PROJECT 1: THREADS
DESIGN DOCUMENT

---- GROUP ----

Xiaojing Hu xhu7@buffalo.edu
Jingyi Zhao jzhao49@buffalo.edu
Zishan Liang zsliang@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.

1. *A comprehensive introduction of stanford's pintos* http://www.cnblogs.com/laiy/
2. *Strengthing Pintos* https://github.com/shrinivasiitg/Strengthening-PintOS
3. *Pintos guide* https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html
4. *Priority inversion and donation* https://web.eecs.umich.edu/~akamil/teaching/sp04/pri/
5. *MLFQS* http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf

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.

thread.h

```
int ticksRemain /*used to record the remaining ticks a thread has before awake*/
struct list sleep_list   /*store the  threads that are put to sleep*/
```

---- ALGORITHMS ----

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

During a call to timer_sleep,

1. **a thread is assigned its ticksRemain**
2. **the thread is inserted in the sleepList using insert_inOrder by comparing its ticksRemain so that the entire sleep list is kept sorted all the time.**
3. **the thread is then blocked**

During a call to timer_interrupt

1. **the interrupt goes in to check if certain threads needs to be awaken.**
2. **it iterates over all sleepList threads to minus their ticksRemain by 1**
3. **if any of the thread's ticksRemain become 0, it triggers a removal. Otherwise the removal is skipped**
4. **during removal, a thread is unbloced by calling thread_unblock() function and then being pushed back in the ready_list. thread_unblock is sorted each time a new thread comes in, according to its priority**

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

Minimizing interrupt hander's action:

in the interrupt handler, inside updateTicks function, there is a flag that become true once a thread's ticksRemain == 0. If this is the case, removal is performed. Otherwise, removal operation is skipped in order to minimize the operation spent in timer_interrupt.

---- SYNCHRONIZATION ----

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

we used interrupt disable. at each call to timer_sleept, the interrupt is then disabled before any threads are inserted to sleep_list. The operations performed inside is atomic operation.

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

The same as before, we used interrrupt disable.

---- RATIONALE ----

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

we considered LaiY's algorithm (1). what he do is basically assign a ticksRemain to each thread during timer_sleep. In timer interrupt, ticksRemain for all threads in all_list is decremented in an iterative manner each time. And then, the entire all_list is iterated over again to search for ticksRemain == 0 in order to awake up threads. Our design is superior in that we used a sleepList to narrow our search scope, and to exclude threads that are not in THREAD_BLOCKED state. We also minimize operations inside timer_interrupt as we mentioned earlier.

Our design is also much better than the original busy waiting in that we save cpu.

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

```
        struct lock* targetLock;
 /* the lock that the thread is try to acquire but     cannot obtain for now */
        struct lock* donatedLock;
/*  lock for which the thread has been donated priority due to it*/
        struct list listOfInfo;
        /* To store the infos in a list */

        struct info{
                int histPriority;
        /*  priority donated to this thread in previously */
                struct lock* histDonatedLock;
        /*the lock that the current thread has been donated due to other thread acquiring this lock */
                struct list_elem elem;
        /*  listElement to insert */
                }
```

sync.h

```
        struct lock:
                struct list waiters;
        /*  list of threads waiting for this lock */
```

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

info & infoList:

info is a customized struct that is used to store a pair of data : for thread A's info, its histPriority is the priority that thread B donates; its histDonatedLock is the lock that thread B try to acquire from A but fails so that thread B donates priority. Thus, listOfInfo of thread A stores all the priority that A had in history, including starting priority. Whenver A is donated higher priority, its previous priority , and its previous donatedLock is pushed top into a list of info(listOfInfo). Whenever A release a lock, its priority is reset, and in the next release, its priority would be recovered to the top of this listOfInfo

---- ALGORITHMS ----

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

a thread once unblocked, is inserted into the ready_list according to its priority. if the running thread has a lower priority than the thread just unblocked, this current thread calls thread_yield. the thread_yield schedule the threads in order to give the thread with the highest priority. Here, it calls next thread_to_run. I reimplement this function using list_max() to return the max priority thread. Sema->waiters , lock->waiters don't automatically sort itself Like I implemented for thread_unblock. For the latter 2 waiters, use list_max() to return the max priority thread.

