Project 2 report

Gaoxiang Deng

300442469

First of all, in order to implement the list_insert_ordered() function, you first need to write a priority comparison function yourself. Add the following comparison function to thread.c:

```
/* ++1.2 Compare priority */
bool compare_priority(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
{
int pa = list_entry(a, struct thread, elem)->priority;
int pb = list_entry(b, struct thread, elem)->priority;
return pa > pb;
}
```

Also declare in thread.h:

```
/* +++1.2 */

bool compare_priority(const struct list_elem *, const struct list_elem *, void *);
```

Modify the scheduling statements in threads_yield(), thread_unblock(), init_thread(): Exchange from list_push_back to list_insert_ordered because need put priority of thread with descending order

Since the whole process involves the operation of locks, the function lock_aquire() that acquires the locks is modified first. That is to say, when each lock is acquired, first check whether the priority of the thread that wants to acquire the lock is greater than the max_priority stored in the lock. If it is greater, you need to set the max_priority stored in the lock to the largest. Priority, and perform a priority donation operation on the thread:

```
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread(lock));

  /** ++1.2 Priority Donate */
  struct thread *current_thread = thread_current();
  struct lock *l;
  enum intr_level old_level;
```

```
  if (lock->holder != NULL && !thread_mlfqs) {
current_thread->lock_waiting = lock;
l = lock;
while (l && current_thread->priority > l->max_priority) {
l->max_priority = current_thread->priority;
thread_donate_priority(l->holder);
l = l->holder->lock_waiting;
}
  }

  sema_down(&lock->semaphore);
  old_level = intr_disable();
  current_thread = thread_current();
  if (!thread_mlfqs) {
current_thread->lock_waiting = NULL;
lock->max_priority = current_thread->priority;
thread_hold_the_lock(lock);
  }
  lock->holder = current_thread;
  intr_set_level(old_level);
  // sema_down (&lock->semaphore);
  // lock->holder = thread_current ();
}
```

The corresponding thread_donate_priority and thread_hold_the_lock functions are implemented in thread.c.

Among them, thread_donate_priority needs to implement a function to update the priority of t by itself, because the thread whose priority is donated is not necessarily a running thread. The thread_set_priority() that comes with the program can only satisfy the priority of updating the current thread, so it is necessary to Revise.

And thread_hold_the_lock is to let the thread obtain the current lock. Since if a thread owns a lock, the priority of the thread must be the maximum value in the queue that owns the lock, so if the priority of the lock is greater than the priority of the thread, the priority of the thread needs to be updated accordingly, and then the lock is added into a queue of locks owned by the thread.

The code for these two functions is as follows:

```
/* ++1.2 Let thread hold a loc*/
void thread_hold_the_lock(struct lock *lock) {
```

```
enum intr_level old_level = intr_disable();
list_insert_ordered(&thread_current()->locks, &lock->elem, lock_cmp_priority, NULL);

if (lock->max_priority > thread_current()->priority) {
thread_current()->priority = lock->max_priority;
thread_yield();
}

intr_set_level(old_level);
}

/* ++1.2 Donate current priority to thread t. */
void thread_donate_priority(struct thread *t) {
enum intr_level old_level = intr_disable();
thread_update_priority(t);

if (t->status == THREAD_READY) {
list_remove(&t->elem);
list_insert_ordered(&ready_list, &t->elem, compare_priority, NULL);
}
intr_set_level(old_level);
}
```

Next, write the thread_update_priority() function as follows. The existing modification function thread_set_priority() can only be used to modify the currently running priority. However, this function can modify the base_priority used to update the currently running thread and determine whether a yield thread is needed according to the new priority.

Before that, improve the function thread_set_priority() that modifies the current thread priority as follows:

```
/* Sets the current thread's priority to NEW_PRIORITY. */
Void thread_set_priority (int new_priority)
{
   /* ++1.2 Handle priority */
int old_priority = thread_current()->priority;
enum intr_level old_level = intr_disable();
struct thread *current_thread = thread_current();
current_thread->base_priority = new_priority;

if (list_empty(&current_thread->locks) || new_priority > old_priority) {
current_thread->priority = new_priority;
thread_yield();
```

```
}
intr_set_level(old_level);
/* +++ 1.2 (OLD )Handle priority */
// if (new_priority < old_priority) {
// thread_yield();
   // }
}
```

