

# CSE 421/521 - Operating Systems Fall 2018

LECTURE - V

## PROJECT-1 DISCUSSION

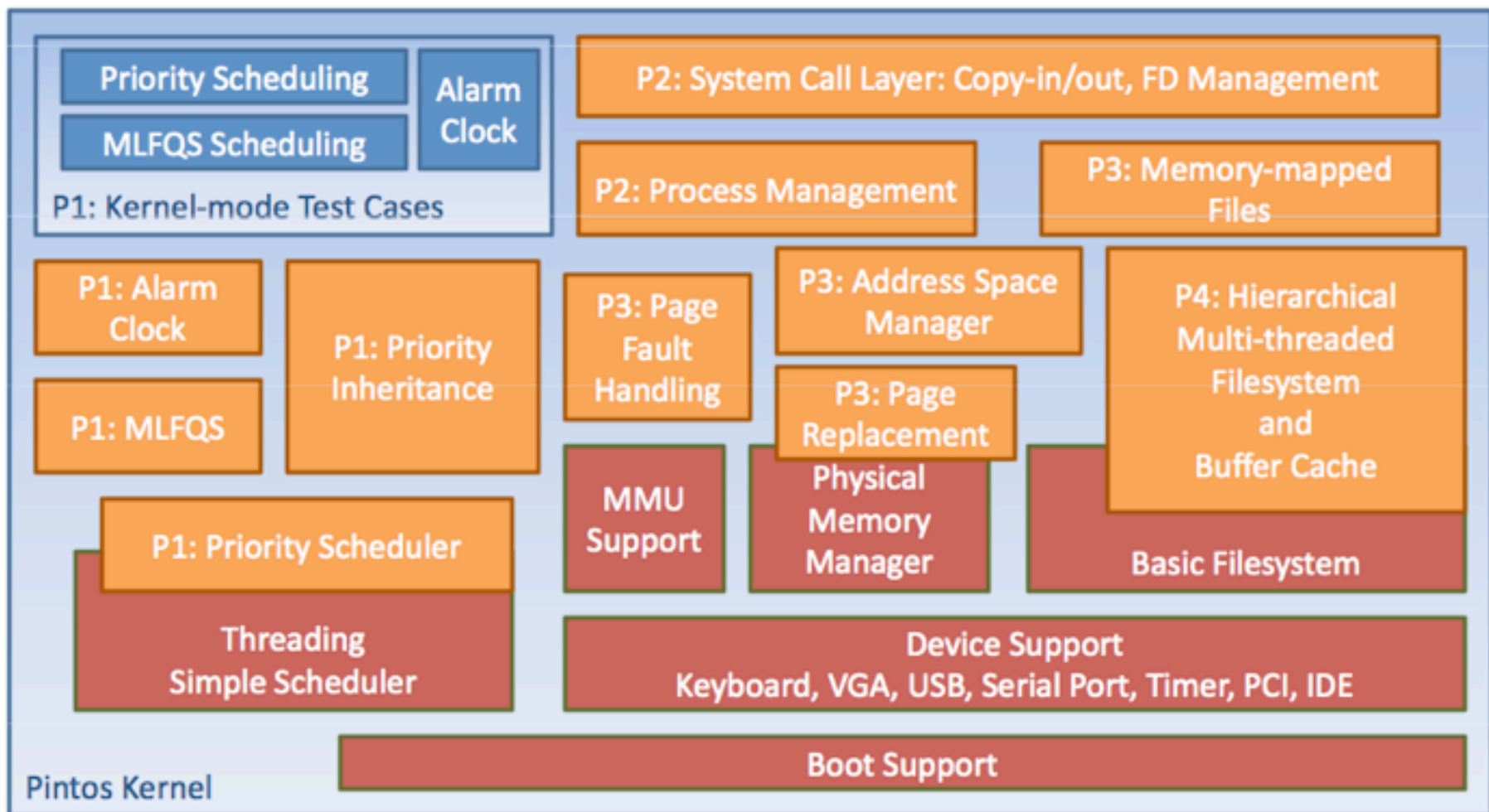
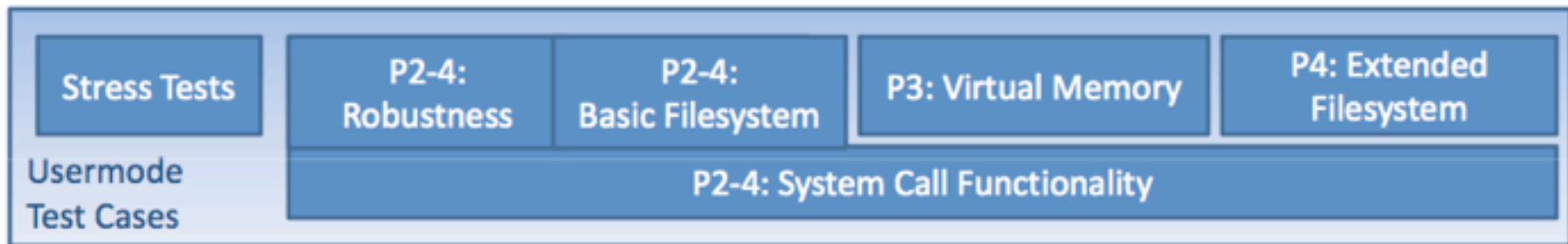
Tevfik Koşar

