CSE 421 Recitation

Project 1 Phase 2: Alarm Clock

Presented by Montana Lee

University at Buffalo

Agenda

1. Phase 2 objectives
2. Understanding the alarm clock
3. What is busy-waiting?
4. Implementing a non-busy waiting solution
5. Understanding Interrupts
6. Step-by-Step
7. Phase 2 files to work in and tests to be passed
8. Ways to succeed

Phase 2 Objectives

- Become much more familiar and comfortable with working with/improving upon the Pintos code.
- Understand how timer interrupts work and are used to create a functioning operating system
- Get ourselves ready for phase 3 (where 75% of the work for project 1 is)

Understanding the Alarm Clock

Pintos Alarm Clock

- The OS functions in time slices known as ‘ticks’
- Threads that need to wait are put to sleep and should wake up after however many ticks they specify they need to be woken up at
- In timer_sleep() of *src/devices/timer.c*, we wake up sleeping threads upon an interrupt after this appropriate amount of ticks
- This is currently done through thread_yield() in a loop, which is busy waiting
- However, we need to do this without busy waiting

Busy Waiting

- ‘Busy waiting’ refers to when something is technically waiting, but is actually consuming resources while doing so
- Think about it like this:
  - Thread A: I want to sleep for two seconds -> While Loop: Is it time? No? You go ahead (Yield)
  - Thread A -> Thread B -> Thread C -> Thread D -> ... -> Thread A
  - Thread A: Is it time? No? You go ahead (Yield)
  - Thread A -> Thread B -> Thread C -> Thread D -> ... -> Thread A
  - Thread A: Is it time? No? You go ahead (Yield)
  - Thread A -> Thread B -> Thread C -> Thread D -> ... -> Thread A
  - (This continues over and over)
  - Thread A -> Is it time? Yes? Exit the loop, we have waited enough.
- Obviously, this is not efficient. The OS is constantly scheduling the thread, which just checks if it can run over and over.

University at Buffalo

Waiting Efficiently

- Instead, let's think of a way to do it such that the OS does not have to schedule the sleeping/waiting thread to check. Then, you can think of it like:
  - Thread A: I want to go to sleep for two seconds, wake me up when it's time **folks** (Referring to other threads, or, more specifically, the running thread when the time comes to wake A).
  - (When a thread is sleeping, it is blocked, so it **does not run**, therefore, it cannot wake itself.)
  - Thread A -> [Blocked] -> Thread B -> Thread C -> Thread D -> Thread B -> Thread C -> Thread D -> Thread B -> Thread C -> … (Continuing over and over)
  - Timer tick -> Running thread (Thread B/C/D/...): Is it time to wake Thread A? No? Ok.
  - Timer tick -> Running thread (Thread B/C/D/...): Is it time to wake Thread A? No? Ok.
  - Timer tick -> Running thread (Thread B/C/D/...): Is it time to wake Thread A? Yes? Wake up! Thread A -> [unblocked] -> … Thread C -> Thread D -> Thread A -> Thread B -> Thread C
- Thus, while sleeping, thread A will not take CPU time. This greatly improves the efficiency of the OS, since it does not waste CPU time.

Implementing the Alarm Clock

- How do we usually signal to something when it should stop (block) and start (unblock)?
  - Semaphores!
- You will want every thread to have a semaphore for sleeping like this, that you can use to put it to sleep and wake it up
- Look at *src/threads/synch.c* for Pintos' semaphore structs and methods
- Be mindful of race conditions between threads. Disable and enable interrupts as appropriate to avoid races between a kernel thread and interrupt handling thread. They can be turned off briefly, but must be turned back on

Implementing the Alarm Clock Cont'd

- Using this semaphore, you need to make the calling thread sleep for '*ticks*' timer ticks. These ticks are passed to the timer_sleep() function
- Look at *src/threads/thread.c*, at *void thread_tick(void)*
  - This is called by the timer interrupt handler every timer tick
  - Therefore, this can be used to check how many 'ticks' have passed
  - This all runs under interrupt context, so look at *src/threads/interrupt.c*, especially at where the interrupt context and *yield_on_return()* are defined

Understanding Interrupts

- Interrupts disrupt the normal processes and demand to be taken care of immediately
- ‘Interrupt context’ is where the interrupt is processed and addressed appropriately. Exiting interrupt context requires that the stack pointer is returned to the previously executing function (which happens when interrupt processing is finished).
- In *timer.c*, *timer interrupt()* is an interrupt process that is run intermittently. This is why interrupts should be turned off some times- so that it can not be interrupted by a possible timer interrupt. Otherwise concurrency issues can happen

Step-By-Step

- Follow the Pintos code to get familiar with the interrupt process and watch Thursday lecture sessions/recordings
- Complete the design document, which explains your planned implementation
- Delete the busy-waiting mechanism currently implemented
- Program your planned implementation (it is okay if it isn't exactly like what you outlined in the design document)
- Debug with GDB and printf() - know that printing something *every* tick will slow the system down. Avoid or use conditionals to print every so many ticks instead

Files to Work In and Tests

- Much of your work this phase will take place in *src/devices/timer.c*
- The tests needed to pass to get full credit for this phase are, in addition to the phase 1 tests:

```
tests/threads/alarm-single
tests/threads/alarm-multiple
tests/threads/alarm-simultaneous
tests/threads/alarm-priority

tests/threads/alarm-zero
tests/threads/alarm-negative
```

*Note that alarm-priority will only pass if your priority scheduler is correctly implemented from phase 1. Phase 1 tests will also be counted in phase 2 grading.

- This phase is *small* compared to phase 3. Do not use this to gauge the difficulty/length of the rest of the project.

University at Buffalo

Files to Work In and Tests Cont'd

- After the deadline, TAs will check your solution and design document.
- If your solution still busy-waits, you will lose a significant amount of points!
- Your design document is split in two and is worth 10% of your final PA-1 grade (5% per phase submitted). It is acceptable if you do not follow the *exact* design for your final implementation.
- You must submit a design document for this phase (as well as phase 3)
- Phase 3 is not entirely dependent on phase 2, so if you cannot complete or pass all tests for phase 2, do not panic. Past phases can continue to be implemented for part of the grade after deadline.

Ways to Succeed in the Project and Course

- Don’t be afraid of the code! Explore everything and read through important files
- Read the documentation!
- Visit office hours and ask questions *early*
- Start *early*. The projects are very time-consuming and require steady, metered work
- Commit often to Github in case you need to revert back
- Communicate with your team member
- Get in contact with staff ASAP if problems arise
- *Don’t cheat*!

Good luck!