+--------------------+

| IIT 3002 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

+--------------------+

---- GROUP ----

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

Sungwook Kim andrew.s.kim95@gmail.com

Yoonjin Park sasaem12@naver.com

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, 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.

struct thread

{

tid\_t tid;

enum thread\_status status;

char name[16];

uint8\_t \*stack;

int priority;

**int64\_t wakeup;**

struct list\_elem allelem;

struct list\_elem elem;

#ifdef USERPROG

uint32\_t \*pagedir;

#endif

unsigned magic;

};

we added the int64\_t type variable called ‘wakeup’ inside struct thread. This variable will save the desired wake up time of the thread,

**static struct list sleep\_list;**

a new thread list that save the threads in the sleeping state. The list is sorted in an ascending order in terms of the wakeup time.

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer\_sleep(),

>> including the effects of the timer interrupt handler.

When the current thread calls timer\_sleep, receiving the ticks value, the function will save the wakeup time, which will be the value of ‘timer\_ticks () + ticks’. Then, the function will disable the interrupt and then save the current thread state into the sleep list. Here, we used the list\_insert\_ordered function so that, the sleep list is simultaneously sorted while it is inserted. After inserting the current thread into sleep list, the running thread is then blocked and interrupt is enabled.

The timer\_interrupt function is called everytime to increase the tick value. Everytime the tick value is increased, the function then searches the sleep\_list for threads that can be awaken. If the function finds threads with the condition (timer\_ticks () > wakeup), the threads will be awaken.

>> A3: What steps are taken to minimize the amount of time spent in

>> the timer interrupt handler?

At first we implemented the list\_sort function inside the timer interrupt handler, instead of using list\_insert\_ordered inside the timer\_sleep function. However, the problem was that every time the tick increases, the handler sorted the sleep\_list, which resulted in longer execution time. Thus, we deleted the list\_sort function inside the timer interrupt handler and instead implemented list\_insert\_ordered inside the timer\_sleep function.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call

>> timer\_sleep() simultaneously?

Inside the timer\_sleep function, we implemented the interrupt disable function prior to the procedure where we put the thread inside the sleeping list and block it. Thus, even though multiple threads calls the timer\_sleep() function simultaneously, any thread that first get to disable the timer interrupt can change the list without considering the race conditions because other threads cannot have access to the list.

>> A5: How are race conditions avoided when a timer interrupt occurs

>> during a call to timer\_sleep()?

Inside the timer\_sleep function, we implemented the interrupt disable function prior to the procedure where we put the thread inside the sleeping list and block it. Thus when the thread disables the interrupt, the timer interrupt cannot occur during execution, and therefore we need not consider race conditions.

---- RATIONALE ----

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

>> another design you considered?

The basic idea of using lists was the same, but our first model we considered was to decrease the ticks saved inside the thread. To elaborate, we first thought of the sleep function to save the ticks into the thread as the name of rest\_ticks, instead of saving the wakeup time. Then, everytime the timer\_interrupt function is called - equivalent to the increase of tick - all the values of rest\_ticks inside the sleep\_list will be decreased by value of 1. After decreasing the value of rest\_ticks, the timer\_interrupt function will search for threads that have the rest\_ticks value equal to zero using the foreach function. Once they find it, it will wake the thread up. The problem is that everytime the tick increases, we have to decrease the rest\_ticks and compare the value for all sleep\_list entries, which means it requires too many operations.

However, if we simply sort the lists and compare the wakeup time and current time, we don’t have to compare the time for all entries. Since the sleep list is sorted, once the threads that has a wakeup time larger than the current time appears, all threads behind that entry will also have a larger value. Thus once we find such threads, we can simply break out from the wakening loop, which will significantly reduce the number of operations.

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 init\_priority;

int priority; /\* Priority. \*/

int64\_t wakeup; /\* Sungwook wakeup time \*/

struct list\_elem allelem; /\* List element for all threads list. \*/

struct list donation;

struct list\_elem donation\_elem;

struct lock \*waiting;

/\* 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. \*/

};

We added integer type variable named init\_priority to save the initial priority before performing the priority donation. We then added a lock type variable ‘waiting’ which indicates which lock the thread is waiting for. Additionally, we added list variable ‘donation’ which keeps track of what threads have performed donation to the lock holding current. ‘Donation-elem’ was added for convenience.

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

>> Use ASCII art to diagram a nested donation. (Alternately, submit a

>> .png file.)

* We used the list data structure to track which thread donated its priority to a certain thread.
* L holds lock A / creates M, H1, and H2
* H2 donates priority to H1 (waiting for lock A)
* H1 donates priority to M (waiting for lock A)
* M donates priority to L (waiting for lock A)
* L releases lock A + priority roll back
* M acquires lock A (waiting == NULL) + releases lock A
* H1 acquires lock A (waiting == NULL) + releases lock A
* H2 acquires lock A (waiting == NULL) + releases lock A

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for

>> a lock, semaphore, or condition variable wakes up first?

We first sorted the waiting list of each sema list according to the thread priority. Then, we sorted the sema lists according to the priority of the thread located at the head of each waiting list. This assures that the thread waiting for a lock, semaphore, or condition variable, with the highest priority, wakes up.

>> B4: Describe the sequence of events when a call to lock\_acquire()

>> causes a priority donation. How is nested donation handled?

If the thread holds the lock, it changes it status to waiting for lock, and adds itself to the donation list for the lock holder. Then priority donation occurs in the following order.

* If (no\_holder\_for\_lock) return;
* If (lock holder’s priority is bigger) return;
* Set the lock holder’s priority = thread’s priority
* Update thread to lock holder
* Change lock value to the lock which the thread is waiting for.

For the case of nested donation, it is handled by simply increasing donation\_level. Also, in order to recover the original priority properly, we sorted the ready list and the donation list according to 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 holder is cleared.
* Remove lock from donation list and set the thread’s priority to the initial value.
* The thread with the highest priority inside the waiting queue acquires the lock.

---- 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?

We disabled the interrupt function before we 1) changed the value of the thread priority, 2) sorted the ready list(if the changed priority value is no longer the highest). We then enabled the interrupt in advance of implementing the yield function.

---- RATIONALE ----

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

>> another design you considered?