CSE 421/521
Introduction to Operating Systems

Farshad Ghanei

Lecture - 05
Project-1 Discussion

* Slides adopted from Prof Kosar and Dantu at UB, “Operating System Concepts” book and supplementary material by A. Silberschatz, P.B. Galvin, and G. Gagne. Wiley Publishers

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Summary

- Threads
  - Concurrent programming
  - Why threads?
  - Threads vs Processes
  - Threading examples
  - Thread pools
  - Threading implementation & multithreading models
  - Threading issues
    - Semantics of fork() and exec()
    - Thread cancellation
    - Signal handling

Today

- Pintos projects:
  - Project-1: Threads
    - Step 1: Preparation
    - Step 2: Setting Up Pintos
    - Step 3: Design Document
    - Step 4: Implementation
    - Step 5: Testing
  - Project-2: User Programs
  - Project-3: Virtual Memory
  - Project-4: File Systems

Pintos

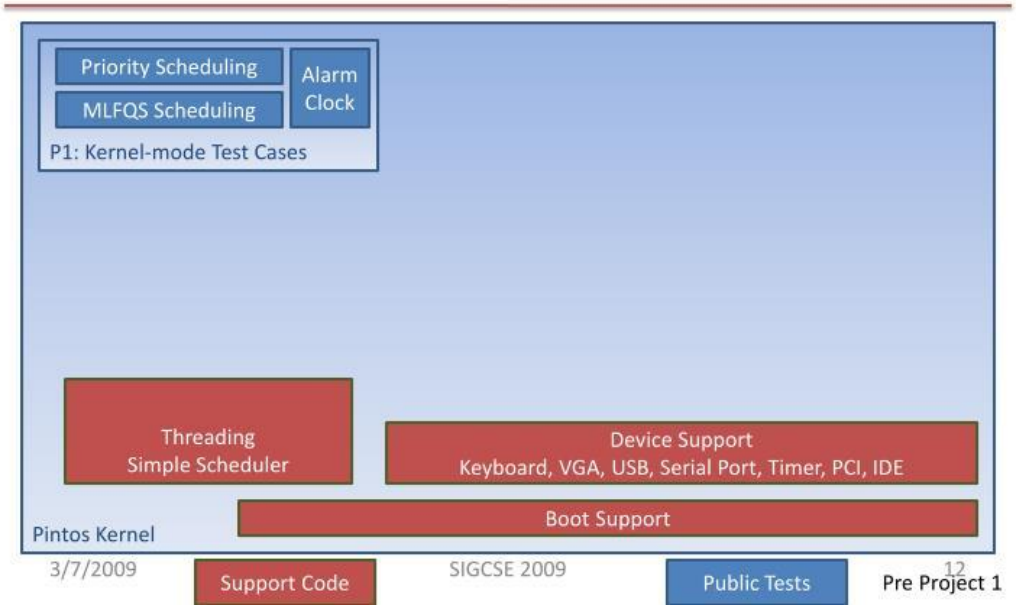
Your programming assignments are based on Pintos operating system:

- A Simple operating system framework for the 80x86 architecture.
- Developed by Stanford University (Originally by Ben Pfaff)
- Could theoretically run on a regular IBM-compatible PC
- Practically, it runs using Bochs and QEMU simulators
  (OS in OS ?)
- Each and every student might have a different environment, which affects development, and running
- We provide a VirtualBox image with Ubuntu 16.04
  - Download VirtualBox and the image, and run it
- We'll run Bochs and QEMU within the VM image.
  (OS in OS in OS ?)

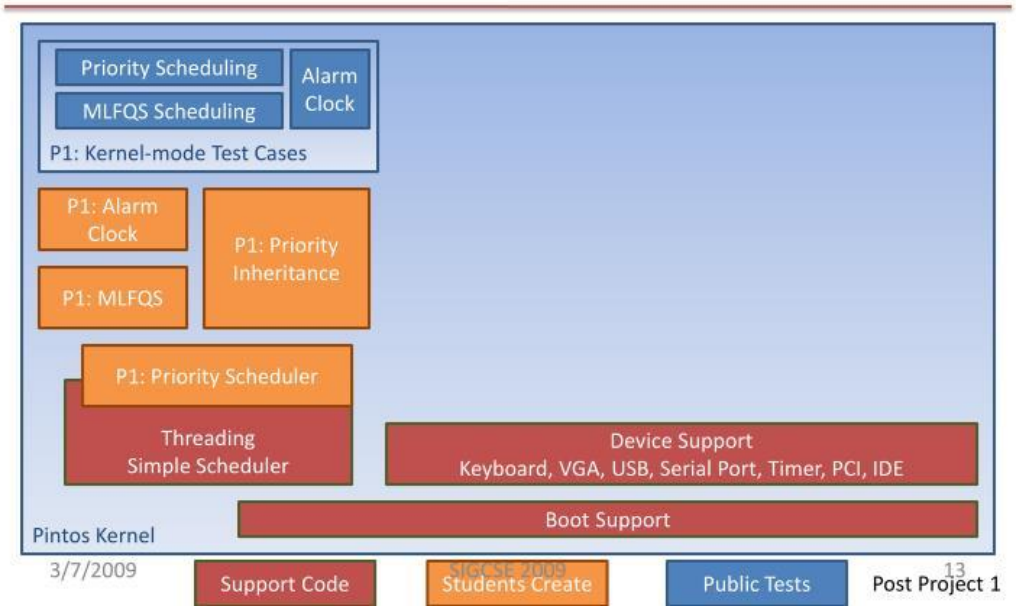
Pintos Projects

- Threads (CSE 421/521 Project-1)
- User Programs (CSE 421/521 Project-2)
- Virtual Memory
- File Systems

