

Pintos Project 3: Thread

(설계 프로젝트 수행 결과)

과목 명 : [CSE4070] 운영체제

담당 교수 : 소정민

조 / 조원 : 1조 / 이규연, 허남규

개발 기간 : 2018/11/22 ~ 2018/12/04

프로젝트 제목 : Pintos Project 3 : Thread

제출일 : 2018/12/4

참여 조원 : 20141552 이규연, 20161662 허남규

Table of Contents

I. 개발 목표	3
II. 개발 범위 및 내용.....	3
A. 개발 범위	3
a) Non-Busy-Waiting Alarm Clock	3
b) Priority 스케줄러	4
B. 개발 내용	4
a) Non-Busy-Waiting Alarm Clock	4
b) Priority 스케줄러	5
III. 추진 일정 및 개발 방법	7
A. 추진 일정	8
B. 개발 방법	8
a) 채점 wrapper 구현	8
b) 알림 script 및 tmux 활용	8
c) Fixed Point Number 구현	8
d) BSD Scheduler (thread_FOREACH)	8
e) Alarm expire, Priority (list_insert_sorted)	9
f) 기타 구현	9
C. 연구원 역할 분담	9
IV. 연구 결과	9
A. 합성 내용 (플로우차트)	9
a) 기존 alarm clock 구조	9
b) 수정된 alarm clock 구조	9
c) Priority 적용 전 scheduling 구조	10
d) Priority 적용 후 scheduling 구조	10
e) BSD priority scheduling 구조	10

B. 제작 내용	11
a) Makefile.build.....	11
b) Thread 모듈.....	11
c) Timer 모듈.....	15
d) Synch 모듈	17
e) Gyu 모듈	17
f) Float 모듈	19
C. 시험 및 평가 내용.....	20
V. 기타.....	21
A. 연구 조원 기여도.....	21
B. 소감.....	21

I. 개발 목표

이번 과제에서는 Pintos에서 제공하는 기본적인 alarm clock과 thread scheduling 시스템을 개선하는 것을 목표로 합니다.

그 과정에서 thread 시스템을 활용한 busy-waiting을 이해하고 이를 개선하기 위해 timer_interrupt 함수*를 수정합니다. 또한, 기존 스케줄러에 priority와 aging 기능을 추가하여 multi-level feedback queue 개념을 실전에 적용합니다.

이로써 OS의 동기화 문제에 관한 이해를 확장하는 것을 목표로 합니다.

* <II-B 개발 내용> 후술

II. 개발 범위 및 내용

A. 개발 범위

a) Non-Busy-Waiting Alarm Clock

기존 Pintos 소스에는 busy-waiting 기반의 alarm clock이 구현되어 있습니다. Thread가 running state와 ready state를 오가며 설정된 시간(ticks)이 지나가기를 기다리도록 구현되어 있습니다. 이번 프로젝트에서는

thread scheduling 계층 아래에 있는 timer_interrupt를 함수를 수정하여 타이머가 만료될 때까지 thread를 block하여 waiting queue에서 CPU 시간을 잡아먹지 않도록 합니다.

b) Priority 스케줄러

Pintos에서는 기본적인 FCFS 형태의 ready queue를 관리하는 스케줄러와 기본적인 semaphore construct가 제공됩니다.

Priority 스케줄러에서는 priority를 도입하기 위해 앞서 언급한 ready queue를 sorted list 형태로 관리하고, 현재 thread와 ready queue에서 기다리고 있는 thread 사이의 priority 순위 변동, pintos manual에 언급된 semaphore 관련 priority inversion 문제를 해결합니다.

또한 nice, load average, recently used cpu 등의 개념을 기반으로 한 BSD 스케줄러를 구현합니다. BSD 관련 연산은 thread system의 기반이 되기 때문에 주기적인 수행을 위해 timer_interrupt를 수정하여 구현합니다.

B. 개발 내용

a) Non-Busy-Waiting Alarm Clock

앞서 언급했듯이 기존의 alarm clock은 thread-scheduling을 이용한 busy-waiting 방식을 사용합니다. Running state로 dispatch 후 단순 비교 연산만 하고 다시 preempt하는 방식이기 때문에 실제 CPU 시간을 많이 사용하지는 않습니다. 하지만, scheduling overhead가 계속해서 발생하기 때문에 개선의 여지가 있습니다.

이번 프로젝트에서는 이를 개선하기 위해 thread scheduling의 기본이 되는 external hardware interrupt (timer_interrupt)를 직접 수정합니다. OS 설계 관점에서 timer_interrupt는 programmable interrupt timer (PIT)에 의해 주기적으로 호출되는 함수입니다. 아래 <devices/timer.c> 내 <timer_init> 함수에서 알 수 있듯이, TIMER_FREQ를 주기로 하여 timer_interrupt가 호출되도록 PIT가 설정되어 있습니다.

```
pit_configure_channel (0, 2, TIMER_FREQ);
intr_register_ext (0x20, timer_interrupt, "8254 Timer");
```

Non-busy-waiting 방식의 alarm clock 구현하기 위해 alarm clock thread가 생성된 후 바로 block 상태로 만들고, timer_interrupt에서 매 tick마다 확인해서 만료된 타이머를 ready 상태로 만듭니다. 즉, thread 시스템의 토대가 되는 timer_interrupt를 직접 수정하여 alarm clock을 개선합니다.

