Pintos Project #1 - FAQ

(Adapted from the Pintos Project by Ben Pfaff)

1 General FAQ

- **How much code will I need to write?**

  Here's a summary of author reference solution, produced by the diffstat program. The final row gives total lines inserted and deleted; a changed line counts as both an insertion and a deletion.

  The reference solution represents just one possible solution. Many other solutions are also possible and many of those differ greatly from the reference solution. Some excellent solutions may not modify all the files modified by the reference solution, and some may modify files not modified by the reference solution.

  ```
  devices/timer.c | 42 +++++-
  threads/fixed-point.h | 120 ++++++++++++++++++
  threads/synch.c | 88 ++++++++++++-
  threads/thread.c | 196 ++++++++++++++++++++++++++++----
  threads/thread.h | 23 +++
  5 files changed, 440 insertions(+), 29 deletions(-)
  ```

  `"fixed-point.h"` is a new file added by the reference solution.

- **How do I update the "Makefile"s when I add a new source file?**

  To add a `".c"` file, edit the top-level `"Makefile.build"`. Add the new file to variable `"dir_SRC"`, where dir is the directory where you added the file. For this project, that means you should add it to `threads_SRC` or `devices_SRC`. Then run make. If your new file doesn't get compiled, run make clean and then try again.

  When you modify the top-level `"Makefile.build"` and re-run `make`, the modified version should be automatically copied to `"threads/build/Makefile"`. The converse is not true, so any changes will be lost the next time you run make clean from the `"threads"` directory. Unless your changes are truly temporary, you should prefer to edit `"Makefile.build"`.

  A new `".h"` file does not require editing the `"Makefile"`s.

- **What does warning: no previous prototype for 'func' mean?**

  It means that you defined a non-static function without preceding it by a prototype. Because non-static functions are intended for use by other `".c"` files, for safety they should be prototyped in a header file included before their definition. To fix the problem, add a prototype in a header file that you include, or, if the function isn't actually used by other `".c"` files, make it static.

- **What is the interval between timer interrupts?** Timer interrupts occur `TIMER_FREQ` times per second. You can adjust this value by editing `"devices/timer.h"`. The default is 100 Hz.

We don't recommend changing this value, because any changes are likely to cause many of the tests to fail.

- **How long is a time slice?**

  There are `TIME_SLICE` ticks per time slice. This macro is declared in `"threads/thread.c"`. The default is 4 ticks.

  We don't recommend changing this value, because any changes are likely to cause many of the tests to fail.

- **Why do I get a test failure in pass()?**

  You are probably looking at a backtrace that looks something like this:

  ```
  0xc0108810:  debug_panic (lib/kernel/debug.c:32)
  0xc010a99f:  pass (tests/threads/tests.c:93)
  0xc010bdd3:  test_mlfqs_load_1 (...threads/mlfqs-load-1.c:33)
  0xc010a8cf:  run_test (tests/threads/tests.c:51)
  0xc0100452:  run_task (threads/init.c:283)
  0xc0100536:  run_actions (threads/init.c:333)
  0xc01000bb:  main (threads/init.c:137)
  ```

  This is just confusing output from the `backtrace` program. It does not actually mean that `pass()` called `debug_panic()`. In fact, `fail()` called `debug_panic()` (via the `PANIC()` macro). GCC knows that `debug_panic()` does not return, because it is declared `NO_RETURN`, so it doesn't include any code in `pass()` to take control when `debug_panic()` returns. This means that the return address on the stack looks like it is at the beginning of the function that happens to follow `fail()` in memory, which in this case happens to be `pass()`.

- **How do interrupts get re-enabled in the new thread following schedule()?**

  Every path into `schedule()` disables interrupts. They eventually get re-enabled by the next thread to be scheduled. Consider the possibilities: the new thread is running in `switch_thread()`, which is called by `schedule()`, which is called by one of a few possible functions:

  - `thread_exit()`, but we'll never switch back into such a thread, so it's uninteresting.

  - `thread_yield()`, which immediately restores the interrupt level upon return from `schedule()`.

