```
+--------------------+
|       CS 521       |
| PROJECT 1: THREADS |
|   DESIGN DOCUMENT  |
+--------------------+
```

---- GROUP ----

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

yuxiang liu <yliu268@buffalo.edu>
minghua wang <minghuaw@buffalo.edu>
shaoming xu <shaoming@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.h
1. Add these elements in struct thread:

```
/* Element for timer_sleep function*/
struct list_elem telem;        /*Element of timer_sleep_list*/
int64_t stop_time;             /*Time that thread stops sleeping*/
struct semaphore sleep_sema;   /*Semaphore to control thread’s sleep*/
```

In timer.c
1. static struct list timer_sleep_list;/*List to record the sleeping thread*/

In synch.c
1. define function: sema_down_by_sleep (struct semaphore *sema, struct list *sleep_list) /*push back thread to sleep_list as soon as doing sema_down.*/

---- ALGORITHMS ----

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

timer_sleep (int64_t ticks) accepts the ticks. If ticks number is smaller than zero, the current thread doesn’t need to yield the CPU. Otherwise, the current thread sets its stop_time as the timer_ticks() + ticks. Then, its semaphore will be initialized to zero and calls sema_down_by_sleep. The sema_down_by_sleep function accepts thread’s sleep sema and timer_sleep_list. Then it will push current thread back to timer_sleep_list as soon as doing semaphore down.

When timer_interrupt() been called, ticks plus one. Then we check timer_sleep_list and sema_up all threads which stop_time is smaller or equal than ticks (current timer) and remove them from timer_sleep_list. Do sema_up on thread will make it go back to ready queue.

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

First, the timer_sleep_list is sorted by stop_time in increased order, So timer_interrupt() will stop its searching as soon as it the stop_time larger than ticks(current). It does not need to search through whole timer_sleep_list.

Second, we only do semp_up one those thread that stop_timer is smaller or equal than ticks(current time).

Third, each thread keeps its own semaphore which means when doing sema_up the sema->waiters list only has one elements making it bound to O(1).

---- SYNCHRONIZATION ----

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

The timer_sleep() function only has one global parameter – timer_sleep_list.So we write a sema_down_by_sleep function and it disables the interrupt and then operates the timer_sleep_list to avoid race condition

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

Because in timer_sleep() the operations on timer_sleep_list are inside sema_down_by_sleep(). sema_down_by_sleep() calls intr_disable(), therefore we can make sure timer interrupt can’t interrupt sema_down_by_sleep() when it does list_push_back operation on current thread’s telem.

---- RATIONALE ----

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

To deal with questions A4 and A5, the other design we considered is using lock.

We set lock between the operation on timer_sleep_list which is exposed in timer_sleep. After the operation finished, timer_sleep does sema_down on thread’s semaphore.Then we use lock_try_acquire in timer interrupt. If lock_try_acquire success, interruptor can operate timer_sleep_list. Otherwise, it can’t and exits.

The lock design has a problem. If timer interruptor happens in the gap between lock_release and sema_down in the function timer_sleep(), the thread already in the timer_sleep_list but its sema_down hasn't been operated. This will make error in interruptor that the thread can't not be sema_up when it needs to be awaked.

To bundle the operations of timer_sleep_list and sema_down can perfetly solve this problem.

