Lecture 11: Quiz #1 Review

Instructor: Mitch Neilsen

Office: N219D

Outline

- Reading:
  - Ch. 1-7 - Process/Thread Management - Quiz #1 – 10/9
  - Ch. 8 - Memory Management
  - Ch. 2 - Systems Structures and System Call Interface
- Homework 2: Due 10/2
- Project 1: Scheduling/Synchronization: **New Due Date 10/7**
  - Alarm Clock (finish this week)
  - Priority-based Scheduler (finish this week)
  - Synchronization and Priority Inheritance (start this week)
  - [Extra Credit] MLFQ Scheduler
- **Quiz #1 – 10/9**

Quote of the Day

"Research is to see what everybody else has seen, and to think what nobody else has thought."

-- Albert Szent-Gyoergi

Quiz #1

- **Chapters 1-7**
  - Introduction
  - Operating System Structures
  - Processes
  - Threads
  - CPU Scheduling
  - Process Synchronization
  - Deadlocks

Operating Systems - Introduction

Operating Systems make hardware useful to the programmer by providing:

- Well-defined interface to the hardware (syscalls)
- Multitasking capability (for higher resource utilization)
- Protection against resource gluttons (pre-emption) and against malicious users (interposition, separation of privileges)
- Important theme: throughput vs. fairness

Processes

- What is in a process control block?
  - Page directory (defines its virtual address space)
  - Saved registers
  - File descriptors
  - Process ID
  - Priority
  - Pointer to the PCB of the next process to run
  - Accounting information (e.g. recent CPU time)
- What happens on a context switch?
  - Save state into PCB (registers, address spaces...)
  - Reload state from PCB

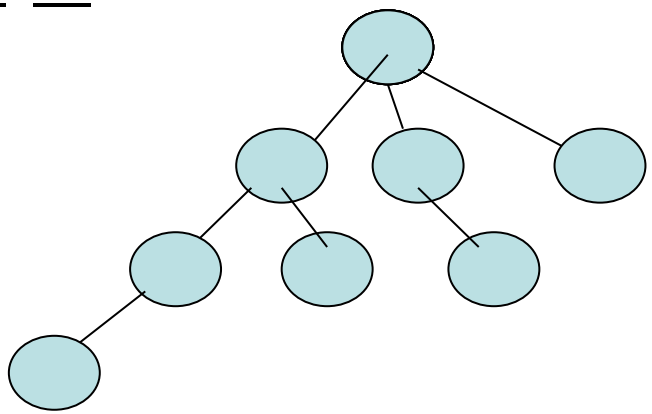
Example Process Question

Including the initial parent process, how many processes are created when the following program is executed? ____. Draw the Process Model.

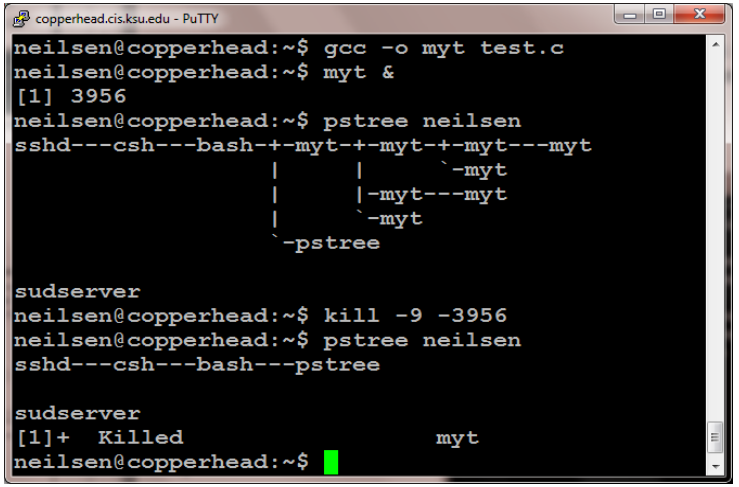
```
#include <stdio.h>
#include <unistd.h>
int main()
{
    /* fork off a child process */
    fork();
    /* fork off another child process */
    fork();
    /* and another */
    fork();
    /* wait for a signal to be received */
    pause();
}
```