University at Buffalo  
September 11th, 2018

# Pintos Projects

1. Threads <-- CSE 421/521 Project 1
2. User Programs
3. Virtual Memory
4. File Systems

# Pintos after full implementation (post prj-4)



Pintos Kernel

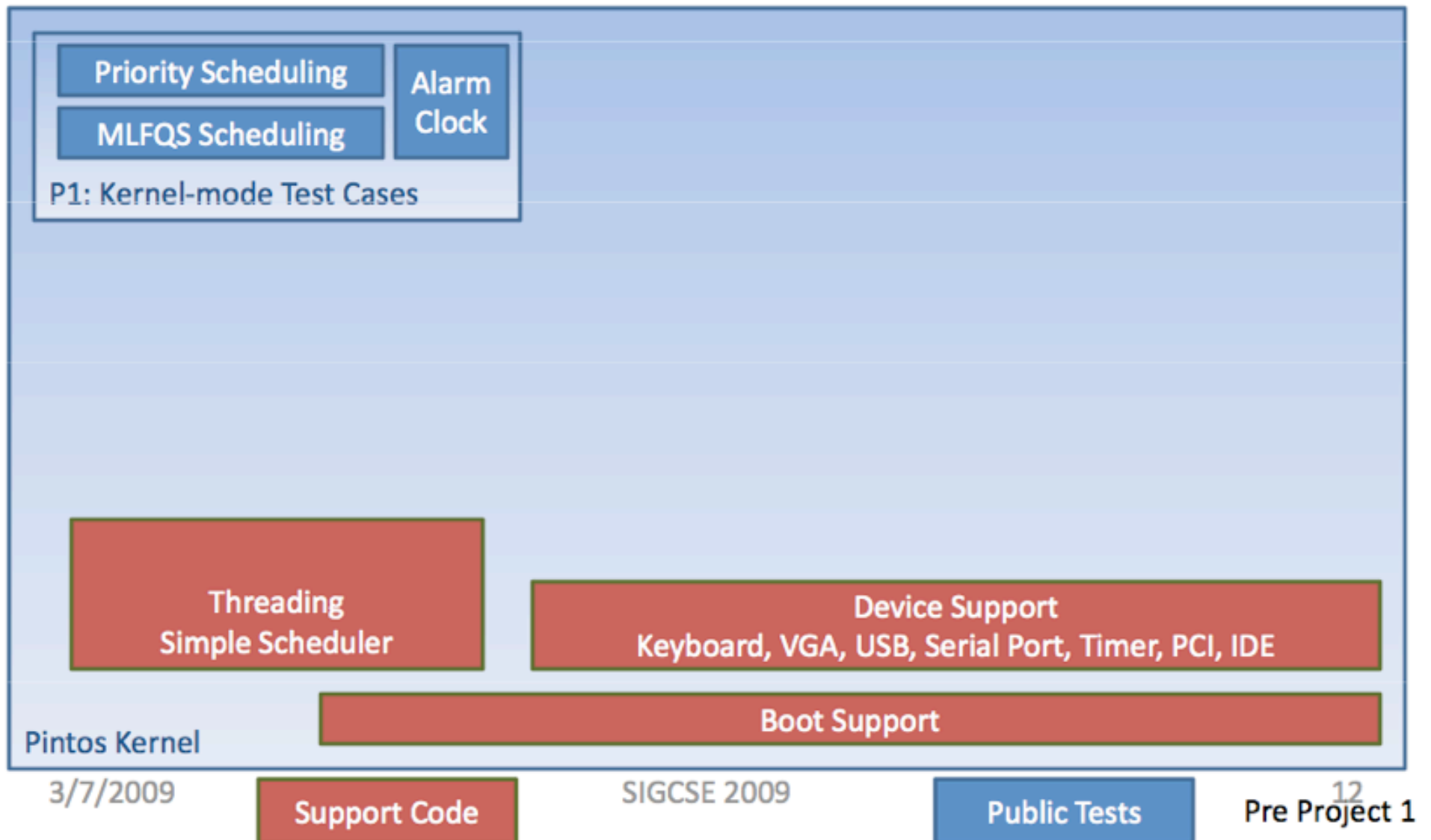
Support Code

Students Create

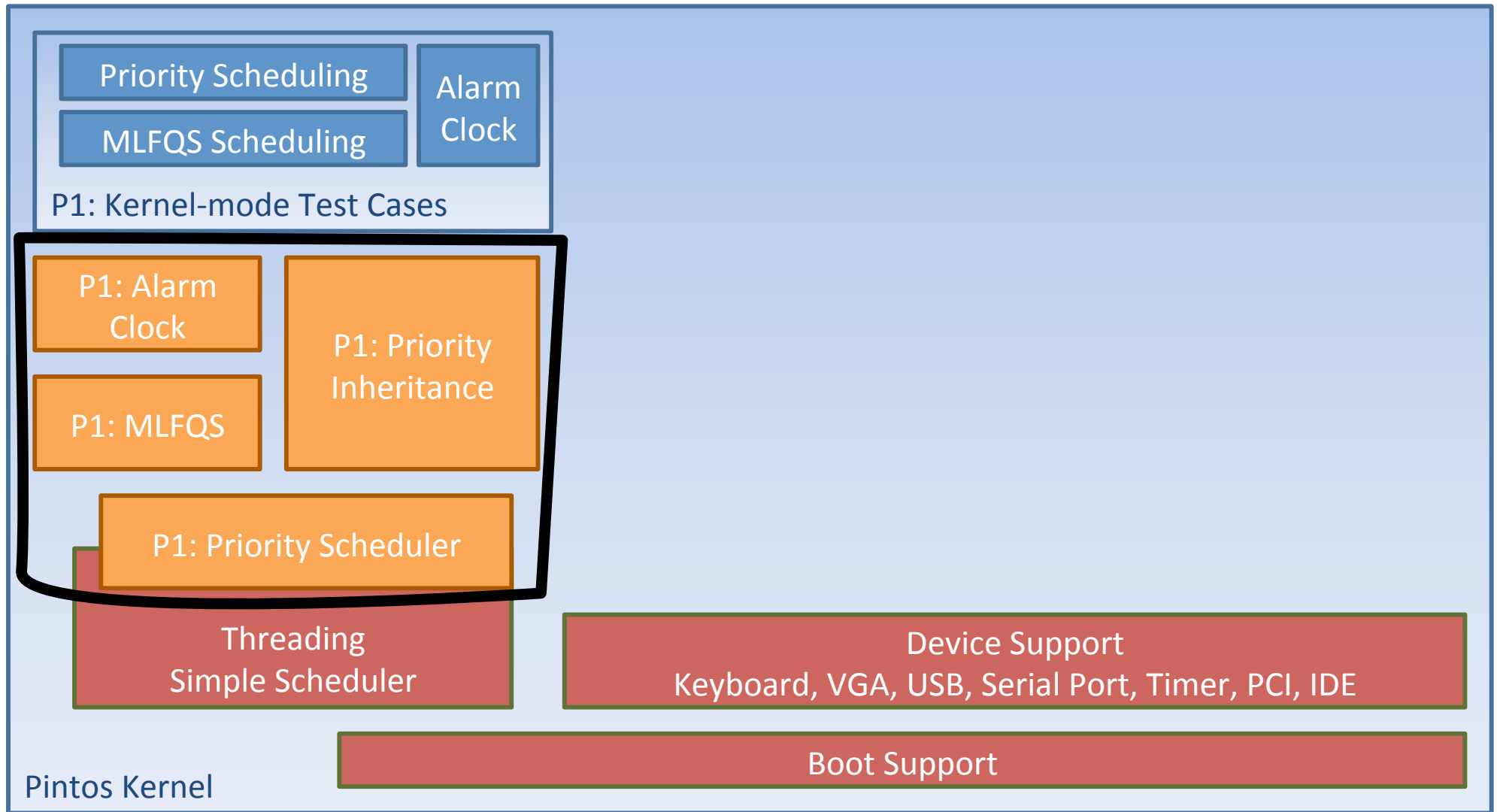
Public Tests

Post Project 4

# Yo will be provided with this (pre prj-1)



# You will implement this (post prj-1)



# Step 1: Preparation

## READ:

Chapters 3-5 from Silberschatz

Lecture slides on Processes, Threads and CPU Scheduling

From Pintos Documentation:

- Section 1 - Introduction
- Section 2 - Threads
- Appendix A1- Pintos Loading
- Appendix A2 - Threads
- Appendix A3 - Synchronization
- Appendix B - 4.4BSD Scheduler

## Step 2: Setting Up Pintos

- Use the Pintos **VM** we have prepared for you:

<http://ftp.cse.buffalo.edu/CSE421/UB-pintos.ova>

- It requires **Virtualbox** software

==> will work on most Linux, Windows, Mac systems

<https://www.virtualbox.org/wiki/Downloads>

- Detailed setup instructions are available on Piazza.

## Step 2b: Verify Setup

- Compile
  - `$ cd $PINTOSDIR/src/threads`
  - `$ make`
- Test
  - `$ cd build`
  - `$ pintos run alarm-multiple`



## Step 3: Implementation

1. Alarm Clock
2. Priority Scheduler (with priority donation)
3. Multilevel Feedback Queue Scheduler

# Task 0: Get Familiar with the Code

- The first task is to read and understand the code for the initial thread system (under the “[pintos/src/threads/](#)” directory).
