Lecture 16: Mass-Storage Systems

Instructor: Mitch Neilsen

Office: N219D

Quote of the Day

"As a cure for worrying, work is better than whisky."

-- Thomas Edison

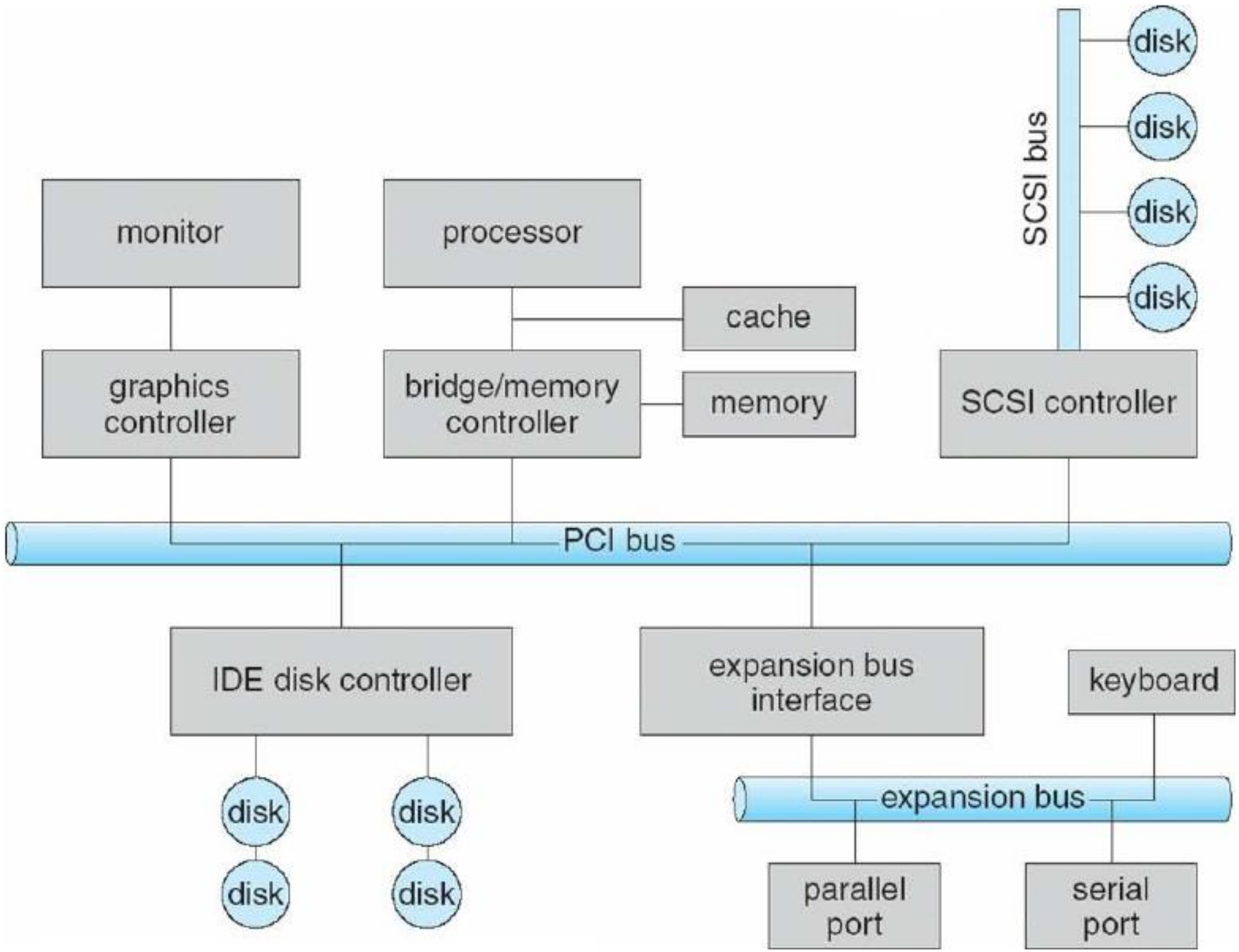
Outline – Chapter 10/12

- **Overview of Mass Storage Structure**
- **Disk Structure**
- **Disk Attachment**
- **Disk Scheduling**
- **Disk Management**
- Swap-Space Management
- RAID Structure
- Stable-Storage Implementation

What is memory?

- **SRAM – Static RAM**
  - Like two NOT gates circularly wired input-to-output
  - 4–6 transistors per bit, actively holds its value
  - Very fast, used to cache slower memory
- **DRAM – Dynamic RAM**
  - A capacitor + gate, holds charge to indicate bit value
  - 1 transistor per bit – extremely dense storage
  - Charge leaks—need slow comparator to decide if bit 1 or 0
  - Must re-write charge after reading, and periodically refresh
- **VRAM – “Video RAM”**
  - Dual ported, can write while someone else reads

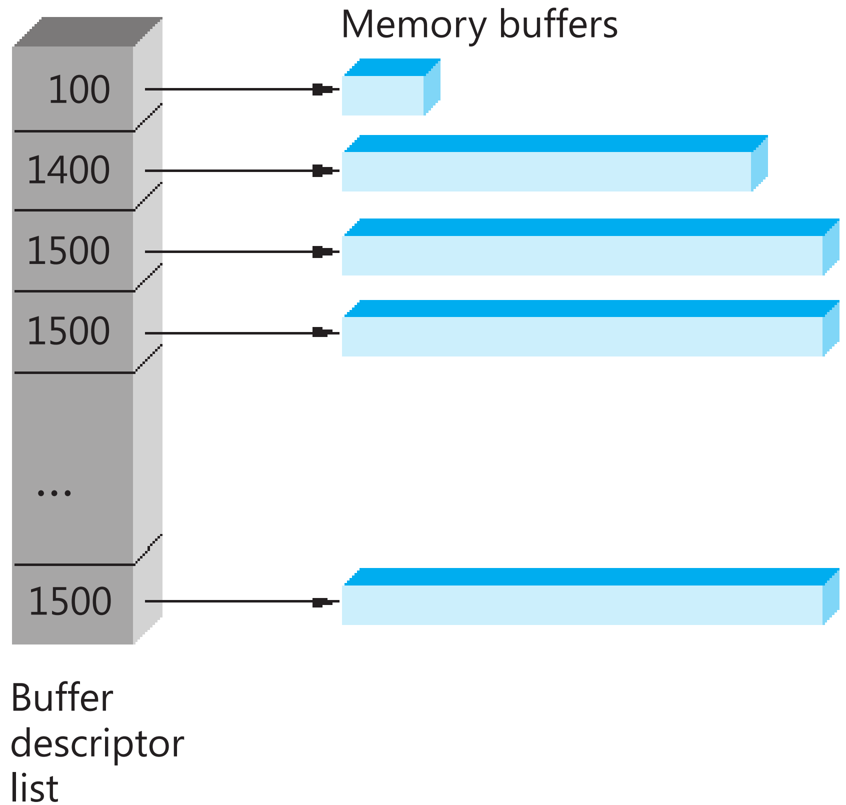
What is I/O bus? E.g., PCI



Communicating with a device

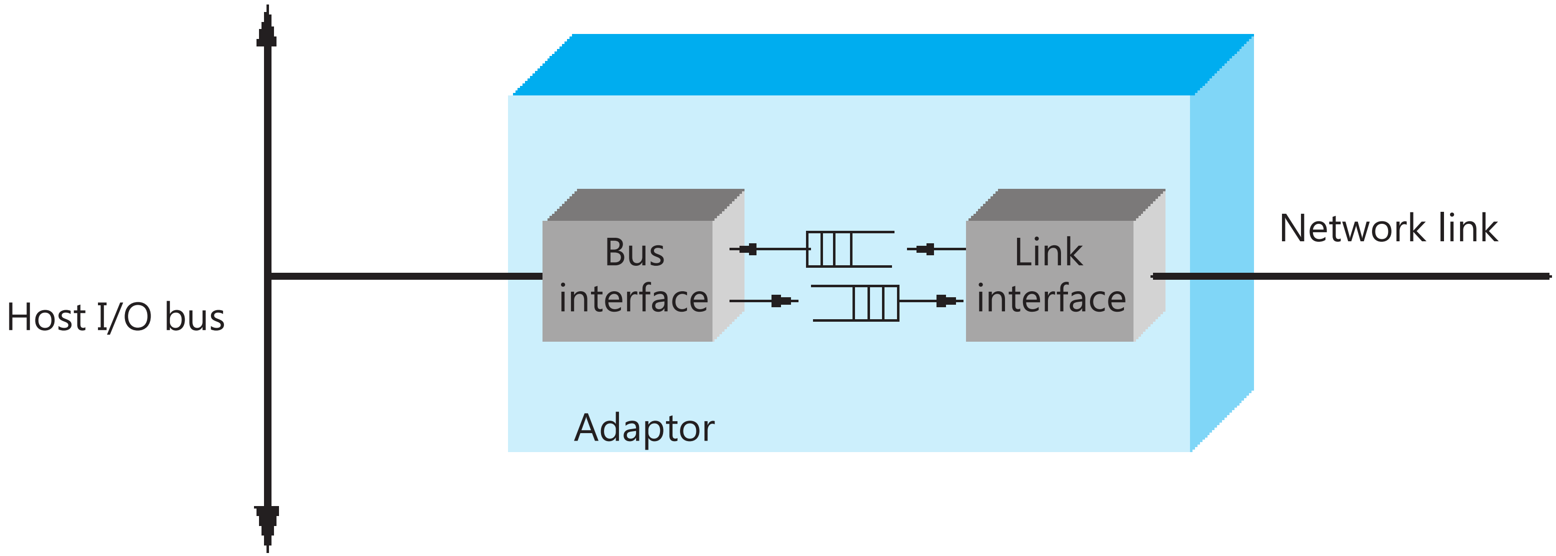
- **Memory-mapped device registers**
  - Certain ***physical*** addresses correspond to device registers
  - Load/store gets status/sends instructions – not real memory
- **Device memory – device may have memory OS can write to directly on other side of I/O bus**
- **Special I/O instructions**
  - Some CPUs (e.g., x86) have special I/O instructions
  - Like load & store, but asserts special I/O pin on CPU
  - OS can allow user-mode access to I/O ports with finer granularity than page
- **DMA – place instructions to card in main memory**
  - Typically then need to "poke" card by writing to register
  - Overlaps unrelated computation with moving data over (typically slower than memory) I/O bus

