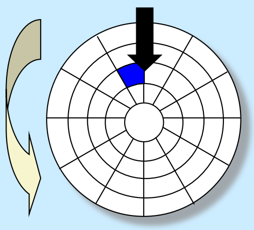
Disk Access – Read



About to read blue sector

Supplied by CMU.

Disk Access – Read

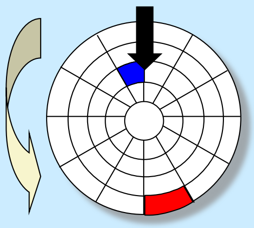


After BLUE
read

After reading blue sector

Supplied by CMU.

Disk Access – Read

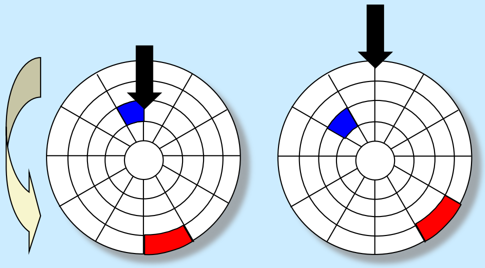


After **BLUE** read

Red request scheduled next

Supplied by CMU.

Disk Access – Seek

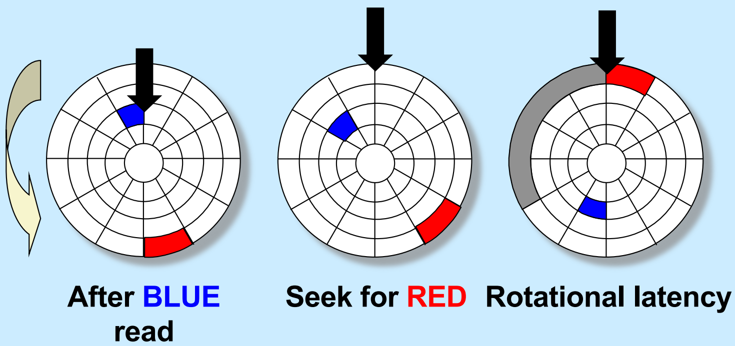


After **BLUE** read Seek for **RED**

Seek to red's track

Supplied by CMU.

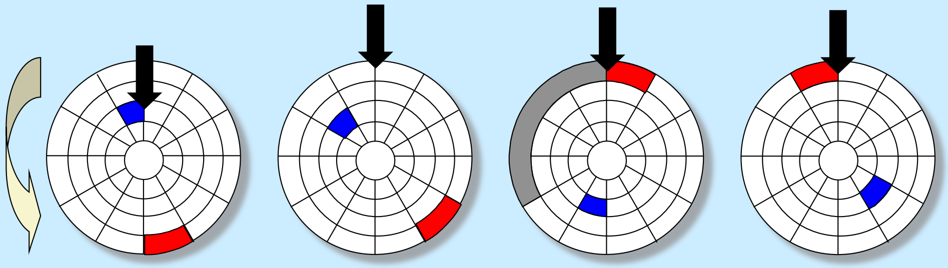
Disk Access – Rotational Latency



Wait for red sector to rotate around

Supplied by CMU.

