CS342: Operating Systems Lab

Department of Computer Science and Engineering,
Indian Institute of Technology, Guwahati
North Guwahati, Assam 781 039

Exercise T02

OS Lessons: Threads Priority Scheduling

Rating: Moderate difficult

Last modified: 20 February 2017

The task set for this exercise is derived from Section 2.2.3 *Priority Scheduling* of PintDoc. The exercise is listed earlier in PintDoc booklet but we prefer to delay it by about a month so that the relevant topics are covered in the class before the students begin this exercise.

You are also advised to check out from your version control repository, the version of the kernel code that you had immediately after the completion of Exercise T01. Do not forget to correct your Perl script `pintos` by setting correct location for `os.dsk`.

The exercise is rated moderate difficult based on the detailed instructions that we provide to you in this document. The instructions provided previously for Exercise T01 have already initiated you to this exercise by setting up `ready_list` to record ready threads in a priority based sorted order for scheduling.

In short, the exercise is built around the augmentation of code related to blocking and unblocking of the threads through synchronization primitives. The code provided to you as original the PintOS kernel, does not impose any order based on the priority of the threads.

The tasks assigned to you in this exercise are to ensure that if one of the several blocked threads can be released, the thread with the highest priority is released first. Three synchronization primitives used in the exercise are: *Semaphore*, *Lock*, and *Condition*. You must read the details in PintDoc and in the relevant comments in the code files. Files `synch.c/.h` and `thread.c/.h` in directory `src/threads` will be the main files where you will work in this exercise.

Task 1: This task is a simple extension of your code for exercise T01. Implement functions `thread_set_priority()` and `thread_get_priority()` as per the requirements set in PintDoc section *2.2.3 Priority Scheduling*. You would recall that you have added a field in `struct thread` to remember the initial (or saved) priority of the thread to enable temporary change in the threads effective priority. The scheduler uses effective priority for selecting the thread to execute.

Do not use these functions, especially `thread_set_priority()`, in your development code. These functions are used by the test scripts (`make check`). Your use of the functions may interfere with the tests and the thus your "pass" count.

Task 2 (Semaphores): Second task you must complete is to augment PintOS so that `sema_up` operation releases the highest priority thread first. The given PintOS code releases the oldest waiting

thread first. In completing this task, please do consider an important constraint that you do not run long computational tasks with interrupts in the disabled state. If this requirement is violated, the clock ticks counts may start to fall behind or become incorrect.

This is a very simple task to complete and reduces your failing test (`make check`) cases to 15 failing tests. Again, total amount of coding is no more than a dozen lines of code.

Task 3 (Conditions): It is a good idea to complete a nearly identical exercise on `Conditions`. Again you require only a few lines of code. However, you need to have a mature understanding of the data-structures available in PintOS code.

The remaining tasks are a little more challenging. First, test your kernel code. At this point, you must match the success rate as printed below in an output of `make check`:

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass 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
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass 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
14 of 27 tests failed.
```

Before you begin the next task, involving synchronization primitive called lock, it would be a good idea to realize that the existing versions of the codes for lock related functions is needed for the next exercise (T03). You may wish to create a copy of these functions within file `synch.c` for use during *2.2.4 Advanced Scheduler* based exercise (Exercise T03).

Task 4: Set up queues to support `Lock`

In implementing the priority requirements for Locks, we need two sets of lists.

1. One set of lists is associated with the locks. One list for each lock.

   This list is a list of threads in state `THREAD_BLOCKED` that are waiting to `acquire` the lock. Fortunately, this list is already available to us with the semaphore associated with the lock. Nothing to do here!

   We have already used the list in a previous task.

2. The other set of lists is associated with the threads. One list for each thread.

   This list is a list of locks that a thread has already acquired (and the thread is holding the lock). There is one such list with each thread.

For ease of programming, you must also keep a reference with each thread to mark the lock it is seeking to acquire. We will refer to this link as `seeking`.

Likewise, with each lock a reference to the thread holding it will be useful. The link may be called `holding`.

The membership of these lists change as a consequence of the events described below. You must add code to file `synch.c` (and `synch.h` and `thread.h/.c`) to correctly update the lists as necessary. An event of interest affecting a list occurs:

1. When a request is made to acquire a lock. The requesting thread is added to the list of threads seeking the lock. This occurs as a part of operation `sema_down()`.
2. When a thread successfully completes `sema_down()` operation on a lock's semaphore. On occurrence of this even the thread is no more among the threads seeking the lock. The thread has the lock, and the lock must be added to the list of locks held by the thread. Final event of interest is,
3. When a thread releases a held lock. The lock is removed from the list of the locks held by the thread and may be passed to a thread seeking to acquire the lock through operation `sema_up()`.

We have not yet completed the requirements set out in PintDoc as priority donation. However, we have all the data structures necessary for meeting this goal.

Task 5: Implement Priority Donation

As explained in PintDoc, a thread may donate its higher priority to a thread with a lower priority if the latter thread holds the lock that the former thread is seeking to acquire. The donation process is recursive in its effect. A thread receiving the benefitting priority may pass the priority onto the thread holding the lock that the benefit-receiver is seeking to acquire

On the other hand, when a priority-donor thread receives the lock it was seeking, it ends the donation of its priority to all threads who might have benefitted from its donation.

A smart student would note that it is very easy to implement this requirement by following the chain of `seeking` and `holding` links we suggested in a previous task. There is no need to limit the length of a donation chain to 8.

Each lock has a *priority* defined by the highest priority of the thread seeking (but not holding it) to acquire the lock.

Similarly, a thread has *donation benefit* (priority) defined by the highest priority of the lock it holds. When a lock holding thread releases the lock, its donation benefit may be affected.

As the priority of a thread changes, it may readjust its (sorted) order in the various lists in which it may be an element (member).

This completes our advice to the students for completing Priority Scheduling exercise.

Test Status on completion of the exercise

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass 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
7 of 27 tests failed.
```


Contributing Authors:

Vishv Malhotra, Gautam Barua, Rashmi Dutta Baruah