Direct Memory Access (DMA) buffers



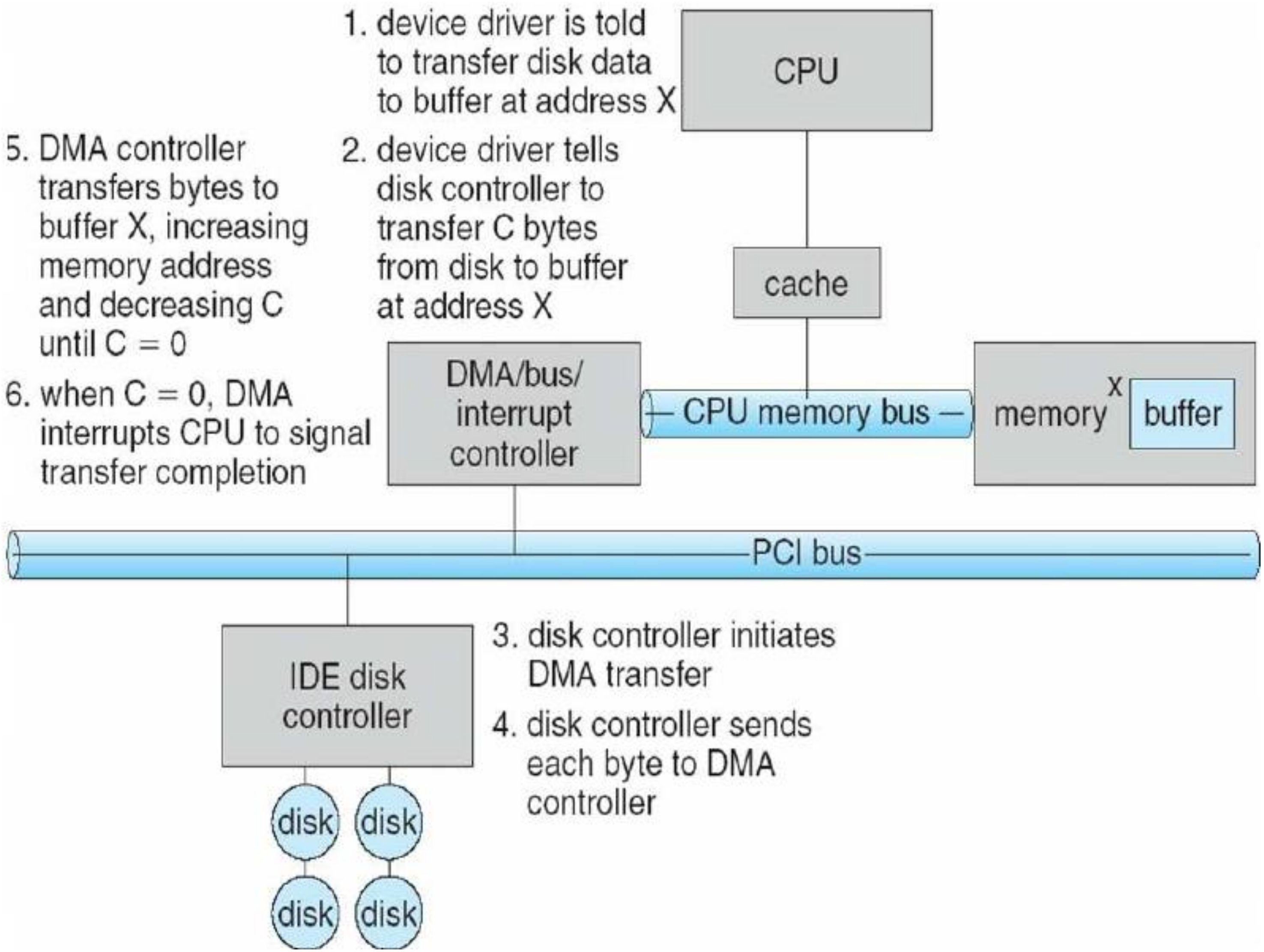
- **Include list of buffer locations in main memory**
- **Card reads list then accesses buffers (w. DMA)**
  - Descriptions sometimes allow for scatter/gather I/O

Example: Network Interface Card



- **Link interface talks to wire/fiber/antenna**
  - Typically does framing, link-layer CRC
- **FIFOs on card provide small amount of buffering**
- **Bus interface logic uses DMA to move packets to and from buffers in main memory**

Example: IDE disk read w/ DMA



Driver architecture

- **Device driver provides several entry points to kernel**
  - Reset, ioctl, output, interrupt, read, write, strategy . . .
- **How should driver synchronize with card?**
  - E.g., Need to know when transmit buffers free or packets arrive
  - Need to know when disk request complete
- **One approach: *Polling***
  - Sent a packet? Loop asking card when buffer is free
  - Waiting to receive? Keep asking card if it has packet
  - Disk I/O? Keep looping until disk ready bit set
- **Disadvantages of polling?**
  - Can’t use CPU for anything else while polling
  - Or schedule poll in future and do something else, but then high latency to receive packet or process disk block

Interrupt driven devices

- **Instead, ask card to interrupt CPU on events**
  - Interrupt handler runs at high priority
  - Asks card what happened (xmit buffer free, new packet)
  - This is what most general-purpose OSes do
- **Bad under high network packet arrival rate**
  - Packets can arrive faster than OS can process them
  - Interrupts are very expensive (context switch)
  - Interrupt handlers have high priority
  - In worst case, can spend 100% of time in interrupt handler and never make any progress – *receive livelock*
  - Best: Adaptive switching between interrupts and polling
- **Very good for disk requests**
- **Rest of today: Disks (network devices later)**

Anatomy of a disk

- **Stack of magnetic platters**
  - Rotate together on a central spindle @3,600-15,000 RPM
  - Drive speed drifts slowly over time
  - Can’t predict rotational position after 100-200 revolutions
- **Disk arm assembly**
  - Arms rotate around pivot, all move together
  - Pivot offers some resistance to linear shocks
  - Arms contain disk heads–one for each recording surface
  - Heads read and write data to platters