Disk Access – Read



After BLUE read

Seek for RED

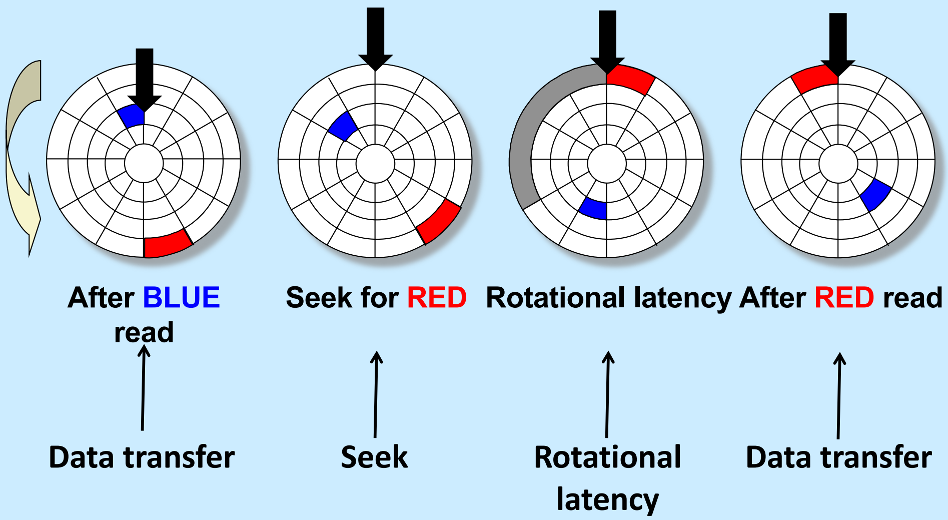
Rotational latency

After RED read

Complete read of red

Supplied by CMU.

Disk Access – Service Time Components



Supplied by CMU.

Disk Access Time

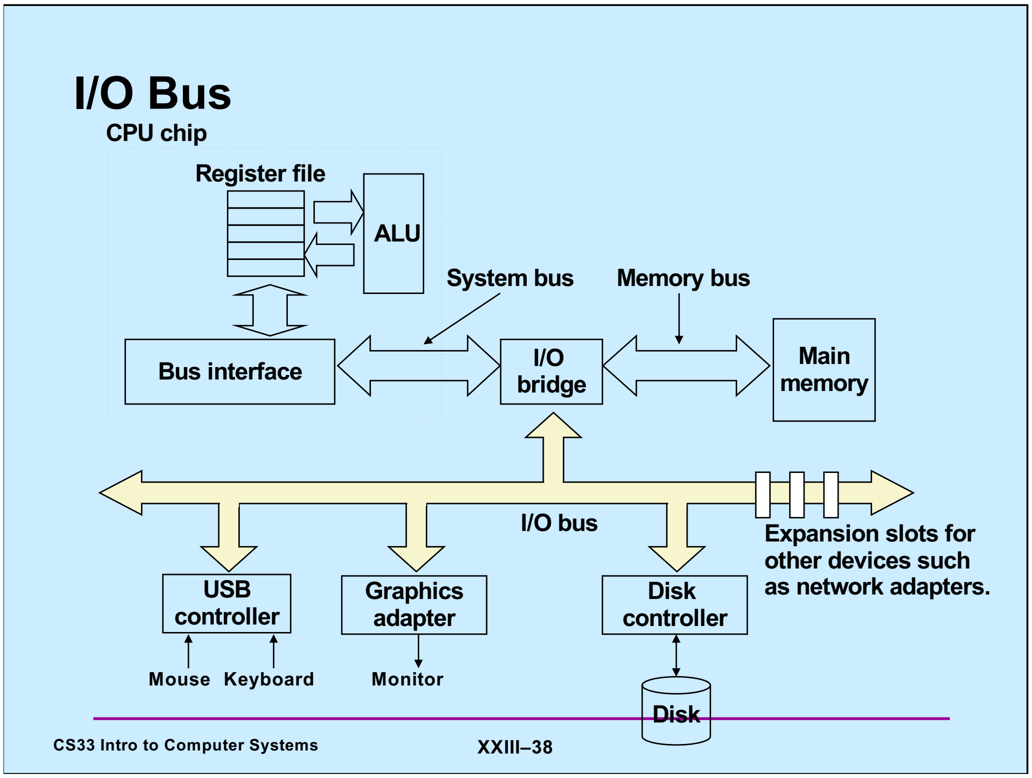
- Average time to access some target sector approximated by :
  - Taccess = Tavg seek + Tavg rotation + Tavg transfer
- Seek time (Tavg seek)
  - time to position heads over cylinder containing target sector
  - typical Tavg seek is 3–9 ms
- Rotational latency (Tavg rotation)
  - time waiting for first 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.

