CS143A
Principles of Operating Systems
Discussion 04: Project 1

Instructor: Prof. Ardalan Amiri Sani
TA: Ping-Xiang (Shawn) Chen

Acknowledgement

The slides are based on the previous discussions from Dr. Saehanseul Yi.

Agenda

- Project 1
- Testing
- Tips

Agenda

- Project 1
- Testing
- Tips

Lab 1: Threads

- Team project: 1-3 people per team
- Grading
  - Design DOC 40%
  - Implementation 60%
- What you need to implement in pintos?
  - Alarm Clock
  - Priority Scheduling
  - Advanced Scheduling

Design Doc

- Coding Standard
- Be careful of long lines, capitalization, spelling, variable names,…
- Coding style
  - Indentation, cramming multiple sentences in a single line, …
  - Descriptive and informative comments
  - Return value checking; error handling (excessive use of ASSERT)
  - If there's repeating code, wrap it as a function and call it multiple times

Background

- Project 1: Threads
  - https://www.ics.uci.edu/~ardalan/courses/os/pintos/pintos_3.html#SEC25
    - Source code structure
    - Development suggestions
- A.3 Synchronization
  - https://www.ics.uci.edu/~ardalan/courses/os/pintos/pintos_7.html#SEC110
    - Disabling interrupts vs. locks
    - Handling interrupts
    - Semaphore/locks

Background

- A.2 Threads
  - https://www.ics.uci.edu/~ardalan/courses/os/pintos/pintos_7.html#SEC106
    - Thread status
    - Thread functions
    - Thread switching
- Advanced scheduler (BSD Scheduler)
  - https://www.ics.uci.edu/~ardalan/courses/os/pintos/pintos_8.html#SEC14
    - Niceness
    - Calculating Priority…
    - Fixed-point real arithmetic

Implementation

```
pingxiac@circinus-48 11:15:41 ~/Pintos/project_1/pintos
$ git diff --stat HEAD
 src/devices/timer.c        |  43 +++++++++++++++++--
 src/threads/fixed-point.h  | 120 ++++++++++++++++++++++++++++++++++++++++++++++++++++
 src/threads/init.c         |  31 +++++++++++++-
 src/threads/synch.c        |  96 +++++++++++++++++++++++++++++++++++++++++++++++---
 src/threads/thread.c       | 195 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++----------
 src/threads/thread.h       |  22 +++++++++-
 6 files changed, 476 insertions(+), 31 deletions(-)
```

(1) Alarm Clock

- Reimplement
  - void timer_sleep (int64_t ticks)
    - busy loop -> timed wait using semaphore
    - Put the thread in ready queue when ready (after x ticks + alpha)
    - ready queue must be accessed atomically
- In short,
  - while(1) {A; B; C;}
  - vs. lock(l)