- Pintos already implements thread creation and thread completion, a simple scheduler to switch between threads, and synchronization primitives (semaphores, locks, condition variables, and optimization barriers).
- For a brief overview of the files in the “threads/” directory, please see “[Section 2.1.2 Source Files](#)” in the Pintos Reference Guide

# Pintos Thread System

- Read threads/thread.c and threads/synch.c to understand
  - How the switching between threads occur
  - How the provided scheduler works
  - How the various synchronizations primitives work

# Important Directories

- **threads/**
  - *Source code for the base kernel, which you will modify starting in project 1.*
- **devices/**
  - *Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in project 1. Otherwise you should have no need to change this code.*
- **lib/kernel/**
  - *Parts of the C library that are included only in the Pintos kernel. Feel free to reuse this code*
- **tests/**
  - *Tests for each project. You can modify this code if it helps you test your submission, but we will replace it with the originals before we run the tests.*

# Files of Interest

- ***thread.c* and *thread.h***
  - Basic thread support. Most of project 1 work. Also see struct thread.
- ***synch.c* and *synch.h***
  - *Synchronization primitives which you can use in all projects*
- ***devices/timer.c* and *devices/timer.h***
  - *Timer ticks, has to be modified for project 1*
- ***lib/kernel/list.c***
  - *Linked list implementation, feel free to reuse*
- ***init.c* and *init.h***
  - Kernel initialization, including main(), the kernel's "main program."