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

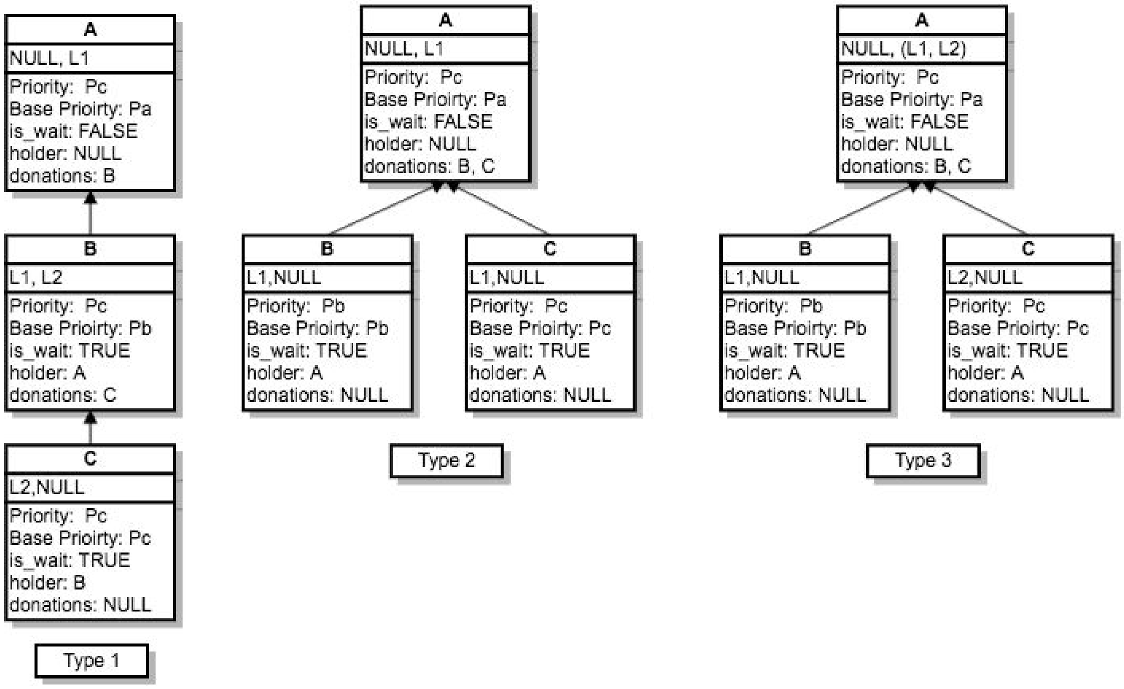
```
/* Used for priority donation*/
struct donation
{
  struct thread* donator;       /* Pointer to the donator thread */
  int priority;                 /* Donate priority*/
  struct list_elem delem;       /* list_elem*/
};
```

'struct' memebers of struct thread

```
    /* Element for priority donation */
    bool is_wait;                 /* TRUE if thread is waiting for lock*/
    struct thread *holder;        /* Saved lock holder pointer */
    int base_priority;            /* Saved priority without donation */
    struct list donations;        /* Saved list of donations prioritys */
```

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

```
  notation rules:
    L means lock.
    B(L1,L2) means B needs L1 and B holds L2.
    A(NULL,(L1,L2)) means A doesn't needs lock and it holds L1 and L2 locks
    A<B means A's priority smaller than B's.
  Types of nested donation:
   Type1: A(NULL,L1)<B(L1,L2)<M(NULL,NULL)<C(L2,NULL)
   Type2: A(NULL, L1) < B(L1,NULL) < M(NULL,NULL) < C(L1, NULL)
   Type3: A(NULL,(L1,L2)) < B(L1, NULL) < M(NULL,NULL) < C(L2, NULL)
```



---- ALGORITHMS ----

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

We rewrite sema_up() function. In sema_up(), the thread_unblock() unblocks the thread with highest priority in the sema->waiters list. sema->waiters list records all threads waiting for the lock.

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

Suppose There are three thread A(NULL,L1)<M(NULL,NULL)<B(L1,NULL). A(NULL,L1) means thread A doesn't need lock and it has lock L1. A<M means A's priority smaller than M.

Now A invokes lock_acquire()function and gets lock L1 before M and B threads start. In lock_aqcuire, A invokes sema_down_by_lock(lock) function which has a while loop to check if sema->value == 0. Since lock not been acquired yet, sema->value is 1, A can skip loop and do sema->value-- operation and goes back lock_acquire(). Then A updates its is_wait = false and holder = NULL to mark itself as the lock holder and then it announce this truth to all threads waiting in lock->semaphore.waiters list.

Now, M starts operating. Since M > A, M kicks A out CPU. Then B comes, B needs L1 so it invokes lock_acquire(). In lock_acquire() B invokes sema_down_by_lock and falls into the while loop since sema->value == 0. In the loop, B push itself in sema->waiter list and invokes donate_to_thread to donate its priority to lock->holder which is A now. Then it updates its is_wait to be true and its holder to be lock->holder and then invoke thread_block(). thread_block() invoke schedule(), therefore A can kick M out and starting running.

donate_to_thread (struct thread* holder, struct thread* donator, int priority). The holder points to the thread which holds the lock. The donator points to the thread which tries to donate lock and to acquire lock. The priority is donator's priority. First, if priority>holder->priority is true, donate_to_thread() updates the holder->priority to be donator's priority. Then inside the first if condition, we check if holder->is_wait is true, if it is true，donate_to_thread() starts recursively calls itself. Similar to bubble sort, the first step makes high priority ascend to right position in the donation chain. Second, after the recursive call, in holder->donations list we try to find the donator, if success, update its priority to the priority that is a parameter of donate_to_thread() function. If failure, we make a new donation struct to record the donator thread and push it to the holder->donations list.

The donate_to_thread guarantees nested donation work accurately. Let us consider these situations.

