Pintos 프로젝트 1. Pintos Thread

(설계 프로젝트 수행 결과)

과목명 : [CSE4070-01] 운영체제
담당교수 : 서강대학교 컴퓨터공학과 박성용

조원 : 83조 권명준, 전해성
개발기간 : 2017. 11. 11. - 2017. 11. 28.

최 종 보 고 서

프로젝트 제목: Pintos 프로젝트 1. Pintos Thread
제출일: 2017. 11. 28.
참여조원: 83조 권명준, 전해성

I. 개발 목표

Priority Scheduler가 어떤 식으로 동작하는지 이해한다. BSD Scheduler는 어떻게 우선순위를 스스로 결정할 수 있으며 어떻게 개발자가 높은 우선순위를 간접적으로 부여해 줄 수 있는지(nice 사용) 이해한다. Thread sleep이 어떻게 CPU 소모를 최소화하며 이루어질 수 있는지 이해한다.

II. 개발 범위 및 내용

가. 개발 범위

Busy waiting으로 구현되어 있는 timer_sleep()을 개선하여 thread가 CPU를 소모하지 않고 대기할 수 있도록 구현한다. 최소한의 기능을 하는 thread scheduler를 개선하여 Priority에 따라 scheduling하도록 thread 관련 함수들을 수정하고 필요한 함수를 새로 만든다. 마지막으로 thread가 최근 CPU 점유 시간과 주어진 nice 값을 토대로 우선순위를 스스로 결정하는 BSD scheduler를 만든다.

나. 개발 내용

1. Alarm Clock

timer.h : struct list * sleep_list를 선언한다. CPU 점유를 하지 않고 있는 대기 상태의 threads를 저장해 놓는 곳이다.

timer.c : void timer_sleep(int64_t ticks), void sleep_list_wakeup(), sleep_list 관련 init, destroy, compare 함수들을 만든다. 더 이상 busy waiting을 하는 방식이 아니라 sleep_list에 넣고 대기할 수 있도록 한다.

2. Priority Scheduler

thread.c : thread_aging()을 통해 500 tick마다 모든 ready_list에 있는 thread의 priority를 1 증가시킴으로써 starvation problem을 해결한다.

static struct thread * next_thread_to_run()에서 ready_list를 priority 순으로 정렬하고 최고 우선순위의 thread를 뽑아 반환하도록 해서 Priority Scheduling이 이루어지도록 한다.

thread_set_priority()를 통해 개발자가 필요에 따라 우선순위를 설정할 수 있도록 구현하였다. thread_get_priority()는 단순히 이를 다시 반환해준다.

3. BSD Scheduler

thread.c : BSD Scheduler에서는 개발자가 임의로 priority를 직접 설정할 수 없고 nice 값을 통해서 간접적으로 우선순위에 영향을 줄 수 있다. nice 값과 최근에 얼마나 CPU를 차지했는지 나타내는 지표인 recent_cpu를 통해 자동으로 priority룰 계산하도록 구현한다. recent_cpu를 계산하기 위해서는 평균적으로 몇 개의 thread가 ready_list에 있었는지를 알려주는 지표인 load_avg를 도입한다.

이를 구현하기 위해 thread_set_nice()를 만들고 또한 init_thread()에서 새 thread가 만들어질 때 부모의 nice와 recent_cpu를 상속하도록 한다. 또한 update_recent_cpu(), update_load_avg(), update_priority()를 통해 주기적으로 이 정보를 업데이트할 수 있도록 한다.

실수 연산을 지원하지 않는 pintos에서 이런 정보들을 계산하기 위한 함수 mult_FPs()등 연산 관련 함수를 구현한다.

timer.c : static void timer_interrupt()에서 1초에 한번 update_load_avg()와 update_recent_cpu()를 호출하고 4 ticks 마다 한번 update_priority()를 호출한다.

III. 추진 일정 및 개발 방법

가. 추진 일정

- 11월 24일 : 개발 내용 파악, Alarm-clock 구현
- 11월 25일 ~ 26일 : Priority Scheduler, BSD Scheduler 구현
- 11월 27일 ~ 28일 : 오류 해결, 보고서 작성

나. 개발 방법

pintos manual의 요구사항에서 무엇이 빠졌는지를 파악한다. priority donate가 빠졌다. 이 부분을 제외하고 작업을 나눈다. Alarm clock을 어떻게 개선할 수 있는지 생각해 보고 가장 먼저 이를 구현한다. 이후 어떻게 효율적으로 scheduling할 수 있을지 의논하고 Priority Scheduler와 BSD를 개발한다. 마지막으로 통과하지 못하는 tests 코드를 분석하여 오류를 수정한다.

