2015년 2학기 운영체제

프로젝트 1 리포트

5조

2009-11604 정태호

2009-11779 이동우

2013-11399 박병준

2013-11431 정현진

1. Wait queue design

- Data Structures

Struct thread에 int64_t wait_start, int64_t wait_length, bool wait_flag 추가

- Algorithms

Timers.c에서 timer_sleep 함수에 계속적으로 thread_yield 함수를 호출하는 부분 삭제 및 thread_sleep함수를 호출하도록 변경. Thread_sleep함수와 sleep상태의 thread를 깨우는 wake_thread함수는 다음과 같다. 이 때, wait_list는 static struct list wait_list; 와 같이 선언되었다. Thread_init 함수에서 list_init(&wait_list); 명령어로 초기화시킨다. Wake_thread 함수는 next_thread_to_run 함수에서 초기에 호출한다.

```
void thread_sleep (int64_t start, int64_t ticks){
    struct thread *cur = thread_current ();
        enum intr_level old_level;
    ASSERT(!intr_context());
        old_level = intr_disable ();
        cur->wait_start = start;
        cur->wait_length = ticks;
        cur->wait_flag = true;
        list_insert_ordered (&wait_list, &(cur->elem), is_less_time, NULL);
        thread_block();
        intr_set_level (old_level);//return to the original interrupt level
}
```

```
void wake_thread (){
        struct thread* th;
        while(!list_empty(&wait_list)) {
                th = list_entry(list_front (&wait_list), struct thread, elem);
        if(timer_elapsed(th->wait_start) >= th->wait_length){
                list_pop_front(&wait_list);             th->wait_length = 0;
            th->wait_start = 0;             th->wait_flag = false;
            thread_unblock(th);
                }       else            break;
        }
}
```

Thread의 우선순위를 정하기 위해 두 threads의 우선순위를 비교하는 is_less_time()함수는 다음과 같다.

```
bool is_less_time (const struct list_elem* a, const struct list_elem* b, void *aux UNUSED){
        struct thread *thread_a = list_entry (a, struct thread, elem);
        struct thread *thread_b = list_entry (b, struct thread, elem);
    if ((thread_a->wait_start + thread_a->wait_length) < (thread_b->wait_start + thread_b->wait_length))
        return true;
    else if( (thread_a->wait_start + thread_a->wait_length) > (thread_b->wait_start + thread_b-wait_length))
        return false;
    else { // if two itmes have same waiting time, compare with priority.
        if(thread_a->priority > thread_b->priority)       return true;
        else        return false;
    }
}
```

2. Priority donation design

- Data Structures

Struct lock에 struct list_elem elem 멤버변수 추가, struct thread에 int old_priority, int set_priority, struct list lock_list 멤버변수 추가

- Algorithms

Thread.c의 init_thread함수에서 멤버변수 old_priority, set_priority와 lock_list를 각각 초기화한다.

```
void thread_set_priority (int new_priority) {
    struct thread *curr = thread_current();
    if(list_empty(&curr->lock_list)) { // Set priority if only there's no donated lock in current thread.
        curr->priority = new_priority;
        thread_yield(); // If priority were changed, change the running thread immediately.
    }else { // otherwise, remember set-value to wait until lock release.
        if(new_priority >= curr->priority) {
            curr->priority = new_priority;
            thread_yield();
        } else {        curr->set_priority = new_priority;            }       }       }
```

한편, wait_list와 ready_list에 thread의 elem을 넣을 때 기존 코드에서는 list_push_back으로 단순히 FCFS방식으로 실행했다면, 수정된 코드에서는 각 쓰레드의 priority를 기준으로 리스트를 정렬하도록 하였다. 이 과정에서 기본 라이브러리의 list_insert_ordered() 함수를 사용하고, 두 쓰레드의 priority를 비교하는 함수로 아래와 같은 is_higher_priority() 함수를 사용하였다. 또한 priority inversion 현상을 제거하기 위하여 synch.c파일에 다음과 같은 priority_donation(),priority_rollback() 함수를 정의하여 사용하도록 하였다.

```
bool is_higher_priority (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED) {
    struct thread *thread_a = list_entry(a, struct thread, elem);
    struct thread *thread_b = list_entry(b, struct thread, elem);
    if(thread_a->priority > thread_b->priority)            return true;
    else            return false;      }
```

```
void priority_donation(struct lock *lock) {
    struct thread *holder = lock->holder;     struct thread *curr = thread_current();
    if( holder != NULL && curr != NULL ) {
        if ( holder->priority < curr -> priority ) {
            if(holder->old_priority == -1) holder->old_priority = holder->priority; // Save priority.
            holder->priority = curr->priority; // donate
  if(!is_in_list(&holder->lock_list, &lock->elem)) list_push_front (&holder->lock_list, &lock->elem);
 // Add a lock to donated list of holder. ( holder reveices donation )
            if(holder->wait_lock != NULL) priority_donation(holder->wait_lock);
        }    }        }
```

```
void priority_rollback(struct lock *lock) {
    struct thread *curr = thread_current();
    if(is_in_list(&curr->lock_list, &lock->elem)) {
        list_remove(&lock->elem); // remove from lock_list(donated lock list) of current thread
        int highest_priority = curr->old_priority;          int origin_priority = curr->old_priority;
        struct list_elem *e;          struct list *locks = &curr->lock_list;
        for(e = list_begin(locks); e != list_end(locks); e = list_next(e)) {
            struct semaphore *sema = &list_entry(e, struct lock, elem) -> semaphore;
            struct list *waiters_list = &sema->waiters;
            struct thread *max_thread = list_entry(list_front(waiters_list), struct thread, elem);
            if(max_thread != NULL) {
              if(max_thread->priority > highest_priority) highest_priority = max_thread->priority;
            }
        }
        curr->priority = highest_priority; // priority roll-back
          if(origin_priority == highest_priority) curr->old_priority = -1;
          if(curr->set_priority != -1) {
            int tmp = curr->set_priority;                 curr->set_priority = -1;
            thread_set_priority(tmp);
        }    }        }
```

