Operating Systems Lab
Dept of CSE, LNMIIT, Jaipur
22 March 2021
Lab Session 08

Introduction

We have already started working with PintOS code in a previous lab session. In this session, we will do our first meaningful alteration to the PintOS kernel.

As you begin work, use Git to record your progress. You may want to reverse your code to one of the previously committed statuses when you encounter difficulties.

You must also familiarise yourself with library `src/lib/kernel/list`. Get working knowledge and understanding of C macros. To begin this learning, use macros available in this list library.

A task from a previous lab is listed here to refresh your memory.

Task 00: Revision

(Sourced from University of Chicago: https://uchicago-cs.github.io/mpcs52030/p0.html) I copy the relevant part of the description from their website:

> The goal of Project 0 is to ensure you have gotten to the point where you are comfortable compiling the Pintos code, running it, as well as running the tests (including individual tests). It will also require that you start familiarizing yourself with the data structures in the Pintos kernel.
>
> To do this, you are only required to implement the Alarm Clock part of Project 1. However, the Alarm Clock requires using a semaphore, but we will not have covered semaphores in depth before Project 0 is due. So, we will tell you the exact semaphore code you need to include:
>
> - You will need to add the following field to `struct thread` in `thread.h`:
>   `struct semaphore timer_sema;`
> - You will need to include the following call in `init_thread` in `thread.c`:
>   `sema_init(&t->timer_sema, 0);`
> - Suppose variable `cur` contains a pointer to the currently running thread (a pointer to `struct thread`, as returned by `thread_current()`). To put that thread to sleep, do the following:
>   `sema_down (&cur->timer_sema);`
> - Suppose variable `th` contains a pointer to a `struct thread` for a thread that was previously put to sleep as described above. To wake up that thread, do the following:
>   `sema_up (&th->timer_sema);`
>
> With the above, you should be able to implement Project 0 just using the Pintos documentation and using what you will learn in the Processes and Threads part of the first lecture.
>
> You will also have to avoid certain race conditions, but this will not require using semaphores, locks, etc. As stated in the Pintos documentation, "the only class of problem best solved by disabling interrupts is coordinating data shared between a kernel thread and an interrupt handler" which is precisely what happens when implementing the Alarm Clock. This, in turn, will require that you understand interrupts and interrupt handlers.

Task 01: On the Mark

Given below is the new code for function `timer_sleep()` in file `devices/timer.c` that you need to block threads for the required number of ticks.

```
void timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);

  if (ticks <= 0)
      return;

  thread_current()->sleep_start = start;
  thread_current()->sleep_ticks = ticks;
  alarm_set();
}
```

You are advised to create two new files `alarm.h` and `alarm.c`, to operate the timer alarms. The functions supporting the thread blocking will be written in files `alarm.h` and `alarm.c`. The alarms will manage thread wait by blocking the threads instead of wasting processor cycles in the busy wait loops.

File `alarm.h` (as coded in your tutor's implementation) is listed here:

```
#ifndef THREADS_ALARMR_H
#define THREADS_ALARMR_H

void alarm_init (void);
void alarm_set (void);
void alarm_tick (void);
void alarm_go(void);

#endif /* threads/alarm.h */
```

Two functions in file `alarm.c` are almost already done for you and listed below! Well, Not fully. But, nearly almost fully. Only some minor addition may be needed to the code given here.

```
void alarm_tick(void)
{
   next_alarm--;
}

void alarm_go(void)
{
    while (next_alarm <= 0)
      alarm_ring();
}
```

Variable `next_alarm` counts the number of ticks to the earliest alarm that needs ringing. As you notice, the count is decreased on each timer tick. And when this count has a value of 0 (or below zero), the alarms are rung by function `alarm_ring()`.

It is your job to write functions listed in file `alarm.h` above. And, to include calls to the functions in files `init.c` and `thread.c` as appropriate. Your solution to Lab06 should provide useful guidance here. However, be sure to not use idle thread for any needed task.

We will describe the functions in the later sections of this document for you to complete.

At this stage, update `thread_SRC` list in file `src/Makefile.build` to include `alarm.c` as a program file in threads collection. Read PintOS document if you need help.

Task 02: `alarm.c`: List `sleepers` and its Semaphore

In our implementation, we will need to track threads that are waiting for their alarms. An ordered (sorted) list called `sleepers` may be defined for this purpose. There are several challenges to overcome as you make this list.

1. List `sleepers` is a list shared among multiple concurrent threads. Therefore, we need a semaphore to coordinate threads' access to the list (`sleepers`).
2. The list will hold threads waiting for alarms. The threads are represented by the data-structure `struct thread`. The list library uses a field of type `struct list_elem` to organise lists. Add necessary new members in `struct thread` (in file `src/thread.h`) to connect threads in list `sleepers`.
3. There are a few members mentioned in file `timer.c` for you to add to `struct thread`.
4. Be sure to include necessary initialisations for various variables and fields in your code. For example, where will you declare `next_alarm`? What will be its initial value?
5. At this stage, you have already created several semaphores. There is one semaphore to coordinate access to list `sleepers`. In addition, for each thread there is a separate semaphore for the thread to block. Each semaphore needs to be properly initialised.
6. The sleeping threads will be blocked on their private `timer_sema`. Each thread has its private `timer_sema` semaphore located in data-structure `struct thread`.

