Final Report

> 11811901 Fanshun

1. Four aspects of your design

1.1 ***Data structures and functions***

1.1.1 Main data structure

- **Thread**
    - `tid_t tid`
    - `enum thread_status status`
        - This is used to mark the thread status.
        - The Status has:
            - `THREAD_RUNNING`
            - `THREAD_READY`
            - `THREAD_BLOCKED`
            - `THREAD_DYING`
    - `int priority`
        - This value is the priority of this thread.
    - `struct list_elem elem`
    - `int64_t ticks_blocked`
        - This value means how many times this thread can blocked. If this value `< 0` means this thread need to be called.
    - `int base_priority`
        - This is a based priority
    - `struct list locks`
        - This list store the all locks which this thread used.
    - `struct lock *lock_waiting`
        - This means the waiting lock of this thread.
    - `int nice`
    - `fixed_t recent_cpu`
- **Lock**
    - `struct thread *holder`
        - This store the thread which hold this lock.
    - `struct semaphore semaphore`
        - This is a binary semaphore controlling access, and defined by TAs.
    - `int max_priority`
        - This is the max priority of this lock.

1.1.2 Some Modified Methods

- In Task 1, modify the `timer_sleep` Method:

```
void
timer_sleep (int64_t ticks)
{
  if (ticks <= 0)
  {
    return;
  }

  // int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  enum intr_level old_level = intr_disable ();
  struct thread *current_thread = thread_current();
  current_thread->ticks_blocked = ticks;
  thread_block();
  intr_set_level(old_level);
  // while (timer_elapsed (start) < ticks)
  //   thread_yield ();
}
```

- In order to make the queue to be a priority queue and order withe the priority, I new some comparators method in `thread.c` and `synch.c`.

```
bool
cmp_thread (const struct list_elem *a, const struct list_elem *b, void *aux
UNUSED)
{
  return list_entry(a, struct thread, elem)->priority > list_entry(b, struct
thread, elem)->priority;
}

bool
cmp_lock (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
{
  return list_entry (a, struct lock, elem)->max_priority > list_entry (b, struct
lock, elem)->max_priority;
}

bool
cmp_cond_sema_priority (const struct list_elem *a, const struct list_elem *b, void
*aux UNUSED)
{
  struct semaphore_elem *sa = list_entry (a, struct semaphore_elem, elem);
  struct semaphore_elem *sb = list_entry (b, struct semaphore_elem, elem);
  return list_entry(list_front(&sa->semaphore.waiters), struct thread, elem)-
>priority > li_entry(list_front(&sb->semaphore.waiters), struct thread, elem)-
>priority;
}
```

There are three comparators. They are used in different place. The `cmp_thread` is used to compare the thread priority; the `cmp_lock` is used to compare the lock priority; the `cmp_cond_sema_priority`is used to make the waiter list to be a priority queue.

- In Task 1, in order to check the thread block times, we new a method in `thread.c` named `blocked_thread_check` :

```
void
blocked_thread_check (struct thread *t, void *aux UNUSED)
{
  if (t->status == THREAD_BLOCKED && t->ticks_blocked > 0)
  {
    t->ticks_blocked--;
    if (t->ticks_blocked == 0)
    {
      thread_unblock(t);
      }
  }
}
```

- To solve the task `tests/threads/alarm-priority`, I modified three methods(`thread_unblock(struct thread *t)`, `init_thread(struct thread *t, const char *name, int priority)`, `thread_yield(void)`), in order to allow threads to be stored in the order of priority when they are queued.

These method all have an list insert method. I change them by add a sort after insert.

```
  list_push_back (&ready_list, &t->elem);
  list_sort(&ready_list,cmp_thread,NULL);
```

1.2 ***Algorithms***

1.2.1 Task 1

The idea of the original code is: `thread_schedule_tail` actually gets the current thread, allocates and restores the state and scene of the previous execution, and clears the resources if the current thread dies.

The schedule is actually to take a thread and switch to continue running. `thread_yield` actually throws the current thread into the ready queue, and then restarts the schedule. `timer_sleep` is within the ticks time. If the thread is in the running state, it keeps throwing it to the ready queue and not letting it execute.

In order to solve the problem of threads constantly going back and forth between the cpu ready queue and the running queue, we waste time. We directly block the thread when calling timer_sleep. Then add a member `ticks_blocked` to the thread structure to record how long this thread has been sleeping.

Then use the operating system's own clock interrupt (per tick will be executed once) to join the thread state detection, decrease `ticks_blocked` by 1 for each check, and wake up the thread if it is decreased to 0.

Later, in order to make it possible to meet priority scheduling. We ensure that this queue is a priority queue when the thread is inserted into the ready queue.

1.2.2 Task 2

After viewing all the tests, organize the tasks that Task2 needs to complete:

- When a thread acquires a lock, if the thread that owns the lock has a lower priority than itself, increase its priority. And if this lock is still locked by another lock, priority will be donated recursively, then after the thread releases the lock, the priority under the undonated logic is restored.
- If a thread is donated by multiple threads, the current priority is maintained at the maximum donation priority (at the time of acquire and release).
- When setting the priority of a thread, if the thread is in the donated state, set the original_priority, then if the set priority is greater than the current priority, the current priority is changed, otherwise the original_priority is restored when the donation status is cancelled.
- When releasing a lock changes the priority of a lock, the remaining donated priority and the current priority should be considered.
- The waiting queue of the semaphore needs to be a priority queue.
- The waiters queue of the condition needs to be a priority queue.
- If the priority changes when the lock is released, preemption can occur.

In `lock_acquire(struct lock *lock)` method. Priority donation is implemented recursively before the P operation, and then after being awakened (at this time the thread already owns the lock), it becomes the owner of the lock.After learning from an online article, the priority donation here is achieved by directly modifying the highest priority of the lock, and then updating the existing priority when calling update.

1.2.2 Task 3

The priority of the update thread is calculated for a fixed period of time in timer_interrupt. Here, the system load_avg and recent_cpu of all threads are updated every `TIMER_FREQ` time.

The thread priority is updated every 4 timer_ticks, and the recent_cpu of each timer_tick running thread is incremented by one. Although we are talking about maintaining 64 priority queue scheduling, the essence is priority scheduling.

1.3 ***Synchronization***

1.3.1 Task1

- The variable `ticks_blocked` and thread list: `ready_list` are shared here.
- Using `thread_unblock(struct thread *t)`, `init_thread(struct thread *t, const char *name, int priority)`, `thread_yield(void)`.These three function make sure the Synchronization.

1.3.2 Task2

- Shared resources include: thread locks and holders, thread lock permissions, currently locked objects, waiter queues, etc.

1.3.3 Task3

- Shared the `priority` of the thread and the `recent_cpu`.

1.4 ***Rationale***

1.4.1 Task1

- Time complexity : O(n), for the two tasks, switching between the READ queue and the RUNNING queue in the CPU is eliminated.
- Space complexity : O(n)

1.4.2 Task2

- **Advantages**: Use the method thread_update_priority () to quickly manage the priority of all threads.
- Time complexity : O(n)
- Space complexity : O(n)

1.4.3 Task3

- Time complexity : O(n), when there are N elements in the queue, all of them need to be executed once, and they only need to be executed once.
- Space complexity : O(n), when there are N elements in the queue, the previous task will not be added to the queue again, so only the space of N elements is occupied.

2. Design Document Additional Questions

```
3.1.2
```

timer ticks	R(A)	R(B)	R(C)	P(A)	P(B)	P(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

timer ticks	R(A)	R(B)	R(C)	P(A)	P(B)	P(C)	thread to run
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	C