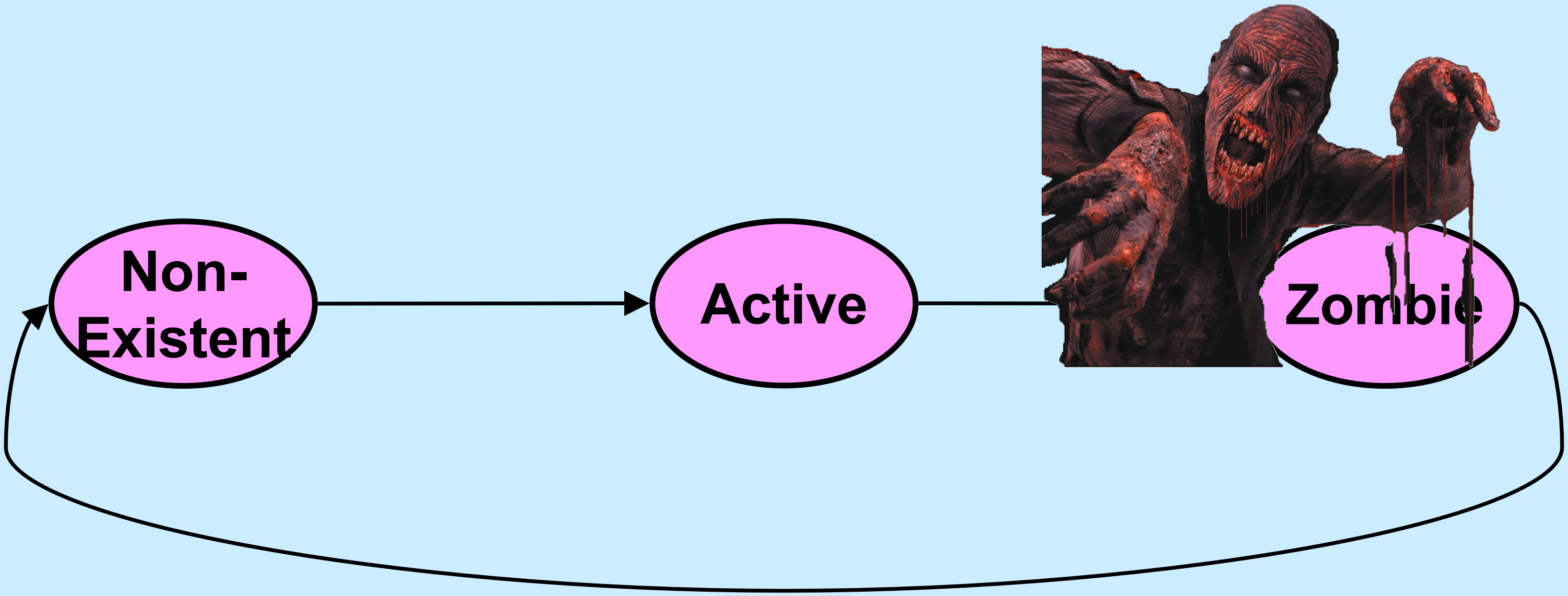
CS 33

Signals Part 3

Process Life Cycle



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)`

(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)