주요 수정/추가 사항은 다음과 같습니다.

- jeepers_sleepers: 현재 block 상태에 있는 alarm clock 리스트입니다. timer_interrupt마다 sequential search를 피하기 위해 알람이 늦게 끝나는 순서대로 정렬되어 있습니다.
- timer_init(): jeepers_sleepers 리스트 초기화
- timer_sleep(): 앞서 언급한 바와 같이 알람의 종료 시간을 계산한 후 jeepers_sleepers 리스트에 현재 thread를 넣은 후에 스스로를 block 시킵니다.
- timer_interrupt(): jeepers_sleepers 리스트에서 시간이 만료된 알람을 확인해서 깨워야 할 thread를 깨웁니다.
- struct thread: alarm clock에 필요한 timer_expiry 변수 추가

b) Priority 스케줄러

상술한 바와 같이 기존 Pintos 소스에 구현된 스케줄러는 기본적인 FCFS 형태로 구현됩니다. Pintos에서 제공하는 리스트 자료 구조를 ready queue로 사용하는 방식입니다. 여기서 priority 개념을 도입하기 위해 이를 priority가 낮은 순으로 sorting하여 관리합니다. 또한 현재 priority가 바뀌거나 새로운 thread가 추가될 때 등의 상황에서 현재 running 상태의 thread가 가장 높은 priority를 갖도록 하기 위해 현재 thread를 yield시키기도 합니다. 또한 priority inversion 문제를 해결하기 위해 semaphore 대기열도 priority가 낮은 순으로 sorting하여 관리합니다.

BSD 스케줄러를 구현할 때에는 nice, load_avg, recent_cpu 등의 수치를 계속해서 업데이트해주어야 하는데, 이 또한 상술한 timer_interrupt 함수를 수정해서 구현합니다.

이번 프로젝트에서 구현한 BSD 스케줄러의 상세 알고리즘은 다음과 같습니다.

- 초기화
 - nice 0 | 부모 thread에서 상속
 - recent_cpu 0 | 부모 thread에서 상속
 - priority thread 생성 시 계산
- 갱신 (주기별)
 - 1 tick: recent_cpu (현재 thread)
 - 4 tick: priority (모든 thread)
 - 1 second : load_avg, recent_cpu (모든 thread)
 - + Nice 값 설정 시 priority 갱신
- 갱신 값
 - $priority = PRI_{MAX} - \frac{recent_{cpu}}{4} - nice * 2$
 - $recent_{cpu} = \frac{2load_{avg}}{2load_{avg}+1} * recent_{cpu} + nice$
 - $load_{avg} = \frac{59}{60}load_{avg} + \frac{1}{60}ready_{threads}$

주요 구현 사항은 다음과 같습니다.

- thread_aging: aging 적용 여부를 확인하기 위한 flag (아래 함수 구현 시 사용)
- parse_options(): thread_aging flag를 받아서 변수를 설정합니다
- struct thread: BSD scheduler에 사용할 nice, recent_cpu 변수 추가
- load_avg: BSD scheduler에 사용할 load_avg 값을 저장하는 static 변수 (gyu.c의 static 변수)
 - encapsulation을 위해 thread.c에서 외부 접근 시 gyu.h에서 제공하는 get_load_avg() 함수 사용
- ready_threads(): BSD scheduling에 사용할 현재 thread의 개수를 반환하는 함수 (thread_current() == idle_thread 여부 고려)
- thread_next_priority(): Waiting queue에서 기다리고 있는 thread 중 priority가 가장 높은 thread (리스트 상에서 가장 앞쪽에 있는 thread)의

priority를 반환하는 함수입니다. 기다리는 thread가 없을 경우 -1을 반환합니다.

- `thread_create()`: 현재 thread보다 새로운 thread의 priority가 높을 경우, 현재 thread를 yield하도록 수정했습니다.
- `thread_init()`: 첫 thread의 nice, recent_cpu 값 및 load_avg 값 초기화
- `init_thread()`: priority 설정 및 nice, recent_cpu 값 상속
- `{set|get}_{priority|nice|load_avg|recent_cpu}`: fixed point 형태로 저장된 (현재 thread의) nice, priority, load_avg, recent_cpu 값에 100을 곱하여 integer 형태로 반환하는 함수 구현. `Set_priority()` 함수는 aging, mlfqs가 해제된 상태에서만 동작하도록 하였고, waiting queue에서 기다리고 있는 thread보다 priority를 낮게 설정할 경우, yield되도록 했습니다.
- `float.h`: fixed point 형태의 숫자의 연산을 처리하는 모듈 구현 (함수 구현 사항은 함수 이름에서 trivial하게 알 수 있습니다)

gyu.h 모듈 내 구현 사항

- `higher_priority()`: priority 정렬에 필요한 `list_less_func` 구현
- `{init|get|update}_load_avg()`: load_avg 값 초기화, 반환, 갱신 구현
- `thread_update_{priority|recent_cpu}()`: priority와 recent_cpu를 갱신하는 `thread_action_func`

timer_interrupt() 내 구현 사항

- 상술한 `thread_action_func`, `update_load_avg`, `thread_FOREACH` 함수를 이용하여 BSD scheduler에 필요한 갱신 사항을 구현했습니다. 세부 사항은 <II-B-(b)> 초반부 설명 참조.

