OS Project 1 -Threads Report

11610303 黄玉安

Introduction

In this project, I done 4 task (Efficient alarm, Priority scheduler, Multi-level Feedback Queue Scheduler: MLFQS).

And understand more about threads, especially semaphore, condition variable, lock etc.

Task 1: Efficient Alarm Clock

Task 1 is to re-implement timer_sleep() so that it executes efficiently without any “busy waiting”. I implement it by

block this sleep thread when it invoke time_sleep(). Then add a member in thread to record how time this thread sleep. In every clock interrupt, OS check the state of every thread, If it is time to weak, it will weak this thread.

Modify structure of thread. Add member in thread.h

```
struct thread
  {
  // other members
  int64_t ticks_blocked; /* record how many ticks this thread sleep */
};
```

When it invoke time_sleep()，

```
void timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);

//  while (timer_elapsed (start) < ticks)
//    thread_yield ();

    if(ticks<=0)
        return;

  enum intr_level old_level = intr_disable ();

  thread_current ()->ticks_blocked = ticks;
  thread_block ();

  intr_set_level (old_level);

}
```

Change original code since it is busy waiting. The reason why disable the interrupt is to measure set ticks_block is a atomic operation (**Sychoronization**). After done this, current thread will be blocked.

Every ticket when clock interrupt take place it will check every thread, if its state is `THREAD_BLOCKED`, its ticks_blocked would decrease by 1. And if the ticks_blocked goes down to 0, it will be weak by change its state to `THREAD_READY`. It it has highest priority it will be schedule to running quickly. By doing this, during the time the thread sleep, other thread with lower priority can have chance to run.

So for every threads in all_listI will call this function in eveay clock interrupt:

```
void check_block_thread(struct thread* thr){
    if(thr->status==THRED_BLOCKED){
        if(thr->ticks_blocked>0)
            ticks_blocked --;
        if(thr->ticks_blocked==0)
            thread_unblock (thr);
    }
}
```

Until now, it has conld pass all the alarm test except for `tests/threads/alarm-priority`, this test need to implement priority schedular when the sleep thread add to ready list. Since it is content of next task, I will talk about it in next part.

Task 2: Priority Scheduler

Alarm Priority Test

In task 2, we need make the scheduler is based on priority.

By analyzing of thread's structure, I see it has a member named priority. The function `schedule` will switch threads, however, it always select the first thread in the list `ready_list`. So to implement the priority schedule, we just need to make `ready_list` is an ordered list.

By looking the source code, I see in function `thread_block()`, `thread_exit()`, `thread_yield()`, it will invock `schedule()`.

However, only `thread_yield` need to be noticed since only it insert thread to `ready list`. And we must keep the ready list is a sorted list. So change the original code in `thread_tield` `list_push_back (&ready_list, &t->elem);` to `list_insert_ordered(&ready_list, &t->elem, comp_less, NULL);`.

More, in function `thread_unblock()`,`init_thread()`, it operate the ready list by inserting a thread, so change `list_push_back` to `ist_insert_ordered` also.

Here, it need to implement compare function: Here is my code:

```
int comp_less(struct list_elem *first, struct list_elem *second, void *aux) {
    struct thread *t_f = list_entry(first, struct thread, elem);
    struct thread *t_s = list_entry(second, struct thread, elem);
    if (t_f->priority > t_s->priority)
        return 1;
    else
        return 0;
}
```

Modify `readylist` need to be protect since it is a gloable variable, that is why disable the interrupt (measure set ticks_block is a atomic operation (**Sychoronization**).

After doing this, we could have pass the `alarm-priority` test.

Other Test for Priority Schedule

However, there are many other tasks about priority schedule.

priority-change: It tell me that I need to modify the function `thread_set_priority(int pri)`, after it change its priority, it should use `thread_yield()` to measure the highest priority thread is running.

priority-sema: When thread call sema_down, if semaphore value is lower than 0, caller will put itself to semaphore's waiters and then block. This test tell us that the list waiters meed to be priority queue.

priority-condvar: It tell us we need to make condition's waiters (is a list) be a priority queue.

priority-preempt & priority-fifo: make sure highest priority thread run first, and same priority run based on round robin.

