CSE 521
PROJECT 1: THREADS
DESIGN DOCUMENT

Aditya Subramanian Muralidaran (adityasu@buffalo.edu)
Pratibha Arjun Barsale (pbarsale@buffalo.edu)
Venkata Krishnan Anantha Raman (va34@buffalo.edu)

ALARM CLOCK
==========

>> 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 list sleep_list

sleep_list : This is the double linked list that holds the threads that are currently sleeping. The list is sorted in the ascending order of time.
The first thread to wake up will be first on the list.

```
struct thread
  {
        <other variables>
        struct list_elem sleep_elem;              /* Sleep List element. */
        <other variables>
         int64_t timer_ticks;  /* Used for holding the timer_tick at which the
                                                  thread is supposed to be woke up*/
  };
```

timer_ticks : The variable timer_ticks is used to hold the time at which the thread has to be waken up. When user calls timer_sleep(x) for a thread this value will be set with current_time + x.

sleep_elem : This variable is used for adding the thread to the sleep list.

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

timer_sleep was originally implemented with busy_wait. The new approach adds the threads to sleep list and wakes it up when the time comes. Moreover we have the list in sorted order so that the first thread to wake up takes the first place in the list.

>> A3: What steps are taken to minimize the amount of time spent in
>> the timer interrupt handler?
sleep_list is based on time, The first thread to wake up will take the first slot. We can stop looping the sleep_list when we find a timer_tick (wake-up time) greater than the current time. With this we can avoid looping through the entire sleep_list.

---- SYNCHRONIZATION ----

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

The Critical section of the code is shared between threads and the interrupt handler. Adding lock will only help safeguard the variable between the threads. Since the list is shared between kernel threads and interrupt handler, this might not help. This may lead to undesirable situations, hence we protect the access the ready_list using interrupt enable/disable.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?
As discussed above the critical section is protected inside interrupt enable/disable. Race conditions are avoided using interrupt disable/enable.

---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to
>> another design you considered?
In the initial phases, It was thought that all_list could itself be used. A loop through every element and check if there is a timer expiring. This looked simple but is bad since we run a loop for all threads on the system even though two threads are sleeping.

After this point, the design evolved and it was decided to use a sorted sleep_list to keep a track of the threads that are currently sleeping on the system. Since we are storing the thread in sorted order the first element in the list is the first thread to be waken up.

The looping time is reduced considerably and the approach is better than the initial approach.

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 {
<other variables>;
int r_priority; /* This holds the running priority of the thread */
int history_priority[8]
/* This variable holds the list of priorities donated to a thread.

Struct lock *lockwaiting;
/* This variable holds the pointer to a lock for whom the thread is currently
waiting. */

<other variables>

};
```

r_priority: This is an element in the thread structure holding running priority of a thread. The threads in the ready queue are sorted based on this value. This value holds the donated priority of a thread.

history_priority[8]: This variable holds the list of priorities donated to a thread. The size indicates the level of priority donation that can be done.

***lockwaiting:** This variable holds the pointer to a lock for whom the thread is currently waiting. If the thread is not waiting for any lock, the value will be NULL.

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

For priority donation we have used r_priority int value to hold the current donated/changed priority of the thread. Also, history_priority[8] stores the list priorities donated to a thread.

Also, a pointer to a lock called *lockwaiting will keep track of the lock the thread is waiting for. Once the thread gets the lock, the value of the pointer will be set to NULL.

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

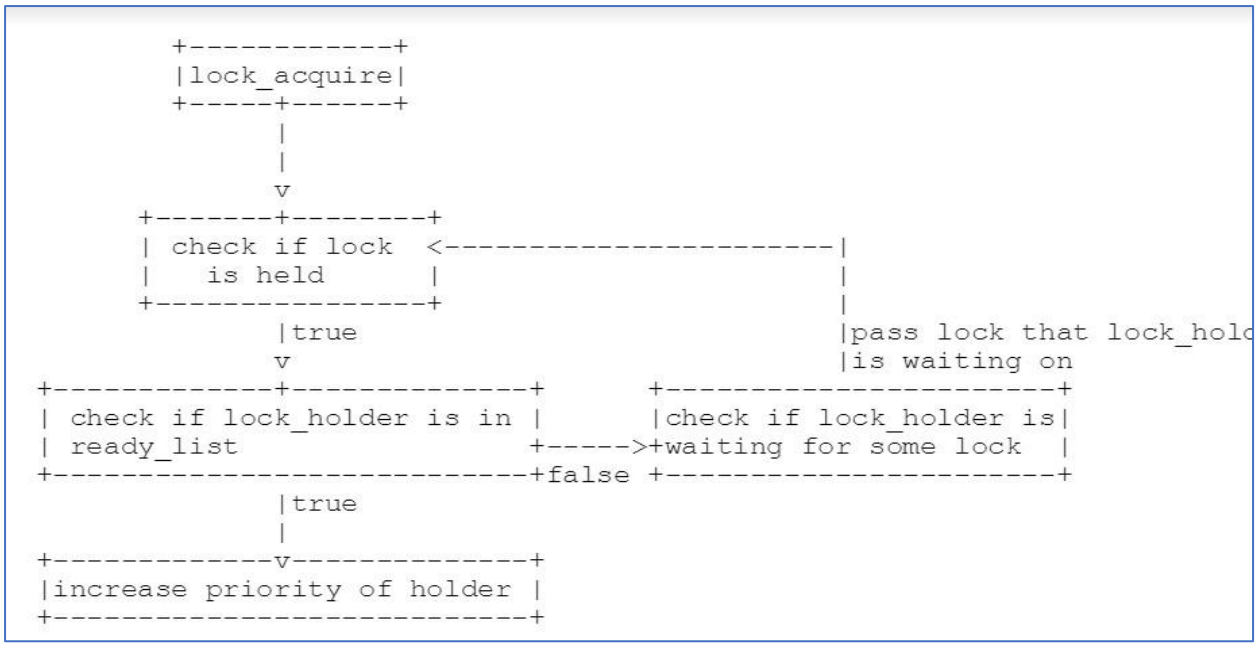

Fig. ASCII diagram for Nested Donation

---- ALGORITHMS ----

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

When a high priority thread waits on a lock/cond-var/semaphore, we have a logic in which we find who is holding the lock and if the lock is blocked who is holding that and who are all waiting. This is done iteratively till 8 levels deep and priorities are donated to low priority threads.

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

When a call to lock_acquire happens. We check if lock is taken by someone else and recursively do this to 8 levels (or) a low priority thread waiting and donate priorities.

>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.
In this scenario since a high-priority thread is waiting for the lock, it is possible that priority donation has already happened. When lock_release is called, and the donated priority is taken out, now the higher priority thread will be scheduled.

---- 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?

At this point, we don't think a race condition is going to happen with respect to priority, since each thread will itself call thread_set_priority.

We are exploring the possibilities for the same.

---- RATIONALE ----

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

This design uses thread->r_priority_list to store a history of the running priorities to take account for donations. We have also introduced a double-linked list which will get us the waiters of every lock in the system. Using this we will be able to find the owner of lock and check if it is in ready_list. If it is not present find the lock it is waiting on and repeat the same process recursively.

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.