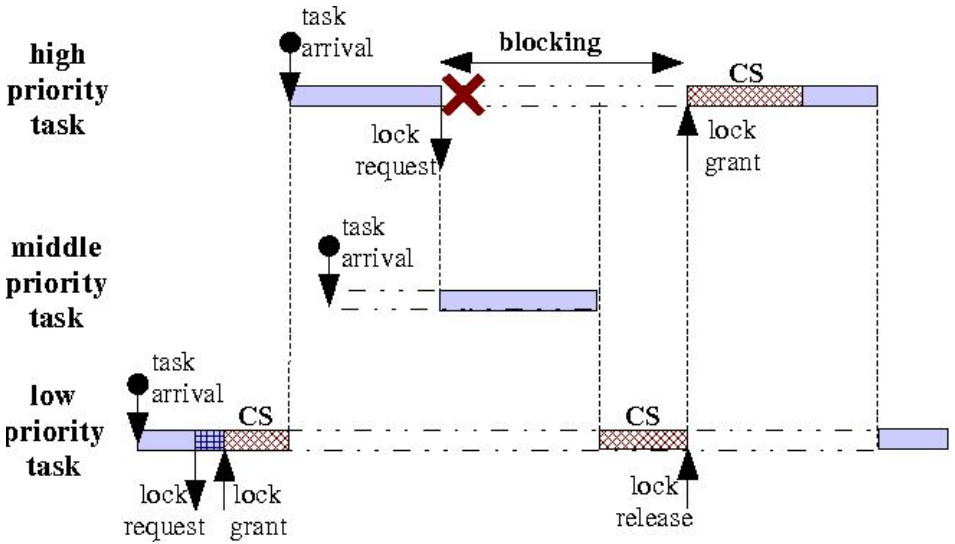
```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

(2) Priority Scheduling

- When a higher priority thread arrives, the current thread must yield immediately
  - What is the function that creates a thread?
- When threads are waiting (lock, semaphore, cond var), higher priority thread must be awakened first
  - What are the functions that make threads awake for each {lock, semaphore, cond var}?
- A thread may lower or raise its priority at runtime (it will be tested)
  - Must immediately yield the CPU when necessary

(2) Priority Scheduling

- Priority Inversion
  - 3 processes with different priorities (L, M, H)
  - L thread is holding the lock that H wants H must wait until L release the lock
- Problem:
  - while L is running in its critical section, another thread, M comes in and runs before L. This is possible because M has nothing to do with the lock
- What we want:
  - Finish L's CS first, then H, then M, then L's remaining section
- Solution: **Priority Inheritance**



Akgul, B. S. et al. "Hardware support for priority inheritance." RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003 (2003): 246-255.

(2) Priority Scheduling

- Priority Inheritance
  - Temporarily promote L's priority to H's so that no other threads whose priority is lower than H cannot preempt
  - In this example, B (Med) cannot preempt A (Low) because A has been promoted to priority 3 (High)
  - After C (High) acquiring the lock, put A's priority back to the original value

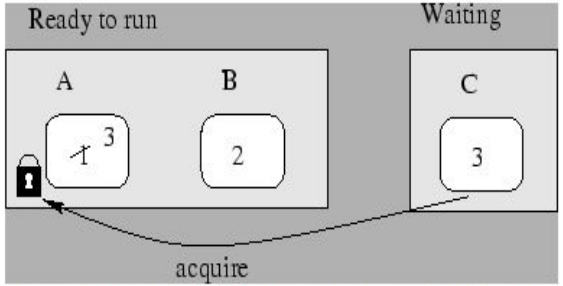


Figure 3: An example of priority donation. Thread C, with priority 3, donates its priority to thread A, with real priority 1. Thread A may now run.

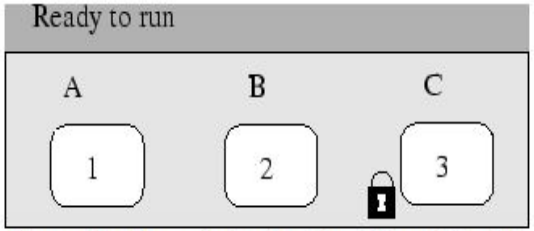
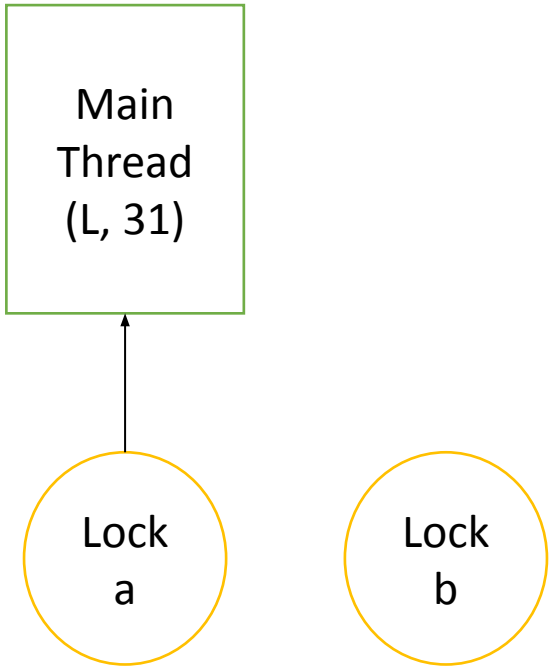


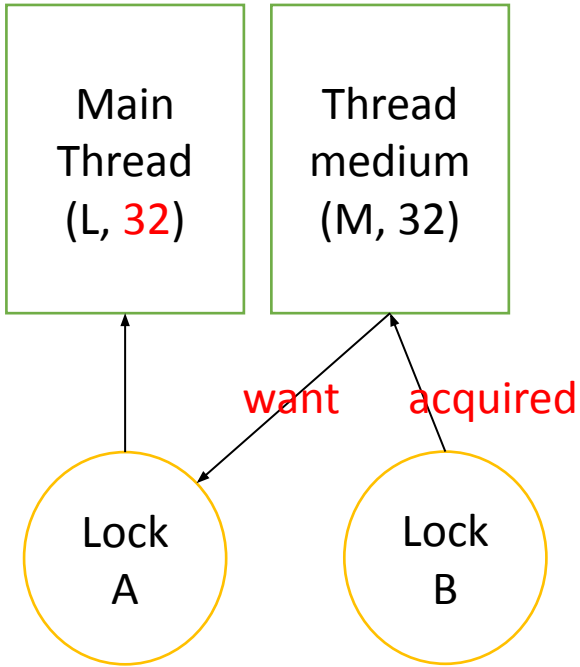