# Pintos Thread System

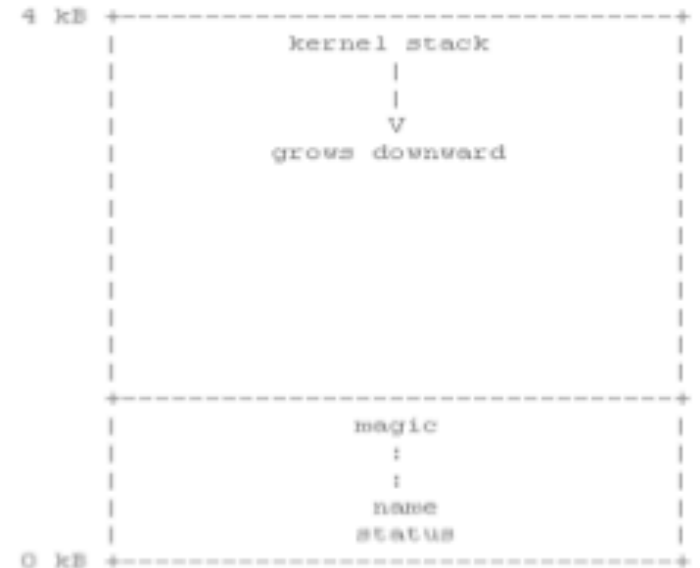
Defined in threads/thread.h:

```
struct thread
{
    tid_t tid;          /* Thread identifier. */
    enum thread_status status; /* Thread state. */
    char name[16]; /* Name (for debugging purposes). */
    uint8_t *stack; /* Saved stack pointer. */
    int priority; /* Priority. */
    struct list_elem allelem; /* List element for all-threads list. */
    /* Shared between thread.c and synch.c. */
    struct list_elem elem; /* List element. */

```

You add more fields here as you need them.

```
#ifndef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir; /* Page directory. */
#endif
    /* Owned by thread.c. */
    unsigned magic; /* Detects stack overflow. */
};
```



# Task 1: Implement Alarm Clock

- Reimplement `timer_sleep()` in `devices/timer.c` without busy waiting

`/* Suspends execution for approximately TICKS timer ticks. */`

```
void timer_sleep (int64_t ticks){  
    int64_t start = timer_ticks ();  
    ASSERT (intr_get_level () == INTR_ON);  
    while (timer_elapsed (start) < ticks)  
        thread_yield ();  
}
```

- Implementation details
  - Remove thread from ready list and put it back after sufficient ticks have elapsed

**Any implementation using busy-waiting will not get full credits!**

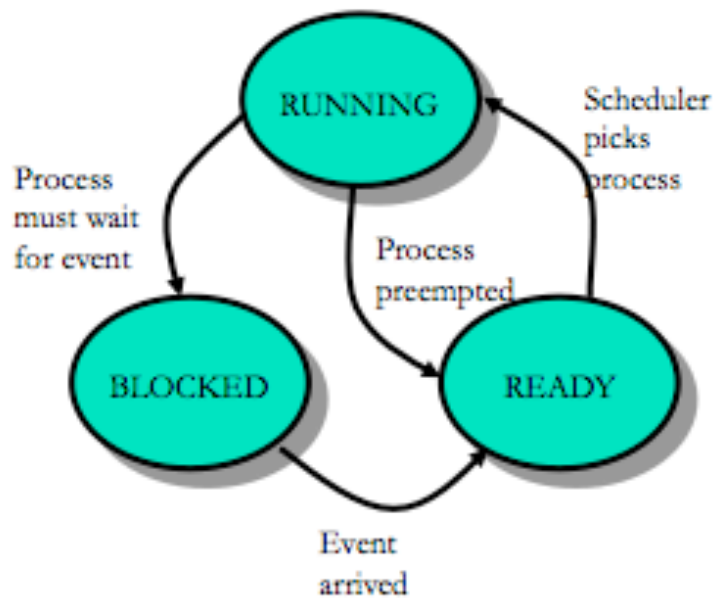
# Task 2A: Implement Priority Scheduler

- Ready thread with highest priority gets the processor
- When a thread is added to the ready list that has a higher priority than the currently running thread, immediately yield the processor to the new thread
- When threads are waiting for a lock, semaphore or a condition variable, the highest priority waiting thread should be woken up first
- Implementation details
  - compare priority of the thread being added to the ready list with that of the running thread
  - select next thread to run based on priorities
  - compare priorities of waiting threads when releasing locks, semaphores, condition variables

← Preemptive



# use `thread_yield()` to implement preemption



- Current thread (“**RUNNING**”) is moved to **READY** state, added to **READY** list.
- Then scheduler is invoked. Picks a new **READY** thread from **READY** list.
- Case a): there’s only 1 **READY** thread. Thread is rescheduled right away
- Case b): there are other **READY** thread(s)
  - b.1) another thread has higher priority – it is scheduled
  - b.2) another thread has same priority – it is scheduled provided the previously running thread was inserted in tail of ready list.
- “`thread_yield()`” is a call you can use whenever you identify a need to preempt current thread.
- **Exception:** inside an interrupt handler, use “`intr_yield_on_return()`” instead

\* re-implement **`next_thread_to_run()`** for priority scheduling

# Priority Inversion

- Strict priority scheduling can lead to a phenomenon called “priority inversion”
- Supplemental reading:
  - What really happened to the Pathfinder on Mars?
- Consider the following example where  $\text{prio}(H) > \text{prio}(M) > \text{prio}(L)$ 
  - H needs a lock currently held by L, so H blocks
  - M that was already on the ready list gets the processor before L
  - H indirectly waits for M
  - (on Path Finder, a watchdog timer noticed that H failed to run for some time, and continuously reset the system)

## Task 2B: Implement Priority Donation

- When a high priority thread H waits on a lock held by a lower priority thread L, donate H's priority to L and recall the donation once L releases the lock
- Implement priority donation for locks
- Important:
  - Remember to return L to previous priority once it releases the lock.
  - Be sure to handle multiple donations (max of all donations)
  - Be sure to handle nested donations, e.g., H waits on M which waits on L...

