

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: 15 August 2017

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

Our focus includes training the students about 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 for this exercise.

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.

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

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

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

```
57 /* Sleeps for approximately TICKS timer ticks.  
58    Interrupts must be turned on. */  
59 void  
60 timer_sleep (int64_t ticks)  
61 {  
62     int64_t start = timer_ticks ();  
63     int64_t wakeup_at = start + ticks;  
64  
65     ASSERT (intr_get_level () == INTR_ON);  
66  
67     /* Put the thread to sleep in timer sleep queue */  
68     thread_priority_temporarily_up ();  
69     thread_block_till (wakeup_at, before);  
70  
71     /* original code -- to be decommissioned */  
72     while (timer_elapsed (start) < ticks)  
73     thread_yield (); */  
74  
75     /* Thread must quit sleep and also free its successor  
76        if that thread needs to wakeup at the same time. */  
77     thread_set_next_wakeup ();  
78     thread_priority_restore ();  
79 }  
80
```

## 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` to the end of time horizon while unblocking the waiting thread. This unblocked thread when prudent will correctly set `next-wakeup-time` before resuming its normal task.)

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 accessible by all 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

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

#### 121 Task 1:

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

#### 129 How to Determine If Your Changes Are Right?

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

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

#### 139 Task 2:

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

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

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

155 We must control access to the list by a proper synchronization tool from `synch.h`.  
156 The tool will deny access to the list by other threads while one thread is changing the

157 list. To keep these “busy” durations short, we have already arranged for temporarily  
158 increase of the scheduling priority of the relevant threads around these two  
159 functions.

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

167 Task 3:

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

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

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

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

192 *DONOT unblock a sleeping thread if there is another blocked thread with an earlier*  
193 *wakeup time.*

194 **Reminders**  
195 Be sure to check out and check-in the versions of your program from and into your  
196 version control repository. Promptly and properly annotate your program with  
197 comments.  
  
198 You may also note that we have implemented basic thread priority idea as it makes  
199 this implementation simpler. This is a bit more than PintDoc 2.2.2 *Alarm Clock* target.  
200 You may wish to see PintOS-T03 guide for an alternate implementation guide.

201 **Results of make check Command Execution:**  
202 pass tests/threads/alarm-single  
203 pass tests/threads/alarm-multiple  
204 pass tests/threads/alarm-simultaneous  
205 pass tests/threads/alarm-priority  
206 pass tests/threads/alarm-zero  
207 pass tests/threads/alarm-negative  
208 FAIL tests/threads/priority-change  
209 FAIL tests/threads/priority-donate-one  
210 FAIL tests/threads/priority-donate-multiple  
211 FAIL tests/threads/priority-donate-multiple2  
212 FAIL tests/threads/priority-donate-nest  
213 FAIL tests/threads/priority-donate-sema  
214 FAIL tests/threads/priority-donate-lower  
215 FAIL tests/threads/priority-fifo  
216 FAIL tests/threads/priority-preempt  
217 FAIL tests/threads/priority-sema  
218 FAIL tests/threads/priority-condvar  
219 FAIL tests/threads/priority-donate-chain  
220 FAIL tests/threads/mlfqs-load-1  
221 FAIL tests/threads/mlfqs-load-60  
222 FAIL tests/threads/mlfqs-load-avg  
223 FAIL tests/threads/mlfqs-recent-1  
224 pass tests/threads/mlfqs-fair-2  
225 pass tests/threads/mlfqs-fair-20  
226 FAIL tests/threads/mlfqs-nice-2  
227 FAIL tests/threads/mlfqs-nice-10  
228 FAIL tests/threads/mlfqs-block  
229 19 of 27 tests failed.  
230 make[1]: \*\*\* [check] Error 1  
231

232

233 Understanding Threads through an Example

234 Describe yourself as a computer processor. You also have a number of subject books  
235 to read. Each reading is a thread.

236 You do not read a book in one sitting; you allocate say an hour for each subject. This  
237 is 4-ticks time quantum for a single period of reading.

238 You can read only one book at a time. Nevertheless, you are concurrently reading  
239 several books! Hope you get the meaning of word concurrent. Books whose reading  
240 is suspended is ready list of books. You will go to one of them when the book you are  
241 reading now is suspended.

242 Now consider your bookshelf where all the books you have suspended are located  
243 with bookmarks on the pages you were reading.

244 The arrangement you use is simple. You read a book for a period. On completion of  
245 the period, you insert the bookmark into the book. Determine the book to resume  
246 reading (scheduling). Take the selected book out of shelf and put the book you were  
247 reading on the shelf. This is switch function in your exercise code.

248 If your reading period is too short then overheads is excessive. Too much time is lost  
249 in replacing books and picking the books. If reading period is too long, you will not be  
250 responding to the learning each subject to match the progress in your classes.

251 Note that thread (reading) is progressing on each book. You can read only one book  
252 at a time. This is all there to a thread!

253 Sometimes your friend borrows your book. That reading is blocked. You cannot  
254 resume the reading until the book is back on your shelf.

255 The present exercise is about a case where you may decide that you do not want to  
256 read a book for a few days. Instead of picking undesired book from shelf and putting  
257 it back on shelf, make arrangements as if your friend has borrowed it!

258 Finally, when you are inserting bookmark in the book you do not want to be  
259 distracted as you may insert the marker at a wrong page. Excluding distractions is  
260 called synchronization. Activities must be synchronized to avoid "race" condition.

261

262 Contributing Authors:

263 Vishv Malhotra, Gautam Barua, Rashmi Dutta Baruah