Software Systems
HW 05 Writeup
Nathan Lintz
4/6/2014

Introduction

In this homework, I modified the pintos timers implementation so that the timer_sleep() method would use semaphores instead of spin locks. There were two main challenges with this homework, first was installing the pintos environment, and second was completing the implementation.

Installation

Installing pintos turned out to be a non-trivial challenge. Many of the libraries would throw strange exceptions when I followed the installation directions at https://sites.google.com/site/softsys14/homeworks/homework-05. Since I couldn't figure out how to install pintos with these directions, I followed another guide (http://pintosiiith.wordpress.com/2012/09/13/install-pintos-with-qemu/) and had no problems. At this point I was up and running and ready to tackle the timer code challenge.

Implementation

I used two online resources to complete the pintos code challenge. The site (http://pintosiiith.wordpress.com/2012/10/01/assignment-2-implement-thread-block-remove-busy-wait/) gave me a general plan of attack for this assignment and told me which test cases I needed to pass to know that I had successfully implemented timers with semaphores. I also used a powerpoint (http://www.cs.utexas.edu/users/ans/classes/cs439/projects/di_project1.pdf) that I found which told me about general tips and tricks used to solve this problem such as not using malloc.

After reading the documents described above, I began writing code for this challenge. There were two files which had to be changed to get timers to work using semaphores. These files were timer.c and thread.h. In thread.h, I had to modify the thread struct and give it three new fields. The fields I added were semaphor sem, int64_t ticks, and list_elem thread_elem. The semaphore was used to sleep and wake the thread, the ticks variable stored the number of ticks a thread had until it was supposed to wake up, and the list_elem, thread_elem, was needed to put the thread into a list of waiting threads.

Modified Thread Implementation

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    struct semaphore sem; // semaphore used to wake and sleep threads
    int64_t ticks; //Ticks to wake up
    struct list_elem thread_elem; // Needs a list_elem field to be added to wait_list

    /* Shared between thread.c and synch.c. */

    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

Now moving on to the timer implementation, the first change I made was creating a list to store the waiting threads. As suggested in the lecture 17 reading, I named this list wait_list.

```
static struct list wait_list = LIST_INITIALIZER (wait_list);
```

This struct was used to store waiting threads. I used the LIST_INITIALIZER macro as suggested in the thread.h documentation. Moreover, I declared the wait_list as static because I read that the kernel can enter panic mode if you don't statically allocate a list and it grows too large.

The next change I made was updating timer_sleep. It was previously implemented using a spin lock so I changed the implementation to use the semaphore I added to the thread implementation.

```
void
timer_sleep (int64_t ticks)
{
  ASSERT (intr_get_level () == INTR_ON);

  intr_disable(); // Disable interrupts in the kernal

  struct thread *currThread = thread_current(); // Get current thread
  int64_t start = timer_ticks (); // Get the current number of ticks

  currThread->ticks = start + ticks;
  list_insert_ordered (&wait_list, &currThread->thread_elem,
    compare_ticks, NULL);
  sema_down(&currThread->sem); // Sleep le thread

  intr_enable(); // Enable interrupts in the kernal
}
```

I left in the ASSERT call at the top of the function because timer_sleep needs to have interrupts enabled when its called. After checking that the interrupts were enabled, I disabled them because I have to write to the queue in a threadsafe manner. Next, I got the current thread, stored the number of ticks until it was supposed to wake up, and inserted it into a list sorted by wakeup time. I then had the thread sleep instead of yielding like in the previous implementation. The main challenge here was realizing that the ticks needed to be stored with reference to the start time since the timer interrupt method has no knowledge of the elapsed time from when a thread was put to sleep.

The last function I modified was the timer interrupt function.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{

  ticks++;
  thread_tick ();

  struct list_elem *e;
  for (e = list_begin (&wait_list); e != list_end (&wait_list);
    e = list_remove (e))
    {
      struct thread *waiting_thread = list_entry (e, struct thread,
        thread_elem); // get the waiting thread
      if (waiting_thread->ticks <= ticks) // check if thread is ready
```

```
      {
        sema_up(&waiting_thread->sem); // wake the thread
      }
      else {
        break;
      }
    }
}
```

In the timer interrupt function, I iterate over the wait_list and look for threads that are ready to wake. Threads whose wakeup time is less than the current number of ticks are woken with a call to sema_up(). Since the list was sorted, if a thread is not ready to wake, none of the threads after it will be ready to wake. Therefore, they don't need to be checked. The hardest part of writing this implementation was realizing that you don't need to disable interrupts in this function despite the fact that you are modifying the queue in a non-thread safe manner. I still don't understand why you don't need to disable interrupts here but I have a theory that since this method is an interrupt handler, we know it will be run till completion.