Lab 1: Design Doc

Project 1: Threads

Preliminaries

> Fill in your name and email address.

Name: Zhang Lingxin

Student ID: 2100013018

Email: zhanglingxin@stu.pku.edu.cn

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.

```
/** This is a struct used for recording sleeping threads status.
 * If a thread falls into sleep, a sleep_entry element is created for it.
 */
struct sleep_entry{
  struct list_elem elem;
  struct thread* thread_pointer; // a pointer to the sleeping thread
  int64_t wakeup_time; // The earlist time a thread can be woken
};

/** A list to store sleep_entry */
struct list sleep_entry_list;
```

ALGORITHMS

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

1. Check if the variable `tick` is non-negative. If not, the function return.
2. Disable interrupt.
3. Create a `sleep_entry` struct and put it into a list to record relative information for later use. e.g. a pointer to the thread and the estimated wake-up time of the thread.
4. Block current thread to make it sleep.
5. Resume interrupt.

In the timer interrupt handler, `timer_interrupt()` would check every sleeping thread in `sleep_entry_list` to decide if it should be woken up and added to `ready_list`.

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

Instead of setting a member variable in the structure, `sleep_length`, and decremented it by 1 when the timer ticks to control the time a thread can be woken, I calculate the expected waking time when a thread begins to sleep is created and compare it with current `ticks` to decide whether it should be woken. This saves excessive cost of state update in timer interrupt handler.

SYNCHRONIZATION

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

When a new `sleep_entry` struct is created and inserted into the list, the interrupts are turned off to avoid race condition (e.g. two `sleep_entry` structures are being inserted into the list at the same time).

A lock in the list can be used as well, which ensures that at most one thread can access the `sleep_entry_list` at the same time.

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

`timer_sleep()` modifies the `sleep_entry_list`. There is a risk that two threads modify the list at the same time. A solution is to disable interrupts. It is feasible to use a lock to protect the `sleep_entry_list` as well.

RATIONALE

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

I've considered setting a member variable in the structure, `sleep_length`, to record total sleeping time. For every struct element in the list, the variable is decremented by 1 when the timer ticks. If the variable decreases to 0, the correspondent thread can be woken.

However, this design results in greater cost for frequent updating of `sleep_length`. The variables in every struct element in the list need to be updated whenever `timer_interrupt()` is called. Instead, I calculate the expected waking time when a struct element is created and just compare it with current `ticks`, and there is no need to update the struct elements after they are created.

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
  {
    ...
    /* For priority donation */
    int init_priority;                  /**< Initial priority before modified */
    struct lock *lock_waiting;         /**< The lock the thread is waiting for */
    struct list lock_holding_list;     /**< The locks the thread is holding */
    ...
  };
```

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

Every thread structure has the variables below:

- `priority`: the current priority, which may be modified by priority donation
- `init_priority`: the initial priority, which can't be modified by priority donation
- `lock_waiting`is the lock the thread is waiting for, NULL if none
- `lock_holding_list` is the list of locks the thread is holding, empty if none

The diagram shows an example of data structure in priority donation. L1, L2 and L3 are locks.

```
            +---------------------------------+
Thread 1:   |  lock_holding_list: NULL        |
            |      priority:      32          | ------+
            |      init_priority: 32          |       |
            |      lock_waiting:  L1          |       |
            +---------------------------------+       |
Thread 2:   |  lock_holding_list: L1          |       |
            |      priority:      32(Thread 1)| <-----+
            |      init_priority: 31          |       |
            |      lock_waiting:  L2          |       |
            +---------------------------------+       |
Thread 3:   |  lock_holding_list: L2          |       |
            |      priority:      32(Thread 1)| <-----+
            |      init_priority: 30          |       |
            |      lock_waiting:  L3          |       |
            +---------------------------------+       |
```

```
 Thread 4:   |  lock_holding_list: L3          |       |
             |      priority:      32(Thread 1)| <-----+
             |      init_priority: 0           |
             |      lock_waiting:  L4          |
             +---------------------------------+
```

ALGORITHMS

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

When a lock (semaphore, condition) is release, we traverse the `waiter` list in `sema_up()` to find the thread with the highest priority. Then we remove it from the list and unblock it.