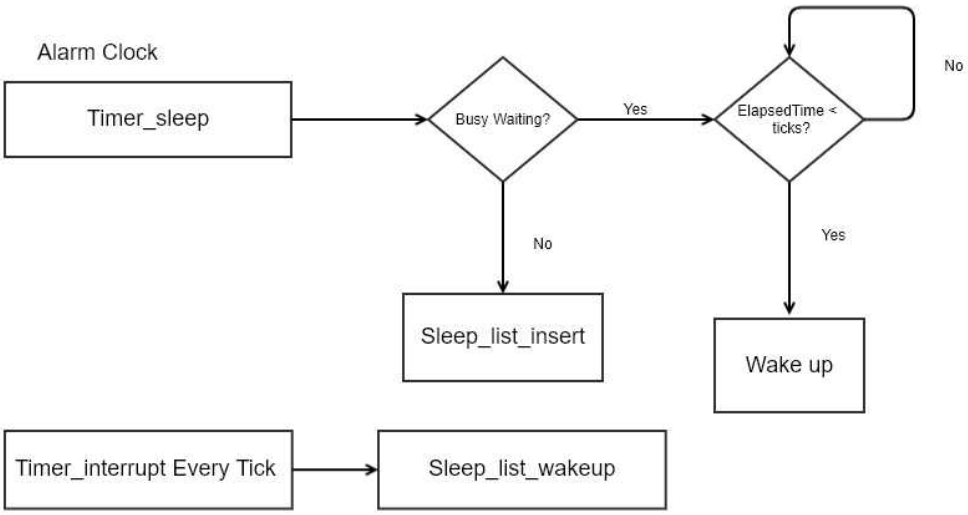
다. 연구원 역할 분담

- 권명준 : Priority Scheduler, BSD Scheduler
- 전해성 : Alarm Clock, BSD Scheduler

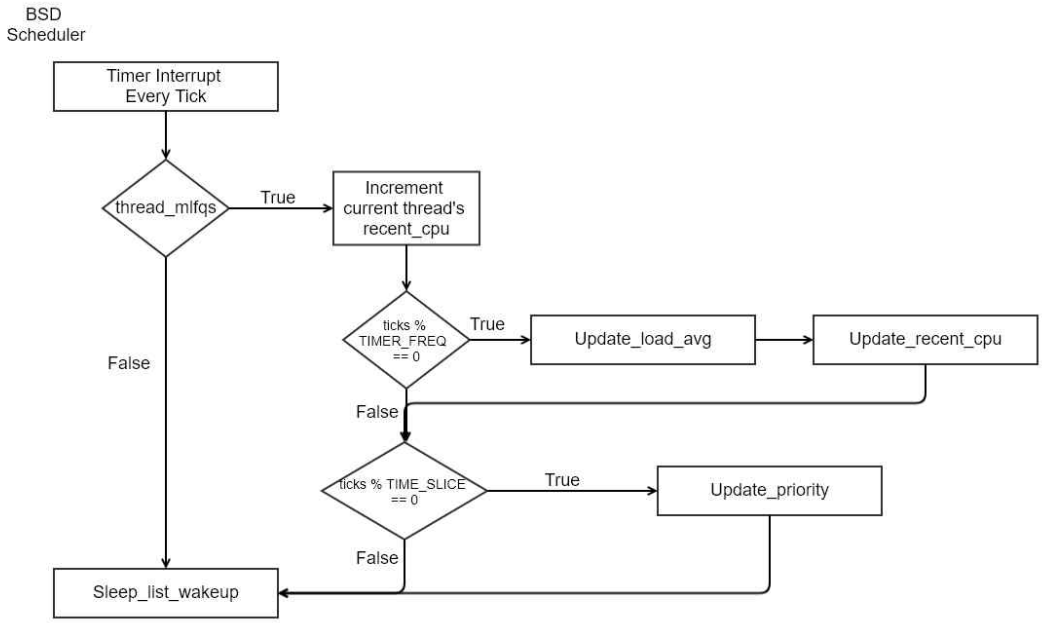
IV. 연구 결과

1. 합성 내용

1-1. Alarm Clock 구현



1-2. BSD Scheduler

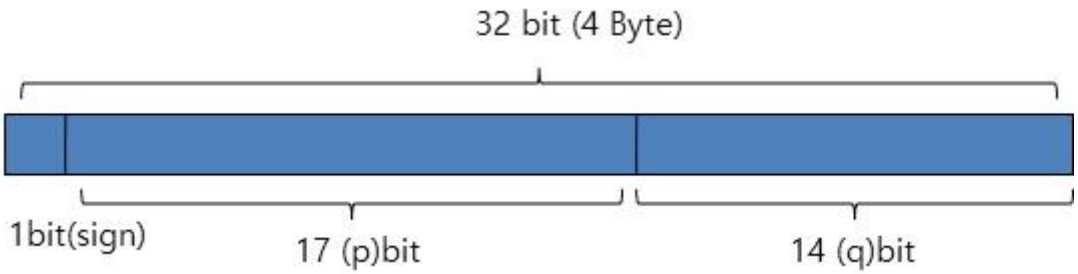


2. 제작 내용

2-1. 자료구조

Fixed Point (가상의 자료구조)

Pintos는 overhead가 큰 부동소수점 연산을 지원하지 않는다. 그러나 BSD Scheduler를 구현하기 위해서는 recent_cpu, load_avg, priority를 수시로 계산해 주어야 하는데 이 계산 과정 그리고 recent_cpu와 load_avg값 자체에서 실수가 사용된다. 따라서 실수를 표현하기 위한 자료구조가 필요하다. 이를 위해 pintos manual에 나와 있듯이 Fixed Point real number를 사용하기로 하였다.



위 그림과 같이 4byte를 사용하기로 하였다. 따라서 int형을 사용하되, 실제 의미하는 값은 int로 읽었을 때의 값에 2^{14}을 나눈 값으로 해석하도록 한다. 역으로 어떤 정수를 FP 타입으로 저장할 때는 2^{14}을 곱한 값으로 저장하도록 한다. 이를 위해서 typedef int FP;를 사용해도 되지만 편의상 이를 생략하고 int를 직접 사용하고 문맥에 따라 읽도록 하였다. 또한 이 타입의 연산을 빨리하기 위해서 thread.c에 int_to_FP, FP_to_int, round_FP, add_FPs, mult_FPs, div_FPs 등의 함수를 추가하여 편리하게 형 변환과 사칙연산 및 반올림이 이루어질 수 있도록 하였다.

struct thread (threads/thread.h)

```
struct thread
 {
/* Owned by thread.c. */
tid_t tid; /* Thread identifier. */
enum thread_status status; /* Thread state. */
char name[16]; /* Name (for debugging purposes). */
uint8_t *stack; /* Saved stack pointer. */
int priority; /* Priority. */
struct list_elem allelem; /* List element for all threads list. */

/* Shared between thread.c and synch.c. */
struct list_elem elem; /* List element. */

#ifdef USERPROG
/* Owned by userprog/process.c. */
