+------------------------------------+
| ECE 434 |
| PROJECT 1: THREADS |
| DESIGN DOCUMENT |
+------------------------------------+

---- GROUP ----

Brian Faure <bdf39@scarletmail.rutgers.edu>
Adam Massoud <ajm323@scarletmail.rutgers.edu>
Wanshu (Wayne) Sun <ws269@scarletmail.rutgers.edu>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, please give them here.

The following tests fail:
priority-donate-nest
priority-donate-sema
priority-donate-lower
priority-sema
priority-condvar
priority-donate-chain

>> 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.

http://web.stanford.edu/class/archive/cs/cs140/cs140.1088/misc/project1-sum08.pdf
https://tssurya.wordpress.com/tag/priority-donation/
http://knowledgejunk.net/2011/05/06/avoiding-busy-wait-in-timer_sleep-on-pintos/

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.

timer.c
static struct list sleeping_threads
- List of sleeping threads; sorted by sleep_time, acts as a priority queue for threads that are blocked

thread.h
→ *The following are all within the 'thread' struct...*
int 64_t sleep_time

- The amount of time/clock ticks a thread will sleep

struct list_elem alarm_elem

- List element iterator pertaining to current thread in the sleeping_threads list

struct semaphore sleep_sem

- Semaphore used to maintain sleeping threads (implemented as a lock)

---- ALGORITHMS ----

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

timer_sleep()
- calculated time (sleep_time) for the current thread to wake up by adding ticks to the global ticks variable timer_ticks()
- called function sleep_list_insert() to properly insert the current thread into our sleeping_threads list based on sleep_time (list maintains a sorted order with the lowest sleep_time value will be at the beginning of the list
- sema_down(&sleep_sem) called to put thread to sleep

timer_interrupt()
- if the sleeping_threads list is not empty, look through the list to see if any threads are ready to be woken (based on if ticks is less than the current thread's sleep_time)
- else, do nothing

---- SYNCHRONIZATION ----

>> A3: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?

Only one thread will be executing at once, while the other threads are sleeping while stored in the list.

---- RATIONALE ----

>> A4: Why did you choose this design? In what ways is it superior to
>> another design you considered?

This design was chosen because it is relatively simple to implement a list of threads. Furthermore, it is very sensible, as if you think of threads as elements sorted based on their wait time, you basically implement a priority queue. So, whenever a thread is being run, the rest of the threads in the priority queue are sleeping. Once the thread is finished running, you simply pop from the front of the list while the rest remain sleeping.

The implementation used seemed to be the only feasible way to avoid busy waiting.

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.

thread.h

int orig_priority

- original priority of a thread (before being donated to)

int donation_array[MAX_DONATION_CT]

- array to keep track of donations. Allocated to size of MAX_DONATION_CT] to implement nested donations

struct lock* waiting_for_lock

- set to NULL if thread is not waiting for lock, set to the lock if thread is waiting for lock. Used for priority donation

synch.h

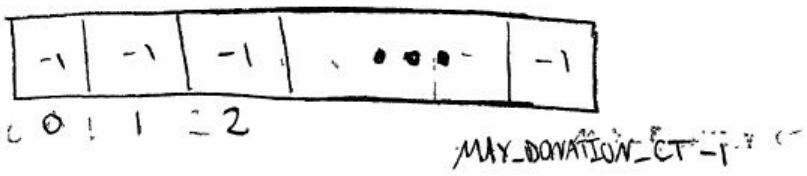
```
struck lock{
        struct list_elem *holder;          // originally - struct thread *holder;
        struct semaphore semaphore;
};
```

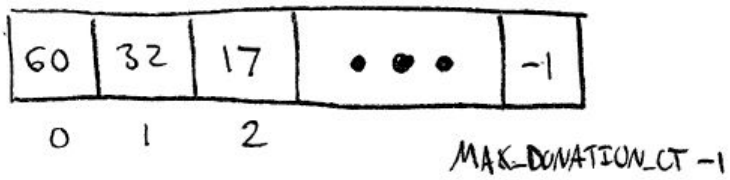
- added holder to identify who is holding the lock

Every instance of list_insert has been changed to list_insert_ordered for the most part. We created a function called

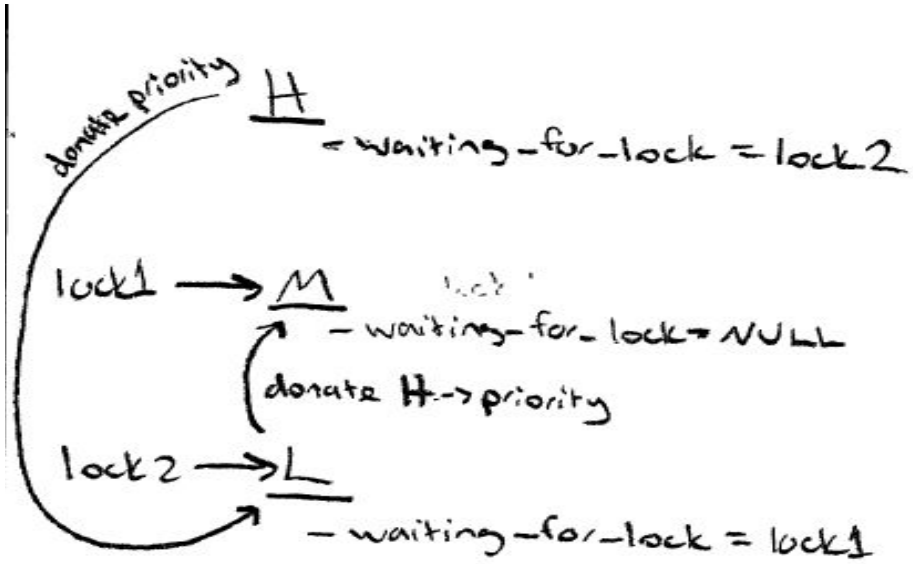
>> B2: Explain the data structure used to track priority donation.

