Report for Operating Systems Assignment-III
Pintos (Scheduling)

By: 2018201035 and 2018201055

Contents

1 Introduction **1**

2 Building and Running Pintos **1**
2.1 Approach / Strategies 1
2.2 Files Modified/Added 1
2.2.1 For Set-up 1
2.2.2 For "Hello Pintos" output 2
2.3 Challenges 2

3 Pre-emption of threads **2**
3.1 Approach / Strategies (including challenges) 2
3.2 Files Modified 6

4 Implementation of Priority Scheduling **6**
4.1 Approach / Strategies (including challenges) 7
4.2 Files Modified 9

5 Implementation of Advanced scheduler (MLFQS)[Bonus] **9**

List of Figures

3-1 Basic execution flow: init.c 3
3-2 Out of Order Execution(1) 4
3-3 Out of Order Execution(2) 4
3-4 Thread unblocking failed 5
3-5 Output illustrating the running of timer (function *timer_interrupt*) 6

1 Introduction

Pintos is a simple operating system framework for the 80x86 architecture. It supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way[1].
In the assignment it was required that the following was implemented:

1. Build Pintos and set it up running in an emulator (preferably QEMU) and run a test program hello.c
2. Construct a basic pre-emptive scheduler by circumventing the busy-wait approach already used.
3. Implementation of priority scheduling
4. Implementation of Advanced scheduler (MLFQS) [**Bonus**]

The various steps taken to extend the functionality of the Pintos OS as well as other details like strategies explored and challenges faced are described in the following sections.

2 Building and Running Pintos

The first task was to set up Pintos and have it running on a simulator (preferably QEMU). The following sub-sections describe it in greater detail.

2.1 Approach / Strategies

Building and running Pintos was achieved by following the guide on Pintos which also contained a description of the assignment[2]. It largely involved modifying/appending to certain files that facilitate building of the Pintos Operating System.
Following which, to have a custom-test case run *(Display Hello Pintos whose code is contained in a file hello.c)* when Pintos boots, various sub-directories/files within the *pintos/src/* directory were examined, especially the files in *pintos/src/tests/threads/* sub-directory. It was realised that if the files that run other test-cases were modified to include the requisite custom test-case, the desired output could be achieved without getting into a more complex task of defining a file-system to enable running of user programs. A file *hello.ck* was also added defining the rules that make evaluation of this test-case (Display Hello Pintos) possible by the *make check* command.

2.2 Files Modified/Added

2.2.1 For Set-up

1. `~/pintos/src/utils/pintos-gdb`

2. `~/pintos/src/threads/Make.vars`
3. `~/pintos/src/utils/pintos`
4. `~/pintos/src/utils/Pintos.pm`

2.2.2 For "Hello Pintos" output

1. `~/pintos/src/tests/threads/Make.tests`
2. `~/pintos/src/tests/threads/hello.c`
3. `~/pintos/src/tests/threads/hello.ck`
4. `~/pintos/src/tests/threads/tests.c`
5. `~/pintos/src/tests/threads/tests.h`

2.3 Challenges

1. The link of the source code of Pintos (provided in the assignment document), turned out to be incompatible with the latest version of perl *(perl 5 version 26)* installed in most machines (compilation error was generated when trying to build the source). It was partly resolved by fixing the source code in the file */pintos/src/utils/pintos* (line 911 changing hex-value to ^V). However, it resulted in a triple fault while booting.
   The final resolution that made building and running possible was to use the latest Pintos source code (which was shared the following day by the TAs).
2. Setting up and configuring *bochs* simulator turned out to be very cumbersome. However the effort was abandoned midway since QEMU was the emulator that was recommended to run Pintos.

3 Pre-emption of threads

Since Pintos by default used busy-waiting instead of pre-emption of a sleeping thread, the task was to modify the source to add the feature of pre-emption of sleeping threads so that while a thread sleeps, some other thread could be run instead which prevents the wastage of CPU cycles. The following sub-sections describe it in greater detail.