uint32_t *pagedir; /* Page directory. */
#endif

/* Owned by thread.c. */
```

```
/* proj 2-1 */
struct thread* parent; /* identifier of parent */
tid_t cur_child; /* tid of current waiting child */
bool child_load_successful;
int child_status; /* exit status of child */
struct semaphore sema;
struct list child_list;

/* proj 2-2 */
struct list file_list; /* current thread open file list */
struct file* cur_file; //KMJ

/* proj 3 threads */
int nice;     /* have value -20 to 20 */
int recent_cpu;
unsigned magic; /* Detects stack overflow. MUST BE AT THE BOTTOM - KMJ */
 };
```

이 프로젝트를 위해서 struct thread에 nice와 recent_cpu 2개의 변수를 추가하였다. 주로 사용한 변수는 이 둘에 더하여 처음부터 정의되었던 priority 변수까지 밑줄 친 세 개다. 여기서 모두 int 타입이지만 recent_cpu는 사실 Fixed Point Real 변수이고 FP끼리의 연산은 상기한 함수를 사용하여 처리하였다.

struct list * sleep_list (devices/timer.c)

sleeping thread를 위한 자료구조이다. 어떤 thread가 sleep 상태로 바뀔 때, 이 list에 추가해놓는다. 이를 원활히 처리하기 위해 sleep_list_init, sleep_list_destroy, sleep_list_less_func, sleep_list_insert, sleep_list_wakeup 함수를 만들어 편리하게 list 연산을 처리할 수 있도록 하였다.

static struct list ready_list (threads/thread.c)

Priority Scheduler와 BSD scheduler 모두 단일 list를 사용하여 구현하였다. 두 경우 모두 priority 순으로 정렬되어 관리가 이루어진다. BSD의 경우 timer interrupt에 따라 주기적으로 update_recent_cpu(), update_load_avg(), update_priority()가 호출되고 이는 nice값과 최근 CPU 점유 시간 등에 따라 자동으로 우선순위를 변경하고 priority가 변경될 때마다 이 list는 정렬된다. 이를 통해 16개의 list를 사용하지 않고도 BSD Scheduler를 구현할 수 있었다.

