CS342: Operating Systems Lab

Department of Computer Science and Engineering,
Indian Institute of Technology, Guwahati
North Guwahati, Assam 781 039 Exercise T-03

Title: Advance Scheduler
OS Lessons: Multi-level Feedback Queues based Scheduling, Software Quality Attributes: Performance, Fairness, Starvation avoidance.
Rating: Easy to Moderate
Last update: 11 September 2017

This exercise is last of the three guides created for PintOS Threads project. Unlike the other exercises under PintOS, the evaluation tests included in `make check` for this exercise test the quality of the finalised kernel code on *performance* and *fairness* attributes. The correctness test (functional correctness) is rather trivial for this exercise. Admittedly, the performance checks are also relatively benign and not difficult to pass.

The *performance* requirement ensures that the periodic requirement of determining the resources used by each thread and changing the thread priorities to reflect the cpu usage is efficient. *Fairness* criteria ensures that threads with similar demand on the resources get similar access to the processor time. *Nice* related tests check that the lower priority threads do not miss the processor time completely.

The guidance provided in this document completes the exercise in three stages (Tasks).

Task 1:

In a previous exercise we have implemented function timer_sleep() by augmenting function thread_tick(). The function tracks the wakeup time of the earliest thread to be woken and unblocks one relevant thread at the scheduled time. Each woken thread is responsible for waking one more thread with the same wakeup time, if present. We were able to distribute the overheads and avoid late wakeups by running the woken threads at the highest priority immediately after their wakeups.

The dynamic nature of the thread priority computation makes this algorithm cumbersome. We suggest a different way to solve the problem. The new way will also prepare you for the next two tasks.

Create a separate *managerial* thread whose sole purpose is to unblock the threads blocked on alarms. The thread becomes active when the current time (`timer tick` count) matches the wakeup time for the (next) earliest wakeup time of a sleeping thread. The thread will unblock all threads in the waiting-for-timer-alarm queue (list `sleepers`) with the same wakeup time as the current time.

It then uses list `sleepers` to determine the wakeup time for its next action. The thread can then block itself until the time so determined for the next wakeup phase. Function `thread_tick()` will unblock this managerial thread at the right time.

Since the wakeup thread is a managerial thread and not among the threads in list `sleepers`, its action code is simple and very easy to write. Interference or likely parallel

access to the list is avoided by ensuring that the (managerial) thread is non-preemptive and has high priority. One advantage of this is that we do not have to disable interrupts while the threads waiting for timer alarms are being unblocked.

The thread is created on the pattern of `idle_thread`. Once created the thread enters an unending loop, where it is blocked to be woken up when some sleeping threads are to be unblocked from their timer wait. It will insert the released threads in `ready_list` and block itself again.

In the changed arrangement, threads call `timer_sleep()` to begin waiting for the timer alarm. All these threads are inserted in sorted list `sleepers`. However, the wakeup managerial thread calls a separate function (`timer_wakeup()`) to unblock the waiting threads. This function delivers most of the functionality assigned to function `thread_set_next_wakeup ()` in exercise PintOS-T01.

```
while (true)
  {
        intr_disable();
        thread_block();
        intr_enable ();

        timer_wakeup();
  }
```

Task 2: The thread to decay `recent_cpu` usage and re-compute priority once every second has nearly the same behaviour as the wakeup thread. You must skillfully complete the previous task before you attempt this. And, then just duplicate the solution from Task 01.

Once again, the thread will perform its actions under the assurance of non-preemption but with the interrupts enabled. Unlike, the wakeup thread, this thread is periodic and unblocked on each 100th tick.

Fortunately, no test included in the test cases prevents us from using list functions `list_sort()` and even `thread_foreach()`. You may, however, wish to not use `thread_foreach()` as the function requires interrupts to be disabled. Use a standard `for-loop` to re-compute `recent_cpu` and thread priorities with interrupts enabled for most parts. This thread, however, is non pre-emptible and runs with the highest priority.

Like idle thread, the two new managerial threads need not be included in the `ready_list`. They certainly should not be included in the computation of `load_avg`. If you include them in the count, PintOS `make check` command will report errors in your `load_avg` value.

Task 3:

Preemption requirement is implemented by limiting the cpu usage-quantum to four ticks. On 4th tick the thread priority of the running thread drops by 1. Do not re-compute `recent_cpu` and `priority` using the formula as `load_avg` value might have changed.

Take note to ensure that non pre-emptible managerial threads are not subject to the time quantum and priority reduction.

Test Status on completion of the exercise

```
pass tests/threads/mlfqs-block
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
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
```

How much code we wrote: Original Vs Threads

File(s)	Unmodified code	Threads implemented
`devices/timer.c (timer.h)`	255 (29)	328 (34)
`threads/synch.c (synch.h)`	338 (51)	571 (56)
`threads/thread.c (thread.h)`	587 (142)	854 (164)

Contributing Authors:

Vishv Malhotra, Gautam Barua, Rashmi Dutta Baruah