  - `thread_block()`, which is called from multiple places:

    `sema_down()`, which restores the interrupt level before returning.

    `idle()`, which enables interrupts with an explicit assembly STI instruction.

    `wait()` in `"devices/intq.c"`, whose callers are responsible for re-enabling interrupts.

  There is a special case when a newly created thread runs for the first time. Such a thread calls `intr_enable()` as the first action in `kernel_thread()`, which is at the bottom of the call stack for every kernel thread but the first.

2 Alarm Clock FAQ

- **Do I need to account for timer values overflowing?**

  Don't worry about the possibility of timer values overflowing. Timer values are expressed as signed 64-bit numbers, which at 100 ticks per second should be good for almost 2,924,712,087 years.

3 Priority Scheduling FAQ

- **Doesn't priority scheduling lead to starvation?**

  Yes, strict priority scheduling can lead to starvation because a thread will not run if any higher-priority thread is runnable. The advanced scheduler introduces a mechanism for dynamically changing thread priorities.

  Strict priority scheduling is valuable in real-time systems because it offers the programmer more control over which jobs get processing time. High priorities are generally reserved for time-critical tasks. It's not "fair," but it addresses other concerns not applicable to a general-purpose operating system.

- **What thread should run after a lock has been released?**

  When a lock is released, the highest priority thread waiting for that lock should be unblocked and put on the list of ready threads. The scheduler should then run the highest priority thread on the ready list.

- **If the highest-priority thread yields, does it continue running?**

  Yes. If there is a single highest-priority thread, it continues running until it blocks or finishes, even if it calls `thread_yield()`. If multiple threads have the same highest priority, `thread_yield()` should switch among them in "round robin" order.

- **What happens to the priority of a donating thread?**

  Priority donation only changes the priority of the donee thread. The donor thread's priority is unchanged. Priority donation is not additive: if thread A (with priority 5) donates to thread B (with priority 3), then B's new priority is 5, not 8.

- **Can a thread's priority change while it is on the ready queue?**

  Yes. Consider this case: low-priority thread L holds a lock that high-priority thread H wants, so H donates its priority to L. L releases the lock and thus loses the CPU and is moved to the ready queue. Now L's old priority is restored while it is in the ready queue.

- **Can a thread's priority change while it is blocked?**

  Yes. While a thread that has acquired lock L is blocked for any reason, its priority can increase by priority donation if a higher-priority thread attempts to acquire L. This case is checked by the priority-donate-sema test.

- **Can a thread added to the ready list preempt the processor?**

  Yes. If a thread added to the ready list has higher priority than the running thread, the correct behavior is to immediately yield the processor. It is not acceptable to wait for the next timer interrupt. The highest priority thread should run as soon as it is runnable, preempting whatever thread is currently running.

- **How does `thread_set_priority()` affect a thread receiving donations?**

  It sets the thread's base priority. The thread's effective priority becomes the higher of the newly set priority or the highest donated priority. When the donations are released, the thread's priority becomes the one set through the function call. This behavior is checked by the priority-donate-lower test.

4 Advanced Scheduler FAQ

- **How does priority donation interact with the advanced scheduler?**

  It doesn't have to. We won't test priority donation and the advanced scheduler at the same time.

- **Can I use one queue instead of 64 queues?**

  Yes. In general, your implementation may differ from the description, as long as its behavior is the same.

- **Some scheduler tests fail and I don't understand why. Help!**

  If your implementation mysteriously fails some of the advanced scheduler tests, try the following:

  - Read the source files for the tests that you're failing, to make sure that you understand what's going on. Each one has a comment at the top that explains its purpose and expected results.

  - Double-check your fixed-point arithmetic routines and your use of them in the scheduler routines.

  - Consider how much work your implementation does in the timer interrupt. If the timer interrupt handler takes too long, then it will take away most of a timer tick from the thread that the timer interrupt preempted. When it returns control to that thread, it therefore won't get to do much work before the next timer interrupt arrives. That thread will therefore get blamed for a lot more CPU time than it actually got a chance to use. This raises the interrupted thread's recent CPU count, thereby lowering its priority. It can cause scheduling decisions to change. It also raises the load average.