Example Process Question

Including the initial parent process, how many processes are created when the following program is executed? _8__. Draw the Process Model.



```
#include <stdio.h>
#include <unistd.h>
int main()
{
    /* fork off a child process */
    fork();
    /* fork off another child process */
    fork();
    /* and another */
    fork();
    /* wait for a signal to be received */
    pause();
}
```



Threads

- Thread: "schedulable execution context"
- Kernel threads (1:1)
  - (+) Can take advantage of multiprocessor
  - (-) Thread operations go through kernel
  - (-) Heavy-weight structures
- User-level threads (n:1)
  - (+) Implemented in a user-level library
  - (+) Simple representation
  - (-) Can't take advantage of multiple CPUs
  - (-) One blocking syscall blocks all threads
- n:m threads
  - Need to coordinate between multiple schedulers

Sample Thread Question

Assume that Pintos's thread_set_priority() function is extended such that it takes an argument of type `tid_t`. `thread_set_priority(t, p)` sets the priority of thread with thread id `t` to `p`.

```
#define HIGH_PRIORITY            PRI_DEFAULT - 5
#define MEDIUM_PRIORITY          PRI_DEFAULT
#define LOW_PRIORITY             PRI_DEFAULT + 5

void printer(void *name)
{
  while (1)
    printf("%s", name);
}

int main()
{
  tid_t t[4];
  int i, c;

  ASSERT (!enable_mlfqs);

  thread_set_priority(thread_current()->tid, HIGH_PRIORITY);
  t[0] = thread_create("a-thread", LOW_PRIORITY, printer, "A");
  t[1] = thread_create("b-thread", LOW_PRIORITY, printer, "B");
  t[2] = thread_create("c-thread", LOW_PRIORITY, printer, "C");
  t[3] = thread_create("d-thread", LOW_PRIORITY, printer, "D");

  c = 0;
  for (i = 0; i < 10; i++)
    {
      thread_set_priority (t[c], MEDIUM_PRIORITY);
      timer_sleep (TIMER_FREQ);
      thread_set_priority (t[c], LOW_PRIORITY);
      c = (c + 1) % 4;
    }
}
```

a) (8 pts) What would this kernel output when run, assuming that strict priority scheduling is used?

Sample Thread Question

```
  thread_set_priority(thread_current()->tid, HIGH_PRIORITY);
  t[0] = thread_create("a-thread", LOW_PRIORITY, printer, "A");
  t[1] = thread_create("b-thread", LOW_PRIORITY, printer, "B");
  t[2] = thread_create("c-thread", LOW_PRIORITY, printer, "C");
  t[3] = thread_create("d-thread", LOW_PRIORITY, printer, "D");

  c = 0;
  for (i = 0; i < 10; i++)
    {
      thread_set_priority (t[c], MEDIUM_PRIORITY);
      timer_sleep (TIMER_FREQ);
      thread_set_priority (t[c], LOW_PRIORITY);
      c = (c + 1) % 4;
    }
}
```

a) (8 pts) What would this kernel output when run, assuming that strict priority scheduling is used?

It would output
AAAAA…AAAAAABBBBBB…BBBBBBCCCCCC…CCCCCDDDDDD…DDDDDDD
AAAAA…AAAAAABBBBBB…BBBBBBCCCCCC…CCCCCDDDDDD…DDDDDDD
AAAAA…AAAAAABBBBBB…BBBBBB
switching 10 times between threads, once every second, until 10 seconds have passed.
[After that, either the kernel would shut down (if –q was given) or the threads would continue to run and output
AAAAA…AAAAAABBBBBB…BBBBBBCCCCCC…CCCCCDDDDDD…DDDDDDD… ad infinitum
switching every time slice.]

