OS HW Report

9조 20175097 안준호 20175104 오정석

HW 2

Implementing Alarm Clock

Problem Definition

현재 timer_sleep 내부는 busy waiting으로 구현되어 스레드가 작동중이지 않을 때에도 리소스를 많이 필요로 해 비효율적이다. 이를 깨우는 시간을 매 시간 체크하지 않고 정확히 스레드가 요구로 하는 시간에 깨우는 함수를 만들어 해결해야 한다.

Algorithm Design

먼저, 스레드를 계속체크하지 않고 깨우기 위해 sleeping_list를 만든다. 스레드가 sleeping_list에 저장되면, 해당 스레드는 작동하지 않는 상태로 간주되어 매 시간 tick을 체크하지 않는다. 스레드가 sleeping_list에 들어가 있는 동안에는 block을 시켜 더이상 실행되지 않도록 막아놓는다. 스레드를 깨워야 할 때에는 sleeping_list에 들어가 있는 스레드를 unblock 시키고 list에서 빼내야 한다.

Implementation

먼저, 스레드마다 깨어날 정보를 담는 변수 wake_time을 선언한다. 현재의 tick이 wake_time보다 커질 경우 thread는 unblock되어야 한다.

```
struct thread{
    ...
    uint64_t wake_time;     //스레드를 깨울 tick을 저장
    ...
}
```

sleeping_list를 선언하고, thread가 sleep 상태가 될 때마다 sleeping_list에 저장한다. 저장된 스레드는 block되도록 한다. 다만 현재 스레드가 idle_thread인 경우 block 시키면 안 된다. OS는 동작해야 하는 스레드가 단 한개 도 없을 경우, idle_thread를 돌려야 하기 때문이다. idle_thread가 돌지 않을 경우 동작 대기중인 스레드가 없을 때 OS는 재부팅 과정을 계속 거쳐야 하므로 비효율적이다. thread_sleep 함수 내부에서는 인터럽트가 발생하면 안 되기 때문에 intr_disable()로 미리 인터럽트를 방지한다.

```
void thread_sleep(int64_t t)
{
    struct thread* cur = thread_current();
    enum intr_level old_level;

    ASSERT(!intr_context());
```

```
    old_level = intr_disable();
    // for idle thread blocking must be ignored
    if(cur != idle_thread) {
        cur->wake_time = t;
        list_insert_ordered(&sleeping_list, &cur->elem,
thread_order_sleep, 0);
        thread_block();
    }
    intr_set_level(old_level);
}
```

일정 tick이 지난 후 sleeping_list에서 스레드를 빼고 unblock 시키는 thread_awake 함수를 만든다. 이 함수는 스레드 고유의 wake_time변수 값이 현재의 tick값보다 큰 경우 스레드를 unblock하는 역할을 한다. thread_awake 역시 인터럽트가 발생해 중단되면 안 되기 때문에 intr_disable로 처리한다.

```
void thread_awake(int64_t tick)
{
    enum intr_level old_level;

    ASSERT(intr_get_level() == INTR_OFF);

    old_level = intr_disable();

     while(!list_empty(&sleeping_list))
     {
         struct thread *t = list_entry (list_front(&sleeping_list), struct
thread, elem);
        if(t->wake_time <= tick) {
            list_pop_front(&sleeping_list);
             thread_unblock(t);
        } else {
            break;
        }
    }
    intr_set_level(old_level);
}
```

위의 함수 thread_sleep과 thread_awake는 각각 스레드가 실제로 sleep되고 awake되는 함수인 timer_sleep과 timer_interrupt 내부에서 호출되어 스레드의 block과 unblock 과정을 수행해야 한다.

```
void timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();
    ASSERT (intr_get_level () == INTR_ON);
    thread_sleep(start + ticks);
}
```

```
static void timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick ();
    thread_awake(ticks);
}
```

Implementing Priority Scheduling

Requirement 1: implementing priority scheduling

Problem Definition

현재 pintos는 priority에 대한 고려를 하지 않고 ready queue에 순서대로 저장하고 사용한다. 이 경우 스레드가 우선순위에 상관없이 실행되어 비효율적이다. 따라서 스레드의 우선순위에 따라 실행 순서가 결정되도록 ready queue에 대기시켜줄 필요성이 있다.

