CSE 421/521
PROJECT 1: THREADS
DESIGN DOCUMENT

---- GROUP ----

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

Yizhan Wu <yizhanwu@buffalo.edu>
Zhenyu Yang <zhenyuya@buffalo.edu>
Wenting Wu <wentingw@buffalo.edu>

---- 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.

- In thread.c, added:
static struct list block_list;
//This is served to store the threads in THREAD_BLOCKED state.

- In thread.h, added attributes to struct thread:
int64_t sleep_until;
//This is served to represent the end timestamp of sleep.

---- ALGORITHMS ----

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

In a call to time_sleep():
(1) Parameter ticks will be checked to ensure validity -- ticks should be positive.
(2) Function thread_sleep_until() will be called. In thread_sleep_until(),

(a) assign value sleep_until (= timer_ticks() + ticks) to attribute sleep_until of the current thread until which the current thread should sleep;
(b) insert the current thread to block_list in the ascending order of attribute sleep_until;
(c) call function thread_block() to block the current thread.

In the timer interrupt handler timer_interrupt():
Attribute sleep_until of threads in block_list will be compared to the global ticks. Threads of which sleep until is smaller than the global ticks will be removed from block_list and unblocked.

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

(1) In time_sleep(), insert the current thread to block_list in the ascending order of attribute sleep_until.
(2) In timer_interrupt(), compare sleep_until of threads in block_list to the global ticks in order and stop comparison when meeting the thread of which sleep_unitl is not larger than the global ticks.

---- SYNCHRONIZATION ----

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

Interrupts are off in timer_sleep() when assigning value sleep_until and inserting the current thread to block_list, thus race condition will not occur.

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

Interrupts are off in timer_sleep().

---- RATIONALE ----

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

The reason we chose this design is that it can avoid busy waiting and is easy to design and implement -- move threads from read_list to block_list when going to sleep and then move back when waking up. We haven’t come up with other designs.

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.

- In thread.h, added attributes to struc thread:
int original_priority; // the original priority
struct lock locked_by; // lock this thread*
struct list threads_locked; // threads locked
struct list_elem donate_elem; // list element for threads_locked

- In thread.c, added:
static const int DEPTH = 8; // limit on depth of nested priority donation

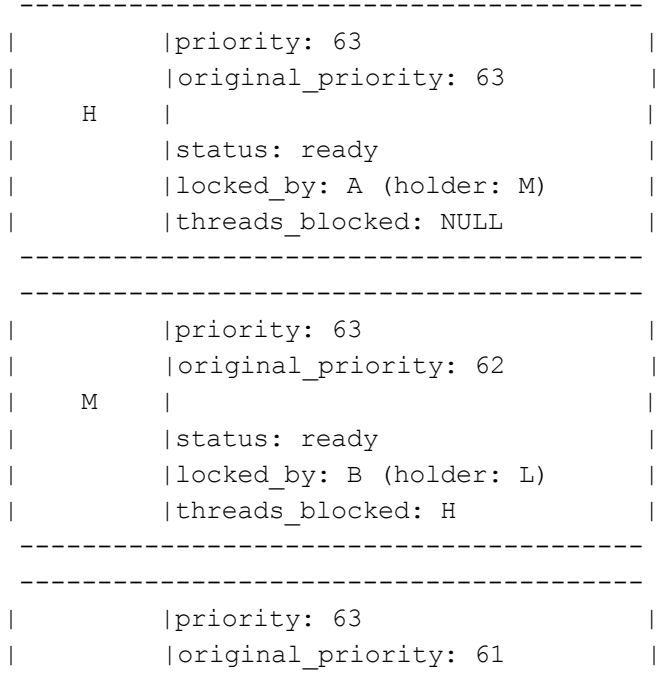
- In synch.c, added attributes to struct semaphore_elem:
struct thread* holder; // used for the threads waiting on COND to wake up in order of priority

>> B2: Explain the data structure used to track priority donation.
>> Use ASCII art to diagram a nested donation. (Alternately, submit a
>> .png file.)

We use the following four attributes in struct thread to track priority donation:
int original_priority; // the original priority
struct lock locked_by; // lock this thread*
struct list threads_locked; // threads locked
struct list_elem donate_elem; // list element for threads_locked

H, M, L are high priority, medium priority and low priority threads.

Phase 1: H is locked by M via lock A, M is locked by L via lock B. H donates its priority to L and M. L is running.