2-2. 알고리즘

2-2-1 Alarm Clock

Busy Waiting으로 구현되어 있는 Alarm Clock을 좀 더 효율적으로 구현하기 위하여 만든 구조를 나타내 주는 순서도이다. 현재 BLOCK상태인 스레드들을 관리하기 위해 새로운 자료구조가 필요하였고, 그를 위해 만든 자료구조가 sleep_list이다. 이 리스트는 BLOCK 상태인 스레드들을 저장하고 있으며 각 스레드들의 wake_up_time에 따라 정렬되어 저장한다. timer_sleep 함수에서 인자로 받은 ticks를 sleep_list_insert에 인자로 넘겨준다. sleep_list_insert에서는 현재 스레드를 인자로 받은 wakeup_time 순서에 맞추어 sleep_list에 넣어주고, 현재 스레드를 ready_list에서 빼준 뒤 BLOCK상태로 만들어준다. 그 다음 새로 scheduling하는 함수를 호출하고 끝난다.

BLOCK된 스레드들을 다시 깨워주는 것은 매 tick마다 호출되는 timer_interrupt에서 처리한다. timer_interrupt에서 sleep_list_wakeup을 호출한다. sleep_list_wakeup은 sleep_list에서 wake_up_time이 현재 tick보다 작은 애들을 전부 꺼내서 ready_list에 넣어주는 역할을 한다.

```
316 /* Wake up threads in sleep list when timer intrrupt occurs */
317 void sleep_list_wakeup() {
318     if(!sleep_list) return ;
319     while(!list_empty(sleep_list)) {
320         struct slept *temp = list_entry(list_front(sleep_list),struct slept, elem);
321         if(temp->wakeup <= timer_ticks()) {
322             list_remove(&(temp->elem));
323             thread_unblock(temp->hold);
324         }
325         else
326             break;
327     }
328 }
```

```
 88 void
 89 timer_sleep (int64_t ticks)
 90 {
 91   int64_t start = timer_ticks ();
 92
 93   ASSERT (intr_get_level () == INTR_ON);
 94   // Added By Jeon Hae Seong
 95   // busy waiting (previous method)
 96   if(!sleep_list) {
 97       while (timer_elapsed (start) < ticks)
 98         thread_yield ();
 99   }
100   // insert threads into sleep_list ordered
101   else {
102       enum intr_level old_level = intr_disable();
103       sleep_list_insert(start+ticks);
104       intr_set_level(old_level);
105   }
106 }
```

```
179 static void
180 timer_interrupt (struct intr_frame *args UNUSED)
181 {
182     enum intr_level old_level;
183   ticks++;
184   thread_tick();
185
186   old_level = intr_disable();
187   //KMJ start
188   if(thread_mlfqs)
189   {
190       struct thread* cur = thread_current();
191       cur->recent_cpu = cur->recent_cpu + int_to_FP(1);
192       if(ticks % TIMER_FREQ == 0){
193           update_load_avg();
194           update_recent_cpu();
195       }
196       if(ticks % 4 == 0) update_priority();
197   }
198   intr_set_level(old_level);
199   // Added By Jeon Hae Seong
200   old_level = intr_disable();
201   sleep_list_wakeup();
202   intr_set_level(old_level);
203 }
```

```
305 /* argument means wakeup time */
306 void sleep_list_insert(int64_t wakeup) {
307     struct slept *temp;
308     temp = (struct slept*)malloc(sizeof(struct slept));
309     temp->wakeup = wakeup;
310     temp->hold = thread_current();
311     list_insert_ordered(sleep_list,&(temp->elem),sleep_list_less_func,NULL);
312     thread_block();
313 }
314
315
```