Disk

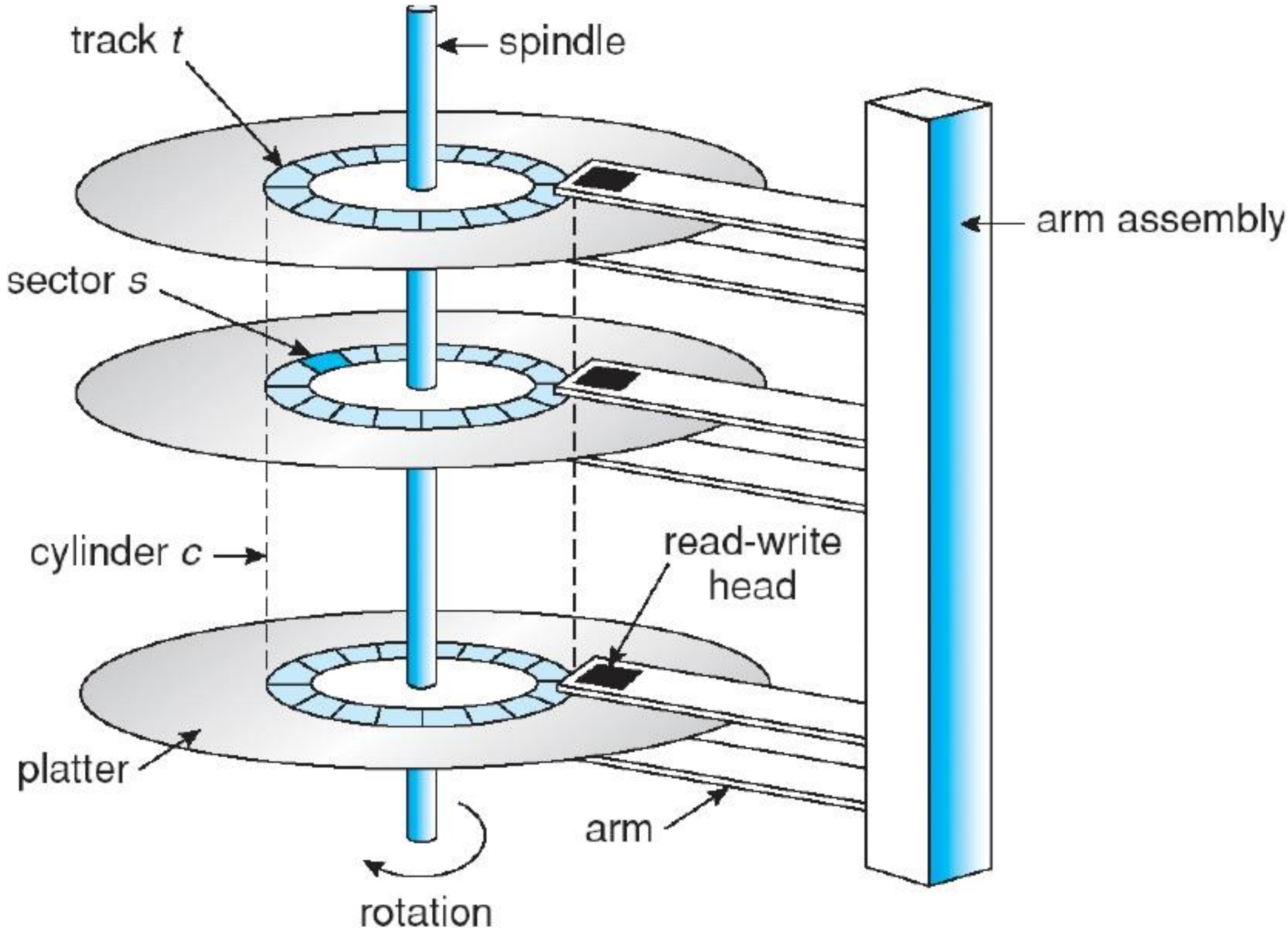
Disk

Disk

Storage on a magnetic platter

- **Platters divided into concentric *tracks***
- **A stack of tracks of fixed radius is a *cylinder***
- **Heads record and sense data along cylinders**
  - Significant fractions of encoded stream for error correction
- **Generally only one head active at a time**
  - Disks usually have one set of read-write circuitry
  - Must worry about cross-talk between channels
  - Hard to keep multiple heads exactly aligned

Cylinders, tracks, & sectors



Disk positioning system

- **Move head to specific track and keep it there**
  - Resist physical socks, imperfect tracks, etc.
- **A *seek* consists of up to four phases:**
  - *speedup*–accelerate arm to max speed or half way point
  - *coast*–at max speed (for long seeks)
  - *slowdown*–stops arm near destination
  - *settle*–adjusts head to actual desired track
- **Very short seeks dominated by settle time (∼1 ms)**
- **Short (200-400 cyl.) seeks dominated by speedup**
  - Accelerations of 40g

Seek details

- **Head switches comparable to short seeks**
  - May also require head adjustment
  - Settles take longer for writes than for reads

    If read strays from track, catch error with checksum, retry

    If write strays, you've just clobbered some other track
- **Disk keeps table of pivot motor power**
  - Maps seek distance to power and time
  - Disk interpolates over entries in table
  - Table set by periodic "thermal recalibration"
  - But, e.g., ∼500 ms recalibration every ∼25 min bad for AV
- **"Average seek time" quoted can be many things**
  - Time to seek 1/3 disk, 1/3 time to seek whole disk

Sectors

- **Disk interface presents linear array of *sectors***
  - Generally 512 bytes, written atomically (even if power failure)
- **Disk maps logical sector #s to physical sectors**
  - *Zoning*–puts more sectors on longer tracks
  - *Track skewing*–sector 0 pos. varies by track (why?)
  - *Sparing*–flawed sectors remapped elsewhere
- **OS doesn't know logical to physical sector mapping**
  - Larger logical sector # difference means larger seek time
  - Highly non-linear relationship (*and* depends on zone)
  - OS has no info on rotational positions
  - Can empirically build table to estimate times

Disk interface

- **Controls hardware, mediates access**
- **Computer, disk often connected by bus (e.g., SCSI)**
  - Multiple devices may contend for bus

Possible disk/interface features:

- **Disconnect from bus during requests**
- **Command queuing: Give disk multiple requests**
  - Disk can schedule them using rotational information
- **Disk cache used for read-ahead**
  - Otherwise, sequential reads would incur whole revolution
  - Cross track boundaries? Can't stop a head-switch
- **Some disks support write caching**
  - But data not stable—not suitable for all requests

SCSI overview

- **SCSI *domain* consists of devices and an SDS**
  - Devices: host adapters & SCSI controllers
  - *Service Delivery Subsystem* connects devices–e.g., SCSI bus
- **SCSI-2 bus (SDS) connects up to 8 devices**
  - Controllers can have > 1 “logical units” (LUNs)
  - Typically, controller built into disk and 1 LUN/target, but “bridge controllers” can manage multiple physical devices
- **Each device can assume role of *initiator* or *target***
  - Traditionally, host adapter was initiator, controller target
  - Now controllers act as initiators (e.g., COPY command)
  - Typical domain has 1 initiator, ≥ 1 targets

SCSI requests

- **A *request* is a command from initiator to target**
  - Once transmitted, target has control of bus
  - Target may disconnect from bus and later reconnect
    (very important for multiple targets or even multitasking)
- **Commands contain the following:**
  - *Task identifier*—initiator ID, target ID, LUN, tag
  - *Command descriptor block*—e.g., read 10 blocks at pos. *N*
  - Optional *task attribute*—SIMPLE, ORDERED, HEAD OF QUEUE
  - Optional: output/input buffer, sense data
  - *Status byte*—GOOD , CHECK CONDITION , INTERMEDIATE, ...

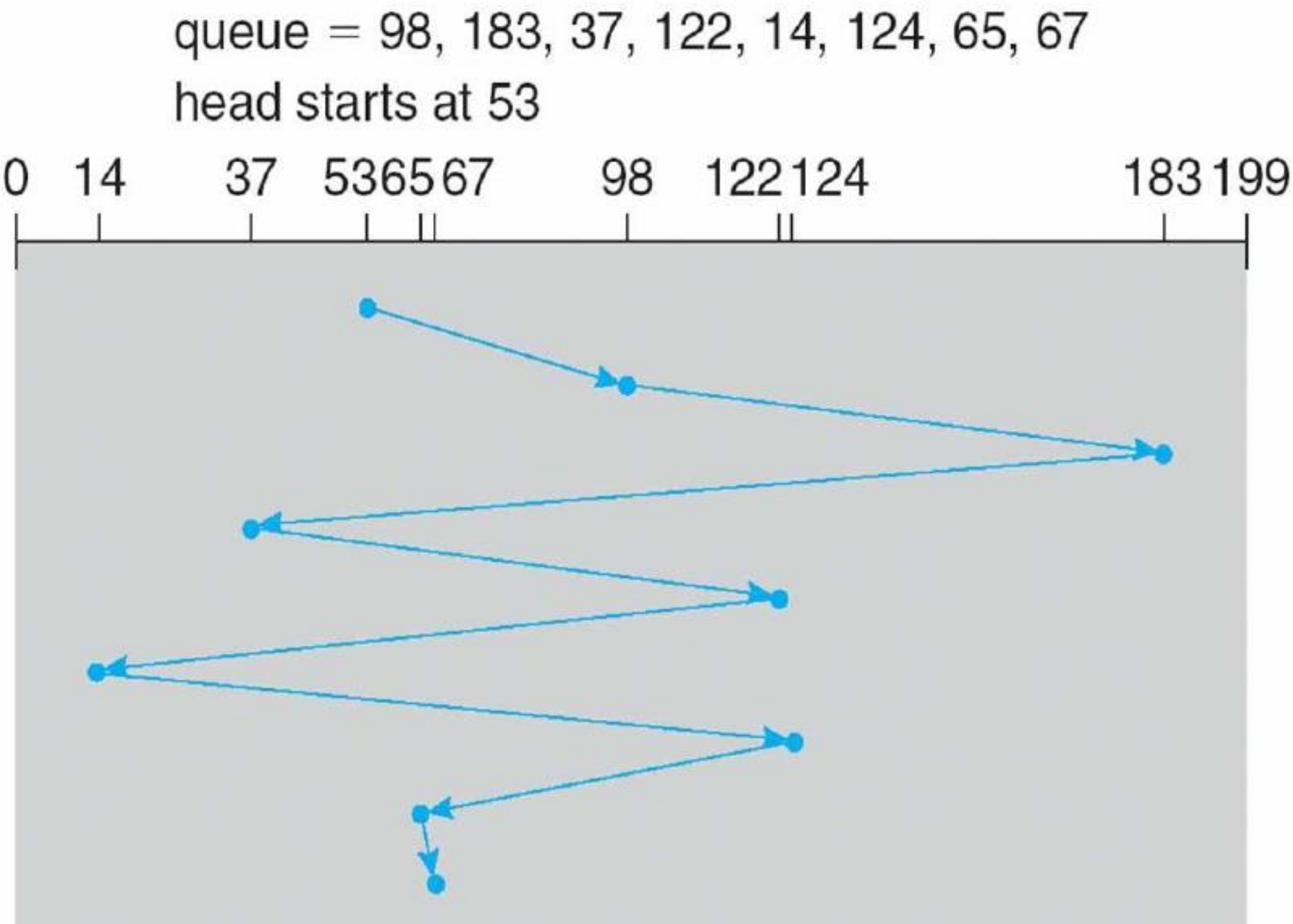
Disk performance

- **Placement & ordering of requests a huge issue**
  - Sequential I/O much, much faster than random
  - Long seeks much slower than short ones
  - Power might fail any time, leaving inconsistent state
- **Must be careful about order for crashes**
  - More on this in next two lectures
- **Try to achieve contiguous accesses where possible**
  - E.g., make big chunks of individual files contiguous
- **Try to order requests to minimize seek times**
  - OS can only do this if it has a multiple requests to order
  - Requires disk I/O concurrency
  - High-performance apps try to maximize I/O concurrency
- **Next: How to schedule concurrent requests**

