Project 4

CSE134 - Fall 2025

November 13, 2025

Due: Friday Dec 04th, 2025 at 11:59 PM.

Learning Objectives:

1. Get comfortable extending Pintos.

2. Practice using synchronization primitives in Pintos.

Requirements

In this assignment, you will make changes to extend the Pintos OS. Instructions on how to start with Pintos, the file structure, the debugging methods available can be found at :

https://web.stanford.edu/class/cs140/projects/pintos/pintos.html

To Do :

1. Reimplement timer_sleep(). There is a working implementation defined in devices/timer.c which "busy waits," that is, it spins in a loop checking the current time and calling thread_yield() until enough time has gone by. You need to reimplement it to avoid busy waiting.

   Function: void timer_sleep (int64_t ticks) Suspends execution of the calling thread until time has advanced by at least x timer ticks. Unless the system is otherwise idle, the thread need not wake up after exactly x ticks. Just put it on the ready queue after they have waited for the right amount of time. timer_sleep() is useful for threads that operate in real-time, e.g. for blinking the cursor once per second.

The argument to timer_sleep() is expressed in timer ticks, not in milliseconds or any another unit. There are TIMER_FREQ timer ticks per second, where TIMER_FREQ is a macro defined in devices/timer.h. The default value is 100. We don't recommend changing this value, because any change is likely to cause many of the tests to fail.

Separate functions timer_msleep(), timer_usleep(), and timer_nsleep() do exist for sleeping a specific number of milliseconds, microseconds, or nanoseconds, respectively, but these will call timer_sleep() automatically when necessary. You do not need to modify them.

If your delays seem too short or too long, reread the explanation of the -r option to pintos (see section 1.1.4 Debugging versus Testing).

Rubric

In keeping with the CSE134 Projects 1-3, we have removed the requirement to submit a Design Document. Hence, the rubric for this assignment will only include the code for the Alarm clock part:

Category	Percentage
Testing	100%

Testing We will use the provided tests for testing the alarm clock. The tests will account for 100% of the grade of this assignment. You can run the tests by following the steps as specified in the official repo. The official repo has additional requirements and tests, which are not applicable to us. Only the tests applicable to the timer and alarm clock requirement are applicable to us.

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.

Hints

The pintos website provides a number of tips and hints for this assignment.

One important 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 working with shared resources correctly and robustly. Simplest solution for synchronization issues in an operating system is to turn off interrupts. With interrupts off, there can be no timer interrupt, hence no task switching, no concurrency and, hence, no race conditions b/w tasks. Since interrupts are disabled, so no race conditions b/w tasks and interrupts either.

But it destroys real time response and also determinism of timing response both of which can only be provided using interrupts. So, it can not be the go-to way. So, we should always use the synchronization primitives (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.