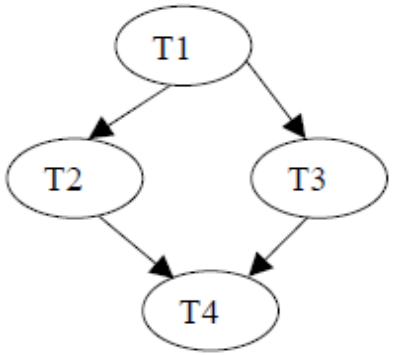
Synchronization

- Sequential consistency
  - Maintain program order on individual processors
  - Ensure write atomicity
  - Memory barriers (or fences) preserve instruction order
- Lots of synchronization primitives -- how to implement lower-level synchronization?
  - **Disabling interrupts** (only on uniprocessors)
  - **Spinlocks** (need atomic read-write instructions in hardware, like test_and_set or xchg)
- Higher-level synchronization
  - **Semaphores**
  - Monitors

Sample Synchronization Question

CS140 Midterm Fall 2005 Initials:_____

6. (10 points) Synchronization
Assume you are given a graph that represents the relationship between four threads (T1, T2, T3, T4). An arrow from one thread (Tx) to another (Ty) means that thread Tx must finish its computation before Ty starts. Assume that the threads can arrive in any order. Use semaphores to enforce this relationship specified by the graph. Be sure to show the initial values and the locations of the semaphore operations.



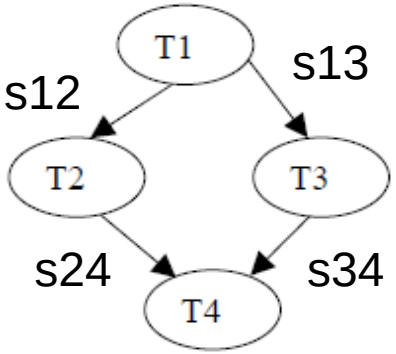

```
// Semaphores definitions and their initial values
```

```
void T1(void)
{
```

```
void T2(void)
{
```

```
void T3(void)
{
```

```
void T4(void)
{
```




```
// Semaphores definitions and their initial values

Semaphore s12 = 0;
Semaphore s13 = 0;
Semaphore s24 = 0;
Semaphore s34 = 0;
```

```
void T1(void)
{
  do work;
  signal(s12);
  signal(s13);
 }

// T1 computation
```

```
void T2(void)
{
  wait(s12);
  do work;
  signal(s24);
 }

// T2 computation
```

```
void T3(void)
{
  wait(s13);
  do work;
  signal(s34);
 }

// T3 computation
```

```
void T4(void)
{
  wait(s24);
  wait(s34);
  do work;
 }

// T4 computation
```

Barrier Synchronization

2. When programming scientific code with multiple threads, sometimes it is useful to have all threads rendezvous at a place in the code. This is normally done with an operation called a "barrier". It works so that when threads call the Barrier() function, none return until all threads have called the function. For example you might find code that looks like:

```
{
    ThreadsRunInParallel();
    // Threads may reach the barrier at different times

    Barrier(ThreadID());

    // Threads should start here together after the barrier
    ThreadsDoSomethingElseInParallel();
}
```

Your job is to write the **Barrier()** function using only semaphores. You are not permitted to use shared variables other than the semaphores. Be sure to show your initial values for the semaphore. You can assume there is a global constant NUM_THREADS that indicates the number of threads in the system. Thread IDs are integers starting at 0 and going to NUM_THREADS-1. NUM_THREADS is less than 100. To simplify things you can assume that **Barrier()** is only called once and all threads participate in the barrier.

Semaphore declaration and initialization:

```
void Barrier(int threadID) {



}
```

Scheduling

- Metrics? Throughput, turnaround time, response time
- Secondary: CPU utilization, waiting time
- Scheduling Algorithms
  - FCFS - simple, but many problematic cases, convoy effect: CPU-bound job holds processor, I/O devices left idle, Long job comes in before short job
  - SJF - minimize turnaround time, but might starve longer jobs; Also, needs to be able to predict the future
  - Round-robin - solves starvation and is responsive
  - Priority-based scheduling: tunable, can prevent starvation through aging

Sample Scheduling Question

- Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time (Burst Time) listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4
P3	1.0	1

- What is the average turnaround time for these processes with the FCFS scheduling algorithm?

- With the SJF Scheduling Algorithm?

Sample Scheduling Question

- Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time (Burst Time) listed. In answering the questions, use **nonpreemptive** scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4
P3	1.0	1

- What is the average turnaround time for these processes with the FCFS scheduling algorithm? Ave TT = ((8-0) + (12-0.4) + (13-1.0))/3 = 31.6/3 = 10.53.