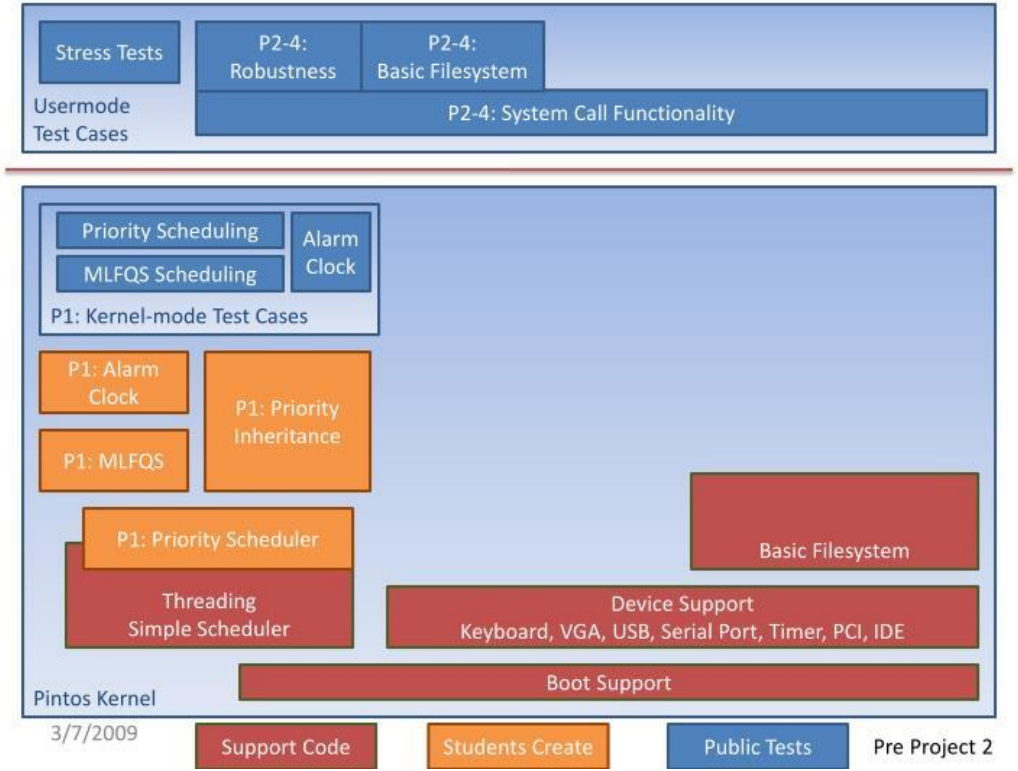
Pintos (1/8) - Pre Project 1



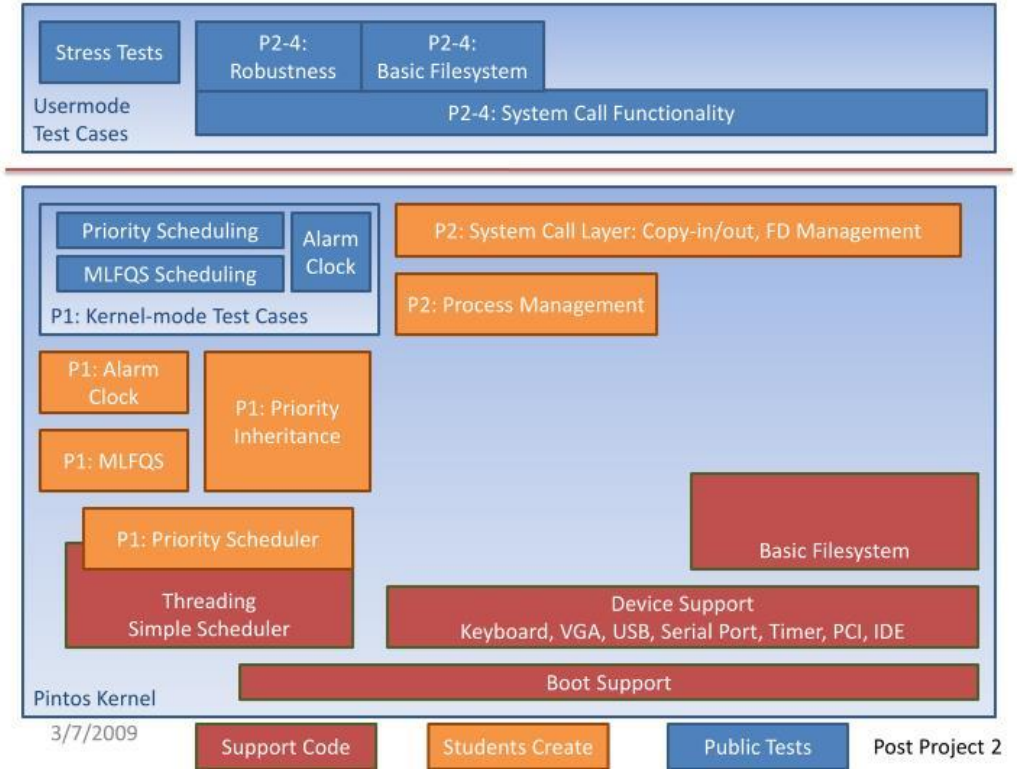
Pintos (2/8) - Post Project 1



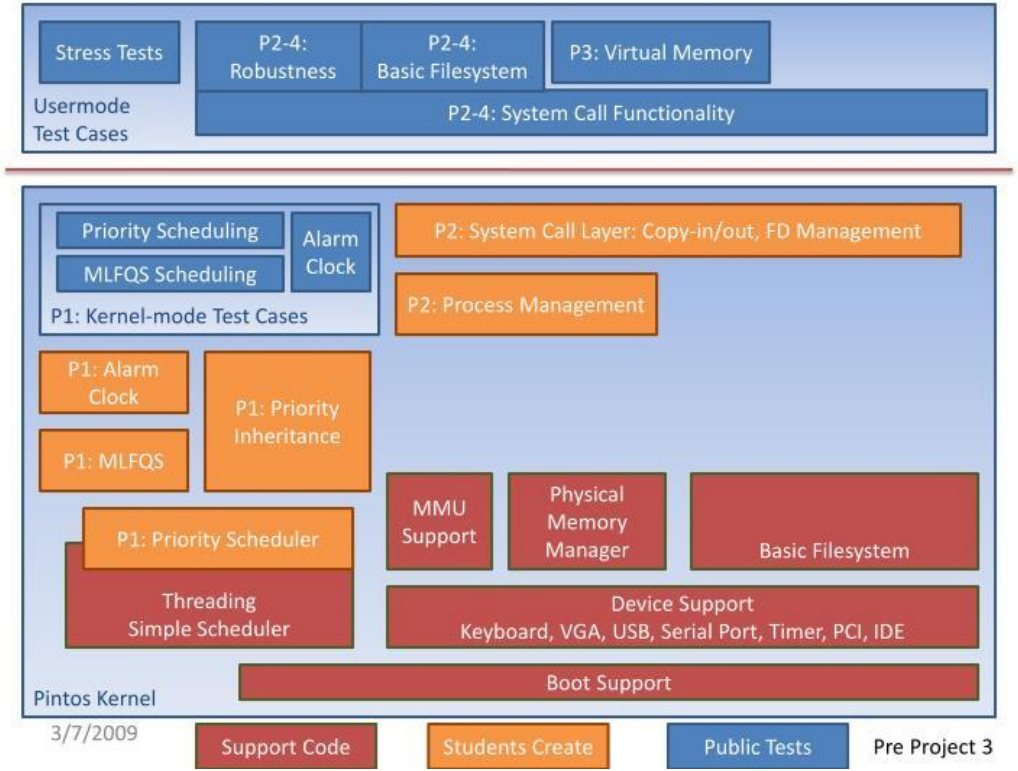
Pintos (3/8) - Pre Project 2