Algorithm Design

ready queue에 스레드를 삽입할 때 우선순위가 정렬되어 삽입되도록 수정한다. ready queue에서 현재 cpu를 점유하고 있는 스레드의 우선순위보다 높은 스레드가 존재한다면 cpu를 양보해야 한다. ready queue에 스레드가 저장되는 것은 스레드가 unblock될 때와 yield될 때이다. 따라서 unblock에서 리스트의 뒤로 스레드를 삽입하는 부분을 list_insert_ordered로 변경해 정렬되어 삽입되도록 한다.

Implementation

먼저, 스레드의 우선순위를 비교하는 함수를 만든다.

```
bool thread_order_priority(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;
}
```

이 함수를 이용해 unblock 또는 yield되어 ready queue에 들어갈 때마다 정렬되어 들어가도록 한다.

```
void thread_unblock (struct thread *t)
{
    enum intr_level old_level;
    ASSERT (is_thread (t));
    old_level = intr_disable ();
    ASSERT (t->status == THREAD_BLOCKED);
    list_insert_ordered (&ready_list, &t->elem, thread_order_priority, 0);
    t->status = THREAD_READY;
    intr_set_level (old_level);
```

```
}

void thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    ASSERT (!intr_context ());
    old_level = intr_disable ();
    if (cur != idle_thread)
        list_insert_ordered (&ready_list,
                             &cur->elem, thread_order_priority, 0);
    cur->status = THREAD_READY;
    schedule ();
    intr_set_level (old_level);
}
```

현재 cpu를 점유중인 스레드와 priority를 비교해 더 높으 스레드가 대기중이면 양보하도록 하는 thread_test_priority를 만든다. 이 함수는 thread_create와 thread_set_priority 함수 내부에서 호출되도록 한다.

```
void thread_test_priority(void) {
    enum intr_level old_level;
    old_level = intr_disable();
    struct thread* cur = thread_current();

    if(!list_empty(&ready_list) && cur->priority  <
list_entry(list_front(&ready_list),
                                                                struct
thread, elem)->priority) {
        thread_yield();
    }
    intr_set_level(old_level);
}
```

Requirement2 : implementing priority donation

Problem Definition

운영체제에서는 스레드 간 공유되는 데이터를 읽고 쓸 때 충돌을 방지하기 위해 lock이 존재한다. 따라서 데이터에 접근해 쓰는 권한은 lock을 가지고 있는 스레드에 제한된다. 하지만 우선순위가 낮은 스레드(L)가 이미 lock을 가져간 상황에서 우선순위가 높은 스레드(H)가 lock을 요청하는 경우, H는 priority scheduling의 경우와 다르게 나중에 실행될 필요가 있다. 이를 위해 H는 일시적으로 자신의 높은 우선순위를 L에게 양보해 먼저 실행되도록 해야 한다. 또한 스레드가 lock, semaphore, condvar를 점유할 때 대기열이 발생하는 문제가 있는데, 수정 이전 pintos는 이를 FIFO로 처리한다. 이 경우 우선순위에 상관 없이 lock을 점유하게 되는 문제가 있어 waiter 대기열에 추가할 때 정렬해 삽입하는 과정이 필요하다.

Algorithm Design

1. Semaphore, Lock, Condvar 스레드가 lock을 점유하려는 경우, 대기열이 발생하게 된다. 이 대기열은 waiters 리스트로 구현되어 있는데, 이 리스트에 스레드를 삽입하는 과정에서 먼저 소팅해주어 priority의 순서대로 lock을 점유하도록 만들어준다.

