TDDB68/TDDE47
Lab 2: Interrupts and threads

Felipe Boeira

January 2019

1 Goal

In this assignment you are supposed to get acquainted with some of the most important mechanisms of operating systems such as timers, threads, interrupts, and, last but not least, synchronization primitives. Proper functioning of any concurrent operating system is simply impossible without them.

2 Overview

This assignment covers:

- Basic interrupt management.
- A first introduction to threads switching.

This lab intends to teach the basics of synchronization, starting from the thread concurrency which is achieved via thread switching triggered by the timer interrupt. Pintos uses three synchronization primitives: semaphores, locks and conditions. All of them are already implemented. You should fully understand how they work before you start using them.

3 Preparatory questions

Before you begin doing your lab assignment you have to answer on the following set of questions to ensure that you are ready to continue.

- What is busy waiting? Why should the programmer avoid busy waiting in a concurrent operating system?
- Explain difference between Yield and Sleep.
- What is the difference between locks and semaphores? (**Hint**: there are two main differences). What is a deadlock?

4 Assignments in detail

Task Your task is to re-implement function `timer_sleep()` which is located in `devices/timer.c`. The purpose of this function is to make a calling process delay for a given time (sleep). The current implementation uses a busy-waiting strategy: it is calling `thread_yield()` and checking the current time in a loop until enough time has gone by. Obviously, this is not acceptable: processor time is a very valuable resource and must not be wasted in busy waiting.

4.1 Implementation Suggestions

In this part of the lab, you need to re-implement `timer_sleep(int64_t ticks)`. If the thread calls `timer_sleep()` then its execution is suspended for (at least) `ticks` ticks. In case there are no other running threads (that is, if the system is idle), then the thread should be awaken after exactly `ticks` ticks. You should not preempt other processes if there are any running after the time has passed by, but rather put our process into the ready-to-run queue and leave the decision when it should be executed to the scheduler.

Hint: Implement a queue for sleeping processes. You can use either original list from Pintos distribution (`lib/kernel/list.[c|h]`), or write your own list implementation.

Note, that the argument to `timer_sleep` is provided in ticks, not in milliseconds. The macro `TIMER_FREQ` defines how many ticks there are per second (defined in `devices/timer.h`).

In your implementation you may modify other functions or add your own code in `timer.c` and `timer.h` files. Note, that the functions similar to `timer_sleep()`:

- `timer_msleep()`
- `timer_usleep()`
- `timer_nsleep()`

rely on `timer_sleep()`, therefore there is no need to modify them. To test your implementation you may run `make SIMULATOR=--qemu check` from the `threads/build` directory. There are a number of tests:

- alarm-single
- alarm-multiple
- alarm-simultaneous
- alarm-zero
- alarm-negative

which will test `timer_sleep()` function in different ways. You may run (and debug if necessary) one test at a time, e.g.: `pintos --qemu -- run alarm-simultaneous` Note, that these tests will also pass at the very beginning because the current implementation of `timer_sleep()` is correct, although it is using busy waiting.
Hint: If you added code inside "threads" in lab 1, protect it with the following:

```
#ifdef USERPROG
..
#endif
```

5 Helpful Information

Code directories: `linuxpintos/src/threads, linuxpintos/src/devices`
Textbook chapters: Chapter 1.5.2: Timer
Chapter 3.2: Process Scheduling
Chapter 6.2: The Critical-Section Problem
Chapter 6.5: Semaphores
Chapter 6.6.1: The Bounded-Buffer Problem
Documentation:
Pintos documentation, and in particular:
Some parts of Project 1 Synchronization
Interrupt Handling