2-2-2 Prirority Scheduler 및 aging

mlfqs 옵션이 없을 때 기본 적으로 수행이 되는 scheduler이다. ready_list에 존재하는 스레드들의 수행 순서를 바꾸게 되는 함수들은 크게 thread_yield와 thread_unblock이 있다. 따라서 스레드들의 priority에 맞춰 수행 순서를 정하기 위하여 이 함수 내에서 scheduling 할 때마다 ready_list에 있는 threads의 priority에 맞춰 non-increasing order로 정렬하였다.

```
567 static struct thread *
568 next_thread_to_run (void)
569 {
570   if (list_empty (&ready_list))
571     return idle_thread;
572   else {
573       list_sort(&ready_list,priority_compare_function,NULL);
574     return list_entry (list_pop_front (&ready_list), struct thread, elem);
575   }
576 }
```

```
287 void
288 thread_unblock (struct thread *t)
289 {
290   enum intr_level old_level;
291
292   ASSERT (is_thread (t));
293
294   old_level = intr_disable ();
295   ASSERT (t->status == THREAD_BLOCKED);
296   list_push_back (&ready_list, &t->elem);
297   list_sort(&ready_list,priority_compare_function,NULL);
298   t->status = THREAD_READY;
299   intr_set_level (old_level);
300 }
301
```

aging과 관련해서는 timer_interrupt가 발생할 때마다 호출되는 thread_tick이라는 함수 내에서 aging옵션이 있다면 thread_aging이라는 함수를 호출하여 수행되도록 구현하였다. aging_ticks라는 변수를 사용하여 매 500 tick마다 ready_list의 모든 스레드의 priority를 1씩 늘리는 것으로 구현하였다.

```
764 #ifndef USERPROG
765 // aging occurs every 500 ticks
766 void thread_aging() {
767     aging_ticks++;
768     if(aging_ticks >= 500) {
769         struct list_elem *e;
770         for(e=list_begin(&ready_list);e!=list_end(&ready_list);e=list_next(e)) {
771             struct thread *t = list_entry(e,struct thread,elem);
772             t->priority++;
773             if(t->priority > PRI_MAX) t->priority = PRI_MAX;
774         }
775         aging_ticks = 0;
776     }
777 }
778 #endif
```

2-2-3 BSD Scheduler

pintos를 실행할 때, mlfqs 옵션이 들어오게 되면 thread_mlfqs를 true로 설정하여 기존 Scheduler 대신에 BSD Scheduler를 사용하도록 한다. timer_interrupt가 호출될 때마다 하는 일은 위에서 다루었는데 mlfqs 옵션이 들어오게 되면 load_avg, recent_cpu, prirority를 일정 주기마다 계산하도록 코드를 수정한 부분이 있다.

```
187   //KMJ start
188   if(thread_mlfqs)
189   {
190       struct thread* cur = thread_current();
191       cur->recent_cpu = cur->recent_cpu + int_to_FP(1);
192       if(ticks % TIMER_FREQ == 0){
193           update_load_avg();
194           update_recent_cpu();
195       }
196       if(ticks % 4 == 0) update_priority();
197   }
```

코드를 보게 되면 매 tick마다 현재 스레드의 recent_cpu를 1씩 늘려준 뒤, load_avg와 recent_cpu는 TIMER_FREQ마다 갱신해주게 되고 priority의 경우 4 tick마다 갱신되도록 한다. TIMER_FREQ의 경우 1초에 해당하는 tick수다. 이 때, load_avg와 recent_cpu는 전부 Fixed Point로 관리하도록 한다.

```
715 /* updating struct thread information - KMJ starts  */
716 void update_load_avg(void){
717     int num=list_size(&ready_list); // num of threads in ready queue
718     struct thread * cur=thread_current();
719     if(cur!=idle_thread)num++; //num = num of threads in ready and running
720     load_avg = add_FPs(mult_FPs(div_FPs(int_to_FP(59),int_to_FP(60)),load_avg)
721           ,mult_FPs(div_FPs(int_to_FP(1),int_to_FP(60)),int_to_FP(num)));
722 }
```

load_avg는 현재 READY상태인 threads 수의 평균을 나타내는 변수로 Fixed Point 관리하도록 한다. 위는 load_avg를 갱신하는 함수로 다음과 같은 수식을 따른다.