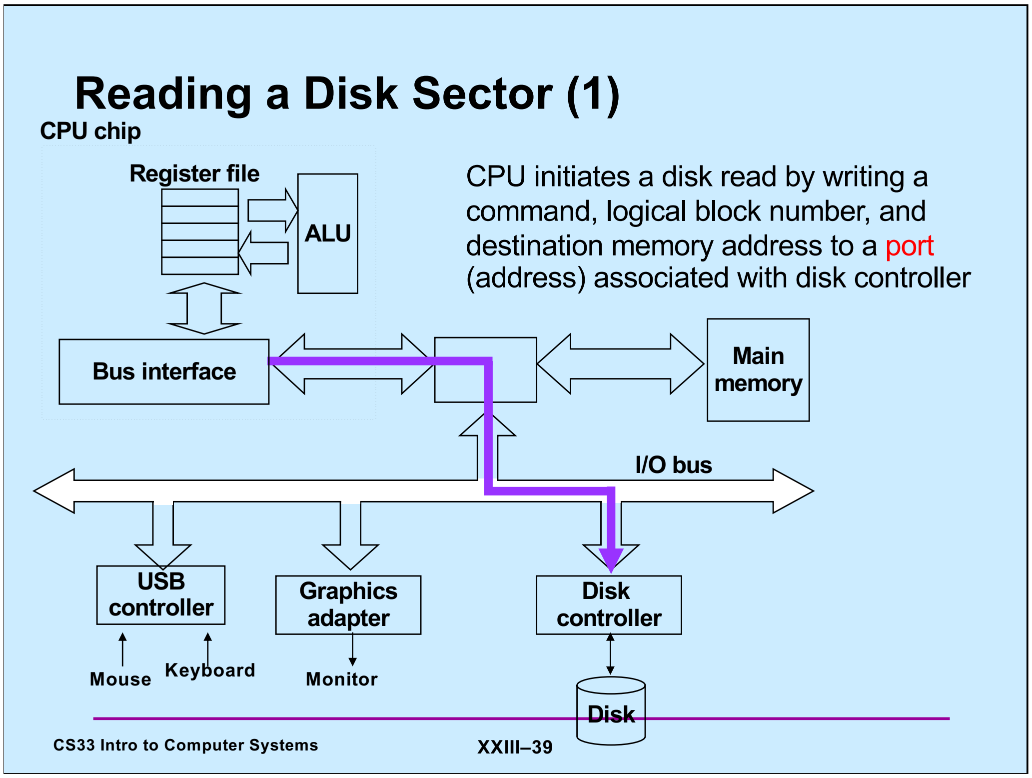
Disk Access Time Example

- 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

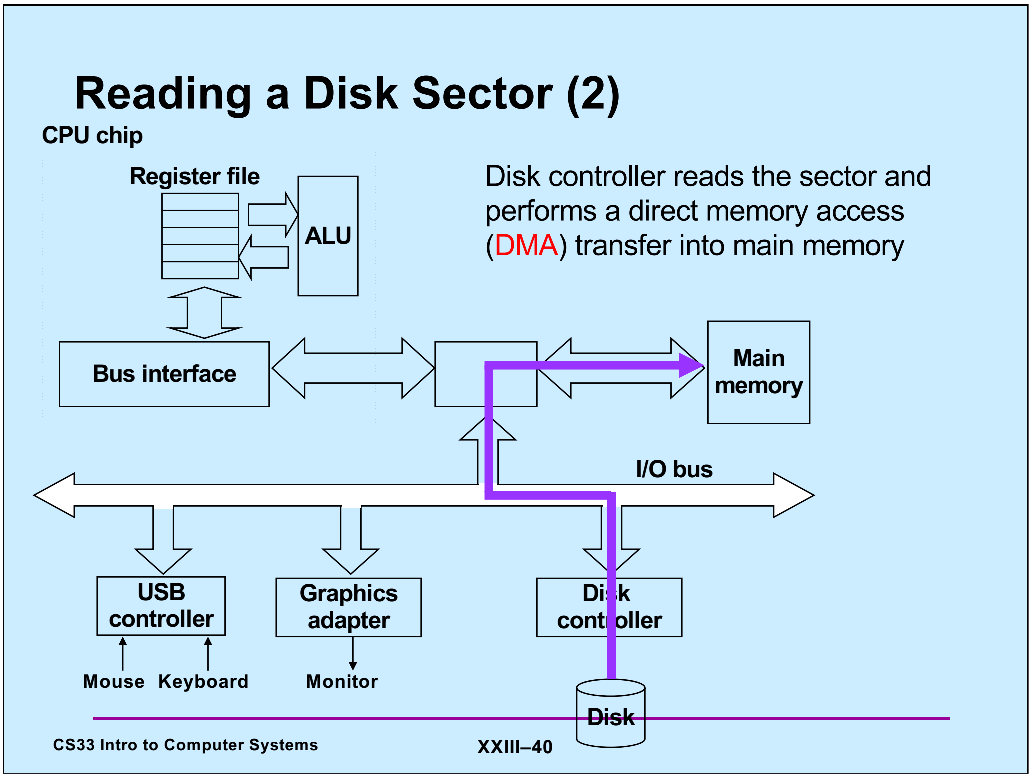
Supplied by CMU.



