Implement a Priority Scheduler

DUE: 6/5/2016 23:59
Worth: 15%

You need to implement a priority based queuing system. The code currently operates a FCFS queue with a periodic interrupt to implement a RR like scheduler (see lecture 8)

A note on marking

Various grades are indicated below, of course this relies on all work to that point being completed well and functioning. Bad design and bad commenting will also reduce the indicated grades for working code.

Getting Started

Download a new Pintos Distribution

This version contains working 'make check' code for project 2. You will find it on the course website under the projects tab.

'make check' and 'make grade'

You will now be able run 'make check', which will test your solution against a set of test programs. Make sure you 'make clean' between runs.
Do not believe the marking output from 'make grade' - I have not finalised this, but it does produce a useful indicative summary.

A Priority Ready Queue (Ind. grade C)

Thread priorities range from PRI_MIN (0) to PRI_MAX (63). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. The initial thread priority is passed as an argument to thread_create(). If there's no reason to choose another priority, use PRI_DEFAULT (31). The PRI_ macros are defined in threads/thread.h, and you should not change their values.

1. When a thread is added to the ready list that has a higher priority than the currently running thread, the current thread **must** immediately yield the processor to the new thread.
2. You **need** to implement the functions (thread_get_priority and thread_set_priority) that allow a thread to examine and modify its own priority. Skeletons for these functions are provided in threads/thread.c.
3. When you raise or lower a threads priority via thread_set_priority, you may need to reorder the ready queue, and take appropriate action if a higher priority thread is now waiting.
4. You need **not** provide any interface to allow a thread to directly modify other threads' priorities.

The test cases you will need to successfully run for this part of the project are:

1. alarm-priority
2. priority-fifo
3. priority-preempt
4. priority-change

Priority and synchronisation (Ind. grade B)

When threads are waiting for a lock, semaphore, or condition variable, the highest priority waiting thread should be awakened first.

1. You **must** implement priority waiting for locks, semaphores, and condition variables.
2. You need **not** worry greatly over the efficiency of your queue.

The test cases you will need to successfully run for this part of the project are:

1. priority-sema
2. priority-condvar

As locks use semaphores, the first test will also test locks.

Priority Donation (Ind. grade A)

One issue with priority scheduling is "priority inversion". Consider high, medium, and low priority threads H, M, and L, respectively. If H needs to wait for L (for instance, for a lock held by L), and M is on the ready list, then H will not get the CPU because the low priority thread L will not get chosen over M. A partial fix for this problem is for H to "donate" its priority to L while L is holding the lock, then recall the donation once L releases (and thus H acquires) the lock.
Implement priority donation. You will need to account for all different situations in which priority donation is required. Be sure to handle multiple donations, in which multiple priorities

are donated to a single thread. You must also handle nested donation: if H is waiting on a lock that M holds and M is waiting on a lock that L holds, then both M and L should be boosted to H's priority.

1. Pay attention to both the ready and various synch queues.
2. You may alter any data-structures to implement priority donation.
3. While there is no testcase below, you should also handle priority donation for condition variables.

Bonus:

1. There is no reason to limit the depth of nested priority donation.
2. Be extra careful when you release locks that the priorities are reset properly.

The test cases you will need to successfully run for this part of the project are:

1. priority-donate-one
2. priority-donate-lower
3. priority-donate-sema
4. priority-donate-multiple
5. priority-donate-multiple2

make sure you checkpoint (save) earlier working versions as you go, in case you break it with a later implementation

Challenge (**NOT MARKED**):

1. priority-donate-nest
2. priority-donate-chain

Please be under no illusions, this part is very challenging. You will need to analyse deeply the dependencies during thread execution. We will absolutely **NOT** provide any help with the bonus problems.

Final Report

Data Structures

Copy into your report 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

Briefly describe what happens in each algorithm you have altered or introduced. What steps are taken to minimize the amount of time spent executing your kernel code?

Synchronization

How are your various data structures protected and how are race conditions avoided in your code.

Specifically ensure you:

1. Defend the performance rationale behind your queue design.
2. Discuss any other design decisions you made.
3. Include and comment on any of the tests your soln failed (that you have attempted in your **submitted** code)

Submission

1. **WARNING** Remove all printf statements and/or msg calls you have made as otherwise the spurious output will confuse the automated test system and you will **not** get credited for passing that test.
2. submit **only and all of your changed files**, we will insert them into a std distribution and then run the test cases over your code.
3. submit your report in **PDF**.

A good idea

- After submitting your files, download them again into a fresh distribution. Then run the tests again to ensure everything is OK.
- Ultimately you are responsible to ensuring you have submitted work that executes on our system.