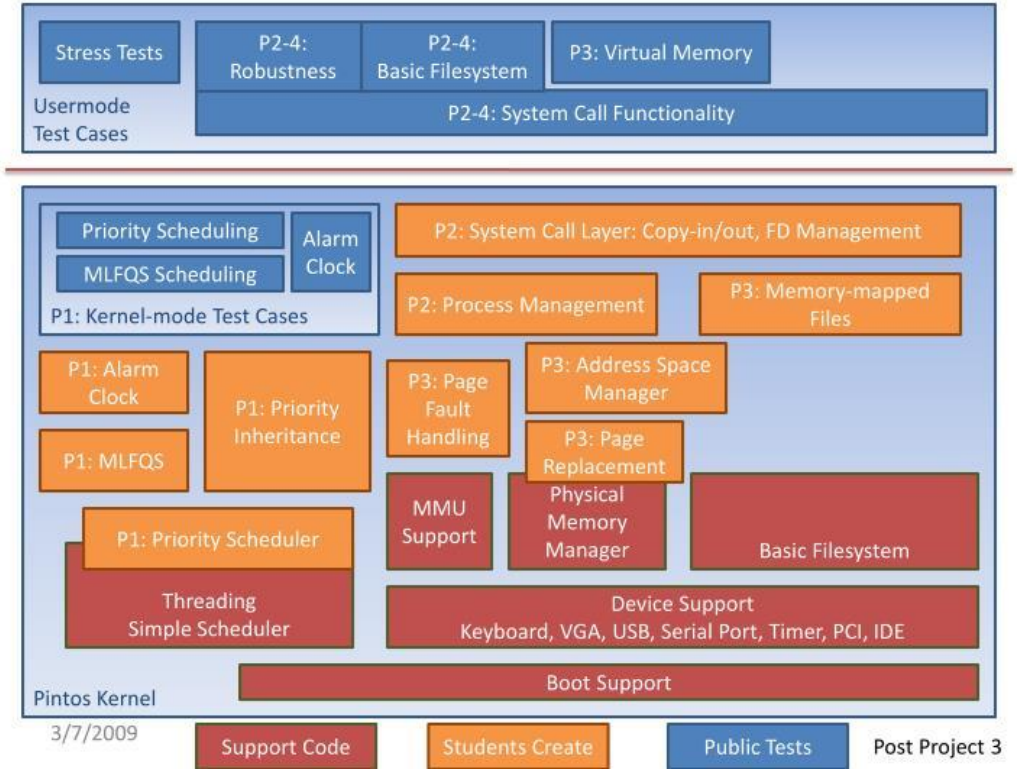
Pintos (4/8) - Post Project 2



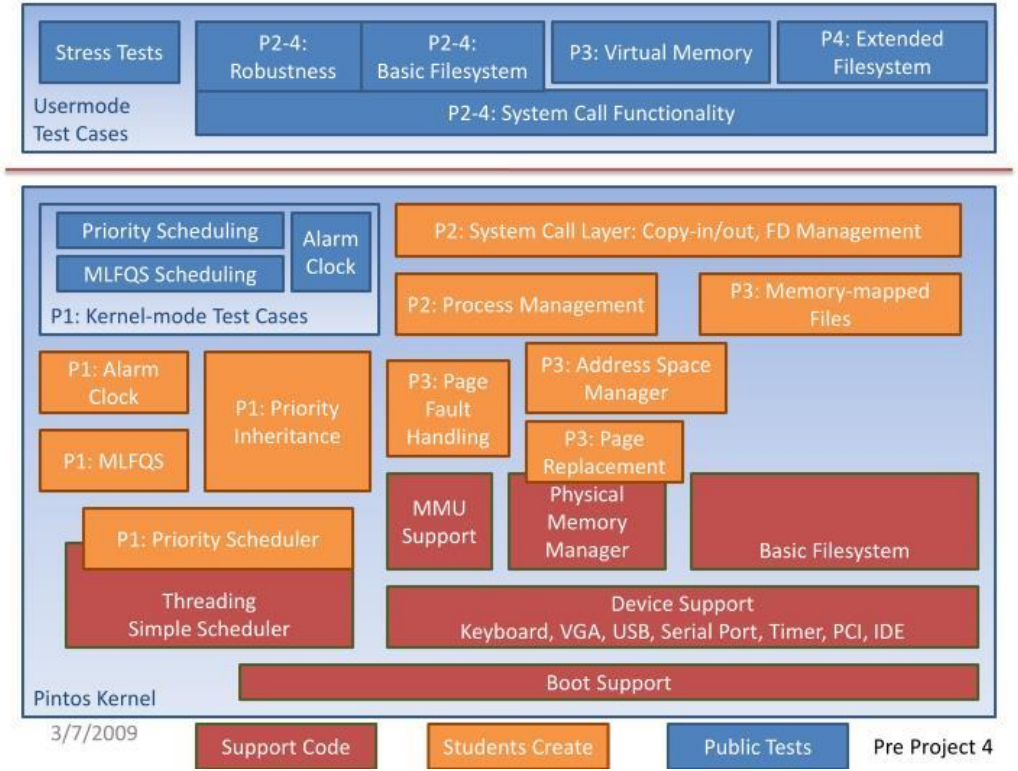
Pintos (5/8) - Pre Project 3



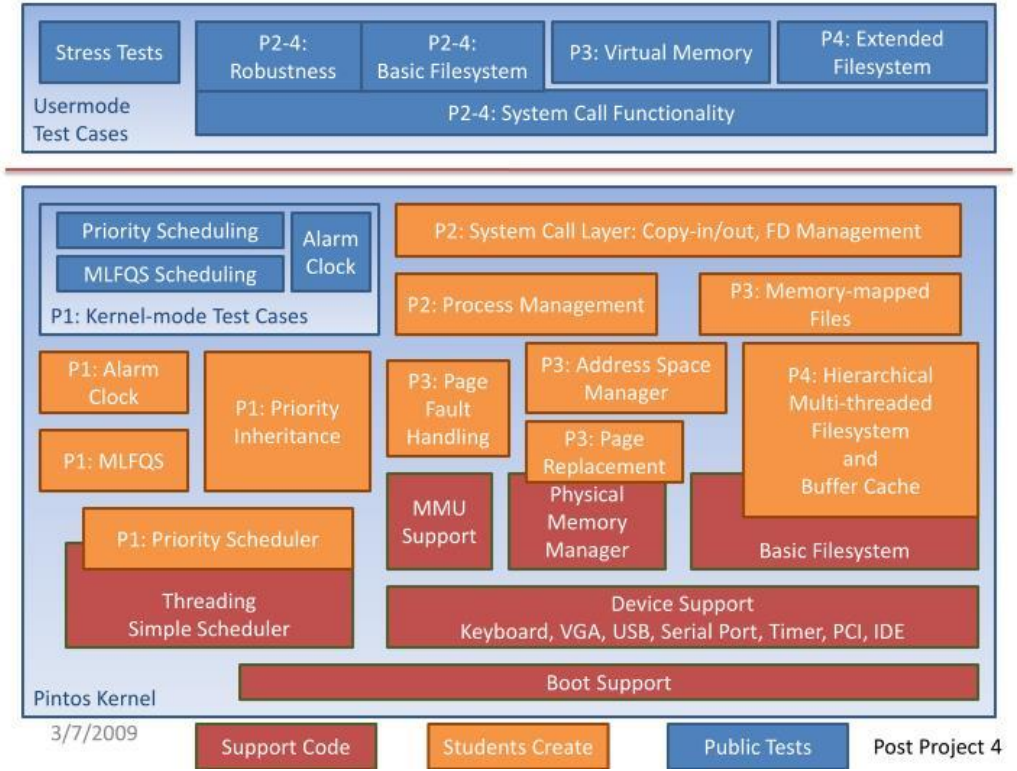
Pintos (6/8) - Post Project 3