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**

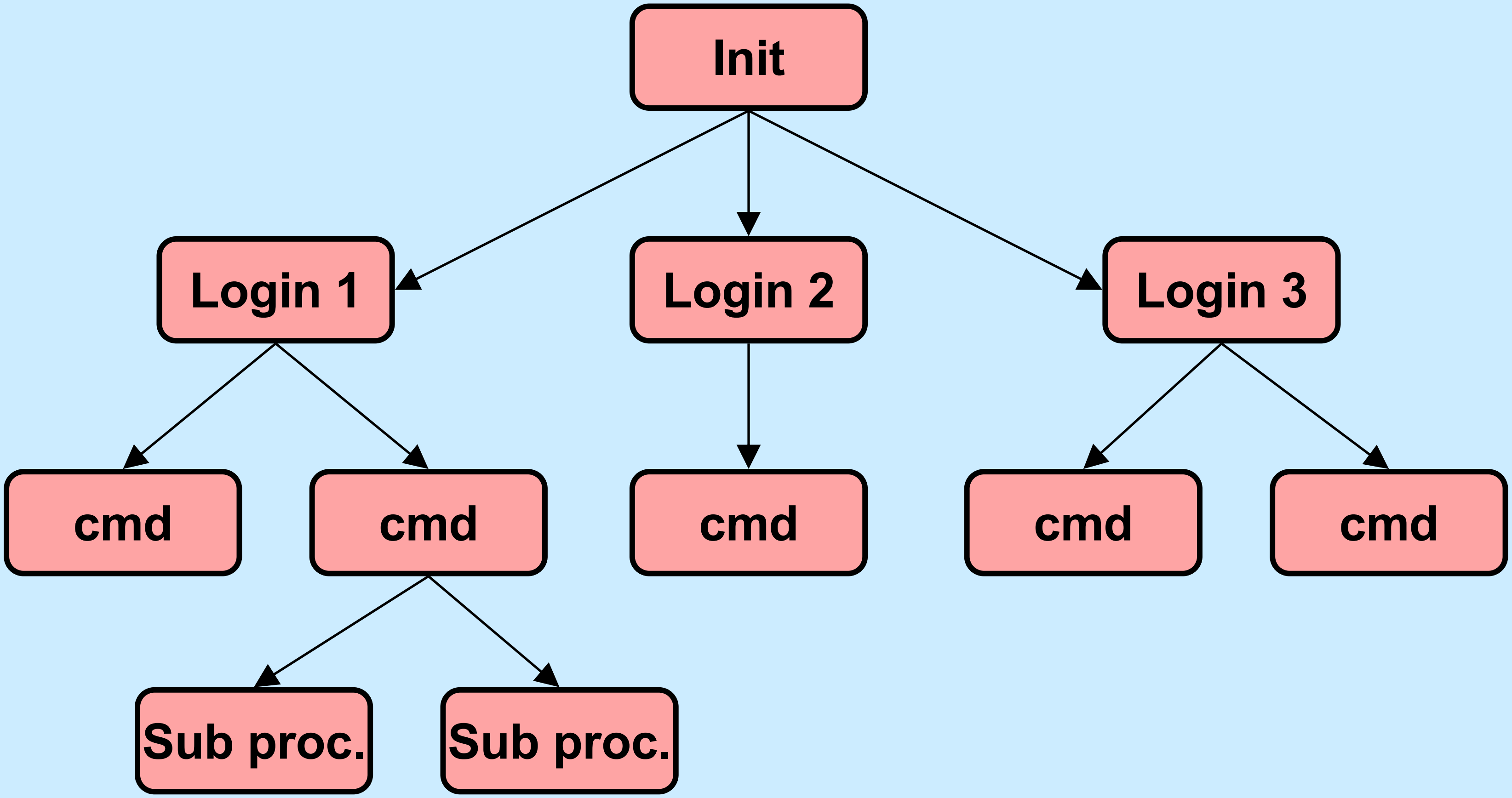
`waitpid` status

- 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

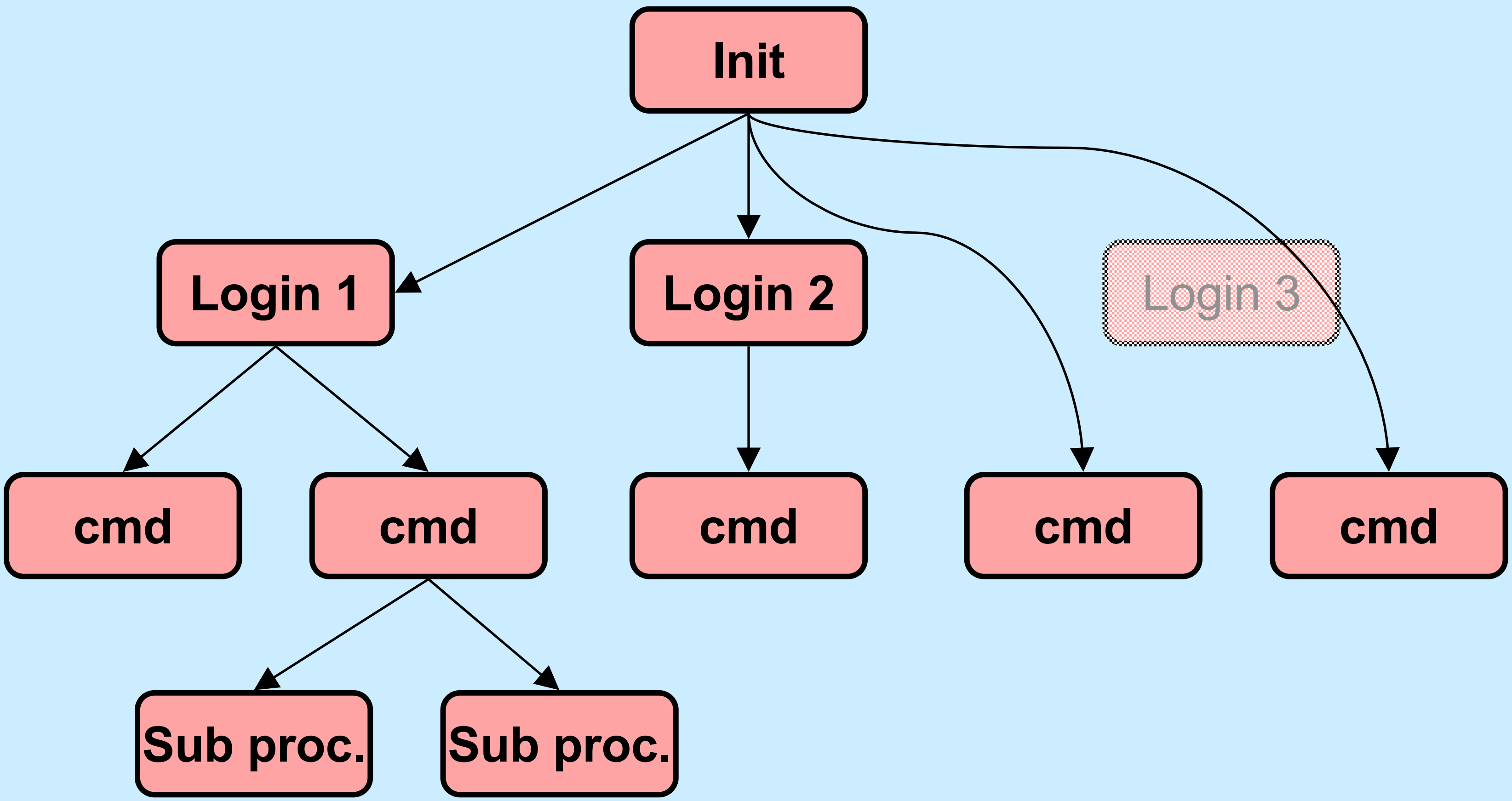
Example (in Shell)

```
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
    ...
  }
}
```

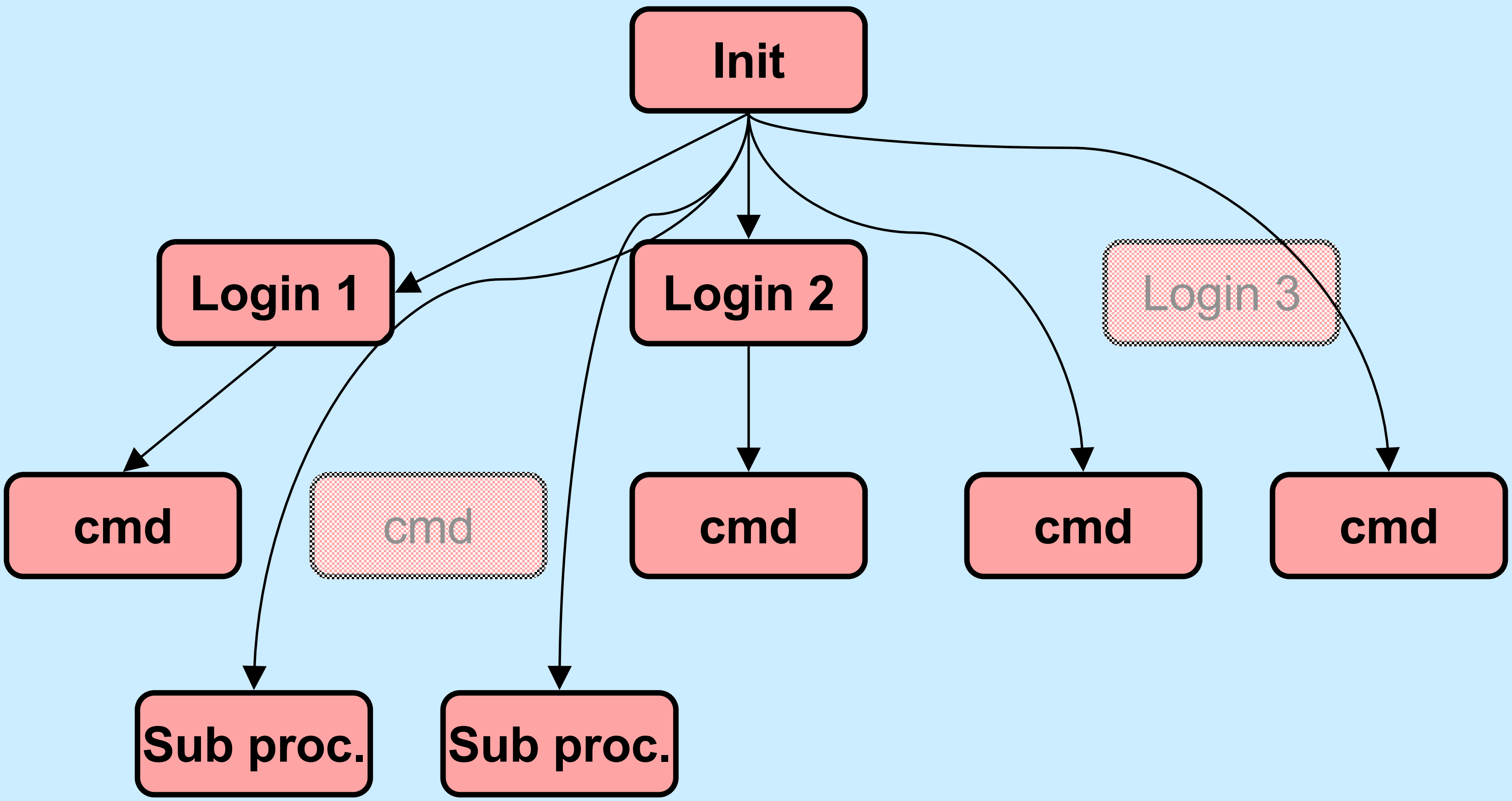

Process Relationships (1)



Process Relationships (2)



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?
}
```

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

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

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

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;
}
```


Asynchronous Signals (1)

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

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

}

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


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);
}
```

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");
}
```