Scheduling: FCFS

- **“First Come First Served”**
  - Process disk requests in the order they are received
- **Advantages**
  - Easy to implement
  - Good fairness
- **Disadvantages**
  - Cannot exploit request locality
  - Increases average latency, decreasing throughput

FCFS example

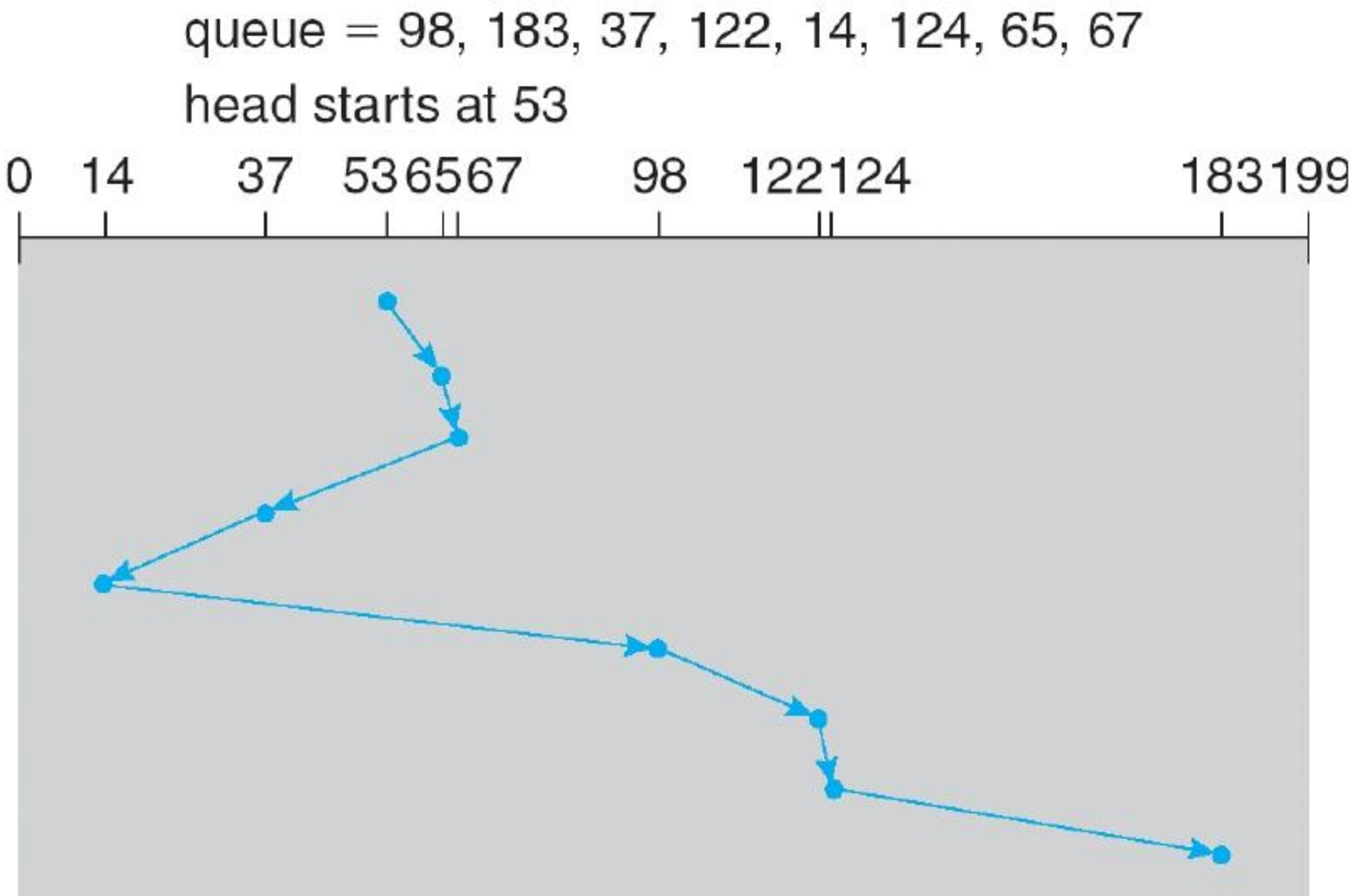


Shortest positioning time first (SPTF)

- **Shortest positioning time first (SPTF)**
  - Always pick request with shortest seek time
- **Advantages**
  - Exploits locality of disk requests
  - Higher throughput
- **Disadvantages**
  - Starvation
  - Don't always know what request will be fastest
- **Improvement: Aged SPTF**
  - Give older requests higher priority
  - Adjust "effective" seek time with weighting factor:

    $T_{\text{eff}} = T_{\text{pos}} - W \cdot T_{\text{wait}}$
- **Also called Shortest Seek Time First (SSTF)**

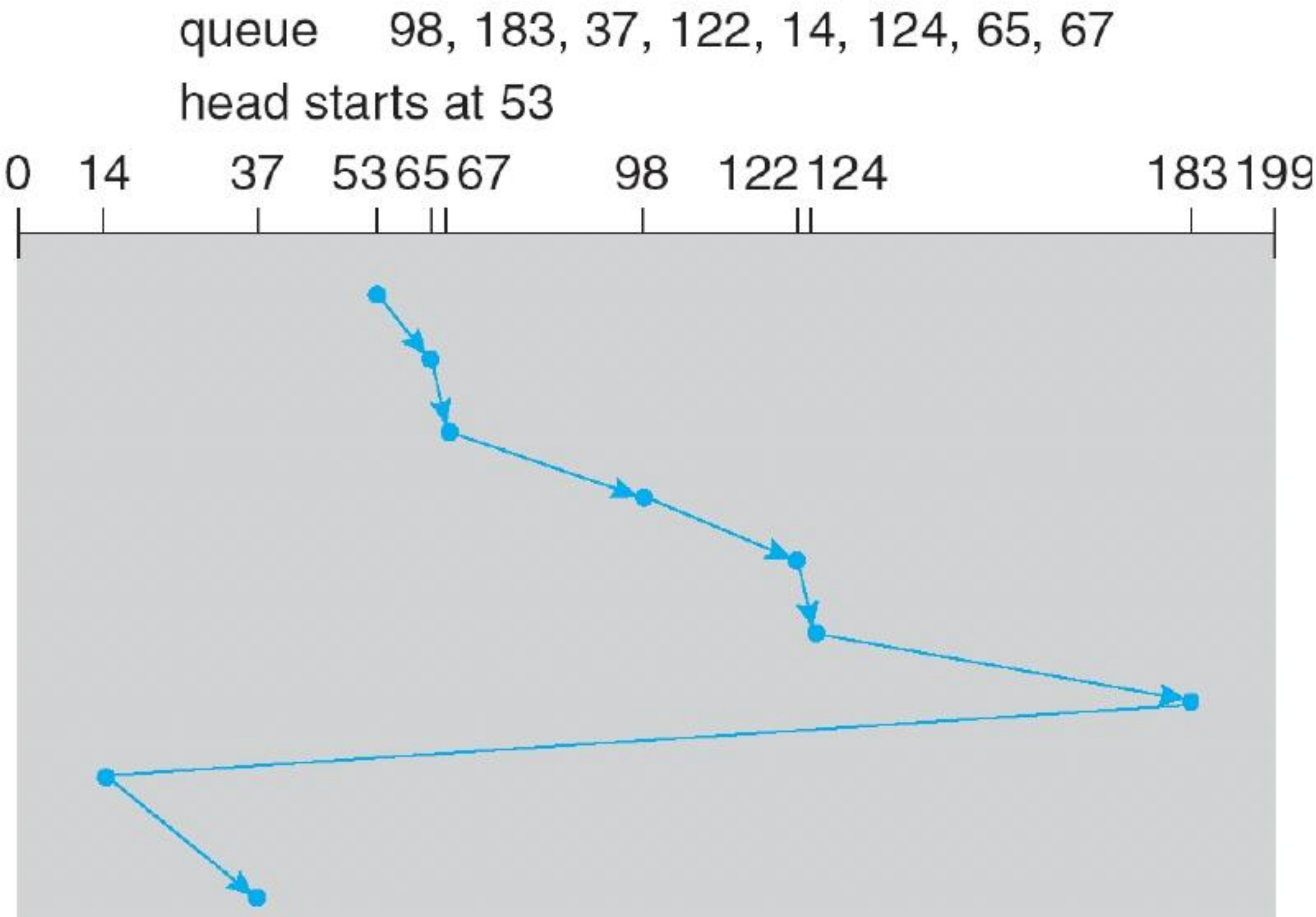
SPTF example



“Elevator ” scheduling (SCAN)

- **Sweep across disk, servicing all requests passed**
  - Like SPTF, but next seek must be in same direction
  - Switch directions only if no further requests
- **Advantages**
  - Takes advantage of locality
  - Bounded waiting
- **Disadvantages**
  - Cylinders in the middle get better service
  - Might miss locality SPTF could exploit
- **CSCAN: Only sweep in one direction**
  Very commonly used algorithm in Unix
- **Also called LOOK/CLOOK in textbook**

CSCAN example



VSCAN(r)

- **Continuum between SPTF and SCAN**
  - Like SPTF, but slightly changes "effective" positioning time

    If request in same direction as previous seek: $T_{\mathrm{eff}} = T_{\mathrm{pos}}$

