2011 Fall CS330 Project 1: THREADS DESIGN DOCUMENT

---- TEAM ----

>> Team name.

#OS_A+

>> Fill in the names, email addresses and contributions of your team members.

Dauren Baitursyn<biddy.as.diddy@kaist.ac.kr> (50)

Yekaterina Abileva <k.abileva@kaist.ac.kr> (50)

contribution1 + contribution2 = 100

>> Specify how many tokens your team will use.

0

---- 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.

- http://www.cs.umd.edu/~shankar/412-Notes/11-NoBusyWaiting.html
- https://josephmate.wordpress.com/2016/02/04/how-to-avoid-busy-waiting/
- https://tssurya.wordpress.com/2014/10/25/priority-scheduling-inversion-and-donation/
- Wikipedia

2011 Fall CS330 Project 1: THREADS DESIGN DOCUMENT

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.c and thread.h

- *void thread_sleep (int64_t)* - Blocks current running thread and puts to sleep list until unblock is called on this thread.
- *void thread_tick (int64_t)* - Added an argument so that for each time tick we could check if it is time to wake up any sleeping threads (just changed declaration).
- *void thread_wake_up (int64_t)* - Checks if sleeping time of threads has expired.
- *static bool compare_time (const struct list_elem *, const struct list_elem *, void * UNUSED)* - Returns true if wake up time of thread A is less than wake up time of thread B, false otherwise.

timer.c

- *void timer_sleep (int64_t)* - Instead of busy waiting, put the thread to sleep list.
- *static void timer_interrupt (struct intr_frame *UNUSED)* - due to change in declaration of function *thread_tick(void)*, added argument to *thread_tick(int64_t).*
- *struct thread { … int64_t wakeup_time; … }* - variable to store the wake up tick of thread that called sleep.

---- ALGORITHMS ----

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

Timer sleeps first adds the ticks to the current number of ticks since OS booted. Then calls *void thread_sleep (int64_t)* with argument mentioned above, which places the current thread to sleep list, and calls *thread_block()* to change its status to *BLOCKED* and *schedule(void)* (which basically switches from current thread to the next thread from the *ready_list* and runs it). During this process, we switch off the interrupts to make the process atomically stable. Otherwise, *timer_interrupt* accesses the *sleep_list* as well, so these processes use shared resources which is dangerous (one thread might modify entity, whereas other try to read the same entity, ending up in race condition).

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

Basically, since our *sleep_list* is always being inserted items in wake up time increasing manner, the timer interrupt handler just checks the beginning of the list, and if the wake up time of the sleeping thread has not yet reached, then it just discards looking for other sleeping threads.

---- SYNCHRONIZATION ----

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

The interrupts are disabled in *timer_sleep()* at the beginning of the call to avoid race conditions.

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

The interrupts are disabled in *timer_sleep()* at the beginning of the call to avoid race conditions. The timer_sleep() function doesn't involve much time, thus disabling interrupts during call to *timer_sleep()* should not take much time. Inserting the thread into a sleep list in sorted manner could be a problem, but assuming that in Pintos not many will be put into sleep one point at time this should not be burden.

We also used the function *intr_yield_on_return()* during the timer_interrupt checking the sleep list. Because interrupts are switched off during the timer interrupt, calling the thread_yield() would cause a problem, thus we used *intr_yield_on_return()* in case sleeping thread needs to be waken up.

---- RATIONALE ----

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

It was relatively easy task to come up with this algorithms. First of all, according to design principles (including DRY), since the *list_insert_ordered(...)* function was already implemented, we thought that using this method, rather than coming up with our own sorting algorithms, to sort the sleeping threads would be a good idea. For the timer interrupt, our decision to implement the sleep list thread checking procedure was to check the sleep list every tick if any thread needs to be waken up, although it seems like a bad idea. Iterating over the whole list of sleeping threads every timer interrupt tick seemed like a burden, thus sleep list was implemented such that it is sorted in wake-up time increasing order. This way, we can avoid iterating through whole list, and can proceed with just checking the front of the list. If front thread of the sleep thread needs to be waken up, then we unblock it, and put it into ready list, and repeat the procedure above, until there the sleep list is empty or front thread's wake up time has not yet come.