```
Static int load_avg;
/* This variable holds the system load average */


Struct thread
{
        int recent_cpu;
        /* This denotes the CPU utilization of a particular thread*/

        int nice;
        /* This indicates the niceness value of a thread */

        <other variables>
}
```

recent_cpu: Initially for main thread, the value of this variable will be set to 0. Then, upon creation of new threads, it will have value that of the parent thread. This value will be recalculated for every thread in the all_list after every one sec.

Nice: This indicates the niceness value of a thread. A positive nice value decreases the priority of that thread and a negative nice value increases its priority. When a new nice value is set for a thread, it recalculates its threads priority.

load_avg: It is initialized to 0 at the beginning and calculated every once second for a thread.

---- 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			priority thread			Thread to run
	A	B	C	A	B	C	
0	0	0	0	63	61	59	A
4	4	0		62	61	59	A
8	8	0	0	61	61	59	A,B
12	10	2	0	60.5	60	59	A
16	14	2	0	59.5	60	59	B
20	14	6	0	59.5	59	59	A
24	18	6	0	58.5	59	59	B,C
28	18	8	2	58.5	59	58	B
32	18	12	2	58.5	58	58	A
36	22	12	2	57.5	58	58	B,C

>> 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?

At some point, the priorities were the same and we used round-robin to resolve it.

>> C4: How is the way you divided the cost of scheduling between code
>> inside and outside interrupt context likely to affect performance?
There is a cost involved in scheduling, we try to balance it out between interrupt context and kernel context

Interrupt Context:
1) Recalculation of the priority of threads happen in normal in the interrupt context.

Thread Context:
1) The niceness of a thread can be altered in runtime. This may force the priority to be recalculated and the high priority thread may get scheduled.

---- 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?

1) The current design chooses to make use of ready queue for MLFQ implementation. The ready queue is in the sorted order of decreasing thread priority. After every 4 ticks, the priority of all the threads are recalculated and the ready queue is rearranged.
2) The advantage of this design is that we are making use of the existing data structure "ready_list".
3) The disadvantage is that after every 4 tick, we need to rearrange the elements in the ready_list as per the new calculated priorities.
4) We initially tried to do with the array of 64 queues which is a better approach than the current approach because we can insert/modify/delete the elements in the list in constant time. This will also reduce the overhead involved in sorted the ready queue. If we had time, we would have completed this approach.

>> 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 arithmetic is not supported on pintos kernel. So, we store the data in Whole Number [17 Bits] + Decimal Number [14 Bits] + sign [1 Bit]. The current design is to implement functions that can be used for performing the fixed-point arithmetic. Functions are added to perform fixed point arithmetic and we use those while computing the priority.

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?
It was moderate.

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?
Yes. It opened up kernel for us.

>> 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?
There can be a wiki page on how to and hacks for pintos. Frequently hit problems and solutions kind of page.

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?
Have only contacted Sharath and he is very helpful and informative.

>> Any other comments?