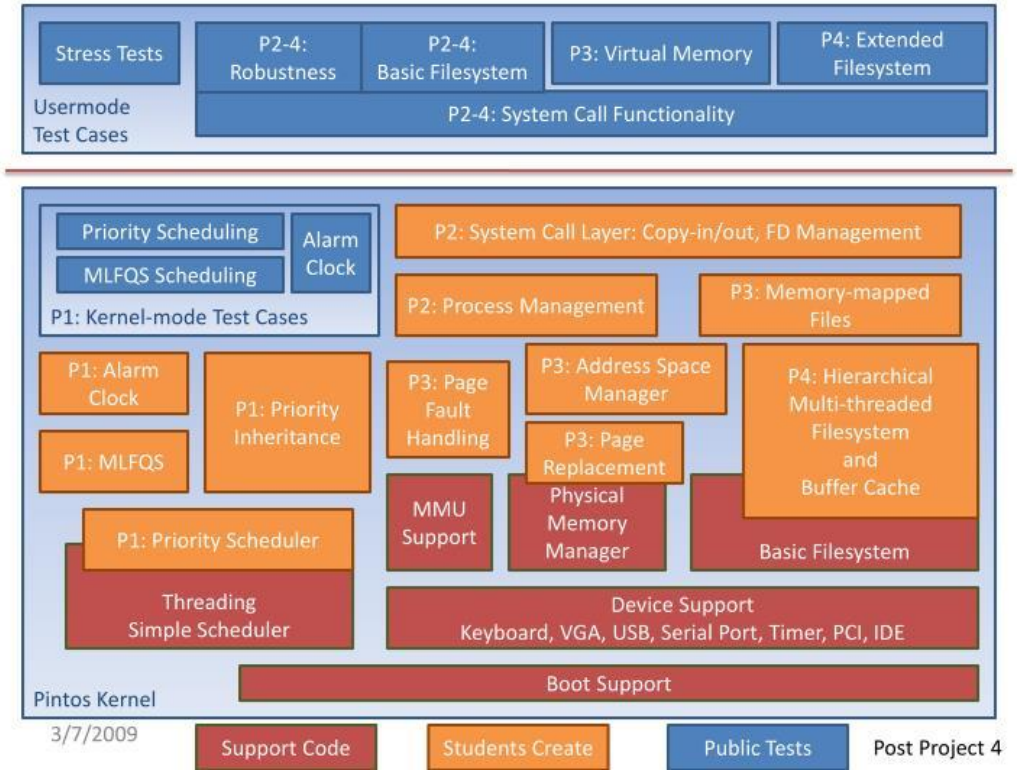
Pintos (7/8) - Pre Project 4



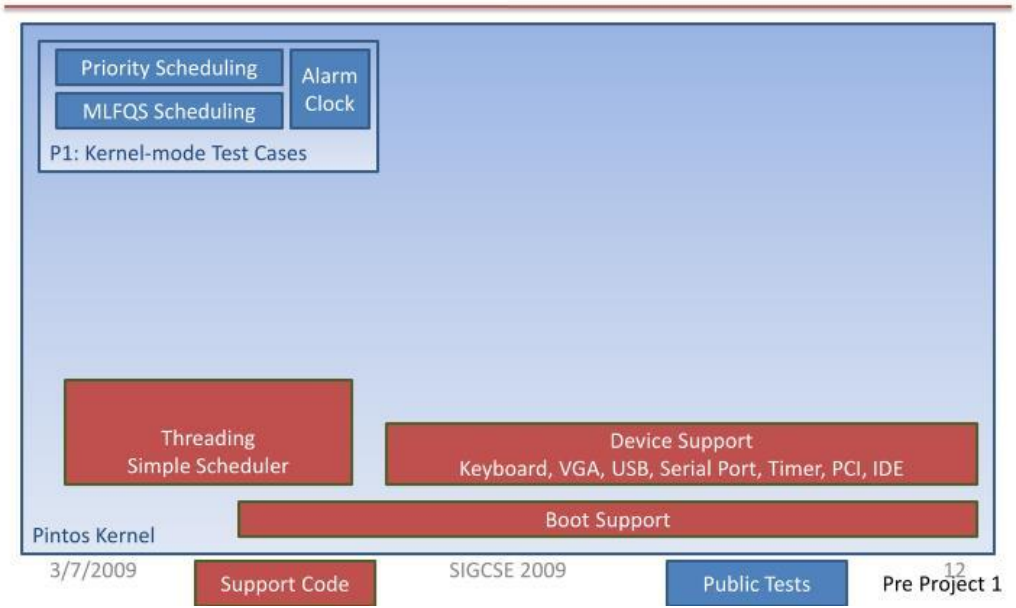
Pintos (8/8) - Post Project 4



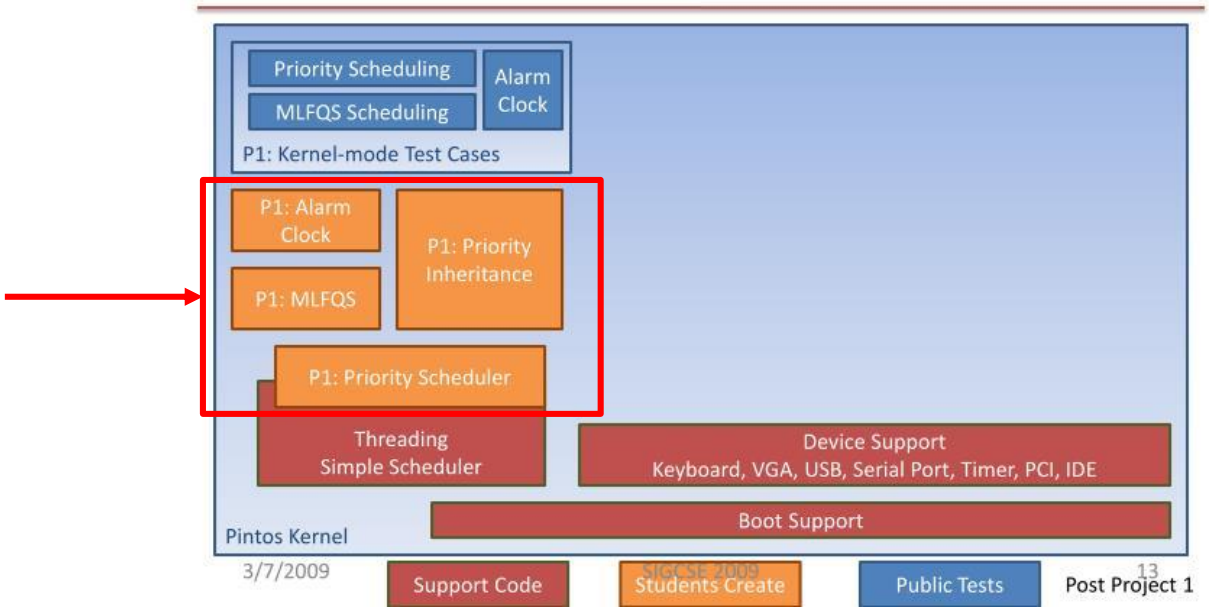
Pintos - After full implementation (Post Project 4)