We used the array called donation_array to monitor donation. We initially set each element of the array equal to -1 to identify that they aren't currently holding multiple donations. Once we start getting into multiple donations, the donations are added and removed to help keep track of the locks. We also kept track of the highest donation value by placing it in the front of our array at all times. Unfortunately, we weren't able to fully implement the donation process which resulted in failed tests.





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



---- ALGORITHMS ----

bool thread_priority_list_less_func(const struct list_elem* a, const struct list_elem* b, void* aux UNUSED)
- returns true if the priority of thread a is less than priority of thread b. Used to make sure that priority list is in descending order (highest priority at front)

void thread_requeue(struct list_elem*)
- puts thread into ready queue where it belongs. Called after donations to put thread with needed lock at front of queue

void insert_ready_list(struct list_elem* el)
- takes thread from sleeping queue and inserts it into ready queue in order from highest priority to lowest

void thread_set_priority(int new_priority)
- sets thread priority to new_priority

void thread_get_priority(void)
- returns current thread's priority

static void init_thread(struct thread* t, const char* name, int priority)
- initializes thread

void donate_priority(struct lock* lock, int track)
- Manage the donation of priority in the case of priority inversion

void lock_release(struct lock *lock)
- Changed to ensure that when a lock is released, the threads which received priority donations have their corresponding donation_array values reset back to -1. This ensures that there are no "leaks" of priority.

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

Whenever we used locks, we took advantage of the semaphores that they contain. We sorted the the threads onto the waiting lists of the locks by priority order. We were not able to solve the condition variable problem.

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

After traversing the 3 ASSERT functions, the first function call within lock_acquire is to the donate_priority function. We created this function to manage the assignment of donations from high priority threads to the lower priority threads. Once inside the donate_priority function for the first time, the if(lock->holder == NULL) statement will cause the function to return if evaluated to true. This means that if the lock which the current thread is trying to acquire is not held by any other thread, it will not have any reason to donate priority thus it will leave. Otherwise, the thread-pointer "thread_with_lock" is assigned to the thread currently holding the requested lock. After this, the function checks whether or not the thread_with_lock thread is running on a donated priority by checking whether the thread_with_lock priority is different than the thread_with_lock original priority ("orig_priority"). If so it goes through the thread_with_lock donation array to find the first available spot (corresponding to value of -1) and moves the priority into that spot. When finished, it makes the thread_with_lock priority equal to the thread_current priority and continues. At this point, the function checks to see if the thread_with_lock is waiting for any other threads to free a lock. If not, it calls the requeue function (find new place for thread_with_lock in the ready queue) and returns. If so, it calls itself again with the new input being the lock required by thread_with_lock while, at the same time, incrementing the track count. On each iteration of the donate_priority function, the track variable is checked against 8 and if it is greater than or equal to it the function returns because this corresponds to the fact that it has traversed 8 levels deep looking for a thread not-dependent on a lock.

---- SYNCHRONIZATION ----

>> B5: Describe a potential race in thread_set_priority() and explain
>> how your implementation avoids it.

In our implementation, we avoid the chance for race conditions by disabling interrupts during this important portion of the code. Because interrupts are disabled, external threads will not be able to breakthrough to the CPU and end the execution of the current thread.

---- RATIONALE ----

>> B6: Why did you choose this design? In what ways is it superior to
>> another design you considered?

We decided to use an array to hold the nested donation values because they allow for random access and are easier to use within for loops when compared to the list provided. We also thought it would be easier to use an array because the list implementation would have forced us to abide by all of the lengthy operations involved with traversal and iteration through the provided lists. Because we are handling simple integers it would not make sense to implement their organization with a data structure that would overcomplicate the operation of the overall system.

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?

The assignment as a whole took an extremely long amount of time. The only resource being the Pintos Document made this very challenging. Going through the internet proved to be useless. Overall, the project instructions should be more guiding because even after you finally figure out what you want to implement, doing so is no easy task. This project was excessively time consuming and difficult; this was assigned to students without a good knowledge of C and a lack of sufficient instructions/resources.

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

Working on this project requires you to understand threads very well. In order to implement the scheduler, you have to be familiar with exactly what is happening in every line of code. The alarm clock implementation helped us get a better understanding of interrupts as well.

>> 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?

A hint to students in the future would be to initially go through the files and try to understand what each function and data structure does/is used for. It would be helpful to give some pseudo code for each problems just so people know where to start. Otherwise you task is very daunting for the first 5 or so hours. Furthermore, we were told that printf could be placed virtually anywhere to help with debugging, but wherever we placed printf statements we would receive errors because the test compared the console output to the “correct” console output (which we were never told). In addition, we went to office hours and were told that thread.c would be the only file we would have to alter in order to implement priority scheduling. It turned out that thread.c was altered a little bit, while the synch files were altered a lot in order to implement the solution we had.

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

We would like to suggest that the test cases come provided with a short summary of what they are testing rather than just having a short 3 word name. Because debugging is painful and hard to learn with this OS’s, a better explanation of what the tests are checking for would allow students to have a better understanding of exactly what is going wrong with their code when they fail certain tests.