2. Multiple Donation 우선순위에 따라 L, M, H 스레드가 존재한다고 가정한다. 한 스레드가 두개 이상의 lock을 가지게 되면, 각 lock마다 donation이 발생 가능하다. 즉, 한 스레드에 양보되는 우선순위게 여러개 발생하게 된다. 이를 위해 양보되는 우선순의들을 저장하는 자료구조를 스레드 구조체 내에 새로 만들고, 양보되는 우선순위의 크기들을 비교해 현재의 우선순위를 결정한다. 예를 들어, 스레드 L이 lock A와 B를 보유하고 있고, M과 H가 각각 A와 B를 요청하는 경우, L은 H의 priority를 갖게 된다. 이 priority는 스레드 L이 작업을 마치고 lock B를 포기함에 따라 M의 priority로 강등되고, 최종적으로 lock A마저 포기하게 되면 자신의 priority로 돌아와야 한다.

3. Nested Donation 우선순위에 따라 L, M, H 스레드가 존재한다고 가정한다. 스레드 L이 처음에 lock A를 요청해 보유하고 있다. M이 lock B를 요청하고, 작업하는 과정에서 lock A를 요청한다. 이후 H가 lock B를 요청한면 Nested Donation의 경우에 해당한다. 시간 순서대로 보면, 처음 L이 lock A를 가지고 있고, M이 lock B를 가지고 있는 상황에서 lock A를 요청하고, H가 lock B를 요청한 상황이다. 이 경우, M이 lock A를 요청하는 상황에서 첫번째 priority donation이 발생한다. 이에 따라 M의 priority가 L에게 양보된다. 이후 H가 lock B를 요청할 때 두번째 priority donation이 발생한다. 이때 L은 H의 priority를 가지게 된다. L이 lock A를 반환하게 되면 L의 priority가 M으로 양보된다. M이 lock B를 반환하게 되면 다시 L이 자신의 priority를 회수하게 되고, H가 실행되게 된다.

Implementation

우선순위를 저장할 struct thread 내 자료구조 선언한다. 해당 스레드에 여러개의 스레드들이 priority를 양보할 경우 (multiple donation)에 대응하기 위해 donations 리스트가 존재한다.

```
struct thread{
  …
  int init_priority;                    // 스레드의 원래 priority 저장
  struct lock *wait_on_lock;            // 스레드가 대기중인 lock의 자료구조 저장
  struct list donations;                // 스레드에 양보된 priority값들 저장
  …
}
```

자신의 priority를 양보하는 함수 donate priority를 구현한다. 이 함수는 lock을 기다리는 스레드가 있을 경우 재귀적으로 호출되어 nested donation에 대응할 수 있도록 구현되어 있다.

```
void donate_priority(struct lock *lock){
    ASSERT(lock != NULL);
    ASSERT(lock->holder != NULL);

    if (thread_current ()->priority > lock->holder->priority){
        lock->holder->priority = thread_current()->priority;
        if(lock->holder->wait_on_lock != NULL){
            donate_priority(lock->holder->wait_on_lock);
        }
    }
}
```

lock_aquire에서 mlfqs가 아니고 lock의 holder가 존재할 때 priority donation이 일어나도록 설정한다. lock의 holder가 없을 때까지 priority donation은 재귀적으로 호출되며 상위 스레드의 우선순위를 lock을 가진 하위 스레드의 우선순위에 대입한다.

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

  if(thread_mlfqs == false) {
    struct thread *curr_thread = thread_current();
    if(lock->holder != NULL){
      donate_priority(lock);
    }
    curr_thread->wait_on_lock = lock;
    sema_down (&lock->semaphore);
    lock->holder = thread_current ();
    curr_thread->wait_on_lock = NULL;
    list_push_back(&curr_thread->donations, &lock->elem);
  } else {
    sema_down(&lock->semaphore);
    lock->holder = thread_current();
  }
}
```

스레드가 작업을 마치고 lock을 포기할 경우, 리스트에서 제거됨과 동시에 priority 값 역시 바꿔야 한다. lock_release에서 lock 스레드 대기열에서 release와 동시에 제거하고 priority를 초기화하도록 설정한다. priority를 초기화하는 함수를 만들어 lock_release 함수 내에서 호출하도록 설정한다.