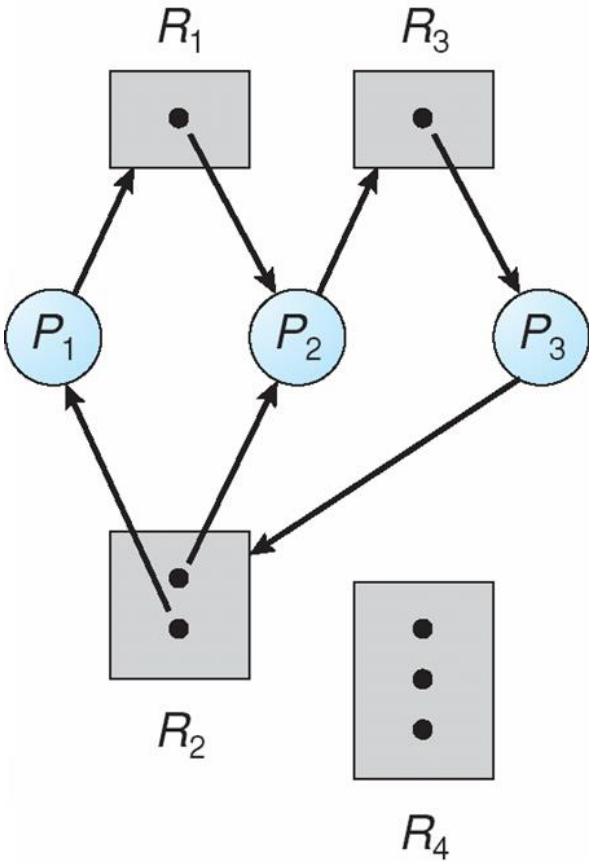
- With the SJF Scheduling Algorithm? AveTT = (8+(9-1)+(13-0.4)) = 28.6/3 = 9.53.

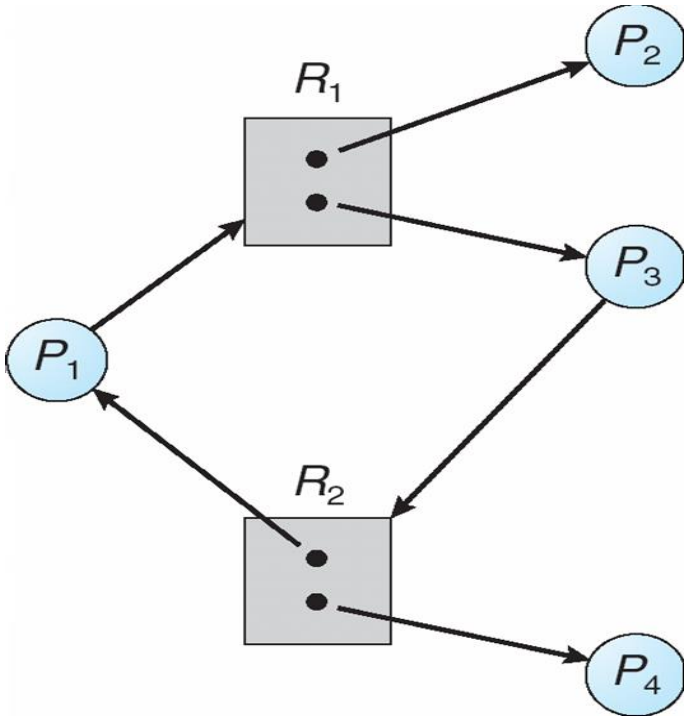
Deadlocks

- Four **required** conditions
  - mutual exclusion,
  - no preemption,
  - hold and wait, and
  - cycle in the graph of waiting processes
- Banker's Algorithm
  - safe vs. unsafe

Sample Deadlock Question

- Deadlocked? Explain briefly why or why not.





Another Problem

CS140 Midterm Winter 2007

1. (8 points) Assume you are building a special operating system that requires executing the expression:

   z = F3(F1(x), F2(F3(y)));

   where x, y, and z are integers and F1, F2, F3 are functions.

   The functions F1() and F3() must execute on thread T1 while the function F2 needs to execute on thread T2. Your job is to write the code to force the expression to be evaluated regardless of the order they are run by the CPU scheduler. Note you will have to add code to all three functions below.