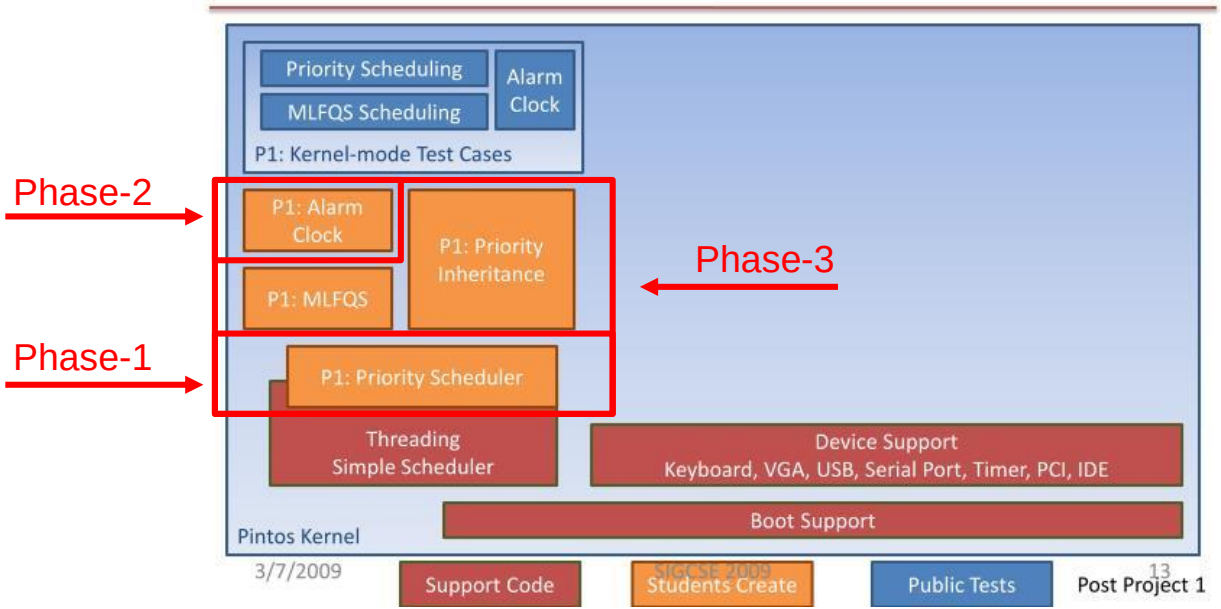
Pintos - You Start from Here (Pre Project 1)



Pintos - You Will Implement This (Post Project 1)



Pintos - You Will Implement This (Post Project 1)



Project-1: Threads

- Step 1: Preparation
- Step 2: Setting Up Pintos
- Step 3: Design Document
- Step 4: Implementation
- Step 5: Testing

Step 1: Preparation

Readings from zyBooks:

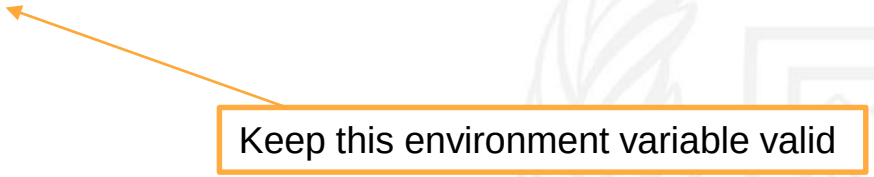
- Chapters 2, 3, 4

Readings from Pintos documentation:

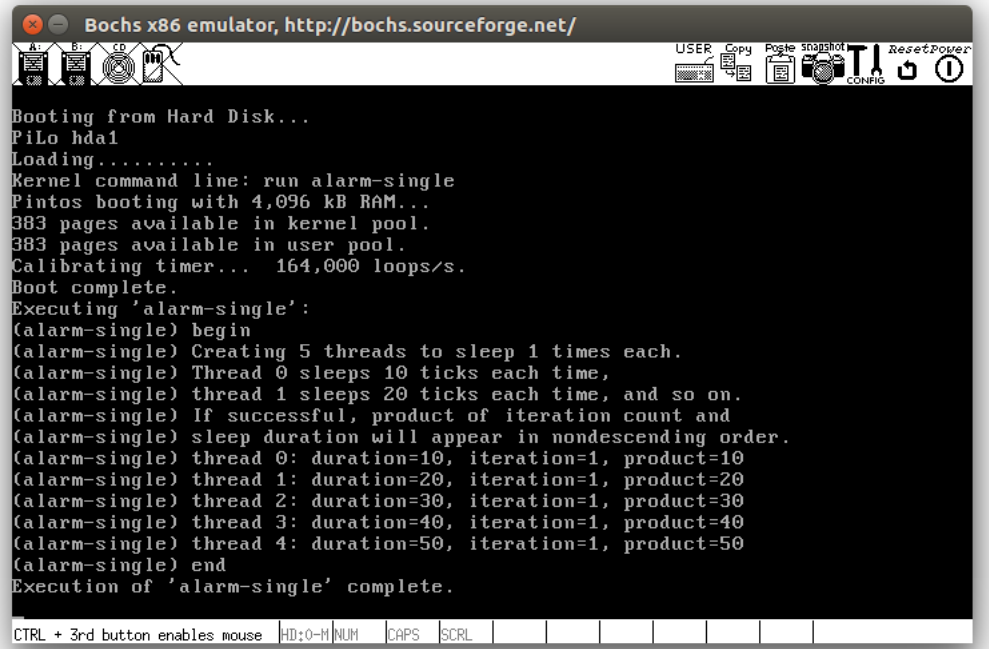
- Chapter 1 - Introduction
- Chapter 2 - Project 1: Threads
- Appendix A - Reference Guide [A.1, A.2, A.3, A.4]
- Appendix B - 4.4BSD Scheduler
- Appendix C - Coding Standards
- Appendix D - Project Documentation

Step 2: Setting Up Pintos

- Use the **VM** we have prepared for you
- Verify setup
  - Compile:
    - `cd $PINTOSDIR/src/threads`
    - `make`
  - Test:
    - `cd build`
    - `pintos run alarm-single`



Step 2: Setting Up Pintos



Get Familiar with the Code

- The first task is to read and understand the code for the initial thread system
  - Under the `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 `src/threads/` directory, please see **Section 2.1.2 Source Files** in the Pintos documentation.

Pintos Thread System

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

- These are **critical** steps.

Important Directories

- `src/threads/`

  Source code for the base kernel, which you will modify starting in project-1.

- `src/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.

- `src/lib/kernel/`

  Parts of the C library that are included only in the Pintos kernel. Feel free to reuse this code.

- `src/tests/`

  Tests for each project. You can read and modify this code to better understand your implementation. However, we will replace them with originals before we run tests.

