Report

Team contributions, Token usage (necessary)

---- TEAM ----

>> Team number

14

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

Sungmin Choi <smc9601@kaist.ac.kr> 50

Taeyoon Kim <tykimseoul@gmail.com> 50

※ contribution1 + contribution2 = 100

>> Specify how many tokens your team will use.

0

Project problems (optional)

---- 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.

https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/Pintos+Guide

We consulted the section for Project 1 in this document.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3ebd33ca-f8a9-41ce-bc3a-ca4b78882497/PintOS-Exercise-T02.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.

In `struct thread`, a member `int64_t wakeup_time;` was added. This int would be used to store what count of `timer_ticks` the parent thread should wake up at.

We also declared a `static struct list sleeping_list`, which stores all currently sleeping lists.

---- ALGORITHMS ----

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

```
void timer_sleep(int64_t ticks) {
    struct thread *current_thread = thread_current();
    enum intr_level old_level = intr_disable();
    if (!is_idle_thread(current_thread)) {
        thread_sleep(current_thread, ticks);
    }
    thread_block();
    intr_set_level(old_level);
}

...

void thread_sleep(struct thread *t, int64_t ticks) {
    int64_t now = timer_ticks();
    t->wakeup_time = now + ticks;
    list_insert_ordered(&sleeping_list, &t->elem, &compare_wakeup_time, NULL);
}

bool compare_wakeup_time(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED){
    struct thread *ta = list_entry(a, struct thread, elem);
    struct thread *tb = list_entry(b, struct thread, elem);
    return ta->wakeup_time < tb->wakeup_time;
}
```

Instead of just yielding the thread at every timer_sleep() call in a while loop, we store at what count of `timer_ticks` we need to wake the thread at and store it in `wakeup_time` of the thread. Then, we store the thread in `sleeping_thread`, ordered such that the front of the list is the thread that should wake up the earliest.

In `timer_interrupt`, the following function is called.

```
void thread_wakeup_all(void){
    struct thread *popped;
    while(!list_empty(&sleeping_list)){
        popped = list_entry(list_front(&sleeping_list), struct thread, elem);
        if(popped->wakeup_time>timer_ticks()){
            break;
        }
        list_pop_front(&sleeping_list);
        thread_unblock(popped);
```

```
    }
}
```

This method looks at the first element of `sleeping_list` and checks whether it should wake up at the current `timer_ticks`. If it should, it unblocks the thread. Otherwise, it does not do anything.

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

We thought it is inefficient to loop through `sleeping_list` at every `timer_interrupt`. So we chose to `list_insert_ordered` at the time we insert a new thread in `sleeping_list` so that at `thread_wakeup_all`, we only have to look at the front of the list.

---- SYNCHRONIZATION ----

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

Since we insert the threads in an ordered manner to the `sleeping_list`, the order of insertion stays consistent, no matter which thread calls `timer_sleep()` first.

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

We disabled interrupts during the call of `timer_sleep`.

---- RATIONALE ----

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

We did not consider any other design.

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.

The struct members shown below are added to `struct thread`. The purpose of each member is shown in comments.

```
struct thread {
    /* Owned by thread.c. */
    /* ... */
    int64_t wakeup_time;                /* time to wake up */
```

```
    struct list holding_locks;           // lock held by this thread
    struct lock *lock_to_wait;          // lock to wait for
    int initial_priority;               // to hold the first priority
    struct list donations_received;     // list of threads that donated to this thread
    struct list_elem donation_elem;     // for adding to a donation list
    bool should_lower;                  // check if priority should be lowered
    int lower_to_amount;                // amount to be lowered to
    /* ... */
};
```

A `list_elem` member is added to struct lock to facilitate addition of a lock to a list.

```
struct lock {
    struct list_elem elem;
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
};
```

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

We had a list of donations for each thread to keep track of which thread donated.

However, this method prevented us from solving the cases for nested donations easily. When the same thread donated more than once with different priorities, we ended up in an infinite loop because there were identical elements in a doubly linked list.

---- ALGORITHMS ----

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

This issue is dealt with in `sema_up` which is called by `lock_release` in `synch.c`. When unblocking a thread, the thread with the highest priority is popped from the list of waiters of the lock. Code shown below.

```
void sema_up(struct semaphore *sema) {
    /* ... */
    if (!list_empty(&sema->waiters)) {
        struct list_elem *max_elem = list_max(&sema->waiters, (list_less_func * ) &priority_ascending, NULL);
        struct thread *max_priority = list_entry(max_elem, struct thread, elem);
        list_remove(max_elem);
        thread_unblock(max_priority);
    }
    /* ... */
}
```

Similarly, the highest priority thread among the waiters of a condition variable is popped. Code shown below.

```
void cond_signal(struct condition *cond, struct lock *lock UNUSED) {
    /* ... */
    if (!list_empty(&cond->waiters)) {
        struct list_elem *max_elem = list_max(&cond->waiters, (list_less_func * ) & semaphore_priority, NULL);
```

```
        struct semaphore_elem *max_priority = list_entry(max_elem, struct semaphore_elem, elem);
        list_remove(max_elem);
        sema_up(&max_priority->semaphore);
    }
}
```

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

`lock_acquire()` calls `should_donate()` first. Should a donation happen, `donate_priority()` is called, which adds the `donation_elem` of the donor thread to the `donations_received` of the donee thread. It does not handle nested donations.

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

`revert_priority()` called by `lock_release()` checks whether the `donations_received` is empty. If not, it continues with reverting of priority in the following manner:

It loops through the threads in `donations_received` and removes the thread from the list if it is waiting for the lock that is currently being released. Then, by default, the thread's `priority` is set to the `initial priority` that it started with (or was manually set as). In the cases in which it has other threads waiting for the lock it holds, it assigns the highest priority out of those threads.

---- SYNCHRONIZATION ----

>> B6: Describe a potential race in thread_set_priority() and explain how your implementation avoids it.

We disable the interrupts during the function.

Can you use a lock to avoid this race?

Yes, by acquiring a lock to the thread that is currently being set the priority of and other threads that call that lock will wait for it to be released.

---- RATIONALE ----

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

Initially, we had a `struct lock holding_lock` member in `struct thread` for the lock this thread is holding. However, in some test cases, a single thread holds more than one lock. To handle these cases, we converted it to a list of locks.

Similarly, we had a `bool received_donation` to check whether the thread had previously received a donation. However, we realized that we not only need to keep track of multiple donations, but also wanted to have an `elem` of the donor thread for each donation. We changed it to `struct list donation_list`.

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 two problems in it, too easy or too hard? Did it take too long or too little time?

Both are too hard, both take too long of a time.

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

This assignment gave us some insight on OS design, but we would have had similar amount of insight without it.

>> 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?

The hints were helpful but we did not understand much during the presentation the TAs gave us in the office hour during which they introduced the lab because we did not have much previous knowledge about the subject.

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

It would be more helpful if there were more frequent and later office hour sessions.

>> Any other comments?

No. :)