Pintos Project 3: Threads

담당 교수 : 김영재 교수

조 / 조원 : 202211589 정서영

개발 기간 : 25.11.04 ~ 25.11.24

I. 개발 목표

본 프로젝트는 Pintos 운영체제에서 효율적인 스레드 스케줄링과 스케줄링 관리에 필요한 동기화 메커니즘을 구현하는 것을 목표로 한다. 현재 Pintos는 단순한 라운드 로빈 스케줄러를 사용하고 있으며, timer_sleep()에서 busy waiting으로 인한 비효율성이 존재하고, 스레드의 우선순위를 고려하지 않는 문제가 있다. 따라서 본 프로젝트에서는 Alarm Clock 기능을 개선하여 CPU 자원을 효율적으로 활용하고, Priority Scheduling을 구현하여 우선순위 기반의 스케줄링이 가능하도록 하며, Aging 기법을 통해 starvation 문제를 해결한다. 추가적으로 BSD Scheduler를 구현하여 Multi-Level Feedback Queue 기반의 스케줄러를 제공한다.

II. 개발 범위 및 내용

A. 개발 범위

1. Alarm Clock

현재 timer_sleep() 함수는 스레드를 재우기 위해 thread_yield()를 반복 호출하는 busy waiting 방식을 사용한다. 이는 스레드가 RUNNING과 READY 상태를 반복하면서 실제로는 아무 작업도 하지 않으면서 CPU 시간을 낭비하는 비효율적인 방법이다. Alarm Clock을 개선하면 대기 중인 스레드를 BLOCKED 상태로 전환하여 지정된 시간이 경과할 때까지 스케줄링 대상에서 완전히 제외할 수 있다. 이를 통해 해당 스레드는 CPU를 사용하지 않게 되며, 그 시간 동안 다른 스레드들이 CPU를 온전히 활용할 수 있어 시스템 전체의 효율성이 향상된다.

2. Priority Scheduling

현재 Pintos는 모든 스레드를 동등하게 취급하는 라운드 로빈 스케줄러를 사용한다. Priority Scheduling을 구현하면 각 스레드의 우선순위를 고려하여 높은 우선순위의 스레드가 먼저 실행되도록 할 수 있다. Priority Aging 기법을 추가로 구현하면 낮은 우선순위의 스레드가 무한정 대기하는 starvation 현상을 해결할 수 있다. Ready list에서 대기 중인 스레드의 우선순위를 시간에 비례하여 점진적으로 증가시킴으로써 오랫동안 대기한 스레드도 실행될 기회를 보장받을 수 있다.

3. Advanced Scheduler

BSD Schedular는 Multi-Level Feedback Queue를 기반으로 하는 범용 스케줄러로, 각 스레드의 최근 CPU 사용량과 nice 값을 고려하여 동적으로 우선순위를 조정한다. 이 스케줄러를 구현하면 CPU를 많이 사용한 스레드의 우선순위를 낮추고 I/O 중심의 스레드의 우선순위를 높여서

시스템의 전반적인 공정성과 응답성을 향상시킬 수 있다. 또한 Fixed-point 연산을 통해 부동소수점 연산 없이도 정밀한 우선순위 계산이 가능하다.

B. 개발 내용

1. Blocked 상태의 스레드를 깨우는 방법

   Blocked 상태의 스레드를 깨우기 위해서는 잠든 스레드들을 관리하는 별도의 큐(sleep_list)가 필요하다. 스레드가 timer_sleep()을 호출하면 해당 스레드를 sleep_list에 넣고, 현재 시간에 대기 시간을 더한 깨어날 시간(wake_up_time)을 계산하여 스레드 구조체에 저장하고, thread_block() 함수를 호출하여 스레드를 BLOCKED 상태로 전환한다.