3.1 Approach / Strategies (including challenges)

To achieve the desired functionality, two approaches were tried.
Firstly the execution flow was mapped using the functions defined in the file */pintos/src/threads/init.c*. The basic execution flow-diagram of the same is

given as

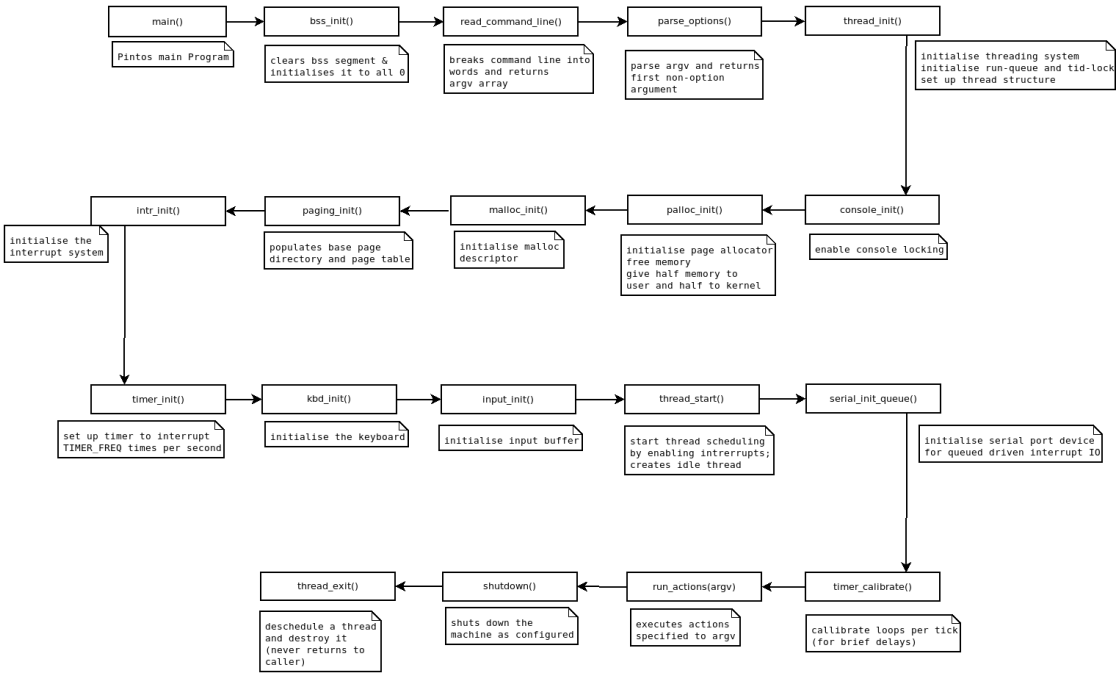

Figure 3-1: Basic execution flow: init.c

Secondly since the assignment mentioned modification of *timer_sleep* method in the file */pintos/src/devices/timer.c*, the initial modifications were made only on that function. The execution flow was studied using various *printf* statements to understand how the code worked. It turned out to be a flawed approach (based on the incorrect inference that on leaving the *timer_sleep* method, through a call to another function, the thread execution always returns back to *timer_sleep*) and the subsequent changes made to the method failed one way or another. The changes made were:

- The first approach tried was to yield the thread by placing in the ready queue and if the thread was again scheduled for execution, if the time it needs to wait had not elapsed, place it again in the ready queue.
  This lead to failure of test cases as it caused out of order execution as illustrated in the following figures. The left side window is the "success" output while the right window shows the failed output obtained (Some additional output is generated using *printf* statements to help understand the execution).