   Shared Variables:

```
int z, x, y;

//Declare shared variables and semaphore with initial values here.
```

```
void ComputeZ() {
```

```
void T1() {
```

```
void T2() {
```

Shared Variables:

```
int z, x, y;
```

//Declare shared variables and semaphore with initial values here.

```
void ComputeZ() {

  StartThread(T1);

  StartThread(T2);

  return;
}
```

```
void T1() {
```

```
void T2() {
```

Another Problem (cont.)

CS140 Midterm Winter 2007

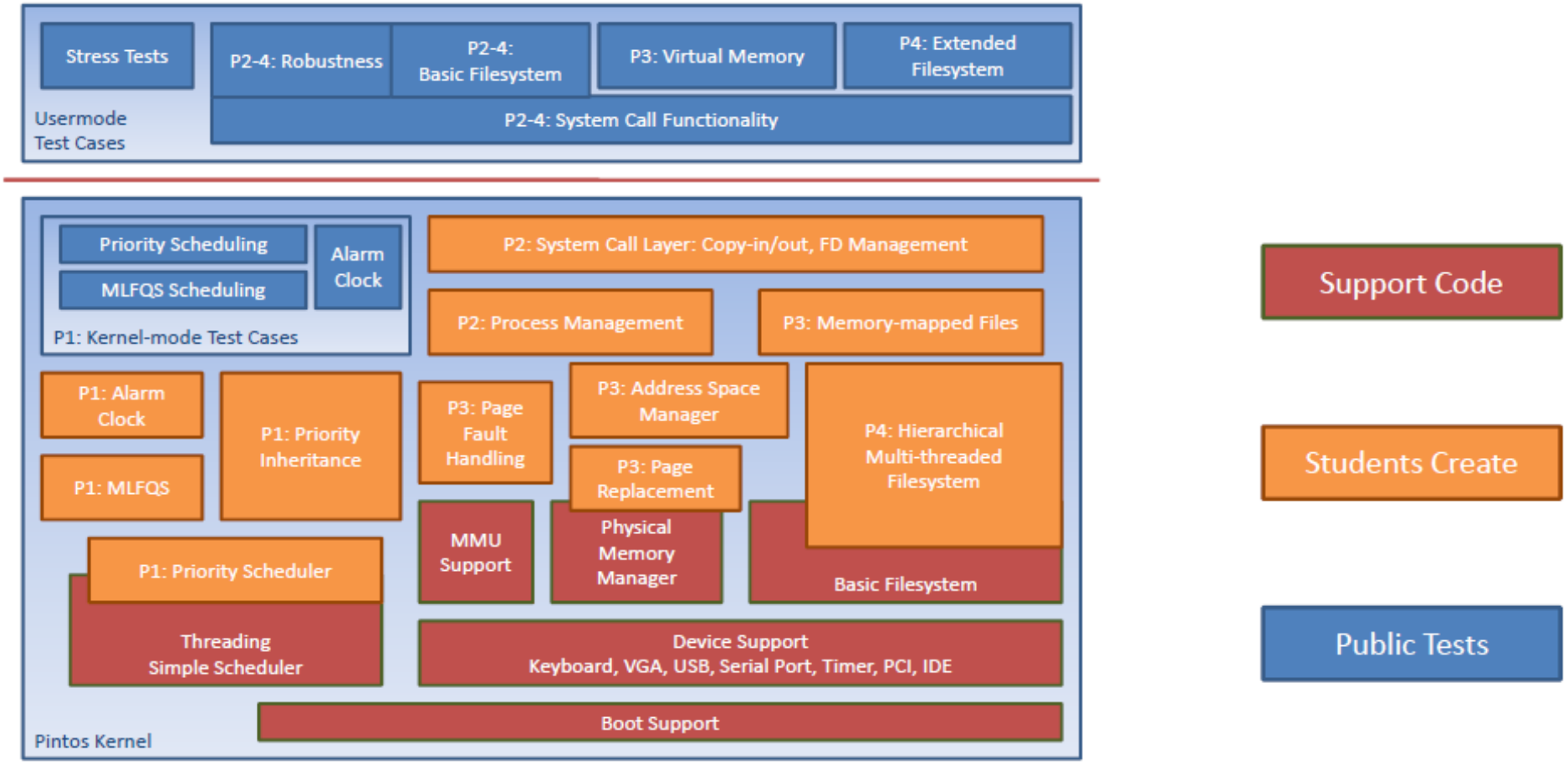
2. (6 points) Does your solution to Problem 1 handle the following cases? If so, explain why. If not, explain the problem and how you could fix it.

