Pintos Task 0

211 Operating Systems
Department of Computing
Imperial College

This task is logically divided into two parts:

- Questions 1-8 test your understanding of Pintos basic concepts. To answer these questions you are invited to carefully read the manual. For some questions, examining the Pintos code can be useful to provide exhaustive answers, which might be awarded with extra marks accordingly. The maximum possible mark in Questions 1-8 is 20.
- Question 9 tests your comprehension and ability of writing Pintos code. To successfully accomplish this task, you first have to download, compile, install and run Pintos, and then develop a simple functionality. The maximum possible mark in Question 9 is 20.

Question 1 - (1 mark)

Which Git command should you run to retrieve a copy of your group's shared Pintos repository in your local directory?

```
git clone git@gitlab.doc.ic.ac.uk:lab1516_spring/pintos_12
```

The above command clones the repository cleanly. Should the repository already exist, the use of `git pull --rebase` will 'pull' down remote changes and rebase them into the current working directory. We use `rebase` when pulling the code to retain our commit history, and follow conflicting changes in files as they occur, rather than just brutally merge two different versions.

Question 2 - (2 marks)

Why is using the `strcpy()` function to copy strings usually a bad idea? (*Hint: identify the problem, give details and discuss possible solutions.*)

Since you do not specify the length of the source string to copy there is a possibility of overflowing the buffer reserved for the output string.

Question 3 - (1 marks)

In Pintos, what is the default length (in ticks and in seconds) of a scheduler time slice? (*(Hint: read the task 0 documentation carefully.)*

The default length of a time slice in ticks is 4 as defined in thread.c. In seconds it is 4 * 1/100 = 0.04 seconds.

Question 4 - (6 marks)

Explain how thread scheduling in Pintos currently works in less than 300 words. Include the chain of execution of function calls. (*Hint: we expect you to at least mention which functions participate in a context switch, how they interact, how and when the thread state is modified, the role of interrupts.)*

`schedule()` picks the next thread to run by calling `next_thread_to_run()`. This function returns the head of the list of threads that are ready or, if there are not ready threads, the idle thread.

`schedule()` then switches to the chosen thread by calling the routine `switch_threads()` defined in switch.S. This function saves the caller's register state and restores the state of the thread it is switching to.

As the previous thread calls the function `switch_threads()`, the chosen thread, which called `switch_threads()` earlier on in the computation, returns from this method. It is to be noted that the two threads call and return from two separate instances of the function `switch_threads()`.

`schedule()` then calls the function `thread_schedule_tail()` that sets the current thread to `THREAD_RUNNING` and starts a new time slice. If the previous thread is not the initial thread (whose memory was not obtained via `palloc()` and if its status is `THREAD_DYING`, it gets freed.

`schedule()` is called every time threads are to be switched, that is in the functions `thread_yield()` `thread_exit()` and `thread_block()`.

Interrupts must be disabled before entering `schedule()` because they interact with the CPU allowing other threads to preempt the currently running thread. When interrupts are turned off there is the certainty of synchronisation as processes are in fact atomic.

Question 5 - (2 marks)

Explain the property of reproducibility and how the lack of reproducibility will affect debugging.

Reproducibility is the property where the same results are produced by running the same program in the same context. Given that every run of Pintos is not necessarily deterministic this means that it does not have reproducibility. This makes debugging more difficult as it is harder to work on a specific error as you cannot guarantee that it will be reproduced.

Question 6 - (2 marks)

How would you print an unsigned 64 bit `int`? (Consider that you are working with C99). Don't forget to state any inclusions needed by your code.

In order to print a 64 bit integer the library inttypes.h must be included. In this library is the macro `PRId64` which can be used with the function `printf`. For example:

```
printf("%" PRId64, i) prints the value of i where i is a 64 bit integer.
```

Question 7 - (3 marks)

Describe the data structures and functions that locks and semaphores in Pintos have in common. What extra property do locks have that semaphores do not?

`sema_init` is equivalent to`lock_init` as is `sema_down` to `lock_acquire`, `sema_up` to `lock_release` as well as `sema_try_down` and `lock_try_acquire`. The lock itself also includes a binary semaphore so they also have the list data structure in common (a list of waiters). The extra property that locks have is that once it is acquired, only the thread that acquired it can then release it.

Question 8 - (3 marks)

In Pintos, a thread is characterized by a struct and an execution stack. What are the limitations on the size of these data structures? Explain how this relates to stack overflow and how Pintos identifies it.

The maximum size for the entire thread is 4kB. However, the maximum size of just the thread struct should be well below 1kB otherwise there won't be enough room for the kernel stack. This relates to stack overflow as if the kernel stack becomes too large it overflows into the thread struct as the stack grows downwards. Pintos identifies stack overflow when the value magic in the struct is not the default value. This is because the magic value is the first value to be overwritten when there is a kernel stack overflow.

