Project 1

Andrew Quinn

Due: Friday April 19th, 2024 at 11:59 PM.
Learning Objectives:
1. Get comfortable extending PintOS.
2. Practice using synchronization primitives in PintOS.

REQUIREMENTS

Your submission should include a design document, a simple shell, and a new implementation of an “alarm clock”. We elaborate on these requirements below:

DESIGN DOCUMENT

Your submission should include a design document that describes key design decisions that you made in your system. The document should be located at `docs/p01.md`. Your design document should include a separate section for each of the two other tasks (shell and alarm clock). For each section, you should outline (1) any data structures that you created or extended in your design (note: you can specify N/A if you do not create any); (2) any algorithms that you created for your design; (3) any synchronization used in your design; and (4) a justification of your design. You should aim to have enough detail in your design that a fellow 134 student would be capable of re-implementing your system by following it.

SIMPLE SHELL

This will be your first chance to extend Pintos! Currently, when PintOS finishes booting, it will check for the supplied command line arguments stored in the kernel image. You will typically pass some tests for the kernel to run, such as:

```
pintos -- run alarm-zero
```

If there is no command line argument passed, i.e., you execute

```
pintos --
```

then PintOS will simply exit. How boring.

You task is to add a shell to PintOS so that the system will run the shell interactively when there are no command line arguments. The shell has *very* simple requirements. It should emit a prompt of `CSE134>` and wait for user input. As the user types in a printable character, the shell should display the character. When a user enters a newline, the shell should parse the input.

If the input is the word `whoami`, you should print your cruzid. If input is the word `exit`, the shell should quit, allowing the kernel to exit. Finally, the shell should print `Invalid command` when provided any other input.

After parsing the input, assuming it is not equal to `exit`, the shell should print the prompt, `CSE134>` again and repeat its operation.

ALARM CLOCK

You should re-implement `timer_sleep()`, defined in `src/devices/timer.c`. Although a working implementation is provided, it "busy waits": i.e., it iteratively checks the current time and calls `thread_yield()` until enough time has gone by. Your implementation should avoid any busy waiting. Here are more details:

`void timer_sleep (int64_t ticks)`:
Suspends execution of the calling thread until time has advanced by at least `ticks` timer ticks. The thread does not need to wake up after *exactly* `ticks` timer ticks. Instead, it should be marked "ready" (i.e., no longer be blocking on any synchronization variables) after it has waited for the right amount of time.

The argument to `timer_sleep()` is expressed in timer ticks, which is not the same thing as milliseconds or other units. In PintOS, there are `TIMER_FREQ` timer ticks per second, which is a macro defined in `src/devices/timer.h`. By default, PintOS sets the value to 100. We don't recommend changing this value, because any change is likely to cause many of the tests to fail.

There are a number of additional timer functions defined in PintOS: `timer_msleep()`, `timer_usleep()`, and `timer_nsleep()`. You do not need to modify these functions as they call `timer_sleep()` internally.

RUBRIC

We will use the following rubric for this assignment:

Category	Percentage
Testing	60%
Design	40%

TESTING. We do not have any tests for the simple shell, and will manually test it. We will evaluate the shell using three (manual) tests: one checking to see that your shell handles the `whoami` command, one that tests whether your shell handles invalid commands correctly, and one that tests whether your shell handles the `exit` command. The three tests will be equally weighted and account for 50% of the testing category (for a total of 30% of the project grade).

We will use the provided tests for testing the alarm clock. The tests will account for 50% of the grade for the testing category of this assignment. You can run these tests by executing the following from the `src/threads/build` directory (after you have run `make` from the `src/trheads` directory):

```
make check
```

To see the tests as they will be weighted for the final score, execute:

```
make grade
```

Note: You will not receive any points if your code uses the provided busy wait solution. You *must* attempt to remove busy waiting from the logic to receive credit, regardless of the output from your makefile.

DESIGN. We will evaluate your design document based upon the following criteria..

1. **Sufficient:** (30%) Does your design document describe the system with sufficient detail as to be re-creatable by an engineer?
2. **Accurate:** (30%) Does your design document accurately describe the design that you implemented?
3. **Correctness:** (30%) Would your proposed design, assuming it were implemented correctly, satisfy the requirements of the assignment?
4. **Simplicity:** (10%) Is your design simple, rather than overly complex? This is obviously a bit subjective, and we will be quite lenient given that this is your first time writing one of these.

HINTS

The pintos website provides a number of tips and hints for this assignment. However, we've removed a good amount of the thread scheduling work from this assignment. So, much of what they talk about does not apply to you. However, one thing to check out is Section 2.1.2; it outlines the various files in each of the directories that you might want to interact for this assignment.

SYNCHRONIZATION

Proper synchronization is an important part of the solutions to these problems. One solution for synchronization issues in an operating system is ot turn off interrupts. With interrupts off, there is no concurrency (and, hence, no race conditions). Yay!

That said, you shouldn't solve most concurrency challenges this way. Disabling interrupts does a number of bad things to your system: you can lose timer ticks and input events. It adds to the interrupt handling latency (which an end-user would perceive). So, you should instead almost always use the synchronization primitives that we learned about in this class (i.e., semaphores, locks, and condition variables)

The only class of problem best solved by disabling interrupts is coordinating data shared between a kernel thread and an interrupt handler. Because interrupt handlers can't sleep, they can't acquire locks (I suggest that you think about why this is!). This means that data shared between kernel threads and an interrupt handler must be protected within a kernel thread by turning off interrupts. You will probably want to turn off interrupts when you handle timer interrupts; but, try to have them off for as little code as possible.

There should be no busy waiting in your submission. A tight loop that calls `thread_yield()` is one form of busy waiting.