$$load\_avg = (59/60) * load\_avg + (1/60) * ready\_threads$$

```
723 void update_recent_cpu(void){
724     //update all recent_cpu in all queue (not ready queue!!)
725     struct list_elem*e;
726     struct thread*t;
727     int rc, nc;
728     for(e=list_begin(&all_list); e!=list_end(&all_list);e=list_next(e)){
729         t=list_entry(e,struct thread, allelem);
730         nc = int_to_FP( t->nice); // type : FP
731         rc = t->recent_cpu; //type : FP
732         t->recent_cpu = mult_FPs(div_FPs(load_avg*2 ,add_FPs(load_avg*2,int_to_FP(1))),rc) + nc;
733     }
734
735 }
```

recent_cpu는 해당 스레드가 최근에 사용했던 cpu 시간을 나타내주는 변수로 이 값이 priority를 갱신하는 데에 사용이 된다. 이 값 또한 load_avg와 같이 TIMER_FREQ 마다 갱신되며 모든 스레드의 recent_cpu를 갱신하도록 한다. 갱신할 때는 다음과 같은 수식을 따른다.

$$recent\_cpu = (2 * load\_avg) / (2 * load\_avg + 1) * recent\_cpu + nice$$

```
745 void update_priority(void){
746     //update all priority in all queue by using information that the thread it self is containing - must be called after calculating recent_cpu and nice.
747     // AND SORT the ready queue!!
748     struct list_elem*e;
749     struct thread*t;
750     int rc, nc;
751     for(e=list_begin(&all_list); e!=list_end(&all_list);e=list_next(e)){
752         t=list_entry(e,struct thread, allelem);
753         nc = int_to_FP( t->nice); // type : FP
754         rc = t->recent_cpu; //type : FP
755         t->priority = PRI_MAX - FP_to_int(div_FPs(rc ,4)) -  FP_to_int(mult_FPs(nc,2));
756         //bound check
757         if(t->priority > PRI_MAX)
758             t->priority = PRI_MAX;
759         else if(t->priority < PRI_MIN)
760             t->priority = PRI_MIN;
761     }
762     list_sort(&ready_list,priority_compare_function,NULL); // sort ready queue
763 }
```

priority는 매 4tick마다 새로 갱신되며, READY 상태인 스레드들 전부의 priority를 계산하도록 한다. 또한 갱신한 뒤에는 새로이 scheduling을 하여 바로 갱신된 priority를 반영하도록 한다. priority를 갱신할 때는 다음과 같은 수식을 따른다.

$$priority = PRI\_MAX - (recent\_cpu / 4) - (nice * 2)$$

3. 시험 및 평가 내용:

3-1. tests

3-1-1. 필수 tests

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-change-2
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-aging
```

pintos/src/threads 디렉터리에서 make check 실행 결과: tests/threads 디렉터리에 있는 프로젝트에서 요구하는 확인 가능한 test 12개를 모두 통과하였다.

3-1-2. priority-lifo