First, A(NULL,L1)<B(L1,L2)<M(NULL,NULL)<C(L2,NULL). B(L1,L2) means B needs L1 and B holds L2. L denotes lock. A<B means A's priority smaller than B. First A starts and gets L1. Then M comes and kick A out. Then B comes and donates its priority to A, since B<M after donation A's priority still lower than M's. Finally C comes, C invokes donate_to_thread and recursively donates its priority to B, and B delivers its new priority to A. Since C > M, now A > M. A can run now.

Second, A(NULL, L1) < B(L1,NULL) < M(NULL,NULL) < C(L1, NULL). A comes first and M comes second, so M kick A out. Now C comes and donates its priority to A make A > M, A starts run. Then B comes and try to donate its priority to A, since now B < M < A, B can't do donations.

Third, A(NULL,(L1,L2)) < B(L1, NULL) < M(NULL,NULL) < C(L2, NULL). A(NULL,(L1,L2)) means A doesn't needs lock and it has L1 and L2 locks. Same as before, M kicks A out. Then B comes and donates its priority to A. Then C comes and donates its priority to A directly. After that A>M, A starts run.

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

Take the third situations A(NULL,(L1,L2)) < B(L1, NULL) < M(NULL,NULL) < C(L2, NULL) as example. Now A calls lock_release() on L2. In lock_release() it set lock->holder = NULL and then cleans its donations list by removing all thread related with L2 in donations list by calling remove_donation_from_thread() function. After that it invokes calculate_priority() to set itself priority to be the highest priority among its base_priority and those priorities recorded in its donations list. Finally, it will invokes sema_up(). sema_up() will invoke thread_unblock() on the the lock waiting thread with highest priority(Now it is C). Then the thread C has higher priority than all thread in ready list (remember the thread_unblock() has yield current thread back to ready list), so in lock_acquire() C awakes and can operate codes after sema_down_by_lock(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?
  In thread_set_priority() we calls calculate_priority() to compare the thread base_priority with the priorities in its donations list. And the donations list will be changed when other threads try to acquire the lock. Race happens. To solve it, we disable interruption when thread_set_priority() needs to manipulate the donations.
  It seems we can't use lock in this situation. Because if we set lock in donations list, when other threads try to acquire the lock, in lock_acquire() they will try to access holder's donations list. If they have to ask lock_acquire() before accessing the list, there will be nested lock_acquie(). So we can't use lock here.

---- RATIONALE ----

>> B7: Why did you choose this design? In what ways is it superior to
>> another design you considered?
  We choose a list of struct donation, and a base_priority to save priority without any donation;

```
# in struct thread:
  struct donation
  {
    struct thread* donator;        /* Pointer to the donator thread */
    int priority;                  /* Donate priority*/
  };
  int base_priority;
```

  We have several reason to choose this design:
1. We want to retrieve priority after one thread release its lock, then it may comes to problems:

How can we delete donations just come from this lock, still remain donations from other lock.

   The problem raises because that consider situation thread A(priority = 1) holds lock 1 and lock 2, and thread B (priority=2) acquire lock 1, and thread C (priority=3) acquire lock 2, so now A has both donations from B and C, and has priority 3. Now A release lock 2, so we need to delete donation from C which is 3, and A should still keep donation from B, which give it priority 2 now.

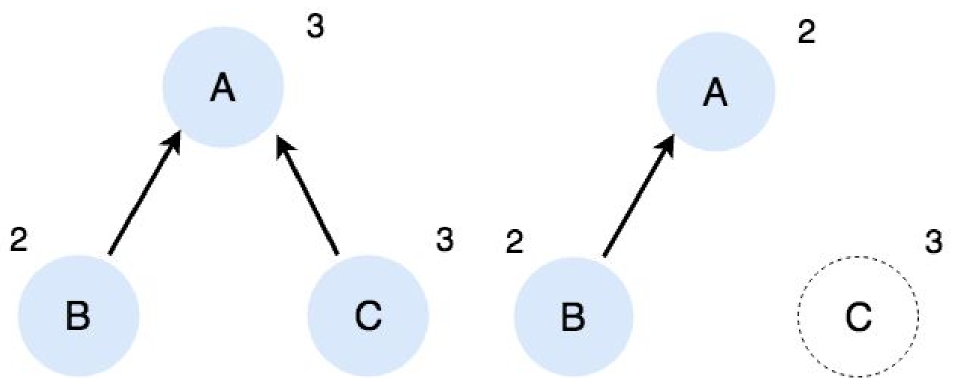


Figure 1 Delete Donation from released lock

At first, we use a single priority to save priority after donations, this approach cannot solve this problem apparently. So we need to use a list of donations instead, and how can we know which lock each donation come from, so that we can delete corresponding donations after lock released? We use a pointer to the donator thread so that every time a thread release a lock, it make check the waiters for the lock, and delete every donation come from the thread in the waiters list.