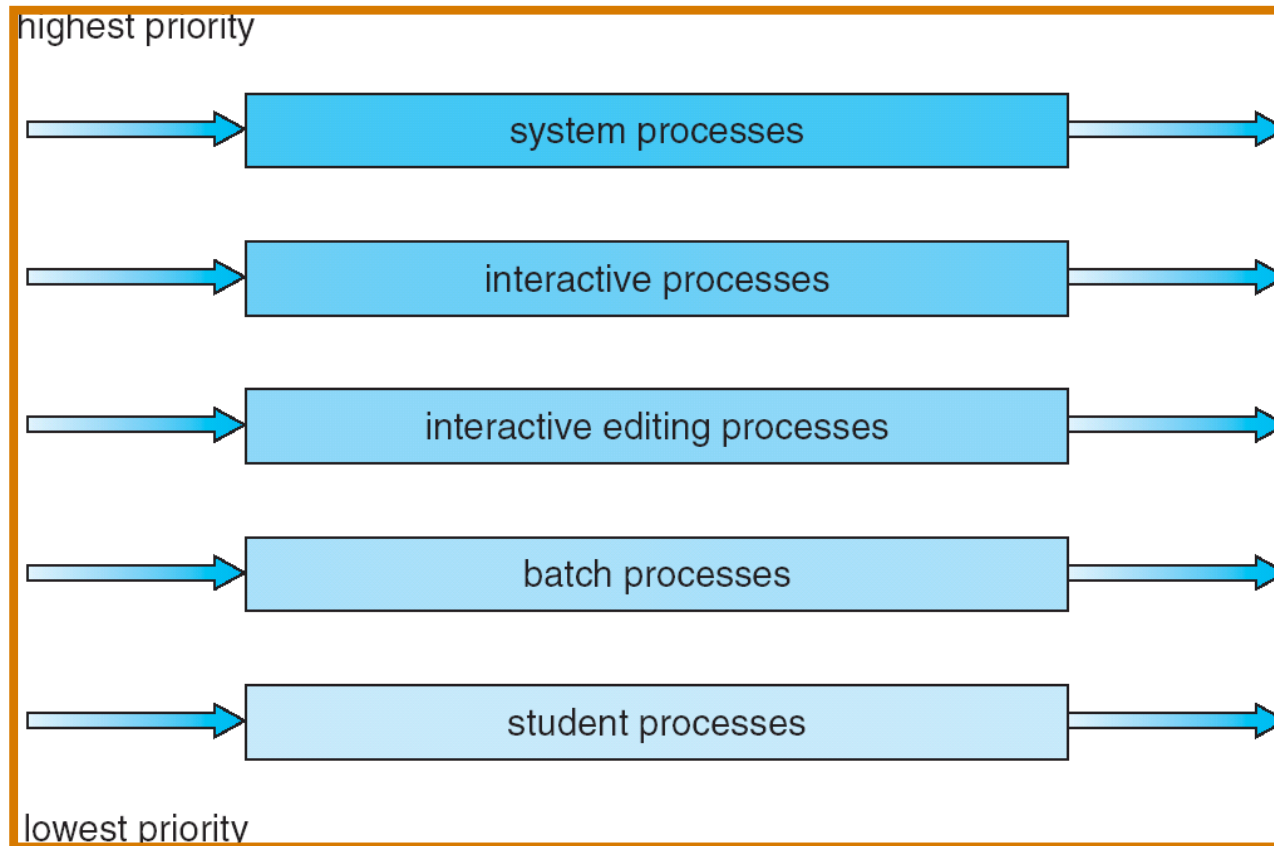
# Synchronization

- Any synchronization problem can be easily solved by turning interrupts off: while interrupts are off, there is no concurrency, so there's no possibility for race conditions. **But, you should NOT do this!**
- Instead, **use semaphores, locks, and condition variables** to solve the bulk of your synchronization problems.
- **Any implementation turning the interrupts off for synchronization purposes, will not get full credits!**
- **The only place you are allowed to turn interrupts off is,** when coordinating data shared between a kernel thread and an interrupt handler. Because interrupt handlers can't sleep, they can't acquire locks.

