UNIVERSITY OF GOTHENBURG

CHALMERS

Lab 2: Alarm Clock in Pintos

Noah Márquez
Madhav Goel

October 6, 2022

Contents

1 DESCRIPTION OF OUR SOLUTION

We have been asked to describe the implementation of our solution, and for doing so we are going to briefly analyze how we implemented each of the following sections.

1 Data Structures

In **timer.c** we added a function that keeps tracks of the sleeping threads. This function, **threads_wakeup()**, decides on whether to unblock the thread if the ticks have reached zero or keep on decreasing the ticks (number of timer ticks since OS booted).

In **thread.h** we added a variable to the *struct thread* called **sleep_timer**, which stores the time to sleep a thread in ticks. This variable was motivated because the processor does not store any information about the state of each thread and so the blocked thread itself should store the information about it.

2 Algorithms

In this section we will briefly describe how our code works, so it can be easier to understand our implementation.

We will start describing what happens in a call to **timer_sleep()**:

1. We check the ticks, to make sure that they are not ≤ 0.
2. After that and before proceeding with anything else, we have to make sure that interrupts are enabled, so we avoid any problems with consistency.
3. Before saving the current thread we disable the interrupt handler to ensure serialization.
4. Then we assign the requested sleep time (in ticks) to the current thread.
5. We block the thread and then set the old interrupt level that was used before the current thread was blocked just to avoid logic crashes.

For each clock tick, a timer interrupt is triggered and **timer_interrupt()**, the interrupt handler, is executed. We exploit this interrupt handler to activate sleeping threads and also update thread sleeping ticks (we will only comment on the things we added to the interrupt handler):

1. The timer interrupt handler will call the given **thread_foreach()** function with **threads_wakeup()** (the function we implemented) as a parameter in every tick to iterate over all the sleeping/blocked threads.
2. If the sleep_timer variable of the thread is equal to 0 (the thread is ready to wake up), then the function will wake the thread up (call to **thread_unblock()** to activate the thread or update its sleep timer). Otherwise, it keeps on decreasing the sleeping ticks of the thread.

In order to minimize the amount of time spent in the timer interrupt handler the blocked threads, which are sleeping threads, are sorted in order when kept in the sleeping list so when it is time to check the threads in order to wake up the one with the sleeping ticks equal to zero, it is easier to find the thread to unblock. Hence, reducing the time spent in the timer interrupt handler.

3 Synchronization

Proper synchronization is an important part of our solution. Our project only requires accessing a little bit of thread state from interrupt handlers. In this project, the timer interrupt needs

to wake up sleeping threads.

We are aware that when we turn off interrupts we have to do so for the least amount of code possible, or we can end up losing important things such as time ticks or input events. That is why to avoid race conditions when multiple threads call **timer_sleep()** simultaneously interrupts are disabled as soon as the first thread calls the function, so no other thread will interrupt it. Hence, there will be no chances of race conditions.

This is as simple as if interrupts are off, no other thread will preempt the running thread, because thread preemption is driven by the timer interrupt. If interrupts are on, as they normally are, then the running thread may be preempted by another at any time, whether between two C statements or even within the execution of one. That is why before doing any operation with our thread we disable the interrupts.

4 Rationale

With our solution we wanted to keep things simple and follow the basics, just to avoid unexpected twists and to write our code in a consistent style.

In the **timer_sleep()** function we followed all the instructions to ensure that we disabled interrupts, gave the thread its corresponding sleeping ticks, blocked the thread and restored the old interrupt level to avoid logical crashes.

The last thing we did is implement a function, **threads_wakeup()**, so that it gets called by **timer_interrupt()** (or better said, by **thread_foreach()**) after each tick. Why this function? It's a simple one, just following the basics, as the loop (**thread_foreach()** function) iterates through all threads which are sleeping/blocked and also are sorted out according to their sleeping ticks. This reduces the time for checking every sleeping thread, so it will be a lot easier to find the thread which needs to be unblocked rather than if the list was not sorted. This way we improve the code's execution/performance, because inside the function **threads_wakeup()** we just do simple checks and we decrease the number of the sleep timer of the thread, operations that are **O(1)** in time and space complexity.