III. 추진 일정 및 개발 방법

A. 추진 일정

- 2018.11.22 ~ 2018.11.29 Custom 채점 시스템 구현 (채점 wrapper)
- 2018.11.30 ~ 2018.12.01 Pintos Manual 학습
- 2018.12.02 ~ 2018.12.04 소스 수정 및 디버깅
- 2018.12.04 ~ 2018.12.05 보고서 작성

B. 개발 방법

a) 채점 wrapper 구현

Directory 변경 없이 테스트를 케이스 단위로 손쉽게 돌려볼 수 있도록 스크립트를 작성하고, 이를 어디서든 실행할 수 있도록 PATH 등의 설정을 담당하는 init.sh 스크립트 또한 새로 작성했습니다. 자세한 구현 사항은 scripts 디렉토리 안의 README.md에서 확인해볼 수 있습니다.

b) 알림 script 및 tmux 활용

채점 도중에 소스를 디버깅할 수 있도록 위의 채점 스크립트를 background에서 실행하기 위해 tmux 커맨드를 활용했습니다. 또한, 채점이 완료되기를 “busy-wait” 하지 않고, 완료되었다는 알림을 받을 수 있도록 slack web hook을 이용한 알림 스크립트를 작성하고 PATH에 추가하여 알림 명령어로 활용했습니다. 이 또한 scripts 디렉토리 안에 구현 및 설명되어 있습니다.

c) Fixed Point Number 구현

Fixed point number 연산 과정에서 프로그래밍 실수를 방지하기 위해 연산을 위한 함수는 모두 float.h 모듈에 구현하고, float.c를 컴파일 단위에 포함시키기 위해 Pintos 메뉴얼에서 설명하는 바와 같이 Makefile.build 파일 안에 float.c 소스 파일을 지정했습니다.

d) BSD Scheduler (thread_FOREACH)

BSD scheduler에서는 특정 시간 단위로 recent_cpu, load_avg, priority를 갱신해주어야 하는데, 이를 간편하게 하기 위해 thread_action_func 형태의 갱신 함수 (thread_update_priority, thread_update_recent_cpu)를 gyu.h

모듈 내에 구현하여, timer_interrupt 함수에서 thread_FOREACH 함수를 통해 thread 단위로 호출하는 방식으로 구현했습니다.

e) Alarm expire, Priority (list_insert_sorted)

Ready queue 및 semaphore waiting queue를 구현할 때 priority 순서로 유지하는데에 있어서도 list_less_func 형태의 priority 비교 함수 expires_earlier() 함수를 만들어서 Pintos에서 제공하는 list 자료구조의 list_insert_sorted() 함수를 활용했습니다.

f) 기타 구현

기타 세부 구현은 C 기본 문법을 사용하여 logic에 맞는 코드를 작성하고, 상술한 방법과 같이 채점을 하고, 디버깅을 하는 방식으로 구현을 해나갔습니다.

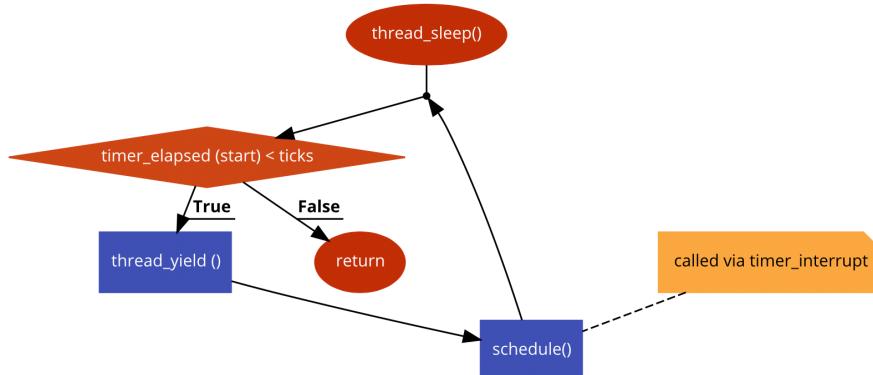
C. 연구원 역할 분담

- 이규연: Alarm timer 구현, priority 스케줄러 구현
- 허남규: 채점 wrapper 구현, BSD 스케줄러 구현, 보고서 작성