# Task 3: Implement Advanced Scheduler

- Implement Multi Level Feedback Queue Scheduler
- Priority donation not needed in the advanced scheduler – two implementations are not required to coexist
  - Only one is active at a time
- Advanced Scheduler must be chosen only if ‘-mlfq’ kernel option is specified
- Read section on 4.4 BSD Scheduler in the Pintos manual for detailed information
- Some of the parameters are real numbers and calculations involving them have to be simulated using integers.
  - Write a fixed-point layer (header file)

# Multilevel Queue Scheduling

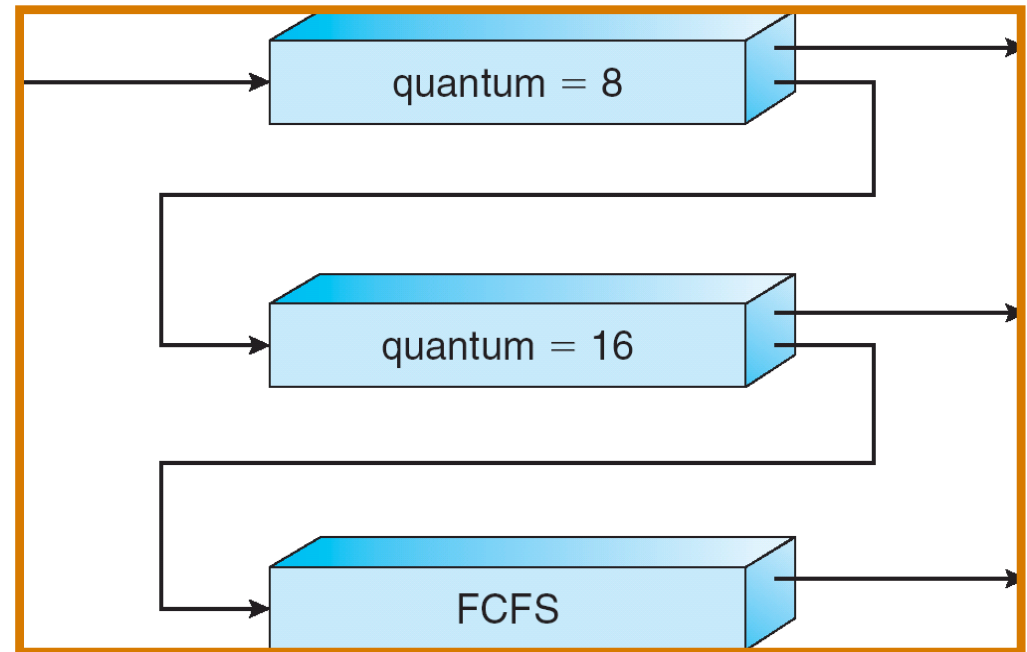


# Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine which queue a process will enter when that process needs service
  - method used to determine when to upgrade a process
  - method used to determine when to degrade a process

# Example of Multilevel Feedback Queue

- Three queues:
  - $Q_0$  - RR with  $q = 8$  ms
  - $Q_1$  - RR with  $q = 16$  ms
  - $Q_2$  - FCFS