Files of Interest

- `thread.c and thread.h`

  Basic thread support. Most of project-1 work.

- `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

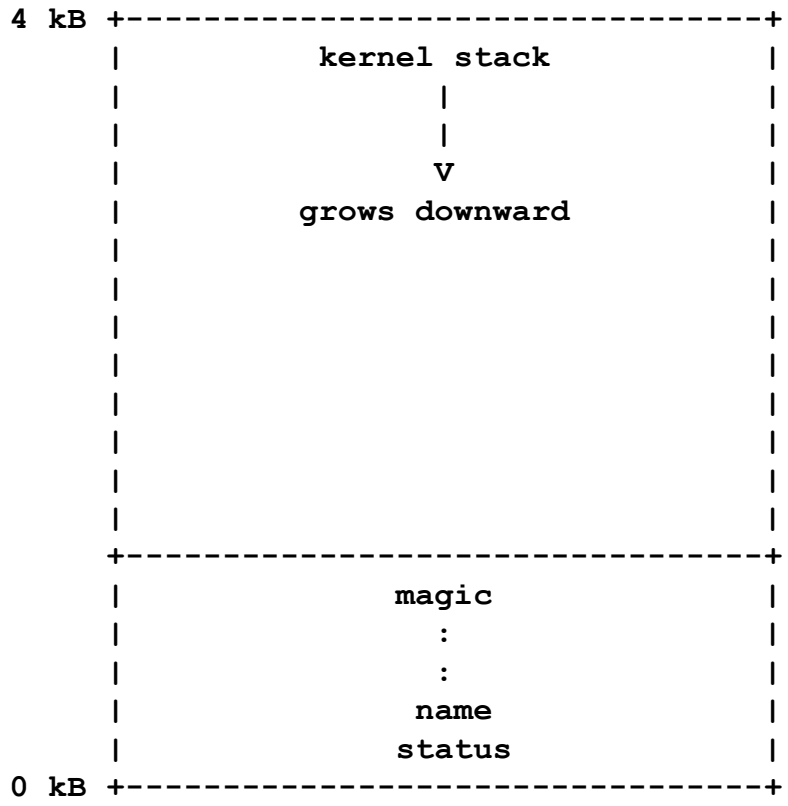
```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved 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. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

Pintos Thread System



Step 3: Design Document

Use the template in `doc/` directory:

- `threads.tmpl`
- `userprog.tmpl`
- `vm.tmpl`
- `filesys.tmpl`

- Copy the `threads.tmpl` file for your design doc submission.

Step 3: Design Document

```
+---------------------+
|        CS 140       |
| PROJECT 1: THREADS  |
|   DESIGN DOCUMENT   |
+---------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

FirstName LastName <email@domain.example>
FirstName LastName <email@domain.example>
FirstName LastName <email@domain.example>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation, course
>> text, lecture notes, and course staff.
```

Step 3: Design Document

```
                     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()?
```

Step 4: Implementation

Phase-1

1. Priority Scheduling

Phase-2

1. Alarm Clock

Phase-3

1. Priority Scheduling (cont.)
2. Priority Donation
3. Multilevel Feedback Queue Scheduler (MLFQS)

Effort and time needed: Phase-3 > Phase-2 > Phase-1

Task 1: Implement Priority Scheduler

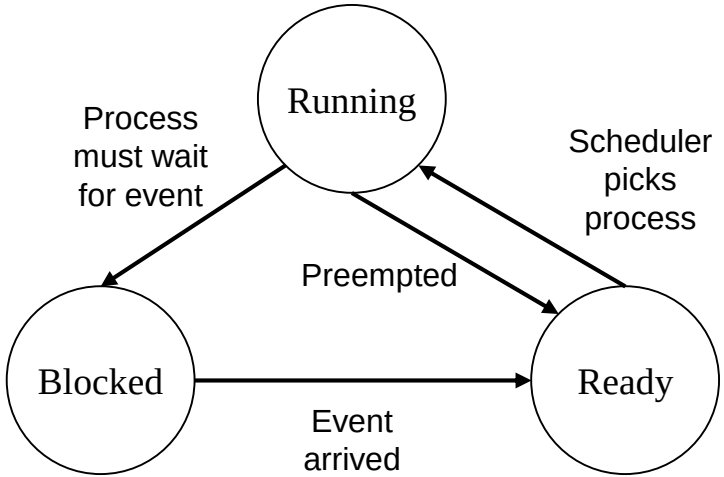
- Ready thread with highest priority gets the processor
- When a thread is added to the ready list that has a higher priority than the currently running thread, immediately yield the processor to the new thread
- Implementation details:
  - Compare priority of the thread being added to the ready list with that of the running thread (preemptive)
  - Select next thread to run based on priorities

Task 1: Implement Priority Scheduler

- 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 1: There is only 1 READY thread. Thread is rescheduled right away.
  - Case 2: There are other READY threads.
    - 2.a) Another thread has higher priority - It is scheduled
    - 2.b) Another thread has same priority - It is scheduled provided the previously running thread was inserted in tail of ready list.
    - 2.c) Other threads have lower priority - Current thread gets rescheduled

Task 1: Implement Priority Scheduler

- `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

Task 1: Implement Priority Scheduler

- One or more threads are in `ready_list`, and getting scheduled by the scheduler.
- Where should we intervene (i.e. preempt current thread)?
  - Change to current threads. E.g. if priority of a thread changes
    - `void thread_set_priority (int new_priority)`
  - A new thread with higher priority. I.e. creation of a new thread.
    - `tid_t thread_create (const char *, int, thread_func *, void *)`

Task 2: Implement Alarm Clock

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

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
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 points!

Task 3: Implement Priority Scheduler

- When threads are blocked and waiting for a lock, semaphore or a condition variable, the highest priority waiting thread should be woken up first.
- Implementation details:
  - Compare priorities of waiting threads when releasing locks, semaphores, condition variables.

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

  prio(*H*) > prio(*M*) > 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 Pathfinder, a watchdog timer noticed that H failed to run for some time, and would reset the system.

Task 4: 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*
- Recall the donation once *L* releases the lock.

Important:

- Remember to return *L* to previous priority once it releases the lock.
- Be sure to handle multiple donations (Multiple threads donating to a single thread)
- Be sure to handle nested donations (*H* waits on *M*, which waits on *L*, ...) up to 8 levels

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.
- **You should NOT do this, unless it is necessary!**
- Instead, use **semaphores**, **locks**, and **condition variables** to solve the bulk of your synchronization problems.
- **Exception:**

  The only place you are allowed to turn interrupts off is, when coordinating data shared between a kernel thread and interrupt handler. Because interrupt handlers can't sleep, they can't acquire locks.
- **Turning the interrupts off for synchronization between kernel threads (where it's not necessary) will lose points.**

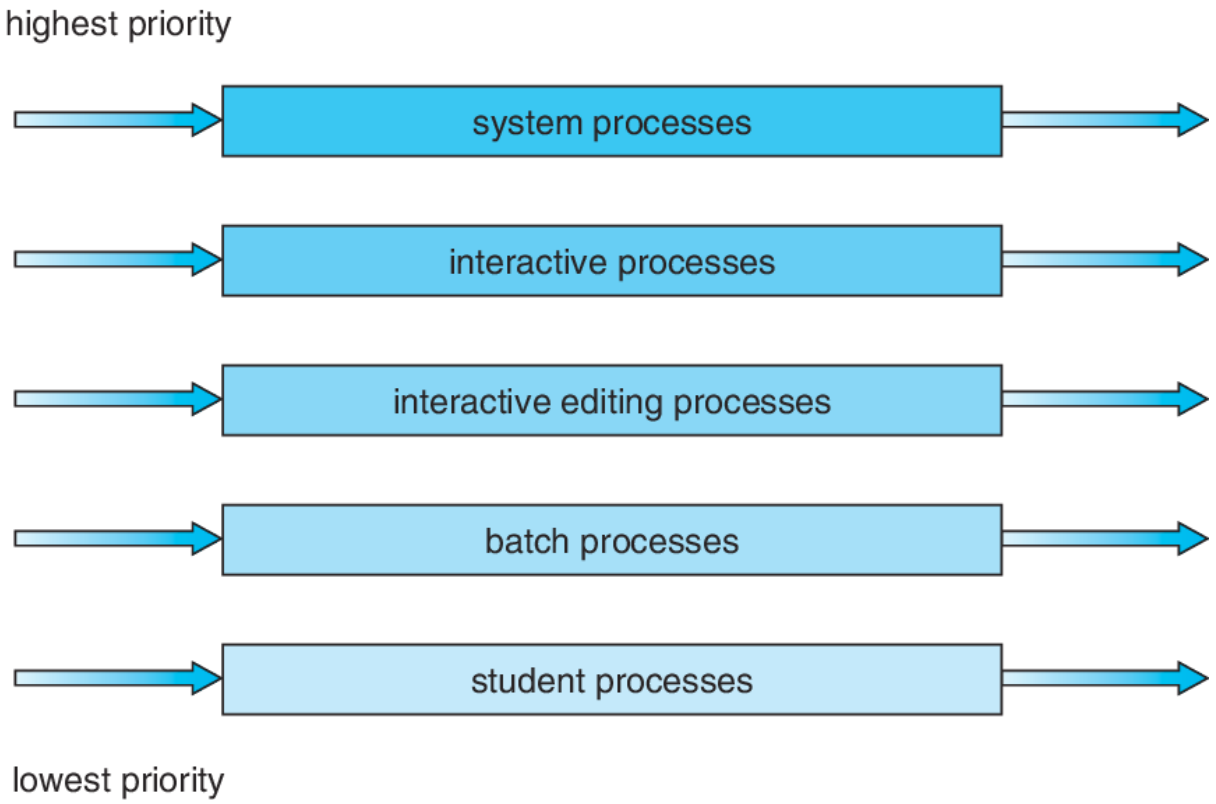
Task 5: Implement Advanced Scheduler

- Implement Multilevel Feedback Queue Scheduler
- Priority donation not needed in the advanced scheduler
  - Only one is active at a time
- Advanced scheduler must be chosen only if "`-mlfqs`" kernel option is specified