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;
    }
  }
}
```


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

Quiz 1

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

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

CS 33

Memory Hierarchy II

What’s Inside A Disk Drive?

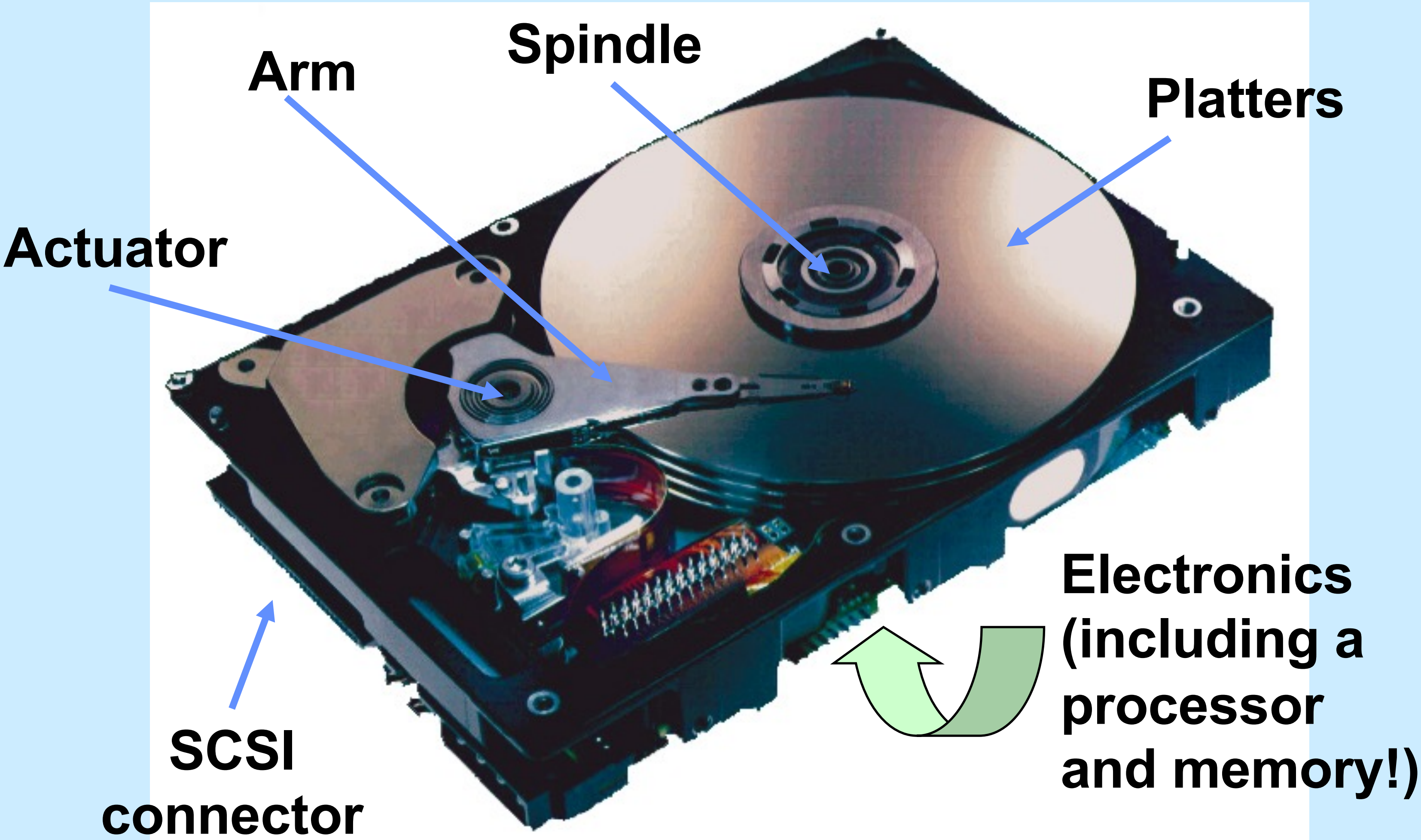
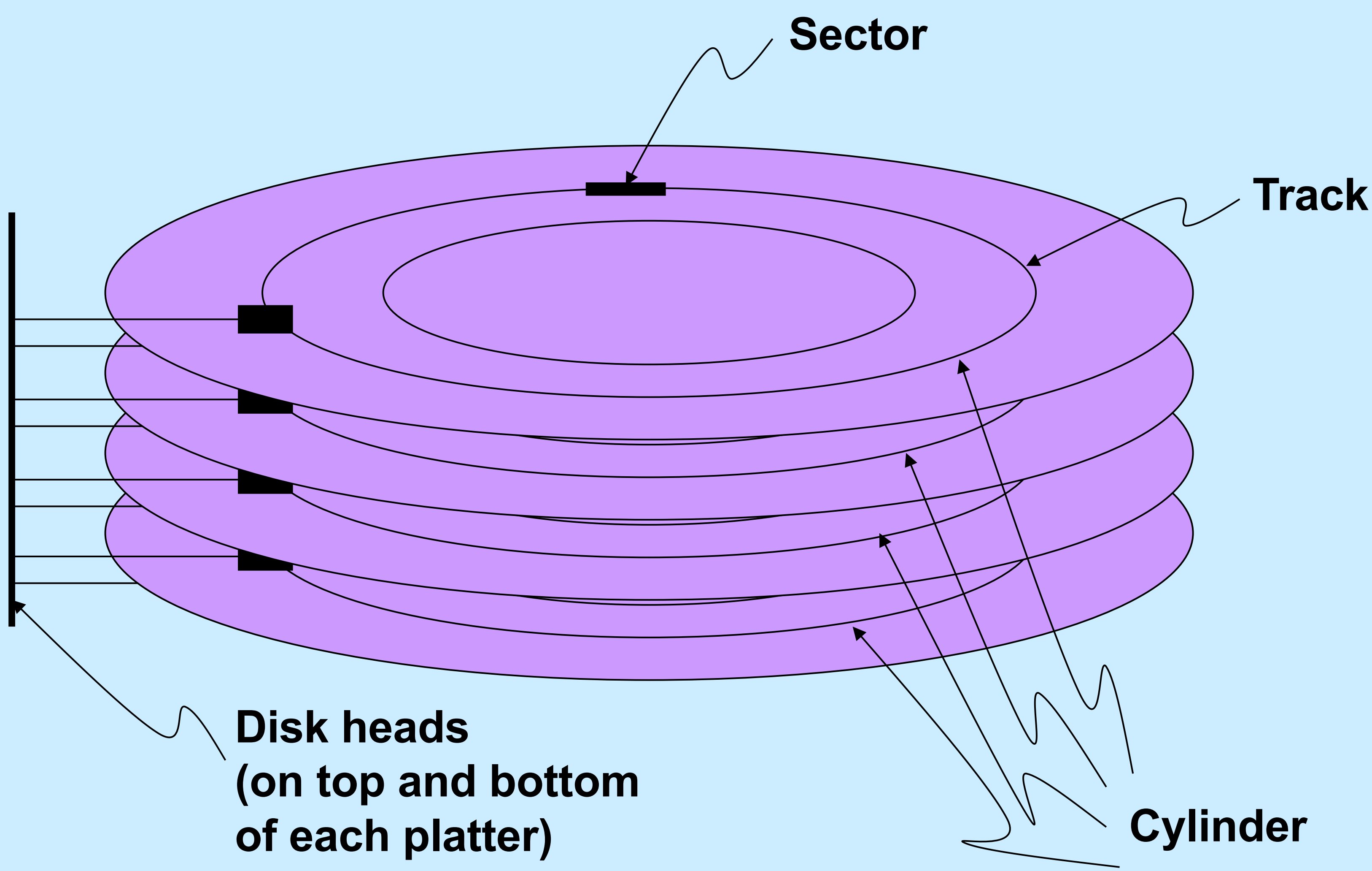


Image courtesy of Seagate Technology

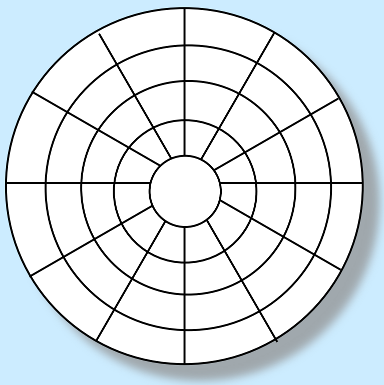
Disk Architecture



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

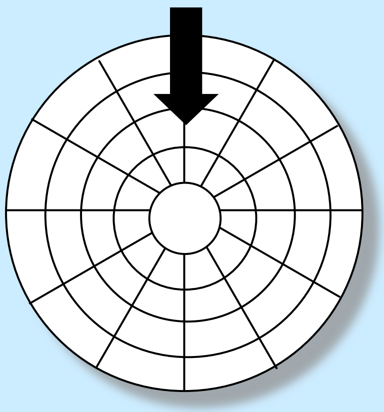
Disk Structure: Top View of Single Platter



Surface organized into tracks

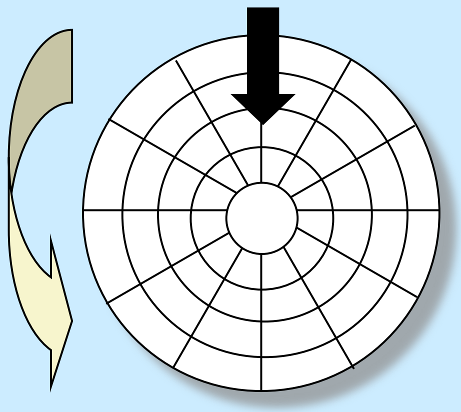
Tracks divided into sectors

Disk Access