So based on analysis above, change thread_set_priority(int new_priority) in thread.c:

```
void thread_set_priority(int new_priority) {
    thread_current()->priority = new_priority;
    thread_yield();
}
```

change code of `sema_down (struct semaphore *sema)` in synch.c: change:

```
void sema_down (struct semaphore *sema) {
 // .....
 while (sema->value == 0)
    {
    list_push_back (&sema->waiters, &thread_current ()->elem);
        thread_block ();
    }

    //....
}
```

to

```
void sema_down (struct semaphore *sema) {
 // .....
 while (sema->value == 0)
    {
//make it has priority
  list_insert_ordered(&sema->waiters, &thread_current ()->elem, comp_less, NULL);
  thread_block ();
  }

    //....
}
```

When a thread called `sema_up (struct semaphore *sema)`, it finnaly need to yield cpu to let other thread whihc has highest thread weak:

```
void sema_up (struct semaphore *sema)
{
// other code
  // yield current thread to let other thread weak
  thread_yield();
}
```

change code of `cond_wait (struct condition *cond, struct lock *lock)` in synch.c:

```
void cond_wait (struct condition *cond, struct lock *lock)
{
  // other code
  // make it insert by priority
  waiter.priority = thread_current()->priority;
  list_insert_ordered(&cond->waiters, &waiter.elem, comp_sema_less, NULL);
//  list_push_back (&cond->waiters, &waiter.elem);
// other code
}
```

The function of comparing waiters' priorty is defined above:

```
/* compare function to cond wait list order insert */
int comp_sema_less(struct list_elem *first, struct list_elem *second, void *aux) {
    struct semaphore_elem *s_f = list_entry (first, struct semaphore_elem, elem);
    struct semaphore_elem *s_s = list_entry (second, struct semaphore_elem, elem);
    if (s_f->priority > s_s->priority)
        return 1;
    else
        return 0;
}
```

Now, we could pass all test in task 1 and task 2 except for test which relative to priority donation.

Priorty Donation

In task 2, we also need to implement priority donation for Pintos lock. To accept a donation priority, we need to add some member in struct thread. When a thread acquires a lock, it need to know which thread has this lock, so some additional member also need to added in struct lock.

Before we do this, first look the priority donation test:

priority-danate-one: It tell us when a thread acquires a lock, if this hold has holded by other thread, and holder's priority is lower than it, it shoud donate its priority to the lock holder.

priority-donate-multiple & priority-donate-multiple2: Thread's priority in ready list is the max value of its base priority and priority that be donated becasuse of lock it holds. Could use a list to record each thread's lock it holdes.

priority-donate-net: Threads can keep donating its priorty to other thread that holds some specially lock until the donated thread is not blocking (since just donated to one thread is not enough since the donated thread may block because of acquire another thread).

priority-donate-lower: When a thread decrease its priorty, its base priority will decrease. And it yield cpu time. However, higher priorty thread can re-donate its priority.

priority-donate-sema: When invock `sema_up (struct semaphore *sema)`, it will weak up one thread in the seam's waiters list. But the priority may be change becasuse of priorty donation during block time. So we need to order sema's waiters list again before unblock them

priority-donate-chain: After thread release the lock, it also need to restore the original priority.