Then write a thread_update_priority() function to update the priority of the currently running thread. Use this function to deal with multiple threads doing priority donations to a thread at the same time. In order to set the priority to the maximum value of the lock, it is necessary to sort the locks obtained by a certain thread, and then take the top element in the queue as the new priority of the current thread.

```
/* ++1.2 Used to update priority. */
void thread_update_priority(struct thread *t) {
enum intr_level old_level = intr_disable();
int max_priority = t->base_priority;
int lock_priority;

if (!list_empty(&t->locks)) {
list_sort(&t->locks, lock_cmp_priority, NULL);
lock_priority = list_entry(list_front(&t->locks), struct lock, elem)->max_priority;
if (lock_priority > max_priority)
max_priority = lock_priority;
}

t->priority = max_priority;
intr_set_level(old_level);
}
```

Obviously, in order to implement the sorting function about locks, the lock_cmp_priority function needs to be implemented correspondingly as follows:

```
/* ++1.2 Compare priority in locks */
bool lock_cmp_priority(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;
}
```

At the same time, add the declarations of several functions just written in thread.h:

image-20191230011352327

The above implements the logic of acquiring the lock. For the release of the lock, the lock needs to be deleted from the lock queue of the thread first, and then the lock is set to not be occupied by any thread, and finally the V operation of the semaphore is performed. Write a thread_remove_lock function here to implement:

```
/* ++1.2 Remove a lock. */
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);
}
```

This function is called when lock_release() is performed:

```
  /* ++1.2 */
  if(!thread_mlfqs){
thread_remove_lock(lock);
  }
```

Finally, modify the remaining queue to be a priority queue. First modify the cond_signal function in synch.c:

image-20191230010232328

Improve its sorting function:

```
/* ++1.2 cond sema comparison function */
bool cond_sema_cmp_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 > list_entry(list_front(&sb->semaphore.waiters), struct thread, elem)->priority;
}
```

Advanced Scheduler

Solution

First, the new algorithm requires fixed-point arithmetic that was not supported in the previous kernel. In order to implement fixed-point operations, create a new fixed_point.h in the thread directory and write the following macros:

```
#ifndef __THREAD_FIXED_POINT_H
#define __THREAD_FIXED_POINT_H
```

```
/* Basic definitions of fixed point. */
typedef int fixed_t;
/* 16 LSB used for fractional part. */
#define FP_SHIFT_AMOUNT 16
/* Convert a value to fixed-point value. */
#define FP_CONST(A) ((fixed_t)(A << FP_SHIFT_AMOUNT))
/* Add two fixed-point values. */
#define FP_ADD(A, B) (A + B)
/* Add a fixed-point value A and an int value B. */
#define FP_ADD_MIX(A, B) (A + (B << FP_SHIFT_AMOUNT))
/* Substract two fixed-point value. */
#define FP_SUB(A, B) (A - B)
/* Substract an int value B from a fixed-point value A */
#define FP_SUB_MIX(A, B) (A - (B << FP_SHIFT_AMOUNT))
/* Multiply a fixed-point value A by an int value B. */
#define FP_MULT_MIX(A, B) (A * B)
/* Divide a fixed-point value A by an int value B. */
#define FP_DIV_MIX(A, B) (A / B)
/* Multiply two fixed-point value. */
#define FP_MULT(A, B) ((fixed_t)(((int64_t)A) * B >> FP_SHIFT_AMOUNT))
/* Divide two fixed-point value. */
#define FP_DIV(A, B) ((fixed_t)((((int64_t)A) << FP_SHIFT_AMOUNT) / B))
/* Get integer part of a fixed-point value. */
#define FP_INT_PART(A) (A >> FP_SHIFT_AMOUNT)
/* Get rounded integer of a fixed-point value. */
#define FP_ROUND(A) (A >= 0 ? ((A + (1 << (FP_SHIFT_AMOUNT - 1))) >>
FP_SHIFT_AMOUNT) \
: ((A - (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT))

#endif /* thread/fixed_point.h */
```

Here, a 16-bit number (FP_SHIFT_AMOUNT) is used as the fractional part of the fixed-point number, that is, all operations need to maintain the integer part from the 17th bit.