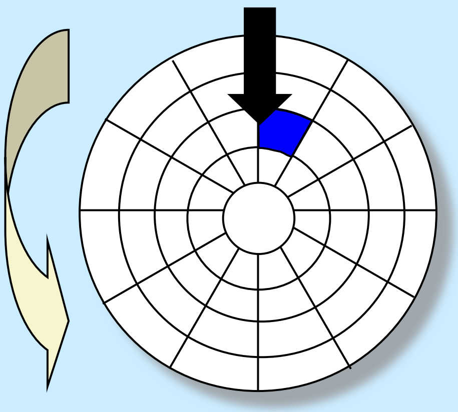
Head in position above a track

Disk Access



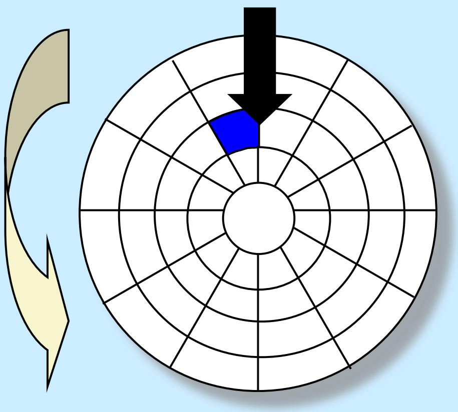
Rotation is counter-clockwise

Disk Access – Read



About to read blue sector

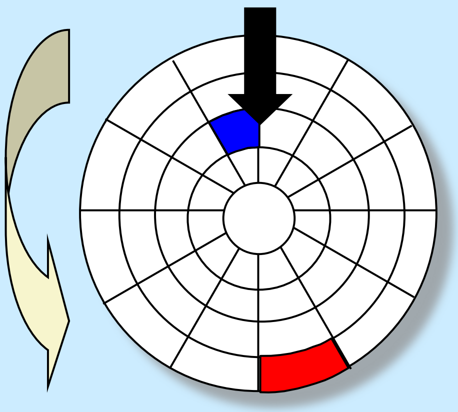
Disk Access – Read



After **BLUE** read

After reading blue sector

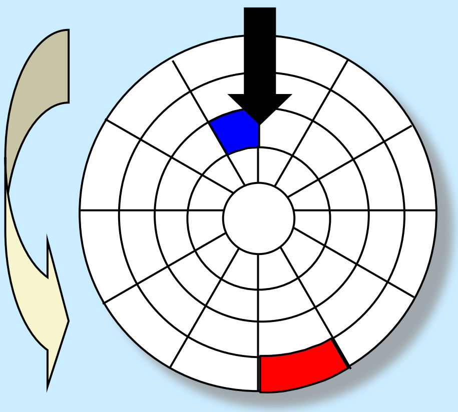
Disk Access – Read



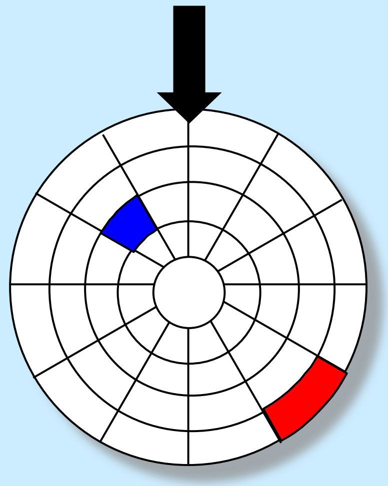
After **BLUE** read

Red request scheduled next

Disk Access – Seek



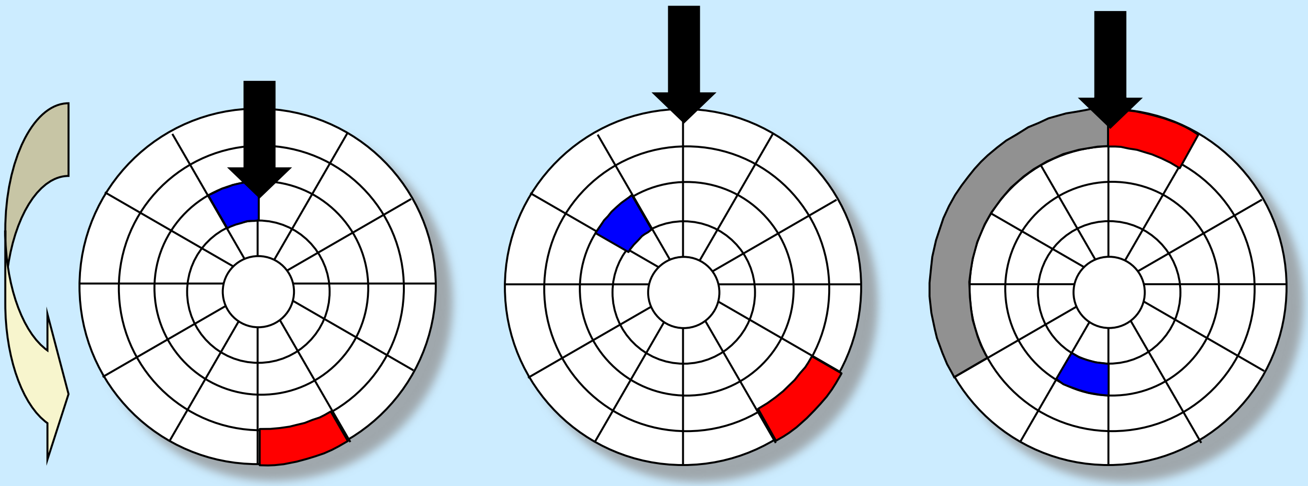
After **BLUE** read



Seek for **RED**

Seek to red's track

Disk Access – Rotational Latency



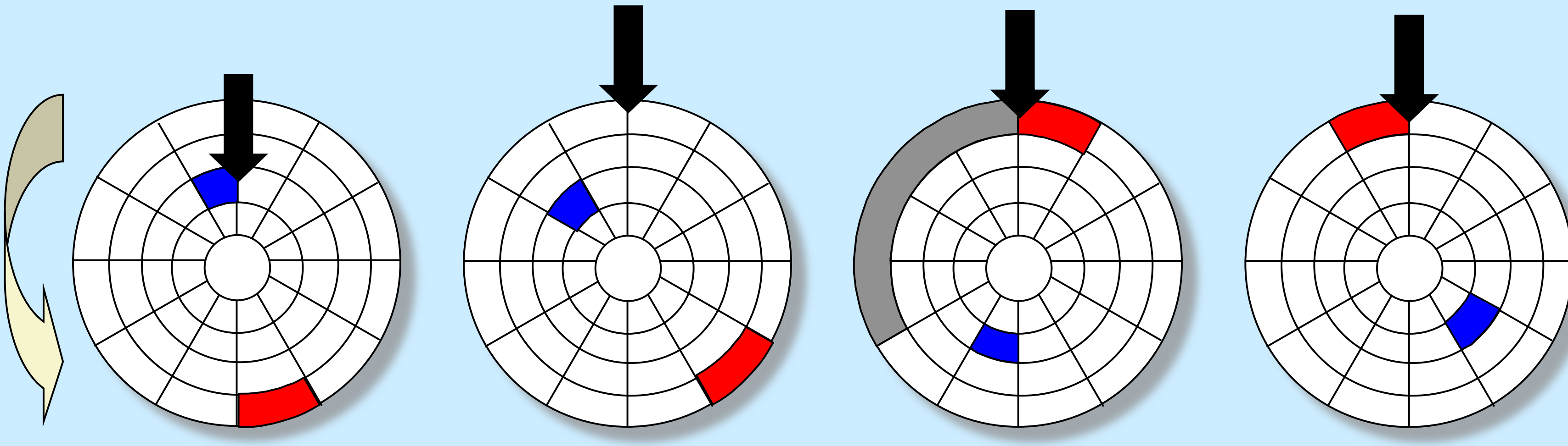
After **BLUE** read

Seek for **RED**

Rotational latency