Based on analysis above, we need to add some menbers to `struct thread` and `lock`:

Add members to thread:

```
struct thread
  {
    // other menbers

    int ori_pri;
    struct list locks; // record the lock this thread holds
    struct lock* want; // record the lock currently this thread want to hold
  };
```

Add members to lock:

```
/* Lock. */
struct lock
  {
    // other menbers

   // do the priority donation
    int max_pri;  // record the max priority this lock get by donating.
    struct list_elem elem;  // used by lock list in thread
};
```

initial lock;s new menber in lock_init:

```
 if(!thread_mlfqs)
    lock->max_pri = 0;
```

When thread acquire lock sucessfully, add lock to its lock list (in lock_acquire()). Use intr_disable() to measure atomic operation.

```
void
lock_acquire (struct lock *lock) {

// other code
    struct thread* t = thread_current();

    if(!thread_mlfqs && lock->holder!=NULL){
        t->want = lock;
      donate_pri(lock, thread_get_priority ());  // will implement later
    }
    sema_down(&lock->semaphore);

    enum intr_level old_level = intr_disable();
    t->want = NULL;
    lock->holder = t;
    if (!thread_mlfqs){
        //add this lock in current thread's lock list
        add_lock(lock);
    }
    intr_set_level (old_level);
```

when thread release lock (in lock_release()):

```
if(!thread_mlfqs) {
      lock->max_pri = 0; // reset the max_pri
      //remove this lock in current thread's lock list
      remove_lock(lock);
  }
```

Then modify thread.c, initial new menber of stuct thread:

```
static void init_thread(struct thread *t, const char *name, int priority) {
    // other code

    // do piroty donation
    list_init(&t->locks);
    t->ori_pri = priority;
    t->want = NULL;

    // other code
 }
```

Implement function `add_lock(struct lock* lock)` and `remove_lock(struct lock* lock)` in thread.c:

```
void add_lock(struct lock* lock){
    struct thread *t = thread_current();
    list_push_back (&t->locks, &lock->elem);

}


void remove_lock(struct lock* lock){
    struct thread *t = thread_current();
    struct list_elem *e;

    for (e = list_begin(&t->locks); e != list_end(&t->locks);
         e = list_next(e)) {
        struct lock *alock = list_entry(e, struct lock, elem);
        if (alock == lock) {
            list_remove(e);
            break;
        }
    }
    update_pri(t);
}

void update_pri(struct thread* t){

    struct list_elem *e;
    int max_pri=t->ori_pri;
    for (e = list_begin(&t->locks); e != list_end(&t->locks);
         e = list_next(e)) {
        struct lock *lockHold = list_entry(e, struct lock, elem);
        if (lockHold->max_pri > max_pri) {
            max_pri = lockHold->max_pri;
        }
    }
    t->priority = max_pri;

    if(t->status == THREAD_READY) {
        enum intr_level old_level = intr_disable();
        list_sort (&ready_list, comp_less, NULL);
//        list_insert_ordered(&ready_list, &t->elem, comp_less, NULL);
        intr_set_level(old_level);
    }

}
```

When thread find that the lock has holder, it will donate its priority to the holder thread, using function `void donate_pri(struct lock* lock, int pri)` in thread.c:

```
void donate_pri(struct lock* lock, int pri){
    enum intr_level old_level = intr_disable();

    while (lock!=NULL) {
        if (pri > lock->max_pri) {
            lock->max_pri = pri;

            ASSERT(lock->holder!=NULL);
            update_pri(lock->holder);

        } else
            break;
        lock = lock->holder -> want;
    }
    intr_set_level (old_level);

}
```

modify function thread_set_priority():

```
void thread_set_priority(int new_priority) {
    if(!thread_mlfqs) {
        struct thread *t = thread_current();
        t->ori_pri = new_priority;
        update_pri(t);
        thread_yield();
    }
}
```

When some thread called `sema_up (struct semaphore *sema)`, priority may have changed, so it must flash the order in sema's wait list (in synch.c):

```
if (!list_empty (&sema->waiters)) {

    if(!thread_mlfqs) {
      list_sort(&sema->waiters, comp_less, NULL);
    }
    thread_unblock(list_entry(list_pop_front(&sema->waiters),
    struct thread, elem));
  }
```

It is noticed that when thread operate gloable varibale (eg ready_list) or lock, It need to disable interrupt to make sure **Synchronization**. So, until now, task 2 is finished.

Task 3: Multi-level Feedback Queue Scheduler

In this part, we also need priority scheduler and using Multi-level Feedback Queue Scheduler (`mlfqs`) instend of priority donation.