```
Executing 'priority-lifo':
(priority-lifo) begin
(priority-lifo) 16 threads will iterate 16 times in the same order each time.
(priority-lifo) If the order varies then there is a bug.
(priority-lifo) iteration: 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
(priority-lifo) iteration: 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
(priority-lifo) iteration: 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13
(priority-lifo) iteration: 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
(priority-lifo) iteration: 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
(priority-lifo) iteration: 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
(priority-lifo) iteration: 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
(priority-lifo) iteration: 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
(priority-lifo) iteration: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
(priority-lifo) iteration: 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
(priority-lifo) iteration: 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
(priority-lifo) iteration: 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
(priority-lifo) iteration: 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
(priority-lifo) iteration: 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
(priority-lifo) iteration: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
(priority-lifo) iteration: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
(priority-lifo) end
Execution of 'priority-lifo' complete.
Timer: 45 ticks
Thread: 0 idle ticks, 46 kernel ticks, 0 user ticks
Console: 1557 characters output
Keyboard: 0 keys pressed
Powering off...
cse20151179@cspro9:~/pintos/src/threads$
```

pintos/src/threads 디렉터리에서 ../utils/pintos -v -- -q run priority-lifo를 실행해서 make check로 확인 불가능한 test인 priority-lifo도 실행하여 보았다.
pintos/src/tests/threads/priority-lifo.c의 코드를 보고 결과의 타당성을 확인한다. 일단 priority-lifo thread는 53행에서 매우 높은 수준의 priority (PRI_DEFAULT+16)를 갖

는다. 그리고 바로 다음 for loop을 돌며 16개의 threads를 priority PRI_DEFAUT+0~PRI_DEFAUT+15까지 바꿔가며 thread_create를 한다. 이때 각 thread는 하나의 lock을 가리키는 포인터를 공유한다. threads 16개 생성이 끝나면 이제 66행에서 이 threads의 부모인 priority-lifo thread의 우선순위를 PRI_DEFAULT로 변경한다. 이렇게 되면 자식 threads보다 낮은 우선순위를 갖게 된다. 또한 set_priority()함수는 thread_yield()를 호출하기 때문에 CPU의 권한은 자식 thread에게 간다. priority가 PRI_DEFAUT+15인 thread부터 순차적으로 자신의 함수(simple_thread_func)를 실행시킨다. 이 함수는 95행에서 *(*data->op)++=data->id;를 수행한다. 일단 (*data->op)++가 실행되어 op가 참조하는 값을 1 증가시킨다. postfix increment는 indirection보다 operator precedence가 더 높기 때문이다. 그리고 이것이 가리키는 곳에 자신의 id값을 16번 쓴다. 이때 한 번 쓰고 lock을 풀어주지만, 나머지 threads들은 priority가 다 다르고 자신이 가장 priority가 높기 때문에 다음 루프에서도 자신이 쓴다. 따라서 op가 최초에 가리키던 곳부터 '15'가 16개 써지고 이후 '14'가 16개 써지는 식으로 해서 마지막에는 '0'이 16번 써진다. 이 과정이 완료되면 모든 자식 threads가 종료되고 다시 priority-lifo thread가 CPU를 차지한다. 여기서 output 포인터가 op의 최초 위치부터 움직이며 id를 읽어 printf로 출력한다. 따라서 15~0순서로 16개씩 출력이 될 것이다. 위의 핀토스 실행 화면에서 이것이 제대로 실행되고 있음을 알 수 있다.

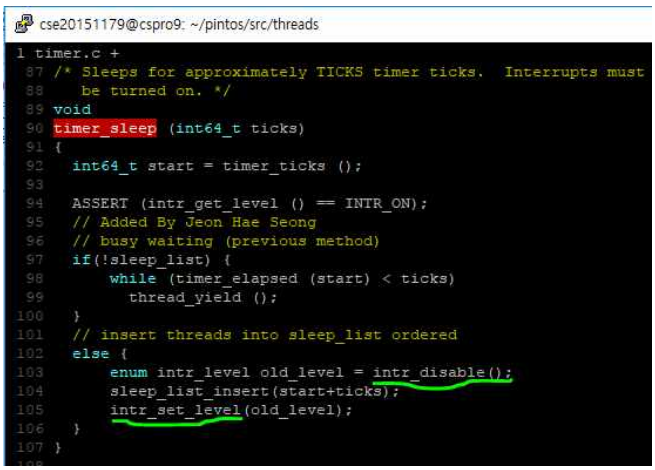
따라서 필수 tests 13개를 모두 통과한 것이다.

3-1-3. BSD Scheduler 관련 mlfqs tests

```
pass tests/threads/mlfqs-block
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
```

BSD Scheduler를 구현하였고 따라서 mlfqs 관련 tests 10개를 모두 통과하였다.

3-2. 보건 및 안정성



timer.c의 timer_sleep()함수를 보면 sleep_list_insert() 전후로 interrupt를 disable해주었다가 다시 able로 바꿔준다. thread를 재우는 과정은 상당히 중요하고 오류가 나면 안되는 과정이라고 판단해서 interrupt를 잠시 꺼주어 이 작업이 제대로 처리되도록 하였다. 이는 thread_unblock, thread_exit, thread_yield, thread_create 등 중요한 작업을 처리할 때 (매뉴얼에 의하면 kernel thread와 interrupt handler 사이에 데이터 공유가 필요할 때) 제한적으로 사용하는 방식이다. (이 경우는 lock을 사용해도 괜찮을 듯하다.) 이렇게 해서 안정성을 높였다.

```
void
thread_set_priority (int new_priority)
{
  if(new_priority > PRI_MAX)
      new_priority = PRI_MAX;
  else if(new_priority < PRI_MIN)
      new_priority = PRI_MIN;
  thread_current ()->priority = new_priority;
```

```
//bound check
if(t->priority > PRI_MAX)
    t->priority = PRI_MAX;
else if(t->priority < PRI_MIN)
    t->priority = PRI_MIN;
```

Priority Scheduler의 경우 thread_set_priority에서 priority 범위를 벗어나는 값으로 변경을 요청할 경우 이 경계를 넘어가지 못하도록 제한을 주었다. BSD Scheduler의 경우에서도 비슷하게 update_priority()에서 범위를 벗어나는 값이 계산되어도 이를 넘지 못하게 하였다.