```
int refresh_priority(void){
    int initial_priority = thread_current()->init_priority;
    struct list *list = &thread_current()->donations;
    if(list_empty(list)){
        return initial_priority;
    }

    struct list_elem *e;
    for (e = list_begin (list); e != list_end (list); e = list_next (e)){
        struct semaphore *sema = &list_entry(e, struct lock, elem)-
>semaphore;
        if (!list_empty (&sema->waiters))
        {
            int temp_priority = list_entry (list_begin (&sema->waiters),
                                        struct thread, elem)->priority;
            if (temp_priority > initial_priority)
                initial_priority = temp_priority;
        }
```

```
    }
    return initial_priority;
}
```

스레드 donation 관계에 따라 스레드의 priority는 지속적으로 변화하므로 그때마다 priority 값을 바꿔줄 함수를 구현한다. thread_set_priority와 thread_get_priority를 구현해 donation이 일어날 때마다 priority들을 비교해 현재의 priority를 갱신한다.

```
void thread_set_priority (int new_priority)
{
    if(thread_mlfqs == false) {
        struct thread *curr_thread = thread_current();
        int priority = curr_thread->priority;
        if(curr_thread->priority==curr_thread->init_priority ||
new_priority > priority){
            curr_thread->priority = new_priority;
        }
        curr_thread->init_priority = new_priority;
        if(priority > curr_thread->priority){
            thread_test_priority();
        }
    }
    else{
        thread_current ()->priority = new_priority;
        thread_test_priority();
    }
}
int thread_get_priority (void)
{
    return thread_current ()->priority;
}
```

Implementing Advanced Scheduler

Problem Definition

우선순위 알고리즘에서 더욱 발전된, 다단계 피드백 큐 스케쥴링 알고리즘을 구현해야 한다. 다단계 피드백 큐란, 프로세스(pintos 에서는 thread)들이 CPU 버스트 성격에 따라서 우선순위를 바꾸는 것이다. 어떤 프로세스가 CPU시간을 너무 많이 사용하면, 그 프로세스의 우선순위는 낮은 우선순위의 큐로 이동된다. 따라서 우리는 동적으로 우선순위가 각각의 스레드마다 배정될 수 있게 하여야 한다. CPU사용량에 따라 우선순위가 수정되고 그를 통해서 스레드를 스케쥴링 하는 것을 구현해야 한다.

Algorithm Design

1. Nice value 스레드의 우선순위는 동적으로 결정되지만, 스레드 마다 어느정도로 다른 스레드에 자신의 우선순위를 양보할 것인지에 대한 정도를 지정할 수 있다. 그 값을 Nice value라고, 하는데 nice 값이 0 이면 아무런 일도 하지 않지

만, 양수면 다른 스레드보다 우선순위가 빨리 떨어지고, 음수면 다른 스레드보다 우선순위가 천천히 떨어지게 된다. 즉 nice value로 다른 스레드들에 대한 상대적인 위치를 지정할 수 있다.

2. Priority의 계산 핀토스에는 0부터 63까지 64개의 우선순위가 지정되어 있다. niec 값이 양수이면 priority가 떨어져야 하며, 최근 CPU를 많이 차지할 수록 우선순위가 떨어져야 한다. 이를 Ageing이라 하기도 한다. 따라서 다음 공식처럼 주어진다면, priority를 계산 할 수 있다. 공식에서 알 수 있다시피, 최근 cpu를 많이 사용할 수록 우선순위가 낮아지기 때문에 기아(starvation)또한 예방 할 수 있다.

$$priority = PRI_MAX - (recent_cpu/4) - (nice * 2)$$

3. CPU 사용량의 계산 CPU사용량을 바로직전 사용한 CPU의 총량으로 해도 되지만, 최근에 사용한 CPU의 점유율 일 수록 더욱 많은 가중치를 줌으로써, CPU의 점유율을 표현할 수 있다. 핀토스에서는 채점의 용의함을 위해서, recent_cpu의 계산은 4틱당 한번씩 하도록 규정하고 있다. $$recent_ cpu=(2*load_ avg)/(2*load_ avg + 1)*recent_ cpu + nice$$
4. Load Average 의 계산 Load Average또한 ready queue에서 동작을 기다리는 큐들의 평균으로 구현 할 수 있다. Load Average의 평균또한 최근에 사용한 값일 수록 더욱 많은 가중치를 부여받게 된다. 핀토스에서는 채점의 용의함을 위해서, load average의 계산은 타이머의 주기마다 한번씩 실행하도록 규정하고 있다.

$$load_avg = (59/60) * load_avg + (1/60) * ready_threads$$

5. 실수의 계산 운영체제에서는 속도상의 문제와 구현의 복잡성때문에 실수의 계산은 커널상에서 제공하지 않는다. 따라서 실수의 계산은 직접 구현해야만 한다. 그러나 pintos에서는 실수의 계산에 대한 가이드라인을 제공하고 있기에, 그에따라 만들면 별 문제 없이 처리 할 수 있다.

Implementation

실수의 계산은 pintos 에서 제공하지 않는다. 따라서 직접 구현해야 한다. 따로 헤더파일과 소스파일을 만드는 방법도 있지만, 구현의 편의성을 위해서 thread.c에 직접 전처리기를 이용하여 정의하였다.