Figure 4: After priority donation. Thread C, with priority 3, is now able to run and thread A's priority dropped back down to 1.

Priority – Nested Inheritance



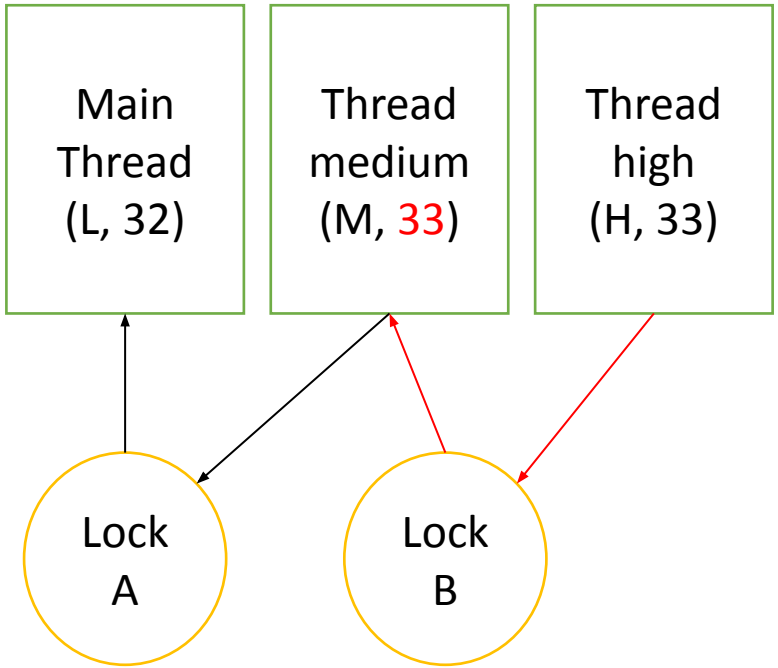
Main thread (L) acquired Lock A

Priority – Nested Inheritance



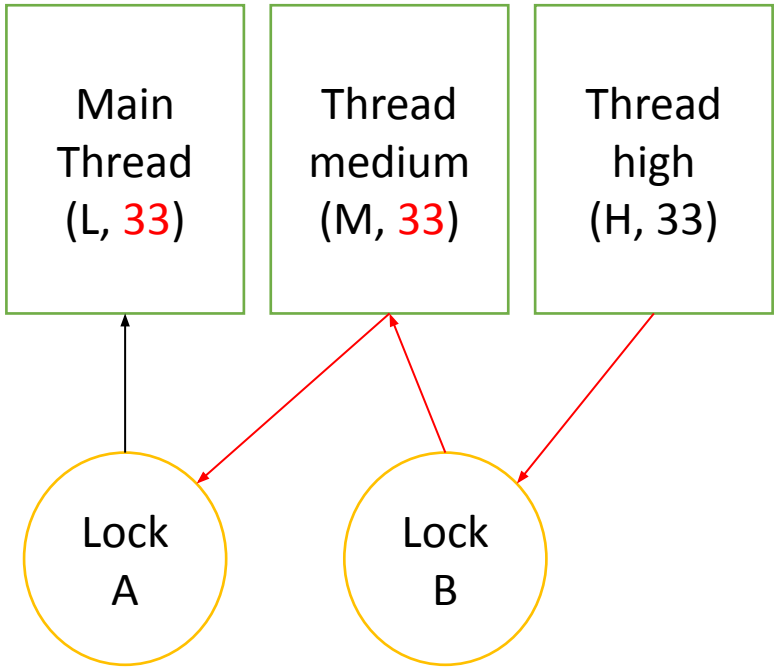
Main thread (L) acquired Lock A
M thread wants Lock A & Lock B
M thread waits for Lock A
M thread acquired Lock B
L's priority is promoted to M thread's

Priority – Nested Inheritance



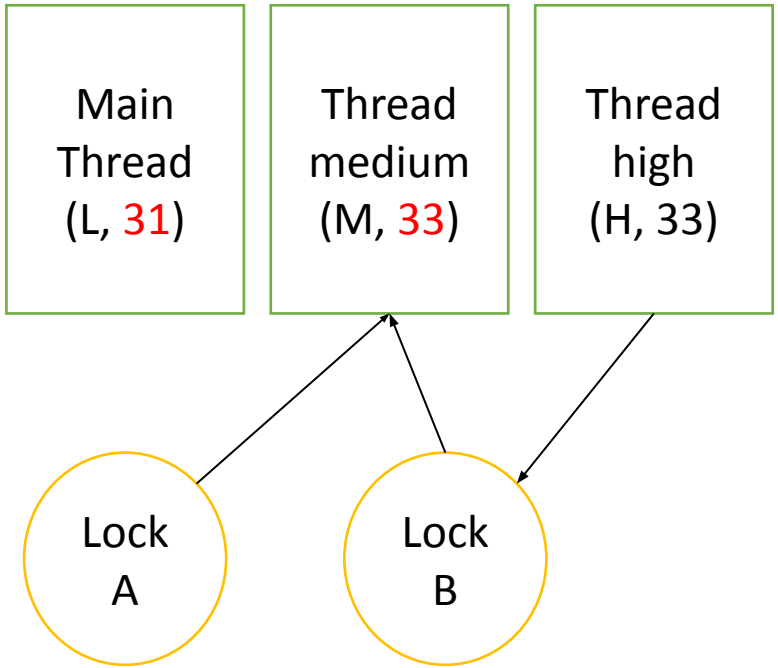
Main thread (L) acquired Lock A
M thread wants Lock A & Lock B
M thread waits for Lock A
M thread acquired Lock B
L thread's priority is promoted to M thread's
H thread arrives and wants Lock B
M thread's priority is promoted

Priority – Nested Inheritance



Main thread (L) acquired Lock A
M thread wants Lock A & Lock B
M thread waits for Lock A
M thread acquired Lock B
L thread's priority is promoted to M thread's
H thread arrives and wants Lock B
M thread's priority is promoted
L thread's priority is promoted again

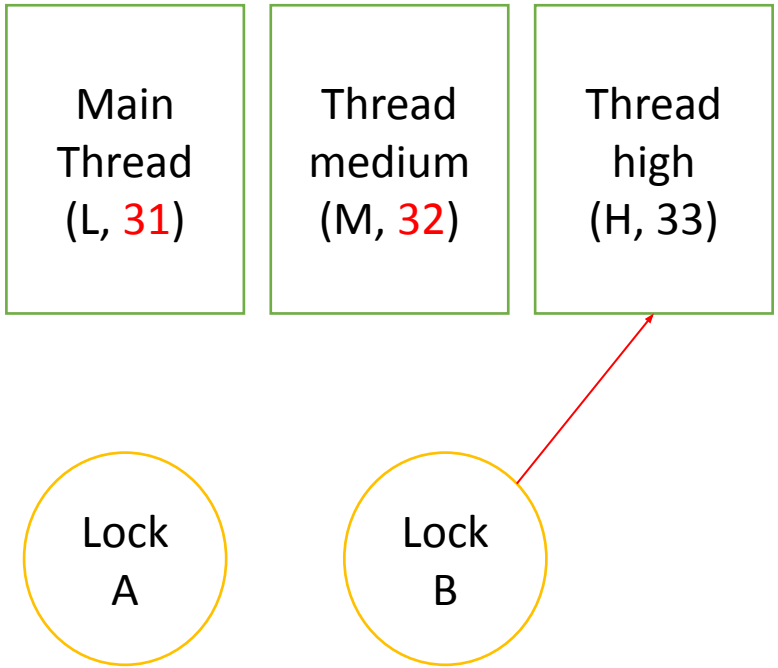
Priority – Nested Inheritance



Main thread (L) releases Lock A

L thread restores its original priority

Priority – Nested Inheritance



Main thread (L) releases Lock A

L thread restores its original priority

M thread releases Lock B

M thread restores its original priority

(3) Advanced Scheduler

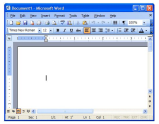
- A multilevel feedback queue scheduler (4.4 BSD scheduler)
  - reduce the average response time
- Priority based. But no priority donation
  - Implement Priority Scheduling first
- To enable mlfqs mode,
  - bool thread_mlfqs set it in init.c next to other option handlings
  - pintos -v -k -T $(TIMEOUT) --mlfqs
  - Giving timeout because of deadlock
  - You can use pintos --help for more information