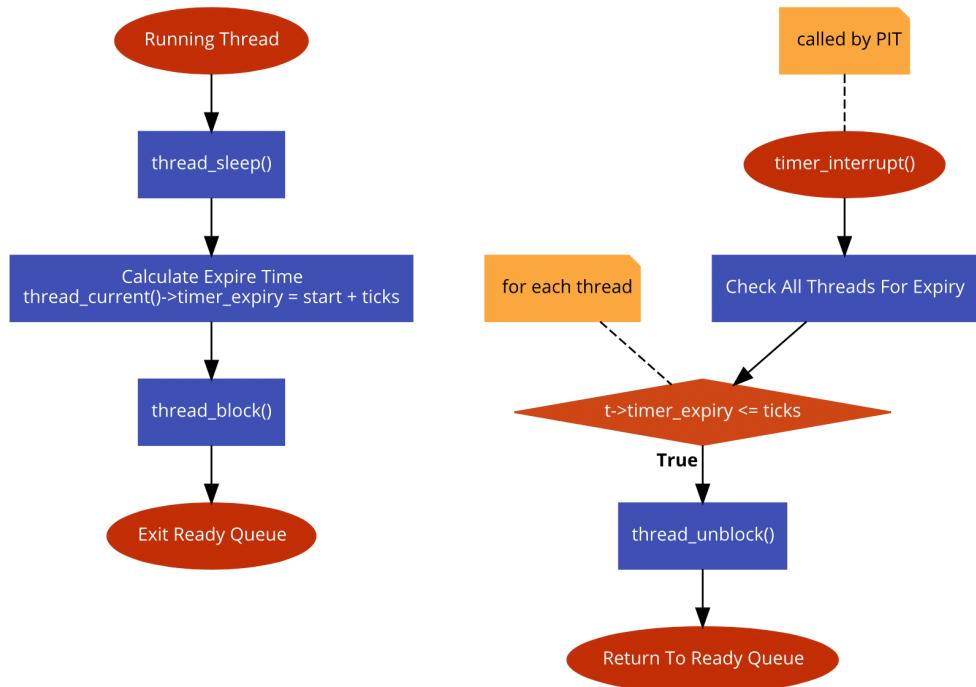
IV. 연구 결과

A. 합성 내용 (플로우차트)

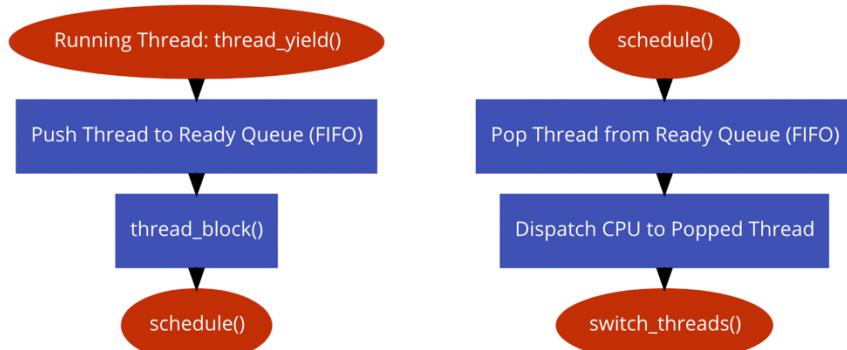
a) 기존 alarm clock 구조



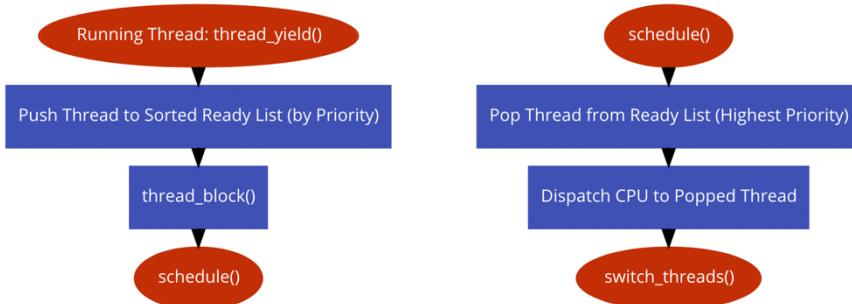
b) 수정된 alarm clock 구조



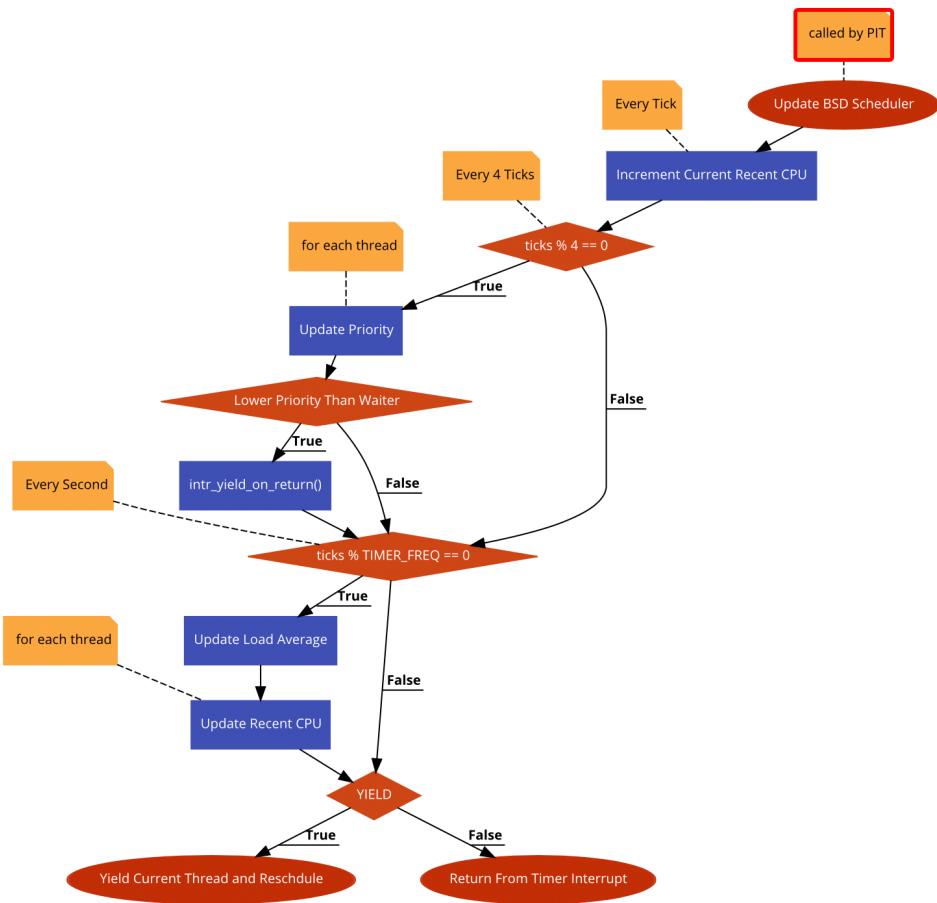
c) Priority 적용 전 scheduling 구조



d) Priority 적용 후 scheduling 구조



e) BSD priority scheduling 구조



B. 제작 내용

a) Makefile.build

```

21 threads_SRC += threads/synch.c      # Synchronization.
22 threads_SRC += threads/palloc.c     # Page allocator.
23 threads_SRC += threads/malloc.c     # Subpage allocator.
24 #-----GYU-----#
25 threads_SRC += threads/gyu.c        # Extra stuff for project 3
26 threads_SRC += threads/float.c      # Fixed point arithmetic for project 3
27 #-----#
28
29 # Device driver code.
30 devices_SRC = devices/pit.c        # Programmable interrupt timer chip.
31 devices_SRC += devices/timer.c     # Periodic timer device.
32 devices_SRC += devices/kbd.c       # Keyboard device.

```

새로운 소스 파일을 컴파일 단위에 추가

b) Thread 모듈

```

79     If true, use multi-level feedback queue scheduler.
80     Controlled by kernel command-line option "-o mlfqs". */
81 bool thread_mlfqs;
82 /*----- GYU -----*/
83 #ifndef USERPROG
84 bool thread_agging;
85 #endif
86 /*----- */
87
88
89 static void kernel_thread (thread_func *, void *aux);
90
91 /*----- */

```

Aging 기능 사용 여부에 관한 boolean flag 추가

```

114 void
115 thread_init (void)
116 {
117     ASSERT (intr_get_level () == INTR_OFF);
118
119 /*----- GYU -----*/
120     init_load_avg();
121 /*----- */
122
123     lock_init (&tid_lock);
124     list_init (&ready_list);
125     list_init (&all_list);
126

```

Thread 시스템 초기화 시 load_avg(gyu.h 모듈 내) 값 초기화

```

252 /* Add to run queue. */
253 thread_unblock (t);
254
255 /*----- GYU -----*/
256 // yield current thread so that the new thread with higher priority can be
257 // dispatched by the scheduler.
258 // this is also applicable to the thread_agging case
259 if (thread_get_priority() < priority) {
260     thread_yield();
261 }
262 /*----- */
263
264 return tid;
265 }
266

```