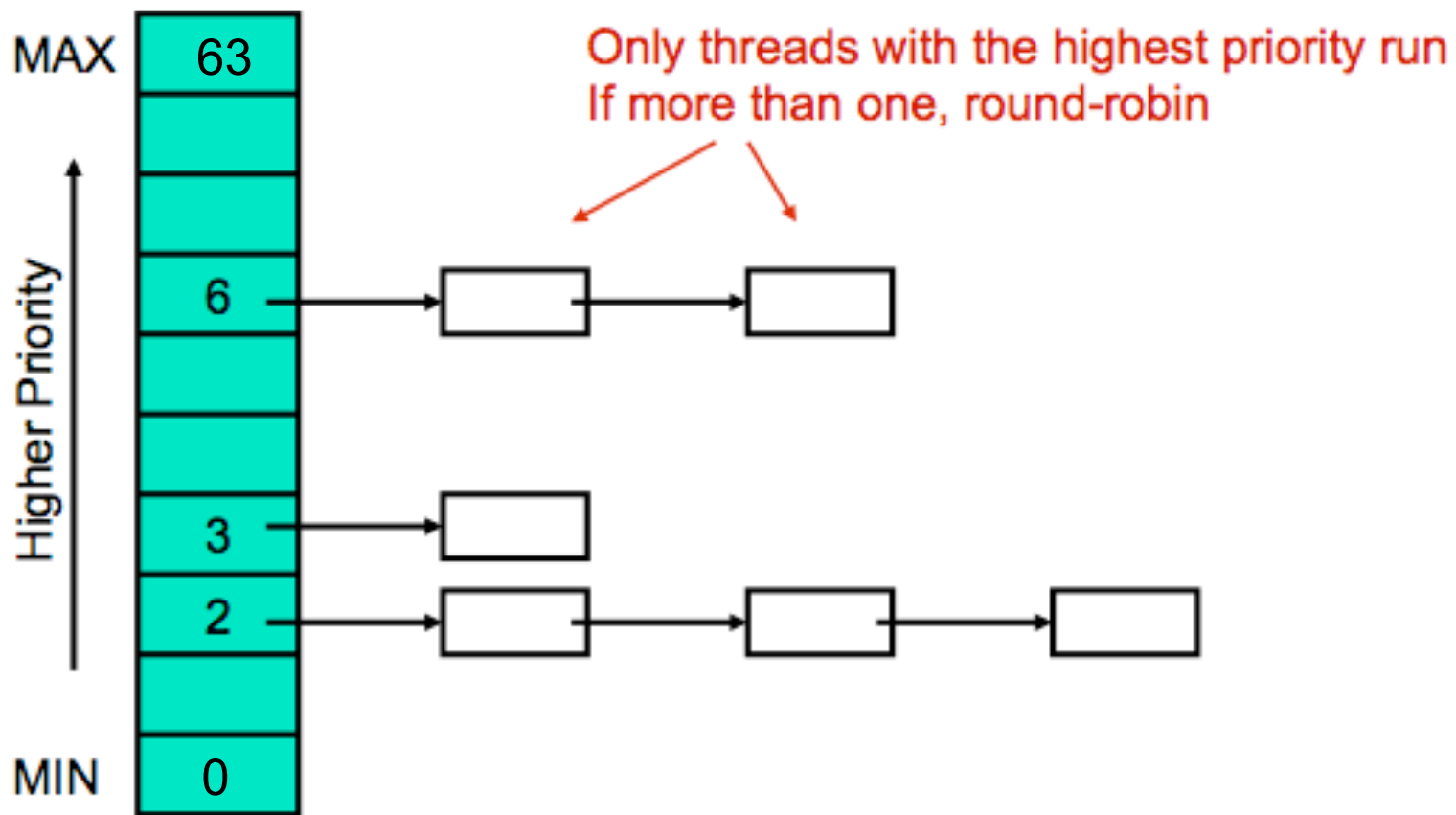


- Scheduling
  - A new job enters queue  $Q_0$  which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue  $Q_1$ .
  - At  $Q_1$  job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue  $Q_2$ .



## 4.4BSD Priority Based Scheduling

4.4BSD scheduler has 64 priorities and thus 64 ready queues, numbered 0 (PRI\_MIN) through 63 (PRI\_MAX).



# Calculating Priority

- NOTE: Lower numbers correspond to lower priorities in 4.4BSD, so that priority 0 is the lowest priority and priority 63 is the highest.
- Every 4 clock ticks, calculate:  
$$\text{priority} = \text{PRI\_MAX} - (\text{recent\_cpu} / 4) - (\text{nice} * 2)$$

*(rounded down to the nearest integer)*
- It gives a thread that has received CPU time recently lower priority for being reassigned the CPU the next time the scheduler runs.

← Aging

# Nice Value

==> how "nice" the thread should be to other threads.

- A nice of zero does not affect thread priority.
- A positive nice, to the **maximum of 20**, decreases the priority of a thread and causes it to give up some CPU time it would otherwise receive.

A negative nice, to the **minimum of -20**, tends to take away CPU time from other threads.

## Calculating recent\_cpu

- An array of  $n$  elements to track the CPU time received in each of the last  $n$  seconds requires  $O(n)$  space per thread and  $O(n)$  time per calculation of a new weighted average.
- Instead, we use an exponentially weighted moving average:
  - **recent\_cpu(0) = 0** // or parent thread's value
  - at each timer interrupt, **recent\_cpu++** for the running thread.
  - and once per second, for each thread:

$$\text{recent\_cpu}(t) = a * \text{recent\_cpu}(t-1) + \text{nice}$$

$$\text{where, } a = (2 * \text{load\_avg}) / (2 * \text{load\_avg} + 1)$$

## Calculating load\_avg

- Estimates the average number of threads ready to run over the past minute.
- Like recent\_cpu, it is an exponentially weighted moving average.
- Unlike priority and recent\_cpu, load\_avg is system-wide, not thread-specific.
- At system boot, it is **initialized to 0**. Once per second thereafter, it is updated according to the following formula:

$$\text{load\_avg}(t) = (59/60) * \text{load\_avg}(t-1) + (1/60) * \text{ready\_threads}$$

- ready\_threads: number of threads that are either running or ready to run at the time of update

# Functions to implement

- Skeletons of these functions are provided in “threads/threads.c”:
- `int thread_get_nice (void)`
- `void thread_set_nice (int new_nice)`
- `void thread_set_priority (int new_priority)`
- `int thread_get_priority (void)`
- `int thread_get_recent_cpu (void)`
- `int thread_get_load_avg (void)`

# Suggested Order of Implementation

- Alarm Clock
  - easier to implement compared to the other parts
  - other parts not dependent on this
- Priority Scheduler
  - needed for implementing Priority Donation and Advanced Scheduler
- Priority Donation | Advanced Scheduler
  - these two parts are independent of each other
  - can be implemented in any order but only after Priority Scheduler is ready

# Debugging your Code

- printf, ASSERT, backtraces, gdb
- Running pintos under gdb
  - Invoke pintos with the gdb option  
`pintos --gdb -- run testname`
  - On another terminal invoke gdb  
`pintos-gdb kernel.o`
  - Issue the command  
`debugpintos`
  - All the usual gdb commands can be used: step, next, print, continue, break, clear etc
  - Use the pintos debugging macros described in manual



## Step 4: Testing

Pintos provides a very systematic testing suite for your project:

### 1. Run all tests:

```
$ make check
```

### 2. Run individual tests:

```
$ make tests/threads/alarm-multiple.result
```

### 3. Run the grading script:

```
$ make grade
```

# make check

- **pass** tests/threads/alarm-single
- **pass** tests/threads/alarm-multiple
- **pass** tests/threads/alarm-simultaneous
- FAIL tests/threads/alarm-priority
- **pass** tests/threads/alarm-zero
- **pass** tests/threads/alarm-negative
- FAIL tests/threads/priority-change
- FAIL tests/threads/priority-donate-one
- FAIL tests/threads/priority-donate-multiple
- FAIL tests/threads/priority-donate-multiple2
- FAIL tests/threads/priority-donate-nest
- FAIL tests/threads/priority-donate-sema
- FAIL tests/threads/priority-donate-lower
- FAIL tests/threads/priority-fifo
- FAIL tests/threads/priority-preempt
- FAIL tests/threads/priority-sema
- FAIL tests/threads/priority-condvar
- FAIL tests/threads/priority-donate-chain
- FAIL tests/threads/mlfqs-load-1
- FAIL tests/threads/mlfqs-load-60
- FAIL tests/threads/mlfqs-load-avg
- FAIL tests/threads/mlfqs-recent-1
- **pass** tests/threads/mlfqs-fair-2
- **pass** tests/threads/mlfqs-fair-20
- FAIL tests/threads/mlfqs-nice-2
- FAIL tests/threads/mlfqs-nice-10
- FAIL tests/threads/mlfqs-block

