Imperial College London
Department of Computing
Pintos

Group 14

Task 0: Questions

Glen Rodgers
Leanne Lyons
Summer Jones
William Aboh

Question 1

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

```
git clone https://gitlab.doc.ic.ac.uk/lab1516_spring/pintos_14.git
```

Question 2

*Why is using the **strcpy()** function to copy strings usually a bad idea?*

When used carelessly this function can overflow the buffer reserved for its output string and corrupt other memory because it does not know how large its destination buffer is.

Question 3

In Pintos, what is the default length (in ticks and in seconds) of a scheduler time slice?

The default length of a scheduler time slice is four ticks (0.04s).

Question 4

Explain how thread scheduling in Pintos currently works in less than 300 words. Include the chain of execution of function calls.

The scheduler decides which thread to run next. (If no thread is ready to run at any given time, then the special idle thread, implemented in **idle()**, runs). At entry into **schedule()**, interrupts must be off and the running process's state must have been changed from **THREAD_RUNNING** to some other state, done by one of the following functions:

- **Thread_exit()**: deschedules the current thread and destroys it. All processes used by this thread are stopped and their resources freed up. Interrupts are turned off before the current thread is removed from all thread lists and set to the status **THREAD_DYING**. Schedule() is then called to decide the next thread to run.
- **Thread_block()**: sets the current thread to sleep. This function can only be called when the interrupts are turned off. The thread status is set to **THREAD_BLOCKED** and **schedule()** is called to set a new running thread.

- **Thread_yield()**: yields the CPU. The current thread is not put to sleep and may be scheduled again immediately at the scheduler's whim. To do this it disables interrupts, pushes the thread onto the ready threads list and sets its status to **THREAD_READY**, if it is not an idle thread. **Schedule()** is called to decide the next thread to run after which this thread could then run again.

The next thread to be run is then switched with the current thread so that is now running. If its status is **THREAD_DYING**, the previous current threads struct and stack are destroyed using **thread_schedule_tail()**. The new current thread has a page of memory allocated to it during the function call and a new time slice is created. All paths into **schedule()** disable interrupts. They eventually get re-enabled by the next scheduled thread. Interrupts prevent threads from being started.

Question 5

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

Reproducibility is the ability to repeat an experiment several times, producing the same result each time. Running Pintos in QEMU is non-deterministic as the timer interrupts will come at irregularly spaced intervals. This affects debugging because an interrupt which may cause a bug in one test run may not occur at the same time in another making it difficult to know when a bug has been fixed. Several test runs may provide a stronger indication of success.

Question 6

In Pintos, how would you print an unsigned 64 bit int? (Consider that you are working with C99). Dont forget to state any inclusions needed by your code.

You must include the **stdint.h** header file which defines the **uint64_t**. You also need to include the **inttypes.h** to allow you to use a predefined macro **PRIu64** to print the unsigned integer type as follows:

```
#include <stdint.h>
#include <inttypes.h>

uint64_t number;
printf("%"PRIu64, number);
```

The "%" must be included as it is not defined in the macro

Question 7

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?

The semaphore struct contains an unsigned integer which can be initialised to any value and also contains a list of waiting threads. The lock contains a semaphore struct which is set to the value 1. That is to say that when **lock_init()** is called, **sema_init()** is called with the value 1. A lock's equivalent of **sema_up()** is called **lock_release()**, and the **sema_down()** function is called **lock_acquire()**. Compared to a semaphore, a lock has one added restriction: only the thread that acquires a lock, called the lock's "owner", is allowed to release it. This owner is represented in the lock struct by a pointer to the thread.

Question 8

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.

A thread structure is allocated a 4kB page which it shared with its kernel stack. The struct is located at the bottom of the page, and the stack at the top. Unfortunately, this poses two problems:

1. If the thread struct grows too large, there will not be enough space for the stack
2. If the stack grows too large, if it 'overflows', the struct will become corrupted

Therefore, the struct should not be any bigger than 1kB and kernel functions should not allocate large structures or arrays as non-static local variables.

Every struct has a 'magic' member, used to detect stack overflow. For a running thread, **thread_current()**, magic is set to **THREAD_MAGIC**. Stack overflow will normally change this value, causing an assertion failure in **thread_current()**. This is how Pintos identifies stack overflow.

Question 9

If test src/tests/devices/alarm-multiple fails, where would you find its output and result logs?

They can be found in

```
pintos-14/src/devices/build/tests/devices
```

in the alarm-multiple.output and alarm-multiple.result files respectively, after running make check. Any errors that occurred will be logged in the alarm-multiple.errors file.