```
/* 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;
```

(3) Advanced Scheduler

- Motivational example
  - Memory-bound task: threads perform a lot of I/O require a fast response time (MS Word) but little CPU time
  - Compute-bound task: threads require more CPU time to finish their work than the memory-bound ones. Less I/O requests
- Thread priority is dynamically determined by the scheduler
  - Sample policy:
    - priority = PRI_MAX - (recent_cpu / 4)

(3) Advanced Scheduler



Word processor (W)
- Interactive
- Less computation

File Compression (Z)
- No interaction
- Compute-intensive

	W	Z	Z	Z	W	Z	Z	W	Z	Z	W
W's Priority	**60**	59	59	60	**60**	59	59	**60**	59	59	**60**
Z's Priority	60	**60**	**59**	**58**	57	**58**	**57**	56	56	55	54

W is in sleep. its recent_cpu gets lower, and the priority goes up

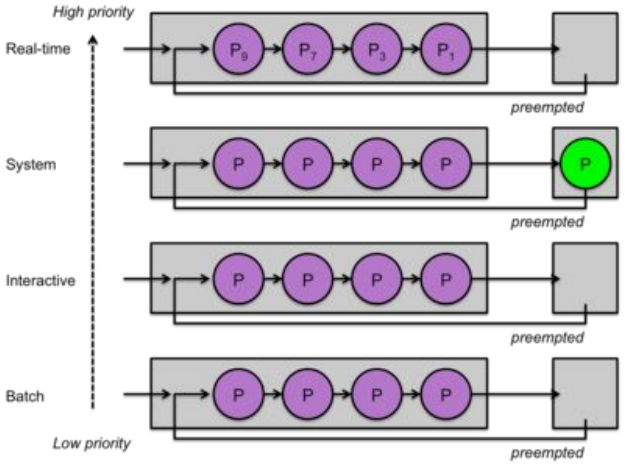
Z's recent_cpu gets higher, and the priority goes down

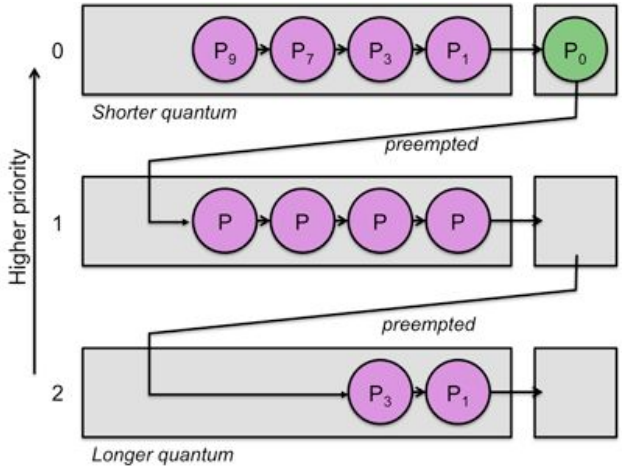
W processes and go to sleep (nothing to compute..)
W is not in the ready queue

Because W's priority is maintained high, whenever it is in the ready queue it will preempt other threads

Multilevel Feedback Queue

- Each value of priority(0~64) has its queue
- Problems of Multilevel queue
  - Convoy effect & Starvation because priorities are fixed
  - Processes cannot move across the queues
- Change the priority dynamically
  - If it was scheduled recently, lower the priority
  - Or we can use process aging (increase priority over time if idle)





Multilevel Feedback Queue

- Some parameters:
  - niceness: how “nice”(20~-19) to other threads.. more likely to yield if the nice value is higher
  - recent_cpu: if recently used, it increases. Decays over time
  - load_avg: a system-wide value. Decays over time
  - priority: a function of (recent_cpu, niceness)

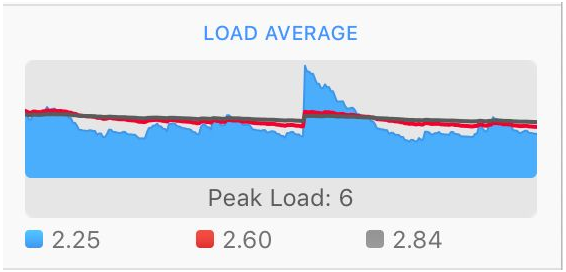
$$recent\_cpu = (2*load\_avg)/(2*load\_avg + 1) * recent\_cpu + nice,$$

recent_cpu++

$$load\_avg = (59/60)*load\_avg + (1/60)*ready\_threads,$$

If *load_avg* is 1, the weight decays to 0.1 approx. 6 sec if *load_avg* is 2, approx. 8 sec

$$priority = PRI\_MAX - (recent\_cpu / 4) - (nice * 2),$$



Multilevel Feedback Queue

- When do we update these parameters?
  - niceness: user specifically sets for each thread
  - recent_cpu: +1 when it's running, calculate below once in a while
  - load_avg: once in a while (every second?)
  - priority: calculate for every process when choosing next thread to run

$$recent\_cpu = (2*load\_avg)/(2*load\_avg + 1) * recent\_cpu + nice,$$

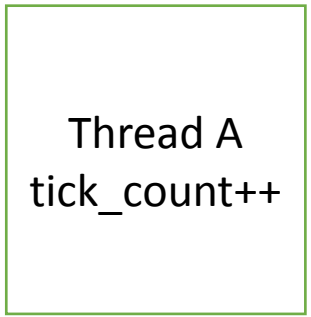
recent_cpu++

$$load\_avg = (59/60)*load\_avg + (1/60)*ready\_threads,$$

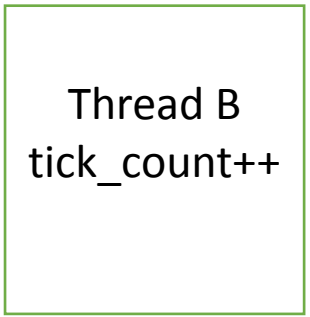
$$priority = PRI\_MAX - (recent\_cpu / 4) - (nice * 2),$$

MLFQS – nice-2 (fair-2, ..)

- Niceness: how nice to other threads (-20 ~ 20) high niceness: lower priorities and yield to other threads low niceness: increase the priority.



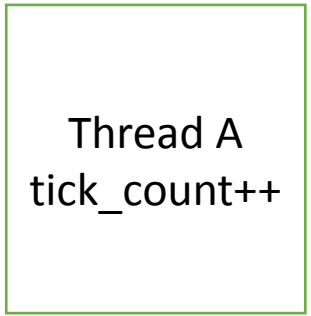
priority: 63
niceness: 0
recent-cpu: 0

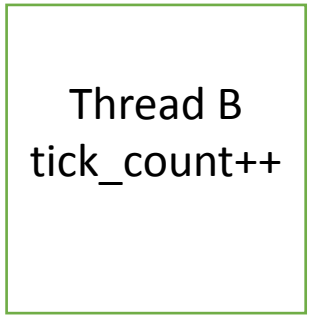


priority: 63
niceness: 5
recent-cpu: 0

MLFQS – nice-2 (fair-2, ..)

- Niceness: how nice to other threads (-20 ~ 20) high niceness: lower priorities and yield to other threads low niceness: increase the priority.





priority: 41
niceness: 0
recent-cpu: 5839.58
tick_count: 1900

(fair) tick_count: 1500

priority: 32
niceness: 5
recent-cpu: 5651.30
tick_count: 1100

tick_count: 1500

Multilevel Feedback Queue Tests

- Fairness: if thread workloads are the same, they are expected to get the same amount of CPU time
- Niceness: nice threads tend to yield to the other threads