2011 Fall CS330 Project 1: THREADS DESIGN DOCUMENT

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.

threads.h and thread.c:

- *struct thread { … int priority_real; … }* - Original priority independent of lock.
- *struct thread { … struct list held_locks; … }* - Locks held by a thread
- *struct thread { … struct lock *waiting_lock; … }* - Lock waiting for donation
- *void thread_unblock (struct thread *)* - Changed the insertion to the ready list so that it inserts in sorted manner.
- *void thread_yield(void)* - Changed the insertion the ready list to so that it inserts in sorted manner.
- *static void init_thread (struct thread *, const char *, int)* - added initializations to newly added members.
-

In synch.h and synch.c the following is added to the struct lock:

- *struct lock { ... list_elem elem; … }* - For tracking elements in the list of locks.
- *void sema_down(struct semaphore *sema)* - Inserting to the list according to priorities of waiters (descending order)
- *void sema_up(struct semaphore *sema)* - If waiters list is not empty, sort the list according to priorities of waiters (descending order), and then pop up the first thread, since they are sorted according to priority and unblock it. Finally call, thread_yield, in case newly unblocked thread has higher priority.
- *static void priority_donate (struct thread *t)* - Function that holds all the nested and multiple priority donation
-
- *void lock_acquire (struct lock *lock)* - changed to hold priority donation and manage the lists with held_locks and waiting_lock of thread
- *void lock_release (struct lock *lock)* - changed to hold priority donation and manage the lists with held_locks and waiting_lock of thread
- *void cond_signal* (struct condition *cond, struct lock *lock UNUSED) - sorting the list of semaphore's waiters according to the priorities in descending order added.
- *bool compare_semaphore_priorities (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)* - new function used in list_inser_ordered for sorting the waiters list of locks and semaphores according to the priorities.

>> B2: Explain the data structure used to track priority donation. Draw a diagram in a case of nested donation.

The main work is happening inside the struct thread, where the list of held_locks and lock *waiting_lock is saved. Basically, we used held_locks to store the threads that are awaiting this thread

to free the lock it has acquired. When higher priority thread tries to acquire the lock, it first checks if the thread's priority is higher than the holder's priority. If so, donation will happen and current thread will go into the holder's held lock list. It will update the priority of the holder. It checks the priorities of the threads that are waiting on the lock of the threads that are waiting on the lock of the current thread (yes, it is not a mistype), if they have higher priority, and if one does, then set the priority of the holder thread to the new one.

---- ALGORITHMS ----

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

When inserting the thread to semaphore's waiters list, we use list_insert_ordered, which ensures that the list is always sorted according to the priorities of threads, in descending order. Therefore, the front thread in the list always have the highest priority and it wakes up first.

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

1. The lock is added to the waiting lock of the thread.
2. Traverse the list with waiting_lock's holders and assign (donate) new_priority to the holder of a lock if it's higher than the holder's initial priority.
3. Repeat for the next depth level, if any. (holding nested donation)
4. Traverse the list of all held locks for the thread and check the priority of the first thread waiting for a lock. Since the list is sorted, the first one is supposed to have the highest priority. If it's higher than the previous one, update.
5. Set the highest priority found to the thread's priority
6. Set current thread as the lock's holder
7. Add the lock to the thread's held_locks list taking into account its priority (list_insert_ordered)

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

1. The lock holder is set to NULL
2. Remove the lock from the list of locks
3. Traverse the list with waiting_lock's holders and set the original lock holder's priority to the highest priority in its held_locks list (nested donation). If there was no donation, set the priority value to its original one. Else if, there was one donation, set the original holder's priority value to its initial one (priority_real)
4. Thread with the highest priority yields the processor.

---- 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?

It can happen when the priority is donated to some thread, but this thread is also trying to change its priority at the same time.

In our implementation it is solved by disabling interrupts. It could have been solved with locks, however in this case the lock should be shared for both, the donor and the thread getting donation, which is not true in our design.

---- RATIONALE ----

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

At first, a good implementation of linked list is already provided in the library, which makes it easier to implement. In addition, we were analyzing many different cases with multiple donation and nested donation, and concluded that the best way for us is to implement the lists for tracking the waiting locks and held locks. This implementation is superior, since it allows to traverse all the locks and threads which are related to the locks, and because the lists are sorted we can immediately pick the thread with the highest priority.

2011 Fall CS330 Project 1: THREADS DESIGN DOCUMENT

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.

Sorry, didn't have time for filling up this survey

>> In your opinion, was this assignment or any one of the two 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?