Wait for red sector to rotate around

Disk Access – Read



After **BLUE** read

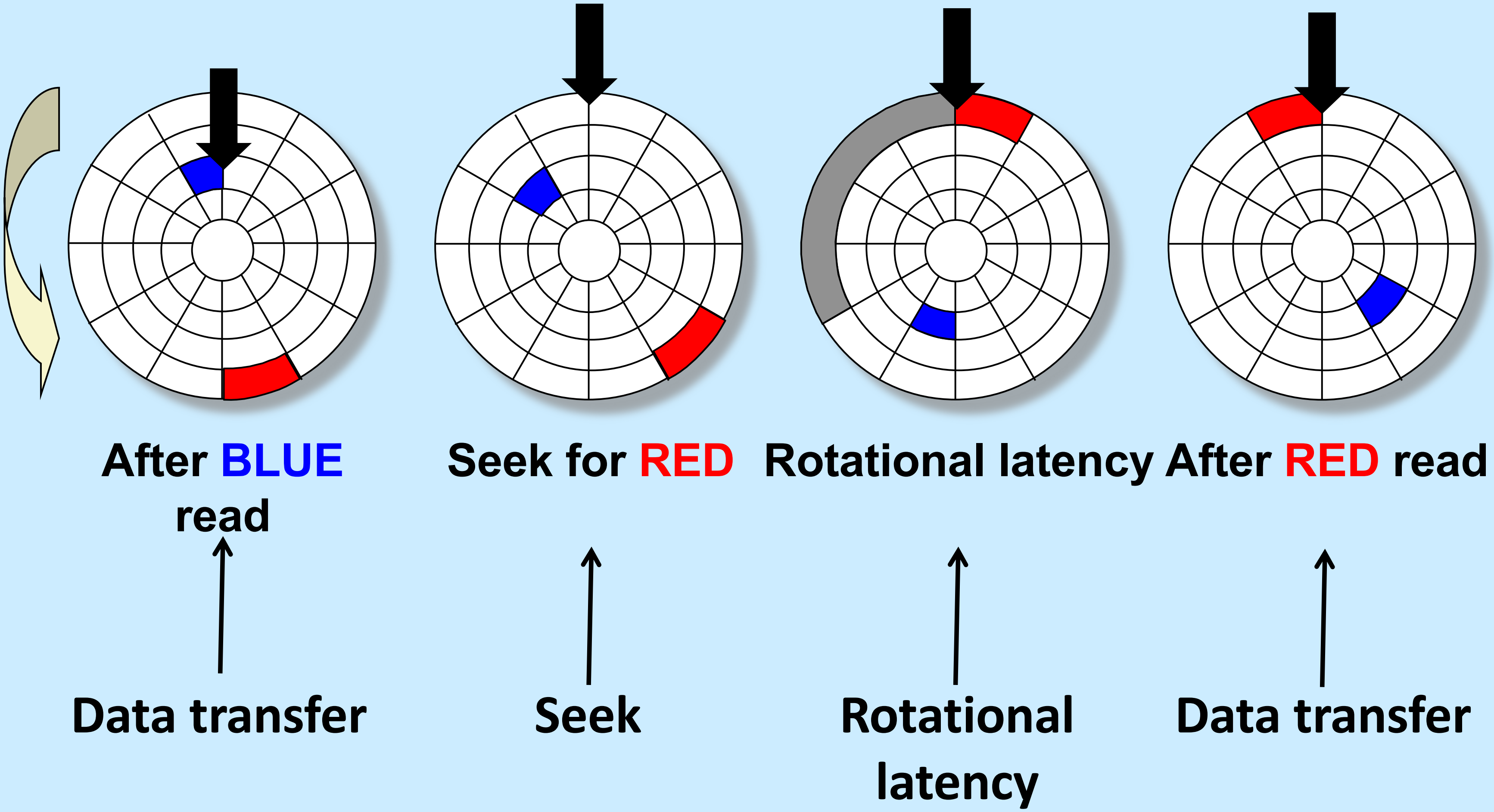
Seek for **RED**

Rotational latency

After **RED** read

Complete read of red

Disk Access – Service Time Components



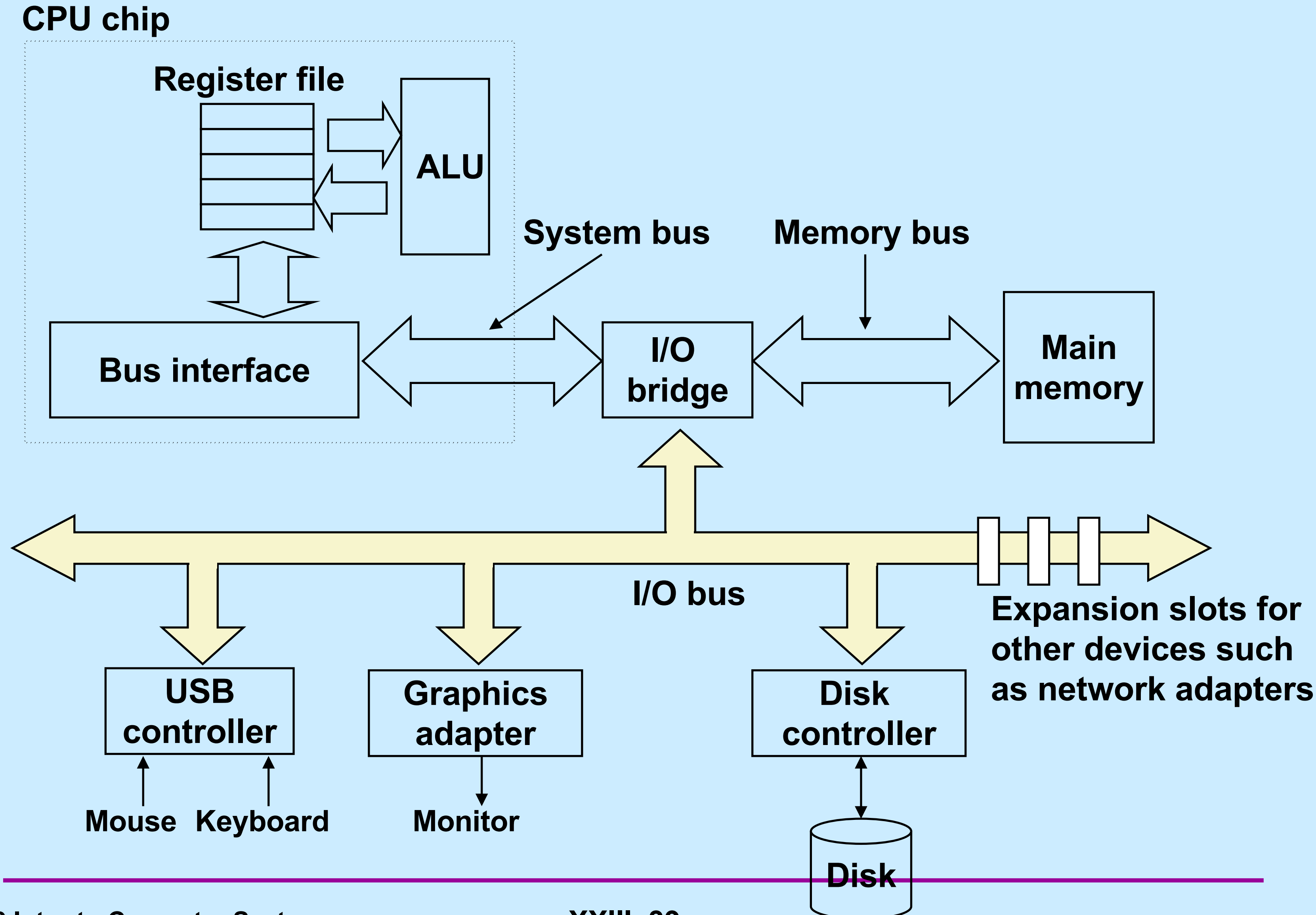
Disk Access Time

- 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

Disk Access Time Example

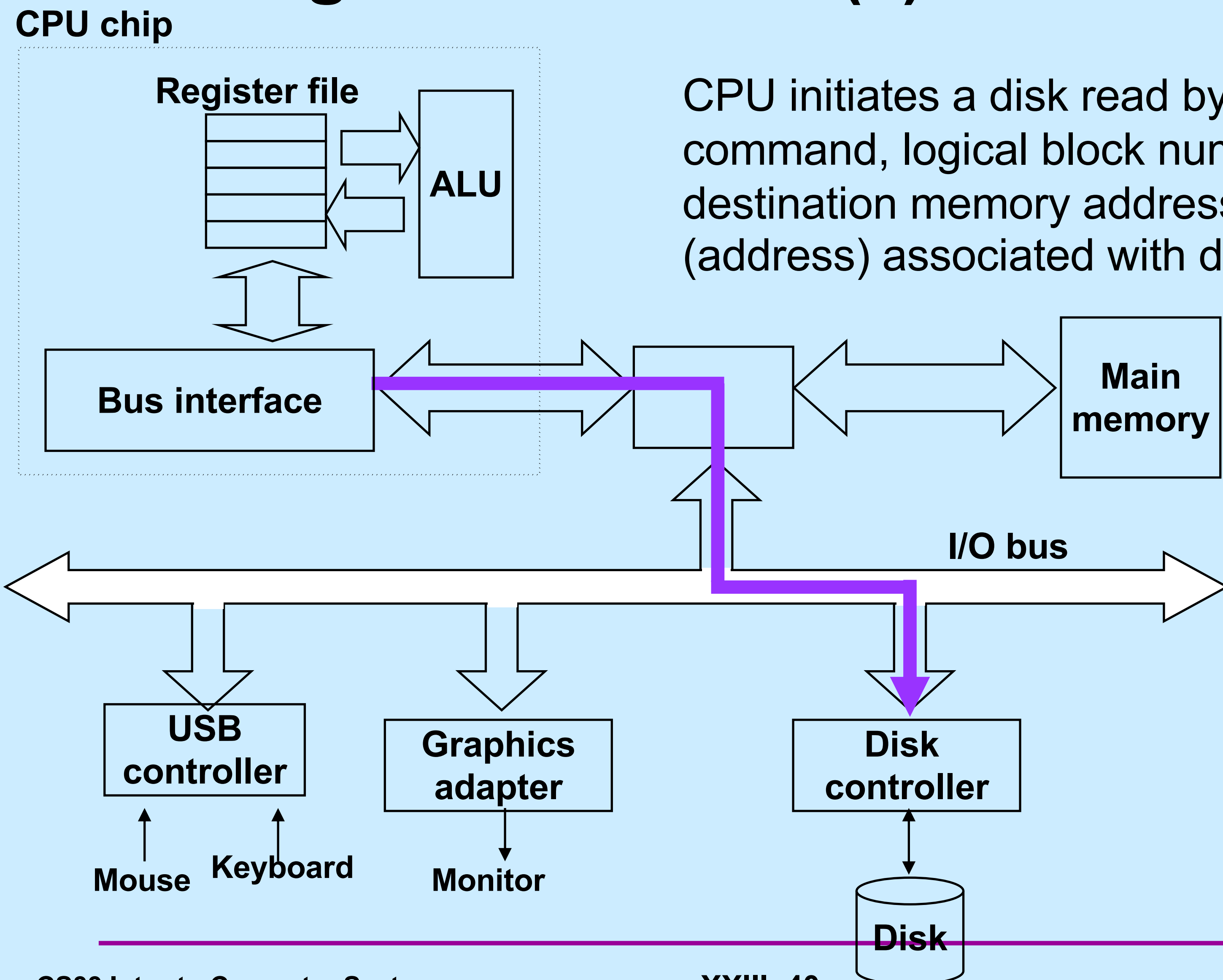
- **Given:**
  - rotational rate = 7,200 RPM
  - average seek time = 9 ms
  - avg # sectors/track = 600
- **Derived:**
  - Tavg rotation = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms
  - Tavg transfer = 60/7200 RPM x 1/600 sects/track x 1000 ms/sec = 0.014 ms
  - Taccess = 9 ms + 4 ms + 0.014 ms