새로운 thread 생성 시 현재 thread보다 priority가 높을 때 yield

```

293 {
294     enum intr_level old_level;
295
296     ASSERT (is_thread (t));
297
298     old_level = intr_disable ();
299     ASSERT (t->status == THREAD_BLOCKED);
300 /*----- GYU -----*/
301 // list_push_back (&ready_list, &t->elem);
302 list_insert_ordered(&ready_list, &t->elem, higher_priority, NULL);
303 /*----- */
304     t->status = THREAD_READY;
305     intr_set_level (old_level);
306 }
307

```

Priority 순으로 ready queue를 유지하기 위해 정렬된 상태를 유지하며 queue에 새로운 thread 삽입 (higher_priority 함수 내용은 후술)

```

362 /* Yields the CPU. The current thread is not put to sleep and
363    may be scheduled again immediately at the scheduler's whim. */
364 void
365 thread_yield (void)
366 {
367     struct thread *cur = thread_current ();
368     enum intr_level old_level;
369
370     ASSERT (!intr_context ());
371
372     old_level = intr_disable ();
373     if (cur != idle_thread) {
374         /*-----GYU-----*/
375         // list_push_back (&ready_list, &cur->elem);
376         list_insert_ordered(&ready_list, &cur->elem, higher_priority, NULL);
377     /*-----GYU-----*/
378     }
379
380     cur->status = THREAD_READY;
381     schedule ();
382 }
```

thread_yield 시 현재 thread를 ready queue에 넣을 때에도 priority 정렬 유지

```

399     }
400 }
401
402 /*-----GYU-----*/
403 /* Sets the current thread's priority to NEW_PRIORITY. */
404
405 /*
406  * If the priority is lower than the next thread to be scheduled, then yield the
407  * current thread so that the next thread w/ higher priority can be dispatched.
408 */
409 void
410 thread_set_priority (int new_priority)
411 {
412     if (!thread_mlfqs) {
413         thread_current ()->priority = new_priority;
414         if (new_priority < thread_next_priority ()) {
415             thread_yield();
416         }
417     }
418 }
419
420 /* Returns the current thread's priority. */
421 int
422 thread_get_priority (void)
423 {
424     return thread_current ()->priority;
425 }
426
427 /* Sets the current thread's nice value to NICE. */
428 .
```