   (a) Would your solution handle the case when ComputeZ() is called twice in a row like:

```
x = …;   y = …;    // Set x and y arguments
ComputeZ();
printf("z = %d\n", z);
x = …;   y = …;    // Set x and y arguments
ComputeZ();
printf("z = %d\n", z);
```

Project 2: User Programs and System Calls

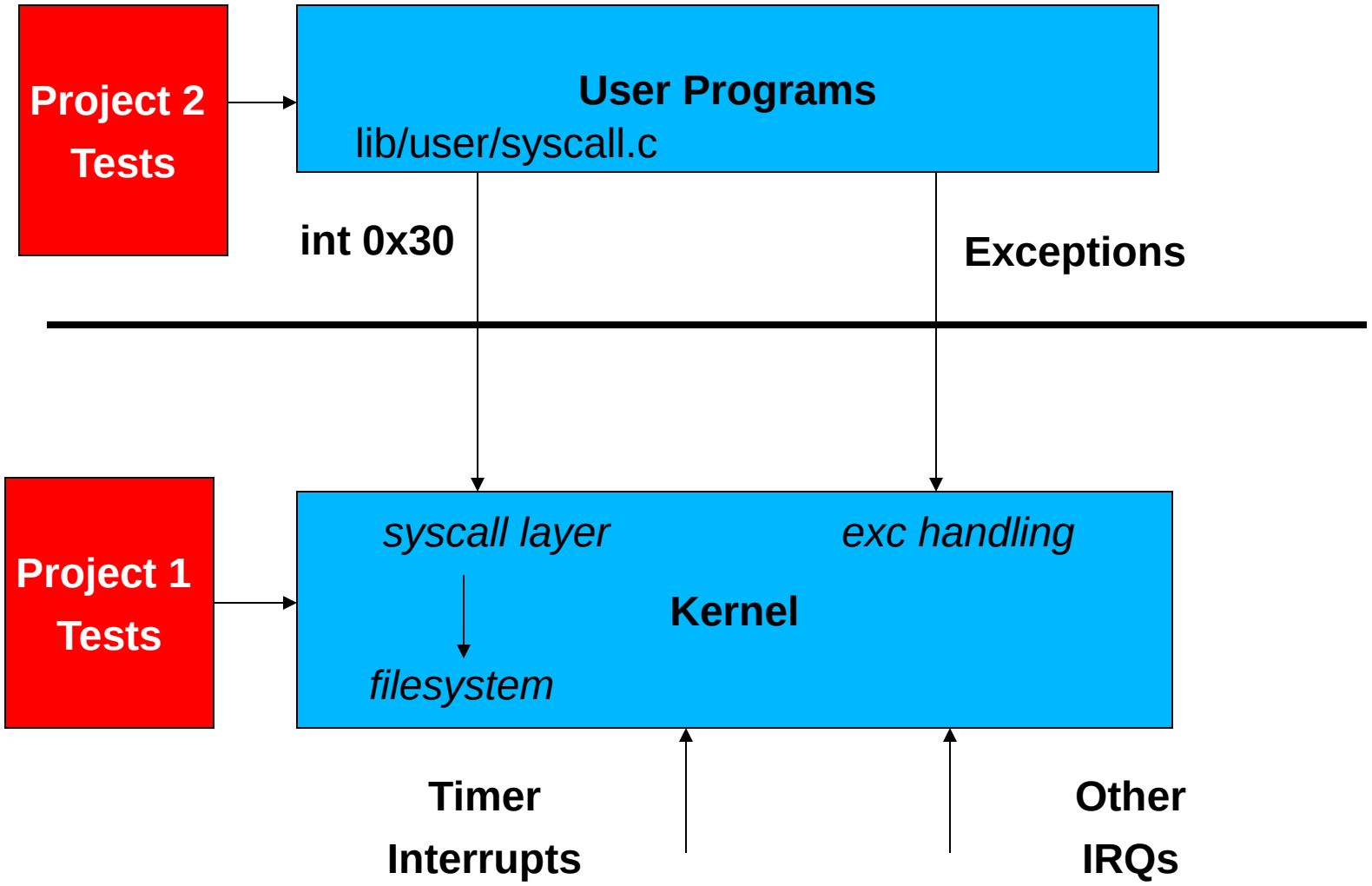
P2: Project 2 – System Calls



Till now…

- All code part of Pintos Kernel
- Code compiled directly with the kernel
  - This required that the tests call some functions whose interface should remain unmodified
- From now on, run user programs on top of kernel
  - Freedom to modify the kernel to make the user programs work