With fixed-point operations, you can modify the original code. First, add the following new definition to the structure definition of the thread:

```
    /* ++1.3 Nice */
    int nice; /* Niceness. */
    fixed_t recent_cpu;
```

With new data members, it is also necessary to set nice and recent_cpu to zero when the init_thread() thread is initialized. Note that recent_cpu is a fixed-point number of 0. We need to define the global variable load_avg in thread.c. Note that the fixed-point type we define

by ourselves is used here. Initialized to 0 in the thread_start() function.

```
  /* ++1.3 mlfqs */
  t->nice = 0;
  t->recent_cpu = FP_CONST(0);
```

In addition to the first time, you also need to add the global variable load_avg to thread.c:

```
/** 1.3 */
fixed_t load_avg;
```

Next, the logic implementation of multi-level feedback scheduling is dealt with.

According to the experimental description, we can know that the bool variable thread_mlfqs indicates whether the advanced scheduler is enabled, and the advanced scheduler should not include the content of priority donation, so the priority donation code implemented in Mission 2 needs to use if judgment to ensure that in Priority donations are not enabled when using the advanced scheduler.

Then modify the timer_interrupt function in timer.c, and add the following code on the basis of completing the modification of task 1:

```
/* ++1.3 mlfqs **/
if (thread_mlfqs)
  {
    thread_mlfqs_increase_recent_cpu_by_one();
    if (ticks % TIMER_FREQ == 0)
      thread_mlfqs_update_load_avg_and_recent_cpu();
    else if (ticks % 4 == 0)
      thread_mlfqs_update_priority (thread_current());
  }
```

Implement thread_mlfqs_increase_recent_cpu_by_one(), thread_mlfqs_update_load_avg_and_recent_cpu(), thread_mlfqs_update_priority(thread_current()) as follows.

The first is thread_mlfqs_increase_recent_cpu_by_one(void). If the current process is not an idle process, the current process will add 1. All operations in the function use fixed-point addition.

```
/* ++1.3 mlfqs */
/* Increase recent_cpu by 1. */
void thread_mlfqs_increase_recent_cpu_by_one(void) {
ASSERT(thread_mlfqs);
ASSERT(intr_context());

struct thread *current_thread = thread_current();
```

```
if (current_thread == idle_thread)
return;
current_thread->recent_cpu = FP_ADD_MIX(current_thread->recent_cpu, 1);
}
```

Next, in the thread_mlfqs_update_load_avg_and_recent_cpu(void) function, first calculate the value of load_avg according to the size of the ready queue, and then update the recent_cpu value and priority value of all processes according to the value of load_avg.

```
/* ++1.3 Every per second to refresh load_avg and recent_cpu of all threads. */
void thread_mlfqs_update_load_avg_and_recent_cpu(void) {
ASSERT(thread_mlfqs);
ASSERT(intr_context());

size_t ready_threads = list_size(&ready_list);
if (thread_current() != idle_thread)
ready_threads++;
load_avg = FP_ADD(FP_DIV_MIX(FP_MULT_MIX(load_avg, 59), 60),
FP_DIV_MIX(FP_CONST(ready_threads), 60));

struct thread *t;
struct list_elem *e = list_begin(&all_list);
for (; e != list_end(&all_list); e = list_next(e)) {
t = list_entry(e, struct thread, allelem);
if (t != idle_thread) {
t->recent_cpu = FP_ADD_MIX(FP_MULT(FP_DIV(FP_MULT_MIX(load_avg, 2),
FP_ADD_MIX(FP_MULT_MIX(load_avg, 2), 1)), t->recent_cpu), t->nice);
thread_mlfqs_update_priority(t);
}
}
}
```

Finally, update the priority value of the current process through the thread_mlfqs_update_priority (struct thread *t) function. And make sure that the priority of each thread is between 0 (PRI_MIN) and 63 (PRI_MAX), so add a logical judgment about the upper and lower limits at the end.

```
/* ++1.3 Update priority. */
void thread_mlfqs_update_priority(struct thread *t) {
if (t == idle_thread)
return;

ASSERT(thread_mlfqs);
ASSERT(t != idle_thread);

t->priority = FP_INT_PART(FP_SUB_MIX(FP_SUB(FP_CONST(PRI_MAX),
```

```
FP_DIV_MIX(t->recent_cpu, 4)), 2 * t->nice));
t->priority = t->priority < PRI_MIN ? PRI_MIN : t->priority;
t->priority = t->priority > PRI_MAX ? PRI_MAX : t->priority;
}
```