Data structure

In this part, we just need to simply modify struct thread. Add member to `struct thread`:

```
struct thread
  {
    // other menbers

    // to implement mlfqs
    int nice;
    fixed_t recent_cpu;
  };
```

And add those code in thread's initial part:

```
// do mlfqs
    t->recent_cpu = FP_CONST(0);
    t->nice = 0;
```

Add a golable varibale in threa.c: `fixed_t load_avg=FP_CONST(0);`

Update recent_cpu and load_avg every 100 time_tickets (1 second), update priorty every 4 time_ticks just using the formula.

```
void thread_tick(void) {
    static fixed_t coef1 = FP_DIV_MIX( FP_CONST(59), 60);
    static fixed_t coef2 = FP_DIV_MIX( FP_CONST(1), 60);
    struct thread *t = thread_current();

    /* Update statistics. */
    if (t == idle_thread)
        idle_ticks++;
#ifdef USERPROG
        else if (t->pagedir != NULL)
          user_ticks++;
#endif
    else {
        kernel_ticks++;
        t->recent_cpu = FP_ADD_MIX(t->recent_cpu,1);
    }

    int64_t tick = timer_ticks ();
    if(thread_mlfqs && tick % TIMER_FREQ == 0){
        load_avg = FP_ADD(
            FP_DIV_MIX( FP_MULT_MIX(load_avg, 59), 60),
            FP_DIV_MIX( FP_CONST(get_ready_threads()),60)
            );
        thread_foreach(check_thread_cpu, NULL);
    }
    if(thread_mlfqs && tick % 4==0){
        thread_foreach(check_thread_pri, NULL);
        list_sort (&ready_list, comp_less, NULL);
    }

    enum intr_level old_level = intr_disable();
    thread_foreach(check_block_thread, NULL);
    intr_set_level(old_level);
    // other code
}


void check_thread_pri(struct thread *thr, void *aux) {
    fixed_t new_pri;
```

```
    new_pri = FP_SUB_MIX(
            FP_SUB( FP_CONST(PRI_MAX),  FP_DIV_MIX(thr->recent_cpu, 4)),
            thr->nice * 2
    );

    int priority = FP_INT_PART(new_pri);
    if(priority > PRI_MAX)
        priority = PRI_MAX;
    if(priority < PRI_MIN)
        priority = PRI_MIN;

    thr->priority = priority;
    ASSERT(PRI_MIN <= thr -> priority && thr->priority <= PRI_MAX );
}

void check_thread_cpu(struct thread *thr, void *aux) {
    fixed_t load_avg_times2 = FP_MULT_MIX(load_avg,2);
    fixed_t coef_cpu = FP_DIV( load_avg_times2, FP_ADD_MIX(load_avg_times2, 1));

    thr->recent_cpu = FP_ADD_MIX(
            FP_MULT_MIX( coef_cpu, FP_ROUND(thr->recent_cpu)),
            thr->nice
    );

}
```

Using function `get_ready_threads()` to get the value of `ready_threads` which helps to calculate load_avg.

```
int get_ready_threads(){
    int ready_threads;
    if(idle_thread!=NULL && thread_current()->tid==idle_thread->tid)
        ready_threads = 0;
    else
        ready_threads = 1;
    ready_threads += list_size (&ready_list);
    return ready_threads;
}
```

Then implement those function according to projec document.

```
/* Returns the current thread's priority. */
int
thread_get_priority(void) {
    return thread_current()->priority;
}

/* Sets the current thread's nice value to NICE. */
void
thread_set_nice(int nice UNUSED) {
    struct thread* t = thread_current();
    t->nice = nice;
    check_thread_pri(t, NULL);

    thread_yield();
    /* Not yet implemented. */
}

/* Returns the current thread's nice value. */
int
thread_get_nice(void) {
```

```
    /* Not yet implemented. */
    return thread_current()->nice;
}

/* Returns 100 times the system load average. */
int
thread_get_load_avg(void) {
    /* Not yet implemented. */
    return FP_ROUND( FP_MULT_MIX(load_avg, 100));
}

/* Returns 100 times the current thread's recent_cpu value. */
int
thread_get_recent_cpu(void) {
    fixed_t recent = thread_current()->recent_cpu;
    /* Not yet implemented. */
    return FP_ROUND( FP_MULT_MIX(recent, 100));
}
```

In this part, since we use fixed_point.h to do float point calculation, we must care about the overflow problem since it can only represent most 16 bit integer in this case. After doing this, I pass all the 27 test.