Priority 설정 시, 기다리고 있는 thread가 priority가 높은 경우에는 현재 thread를 yield.

```

426 /* Sets the current thread's nice value to NICE. */
427 void
428 thread_set_nice (int nice UNUSED)
429 {
430     struct thread *current = thread_current();
431     current->nice = nice;
432
433     thread_update_priority(current, NULL);
434     if (current->priority < thread_next_priority())
435         thread_yield();
436 }
437
438 /* Returns the current thread's nice value. */
439 int
440 thread_get_nice (void)
441 {
442     return thread_current()->nice;
443 }
444
445

```

Nice 설정 시 priority 다시 계산 후 기다리고 있는 thread보다 priority가 낮아지면 yield.

```

446 /* Returns 100 times the system load average. */
447 int
448 thread_get_load_avg (void)
449 {
450     return (int) (((int64_t) get_load_avg() * 100) >> Q);
451 }
452
453 /* Returns 100 times the current thread's recent_cpu value. */
454 int
455 thread_get_recent_cpu (void)
456 {
457     const int recent_cpu = thread_current()->recent_cpu;
458     return (int) (((int64_t) recent_cpu * 100) >> Q);
459 }
460

```

현재 thread의 recent_cpu와 load_avg를 fixed-point에서 100배 증가시킨 일반 int로 반환하기 위한 연산 과정

```

534 static void
535 init_thread (struct thread *t, const char *name, int priority)
536 {
537     ASSERT (t != NULL);
538     ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
539     ASSERT (name != NULL);
540
541     memset (t, 0, sizeof *t);
542     t->status = THREAD_BLOCKED;
543     strlcpy (t->name, name, sizeof t->name);
544     t->stack = (uint8_t *) t + PGSIZE;
545     /*----- GYU -----*/
546     if (thread_aging) {
547         // you should check for priority inversion after call to init_thread
548         t->nice = running_thread()->priority;
549         t->recent_cpu = running_thread()->recent_cpu;
550         thread_update_priority(t, NULL);
551     } else {
552         t->priority = priority;
553     }
554     /*-----*/
555     t->magic = THREAD_MAGIC;
556     list_push_back (&all_list, &t->allelem);

```

Thread 초기화 시 nice, recent_cpu 상속 및 priority 재연산 (혹은 인자로 받은 priority로 설정)

```

690
691 ----- GYU -----
692 /*
693 */
694 * get number of ready threads, including the currently running thread (unless
695 * it is the idle thread
696 */
697 int ready_threads () {
698     int count = list_size(&ready_list);
699     int current = (thread_current() != idle_thread);
700     return count + current ;
701 }
702
703 /*
704 * get priority of the next thread if it exists
705 */
706 int thread_next_priority() {
707     if (!list_empty(&ready_list))
708         return list_entry(list_front(&ready_list), struct thread, elem)->priority;
709     else
710         return -1; // since PRI_MIN == 0
711 }
712 ----- GYU -----

```

현재 running 또는 ready 상태인 thread의 개수 연산 및 다음 기다리는 thread의 priority 반환.

c) Timer 모듈

```

72 void
73 timer_init (void)
74 {
75     pit_configure_channel (0, 2, TIMER_FREQ);
76     intr_register_ext (0x20, timer_interrupt, "8254 Timer");
77 ----- GYU -----
78     list_init(&jeepers_sleepers);
79 ----- GYU -----
80 }
81

```

타이머 시스템 초기화 시 jeepers_sleepers (대기 중인 thread 리스트) 초기화

```

50 ----- GYU -----
51 /*
52 * list_less_func for ordered insertion to jeepers_sleepers -GYU
53 */
54 static bool
55 expires_earlier(const struct list_elem *a,
56                  const struct list_elem *b,
57                  void *aux UNUSED)
58 {
59     struct thread *ta = list_entry(a, struct thread, elem);
60     struct thread *tb = list_entry(b, struct thread, elem);
61     return ta->timer_expiry < tb->timer_expiry;
62 }
63
64 /*
65 * List of sleeping threads (ordered) -GYU
66 */
67 static struct list jeepers_sleepers;
68 ----- GYU -----

```

Jeepers_sleepers 리스트에 thread 추가 시 타이머 종료 시간 순으로 정렬된 상태로 추가하기 위한 list_less_func

```

126
127 /*----- GYU -----*/
128 /* Sleeps for approximately TICKS timer ticks. Interrupts must
129   be turned on. */
130
131 /*
132  * Disable interrupts, calculate ending time and add to timer queue
133  * that will be monitored by the system timer (timer_interrupt()), block
134  * the thread and wait for the system to unblock it. -GYU
135 */
136 void
137 timer_sleep (int64_t ticks)
138 {
139     int64_t start = timer_ticks ();
140     int64_t end = start + ticks;
141     ASSERT (intr_get_level () == INTR_ON);
142
143     enum intr_level old_level = intr_disable();
144
145     thread_current ()->timer_expiry = end;
146     list_insert_ordered(&jeepers_sleepers, &thread_current ()->elem, expires_earlier, NULL);
147     thread_block();
148
149     intr_set_level (old_level);
150 }
151 /*----- */

```

종료 시간 연산, jeepers_sleepers에 현재 thread 추가 후 ready queue에서 제거

```

232 static void
233 timer_interrupt (struct intr_frame *args UNUSED)
234 {
235     ticks++;
236
237     // check for threads to wake up
238     struct list_elem *e;
239     for (e = list_begin(&jeepers_sleepers);
240         e != list_end(&jeepers_sleepers); ) {
241         struct thread *t = list_entry(e, struct thread, elem);
242         if (t->timer_expiry <= ticks) {
243             e = list_remove(e);
244             thread_unblock(t);
245         } else break;
246     }