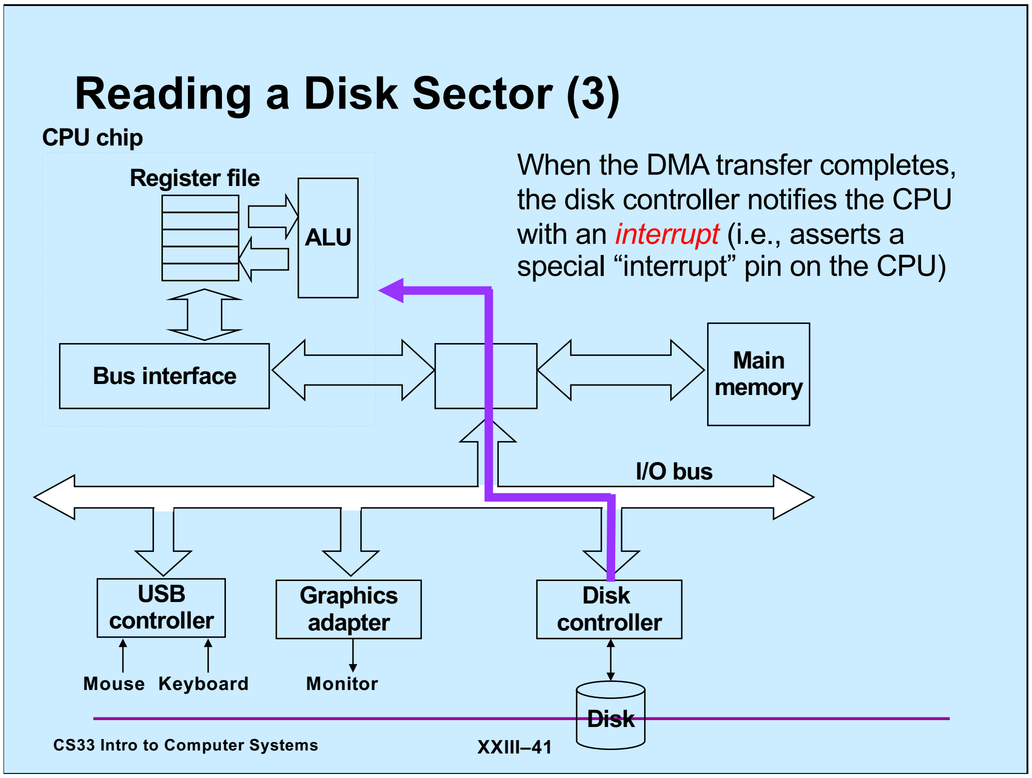
Supplied by CMU.



Supplied by CMU.