- **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

I/O Bus


CPU chip
Register file
ALU
System bus
Memory bus
Bus interface
I/O bridge
Main memory
I/O bus
Expansion slots for other devices such as network adapters.
USB controller
Graphics adapter
Disk controller
Mouse
Keyboard
Monitor
Disk

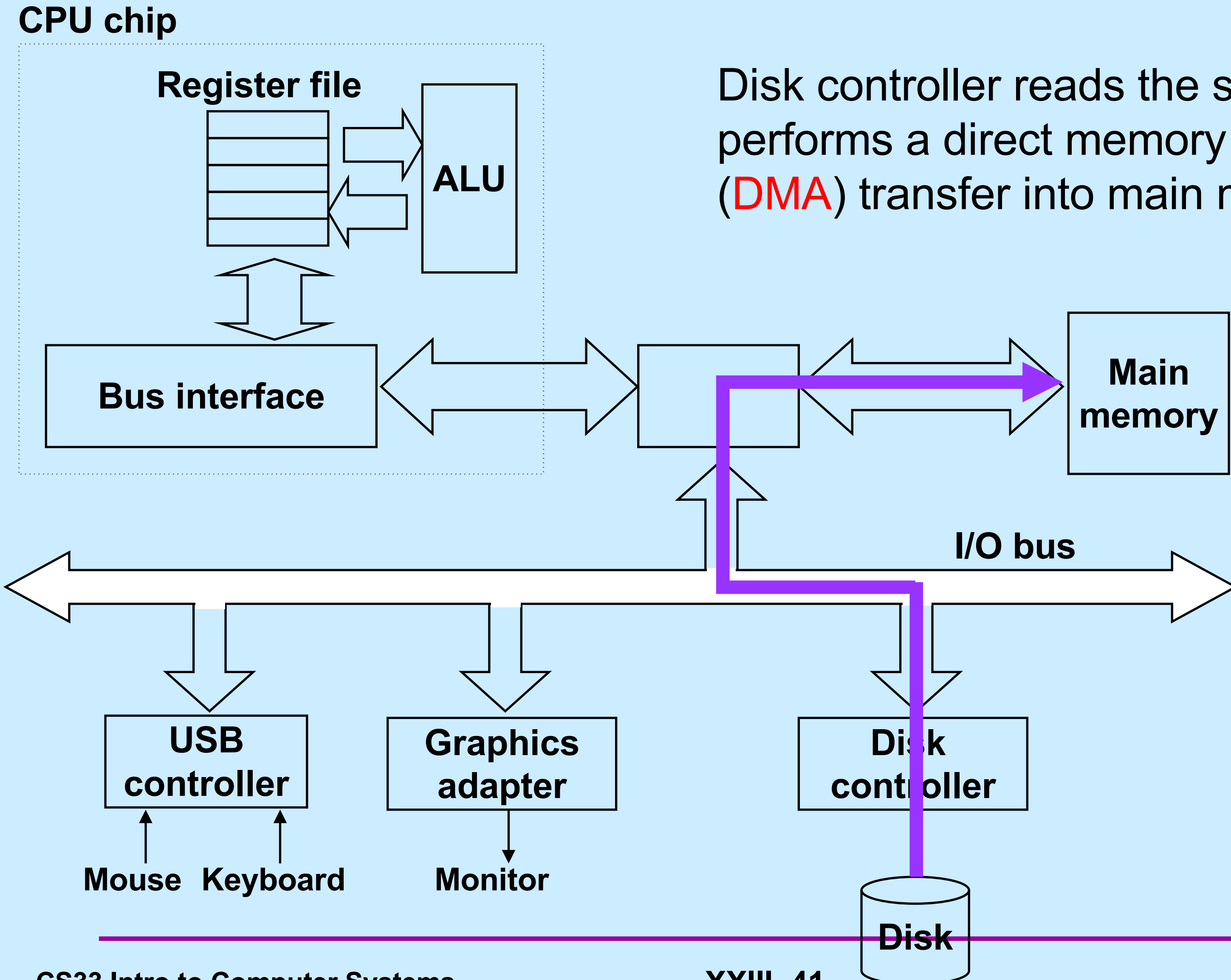
Reading a Disk Sector (1)

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

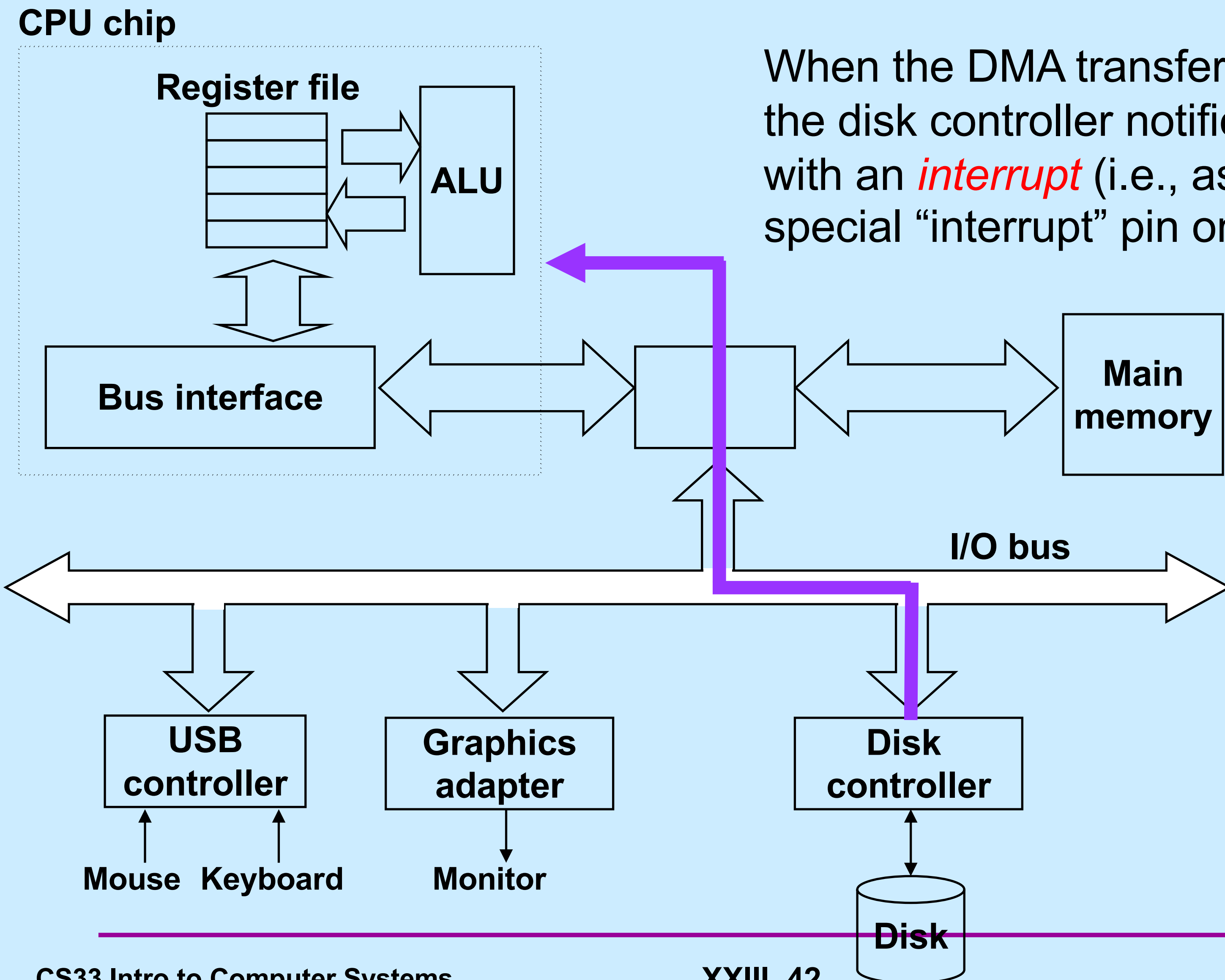


Reading a Disk Sector (2)

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

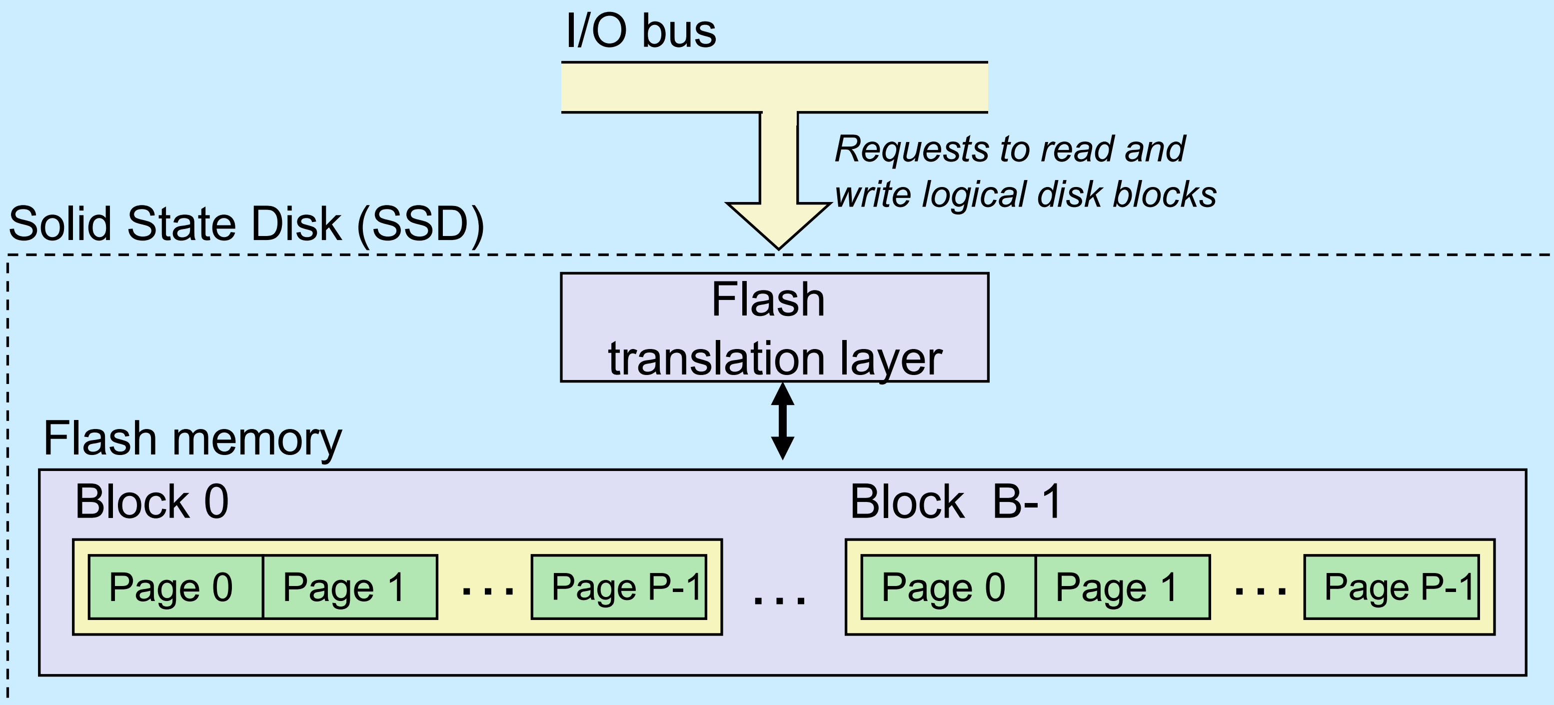


Reading a Disk Sector (3)



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

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**

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