```
#define F 16384
#define REAL_TO_INT(x) ((x) / F)
#define INT_TO_REAL(n) ((n) * F)
#define REAL_TO_INT_ROUND(x) ((x) >= 0 ? (((x) + F/2))/F : (((x) -
F/2))/F)
#define ADD_REAL(x,y) ((x) + (y))
#define SUBTRACT_REAL(x,y) ((x) - (y))
#define ADD_REAL_INT(x,n) ((x) + (n)*F)
#define SUBTRACT_REAL_INT(x,n) ((x) - (n)*F)
#define MUL_REAL(x,y) (((int64_t) (x)) * (y) /F)
#define MUL_REAL_INT(x,n) ((x) * (n))
#define DIV_REAL(x,y) (((int64_t) (x)) * F / (y))
#define DIV_REAL_INT(x,n) ((x)/n)
```

nice, recent_cpu 그리고 load_avg는 각각 초기값이 정의되어야 한다. 모두 0으로 초기화 한다. 그러나 load_avg는 모든 스레드가 공유하는 것이기 때문에, thread_start 에서 정의하고, nice와 recent_cpu는 스레드가 처음 만들어 질때 혹은 스레드가 부모 스레드를 상속받을 때 각각 초기화 되어야 한다. 만약 부모 스레드를 상속 받을 때 초기화 된다면, 부모 스레드의 nice와 recent_cpu를 가져오도록 구현해야 한다.

```
void thread_init (void)
{
    ...

    initial_thread->nice = DEFAULT_NICE;
    initial_thread->recent_cpu = DEFAULT_RECENT_CPU;
}

void thread_start (void)
{
    ...

    load_average = DEAFULT_AVG_LOAD;

    ...
}

init_thread (struct thread *t, const char *name, int priority)
{
    ...

    t->nice = running_thread()->nice;
    t->recent_cpu = running_thread()->recent_cpu;

    ...
}
```

Nice 값은 다음과 같이 set 되고 get 된다. nice 값이 바뀌면 필연적으로 priority도 바뀌기 때문에, nice 값이 변경되는 것과 동시에 priority를 업데이트 해주어야 한다.