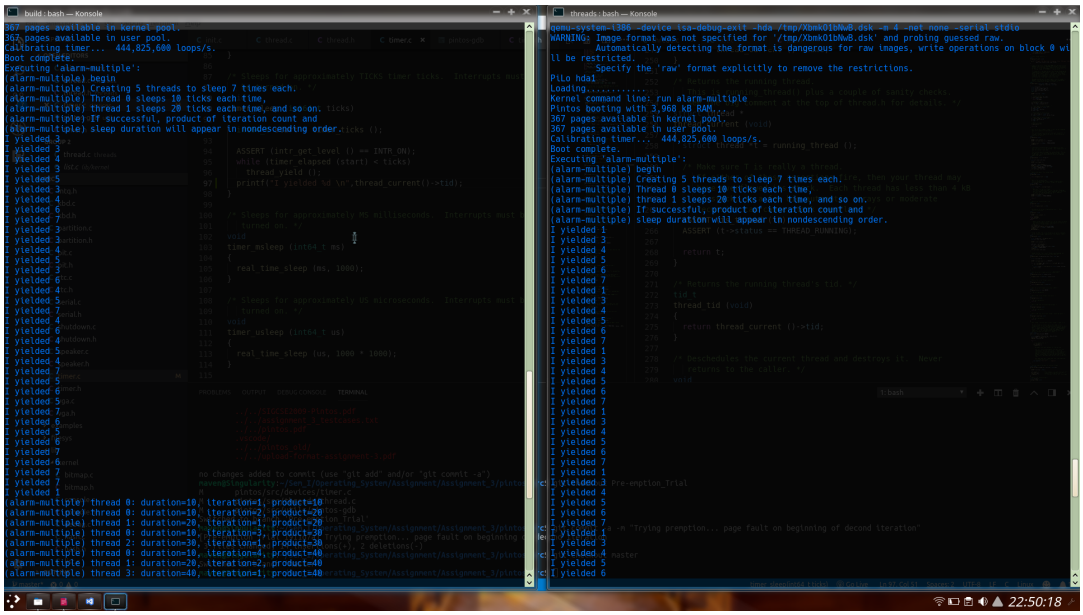

Figure 3-2: Out of Order Execution(1)

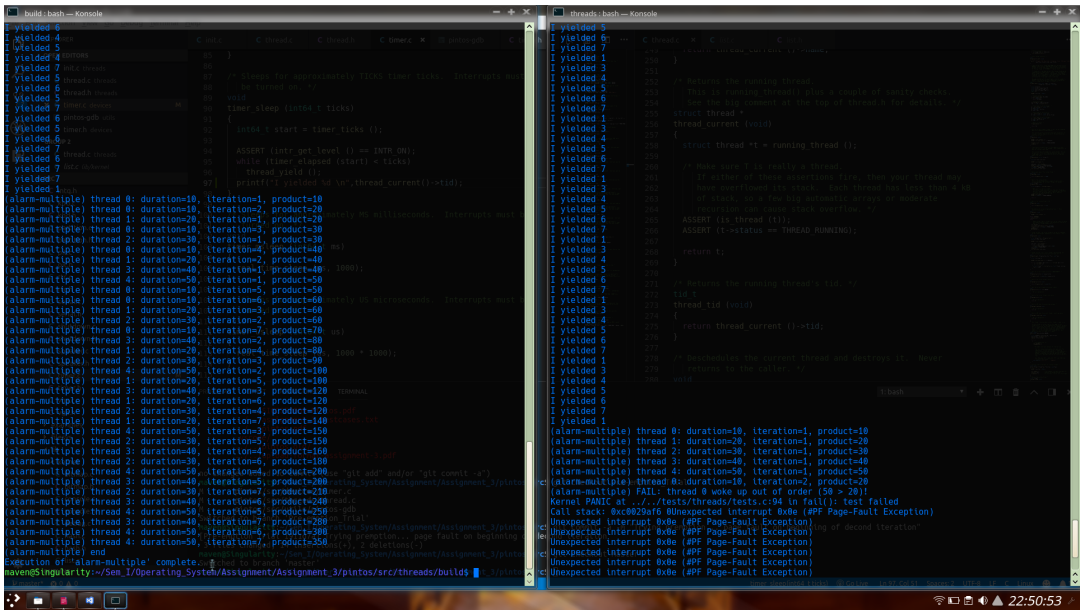

Figure 3-3: Out of Order Execution(2)

- The second approach tried was to block the thread and then placing it at the end of the ready queue and unblocking it by defining a condition in *timer_sleep* that unblocked the thread when it had slept for the required time.
  This approach failed as the *timer_sleep* function was not getting called by the bocked thread that made it impossible to unblock the thread resulting in the thread unable to execute further.This fate was met by all the threads and none of them executed further evoking the need to manually terminate the simulation. The output obtained in using this approach is shown in the figure below.