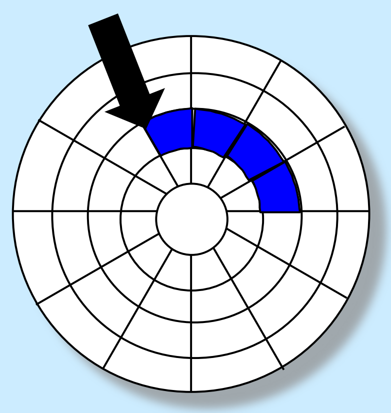
- Why are random writes so slow?
  - erasing a block is slow (around 1 ms)
  - modifying a page triggers a copy of all useful pages in the block
    - » find a used block (new block) and erase it
    - » write the page into the new block
    - » copy other pages from old block to the new block

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 2024, about 1+ϵ times more expensive per byte
- **Applications**
  - smart phones, laptops, desktops

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

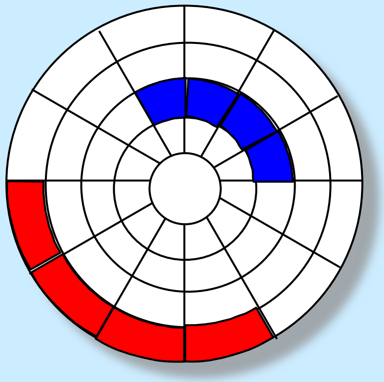
We have two files on the same (rotating) disk. The first file's data resides in consecutive sectors on one track, the second in consecutive sectors on another track. It takes a total of *t* seconds to read all of the first file then all of the second file.

