Design Optimization of Computing Systems

Autumn Semester 2024

Assignment 2: Introduction to PintOS and improving threads in PintOS
Assignment given on: 26/09/2024

Assignment deadline: 16/10/2024

PintOS (Ben Pfaff et al.) [http://pintos-os.org/SIGCSE2009-Pintos.pdf] is a popular minimal operating used at many Universities – in India and overseas – to train students in the internal details of the Unix-like operating systems. PintOS is minimal since many OS capabilities are missing and you will be asked to implement them by modifying the codebase.

The primary document that we use in this subject was originally written at Stanford University [https://web.stanford.edu/class/cs140/projects/pintos/pintos.pdf]. The more detailed version of PintOS and the projects used in Stanford is here: https://web.stanford.edu/class/cs140/projects/pintos/pintos.html (This should suffice to give you a basic overview of PintOS).

This warming-up assignment has two parts: In the first one we ask you to install pintOS on a virtual machine with all the dependencies. In the second one we ask you to improve the thread implementation supported by PintOS.

Part-1: Installation of PintOS

We can install PintOS in many operating systems (like Windows, linux, mac) with slightly different set-up instructions for each platform. We are giving the instructions for a Linux based systems

- Please follow the steps described in this github link.

Part 2: Implement an Alarm clock (80 marks)

Problem description:

By default, PintOS kernel provides busy waiting when a thread needs to block. You need to change the behavior and block the thread till a fixed amount of timer ticks pass. You will work primarily in the “threads” directory of the PintOS distribution.

Before you start coding, go through this link: https://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html#SEC18 . it will provide you an understanding of the gist and functionalities of each relevant file.

Task:

Reimplement timer_sleep(), defined in devices/timer.c. Currently the implementation "busy waits" that is it spins in a loop checking the current time and calling thread_yield() until enough time has gone by. Here is the description of the function.

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.

Reimplement the function to avoid busy waiting.

Part 3: Implementing a wakeup thread (40 marks)

This assignment is an extension of previous assignment where you improved the thread implementation supported by PintOS. Specifically, in the previous assignment you improved the timer_sleep(), defined in devices/timer.c and avoided busy waiting. Now a kernel scheduler needs to ensure *fairness* and *performance*.

The performance requirement ensures that the periodic requirement of determining the resources used by each thread and changing the thread priorities to reflect the cpu usage is efficient. Fairness criteria ensures that threads with similar demand on the resources get similar access to the processor time. So, your lower priority threads should not miss the processor time completely.

To that end, while removing the busy wait some of you might have already created a separate wakeup thread (Of course you might have completed the assignment without the wakeup thread too). However, we are providing a sketch using which you can create the wakeup thread below. This assignment depends on mechanisms similar to the wakeup thread.

The sole purpose of the wakeup thread is to unblock the threads blocked on alarms. The thread becomes active when the current time (timer tick count) matches the wakeup time for the (next) earliest wakeup time of a sleeping thread. The thread will unblock all threads in the waiting-for-timer-alarm queue (let's call it sleepers) with the same wakeup time as the current time.

Wakeup thread then uses list sleepers to determine the wakeup time for its next action. The thread can then block itself until the time is determined for the next wakeup phase. Function thread_tick() will unblock this wakeup thread at the right time.

Since the wakeup thread is a managerial thread and not among the threads in list sleepers, its action code is simple and very easy to write. Interference or likely parallel access to the list of waiting threads is avoided by ensuring that the (wakeup) thread is non-preemptive and has high priority. One advantage of this is that we do not have to disable interrupts while the threads waiting for timer alarms are being unblocked. Once created the thread enters an infinite loop, where it is blocked to be woken up when some sleeping threads are to be unblocked from their timer wait. It will insert the released threads in ready_list and block itself again.

In this arrangement, threads call timer_sleep() to begin waiting for the timer alarm. All these threads are inserted in sorted list sleepers. However, the wakeup managerial thread calls a separate function (timer_wakeup()) to unblock the waiting threads.

Your task is to implement the wakeup thread-based modification of the previous assignment, where the wakeup just unblocks a sleeping thread when it's time to wake it up.

Submission Guideline:

You need to upload a zip containing the files you changed along with a design document in Moodle. There should be one submission from each group. Name your zip file as "**Assgn2_<rollno>.zip**". The zip file should contain:

- The files that you changed.
- A design document. You can find the template for design document here: https://web.stanford.edu/class/cs140/projects/pintos/threads.tmpl. Fill it up according to your implementation and include it in your zip file.

Grading scheme for part 2:

The total marks for this part of the assignment (80 marks) is divided as follows.

Demo the implementation to your assigned TAs and show that your code works as intended	**30 marks**
During demo you need to demonstrate that you have indeed implemented the algorithm promised in the design document	**50 marks**

Grading scheme for part 3:

The total marks for this part of the assignment (40 marks) is divided as follows.

Demo the implementation to your assigned TAs and show that your code works as intended	**10 marks**
During demo you need to demonstrate that you have indeed implemented the algorithm promised in the design document	**30 marks**