```

현재 돌아가고 있는 타이머를 순회하며 종료된 타이머를 가진 thread를 ready queue로 반환 (정렬된 상태이기 때문에 끝까지 순회할 필요는 없음)

```

249     if (thread_aging || thread_mlfqs) {
250         if (true) {
251             // 1. increment recent_cpu
252             thread_current ()->recent_cpu += 1 << Q;
253         }
254
255         if (ticks % 4 == 0) {
256             // 1. update all priority
257             thread_foreach(thread_update_priority, NULL);
258
259             // yield if priority has been inverted
260             if (thread_current ()->priority < thread_next_priority ())
261                 intr_yield_on_return();
262         }
263
264         if (ticks % TIMER_FREQ == 0) {
265             // 1. update load_avg
266             update_load_avg();
267             // 2. update all recent_cpu
268             thread_foreach(thread_update_recent_cpu, NULL);
269         }
270     }

```

Aging 또는 mlfqs가 활성화된 상태일 경우, 앞서 언급한 BSD scheduler에 필요한 연산 수행

d) Synch 모듈

```
134 void
135 sema_up (struct semaphore *sema)
136 {
137     enum intr_level old_level;
138
139     ASSERT (sema != NULL);
140
141     old_level = intr_disable ();
142
143     bool preempt = false;
144     if (!list_empty (&sema->waiters)) {
145         struct thread *thread;
146         thread = list_entry(list_pop_front (&sema->waiters), struct thread, elem);
147
148         thread_unblock (thread);
149         preempt = (thread_get_priority() < thread->priority);
150     }
151
152     sema->value++;
153     intr_set_level (old_level);
154
155     /*-----GYU-----*/
156     // if the awaken thread has a higher priority than the current thread,
157     // then preempt!
158     if (preempt) thread_yield();
159     /*-----GYU-----*/
160 }
```

현재 semaphore를 기다리고 있다가 풀린 thread의 priority가 현재 돌아가고 있는 thread의 priority보다 높은 경우, 풀린 thread가 실행될 수 있도록 현 thread를 yield

```
79 void
80 sema_down (struct semaphore *sema)
81 {
82     enum intr_level old_level;
83
84     ASSERT (sema != NULL);
85     ASSERT (!intr_context ());
86
87     old_level = intr_disable ();
88     while (sema->value == 0)
89     {
90     /*-----GYU-----*/
91     // insert sema waiter in order of higher priority (highest priority thread
92     // goes to the front)
93     // list_push_back (&sema->waiters, &thread_current ()->elem);
94     list_insert_ordered(&sema->waiters, &thread_current ()->elem,
95                         higher_priority, NULL);
96     /*-----GYU-----*/
97 }
```

Priority가 높은 순서대로 semaphore를 풀 수 있도록 (잡을 수 있도록) 새로운 thread를 waiting queue에 추가할 때 priority기준 정렬된 상태로 추가

e) Gyu 모듈

BSD scheduler 갱신 및 thread priority 순위 관련

```

17
18 static int load_avg;
19
20 bool
21 higher_priority(const struct list_elem *left,
22                  const struct list_elem *right,
23                  void *aux UNUSED) {
24     struct thread *left_thread = list_entry(left, struct thread, elem);
25     struct thread *right_thread = list_entry(right, struct thread, elem);
26     return left_thread->priority > right_thread->priority;
27 }

```

Ready queue 및 semaphore waiting queue를 priority 기준 정렬된 상태로 유지하기 위한 list_less_func

```

29 void
30 init_load_avg(void) {
31     load_avg = 0;
32 }
33
34 void
35 update_load_avg(void) {
36     int f_59 = (59 << Q) / 60;
37     int f_01 = (1 << Q) / 60;
38
39     int f0 = float_times_float(f_59, load_avg);
40     int f1 = f_01 * ready_threads();
41
42     load_avg = f0 + f1;
43 }
44
45 int
46 get_load_avg() {
47     return load_avg;
48 }

```

Load_avg 초기화, 갱신, 반환 관련 함수. Gyu 모듈 내에 load_avg 원본을 encapsulate하기 위해 관련 함수 다수 제공. Fixed-point number 연산을 위한 관련 함수 호출 및 bit-shift 활용

```

50 void
51 thread_update_priority(struct thread *t, void *aux UNUSED) {
52     int f_max = PRI_MAX << Q;
53     int f_r4 = t->recent_cpu / 4;
54     int f_n2 = t->nice * (2 << Q);
55
56     t->priority = (f_max - f_r4 - f_n2) >> Q;
57
58     if (t->priority < PRI_MIN) t->priority = PRI_MIN;
59     else if (PRI_MAX < t->priority) t->priority = PRI_MAX;
60 }
61
62 void
63 thread_update_recent_cpu(struct thread *t,
64                          void *aux UNUSED) {
65     int f_coef = float_over_float(2 * load_avg, 2 * load_avg + (1 << Q));
66     int f_term = float_times_float(f_coef, t->recent_cpu);
67     int f_nice = t->nice * (1 << Q);
68     t->recent_cpu = f_term + f_nice;
69 }

