Project 1 Overview

CS162, 7/1-2/2015
Deadlines

- **Design document:** 7/2/15 at midnight
- **Design review:** 7/6/15, sign up by 7/1/15
- **Checkpoint 1:** 7/7/15
- **Project deadline:** 7/13/15
What are you building?

1. Timer —> “Alarm Clock”

2. A priority scheduler that implements priority donation for locks

3. A BSD4.4 style priority scheduler
What does Pintos already provide?

- Thread creation and completion
- Simple scheduler
- Standard synchronization data structures
  - Locks
  - Semaphores
  - Condition variables
  - Barriers
Task 1:
Sleeping Timer/Alarm Clock
Alarm Clock

• **Your task:** reimplement `timer_sleep`

• What does `timer_sleep` do?
  
  • Running `timer_sleep(n)` puts the calling thread to sleep until *at least* `n` ticks from now

  • `timer_sleep` currently busy-waits via yields. Why is this bad?
What is a timer?

• On many systems, there will be some “clock” that triggers a hardware interrupt at some known rate

• This allows us to implement periodic events

• With this timer, we can implement a variety of sleep functions, e.g.:
  
  • `timer_msleep(...)`
  
  • `timer_usleep(...), etc.`
What should your `timer_sleep(...)` do?

- A correct solution will:
  - Put the calling thread to sleep when it calls the `timer_sleep(n)` function, and leave this thread sleeping until `n` ticks from now
  - Not miss any ticks
  - Never “fail” and fall back on busy-waiting

Contemplate: Are there any corner cases that could cause your scheduler to fail? Why?
Points to consider/optimize

• If I store information about when to wake sleeping threads, what are efficient data structures and layouts for this information?

• Is it more efficient to store the time when a thread should be woken up, or the time until a thread should be woken up?

• Will my solution always wake a thread up exactly when it should wake up? Will I ever wake a thread up late?
Task 2: Priority Scheduler
Priority scheduler

• **Your task:** implement a priority scheduler

• Priorities range from 0 (low) to 63 (high)

• A correct implementation will:
  
  • Always schedule the highest priority thread that is runnable
  
  • Preempt the current thread if a higher priority thread becomes runnable
  
  • A thread can change its priority whenever it is running
  (If a thread changes its priority and is no longer the highest priority thread, what happens?)
What is priority donation, and why is it important?

• Priority donation is a partial solution to the priority inversion problem

• Thought question: if $t_1$ is waiting on $t_2$, and $t_2$ has lower priority than $t_1$, what has effectively happened?

• Is priority inversion actually a problem?

  • Yes! Priority inversion led to system instability in the Mars Pathfinder robot. :(

What we expect from your Priority Donation scheme

• Whenever a thread \( t_H \) is waiting on a lock, and another thread \( t_L \) has that lock, \( t_L \) should inherit \( t_H \)'s priority.

• Donation should occur if multiple threads are waiting on a single thread.

• Donation should also occur if there is nested waiting (e.g., \( t_H \) waits on \( t_M \) and \( t_M \) waits on \( t_L \)).

• Donations should be revoked when a thread unblocks.
Important design question: What is the correct donation behavior in each multiple donation case?
Points to consider/optimize

• Is there an upper bound on how many threads a thread can *donate* priority to? If so, what is it?

• Is there an upper bound on how many threads a thread can *receive* priority from? If so, what is it?

• Is there a limit to how deep *nested* priority donation can go? If so, what is it?
A last general comment

- Priority *scheduling* is much less complex than priority *donation*

- Your design should focus more on priority donation and the various corner cases you might encounter

- However, before implementing priority donation, make sure that your priority scheduler works!
Task 3:
Advanced Scheduler
Advanced Scheduler

- **Your task:** implement a 4.4BSD-style scheduler

- 4.4BSD uses a multilevel feedback queue scheduler:
  - This is a type of priority scheduler.
  - However, threads *do not* set their own priority.
  - The scheduler *does not* perform donation.
Note:
We must be able to choose between the basic priority scheduler and multi-level feedback queue scheduler at startup by passing the \(-mlfqs\) argument.
Multi-level Feedback Queue

- **Multi-level:** The scheduler has multiple queues that correspond to different priority levels

- **Feedback:** Over the lifetime of a thread, the scheduler will dynamically move the thread between different priority levels

- Whenever we need to schedule a thread, we will schedule a thread from the highest priority non-empty queue.

- **Important note:** We expect the individual priority-specific queues to perform round robin scheduling of threads.
How is priority calculated?

- We use the same number of priority levels as in the priority scheduler (64: 0 = low, 63 = high)
  - How many queues does our scheduler have?
- We calculate priority with the equation:
  
  \[ \text{priority} = \text{PRI}_{\text{MAX}} - \left( \frac{\text{recent_cpu}}{4} \right) - (\text{nice} \times 2) \]

- When do we calculate priority?
How is priority calculated?

• We use the same number of priority levels as in the priority scheduler (64: 0 = low, 63 = high)
  • How many queues does our scheduler have?

• We calculate priority with the equation:

  \[
  \text{priority} = \text{PRI\_MAX} - (\text{recent\_cpu} / 4) - (\text{nice} * 2)
  \]

• When do we calculate priority?
  • On thread startup, and every 4th tick thereafter
Niceness

• Although the MLFQS don’t allow threads to specify a priority, threads are allowed to set their niceness

• Niceness exists on a scale between -20 and 20 and parametrizes how likely a thread is to yield time to/take time from another thread.

  • Niceness = 0: No preference
  • Niceness = -20: Prefer to take time from other threads
  • Niceness = 20: Prefer to yield time to other threads
Calculating Recent CPU Usage

• General principle: weight calculation of CPU usage to favor recent usage

• We can do this by using an exponentially weighted moving average: \( x(t) = ax(t-1) + f(t) \)

• On every timer interrupt, we increment \( \text{recent}_\text{cpu} \) by 1 for the running thread

• Once per second, we update \( \text{recent}_\text{cpu} \) for all threads using \( \text{load}_\text{avg} \) (see project handout)
Nits: CPU Usage

- The tests assume that the “once per second” updates happen when
  `timer_ticks() % TIMER_FREQ == 0`

- Unlike priorities, `recent_cpu` can go negative

- If ordered incorrectly, the calculations performed to recalculate CPU usage can lead to overflow