    Otherwise: $T_{\mathrm{eff}} = T_{\mathrm{pos}} + r \cdot T_{\mathrm{max}}$
  - when r = 0, get SPTF, when r = 1, get SCAN
  - E.g., r = 0.2 works well
- **Advantages and disadvantages**
  - Those of SPTF and SCAN, depending on how r is set

Flash memory

- **Today, people increasingly using flash memory**
- **Completely solid state (no moving parts)**
  - Remembers data by storing charge
  - Lower power consumption and heat
  - No mechanical seek times to worry about
- **Limited # overwrites possible**
  - Blocks wear out after 10,000 (MLC) – 100,000 (SLC) erases
  - Requires *flash translation layer* (FTL) to provide *wear leveling*, so repeated writes to logical block don't wear out physical block
  - FTL can seriously impact performance
  - In particular, random writes *very* expensive [Birrell]
- **Limited durability**
  - Charge wears out over time
  - Turn off device for a year, you can easily lose data

Types of flash memory

- **NAND flash (most prevalent for storage)**
  - Higher density (most used for storage)
  - Faster erase and write
  - More errors internally, so need error correction
- **NOR flash**
  - Faster reads in smaller data units
  - Can execute code straight out of NOR flash
  - Significantly slower erases
- **Single-level cell (SLC) vs. Multi-level cell (MLC)**
  - MLC encodes multiple bits in voltage level
  - MLS slower to write than SLC

NAND Flash Overview

- **Flash device has 2112-byte *pages***
  - 2048 bytes of data + 64 bytes metadata & ECC
- ***Blocks* contain 64 (SLC) or 128 (MLC) pages**
- **Blocks divided into 2–4 *planes***
  - All planes contend for same package pins
  - But can access their blocks in parallel to overlap latencies
- **Can *read* one page at a time**
  - Takes 25 μs + time to get data off chip
- **Must *erase* whole block before *programming***
  - Erase sets all bits to 1—very expensive (2 msec)
  - Programming pre-erased block requires moving data to internal buffer, then 200 (SLC)–800 (MLC) μs

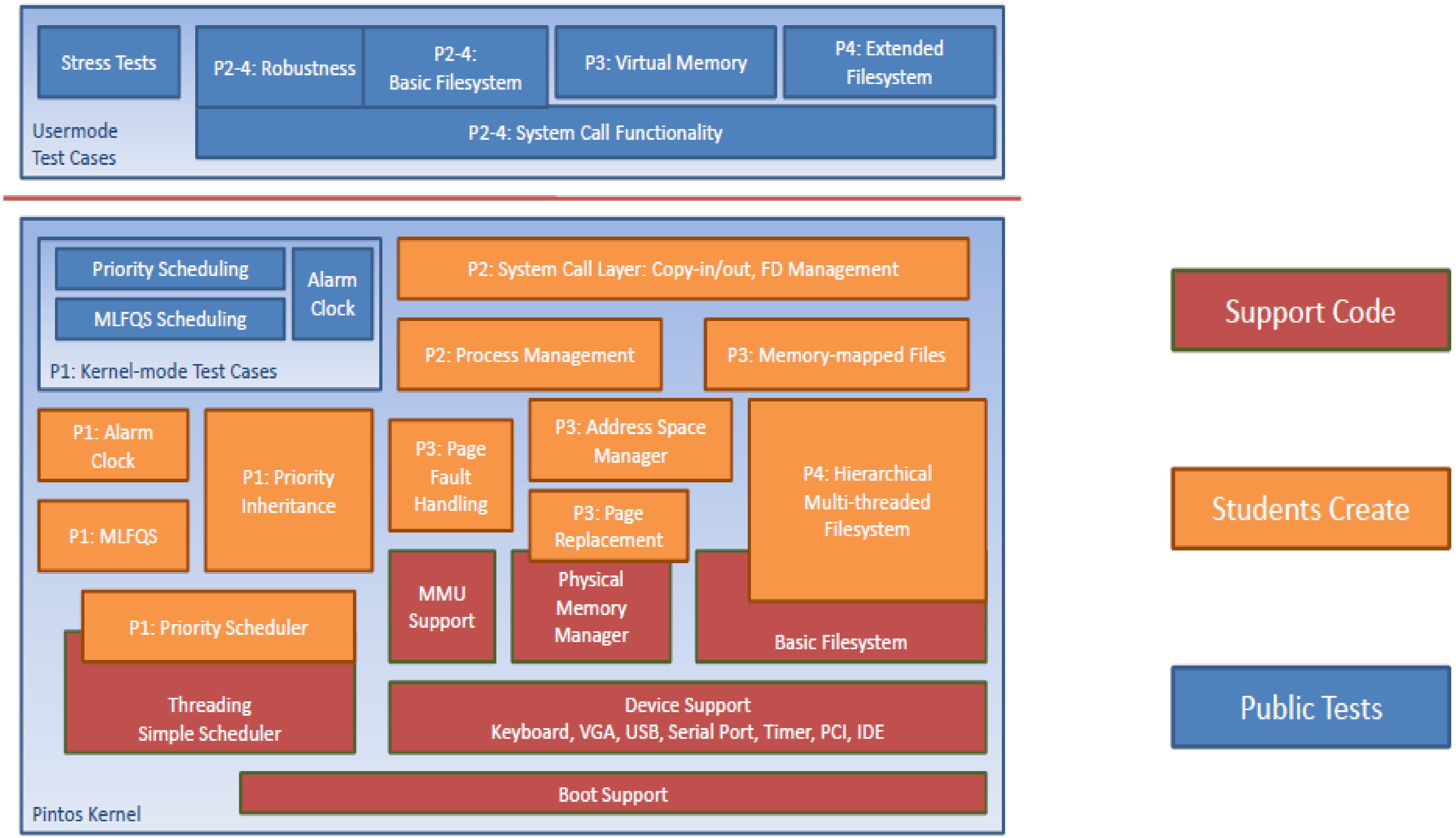
Flash Characteristics [Caulfield]

Parameter	SLC	MLC
Density Per Die (GB)	4	8
Page Size (Bytes)	2048+32	2048+64
Block Size (Pages)	64	128
Read Latency (μs)	25	25
Write Latency (μs)	200	800
Erase Latency (μs)	2000	2000
40MHz, 16-bit bus Read b/w (MB/s)	75.8	75.8
Program b/w (MB/s)	20.1	5.0
133MHz Read b/w (MB/s)	126.4	126.4
Program b/w (MB/s)	20.1	5.0

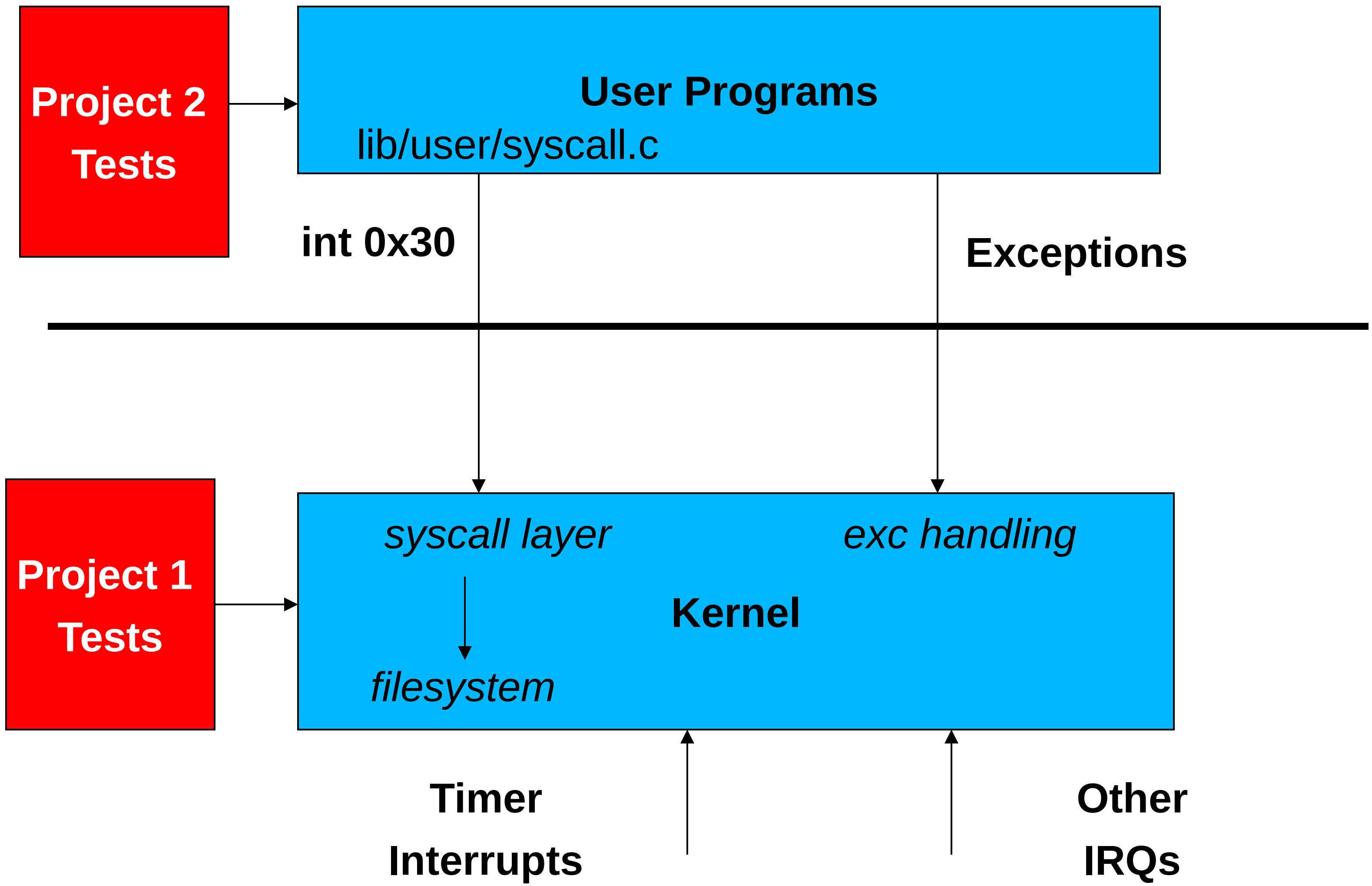
Summary

- Read Ch. 10 – Mass-Storage Structure
- Project 2
  - TYPO: `power_off()` should be `shutdown_power_off()`.