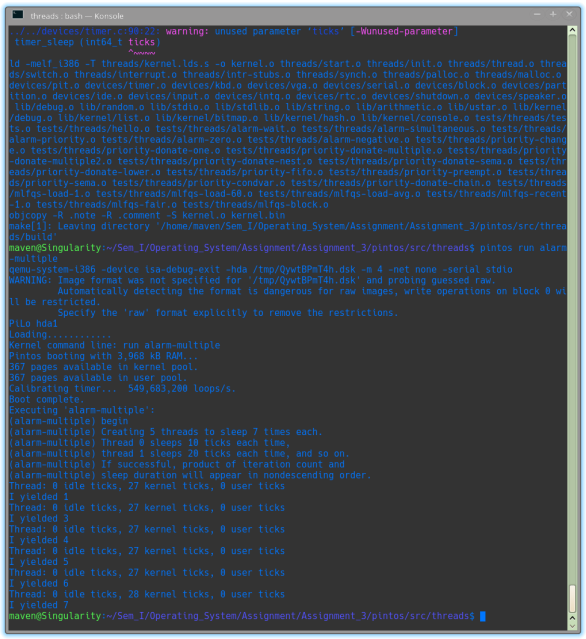

Figure 3-4: Thread unblocking failed

Thirdly it was realised (due to the aforementioned failed attempt) that the blocked should be unblocked in a function that is executed all the time. *timer_interrupt* is a function which runs at each timer tick thus it was a suitable candidate to call the method to unblock the blocked thread. The next problem was to figure out how to store information regarding the sleeping time remaining of a thread as well as maintaining a list to store all the threads that have been blocked. The solution implemented is as follows:

- The thread structure *struct thread* was modified and a field was added to keep track of how much time the thread needs to sleep
- A waiting-list was defined which stored all blocked threads

- In the *timer_sleep* function, the threads that executed this function were blocked and put in the waiting list.
- A comparator function was defined that compared the times each thread needed to sleep for
- In the *timer_interrupt* function, code to find the thread to be woken-up was added and the chosen thread was un-blocked (which put it in the ready queue for scheduling)

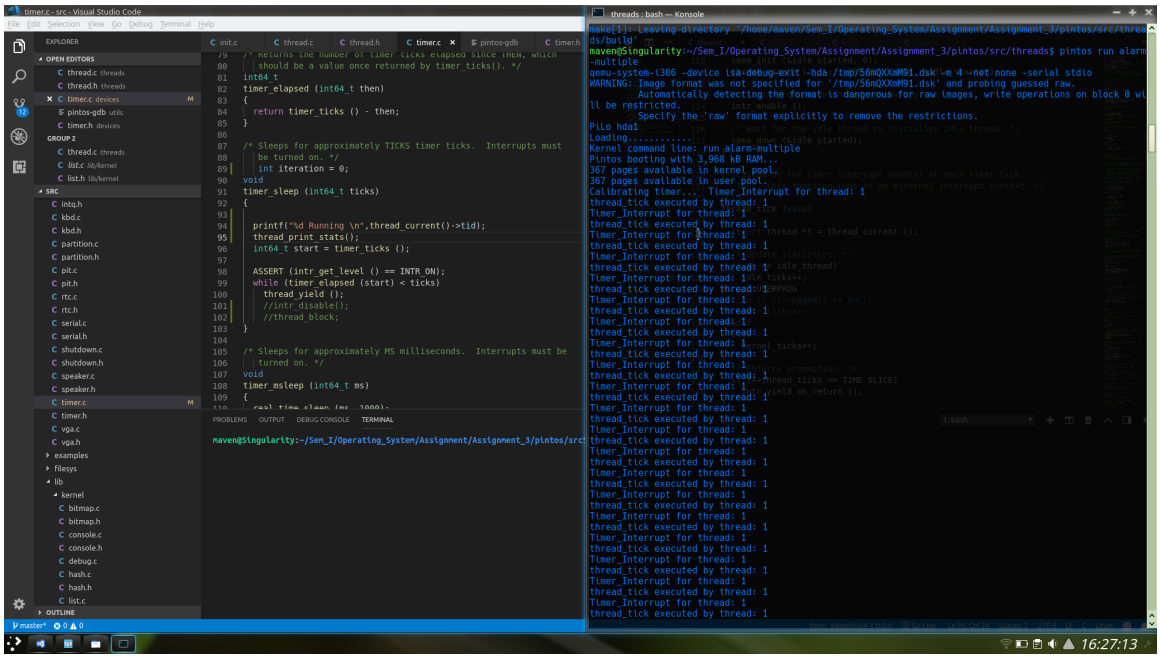

Figure 3-5: Output illustrating the running of timer (function *timer_interrupt*)

3.2 Files Modified

1. `~/pintos/src/threads/thread.h`
2. `~/pintos/src/devices/timer.c`

4 Implementation of Priority Scheduling

This part required an implementation of a basic priority scheduling scheme. It was required that if a thread was created with a higher/lower priority or if it changed priority during runtime, it must be scheduled or should yield according to its priority. The following sub-sections describe it in greater detail.