Project 1 and Project 2



When does a process need to access OS functionality?

- Here are several examples:
  - Reading a file. The OS must perform the file system operations required to read the data off of disk.
  - Creating a child process. The OS must set stuff up for the child process.
  - Sending a packet out onto the network. The OS typically handles the network interface.

Why have the OS do these things?

- Why doesn't the process just do them directly?
  - **Convenience.** Implement the functionality once in the OS and encapsulate it behind an interface that everyone uses. So, processes just deal with the simple interface, and don't have to write complicated low-level code to deal with devices.
  - **Portability.** OS exports a common interface typically available on many hardware platforms. Applications do not contain hardware-specific code.
  - **Protection.** If give applications complete access to disk or network or whatever, they can corrupt data from other applications, either maliciously or because of bugs. Having the OS do it eliminates security problems between applications. Of course, applications still have to trust the OS.

How do processes invoke OS functionality?

- By making a system call.
  - Conceptually, processes call a subroutine that goes off and performs the required functionality. But OS must execute in supervisor mode, which allows it to do things like manipulate the disk directly.
  - To switch from normal user mode to supervisor mode, most machines provide a system call instruction.
    - This instruction causes an exception to take place.
    - The hardware switches from user mode to supervisor mode and invokes the exception handler inside the operating system.
  - There is typically some kind of convention that the process uses to interact with the OS.

Let's do an example - Open() system call.

- System calls typically start out with a normal subroutine call.

```
int handle = open("sample.txt");
```

- Open() executes a syscall instruction, which generates a system call exception 0x30.

```
syscall1 (SYS_OPEN, file);
```

- By convention, the Open subroutine puts a number on the stack to indicate which routine (SYS_OPEN = 6) should be invoked.
  - Inside the exception handler the OS looks at the stack to figure out what system call it should perform.
- The Open system call also takes a parameter. By convention, the compiler also puts this (e.g., a pointer to the filename) on the stack.
  - More conventions: return values are put into the %EAX register.
- Inside the exception handler, the OS figures out what action to take, performs the action, then returns back to the user program.

SYS_OPEN defined in lib/syscall-nr.h

```
#ifndef __LIB_SYSCALL_NR_H
#define __LIB_SYSCALL_NR_H

/* System call numbers. */
enum
  {
    /* Projects 2 and later. */
    SYS_HALT,                 /* Halt the operating system. */
    SYS_EXIT,                 /* Terminate this process. */
    SYS_EXEC,                 /* Start another process. */
    SYS_WAIT,                 /* Wait for a child process to die. */
    SYS_CREATE,               /* Create a file. */
    SYS_REMOVE,               /* Delete a file. */
    SYS_OPEN,                 /* Open a file. */
…
```

Thus, **SYS_OPEN = 6.**

Open( ) system call details

In pintos/src/lib/user/syscall.c:

```
int
open (const char *file)
{
  return syscall1 (SYS_OPEN, file);
}
.. (and above)..

/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an `int'. */
#define syscall1(NUMBER, ARG0)                                           \
        ({                                                               \
          int retval;                                                    \
          asm volatile                                                   \
            ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
               : "=a" (retval)                                           \
               : [number] "i" (NUMBER),                                  \
                 [arg0] "g" (ARG0)                                       \
               : "memory");                                              \
          retval;                                                        \
        })
```

Initialize syscall handler

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

Add sys_open function to syscall.c

```
+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)
+        {
+          struct thread *cur = thread_current ();
+          handle = fd->handle = cur->next_handle++;
+          list_push_front (&cur->fds, &fd->elem);
+        }
+      else
+        free (fd);
+      lock_release (&fs_lock);
+    }
+
+  palloc_free_page (kfile);
+  return handle;
+}
```

