CS130 Project 1 Design Document

Group

- Cunhan You: youch@shanghaitech.edu.cn
- Junda Shen: shenjd@shanghaitech.edu.cn

Task 1: Alarm Lock

Data Structure

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`:

```
struct thread
{
    /* The rest ticks the thread should be waken up */
    int64_t ticks_wake;
};
```

In `timer.c`:

```
/*The sleeping processes*/
static struct list waiting_threads;
```

Algorithms

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

1. Disable interrupt.
2. Set the `ticks_wake` of the current thread and insert it into `waiting_threads` by its `ticks_wake`.
3. Block the current thread. (Will be checked and waken by the timer interrupt handler later)
4. Restore interrupt.

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

Check the `thread->ticks_wake` in the `ticks_wake` every tick.

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

Just block the current thread.

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

Disable interrupt operation when set the `ticks_wake` of the current thread.

Rationale

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

It is very simple for me to implement.

Task 2: 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`:

```
struct thread
{
    /* Base priority. */
    int base_priority;

    /* Locks that the thread is holding. */
    struct list locks;

    /* The lock blocking the thread. */
    struct lock *lock_waiting;
};
```

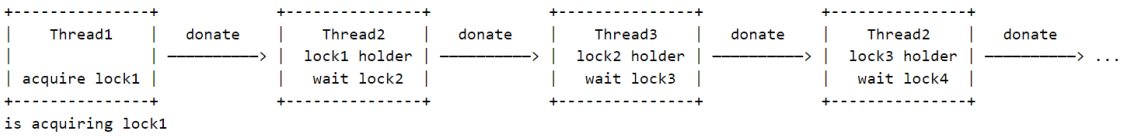
In `sync.h`:

```
struct lock
{
    /* List element for priority donation, used to construct the list of lock
later. */
    struct list_elem elem;

    /* Max priority among the threads acquiring the lock. */
    int max_priority;
};
```

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

1. When a thread acquires a lock, if the priority of the thread owning the lock is lower than itself, it will raise its priority. If the lock is still locked by other locks, it will donate the priority recursively, and then recover the priority under the logic of non donation after the thread releases the lock.
2. If a thread is donated by multiple threads, maintain the current priority as the maximum value of the donation priority —— `max_priority` (at the time of acquire and release).
3. When setting the priority of a thread, if the thread is in the donated state, set original priority —— `base_priority`. Then, if the priority set is greater than the current priority, the current priority will be changed. Otherwise, original priority will be restored when the donation status is cancelled.
4. When releasing a lock changes the priority of a lock, the remaining donated priority and the current priority should be considered.



Algorithms

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

Operate the `ready_list` as a priority queue, i.e. use `list_insert_ordered()` when inserting new threads by its priority.

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

1. Recursively donate the priority before `sema_down()` —— solve nested donation problem:

   ```
   thread_donate_priority (lock);
   ```

2. `sema_down()`:

```
sema_down (&lock->semaphore);
```

3. Disable interrupt and hold the lock:

```
enum intr_level old_level = intr_disable ();
if (!thread_mlfqs)
  thread_hold_the_lock (lock);

lock->holder = thread_current ();
```

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

1. Remove the lock from the current thread:

```
thread_remove_lock (lock);
lock->holder = NULL;
```

Details of `thread_remove_lock()`:

```
/* Disable interrupt, the remove the lock and update priority. */
void thread_remove_lock (struct lock *lock)
{
    enum intr_level old_level = intr_disable ();
    list_remove (&lock->elem);
    thread_update_priority (thread_current ());
    intr_set_level (old_level);
}
```

2. `sema_up()`:

```
sema_up (&lock->semaphore);
```

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?

Call `thread_yield()` to immediately reconsider the execution order of all threads and rearrange the execution order after setting the priority.

Rationale

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

Sorry, I can't think of another way to implement it.

Task 3: 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 `fixed-point.h`:

```
/* Make fixed_point_t stand for int */
typedef int fixed_point_t;
```

In `thread.h`:

```
struct thread
{
    /* nice value for the thread */
    int nice;

    /* estimate of the CPU time the thread has used recently */
    int64_t recent_cpu;
};

/* Global variable for system-wise load average */
fixed_point_t load_avg;
```

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:

(Assume the `TIMER_FREQ` is 100)

timer ticks	recent_cpu A	recent_cpu B	recent_cpu C	priority A	priority B	priority C	thread to run
0	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	8	4	0	61	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
28	16	8	4	59	59	58	B
32	16	24	4	59	58	58	A
36	20	12	4	58	58	58	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?

1. There's no specification whether `recent_cpu` should be updated before priority being updated or not. Our method updates `recent_cpu` before updating priority.
2. There's no specification for situations that multiple threads have the same priority. Our method chooses the thread that has the longest time not being run, which matches the behavior of our scheduler due to the implementation of `thread_cmp_priority()`.

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

The decisions like whether increase `recent_cpu`, update `load_avg` or update `priority` are made inside the interrupt context, which ensure low latency and accurate timer ticks. The actual behavior for those functions are realized outside the interrupt context, in fact, those code resides in `threads.c` and has better connection with the struct thread, but this may decrease the efficiency a bit.

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?

This design follows the specification carefully which is exactly its advantage. However, we had to admit, some algorithms implemented for this design is too simple and might not be realistic for large OS. Maybe algorithms should be improved if more time is given.

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?

The "new" type `fixed_point_t` actually comes `int` but has a clearer name, it is more readable if that type is used in which fixed point arithmetic is needed. Macros can be interpreted efficiently and they are usually highlighted by some modern IDE, thus writing them provides more convenience. Besides, using macros alleviates another C file thus the source tree is more clean.

Contributions

Cunhan You: Alarm Clock & Priority Scheduling

Junda Shen: Advanced Scheduler