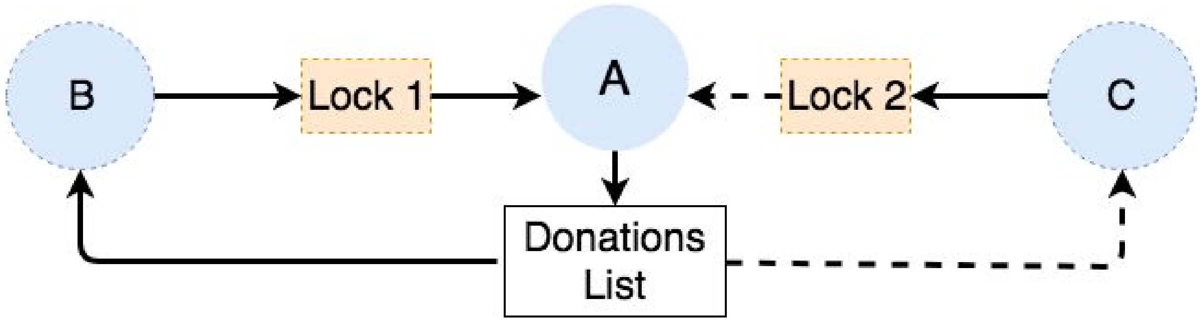


Figure 2 Detail when A release lock 2

We choose to save the pointer to the thread to indicate thread instead of thread id. The reason is that when we allocate id to thread, we need to acquire lock, so at that time donation happens but we haven’t id for the thread yet, so it is not a appropriate approach.

2. We need to handle set_priority function correctly.

When set_priority happens, we only change the priority without concern any donation. So We use a single integer base_priority to save that priority. So when set_priority happens, we only need to change the base_priority and check if we have another higher donation priority.

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.

In thread.h

```
  /* Element for advanced Scheduler */
    int nice;                       /* Nice value */
    fix_p recent_cpu;               /* Recent cpu */
```

In thread.c

```
  /* Number of thread in ready list except idle_thread */
      static int ready_list_size;
  /* Load avg for mtfq Average number of threads ready to run over the past
minute */
```

```
      static fix_p load_avg;
  /* Number for calculate load_avg p1 is fixed point 59/60 p2 is fixed point
1/60 */
      static fix_p l1;
      static fix_p l2;
```

In fixedpoint.h (new file)

```
  typedef int fix_p;             /* the fixed point value*/
  #define DIFF 16384             /*2 power of 14. Using to convert integer
to fixed point value*/
```

---- 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:

	recent_cpu			priority			thread to run
timer ticks	A	B	C	A	B	C	
0	0	0	0	63	61	59	A
4	3	1	2	62	60	58	A
8	6	2	4	61	60	58	A
12	9	3	5	60	60	57	A
16	11	4	6	60	60	57	A
20	13	4	7	59	60	47	B
24	11	8	8	60	59	57	A
28	13	8	9	59	59	57	A
32	15	8	10	59	59	56	A
36	16	8	11	59	59	56	A

	A	B	C		PRI_MAX	load_avg
nice	0	1	2		63	3

>> 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?

Yes, there are ambiguities.

In this case I use these rules to resolve them.

First, at each timer ticks, the running thread's recent CPU will increase.

Second, at every four ticks,I use fomular (recent_cpu = (2*load_avg) / (2*load_avg + 1) * recent_cpu + nice) and fomular (priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)) to compute the priority of each thread.

Not, because in scheduler of pintos the priority and recent cpu will be recomputed by each TIMER_FREQ(100) ticks. But in this case I use 4 ticks instead of TIMER_FREQ(100) ticks.

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

The load_avg calculation needs to use the ready_list_size. Instead of looping through the ready_list inside interrupt context, we have calculated the read_list_size outside the the interrupt context so that to enhance 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?

We use a single list to implement ready queue. It is easy to implement and very convenient to use. However this design makes context switch cost more time, since every time we want to find next thread to run, we need to walk through the whole list and find the next highest priority thread, time complexity will reach O(n).

In order to improve our design, we consider to use a list array instead of a single list as out ready queue. We will have 64 element in that list array, each index of a list element indicates its priority. So when comes to context switch, we only have to go through every list once, the time complexity become O(1) now.

>> 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?

We have written the fixedpoint.c and fixedpoint.h to implement the fixed-point math. In fixedpoint.h we define a new type typedef int fix_p. This type is to store the fixed-point number. The reason we did so is to make code readable and reusable.

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?