## Grading - Alarm Clock: 18 pts

- 4 pts - tests/threads/alarm-single
  - 4 pts - tests/threads/alarm-multiple
  - 4 pts - tests/threads/alarm-simultaneous
  - 4 pts - tests/threads/alarm-priority
  - 1 pts - tests/threads/alarm-zero
  - 1 pts - tests/threads/alarm-negative
- 
- If Alarm clock implementation is based on “busy-waiting” ( -14 points )
  - If interrupts are turned off as the only synchronization mechanism ( -14 points )

## Grading - Priority Scheduler: 38 pts

- 3 pts - tests/threads/priority-change
- 3 pts - tests/threads/priority-preempt
- 3 pts - tests/threads/priority-fifo
- 3 pts - tests/threads/priority-sema
- 3 pts - tests/threads/priority-condvar
- 3 pts - tests/threads/priority-donate-one
- 3 pts - tests/threads/priority-donate-multiple
- 3 pts - tests/threads/priority-donate-multiple2
- 3 pts - tests/threads/priority-donate-nest
- 5 pts - tests/threads/priority-donate-chain
- 3 pts - tests/threads/priority-donate-sema
- 3 pts - tests/threads/priority-donate-lower

## Grading - MLFQ Scheduler: 37 pts

- 5 pts - tests/threads/mlfqs-load-1
- 5 pts - tests/threads/mlfqs-load-60
- 5 pts - tests/threads/mlfqs-load-avg
- 5 pts - tests/threads/mlfqs-recent-1
- 5 pts - tests/threads/mlfqs-fair-2
- 3 pts - tests/threads/mlfqs-fair-20
- 4 pts - tests/threads/mlfqs-nice-2
- 2 pts - tests/threads/mlfqs-nice-10
- 5 pts - tests/threads/mlfqs-block

# Grading

**TOTAL 105 points:** 93 points for the implementation + 12 points for the design document:

- **18 points:** A completely working Alarm Clock implementation that passes all six (6) tests.
- **38 points:** A fully functional Priority Scheduler that passes all twelve (12) tests.
- **37 points:** A working advanced scheduler that passes all nine (9) tests.
- **12 points:** A complete design document.

# make grade (1)

## SUMMARY OF INDIVIDUAL TESTS

Functionality and robustness of alarm clock (tests/threads/Rubric.alarm):

- 4/ 4 tests/threads/alarm-single
- 4/ 4 tests/threads/alarm-multiple
- 4/ 4 tests/threads/alarm-simultaneous
- 4/ 4 tests/threads/alarm-priority
  
- 1/ 1 tests/threads/alarm-zero
- 1/ 1 tests/threads/alarm-negative

- Section summary.

- 6/ 6 tests passed
- 18/ 18 points subtotal

Functionality of priority scheduler (tests/threads/Rubric.priority):

- 3/ 3 tests/threads/priority-change
- 3/ 3 tests/threads/priority-preempt
  
- 3/ 3 tests/threads/priority-fifo
- 3/ 3 tests/threads/priority-sema
- 3/ 3 tests/threads/priority-condvar

## make grade (2)

TOTAL TESTING SCORE: 100.0%  
ALL TESTED PASSED -- PERFECT SCORE

-----

### SUMMARY BY TEST SET

| Test Set                      | Pts | Max | % Ttl   | % Max  |
|-------------------------------|-----|-----|---------|--------|
| tests/threads/Rubric.alarm    | 18/ | 18  | 20.0%/  | 20.0%  |
| tests/threads/Rubric.priority | 38/ | 38  | 40.0%/  | 40.0%  |
| tests/threads/Rubric.mlfqs    | 37/ | 37  | 40.0%/  | 40.0%  |
| Total                         |     |     | 100.0%/ | 100.0% |

-----

*Pintos include fully automated grading scripts, students see score before submission*



## Grading - an important note!

- **CHECK CODE:** If Alarm clock implementation is based on “busy-waiting” ( -14 points )
- **CHECK CODE:** If interrupts are turned off as the only synchronization mechanism ( -14 points )

# Step 5: Design Document

Use the template in Section 2.2.1 of the Pintos documentation.

```

                                ALARM CLOCK
                                =====

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration.  Identify the purpose of each in 25 words or less.

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(),
>> including the effects of the timer interrupt handler.

>> A3: What steps are taken to minimize the amount of time spent in
>> the timer interrupt handler?

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?

---- RATIONALE ----

>> A6: Why did you choose this design?  In what ways is it superior to
>> another design you considered?
```

# Submission

## 1. Design document

- due by **October 1st @11:59PM**
- submit using your **submit\_cse421** or **submit\_cse521** script

## 2. All source code (the full source tree)

- due by **October 21st @11:59PM**
- submit via **AutoLab** auto-grading system

# Late Policy

Up to 24 hour late submission is accepted with 20% penalty:

❖ before 11:59pm on 10/21 --> no penalty

between 11:59pm on 10/21 and 11:59pm on 10/22 --> 20%  
penalty

after 11:59pm on 10/22 --> submission not accepted!