Add sys_open to syscall_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},
+      …
+    }
```

Add sys_open to syscall_handler

```
+  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]);
+}
```

Interrupts vs. Exceptions

- The difference between interrupts and exceptions is that
  - **interrupts** are generated by external events (the disk IO completes, a new character is typed at the console, etc.), and
  - **exceptions** are generated by a running program.

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

- Creating a simulated disk
  - pintos-mkdisk fs.dsk 2
- Formatting the disk
  - pintos -f -q
  - This will only work after your kernel is built!
- Copying the program into the disk
  - pintos -p ../../examples/echo -a echo -- -q
- Running the program
  - pintos -q run ’echo x’
- Single command:

pintos --fs-disk=2 -p ../../examples/echo -a echo -- -f -q run ’echo x’

- $ **make check** – 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

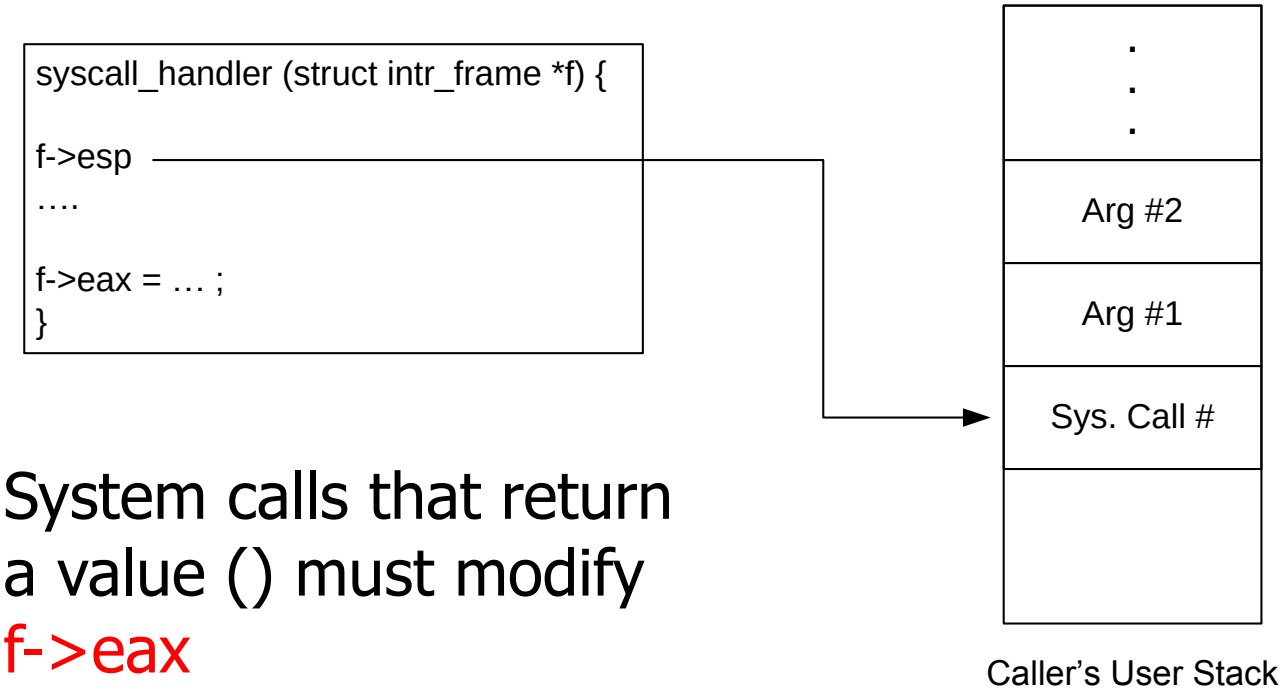
- Process Termination Messages
- Argument Passing
- System Calls
- Deny writes to executables

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:



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

Summary

- Read Ch. 1-8
- Processes and Threads (Ch. 4)
- Process Scheduling (Ch. 5)
- Synchronization (Ch. 6)
- Deadlock (Ch. 7)
- Memory Management (Ch. 8)
- Project 1 – Scheduling and Synchronization
- Quiz #1 – Ch. 1-7 – 10/9