```
squish-pty bochs -q
Pintos hda1^M
Loading...........^M
Kernel command line: -q -mlfqs run mlfqs-fair-2
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
Boot complete.
Executing 'mlfqs-fair-2':
(mlfqs-fair-2) begin
(mlfqs-fair-2) Starting 2 threads...
(mlfqs-fair-2) Starting threads took 6 ticks.
(mlfqs-fair-2) Sleeping 40 seconds to let threads run, please wait...
(mlfqs-fair-2) Thread 0 received 1501 ticks.
(mlfqs-fair-2) Thread 1 received 1500 ticks.
(mlfqs-fair-2) end
Execution of 'mlfqs-fair-2' complete.
Timer: 4059 ticks
Thread: 1000 idle ticks, 3062 kernel ticks, 0 user ticks
Console: 630 characters output
Keyboard: 0 keys pressed
Powering off..
```

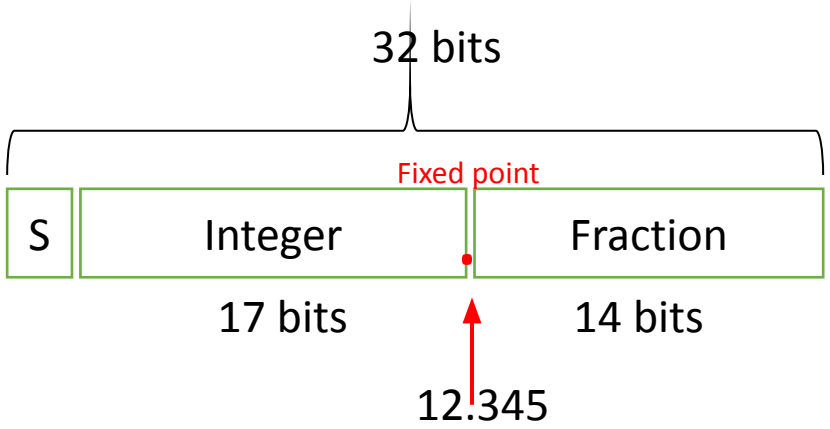
```
squish-pty bochs -q
Pintos hda1^M
Loading...........^M
Kernel command line: -q -mlfqs run mlfqs-nice-2
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  204,600 loops/s.
Boot complete.
Executing 'mlfqs-nice-2':
(mlfqs-nice-2) begin
(mlfqs-nice-2) Starting 2 threads...
(mlfqs-nice-2) Starting threads took 6 ticks.
(mlfqs-nice-2) Sleeping 40 seconds to let threads run, please wait...
(mlfqs-nice-2) Thread 0 received 1916 ticks.
(mlfqs-nice-2) Thread 1 received 1085 ticks.
(mlfqs-nice-2) end
Execution of 'mlfqs-nice-2' complete.
Timer: 4059 ticks
Thread: 1000 idle ticks, 3062 kernel ticks, 0 user ticks
Console: 630 characters output
Keyboard: 0 keys pressed
Powering off...
```

Multilevel Feedback Queue Summary

- A queue for each priority
- Task priorities can change dynamically
  - Parameters and rules can vary depending on implementation
  - A thread can move across the queues
  - It can achieve a good balance between I/O bound and compute bound processes
  - Fights the convoy effect and starvation

Fixed-point Real Arithmetic

$$12.345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10^{\overbrace{-3}^{\text{exponent}}}}_{\text{base}}$$



- Floating-point arithmetic
  - Exponent can change (floating point)
  - Large coverage of numbers
  - Expensive computation
  - Typically performed by Floating Point Unit (FPU)

- Fixed-point arithmetic
  - Fixed size for integer and fraction parts
  - Small coverage of numbers
  - Less computation
  - CPU's ALU can handle it fast enough

Agenda

- Project 1
- Testing
- Tips

A lot of tests!