```
/* If false (default), use round-robin scheduler.
   If true, use multi-level feedback queue scheduler.
   Controlled by kernel command-line option "-o mlfqs". */
bool thread_mlfqs;
```

- Read Appendix B - 4.4BSD Scheduler in Pintos manual for detailed information.
- Some of the parameters are real numbers and calculation involving them have to be simulated using integers.

Multilevel Queue Scheduler

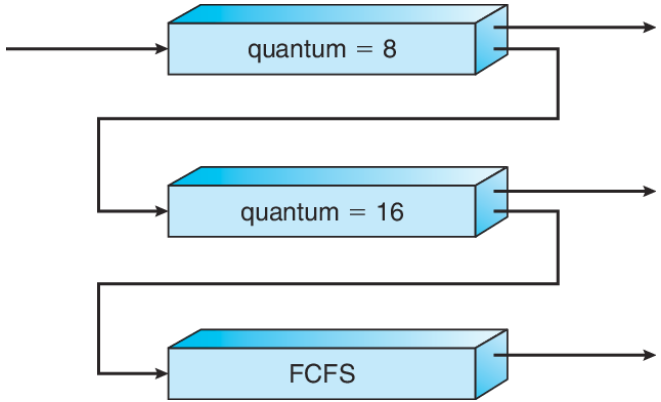


Multilevel Feedback Queue Scheduler

- 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 Scheduler

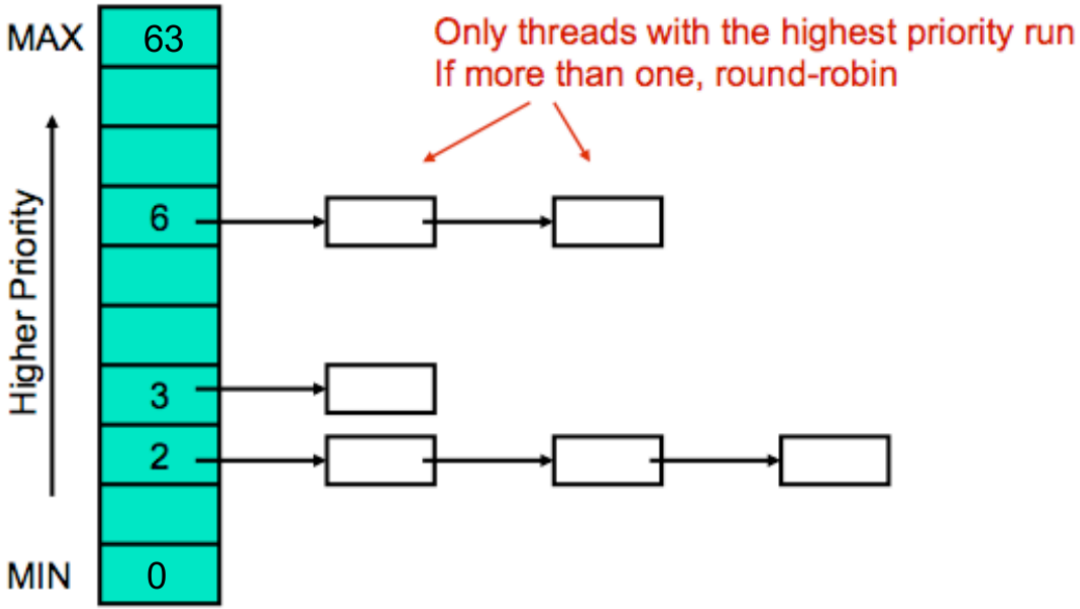
- Three queues:
  - Q_0 - RR with q = 8 ms
  - Q_1 - RR with q = 16 ms
  - Q_2 - FCFS



- 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

MLFQS: 4.4BSD Priority Based Scheduler

4.4BSD scheduler has 64 priorities and thus 64 ready queues, numbered 0 (PRI_MIN) through 63 (PRI_MAX).



MLFQS: 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:

priority = PRI_MAX - (recent_cpu / 4) - (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)

MLFQS: "*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.

Read Pintos manual Section "B. 4.4BSD Scheduler" for details

MLFQS: 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 a exponentially weighted moving average:
  - **recent_cpu(0) = 0** for initial thread, and parent thread's value for other threads.
  - At each timer interrupt, **recent_cpu** incremented by 1 for the running thread.
  - And once per second, for each thread:

$$a = (2 * load\_avg) / (2 * load\_avg + 1)$$

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

Read Pintos manual Section "B. 4.4BSD Scheduler" for details

MLFQS: 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:

  load_avg(t) = (59 / 60) * load_avg(t - 1) + (1 / 60) * ready_threads

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

Read Pintos manual Section "B. 4.4BSD Scheduler" for details

Functions to implement

Skeletons of these functions are provided in `src/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

Phase-1

- Priority Scheduler
  - Needed for implementing Priority Donation and Advanced Scheduler

Phase-2

- Alarm Clock
  - Needs more understanding of Pintos and synchronization.
  - Other parts do not depend on this

Phase-3

- Priority Scheduler (cont.)
- Priority Donation
- Advanced Scheduler

Debugging your Code

- printf, ASSERT, backtraces, gdb
- Running pintos under gdb
  - Invoke pintos with the gdb option (Note the spaces and hyphens).
    - `pintos --gdb -- run testname` Do not copy command from PDF
  - On another terminal from build/ directory, 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.
  - Psst… Use the pintos debugging macros described in manual (e.g. `dumplist`)

How Much Code?