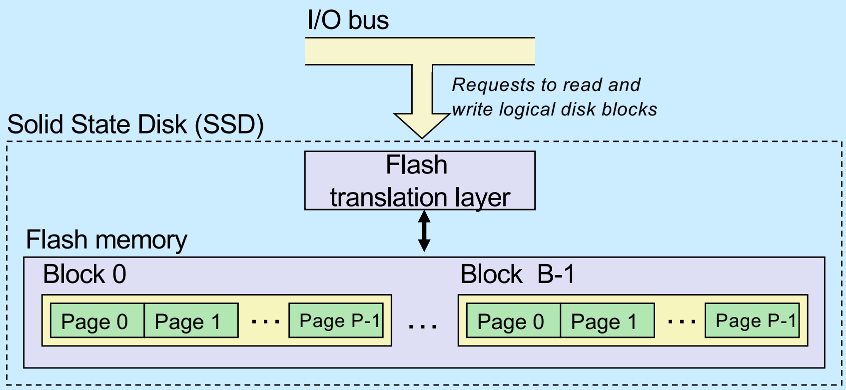


Supplied by CMU.



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

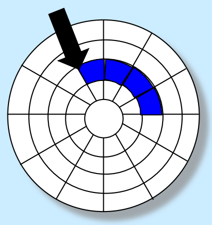
- Advantages
  - no moving parts → faster, less power, more rugged
- Disadvantages
  - have the potential to wear out
    - mitigated by “wear-leveling logic” in flash translation layer
    - e.g. Intel X25 guarantees 1 petabyte (10^{15} bytes) of random writes before they wear out
  - in 2010, about 100 times more expensive per byte
  - in 2017, about 6 times more expensive per byte
  - in 2023, about 1-1.5 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

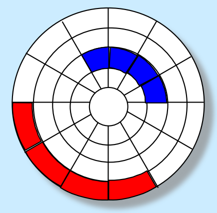


Quiz 2

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 (within a factor of 2)**
- **c) much 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

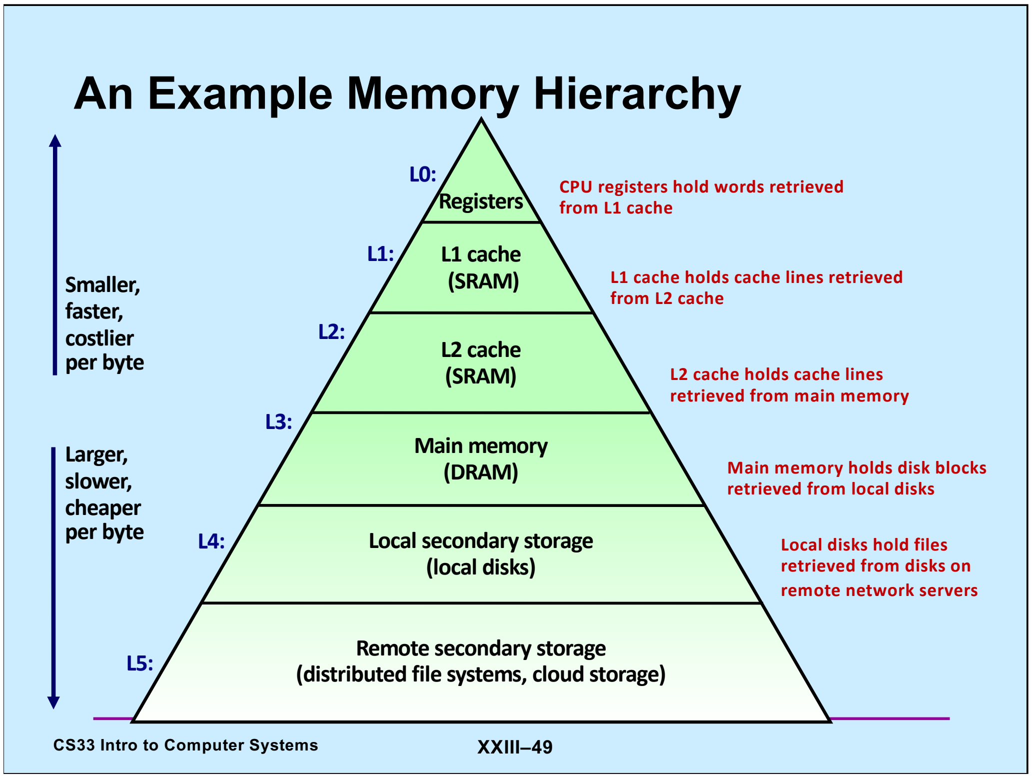
b) about the same amount of time (within a factor of 2)

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.



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