```
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%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
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
      ** 0/ 4 tests/threads/alarm-priority

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

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

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

```
Functionality of advanced scheduler (tests/threads/Rubric.mlfqs):
      ** 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
```

Where are the tests?

- pintos/src/tests/threads
  - *.c files implement test functions
  - *.ck files are for checking the results
  - tests.c contains the pointer to test functions
  - Make.test contains the test list

```
$ ls tests/threads/
alarm-multiple.ck       Grading           mlfqs-load-avg.ck        priority-donate-lower.c       priority-fifo.c
alarm-negative.c        Make.tests        mlfqs-nice-10.ck         priority-donate-lower.ck      priority-fifo.ck
alarm-negative.ck       mlfqs-block.c     mlfqs-nice-2.ck          priority-donate-multiple2.c   priority-preempt.c
alarm.pm                mlfqs-block.ck    mlfqs.pm                 priority-donate-multiple2.ck  priority-preempt.ck
alarm-priority.c        mlfqs-fair-20.ck  mlfqs-recent-1.c         priority-donate-multiple.c    priority-sema.c
alarm-priority.ck       mlfqs-fair-2.ck   mlfqs-recent-1.ck        priority-donate-multiple.ck   priority-sema.ck
alarm-simultaneous.c    mlfqs-fair.c      priority-change.c        priority-donate-nest.c        Rubric.alarm
alarm-simultaneous.ck   mlfqs-load-1.c    priority-change.ck       priority-donate-nest.ck       Rubric.mlfqs
alarm-single.ck         mlfqs-load-1.ck   priority-condvar.c       priority-donate-one.c         Rubric.priority
alarm-wait.c            mlfqs-load-60.c   priority-condvar.ck      priority-donate-one.ck        tests.c
alarm-zero.c            mlfqs-load-60.ck  priority-donate-chain.c  priority-donate-sema.c        tests.h
alarm-zero.ck           mlfqs-load-avg.c  priority-donate-chain.ck priority-donate-sema.ck
```

Where are the tests?

- pintos/src/tests/tests.c
  - pointer to each test func
    - If you failed on one of these,
      - Track down the test function
      - Add msg() or use GDB to debug

```
static const struct test tests[] =
  {
    {"alarm-single", test_alarm_single},
    {"alarm-multiple", test_alarm_multiple},
    {"alarm-simultaneous", test_alarm_simultaneous},
    {"alarm-priority", test_alarm_priority},
    {"alarm-zero", test_alarm_zero},
    {"alarm-negative", test_alarm_negative},
    {"priority-change", test_priority_change},
    {"priority-donate-one", test_priority_donate_one},
    {"priority-donate-multiple", test_priority_donate_multiple},
    {"priority-donate-multiple2", test_priority_donate_multiple2},
    {"priority-donate-nest", test_priority_donate_nest},
    {"priority-donate-sema", test_priority_donate_sema},
    {"priority-donate-lower", test_priority_donate_lower},
    {"priority-donate-chain", test_priority_donate_chain},
    {"priority-fifo", test_priority_fifo},
    {"priority-preempt", test_priority_preempt},
    {"priority-sema", test_priority_sema},
    {"priority-condvar", test_priority_condvar},
    {"mlfqs-load-1", test_mlfqs_load_1},
    {"mlfqs-load-60", test_mlfqs_load_60},
    {"mlfqs-load-avg", test_mlfqs_load_avg},
    {"mlfqs-recent-1", test_mlfqs_recent_1},
    {"mlfqs-fair-2", test_mlfqs_fair_2},
    {"mlfqs-fair-20", test_mlfqs_fair_20},
    {"mlfqs-nice-2", test_mlfqs_nice_2},
    {"mlfqs-nice-10", test_mlfqs_nice_10},
    {"mlfqs-block", test_mlfqs_block},
  };
```