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

When the current thread tries to acquire lock, I use a while(!try_acquire_lock) as a criteria to see if nested donation is needed. While the current thread keeps not acquiring the lock, it donates its priority to the lock that is holding this lock. Then, lock -> waiters = current thread, and the current thread is blocked. during nested donations, the thread Y that is receiving higher priority from thread X than its current priority is checked for nested donation. if thread Y is waiting on some lock, then the thread Z on which this lock is waiting is donated priority. eventually Z will have the as high priority as X (assuming original X > Y > Z). this process is implemented iteratively by calling refreshPriority, for example, refreshPriority (Y, x's priority, x's targetLock).

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

when the lock is released, the highest priority thread in current lock's waitinglist is unbloced and thus acquire this lock. if the current thread has been donated priority due to this other thread acquiring this lock, the thread's priority is recovered to previous priority in its listInfo. Iterate over curr lock's listOfInfo to erase any info that is associated with this 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 his race?

it can happen during priority donation: a thread could be donated priority by other thread while also trying to call thread_set_priority(). when we are doing priority donation in lock_acquire, we diable the interrupt, so thread_set_priority() will not be called until the end of propagation of donation. Since thread dont have any field that resembles a shared lock, it cannot be avoided.

---- RATIONALE ----

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

This design use a list to store all nested donation history(info). it also iteratively check to ensure the donation is actually propagate to the end. The added ListOfInfo make it easier to to refresh thread's priority during lock acquire or lock release. There is no other design that we came up yet.

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.

fixed_point.h

```
Declare a number of fixed point operations
typedef fixed_t            /* defined type, used to represent its a float number */
```

thread.h

in struct thread:

```
int nice;                  /* threads' current nice value */
fixed_t  recent_cpu;       /* threads recent cpu calculated */
```

thread.c

```
fixed_t load_avg       /*  average number of threads ready to run over the past minute */
```

---- 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	B	C	priority A	B	C	thread to run	Thread with same priority
0	0	0	0	63	61	59	A	B, C
4	4	0	0	62	61	59	A	B, C
8	8	0	0	61	61	59	B	A, C
12	8	4	0	61	60	59	A	B, C
16	12	4	8	60	60	59	B	A, C
20	12	8	0	60	59	59	A	B, C
24	16	8	0	59	59	59	B	A, C
28	16	8	0	59	59	58	A	A, B
32	16	12	4	59	58	58	A	C, B
36	20	12	4	58	58	58	B	B, 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?

ans: if more than 2 threads has the highest priority, then pop the thread at the front of the ready list.
ans: when same priority occurs, round robin method is used.

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

ans: We do the scheduling completely inside the timer_interrupt handler. Since timer_interrupt is so frequently called, we would have to spend a lot of resource calculating recent_cpu thread priority,and load_avg. Although this method is quite straightforward and seems past the test, it will lower the performance. we did not figure any other way to update cpu/priority outside timer_interrupt.

---- 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?

pro: straight forward, 2 if conditions used in interrupt handler to decide when to update.recent cpu, priority of a thread.

con: as answered in the previous question, the scheduling is all done inside the interrupt handler,it would likely to affect performance once the thread numbers goes up. We heard that MLFQS is partly based on Priority Scheduler. As we implement Priority scheduler and MLFQS separately, we failed the test of MLFQS-1 and MLFQS-load-60, and we don't have much time to figure out why.

>> 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?

ans: We define type of fixed_t in order to mimic the floating number that is not available in Pintos. This fixed_t type is used in load_avg and recent_cpu calculation.These functions are quite straightforward and is not going to change, so we used Macro to define it in the header. Calling these arithmetics are less messy in this way.