```
sema_up (struct semaphore *sema)
{
  ...
  if (!list_empty (&sema->waiters)) {
    /* wake up the proper waiter */
    struct list_elem *prior_waiter =
              list_min(&sema->waiters, thread_compare_priority, NULL);
    list_remove(prior_waiter);
    thread_unblock(list_entry(prior_waiter, struct thread, elem));
  }
  ...
}
```

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

The sequence of events when a call to `lock_acquire()` causes a priority donation:

1. Check the necessary conditions with `ASSERT()`: (e.g. `lock != NULL`)
2. Check if the current thread is waiting for a lock.
    a. If yes, using a `while` loop to make priority donation (explained later)

    b. If not, break
3. Call `sema_down()` to acquire the lock
4. State update:
    a. Set the current thread as the holder of the lock
    b. Insert the lock into the `lock_holding_list` of the current thread

A nested donation works with a `while` loop. Suppose thread A is asking a lock holden by thread B with `lock_acquire()`, and B is waiting for a lock holden by thread C.

- Check if A's priority is greater than B's. If yes, A donates priority to B.
- Check if B is waiting for another lock. If yes, and the lock is holden by thread C whose priority is smaller than B's, B donates priority to C.
- Go on until the condition is false, or the loop reaches the maximal depth 8 (as recommended by documentation)

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

Suppose the `lock_release()` is called by thread B on a lock that thread A is waiting for.

1. Check the necessary conditions with `ASSERT()`: (e.g. `lock != NULL`)
2. Remove the lock from the holding list of thread B
3. Restore the priority of B to the larger of
    - B's `init_priority`
    - the largest priority in B's current holding list
4. Set the holder of the lock to `NULL`
5. Call `sema_up()` to release the 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?

The function can only be called by a thread to set its own priority. However, when thread A sets its own priority with the function, it is possible that thread B tries to use `lock_acquire()` to acquire a lock holden by A and therefore sets A's priority.

To avoid this, the interrupts are disabled when the priority is changed.

A lock can't be used here, for the race happens between `thread_set_priority()` and `lock_acquire()`. If a lock is used here, there may be potential deadlock(thread A holding the old lock and asking for the new priority lock in `thread_set_priority()`, while thread B holding the new priority lock and asking for the old lock/semaphore in `lock_acquire()`).

RATIONALE

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

For `lock_holding_list` in every thread structure, my implement keeps a sorted list and pops the front out when needed instead of iterating over the threads. A sorted list can reduce the cost of iteration.

I've consider writing a function for priority donation and calling it recursively in `lock_acquire()`. This would make `lock_acquire()` and priority donation processlook more clear. However, I gave it up for the excessive cost of recursion.

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.

```
/* System load average */
static int load_avg = 0;

struct thread
  {
    ...
    /* for multilevel feedback queue scheduler */
    int nice;                           /** Niceness of the thread*/
    int recent_cpu;                     /** Recent CPU usage */
    ...
  };
```

ALGORITHMS

> C2: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

- inside interrupt context:
  - update `load_avg`, `recent_CPU` and `priority` in appropriate time
  - decide the next thread to run
- outside interrupt context:

  Almost nothing is done about MLFQS.

As I put most code inside interrupt context, most cost happens here, especially when the ticks number is a multiple of `TIME_FREQ` and `TIME_SLICE`.

RATIONALE

> C3: 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?

- Advantages:

  As most status updates happen inside interrupt context, the behavior of the scheduler is easy to track and the code is relatively clear.

I sort `ready_list` according to new priority after priorities is updated in a interrupt, so the front element of the `ready_list` is just the thread with the maximal priority, which is the next thread to run.

- Disadvantages:

  `load_avg`, `recent_CPU` and `priority` are updated when the ticks number is a multiple of `TIME_FREQ` and `TIME_SLICE`. As a result, these interrupts will be much longer than other interrupts. And sorting `ready_list` after priorities is updated in a interrupt increases the overhead further.

- Improvement:

  As I mentioned above, I sort `ready_list` according to new priority after priorities is updated in a interrupt. One of my friends says that sorting results in large overhead and iterating `ready_list` for the thread with maximal priority is better.

  So there are two possible implement:

  - Sort `ready_list` according to new priority after priorities is updated in a interrupt. Pop the front element from `ready_list` when deciding the next thread to run.
  - Never sort `ready_list`, keeping it unordered.
    Iterate `ready_list` for the thread with maximal priority when deciding the next thread to run.

  I hope I could have time to try the second implement and compared their efficiency.

> C4: 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?

I use a set of static inline functions to implement fixed-point math. Here are some example:

```
/* transfer a fixed-point number into an integer (truncated) */
static inline int
ftoi_zero(int x){
    return x / f;
}

/* transfer a fixed-point number into an integer (rounded) */
static inline int
ftoi_round(int x){
    return x>=0 ? ((x+f/2)/f) : ((x-f/2)/f);
}

/* Multiply a fixed-point number by another */
static inline int
mul_ff(int x, int y){
    return ((int64_t)x) * y / f;
}
```

It is feasible to use an abstract data type or macros to implement the function. However, I chose this implement for the reasons below:

- Fixed-point math is not frequently used. In fact, I use these functions in less than 10 places in the function. I didn't see the point why I should implement it finely. ~~(In other words, I am lazy)~~
- Macros are error prone. That's what I learned from Malloclab in ICS last term.
- I am not as familiar with abstract data type in C as in C++.

So I chose to use a set of functions. That's the most labor-saving way for me :)