P2: Project 2 – System Calls



Project 1 and Project 2



Using the File system

- Interfacing with the file system
- No need to modify the file system
- Certain limitations
  - No internal synchronization
  - File size fixed
  - File data allocated as a single extent
  - No subdirectories
  - File names limited to 14 chars
  - System crash might corrupt the file system
- Files to take a look at: ‘filesys.h’ & ‘file.h’

Some commands

- In **userprog/build**, create a new simulated disk
  - pintos-mkdisk fs.dsk --filesys-size=2
- Format the disk
  - pintos -f -q
  - This will only work after your disk is created and kernel is built!
- Copy the program "echo" onto the disk
  - pintos -p ../../examples/echo -a echo -- -q
- Run the program
  - pintos -q run ’echo x’
- All in a single command:

```
pintos --filesys-size=2 -p ../../examples/echo -a echo -- -f
   -q run 'echo x'
```

`pintos ... -v -- ..` – to from terminal without X11 server

- $ **make check** or **make grade** – builds the disk automatically
  - You can just copy & paste the commands that make check does!

Various directories

- Few user programs:
  - src/examples
- Relevant files:
  - userprog/
- Other files:
  - threads/

Project 2 Requirements

- Process Termination Messages
- Argument Passing
- System Calls
- Deny writes to executables

1. Process Termination

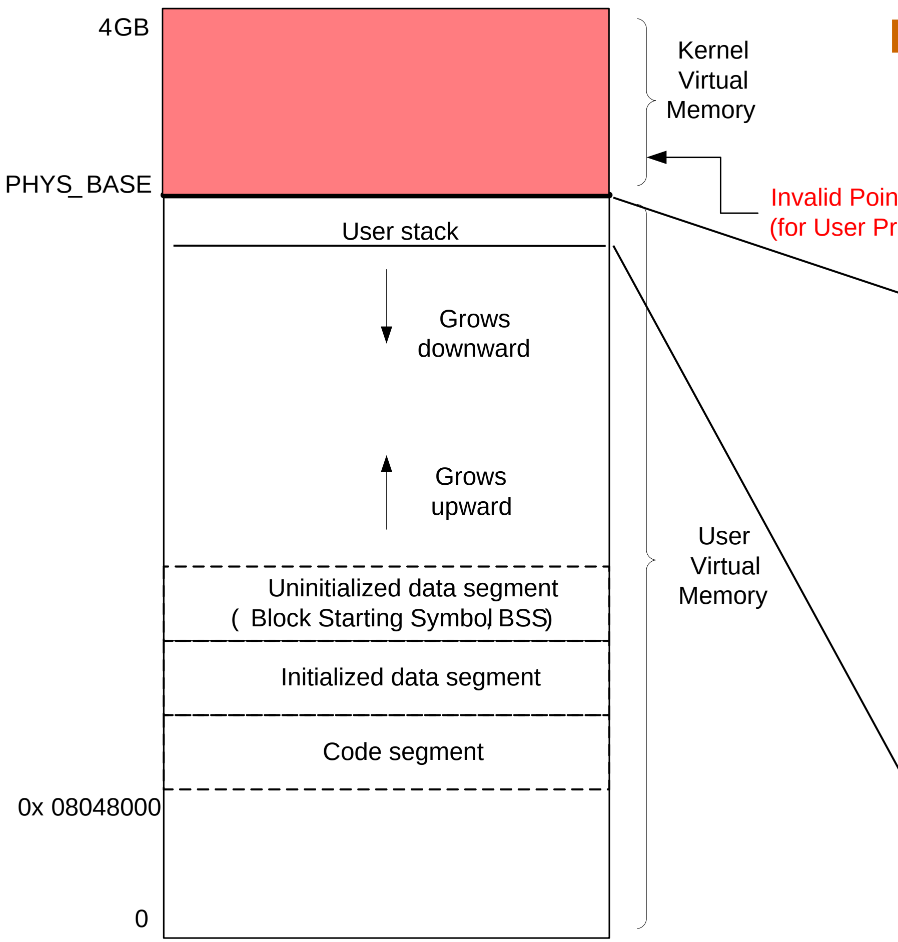
- When a Process Terminates
  - printf ("%s: exit(%d)\n", ...);
- Name: Full name passed to process_execute() in process.c
- Exit Code
- Do not print any other message!

2. Argument Passing

- No support currently for argument passing
- Change *esp = PHYS_BASE to *esp = PHYS_BASE – 12 in setup_stack() to get started
- Change process_execute() to process multiple arguments
- Can limit the arguments to fit in a page(4 kb)
- String Parsing: strtok_r() in lib/string.h

```
pgm.c
main(int argc,
         char *argv[]) {
         …
   }
$ pintos run ‘pgm alpha beta’
argc = 3
argv[0] = “pgm”
argv[1] = “alpha”
argv[2] = “beta”
```

Memory Layout



PHYS_BASE = 3GB

Address	Name	Data	Type
0xbffffffc	argv[3][...]	'bar\0'	char[4]
0xbffffff8	argv[2][...]	'foo\0'	char[4]
0xbffffff5	argv[1][...]	'-l\0'	char[3]
0xbfffffed	argv[0][...]	'/bin/ls\0'	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

Setting up the Stack

How to setup the stack for the program - /bin/ls –l foo bar

Address	Name	Data	Type
0xbffffffc	argv[3][...]	'bar\0'	char[4]
0xbffffff8	argv[2][...]	'foo\0'	char[4]
0xbffffff5	argv[1][...]	'-l\0'	char[3]
0xbfffffed	argv[0][...]	'/bin/ls\0'	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

3. System Calls – already discussed some; e.g. open( ) system call

- No Support for system calls currently
- Implement the system call handler in userprog/syscall.c
- System call numbers defined in lib/syscall-nr.h
- Process Control: exit, exec, wait
- File system: create, remove, open, filesize, read, write, seek, tell, close
- Others: halt

System Call Details

- Types of Interrupts – External and Internal
- System calls – Internal Interrupts or Software Exceptions
- 80x86 – ‘int’ instruction to invoke system calls
- Pintos – ‘int $0x30’ to invoke system call

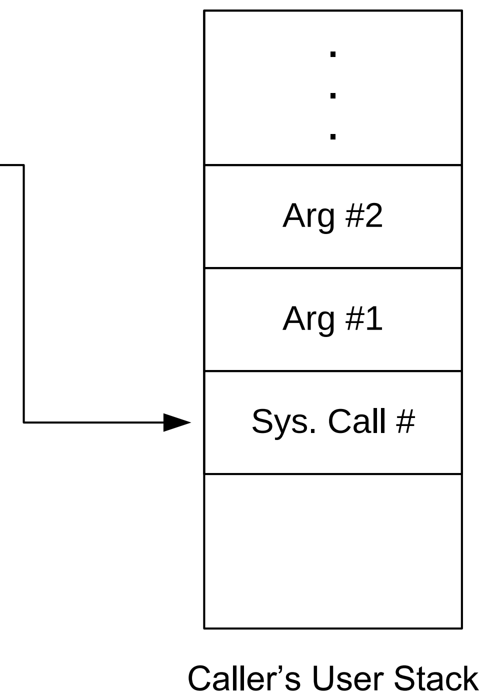
Continued…

- A system call has:
  - System call number
  - (possibly) arguments
- When syscall_handler() gets control:

```
syscall_handler (struct intr_frame *f) {

f->esp
….

f->eax = … ;
}
```



- System calls that return a value () must modify f->eax

System calls – File system

- Decide on how to implement the file descriptors – keep it simple -- O(n) data structures will entail no deduction
- Access granularity is the entire file system – add one global lock
- write() – fd 1 writes to console – use putbuf() to write entire buffer to console
- read() – fd 0 reads from console – use input_getc() to get input from keyboard
- Implement the rest of the system calls

System calls – Process Control

- wait(pid) – Waits for process pid to die and returns the status pid returned from exit
- Returns -1 if
  - pid was terminated by the kernel
  - pid does not refer to child of the calling thread
  - wait() has already been called for the given pid
- exec(cmd) – runs the executable whose name is given in command line
  - returns -1 if the program cannot be loaded
- exit(status) – terminates the current user program, returns status
  - status of 0 indicates success, non zero otherwise

Process Control – continued…

- Parent may or may not wait for its child
- Parent may call wait() after child terminates!
- Implement process_wait() in process.c
- Then, implement wait() in terms of process_wait()
- Cond variables and/or semaphores will help
  - Think about what semaphores may be used for and how they must be initialized

```
main() {
int i; pid_t p;
p = exec("pgm a b");
i = wait (p);
... /* i must be 5 */
}
```

```
main() {
int status;
... status = 5;
exit(status);
}
```

pgm.c

Memory Access

- Invalid pointers must be rejected. Why?
  - Kernel has access to all of physical memory including that of other processes
  - Kernel like user process would fault when it tries to access unmapped addresses
- User process cannot access kernel virtual memory
- User Process after it has entered the kernel can access kernel virtual memory and user virtual memory
- How to handle invalid memory access?