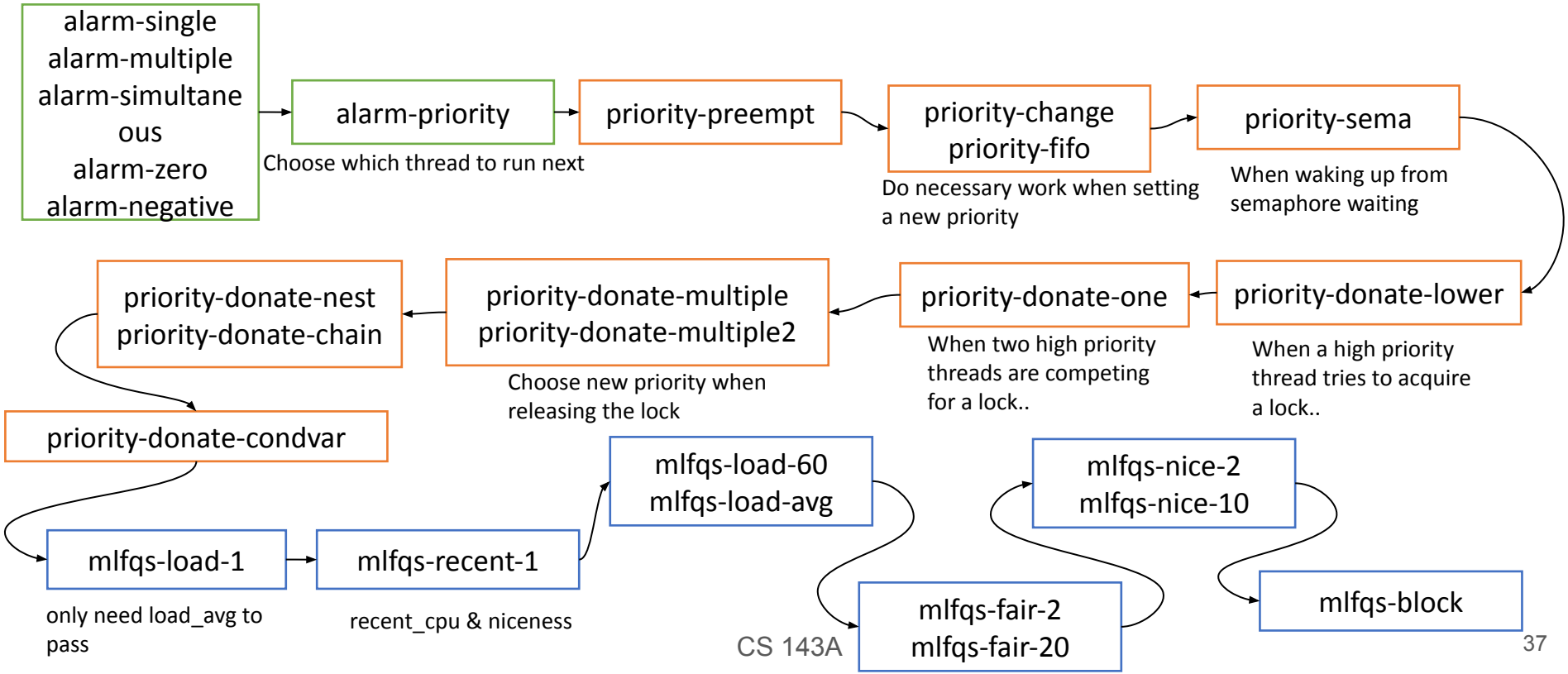
How to run the tests?

- Testing all tests
  - Go to build folder, then type make check
    - $ cd ~/Pintos/pintos/src/threads/build
    - $ make check
  - You can go to file result to see detail testing results.
- To get approximate grades
  - Go to build folder, then type make grade
    - $ cd ~/Pintos/pintos/src/threads/build
    - $ make grade
  - You can go to file grade to see the grading rubric.

What if I want to test each test cases individually?

- Individual test (without grading)
  - Make sure you run make before these individual tests
    - $ pintos -v -k -T 60 --bochs -- -q run alarm-multiple
    - $ pintos -v -k -T 60 --bochs -- -mlfqs -q run mlfqs-fair-2
- Individual test (with grading)
  - A given test t writes its output to t.output, then a script scores the output as "pass" or "fail" and writes the verdict to t.result.
  - To run and grade a single test, make the .result file explicitly from the build directory, e.g.
    - $ make tests/threads/alarm-multiple.result (we are now in build directory)
  - If make says that the test result is up-to-date, but you want to re-run it anyway
    - $ rm -f tests/threads/alarm-multiple.output
    - $ make tests/threads/alarm-multiple.result

Recommended test order



Agenda

- Project 1
- Testing
- Tips

threads/thread.c: schedule()