```
devices/timer.c           |   42 +++++-
threads/fixed-point.h     |   120 ++++++++++++++++++
threads/synch.c           |   88 ++++++++++++-
threads/thread.c          |   196 ++++++++++++++++++++++++++++++++----
threads/thread.h          |   23 +++
5 files changed, 440 insertions(+), 29 deletions(-)
```

- This reference solution represents just one possible solution.

Step 5: Testing

- Pintos provides a very systematic testing suite for your project:
  - Compile:
    - `make clean`
    - `make`
  - Run all tests (pass/fail):
    - `make check`
  - Run individual tests:
    - `make build/tests/threads/alarm-multiple.result`
  - Run the grading script (gives useful information):
    - `make grade`

Step 5: Testing

- **`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
20 of 27 tests failed.
```

Grading

- **make grade**
- src/threads/build/grade:

```
TOTAL TESTING SCORE: 24.2%
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                        Pts Max  % Ttl  % Max
----------------------------------------------- --- --- ------ ------
tests/threads/Rubric.alarm                       14/ 18  15.6%/ 20.0%
tests/threads/Rubric.priority                     0/ 38   0.0%/ 40.0%
tests/threads/Rubric.mlfqs                        8/ 37   8.6%/ 40.0%
----------------------------------------------- --- --- ------ ------
Total                                                    24.2%/100.0%
```

Grading - Alarm Clock: 14/18 pts

```
    4/ 4 tests/threads/alarm-single
    4/ 4 tests/threads/alarm-multiple
    4/ 4 tests/threads/alarm-simultaneous
** 0/ 4 tests/threads/alarm-priority

    1/ 1 tests/threads/alarm-zero
    1/ 1 tests/threads/alarm-negative
```

Phase-2

```
- Section summary.
      5/  6 tests passed
     14/ 18 points subtotal
```

All tests in Phase-1 are included in Phase-2.

- If alarm clock implementation is based on “busy-waiting”, or
- If interrupts are turned off excessively for synchronization (between kernel threads)
  - ➢ **You will lose points**

Grading - Priority Scheduler: 0/38 pts

```
** 0/ 3 tests/threads/priority-change
** 0/ 3 tests/threads/priority-preempt

** 0/ 3 tests/threads/priority-fifo
** 0/ 3 tests/threads/priority-sema
** 0/ 3 tests/threads/priority-condvar

** 0/ 3 tests/threads/priority-donate-one
** 0/ 3 tests/threads/priority-donate-multiple
** 0/ 3 tests/threads/priority-donate-multiple2
** 0/ 3 tests/threads/priority-donate-nest
** 0/ 5 tests/threads/priority-donate-chain
** 0/ 3 tests/threads/priority-donate-sema
** 0/ 3 tests/threads/priority-donate-lower

- Section summary.
      0/ 12 tests passed
      0/ 38 points subtotal
```

Phase-1

Phase-3

Grading - MLFQ Scheduler: 8/37 pts

```
** 0/ 5 tests/threads/mlfqs-load-1
** 0/ 5 tests/threads/mlfqs-load-60
** 0/ 3 tests/threads/mlfqs-load-avg

** 0/ 5 tests/threads/mlfqs-recent-1

   5/ 5 tests/threads/mlfqs-fair-2
   3/ 3 tests/threads/mlfqs-fair-20

** 0/ 4 tests/threads/mlfqs-nice-2
** 0/ 2 tests/threads/mlfqs-nice-10

** 0/ 5 tests/threads/mlfqs-block

- Section summary.
      2/  9 tests passed
      8/ 37 points subtotal
```

Phase-3

All tests in “make check” are included in Phase-3.

Grading

- Your “source code” consists of weighted average of the three phases.
  (10%, 25%, 65% and Due dates: Sep 17th, Oct 1st, Oct 15th)
- Your “design document” is 10% of your Project-1 grade, and “source code” is 90%

- The points are weighted. `make grade` will give you a score out of 100% for your code, based on the passed tests and their weight.
- You can consider that for summary of your tests. Our grading scheme is different, as our requirements for phase-1, phase-2, phase-3 are different.
- Check autograder submission score to be consistent with what you get on your VM.
  - Reminder: points will be taken off for busy-waiting and/or turning interrupts off unnecessarily.

Submission

- Submission will be via AutoLab autograder.
  - The instructions will be posted on Piazza.
  - You'll have unlimited submissions, submit early and re-submit.
  - Every phase needs registration, group formation, submission etc. It is the responsibility of both members to ensure this.

- Due days and times are Fridays 11:59 PM Eastern Time

- Refer to LATE SUBMISSION policy.

Assignments

- If not yet, submit/join your groups on github classroom.
- If you do not have a group, post a private note on Piazza.
- Understand the initial pintos code in the files related to Project-1.
  - thread.c , thread.h
  - timer.c , timer.h
  - init.c , init.h
  - interrupt.c , interrupt.h
  - synch.c , synch.h
  - list.c , list.h
- Get started!!!
- Reading: Chapter 4 (Concurrency) from zyBooks

Summary

- Pintos projects:
  - Project-1: Threads
    - Step 1: Preparation
    - Step 2: Setting Up Pintos
    - Step 3: Design Document
    - Step 4: Implementation
    - Step 5: Testing
  - Project-2: User Programs
  - Project-3: Virtual Memory
  - Project-4: File Systems

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from University of Nevada, Reno
- T. Kosar and K. Dantu from University at Buffalo
- Pintos Manual
- Pintos Notes and Slides by A. He (Stanford), A. Romano (Stanford), J. Sundararaman & X. Liu(Virginia Tech), J. Kim (SKKU)