```

BSD scheduler에서 thread 단위로 recent_cpu, priority를 갱신하기 위한 thread_action_func.. Fixed-point number 연산을 위한 관련 함수 호출 및 bit-shift 활용

f) Float 모듈

Fixed-point number (이하 float)와 일반 int 관련 연산 처리

```
18
19 const int Q = 14;
20
21 int int_plus_float(int i, int f) {
22     return (i << Q) + f;
23 }
24
25 int int_minus_float(int i, int f) {
26     return (i << Q) - f;
27 }
28
29 int int_times_float(int i, int f) {
30     return i * f;
31 }
32
33 int int_over_float(int i, int f) {
34     return ((int64_t) i << (2 * Q)) / f;
35 }
36
37
38 int float_minus_int(int f, int i) {
39     return f - (i << Q);
40 }
41
42 int float_over_int(int f, int i) {
43     return f / i;
44 }
45
46 int float_plus_float(int f1, int f2) {
47     return f1 + f2;
48 }
49
50 int float_minus_float(int f1, int f2) {
51     return f1 - f2;
52 }
53
54 int float_times_float(int f1, int f2) {
55     int64_t precise = ((int64_t) f1 * f2) >> Q;
56     return (int) precise;
57 }
58
59 int float_over_float(int f1, int f2) {
60     // int less_precise = f1 / (f2 >> Q);
61     int64_t precise = ((int64_t) f1 << Q) / f2;
62     return (int) precise;
63 }
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

C. 시험 및 평가 내용

- 다음은 make check 및 priority-lifo 수행 결과입니다: 요구 사항에 명시된 모든 테스트 케이스에 대하여 성공적으로 수행하였습니다.

```
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
FAIL tests/threads/priority-donate-one
FAIL tests/threads/priority-donate-multiple
FAIL tests/threads/priority-donate-multiple2
FAIL tests/threads/priority-donate-nest
FAIL tests/threads/priority-donate-sema
FAIL tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-aging
FAIL tests/threads/priority-condvar
FAIL tests/threads/priority-donate-chain
pass tests/threads/mlfqqs-load-1
pass tests/threads/mlfqqs-load-60
pass tests/threads/mlfqqs-load-avg
pass tests/threads/mlfqqs-recent-1
pass tests/threads/mlfqqs-fair-2
pass tests/threads/mlfqqs-fair-20
pass tests/threads/mlfqqs-nice-2
pass tests/threads/mlfqqs-nice-10
pass tests/threads/mlfqqs-block
```

```
cse20161662:~/pintos/src/threads$ pintos run -v -- -q run priority-lifo
qemu -nbd /tmp/0Bt6lS1gyc.dsk -fdb filesys.dsk -m 4 -net none -nographic -monitor null
WARNING: Image format was not specified for '/tmp/0Bt6lS1gyc.dsk' and probing guessed raw.
          Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
          Specify the 'raw' format explicitly to remove the restrictions.
WARNING: Image format was not specified for 'filesys.dsk' and probing guessed raw.
          Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
          Specify the 'raw' format explicitly to remove the restrictions.
Pito hdal
Loading.....
Kernel command line: -q run priority-lifo
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 235,520,000 loops/s.
Boot complete.
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: 48 ticks
Thread: 0 idle ticks, 48 kernel ticks, 0 user ticks
Console: 1557 characters output
Keyboard: 0 keys pressed
Powering off...
```

V. 기타

A. 연구 조원 기여도

- 이규연 50%, 허남규 50%

B. 소감

- 프로젝트 시작 전, Pintos manual, 특히 appendix 부분에서 Pintos 전반적인 구조를 파악하면서 강의 중에 배운 다양한 OS 개념 (threads, semaphore, interrupts) 등의 실제 구현에 관한 다양한 insight를 얻을 수 있었습니다.
 - thread_yield 시 interrupt 상태에서 어셈블리 코드를 통해 context switching을 수행하고 switch_thread() 함수 내에서 “실제 thread 변경”이 일어나는 과정에 관해 직관적인 이해가 가능했습니다.
- Custom 채점 명령어와 개인 알림 명령어, 그리고 tmux를 이용해서 프로젝트 채점을 background에서 돌리고 알림을 받음으로써 생산성을 확연히 높일 수 있었습니다. 그 과정에서 bash script 작성 또한 조금 더 깊이 있게 익힐 수 있었습니다.
- Fixed point number에 관한 연산을 구현하고 디버깅하면서 bit-shift의 활용을 익힐 수 있었고, overflow를 처리해야 하는 상황에서 대처하는 법을 배울 수 있는 좋은 기회였습니다.
- Pintos에서 제공하는 기본 자료구조와 함수 포인터를 이용한 작업 처리 함수 (list_insert_sorted & list_less_func, thread_FOREACH & thread_action_func)를 사용하고 그 설계를 살펴보면서 OOP 개념과 함수형 프로그래밍 개념을 C언어 상에서 깔끔하게 구현하는 방식에 관한 노하우를 얻을 수 있었습니다.
- 내부적인 timer_interrupt 함수를 수정하면서 alarm clock, priority 스케줄링을 실제 구현하면서 강의 시간에 배운 동기화, 스케줄링 기법에 대해 더 깊이 있게 이해할 수 있었습니다.