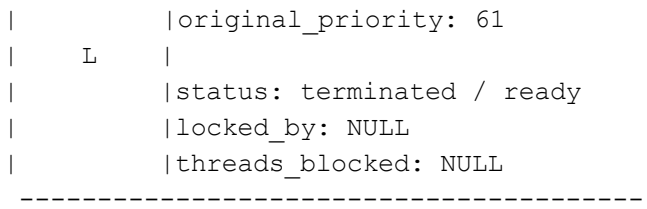
```
|    L    |                                |
|         |status: running                 |
|         |locked_by: NULL                 |
|         |threads_blocked: M              |
 ----------------------------------------
```

Phase 2: Lock B is released. L is back to original priority. M is running.

```
 ----------------------------------------
|         |priority: 63                    |
|         |original_priority: 63           |
|    H    |                                |
|         |status: ready                   |
|         |locked_by: A (holder: M)        |
|         |threads_blocked: NULL           |
 ----------------------------------------
 ----------------------------------------
|         |priority: 63                    |
|         |original_priority: 62           |
|    M    |                                |
|         |status: running                 |
|         |locked_by: NULL                 |
|         |threads_blocked: H              |
 ----------------------------------------
 ----------------------------------------
|         |priority: 61                    |
|         |original_priority: 62           |
|    L    |                                |
|         |status: terminated / ready      |
|         |locked_by: NULL                 |
|         |threads_blocked: NULL           |
 ----------------------------------------
```

Phase 3: Lock A is released. M is back to original priority. H is running.

```
 ----------------------------------------
|         |priority: 63                    |
|         |original_priority: 63           |
|    H    |                                |
|         |status: running                 |
|         |locked_by: NULL                 |
|         |threads_blocked: NULL           |
 ----------------------------------------
 ----------------------------------------
|         |priority: 62                    |
|         |original_priority: 62           |
|    M    |                                |
|         |status: terminated / ready      |
|         |locked_by: NULL                 |
|         |threads_blocked: NULL           |
 ----------------------------------------
 ----------------------------------------
|         |priority: 61                    |
```



---- ALGORITHMS ----

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

Insert threads into waiting lists in the descending order of priority and pop out the first thread in the lists.

Lock -- ready_list
Semaphore -- sema->waiters
Condition -- cond->waiters

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

The priority p1 of the current thread T1 will donated to the holder T2 of this lock if p1 is higher than the priority of the lock holder and then the lock holder T2 will donate the higher priority to the its lock holder T3 if it’s locked ...

Nested donation is handled by keeping track of locks mentioned above.

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

(1)The threads locked by the lock will be removed from threads_blocked.
(2)The priority of the current thread will be updated to the highest priority in threads_blocked.
(3)The holder of the lock will be set to NULL.
(4)Sema up.
(4)The current thread will yield the CPU to the higher-priority thread.

---- 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?

In thread_set_priority() and priority donation, priority can be changed in both ways.

A lock is not able to avoid this race, since the interrupt handler cannot acquire locks.

---- RATIONALE ----

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

The reason we choose this design is that the logic behind this design is easy to understand and implement -- when lock is acquired, we build the structure to keep trace -- the thread locked is connected to the holder of lock and to the threads it locks, which is like a tree. We haven't come up with other designs.

ADVANCED SCHEDULER

---- 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.

- Added a new header file fixed_point.h, defined a new type called *fixed_point* which is used for fixed point calculation:
typedef int fixed_point;

- In thread.h, added new variable:
int nice
fixed_point recent_cpu
fixed_point load_avg
//These variables are for the use of calculating priority of threads.

---- 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	recent_cpu			priority			thread
ticks	A	B	C	A	B	C	to run
0	0	1	2	63	61	59	A
4	4	1	2	62	61	59	A
8	7	2	4	61	61	58	B
12	6	6	6	61	59	58	A
16	9	6	7	60	59	57	A
20	12	6	8	60	59	57	A
24	15	6	9	59	59	57	B

28	14	10	10	59	58	57	A
32	16	10	11	58	58	56	B
36	15	14	12	59	57	56	A

>> 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?

Ambiguities are mainly in the area of deciding values in the table, for example, we don't consider values like load_avg, recent_cpu, as well as priorities for each threads. In that case, we assume that each time slice is 4 ticks, thus we will add 4 ticks to recent_cpu for every 4 ticks. We also assume that if both the current running thread and the ready thread have same highest priority, current thread will continue to run, which means round robin scheduling. The scheduling method is also used in our implementation.

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

If values like load_avg, recent_cpu need to be calculated for all threads every time, CPU will likely waste too much resource on that. Therefore, for a system that has a lot of threads, tuning up the cost of scheduling inside the interrupt context will affect the overall 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?

Advantages :
Only added a few extra variables, which makes it easier to modify. Fixed point calculation is done using functions and logic that system already supports. Also minimized the influence to system performance while keeping the function intact.

Disadvantages:
Turning off interrupts may affect performance due to flexibility of allocating resources is limited.

To improve the design, what we can do is using a lock for variable rather than turning off interrupts.

>> C6: The assignment explains arithmetic for fixed-point math in
>> detail, but it leaves it open to you to implement it. Why did you
>> decide to implement it the way you did? If you created an
>> abstraction layer for fixed-point math, that is, an abstract data
>> type and/or a set of functions or macros to manipulate fixed-point
>> numbers, why did you do so? If not, why not?

Fixed-point math is implemented in a seperate header file. Fixed-point math provides a small, fast alternative to floating-point numbers in situations where small rounding errors are acceptable. The reason to implement it in header file is that standard functions were used during implementation, and to call these functions in thread.c during calculation is a good reason for implementation. When using the fixed-point variables in an application, abstracting functions and using new variables also provides good readability.

SURVEY QUESTIONS

Answering these questions is **optional**, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems
>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in
>> future quarters to help them solve the problems? Conversely, did you
>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?

>> Any other comments?