CIS 520 - Programming Project #1

---- GROUP ----

>> Fill in the names and email addresses of your group members.

Melissa Coats <mcoats94@ksu.edu>
Richard Habeeb <habeebr@ksu.edu>
Mike McCall <phen@ksu.edu>
...

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for
>> the TA, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while
>> preparing your submission, other than the Pintos documentation,
>> course text, lecture notes, and course staff.

ALARM CLOCK

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

static struct list wait_list; //holds the sleeping threads in order, so that the first element of the list should wake up first
int64_t wake_up_time; //contains the time when the thread should be woken up
struct list_elem sleep_elem; //allows a thread to be in the wait_list
struct semaphore sleep_sema; //puts the thread to sleep until its wake-up time

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to your timer_sleep(),
>> including the effects of the timer interrupt handler.

For the given thread, the timer_sleep method finds a wake-up time for the thread by getting the current number of system ticks and adding the number of ticks the thread should be asleep. Then, it adds the thread to the wait_list, ordered based on when it can wake up with earlier threads first. Finally, the thread is put to sleep.

The timer interrupt is called every tick of the cpu. In the handler, we look through the list and compare the wake-up times with the current number of ticks and wake the current thread, until the current number of ticks is less than the wake up time of the current thread. Any thread we wake up is removed from the list, and we exit from the interrupt handler.

>> A3: What steps are taken to minimize the amount of time spent in
>> the timer interrupt handler?

Primarily, the wait list is ordered, so you only have to go through the list until you hit the first element that shouldn't be woken up yet, which saves a lot of time compared to running through the entire list each time. Additionally, only things necessary to fix the list and get the threads running are in the interrupt; nothing unnecessary is in there.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?

It doesn't really matter if multiple threads call timer_sleep simultaneously. All that would happen is that something might be a little delayed in getting put into the list or in being put to sleep. There shouldn't really be any conflicts caused here.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?

We used a semaphore (sleep_sema) to avoid these race conditions. Whenever timer_interrupt is called to wake up threads, sema_up is called on the sleep_sema for any threads ready to wake up.

---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to
>> other designs that you considered?

This design is just generally the most efficient and elegant solution we came up with. One really important aspect of our final design was ordering the wait_list- this allowed us to save time in the interrupt handler (as opposed to running through an unsorted list), which is an important place to shave off any excess code. Really most of the design of this program was fairly straightforward after our discussions about the project in class, so most of our original plans held over through to the final design.

PRIORITY SCHEDULING

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

```
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. */

        /* Shared between thread.c and synch.c. */
        struct list_elem elem;      /* List element. */

        /* We added these two structs. */
        struct list_elem donor_elem;    /* If this thread is donating its priority, it will be in a list */
        struct list donor_list;         /* This thread keeps track of who is donating priority to it */

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

        /* Added for alarm clock. */
        int64_t wake_up_time; /* Time when thread is to wake up. */
        struct list_elem sleep_elem; /* For adding a thread to the wait list. */
```

```
        struct semaphore sleep_sema; /* For keeping a thread sleeping until wake up time. */

        /* Owned by thread.c. */
        unsigned magic;                 /* Detects stack overflow. */
 };
```
// We added the list to keep track of the donors of priority to the thread. The list elem is added to represent this thread in other donor lists.

>> B2: Explain the data structure used to track priority donation.

We used a donor list to keep track of all the processes that were donating their priority to the current process. Each process has its own donor list, which can be empty if no other processes are waiting on a resource it is holding. The donor list contains all the threads that are waiting on the current process, and by having a link to the threads it can access their priorities.

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?

In sema_up and cond_signal we search the waiter lists for the thread with the highest priority using list_max as was done in the slides for priority scheduling. The thread returned is woken up.

>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?

If we can't sema_down, thread_donate_priority is called on the holder of the lock. Priority donation works by merely adding a list_elem referring to this thread to the donor list of the donee. This works because we search for effective priority any time we need the priority. We do not save the effective priority, as it was an unnecessary step in completing the tests.

Nested donation works because we search recursively through the donor lists of donors in donor lists when finding effective priority.

>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.

The lock_relase() cycles calls thread_purge_donors for the thread that was holding the lock. The donors that are purged from this thread's donor list are those that were waiting on the lock. The

holder of the lock is then set the null and we sema_up. This sema_up will wake up the thread with the highest priority as was mentioned earlier in this document.

---- SYNCHRONIZATION ----

>> B6: Describe a potential race in thread_set_priority() and explain
>> how your implementation avoids it. Can you use a lock to avoid
>> this race?

If you were to be searching for effective priority for a thread multiple times simultaneously when multiple threads start waiting on shared resources, it might be the case that the lower effective priority could be the one that updates the priority of the donee thread. We don't actually update priority in our implementation, so our race condition would be executing multiple lookups of effective priority simultaneously and getting back the lower one last, which would not be the proper effective priority.

A lock could be used to avoid this race. For example, if we were updating an effective priority for a thread each time some other thread starts waiting on a shared resource, we could lock the effective priority variable while each effective priority determination occurs.

---- RATIONALE ----

>> B7: Why did you choose this design? In what ways is it superior to
>> another design you considered?

We chose our design because it was the simplest way we could think of to implement priority scheduling and donation. Because we get the effective priority recursively each time we need the threads priority, we do not have to worry about when the effective priority needs to be updated.

Other than the priority donation aspect of this part, we looked for preemption possibilities in all of the obvious places in a straightforward manner (each time a thread is added to the ready list and each time a thread is done waiting on a sema/cond/lock).

ADVANCED SCHEDULER [EXTRA CREDIT]

---- DATA STRUCTURES ----

>> C1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

---- ALGORITHMS ----

>> C2: Suppose threads A, B, and C have nice values 0, 1, and 2. Each
>> has a recent_cpu value of 0. Fill in the table below showing the
>> scheduling decision and the priority and recent_cpu values for each
>> thread after each given number of timer ticks:

timer ticks	recent_cpu A	recent_cpu B	recent_cpu C	priority A	priority B	priority C	thread to run
0							
4							
8							
12							
16							
20							
24							
28							
32							
36							

>> C3: Did any ambiguities in the scheduler specification make values
>> in the table uncertain? If so, what rule did you use to resolve
>> them? Does this match the behavior of your scheduler?

>> C4: How is the way you divided the cost of scheduling between code
>> inside and outside interrupt context likely to affect performance?

---- RATIONALE ----

>> C5: Briefly critique your design, pointing out advantages and
>> disadvantages in your design choices. If you were to have extra
>> time to work on this part of the project, how might you choose to
>> refine or improve your design?