CSE 421/521 - Operating Systems
Fall 2019

Recitation - II
Introduction to Pintos
Prof. Tevfik Kosar

Presented by Naveena Alango

University at Buffalo

September 16-19th, 2019

Verify Setup

- Compile
  - $ cd $PINTOSDIR/src/threads
  - $ make
- Test
  - $ cd build
  - $ pintos run alarm-multiple
- New window popped up?

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

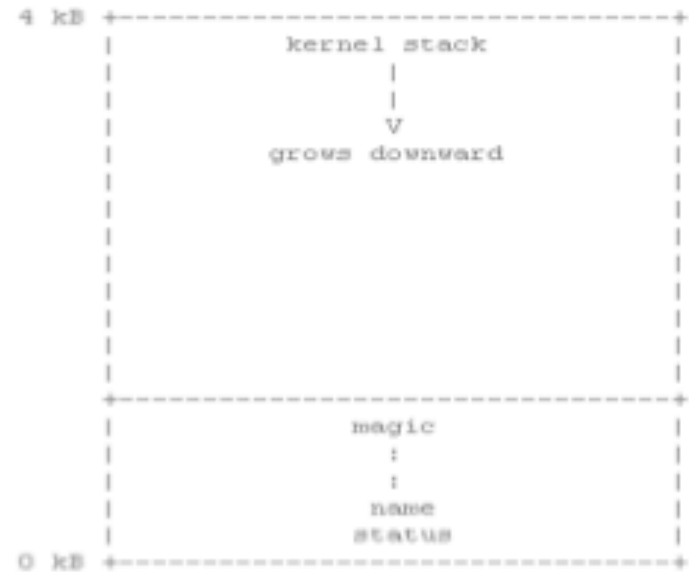
Pintos Thread System

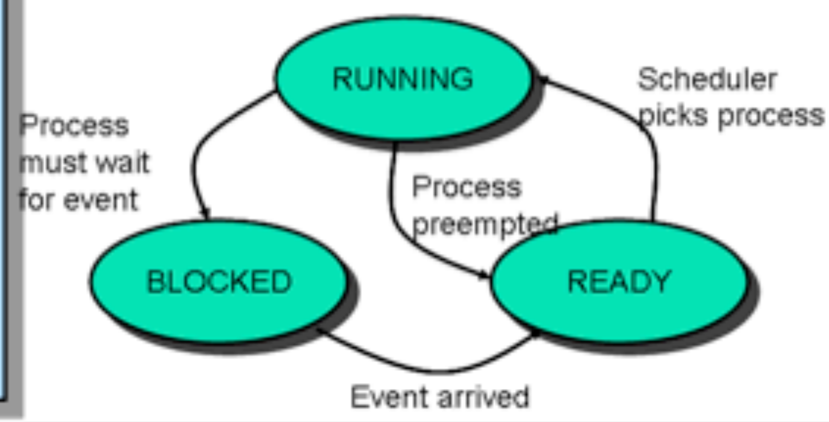
Defined in threads/thread.h:

```
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.

#ifdef 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

=> Get rid of the while loop!

=> Put the thread in to the waiting queue

=> sema_down()   // the thread will be waken up when sema_up() is called

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

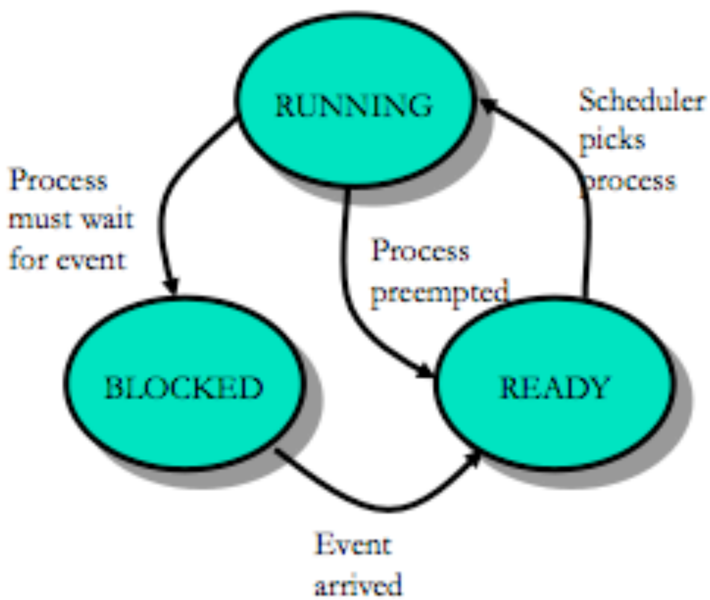
Preemptive

=> The default scheduler is very primitive, FCFS, and needs to be changed

=> Search the threads according to their priorities, and release highest…

=> You can reuse code from lib/kernel/list.c

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 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 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!**

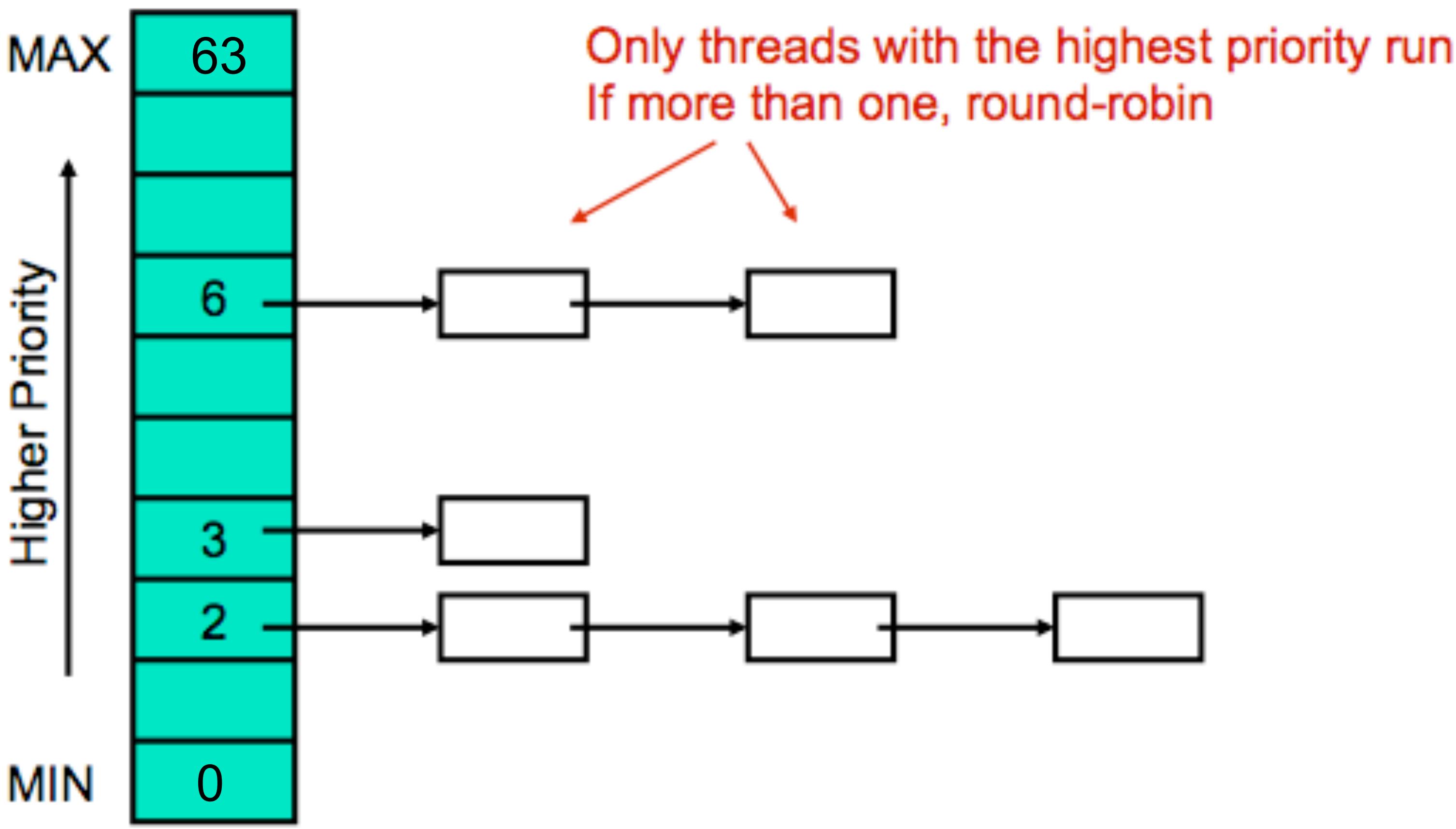
- **The only place you are allowed to turn interrupts off is,** when coordinating data shared between a kernel thread and an interrupt handler. Because interrupt handlers can't sleep, they can't acquire locks.

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 '–mlfqs' 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)

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:

  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

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

- will be implemented in **thread_get_recent_cpu(void)**
- 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`** *// or parent thread's value*
  - *at each timer interrupt,* **`recent_cpu++`** *for the running thread.*
  - *and once per second, for each thread:*

recent_cpu(t) = a * recent_cpu(t-1) + nice

where, **`a = (2*load_avg)/(2*load_avg + 1)`**

Calculating load_avg

- will be implemented in **thread_get_load_avg(void)**
- 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

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

Find a line in a file recursively using ***grep***

ex) $ grep -rin “timer_sleep”

- *ASSERT*
  - *Defined in*
  - *ASSERT(expression): Test the value of expression. If it evaluates to zero (false), the kernel panics.*

- Using ***printf***
  - You can call it from (practically) anywhere in the kernel
- Using ***GDB***

Debugging with GDB

- On first terminal:
  $ pintos --gdb -- run alarm-multiple

- In another terminal
  $ pintos-gdb kernel.o
  - (gdb) target remote localhost:1234 OR
  - (gdb) debugpintos

GDB

- **Continue**
  - (gdb) c
- **Break**
  - (gdb) break function
  - (gdb) break file:line
- **Print**
  - (gdb) p expression
- **Next**
  - (gdb) n
- **List**
  - (gdb) l

Testing

- Make grade
  - Possible to see actual pintos running command
  - ./threads/build/grade // contains summary of all tests.
- threads/build/tests/threads/
- *.error
- *.output
- *.result

Expected Lines of 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(-)
- ***‘fixed-point.h’*** is a new file added by the reference solution.