   이후 매 tick마다 호출되는 timer_interrupt() 함수에서 sleep_list를 순회하며 현재 시간이 각 스레드의 wake_up_time 이상인지 확인한다. 조건을 만족하는 스레드는 sleep_list에서 제거하고 thread_unblock() 함수를 호출하여 READY 상태로 전환한다.

2. Priority Scheduling에 따라, Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 처리하는 방법

   Priority Scheduling에서는 항상 가장 높은 우선순위를 가진 스레드가 CPU를 점유해야 한다. Ready list에 현재 실행 중인 스레드보다 높은 우선순위를 가진 새로운 스레드가 들어오면 즉시 선점이 발생해야 한다.

   따라서 새로운 스레드가 ready_list에 삽입될 때 해당 스레드의 우선순위와 현재 실행 중인 스레드의 우선 순위를 비교한다. 새로운 스레드의 우선순위가 더 높다면 현재 실행 중인 스레드는 즉시 thread_yield()를 호출하여 CPU를 양보한다. 호출된 thread_yield()는 현재 스레드를 ready_list에 삽입하고 schedule()을 호출하여 다음 실행할 스레드를 선택한다. schedule()은 next_thread_to_run()을 통해 ready_list에서 가장 높은 우선순위의 스레드를 선택한다.

   이를 위해 ready_list는 항상 우선순위 기준으로 내림차순 정렬된 상태를 유지해야 한다. 스레드가 삽입되는 모든 시점에서 우선순위에 맞는 적절한 위치에 삽입하고, 우선순위가 변경될 때도 리스트 순서를 재정렬한다.

3. Advanced Scheduler에서 priority 계산에 필요한 각 요소

   (1) priority : 스레드의 우선순위로, 0(PRI_MIN)부터 63(PRI_MAX)까지의 범위를 가진다. priority의 경우 다음 공식으로 계산되고, 모든 스레드의 우선순위는 4 tick마다 다시 계산된다.

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

(2) nice : nice 값은 -20부터 20까지의 범위를 가진다. 양수 값은 우선순위를 낮추어 다른 스레드에게 CPU를 양보하고, 음수 값은 우선순위를 높인다. 0은 우선순위에 영향을 주지 않는다. 초기값은 0이며, 자식 스레드는 부모의 값을 상속받는다.

(3) recent_cpu : 스레드가 최근에 사용한 CPU 시간의 가중 평균을 나타내는 값이다. RUNNING 상태의 스레드는 recent_cpu 값이 매 tick마다 1씩 증가하며, 모든 스레드의 recent_cpu는 매 초마다 다음 공식으로 업데이트된다. 초기값은 0이며, 자식 스레드는 부모의 값을 상속받는다.

- *recent_cpu = (2 * load_avg) / (2 * load_avg + 1) * recent_cpu + nice*

(4) load_avg : 시스템 전체의 평균 부하를 나타내는 값이다. 시스템 부팅 시 0으로 초기화되며, 매 초마다 다음 공식으로 업데이트된다. 여기서 ready_threads는 idle 스레드를 제외한 READY 또는 RUNNING 상태의 스레드 개수이다.

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

III. 추진 일정 및 개발 방법

A. 추진 일정

기간	추진 내용
11/4 ~ 11/7	Pintos 소스 코드 분석 및 명세서 검토
11/8 ~ 11/12	Alarm Clock 구현 및 테스트
11/13 ~ 11/17	Priority Scheduling 구현 및 테스트
11/18 ~ 11/20	Priority Aging 구현 및 테스트
11/21 ~ 11/23	Advanced Schedular 구현 및 테스트
11/23 ~ 11/24	전체 테스트 검증 및 문서 작성

B. 개발 방법

1. Alarm Clock

먼저 threads/thread.h의 thread 구조체에 wake_up_time을 추가하여 스레드가 깨어나야 할 시간을 저장한다. device/timer.c에 전역 변수로 sleep_list를 선언하고, timer_init() 함수에서 list_init()을 호출하여 sleep_list를 초기화한다.