```
void
thread_set_nice (int nice)
{
  /*written by KMJ */
    if(nice>20)
        nice=20;
    else if(nice<-20)
        nice=-20;
    thread_current()->nice=nice;
```

BSD에서 set_nice의 경우에도 정해진 값 -20~20을 벗어나지 못하도록 설정해주었다.

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick();
  //KMJ start
  if(thread_mlfqs)
  {
      struct thread* cur = thread_current();
      cur->recent_cpu = cur->recent_cpu + int_to_FP(1);
      if(ticks % TIMER_FREQ == 0){
          update_load_avg();
          update_recent_cpu();
      }
      if(ticks %4==0)
          update_priority();
  }

  // Added By Jeon Hae Seong
  enum intr_level old_level = intr_disable();
  sleep_list_wakeup();
  intr_set_level(old_level);
}
```

BSD의 경우 timer_interrupt에서 1초에 한번 load_avg와 recent_cpu를 재계산한다. 이 때 time.h의 시간을 사용하지 않고 tick % TIMER_FREQ을 사용했기 때문에, 매번 같은 입력을 해도 같은 결과가 나오도록 구현하였다. 만약 time.h의 시간 차이를 사용했다면 CPU의 처리 결과에 따라서 매 실행 시 다소 다른 결과를 냈을 것이다. 모든 함수가 ticks에 맞춰서 작동하기 때문에 항상 동일한 결과를 낸다.

3-3. 생산성 및 내구성

ready_list를 Priority 수만큼 사용하는 대신 하나만을 사용하고 여기에서 BSD scheduler의 기능을 똑같이 구현했다. 이를 통해 불필요한 공간을 줄였다. 또한 Busy waiting을 원천 차단하고 thread_block()을 통해 비효율적인 CPU의 낭비를 막았다. 그리고 sleep_list에 thread를 삽입할 때 단순히 list_insert를 하지 않고 삽입할 때부터 ticks에 따라 정렬하여 삽입함으로써 pop할 때마다 전체를 탐색하지 않고 바로 맨 앞 원

소를 꺼낼 수 있도록 구현하였다. 이를 통해 시간 효율을 높였다. Fixed Point real number 타입을 새로 만들지 않고 기존의 int형을 사용하고 대신 이에 따른 연산 함수들을 사용함으로써 불필요한 공간은 줄이고 연산 실수도 막고 가독성도 높였다.

V. 기타

1. 연구 조원 기여도: 권명준(50%) 전해성(50%)

2. 기타 본 설계 프로젝트를 수행하면서 느낀 점을 요약하여 기술하라. 내용은 어떤 것이든 상관이 없으며, 본 프로젝트에 대한 문제점 제시 및 제안을 포함하여 자유롭게 기술할 것.
(느낀점)
- 권명준: 지난번에는 전해성씨가 더 많이 기여했기에 이번에는 내가 더 많이 하려고 노력했고 좋은 결과가 나와서 기쁘다. 이번에 가장 오랫동안 발목을 잡았던 부분은 BSD Scheduler 관련 tests가 알 수 없는 이유로 테스트를 통과하지 않았던 문제이다. 이것은 실수 연산을 처리하면서 Fixed Point real과 int를 딱히 type을 새로 만들지 않고 모두 int로 사용하면서 문맥에 따라 다르게 해석하려고 해서 real value가 올 자리에 integer value가 왔는데 아무도 눈치채지 못해서 일어난 문제였다. 이를 발견하고 테스트를 돌리자 모든 테스트가 한 번에 통과되었다. 앞으로는 typedef int FP;와 같이 번거롭더라도 새 자료형을 만들어서 사용하는 게 좋을 것 같다. 객체 지향 언어라면 반드시 class를 만들어서 사용하도록 해야겠다. C++이었다면 연산자 오버로딩을 해서 사용했어도 괜찮을 것 같다.

- 전해성 : 프로젝트를 너무 늦게 시작하면 사람이 피폐해지고 학교를 그만두고 싶어진다는 점을 절실하게 느꼈다. 그리고 이 프로젝트가 왜 Project 1인지 알 거 같았다. thread에 관련된 함수들을 미리 써보고 이해해보라는 것이 느껴지는 프로젝트였다.