```
void thread_set_nice (int nice)
{
    intr_disable();
    struct thread* t = thread_current();
    t->nice = nice;
    if(idle_thread != t) {
        t->priority = PRI_MAX - REAL_TO_INT_ROUND((t->recent_cpu/4)) - t-
>nice * 2;
    }

    if(t->priority > PRI_MAX) {
        t->priority = PRI_MAX;
    } else if(t->priority < PRI_MIN) {
        t->priority = PRI_MIN;
    }
    if(!list_empty(&ready_list) && thread_current()->priority  <
list_entry(list_front(&ready_list), struct thread, elem)->priority) {
        thread_yield();
```

```
    }
    intr_enable();
}

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

스레드들을 적절히 동적으로 스케줄링 하기 위해서는 recent_cpu, load_avg 그리고 priority를 적절한 틱당 업데이트 해주어야 한다. 그들 각각의 설명은 위에 있는 대로 구현하면 된다. 여기서 주의해야 할점은, priority를 clamp 하는 것과 채점상의 문제로 인하여 각각의 함수를 호출하는 것을 pintos document에 나와있는 대로 정확히 서술하여야 한다는 점이다. 또한 수식에 있어서 실수와 정수의 확실한 구분과 정확한 실수 계산 함수의 구현을 하여야 한다.

```
void mlfqs_priority(struct thread *t) {
    if(idle_thread != t) {
        t->priority = PRI_MAX - REAL_TO_INT_ROUND((t->recent_cpu/4)) - t-
>nice * 2;
    }

    if(t->priority > PRI_MAX) {
        t->priority = PRI_MAX;
    } else if(t->priority < PRI_MIN) {
        t->priority = PRI_MIN;
    }
    if(!list_empty(&ready_list) && thread_current()->priority  <
list_entry(list_front(&ready_list), struct thread, elem)->priority) {
        intr_yield_on_return();
    }
}

void mlfqs_recent_cpu(struct thread *t) {
    if(idle_thread != t) {
        t->recent_cpu =
ADD_REAL_INT(MUL_REAL(DIV_REAL(MUL_REAL_INT(load_average , 2),
ADD_REAL_INT(MUL_REAL_INT(load_average, 2),1)), t->recent_cpu), t->nice);
    }
}

void mlfqs_load_avg(void) {
    int ready_thread_size = (int)list_size(&ready_list) +
(thread_current() != idle_thread ? 1 : 0);
    load_average = ADD_REAL(MUL_REAL(DIV_REAL(59,60), load_average),
MUL_REAL_INT(DIV_REAL(1,60), ready_thread_size));
}

void mlfqs_increment(void) {
    if(idle_thread != thread_current()) {
```

```
        thread_current()->recent_cpu = ADD_REAL_INT(thread_current()-
>recent_cpu, 1);
    }
}

void mlfqs_recalc_cpu_priority(void) {
    struct list_elem* e;

    ASSERT (intr_get_level () == INTR_OFF);

    for(e = list_begin(&all_list); e != list_end(&all_list); e =
list_next(e)) {
        //printf("0x%x\n", e);
        struct thread *t = list_entry(e, struct thread, allelem);
        mlfqs_recent_cpu(t);
        mlfqs_priority(t);
    }
}
```

마지막으로, alarm 인터럽트가 발생할 때마다, 적절한 순간에 위에 서술한 함수를 호출하여 recent_cpu, load_avg 그리고 priority를 업데이트 해야한다. 이때 틱당 현재 실행되는 스레드의 recent_cpu값이 1씩 증가하고 4틱당 현재 스레드의 priority를 계산하여 기존에 계산해둔 priority들과 비교후 문맥교환한다. 마지막으로, 1초 (Imter_FREQ)가 지날때 마다, load_avg 그리고 전체 스레드의 cpu_recent와 priority를 다시 계산하여 전체적인 Advanced Scheduler 를 구현하게 된다. 주의해야 할 점은 recent_cpu가 틱당 1씩 증가하는 것과 TIMER_FREQ마다 다시 계산되는 것을 분리하는 것이었다.

```
static void timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_awake(ticks);

    if(thread_mlfqs == true) {
        mlfqs_increment();
        if(timer_ticks() % TIMER_FREQ == 0) {
        mlfqs_load_avg();
        mlfqs_recalc_cpu_priority();
        }
        if(timer_ticks() % 4 == 0) {
        mlfqs_priority(thread_current());
        }
    }

    thread_tick ();
}
```