Lest we forget, the list `sleepers` is a sorted list. Let us look at this issue in some detail next.

Task 03: Define Function `before()`

Each sleeping thread sets the time for its alarm. The next alarm to ring is the one that is the earliest among these set alarm times. To determine the time for the earliest alarm, list `sleepers` must be ordered by time. This is done by defining a comparison function `before()`. This function is used when calling function `list_insert_ordered()`. Function `before` is a parameter in the call.

Your learning does not end here! As you write function `before()`, you need to learn to use C macro `list_entry()`. This macro helps the programmers to get an enclosing `struct thread` from a `struct list_elem` embedded in it.

The good news is that there are many code examples in various `.c` files within PintOS code. Look in file `synch.c`.

The following functions will need codes to set up list `sleepers`.

Task 04: Define function `alarm_set()`

Function `alarm_set()` adds a thread seeking to set a timer alarm in list sleepers. Once a thread has been included in the list, and you have released the semaphore controlling the list, you will block the thread by invoking operation `sema_down (&cur->timer_sema);`

A blocked thread stays blocked until some other thread unblocks it. Function `alarm_go()` calls `alarm_ring()` to unblock thread at the front of list sleepers.

Task 05: Define function `alarm_ring()`

This function does the task of releasing a blocked thread by invoking operation `sema_up (&th->timer_sema);`

Note that the thread releasing a blocked thread is different from the thread that is being released. This situation is different from the setting we had previously when we were setting the alarms. In that case, the thread was setting the alarm for itself.

After unblocking a thread, the function also adjusts `next_alarm` to a new value. The new value is determined by the thread at the front of list `sleepers`.

Task 06: Where to embed call `alarm_go()`?

You need careful exploration of PintOS code to determine the appropriate location to call function `alarm_go()`. Is function `schedule()` a good opening for this call? There may be another more appropriate function for the call.

In a previous lab session exercise, we passed this burden to thread idle. This was an inappropriate obligation on the thread.

Another challenge that needs careful consideration is the need to ring the alarm as soon as the set time is reached. The alarm ring should not be delayed. Carefully consider the use of function `intr_yield_on_return()`. Think carefully and determine if your implementation needs to invoke this function.

Task 07 (Not for assessment): Experiments with Stack space and Delays in setting an alarm

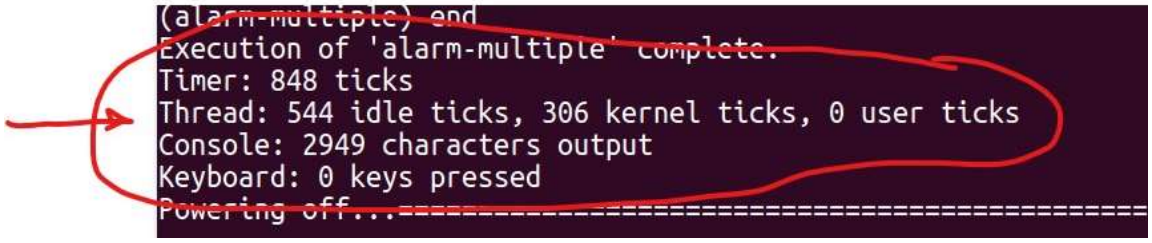
A thread running in the kernel works under a severe stack size constraint. You may test these limitations by calling the recursive function `sumSeries()` with parameter values in the range 200 to 300. Be ready to also try other values.

A convenient place to invoke function `sumSeries()` is in function `alarm_set()`. Take care to not place the call in the critical section.

```
15
16 static long sumSeries(long i);
17
18 static long sumSeries (long i) {
19    if (i <= 0)
20       return 0;
21    return i+sumSeries(i-1);
22 }
23
```

A good call is `sumSeries(220)` – the larger values of the parameter disturb `struct thread` field `magic`. Try and see what havoc inappropriate parameter values wreck.

The execution times (in ticks) for different execution modes are printed and can be read near the end of the output for command: `pintos run` *<program-name>*.




```
static int64_t work_a_tick (int64_t when)
{
     int64_t loops;
     int64_t start;

  if (when < 5) return 0;
  start = timer_ticks();
  while (timer_elapsed (start) < 1) {
       loops = 1000;
       while (loops-- > 0)
          barrier ();
       thread_yield();
  }
  return timer_elapsed (start);
}
```

A different experiment that you may try is based on adding work of a fixed duration. You can create a work of approximately one tick duration as in function `work_a_tick()`. Conduct Experiments by adding some short works of 1 to 3 ticks before setting the alarm in function

`alarm_set()`. This can be done by making 1, 2, or 3 calls to function `work_a_tick()` outside the critical section.

Students who cannot complete this exercise in Lab Session 08 must continue working on this exercise in Lab Session 10. The tasks chosen for Lab Session 10 require students to be proficient with the lessons of Lab Session 08.

Operating Systems Lab
Dept of CSE, LNMIIT, Jaipur
05 April 2021
Lab Session 10

No separate exercise document will be prepared for Lab Session 10. However, I have uploaded some documents prepared at IIT Guwahati to provide guidance for LNMIIT students to complete Lab10 tasks.

Students are required to demonstrate successful completion of PintOS code that passes all tests in project threads except MLFQS related tests.