Memory Access – contd…

- Two methods to handle invalid memory access
  - Verify the validity of user provided pointer and then dereference it
    - Look at functions in userprog/pagedir.c, threads/vaddr.h
    - Strongly recommended!
  - Check if user pointer is below PHYS_BASE and dereference it
    - Could cause page fault
    - Handle the page fault by modifying the page_fault() code in userprog/exception.c
  - Make sure that resources are not leaked

Some Issues to look at…

- Check the validity of the system call parameters
- Every single location should be checked for validity before accessing it; e.g. not only for f->esp, but also the locations f->esp +1, f->esp+2 and f->esp+3 should be checked
- Read system call parameters into kernel memory (except for long buffers) – write a copy_in function for this purpose.

Denying writes to Executables

- Use file_deny_write() to prevent writes to an open file
- Use file_allow_write() to re-enable write
- Closing a file will re-enable writes

Suggested Order of Implementation

- Implement the system call infrastructure
- Change process_wait() to a infinite loop to prevent pintos getting powered off before the process gets executed
- Implement exit system call
- Implement write system call
- Start making other changes

Pintos Project 2 Sample Test

- Example Open System Call
- Test: tests/userprog/open-normal.c

```
/* Open a file. */
#include <syscall.h>
#include "tests/lib.h"
#include "tests/main.h"

void
test_main (void)
{
  int handle = open ("sample.txt");
  if (handle < 2)
    fail ("open() returned %d", handle);
}
```

userprog/ - make check

```
gcc -c ../../tests/userprog/open-normal.c -o tests/userprog/open-normal.o
 -g -msoft-float -O -fno-stack-protector -nostdinc -I../.. -I../../lib
 -I../../lib/user -I. -Wall -W -Wstrict-prototypes -Wmissing-prototypes
 -Wsystem-headers  -MMD -MF tests/userprog/open-normal.d
gcc  -Wl,--build-id=none -nostdlib -static -Wl,-T,../../lib/user/user.lds
 tests/userprog/open-normal.o tests/main.o tests/lib.o lib/user/entry.o
 libc.a -o tests/userprog/open-normal
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/open-normal
 -a open-normal -p ../../tests/userprog/sample.txt -a sample.txt -- -q
 -f run open-normal < /dev/null 2> tests/userprog/open-normal.errors
 > tests/userprog/open-normal.output
perl -I../.. ../../tests/userprog/open-normal.ck
 tests/userprog/open-normal tests/userprog/open-normal.result
pass tests/userprog/open-normal
----------------------------------------------------------------------
ar r libc.a lib/debug.o lib/random.o lib/stdio.o lib/stdlib.o lib/string.o
 lib/arithmetic.o lib/ustar.o lib/user/debug.o lib/user/syscall.o
 lib/user/console.o
ranlib libc.a
```

$ objdump -D open-normal

```
080480a0 <test_main>:
 80480a0:       55                      push   %ebp
 80480a1:       89 e5                   mov    %esp,%ebp
 80480a3:       83 ec 18                sub    $0x18,%esp
 80480a6:       c7 04 24 6a a7 04 08    movl   $0x804a76a,(%esp)
 80480ad:       e8 1f 21 00 00          call   804a1d1 <open>
 80480b2:       83 f8 01                cmp    $0x1,%eax
..
0804a1d1 <open>:
 804a1d1:       55                      push   %ebp
 804a1d2:       89 e5                   mov    %esp,%ebp
 804a1d4:       ff 75 08                pushl  0x8(%ebp)
 804a1d7:       6a 06                   push   $0x6
 804a1d9:       cd 30                   int    $0x30
 804a1db:       83 c4 08                add    $0x8,%esp
 804a1de:       5d                      pop    %ebp
 804a1df:       c3                      ret
..
0804a76a <.rodata.str1.1>:
 804a76a:       73 61 6d 70 6c 65 2e 74 78 74 00
                 s  a  m  p  l  e  .  t  x  t
```

src/userprog/syscall.c

```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler,
   "syscall");
  lock_init (&fs_lock);
}
```

src/userprog/syscall.c

```
/* System call handler. */
static void
syscall_handler (struct intr_frame *f)
{
  typedef int syscall_function (int, int, int);
  /* A system call. */
  struct syscall
    {
      size_t arg_cnt;           /* Number of arguments. */
      syscall_function *func;   /* Implementation. */
    };
  /* Table of system calls. */
  static const struct syscall syscall_table[] =
    {
      {0, (syscall_function *) sys_halt},
      {1, (syscall_function *) sys_exit},
      {1, (syscall_function *) sys_exec},
      {1, (syscall_function *) sys_wait},
      {2, (syscall_function *) sys_create},
      {1, (syscall_function *) sys_remove},
      {1, (syscall_function *) sys_open},  <-- call number 6
```

src/userprog/syscall.c

```
 ...
  const struct syscall *sc;
  unsigned call_nr;
  int args[3];

  /* Get the system call. */
  copy_in (&call_nr, f->esp, sizeof call_nr);
  if (call_nr >= sizeof syscall_table / sizeof *syscall_table)
    thread_exit ();
  sc = syscall_table + call_nr;

  /* Get the system call arguments. */
  ASSERT (sc->arg_cnt <= sizeof args / sizeof *args);
  memset (args, 0, sizeof args);
  copy_in (args, (uint32_t *) f->esp + 1, sizeof *args * sc->arg_cnt);

  /* Execute the system call,
     and set the return value. */
  f->eax = sc->func (args[0], args[1], args[2]);
}
```

src/userprog/syscall.c

```
/* Open system call. */
static int
sys_open (const char *ufile)
{
  char *kfile = copy_in_string (ufile);
  struct file_descriptor *fd;
  int handle = -1;

  fd = malloc (sizeof *fd);
  if (fd != NULL)
    {
      lock_acquire (&fs_lock);
      fd->file = filesys_open (kfile);
      if (fd->file != NULL)
...
```

Copy a byte from user space to kernel space

```
/* Copies a byte from user address USRC to kernel address DST. */
static inline bool
get_user (uint8_t *dst, const uint8_t *usrc)
{
  int eax;
  asm ("movl $1f, %%eax; movb %2, %%al; movb %%al, %0; 1:"
       : "=m" (*dst), "=&a" (eax) : "m" (*usrc));
  return eax != 0;
}
/* Writes BYTE to user address UDST. */
static inline bool
put_user (uint8_t *udst, uint8_t byte)
{
  int eax;
  asm ("movl $1f, %%eax; movb %b2, %0; 1:"
       : "=m" (*udst), "=&a" (eax) : "q" (byte));
  return eax != 0;
}
```

Copy a string from user space to kernel space

```
static char *
copy_in_string (const char *us)
{
  char *ks;
  size_t length;
  ks = palloc_get_page (0);
  if (ks == NULL)
    thread_exit ();
    for (length = 0; length < PGSIZE; length++)
    {
      if (us >= (char *) PHYS_BASE || !get_user (ks + length, us++))
      {
          palloc_free_page (ks);
          thread_exit ();
      }
      if (ks[length] == '\0')
        return ks;
    }
  ks[PGSIZE - 1] = '\0';
  return ks;
}
```

Run a single test

- From userprog/build:
  - **`rm tests/userprog/open-normal.output`**
  - **`make test/userprog/open-normal.output`**

```
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/open-normal
-a open-normal -p ../../tests/userprog/sample.txt -a sample.txt -- -q  -f
run open-normal < /dev/null 2> tests/userprog/open-normal.errors >
tests/userprog/open-normal.output
```

examples/shell.c

```
int main (void)
{
  printf ("Shell starting..\n");
  for (;;)
    {
      char command[80];
      /* Read command. */
      printf ("--");
      read_line (command, sizeof command);

      /* Execute command. */
      if (!strcmp (command, "exit"))
        break;
      else if (!memcmp (command, "cd ", 3))
        {
          if (!chdir (command + 3))
            printf ("\"%s\": chdir failed\n", command + 3);
        }
      else if (command[0] == '\0')
```

examples/shell.c

```
else if (command[0] == '\0')
        {
          /* Empty command. */
        }
      else
        {
          pid_t pid = exec (command);
          if (pid != PID_ERROR)
            printf ("\"%s\": exit code %d\n", command, wait (pid));
          else
            printf ("exec failed\n");
        }
    }

  printf ("Shell exiting.");
  return EXIT_SUCCESS;
}
```

$ objdump -D examples/shell

```
080480cc <main>:
 80480cc:       55                      push   %ebp
 80480cd:       89 e5                   mov    %esp,%ebp
 80480cf:       83 e4 f0                and    $0xfffffff0,%esp
 80480d2:       57                      push   %edi
 80480d3:       56                      push   %esi
 80480d4:       53                      push   %ebx
 80480d5:       83 ec 74                sub    $0x74,%esp
 80480d8:       c7 04 24 3e a1 04 08    movl   $0x804a13e,(%esp)
 80480df:       e8 23 1d 00 00          call   8049e07 <puts>
 80480e4:       c7 04 24 5a a1 04 08    movl   $0x804a15a,(%esp)
 80480eb:       e8 2f 0c 00 00          call   8048d1f <printf>
 80480f0:       8d 44 24 18             lea    0x18(%esp),%eax
 80480f4:       89 44 24 68             mov    %eax,0x68(%esp)
 80480f8:       89 c3                   mov    %eax,%ebx
 80480fa:       c7 44 24 08 01 00 00    movl   $0x1,0x8(%esp)
 8048101:       00
 8048102:       8d 44 24 6f             lea    0x6f(%esp),%eax
 8048106:       89 44 24 04             mov    %eax,0x4(%esp)
 804810a:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048111:       e8 a9 1a 00 00          call   8049bbf <read>
```

