1 CS342: Operating Systems Lab

2 Department of Computer Science and Engineering,

3 Indian Institute of Technology, Guwahati

4 North Guwahati, Assam 781 039   Exercise T02

5

6 OS Lessons: Threads Priority Scheduling

7 Rating: Moderate difficult

8 Last modified: 18 August 2017

9

10 The task set for this exercise is derived from Section 2.2.3 *Priority Scheduling* of PintDoc. You are
11 advised to check out from your version control repository, the completed code for Exercise T01.

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

16 The exercise is designed to augment code related to blocking and unblocking of the threads through
17 synchronization primitives. The code provided to you in the original PintOS kernel, does not support
18 features related to the priority threads. In this exercise, you will add priority flavours to the
19 synchronization primitives.

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

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

31 Do not use two functions named in the paragraph above, especially
32 `thread_set_priority()`, in your development code. These functions are used by the test
33 scripts (`make check`). If you use these functions in your code you may interfere with the tests and
34 the thus the "pass" count reported by the script `make check`.

35 **Task 2 (Semaphores)**: Second task you must complete is to augment PintOS so that `sema_up`
36 operation releases the highest priority thread first. The unmodified/original PintOS code releases the
37 oldest waiting thread first. In completing this task, please do remember an important constraint that
38 you do not run long computational tasks with (external) interrupts in the disabled state. If this
39 requirement is violated, the clock ticks counts may start to fall behind or become incorrect.

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

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

45 The remaining tasks are a little more challenging. First, test your kernel code. At this point, you must
46 match or exceed the success rate printed below:

```
47  pass tests/threads/alarm-single
48  pass tests/threads/alarm-multiple
49  pass tests/threads/alarm-simultaneous
50  pass tests/threads/alarm-priority
51  pass tests/threads/alarm-zero
52  pass tests/threads/alarm-negative
53  pass tests/threads/priority-change
54  FAIL tests/threads/priority-donate-one
55  FAIL tests/threads/priority-donate-multiple
56  FAIL tests/threads/priority-donate-multiple2
57  FAIL tests/threads/priority-donate-nest
58  FAIL tests/threads/priority-donate-sema
59  FAIL tests/threads/priority-donate-lower
60  pass tests/threads/priority-fifo
61  pass tests/threads/priority-preempt
62  pass tests/threads/priority-sema
63  pass tests/threads/priority-condvar
64  FAIL tests/threads/priority-donate-chain
65  FAIL tests/threads/mlfqs-load-1
66  FAIL tests/threads/mlfqs-load-60
67  FAIL tests/threads/mlfqs-load-avg
68  FAIL tests/threads/mlfqs-recent-1
69  pass tests/threads/mlfqs-fair-2
70  pass tests/threads/mlfqs-fair-20
71  FAIL tests/threads/mlfqs-nice-2
72  FAIL tests/threads/mlfqs-nice-10
73  FAIL tests/threads/mlfqs-block
74  14 of 27 tests failed.
```

75

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

80 **Task 4: Set up queues to support `Lock`**
81 In implementing the priority requirements for Locks, we need two sets of lists.

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

83

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

87
88   We have already used the list in a previous task.
89

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

91

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

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

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

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

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

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

112 Task 5: Implement Priority Donation
113 As explained in PintDoc, a thread may donate its higher priority to a thread with a lower priority if the
114 latter thread holds the lock that the former thread is seeking to acquire. The donation process is
115 recursive in its effect. A thread receiving the benefitting priority may pass the priority onto the thread
116 holding the lock that the benefit-receiver is seeking to acquire

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

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

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

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

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

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

129 Test Status on completion of the exercise
```
130 pass tests/threads/alarm-single
131 pass tests/threads/alarm-multiple
132 pass tests/threads/alarm-simultaneous
133 pass tests/threads/alarm-priority
134 pass tests/threads/alarm-zero
135 pass tests/threads/alarm-negative
136 pass tests/threads/priority-change
137 pass tests/threads/priority-donate-one
138 pass tests/threads/priority-donate-multiple
139 pass tests/threads/priority-donate-multiple2
140 pass tests/threads/priority-donate-nest
141 pass tests/threads/priority-donate-sema
142 pass tests/threads/priority-donate-lower
143 pass tests/threads/priority-fifo
144 pass tests/threads/priority-preempt
145 pass tests/threads/priority-sema
146 pass tests/threads/priority-condvar
147 pass tests/threads/priority-donate-chain
148 FAIL tests/threads/mlfqs-load-1
149 FAIL tests/threads/mlfqs-load-60
150 FAIL tests/threads/mlfqs-load-avg
151 FAIL tests/threads/mlfqs-recent-1
152 pass tests/threads/mlfqs-fair-2
153 pass tests/threads/mlfqs-fair-20
154 FAIL tests/threads/mlfqs-nice-2
155 FAIL tests/threads/mlfqs-nice-10
156 FAIL tests/threads/mlfqs-block
157 7 of 27 tests failed.
```
158
159

160 Contributing Authors:
161 **Vishv Malhotra, Gautam Barua, Rashmi Dutta Baruah**