devices/timer.c의 timer_sleep() 함수에서 현재 스레드를 thread_current()로 가져와서 해당 스레드의 wake_up_time을 설정한다. 이후 list_push_back()으로 sleep_list에 현재 스레드의 elem을 추가하고, thread_block()을 호출해 BLOCKED 상태로 만든다.

thread_intrerrupt() 함수에서는 sleep_list를 순회하면서, 현재 요소를 struct thread로 변환하고 해당 스레드에 저장된 wake_up_time을 확인한다. 해당 스레드가 깨어나야 하는 시간을 만족했다면, list_remove()로 sleep_list에서 제거하고 thread_unblock()을 호출하여 READY 상태로 변경한다.

2. Priority Scheduling

threads/thread.h에 thread_priority_compare()와 thread_insert_ready_list() 함수의 프로토타입을 선언한다. threads/thread.c에 thread_priority_compare() 함수를 작성하여 두 스레드의 우선순위를 비교하도록 하고, thread_insert_ready_list() 함수를 작성하여 ready_list에 우선순위 순서로 스레드를 삽입하도록 한다.

다음으로 thread_create() 함수를 수정하여 새 스레드 생성 후 우선순위를 비교하고 필요 시 thread_yield()를 호출하여 선점이 발생하도록 한다. thread_yield()와 thread_unblock() 함수에서는 list_push_back() 대신 thread_insert_ready_list()를 사용하여 ready_list에 우선순위 순서대로 삽입될 수 있도록 한다. 또한 thread_set_priority() 함수를 수정하여 현재 스레드의 우선순위를 변경한 후, ready_list에 있는 첫 번째 스레드의 우선순위와 비교하여 필요 시 thread_yield()를 호출하여 선점이 발생하도록 한다.

threads/synch.c의 sema_up() 함수도 수정하여 waiters 리스트를 list_sort()로 정렬한 후 가장 높은 우선순위 스레드를 깨우도록 하고, 현재 실행 중인 스레드보다 깨어난 스레드의 우선순위가 높을 수 있으므로 thread_yield()를 호출하도록 한다.

Priority Aging 기법을 구현하기 위해서 먼저 명세서에 따라 thread_prior_aging 플래그를 추가하고, thread_prior_aging이 true인 경우 thread_aging() 함수를 호출하도록 한다. 이후 threads/thread.c에 thread_aging() 함수를 작성하여 해당 함수에서 ready_list를 순회하며 각 스레드의 우선순위를 1씩 증가시킨다.

3. Advanced Scheduler

먼저 threads/thread.h에 고정 소수점 연산을 위한 매크로들을 정의한다. F를 (1<<14)로 정의하고, INT_TO_FP, FP_TO_INT, FP_TO_INT_ROUND, ADD_FP, SUB_FP, ADD_MIX, SUB_MIX, MULT_FP, MULT_MIX, DIV_FP, DIV_MIX 매크로를 구현한다.

threads/thread.h의 thread 구조체에 nice와 recent_cpu를 추가한다. threads/thread.c에 전역 변수로 load_avg를 선언하고, load_avg 계산을 위한 LOAD_AVG_NUMERATOR(59)와 LOAD_AVG_DENOMINATOR(60) 상수를 정의한다. thread_init() 함수에서 load_avg를 0으로 초기화하고, init_thread() 함수를 수정하여 새 스레드의 nice와 recent_cpu를 초기화한다. running_thread가 initial_thread인 경우 각각 0으로 설정하고, 그렇지 않으면 부모 스레드의 값을 상속받도록 한다.

thread_tick() 함수를 수정하여 BSD Scheduler 로직을 추가한다. thread_mlfqs가 true인 경우에만 동작하도록 하고, 매 tick마다 recent_cpu를 1 증가시킨다. 매 초마다 load_avg와 모든 스레드의 recent_cpu를 재계산하도록 하고, 4 tick마다 모든 스레드의 우선순위를 재계산한다.