$ objdump -D examples/shell

```
..
 80481f1:       80 7c 24 18 00          cmpb   $0x0,0x18(%esp)
 80481f6:       0f 84 e8 fe ff ff       je     80480e4 <main+0x18>
 80481fc:       8d 44 24 18             lea    0x18(%esp),%eax
 8048200:       89 04 24                mov    %eax,(%esp)
 8048203:       e8 50 19 00 00          call   8049b58 <exec>
..
08049bbf <read>:
 8049bbf:       55                      push   %ebp
 8049bc0:       89 e5                   mov    %esp,%ebp
 8049bc2:       ff 75 10                pushl  0x10(%ebp)
 8049bc5:       ff 75 0c                pushl  0xc(%ebp)
 8049bc8:       ff 75 08                pushl  0x8(%ebp)
 8049bcb:       6a 08                   push   $0x8
 8049bcd:       cd 30                   int    $0x30
 8049bcf:       83 c4 10                add    $0x10,%esp
 8049bd2:       5d                      pop    %ebp
 8049bd3:       c3                      ret
```

syscall exec() calls process_execute

```
/* Starts a new thread running a user program loaded from
   FILENAME.  The new thread may be scheduled (and may even exit)
   before process_execute() returns.  Returns the new process's
   thread id, or TID_ERROR if the thread cannot be created. */
tid_t process_execute (const char *file_name)
{
  struct exec_info exec;
  char thread_name[15];
  char *save_ptr;
  tid_t tid;
  /* Initialize exec_info. */
  exec.file_name = file_name;
  sema_init (&exec.load_done, 0);
  /* Create a new thread to execute FILE_NAME. */
  strlcpy (thread_name, file_name, sizeof thread_name);
  strtok_r (thread_name, " ", &save_ptr);
  tid = thread_create (thread_name, PRI_DEFAULT, start_process, &exec);
```

process_execute

```
/* Starts a new thread running a user program loaded from
   FILENAME.  The new thread may be scheduled (and may even exit)
   before process_execute() returns.  Returns the new process's
   thread id, or TID_ERROR if the thread cannot be created. */
tid_t process_execute (const char *file_name)
{
..

  if (tid != TID_ERROR)
  {
      sema_down (&exec.load_done);
      if (exec.success)
        list_push_back (&thread_current ()->children,
                        &exec.wait_status->elem);
      else
        tid = TID_ERROR;
  }
  return tid;
}
```

start_process

```
static void start_process (void *exec_)
{
  struct exec_info *exec = exec_;
  struct intr_frame if_;
  bool success;

  /* Initialize interrupt frame and load executable. */
  memset (&if_, 0, sizeof if_);
  if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
  if_.cs = SEL_UCSEG;
  if_.eflags = FLAG_IF | FLAG_MBS;
  success = load (exec->file_name, &if_.eip, &if_.esp);

  /* Allocate wait_status. */
  if (success)
  {
      exec->wait_status = thread_current ()->wait_status
        = malloc (sizeof *exec->wait_status);
      success = exec->wait_status != NULL;
  }
```

start_process

```
/* Initialize wait_status. */
 if (success)
 {
     ..
 }
 /* Notify parent thread and clean up. */
 exec->success = success;
 sema_up (&exec->load_done);
 if (!success)
   thread_exit ();
 /* Start the user process by simulating a return from an
    interrupt, implemented by intr_exit (in
    threads/intr-stubs.S).  Because intr_exit takes all of its
    arguments on the stack in the form of a `struct intr_frame',
    we just point the stack pointer (%esp) to our stack frame
    and jump to it. */
 asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
}
```

Cycle Counters

- Most modern systems have built in registers that are incremented every clock cycle
  - Very fine grained
  - Maintained as part of process state
    - In Linux, counts elapsed global time
- Special assembly code instruction to access
- On (recent model) Intel machines:
  - 64 bit counter.
  - RDTSC instruction sets `%edx` to high order 32-bits, `%eax` to low order 32-bits
- Aside: Is this a security issue?

Cycle Counter Period

- Wrap Around Times for 550 MHz machine
  - Low order 32 bits wrap around every $2^{32} / (550 * 10^6) =$ 7.8 seconds
  - High order 64 bits wrap around every $2^{64} / (550 * 10^6) =$ 33539534679 seconds
    - 1065 years
- For 2 GHz machine
  - Low order 32 bits every 2.1 seconds
  - High order 64 bits every 293 years

Measuring with Cycle Counter

- Idea
  - Get current value of cycle counter
    - store as pair of unsigned's `cyc_hi` and `cyc_lo`
  - Compute something
  - Get new value of cycle counter
  - Perform double precision subtraction to get elapsed cycles

```
/* Keep track of most recent reading of cycle counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

void start_counter()
{
  /* Get current value of cycle counter */
  access_counter(&cyc_hi, &cyc_lo);
}
```

Accessing the Cycle Counter

- GCC allows inline assembly code with mechanism for matching registers with program variables
- Code only works on x86 machine compiling with GCC

```
void access_counter(unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Emit assembly with `rdtsc` and two `movl` instructions

Closer Look at Extended ASM

```
asm("Instruction String"
      : Output List
      : Input List
      : Clobbers List);
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Instruction String
  - Series of assembly commands
    - Separated by "`;`" or "`\n`"
    - Use "`%%`" where normally would use "`%`"

Closer Look at Extended ASM

```
asm("Instruction String"
      : Output List
      : Input List
      : Clobbers List
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Output List
  - Expressions indicating destinations for values `%0`, `%1`, …, `%j`
    - Enclosed in parentheses
    - Must be *lvalue*
      - Value that can appear on LHS of assignment
  - Tag `"=r"` indicates that symbolic value (`%0`, etc.), should be replaced by a register

Closer Look at Extended ASM

```
asm("Instruction String"
        : Output List
        : Input List
        : Clobbers List)
}
```

```
void access_counter
   (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Input List
  - Series of expressions indicating sources for values `%j+1`, `%j+2`, …
    - Enclosed in parentheses
    - Any expression returning value
  - Tag `"r"` indicates that symbolic value (`%0`, etc.) will come from register

Closer Look at Extended ASM

```
asm("Instruction String"
       : Output List
       : Input List
       : Clobbers List);
}
```

```
void access_counter
   (unsigned *hi, unsigned *lo)
{
   /* Get cycle counter */
   asm("rdtsc; movl %%edx,%0; movl %%eax,%1
       : "=r" (*hi), "=r" (*lo)
       : /* No input */
       : "%edx", "%eax");
}
```

- Clobbers List
  - List of register names that get altered by assembly instruction
  - Compiler will make sure doesn't store something in one of these registers that must be preserved across `asm`
    - Value set before & used after

Completing Measurement

- Get new value of cycle counter
- Perform double precision subtraction to get elapsed cycles
- Express as `double` to avoid overflow problems

```
double get_counter()
{
  unsigned ncyc_hi, ncyc_lo
  unsigned hi, lo, borrow;
  /* Get cycle counter */
  access_counter(&ncyc_hi, &ncyc_lo);
  /* Do double precision subtraction */
  lo = ncyc_lo - cyc_lo;
  borrow = lo > ncyc_lo;
  hi = ncyc_hi - cyc_hi - borrow;
  return (double) hi * (1 << 30) * 4 + lo;
}
```

Timing With Cycle Counter

- Determine Clock Rate of Processor
  - Count number of cycles required for some fixed number of seconds

```
double MHZ;
int sleep_time = 10;
start_counter();
sleep(sleep_time);
MHZ = get_counter()/(sleep_time * 1e6);
```

- Time Function P( )
  - First attempt: Simply count cycles for one execution of P

```
double tsecs;
start_counter();
P();
tsecs = get_counter() / (MHZ * 1e6);
```

Example – testClock.c

```
#include <stdio.h>
#include "clock.h"


int main()
{

  double cycles, Mhz;

  Mhz = mhz(1);
  start_counter();
  sleep(2);
  cycles = get_counter();

  printf("cycles = %f, MHz = %f, cycles/Mhz = %f\n", cycles, Mhz,
 cycles/Mhz);
  printf("elapsed time = %f seconds \n", cycles/(1.0e6*Mhz));

  return 0;
}
```

```
Processor Clock Rate ~= 2673.5 MHz
cycles = 5343976388.000000, MHz = 2673.526339, cycles/Mhz = 1998849.351153
elapsed time = 1.998849 seconds
```

Measurement Pitfalls

- Overhead
  - Calling `get_counter()` incurs small amount of overhead
  - Want to measure long enough code sequence to compensate

Summary

- Read Ch. 1-10
- Processes and Threads (Ch. 4)
- Process Scheduling (Ch. 5)
- Synchronization (Ch. 6)
- Deadlock (Ch. 7)
- Memory Management (Ch. 8)
- Virtual Memory (Ch. 9)
- Mass-Storage Structure (Ch. 10)
- Project #2 – System Calls and User-Level Processes