3. Testing results of ‘alarm-multiple’

PiLo hda1

Loading..........

Kernel command line: -q run alarm-multiple

Pintos booting with 4,096 kB RAM...

383 pages available in kernel pool.

383 pages available in user pool.

Calibrating timer... 204,600 loops/s.

Boot complete.

Executing 'alarm-multiple':

(alarm-multiple) begin

(alarm-multiple) Creating 5 threads to sleep 7 times each.

(alarm-multiple) Thread 0 sleeps 10 ticks each time,

(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.

(alarm-multiple) If successful, product of iteration count and

(alarm-multiple) sleep duration will appear in nondescending order.

(alarm-multiple) thread 0: duration=10, iteration=1, product=10

(alarm-multiple) thread 1: duration=20, iteration=1, product=20

(alarm-multiple) thread 0: duration=10, iteration=2, product=20

(alarm-multiple) thread 2: duration=30, iteration=1, product=30

(alarm-multiple) thread 0: duration=10, iteration=3, product=30

(alarm-multiple) thread 3: duration=40, iteration=1, product=40

(alarm-multiple) thread 1: duration=20, iteration=2, product=40

(alarm-multiple) thread 0: duration=10, iteration=4, product=40

(alarm-multiple) thread 4: duration=50, iteration=1, product=50

(alarm-multiple) thread 0: duration=10, iteration=5, product=50

(alarm-multiple) thread 2: duration=30, iteration=2, product=60

(alarm-multiple) thread 1: duration=20, iteration=3, product=60

(alarm-multiple) thread 0: duration=10, iteration=6, product=60

(alarm-multiple) thread 0: duration=10, iteration=7, product=70

(alarm-multiple) thread 3: duration=40, iteration=2, product=80

(alarm-multiple) thread 1: duration=20, iteration=4, product=80

(alarm-multiple) thread 2: duration=30, iteration=3, product=90

(alarm-multiple) thread 4: duration=50, iteration=2, product=100

(alarm-multiple) thread 1: duration=20, iteration=5, product=100

(alarm-multiple) thread 3: duration=40, iteration=3, product=120

(alarm-multiple) thread 2: duration=30, iteration=4, product=120

(alarm-multiple) thread 1: duration=20, iteration=6, product=120

(alarm-multiple) thread 1: duration=20, iteration=7, product=140

(alarm-multiple) thread 4: duration=50, iteration=3, product=150

(alarm-multiple) thread 2: duration=30, iteration=5, product=150

(alarm-multiple) thread 3: duration=40, iteration=4, product=160

(alarm-multiple) thread 2: duration=30, iteration=6, product=180

(alarm-multiple) thread 4: duration=50, iteration=4, product=200

(alarm-multiple) thread 3: duration=40, iteration=5, product=200

(alarm-multiple) thread 2: duration=30, iteration=7, product=210

(alarm-multiple) thread 3: duration=40, iteration=6, product=240

(alarm-multiple) thread 4: duration=50, iteration=5, product=250

(alarm-multiple) thread 3: duration=40, iteration=7, product=280

(alarm-multiple) thread 4: duration=50, iteration=6, product=300

(alarm-multiple) thread 4: duration=50, iteration=7, product=350

(alarm-multiple) end

Execution of 'alarm-multiple' complete.

Timer: 890 ticks

Thread: 599 idle ticks, 293 kernel ticks, 0 user ticks

Console: 2952 characters output

Keyboard: 0 keys pressed

Powering off..

4. Testing results of ‘alarm-priority’

PiLo hda1

Loading..........

Kernel command line: -q run alarm-priority

Pintos booting with 4,096 kB RAM...

383 pages available in kernel pool.

383 pages available in user pool.

Calibrating timer... 204,600 loops/s.

Boot complete.

Executing 'alarm-priority':

(alarm-priority) begin

(alarm-priority) Thread priority 30 woke up.

(alarm-priority) Thread priority 29 woke up.

(alarm-priority) Thread priority 28 woke up.

(alarm-priority) Thread priority 27 woke up.

(alarm-priority) Thread priority 26 woke up.

(alarm-priority) Thread priority 25 woke up.

(alarm-priority) Thread priority 24 woke up.

(alarm-priority) Thread priority 23 woke up.

(alarm-priority) Thread priority 22 woke up.

(alarm-priority) Thread priority 21 woke up.

(alarm-priority) end

Execution of 'alarm-priority' complete.

Timer: 588 ticks

Thread: 474 idle ticks, 116 kernel ticks, 0 user ticks

Console: 837 characters output

Keyboard: 0 keys pressed

Powering off..