CS342: Operating Systems Lab.

Department of Computer Science and Engineering,
Indian Institute of Technology, Guwahati
North Guwahati, Assam 781 039 Exercise T-01

OS Lessons: Busy wait, Thread states, Timer Alarms

Rating: Moderately difficult

Last update: 20 February 2017

Primary reference: Pintos by Ben Pfaff (Referred below as PintDoc)

Our focus continues to include training with the *Developmental Tools* (Appendix F) and *Debugging Tools* (Appendix E) suggested in PintDoc. To support and set a PintOS specific goal we will add a kernel augmentation exercise from PintDoc. This exercise is described in Section 2.2.2 *Alarm Clock*. We also add the first two paragraphs of Section 2.2.3 *Priority Scheduling* in this exercise. Rest of this document adds to the description provided in PintDoc to guide the students towards a solution.

We do recognize that at this time of the semester CS342 students are yet to learn concurrent programming topics; specifically, how the activities happening independently yet simultaneously coordinate their actions inside a computer – try walking in a crowded room with everyone blindfolded. This document explains how to organize your program and active entities called *threads*. You must use the tools listed in the previous paragraph to locate functions in PintOS code: understand them and use them as needed to develop the solution code.

A typical OS kernel hosts a number of threads. A thread is an independent activity capable of executing its instructions on a computer. We will ignore all issues that concern with the creation, resource allocation and termination of a thread to focus on how a thread is scheduled to run on a computer processor.

Assume a single processor computer; only one thread can be executing at any given time. The thread is said to be in state `THREAD_RUNNING`. The other threads must wait for their turns. A threads waiting for its turn to use the processor is a ready thread. And, its state is `THREAD_READY`. The threads that are not seeking use the processor at a point in time are blocked threads in state `THREAD_BLOCKED`. These threads may be waiting for an event. In the exercise for this week, the events of interest to the waiting/blocked threads are the timer events. These are also called timer alarms or just alarms.

To ensure that all ready threads receive a fair use of the processor time, the ready threads are scheduled by a mechanism whose details we will ignore in this exercise. All that matters to us is the fact that each thread gets a small amount of the processor time (4 ticks of a clock) to run before being told to wait for the next turn.

As explained in PintDoc, time is measured inside a processor as ticks. A tick counting mechanism is included in the given PintOS code. A thread that wants to sleep and not do anything for a certain number of ticks, may pass or waste its turn till the time has progressed adequately. The current implementation wastes the time by calling `thread_yield()` – the thread lets some other unknown thread use the processor time. This is bad because the scheduler must keep asking every thread frequently and/or periodically if the thread wants to use the processor. Further, the thread being asked to use the slack time may not be interested – the thread will in turn pass the option to yet another thread. The chain of threads unwilling to use the available processor time slot may be arbitrarily long. The blind `thread_yield()` does not direct the request to make use of the free processor time to the threads willing to use the time. An obvious waste of the processor time.

A better method is to make the threads waiting for alarms blocked till the required number of ticks have been counted. The better algorithm is outlined as the changed code of function `timer_sleep()` in file `devices/timer.c`:

```
/* Sleeps for approximately TICKS timer ticks.
   Interrupts must be turned on. */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();
        int64_t wakeup_at = start + ticks;

  ASSERT (intr_get_level () == INTR_ON);

        /* Put the thread to sleep in timer sleep queue */
        thread_priority_temporarily_up ();
        thread_block_till (wakeup_at, before);

  /* original code -- to be decommissioned */
  while (timer_elapsed (start) < ticks)
    thread_yield ();  */

        /* Thread must quit sleep and also free its successor
           if that thead needs to wakeup at the same time. */
        thread_set_next_wakeup ();
        thread_priority_restore ();
}
```

Outline of the proposed algorithm

A thread planning to sleep must arrange to be woken up at a suitable time. Once the arrangement is made, the thread can block itself. The OS scheduler will not consider the blocked threads for allocation of time slot for execution on the processor.

The OS, however, must keep track of the time at which a blocked thread needs unblocking. Obviously, the thread of interest for this purpose is the one that has the earliest wakeup time among those who are blocked/sleeping for their timer alarms. The OS only needs to watch one (wakeup) time: the time for the next wakeup.

The time for the next alarm that the OS tracks can change only under two conditions (and, no other!):

- A new thread joins the set of sleeping threads and has wakeup time before the current earliest/next wakeup time. This change can be made by the thread planning to sleep before it actually blocks. Function `thread_block_till (wakeup_at, before)` implements this requirement. Or,
- A sleeping thread is woken at the end of its sleep time. The thread must look at all the remaining threads that are sleeping to find the nearest time for the next wakeup. Set the system to perform next wakeup at this new time. This is among the jobs for function `thread_set_next_wakeup ()`.

We can let function `thread_tick()` do *one* unblock of *one* sleeping thread at each wakeup time. (Significance of emphasis on words *one* will become clear a little later. A simple way to impose this restriction is by changing next-wake-time after unblocking one waiting thread. This unblocked thread when prudent will correctly set the next-wakeup-time.)

There are more details to attend but we must wait for them.