4.1 Approach / Strategies (including challenges)

Firstly since the problem statement required modification of the scheduler, an attempt was made to modify the *next_thread_to_run()* function which is called by the *schedule()* function. The *next_thread_to_run()* function chooses the next thread to run by popping the thread at the front of the ready-list. The modification made was to choose the thread of the highest priority to run next. However contrary to expectation, it resulted in Pintos unable to boot (the booting process was stuck while the operation that calibrates the timer). Later it was found the approach failed as the modifications made to *next_thread_to_run()* function, did return the correct thread to run, however it did not remove that thread from the ready queue, hence the result was that the same thread was running again and again leading to boot failure.
The second approach that was tried was to insert the threads in the ready queue according to the order of their priority (highest priority thread was inserted to the front of the queue and so on). This approach synced well with the *next_thread_to_run()* function as this function picks the next thread to schedule from the front of the queue and due to the modifications in the thread insertion (in the ready queue) method, the highest priority thread was always scheduled as the next thread to run. The solution implemented is as follows (by making changes to the *thread.c* file):

- A comparator function (*thread_compare_priority*) was defined that was used to compare the priorities of two threads (to support orderly insertions into the ready queue)
- The *thread_ticks* function was modified to allow for pre-emption of currently running thread if its priority is no longer the highest
- The *thread_create* function was modified so that if the created thread has a higher priority than the currently running thread, it will pre-empt the current running thread
- The *thread_unblock* function was modified so that the thread that unblocks is inserted into the ready-queue according to the priority of the thread
- The *thread_yield* function was modified so that the thread that yields the CPU is inserted back into the ready-queue according to the priority of the thread
- The *thread_set_priority* function was modified so that if the priority of a thread is modified at runtime, it is scheduled/pre-empted based on the new priority value

After making the aforementioned changes some more modifications were required to make the priority scheduling work with the semaphore mechanism as well as the monitor (*COND* signalling mechanism).
The following changes were made to support the additional requirement of making the priority scheduler work with semaphores:

- The *sema_down* and *sema_up* functions were examined and output statements (using *printf()*) were added to know more about their working. These functions were called by the *lock_acquire* and *lock_release* functions respectively which was in turn called by the *allocate_tid* function in the *thread.c* file

- Following this the *sema_down* function was modified by inserting the semaphore in the *sema→waiters* list in an orderly fashion. Since the semaphore priority was the same as the thread priority (found out by examining the *priority-sema.c* file), the *thread_compare_priority* function was reused from *thread.c* file. In order to reuse this comparator function, it was declared in the *thread.h* file (which *synch.c* file imported)

- The *sema_up* function was then modified. Firstly to check which thread was being woken up, its *tid* was printed. On examining this output it was found that though the threads were removed in the correct order from the *sema→waiters* list, they were not executing in correct order because at that time some other thread was running and it was not yielding control to the higher priority thread just removed from the *sema→waiters* list.

- To remedy this situation, the changes made to the *thread_tick* function in *thread.c* (pre-empting current running thread if its priority is less than a thread in the *ready_list*) were modularised into a function called *is_thread_preempt_required* and also declared in the *thread.h* file

- The *is_thread_preempt_required* function was then called in the *sema_up* function to check if the thread removed from the *sema→waiters* list has a higher priority than the current running thread. If it does, the current running thread yields the CPU

The following changes were made to support the additional requirement of making the priority scheduler work with monitor:

- Since the *cond_wait* function also inserted elements into the *cond→waiters* list, the function was modified to insert the elements in order. On examining the *priority-condvar.c* file it was found that the priority of the condition variable was the same as the priority of that thread (current thread). Thus it was decided to use the same function that compares thread priorities to help insert into the *cond → waiters* list in order. However since the elements of the list *cond→waiters* were of the *semaphore_elem* type, the comparator function did not work

- To resolve this issue, another field (*int sema_priority*) was added to the *semaphore_elem* structure to store the priority which will be used for ordered insertions to the *cond→waiters* list. Another comparator function (*waiter_compare_priority*) was written to compare the priorities of two *semaphore_elem* elements.

- Then the *cond_wait* function was modified to first assign the priority to the *semaphore_elem* element and then insert into the *cond→waiters* list in an orderly manner

4.2 Files Modified

1. `~/pintos/src/devices/thread.c`
2. `~/pintos/src/threads/thread.h`
3. `~/pintos/src/threads/synch.c`

5 Implementation of Advanced scheduler (MLFQS)[Bonus]

The Advanced Scheduler (MLFQS) was not implemented.

References

[1] Ben Pfaff et al.
https://web.stanford.edu/class/cs140/projects/pintos/pintos.html

[2] Ben Pfaff et al.
https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html#SEC2