- Core function for scheduling
- Pick next thread to run
- switch_threads(): context switch
  - Save current registers
  - restore next thread's registers
- thread_schedule_tail()
  - initialize thread_tick
  - mark thread status as running

```
/* Schedules a new process.  At entry, interrupts must be off and
   the running process's state must have been changed from
   running to some other state.  This function finds another
   thread to run and switches to it.

   It's not safe to call printf() until thread_schedule_tail()
   has completed. */
static void
schedule (void)
{
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

Paths to schedule(): timer interrupt

- Pintos is currently set to have 100 ticks per second (100 timer interrupt)
- devices/timer_init(): register timer_interrupt()
- threads/intr-stubs.S
- threads/interrupt.c: intr_handler()
  - devices/timer.c: timer_interrupt()
    - threads/thread.c: thread_tick()
    - calc priority here is recommended (mlfqs)
    - if thread_ticks > TIME_SLICE then
      - intr_yield_on_return()
  - Waking sleeping thread here is recommended (timed wait)
  - if yield_on_return
    - thread_yield(): put current thread in the ready list
    - schedule()

```
/* Number of timer interrupts per second. */
#define TIMER_FREQ 100

void timer_init (void);
void timer_calibrate (void);
```

Paths to schedule()

- Other functions calling schedule()
  - thread_yield()
  - thread_block()
  - thread_exit()
- What about thread_unblock() ?
  - Within this function, a thread is added to the ready_list
  - If that newly added thread has a higher priority than the current one, we need to preempt the current thread immediately
  - Call thread_yield()

Tips for Alarm – priority alarm

- Make a list for sleeping threads
- Wake up threads in timer_interrupt() by comparing ticks
  - timer_interrupt() is invoked very frequently, so no heavy computation
- Use list_insert_ordered() to get an ordered list in terms of wakeup time
  - Stop searching if current tick < wakeup tick
- If thread_yield() has to be called, beware of the interrupt on/off state
  - thread_yield() disables interrupt so possible deadlock
  - In that case, use intr_yield_on_return() instead

Tips for Priority Donation

- Make a list of donors in struct thread
  - There could be multiple threads that want the same lock
  - The thread may hold multiple locks
- lock_acquire(), lock_release()
- Search for the max. priority donor and perform the donation for the current thread
- After donation, call the priority donation function for donee (recursion) for handling priority chain
- Re-calculate the priority donation chain whenever the priority changes
  - thread_set_priority()
  - mlfqs
- When priority changes, make sure to yield to highest priority thread

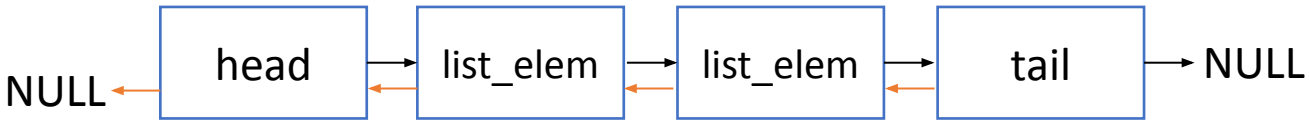
Tips for MLFQS

- Update recent_cpu for all threads in thread_ticks()
- Update the current thread’s priority in thread_ticks()
  - Refresh the priority donation chain if the current thread is involved
- Update the priority and donation chains
  - thread_ticks()
  - thread_set_priority()
  - thread_set_nice()

Pintos List Functions

- pintos/src/lib/kernel/list.h
- Doubly linked list
  - (+) No initial size (vs. array)
  - (+) Frequent insertion & deletion
  - (-) No random access (no index)

```
/* List element. */
struct list_elem
  {
    struct list_elem *prev;     /* Previous list element. */
    struct list_elem *next;     /* Next list element. */
  };

/* List. */
struct list
  {
    struct list_elem head;      /* List head. */
    struct list_elem tail;      /* List tail. */
  };
```



Pintos List Functions

- Example: semaphore
  - S – integer variable (non-negative)
  - init: S = some value
  - sema_down(wait): if (S == 0) {add_to_list; block;} else {S--;}
  - sema_up(signal): if (!wait_list_empty) {unblock} S++;

```
/* Initializes semaphore SEMA to VALUE.  A semaphore is a
   nonnegative integer along with two atomic operators for
   manipulating it:

   - down or "P": wait for the value to become positive, then
     decrement it.

   - up or "V": increment the value (and wake up one waiting
     thread, if any). */
void
sema_init (struct semaphore *sema, unsigned value)
{
  ASSERT (sema != NULL);

  sema->value = value;
  list_init (&sema->waiters);
}
```

Pintos List Functions

```
/* Down or "P" operation on a semaphore.  Waits for SEMA's value
   to become positive and then atomically decrements it.

   This function may sleep, so it must not be called within an
   interrupt handler.  This function may be called with
   interrupts disabled, but if it sleeps then the next scheduled
   thread will probably turn interrupts back on. */
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_push_back (&sema->waiters, &thread_current ()->elem);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

```
/* A counting semaphore. */
struct semaphore
  {
    unsigned value;             /* Current value. */
    struct list waiters;        /* List of waiting threads. */
  };
```

Pintos List Functions

```
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 List Functions

```
/* Up or "V" operation on a semaphore.  Increments SEMA's value
   and wakes up one thread of those waiting for SEMA, if any.

   This function may be called from an interrupt handler. */
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  sema->value++;
  intr_set_level (old_level);
}
```

Thank you. Any Questions?