Question 9 - (1 mark)

If test `src/tests/devices/alarm-multiple` fails, where would you find its output and result logs? Provide both paths and file names. (*Hint: you might want to run this test and find out.*)

Errors are stored in `tests/devices/alarm-single.errors`.
The output is stored in `tests/devices/alarm-multiple.output`.

Question 10 - (20 marks) - The Alarm Clock

In this question, you are requested to implement a simple functionality in Pintos and to answer the questions below.

Coding the Alarm Clock in Pintos

Reimplement `timer_sleep()`, defined in '`devices/timer.c`'. Although a working implementation of `timer_sleep()` is provided, it "busy waits," that is, it spins in a loop checking the current time and calling `thread_yield()` until enough time has gone by. Reimplement it to avoid busy waiting (**10 marks**). Further instructions and hints can be found in the Pintos manual.

You also need to provide a design document which answers the following questions:

Data Structures

A1: (**2 marks**) Copy here the declaration of each new or changed '`struct`' or '`struct`' member, global or static variable, '`typedef`', or enumeration. Identify the purpose of each in 25 words or less.

```
// File: src/threads/thread.h
struct thread {
  ...
  int64_t sleep_until;               /* Earliest time to wake thread. */
  struct semaphore sema;             /* Blocks and unblocks thread. */
  ...
}
```

Changes made to `struct thread` are shown above.

We used a 64-bit integer to store the time left until the thread should be unblocked and run. We had chosen a signed integer to ensure that if a negative value was given, the thread would awaken again immediately rather than the value overflowing and the thread sleeping for a long time.

This was a vital component of ridding busy waiting. Also, we used a semaphore per thread to give a higher level construct to manipulate thread (un)blocking.

Algorithms

A2: (**2 marks**) Briefly describe what happens in a call to `timer_sleep()`, including the actions performed by the timer interrupt handler on each timer tick.

When `timer_sleep()` is called the current thread has its `sleep_until` set to the sum of the ticks specified by the caller and the current time, which specifies the earliest time the thread can be awoken. After asserting interrupts are on, `sema_down()` is called on the current thread. Since each thread's semaphore is initialised to 0, this blocks the thread until `sema_up()` is called. Furthermore, each timer tick `timer_interrupt()` will increment ticks and call `thread_tick()`, then disable interrupts and notify threads to wake up. This is done using `thread_foreach()`, which iterates through all threads, passing a pointer to the notify function, which checks each thread and if it is blocked, due to a `sema_down()` called in `timer_sleep()`, and it is past the required `sleep_until()`, the thread will be awoken by calling `sema_up()`, which will unblock the thread. (The thread will finish executing `sema_down()`, resetting the semaphore value to 0 allowing it to block again when required).

A3: (**2 marks**) What steps are taken to minimize the amount of time spent in the timer interrupt handler?

We made sure the instructions in `timer_interrupt()` were as minimal as possible: the function called only checking 2 boolean conditions, then calling `sema_up()` if required. Furthermore, the function fails fast: if the first boolean check fails the function will exit. We do check each thread every tick since we use that `all_threads` list, however we chose this design against maintaining an ordered list as we felt this was more efficient overall.

Synchronization

A4: (**1 mark**) How are race conditions avoided when multiple threads call `timer_sleep()` simultaneously?

There should not be any race conditions in `timer_sleep()`, since the only operations occuring are setting the `sleep_until` member, then calling `sema_down()` which blocks interrupts. Each thread's `struct` contains a semaphore. In `timer_sleep()`, the function `sema_down()` is called on the current thread. This then blocks the current thread.

A5: (**1 mark**) How are race conditions avoided when a timer interrupt occurs during a call to `timer_sleep()`?

Inside the `timer_interrupt()` function called by `timer_sleep()`, interrupts are disabled before accessing the `all_threads()` list thus ensuring synchronisation, preventing other threads from accessing it, then enabled afterwards. We had to disable interrupts rather than use synchronisation primitives due to being inside the interrupt handler.

Rationale

A6: (**2 marks**) Why did you choose this design? In what ways is it superior to another design you considered?

Initially we created a list of sleeping threads, initialised at the same time as the other lists of threads. We kept the list sorted by using `list_insert_ordered` and passing a pointer to a function we created which compared the `sleep_until` members of the threads being compared. Each tick, using `timer_interrupt`, the thread at the head of the list would be checked to see if it could be woken. This design led to lots of bugs with the list which we were having trouble catching, and ultimately the overheads in terms of space, in having an extra list, and time, in inserting threads into the ordered list. Therfore we tried an alternative design, where we simply iterated through the existing list of all threads using `thread_foreach` in each `timer_interrupt` and passed a pointer to a notify function which checked if a thread was blocked, and if the current time was past its `sleep_until` time, and in this case unblocked the thread using `sema_up`. This added the extra time overhead of checking every thread every tick to see if it is blocked and can be awoken, but we decided this was better than maintaining an extra list.