If you are wondering about two other functions `thread_priority_temporarily_up()` and `thread_priority_restore()`, the answer has many parts:

- The former function is called when a thread is about to block, so its temporary higher priority is not too restrictive to the other threads.
- Note there are several sleeping threads. So we need a list of sleeping (blocked) threads. This list is a shared list among all sleeping threads. A shared resource such as this list is only usable serially on one-thread-at-a-time basis (mutual exclusion). To keep the wall-clock time for the use-duration short we may wish to run the threads using the list at the highest priority levels.
- The thread that was blocked was good to the other threads as other threads could use the freed processor time. When a thread wakes, the benefitted

threads may be nice to it in-turn. We delay the restoration of the priority levels to the last step in the proposed algorithm.

Task 1:

It is instructive to work on these two priority-changing functions first to gain familiarity with the kernel code. Search the code using the development tools (`cscope` and/or `ctags`) to see how a `ready_list` is used to record all ready threads in the system. It will take about a dozen lines of new code and changes to implement functions `thread_priority_temporarily_up()` and `thread_priority_restore()`. You will require a new member in `struct thread` to save priority.

How to Determine If Your Changes Are Right?

When you run `make check` to test unmodified PintOS implementation under directory `pintos/src/thread` you have only 19 failed tests. On completion of the priority changing functions (we are still using the original while-loop with `thread_yield()` to wait for the specified number of ticks in function `timer_sleep()`), your success count for command `make check` should remain unchanged.

Also, note that on completion of this stage, `ready_list` is a sorted list sorted by thread priority. And, you would have written a function of type `list_less_func` to compare threads in `ready_list` for their correct position in the list.

Task 2:

Now is the time to introduce another sorted list to hold threads that are blocked on their timer alarms. For convenience, we refer to it as list `sleepers`. Remember that this list should only be accessed by one thread at a time. Only after a thread has satisfied its needs is list `sleepers` allowed to be accessed by another thread.

Let us now describe an algorithm for other two functions: `thread_block_till (wakeup_at, before)` and `thread_set_next_wakeup ()`. Function parameter `before` is a function to compare the threads in list `sleepers` and its prototype is `bool before (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)` also aliased as `list_less_func`.

As list `sleepers` may be a long list, some list functions described in `kernel/list.h` may take multiple ticks to complete. This may cause other threads to be unblocked and they too may access list `sleepers` before a previous thread has finished its use of the list. This is not safe and we do not want to let it happen.

We must control access to the list by a proper synchronization tool from `synch.h`. The tool will deny access to the list by other threads while one thread is changing the list. To keep these “busy” durations short, we have already arranged for temporarily increase of the scheduling priority of the relevant threads around these two functions.

The thread must release the synch object before it enters the block state. If access to list `sleepers` is not free when the “controlling” thread is blocked, other threads will remain unable to access list `sleepers`. Even a tiny time (just a few instructions) between the release of a synchronization control and the actual block of the thread is a potential trouble spot. You must manage it. Fortunately, the task is not too difficult. Surely, there are available solutions in the kernel code itself for you to learn the trick.

Task 3:

When a `sleeper` is woken (unblocked), the situation is trickier. On any given single timer tick, there may be several sleeping threads seeking to wake up. How do you attend to all of them within a single tick by calling `thread_unblock()` for each of them?

We suggest, you write code for this in function `thread_set_next_wakeup ()` which was primarily designed to remove thread `thread_current()` from list `sleepers` when the unblocked thread is scheduled after it is woken/unblocked from its sleep by function `thread_ticks()`.

This is not all. The function must also make sure that if there are other sleeping threads that have the same wakeup time as the thread just woken up, then they are a not left blocked. If a sleeping thread with the matching wakeup time is left waiting, then the waiting thread can only wakeup on a later timer tick. On the other hand, we cannot spend too much time waking all such threads because we do not want to hold the timer-tick in disabled state for too long. A missed tick would drift the system clock. We solved the problem by limiting to one thread unblock in function `thread_ticks()`.

The task of unblocking other simultaneous wakeups is easily shared among the multiple threads. Each unblocked thread is required to unblock one thread that has the same wakeup time as itself. This action will recursively unblock all threads which have the matching wakeup times. All except the first call to `thread_unblock()` will be from function `thread_set_next_wakeup ()`. An unblocked thread that does not unblock another thread sets the next-wake-up time. Suffice to remind that this time was set far into the future to prevent multiple unblocks from function `thread_tick()`.

Never unblock a sleeping thread if there is another unblocked thread with an earlier wakeup time.

Reminders

Be sure to check out and check-in the versions of your program from and into your version control repository. Promptly and properly annotate your program with comments.

You may also note that we have implemented basic thread priority idea as it makes this implementation simpler. This is a bit more than PintDoc 2.2.2 *Alarm Clock* target. You may wish to see PintOS-T03 guide for an alternate implementation guide.

Results of `make check` Command Execution:

```
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
FAIL 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
FAIL tests/threads/priority-fifo
FAIL tests/threads/priority-preempt
FAIL tests/threads/priority-sema
FAIL 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
19 of 27 tests failed.
make[1]: *** [check] Error 1
```

Contributing Authors:

Vishv Malhotra, Gautam Barua, Rashmi Dutta Baruah