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) much more time

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



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

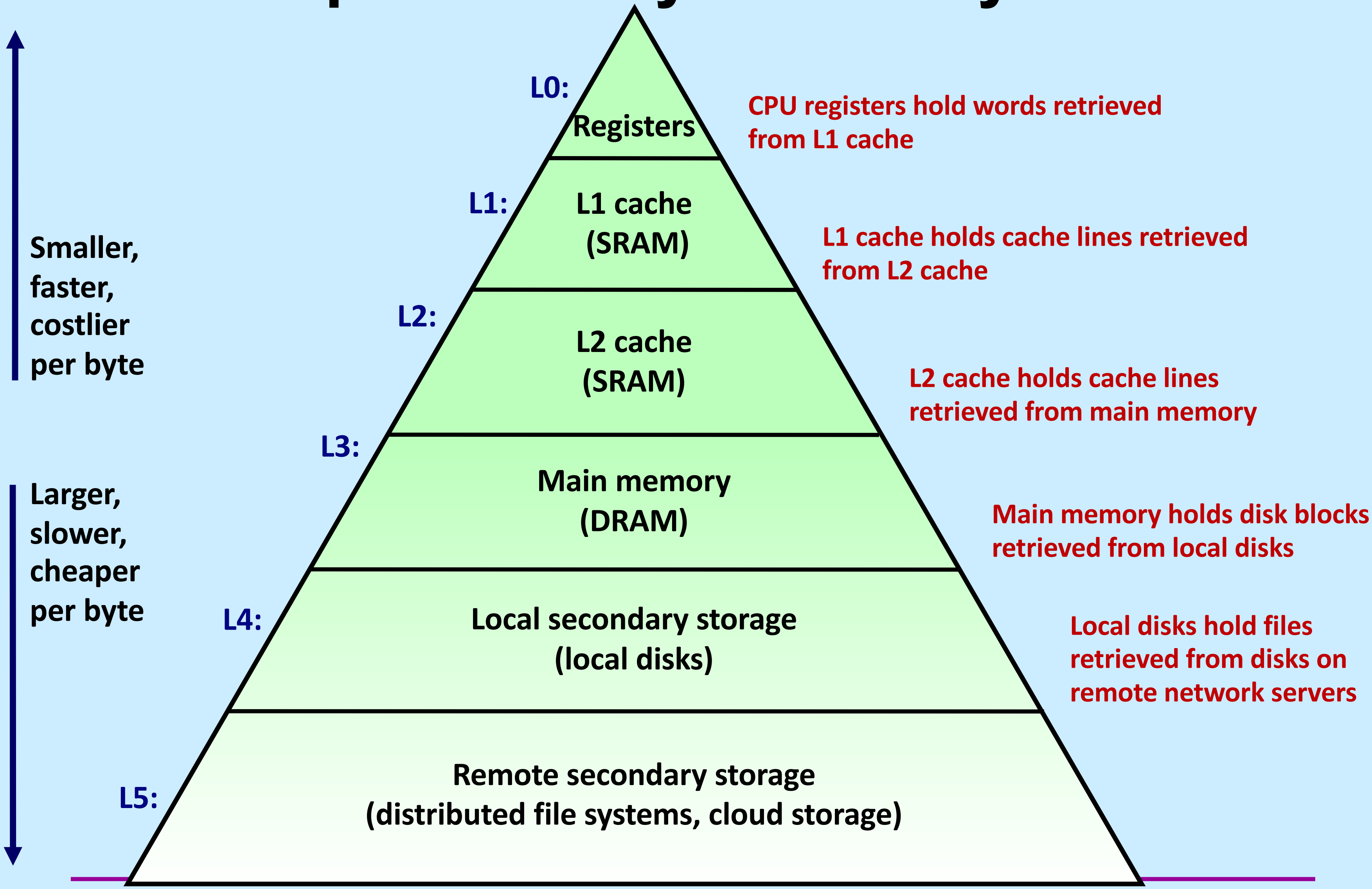
b) much more time

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

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**

An Example Memory Hierarchy



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

Disks Are Still Important

- Cheap
  - cost/byte less than SSDs (but not by much)
- (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 2 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