Task 4 Debug Pintos

Change some line of code can correct the result (in line thread.c:443).

change:

```
 t->recent_cpu = FP_ADD_MIX(
              FP_DIV(FP_MULT(FP_MULT_MIX(load_average, 2), t->recent_cpu),
FP_ADD_MIX(FP_MULT_MIX(load_average, 2), 1)),
              t->nice);
```

to

```
 t->recent_cpu = FP_ADD_MIX(
              FP_MULT(
                      FP_DIV ( FP_MULT_MIX(load_average, 2), FP_ADD_MIX (
FP_MULT_MIX(load_average, 2), 1)) ,  t->recent_cpu), t->nice);
```

My step is like these:

I guess the problem is in the process fixed_point number calculation. Since it only has 32 bit to store numbers, there may be a overflow problem. I print something and Use ASSERT to judge if it appears overflow by looking if the value of `FP_MULT(FP_MULT_MIX(load_average, 2), t->recent_cpu)` would be negative.

This is the code (in update_recent_cpu_all(void): thread.c):

```
    printf("load_avg_2_times_cpu %d\n",
       FP_MULT(FP_MULT_MIX(load_average, 2), t->recent_cpu));
  ASSERT(FP_MULT(FP_MULT_MIX(load_average, 2), t->recent_cpu) >= 0);

  t->recent_cpu = FP_ADD_MIX(
            FP_DIV(FP_MULT(FP_MULT_MIX(load_average, 2), t->recent_cpu),
FP_ADD_MIX(FP_MULT_MIX(load_average, 2), 1)),
            t->nice);

 ASSERT(t->recent_cpu >= 0);
```

This is the result, we can see it appeared overflow really.

```
load_avg_2_times_cpu 2114287515
load_avg_2_times_cpu 2114463092
load_avg_2_times_cpu 2114553752
load_avg_2_times_cpu 2114882754
load_avg_2_times_cpu 2114868396
load_avg_2_times_cpu 2103346428
load_avg_2_times_cpu 2099520130
load_avg_2_times_cpu 493829500
load_avg_2_times_cpu -2102040746
Kernel PANIC at ../../threads/thread.c:443 in update_recent_cpu_all(): asserti
on `FP_MULT(FP_MULT_MIX(load_average, 2), t->recent_cpu) >= 0' failed.
Call stack: 0xc002a246 0x725f6574Unexpected interrupt 0x0e (#PF Page-Fault Exc
eption)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
```

And it is appeared in 38 s which is close to the error occurred.

```
(mlfqs-load-60) After 30 seconds, load average=24.29.
(mlfqs-load-60) After 32 seconds, load average=25.47.
(mlfqs-load-60) After 34 seconds, load average=26.61.
(mlfqs-load-60) After 36 seconds, load average=27.71.
(mlfqs-load-60) After 38 seconds, load average=28.78.
Kernel PANIC at ../../threads/thread.c:443 in update_recent_cpu_all(): asserti
on `FP_MULT(FP_MULT_MIX(load_average, 2), t->recent_cpu) >= 0' failed.
Call stack: 0xc002a1d6 0x725f6574Unexpected interrupt 0x0e (#PF Page-Fault Exc
eption)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
```

So I changed the order of calculate (Divide first, then multiply it), Then this bug is solved.

```
s/threads/mlfqs-block.result
pass tests/threads/mlfqs-block
pass tests/threads/alarm-priority
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 10 tests passed.
```