thread_get_nice()은 현재 스레드의 nice 값을 반환한다. thread_set_nice()는 현재 스레드의 nice 값을 설정하고, calculate_priority() 함수를 호출하여 우선순위를 재계산한다. 그리고 ready_list가 비어있지 않고, 첫 번째 스레드의 우선순위가 현재 스레드보다 높으면 thread_yield()를 호출한다.

thread_get_recent_cpu()는 현재 스레드의 recent_cpu에 100을 곱한 후 반올림하여 반환한다. 추가로 recent_cpu 계산에 사용하기 위해 calculate_recent_cpu()와 update_recent_cpu() 함수를 구현하는데, calculate_recent_cpu() 함수는 recent_cpu 공식을 사용하여 해당 값을 계산하고, update_recent_cpu() 함수는 all_list를 순회하면서 각 스레드에 대해 calculate_recent_cpu() 함수를 호출한다.

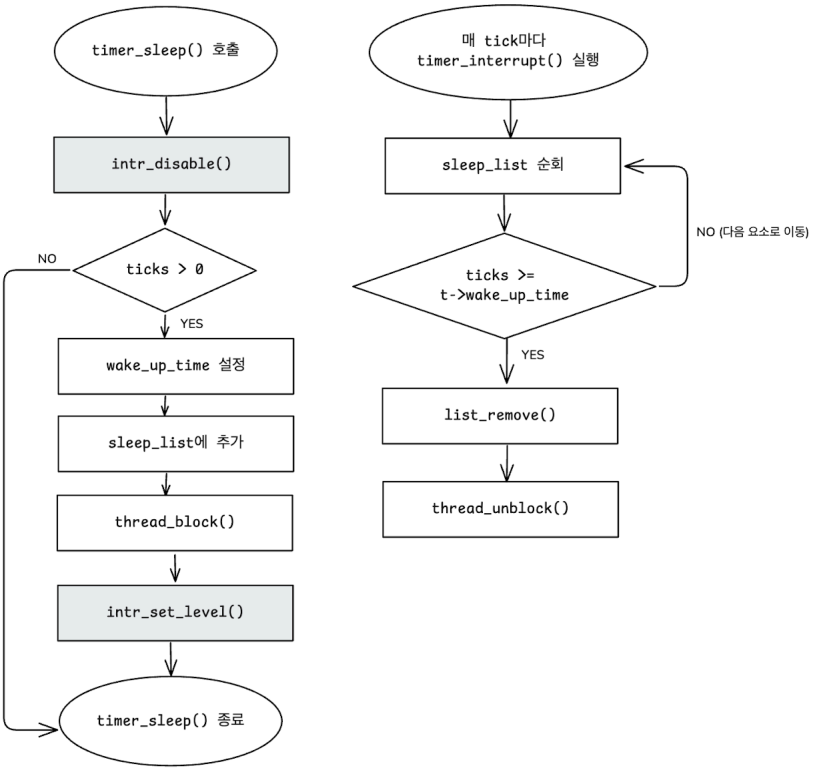
thread_get_load_avg()는 load_avg에 100을 곱한 후 반올림하여 반환한다. 추가로 load_avg 계산에 사용하기 위해 update_load_avg()와 get_ready_threads() 함수를 구현한다. update_load_avg() 함수는 load_avg 공식을 사용하여 해당 값을 계산하고, get_ready_threads() 함수는 READY 또는 RUNNING 상태인 스레드의 개수를 구하도록 한다.

마지막으로 우선순위 계산을 위해서는 calculate_priority() 함수와 update_priorities() 함수를 구현하는데, calculate_priority() 함수는 priority 공식을 사용하여 우선순위 값을 계산하고, update_priorities() 함수는 all_list를 순회하면서 각 스레드에 대해 calculate_priority() 함수를 호출한다.

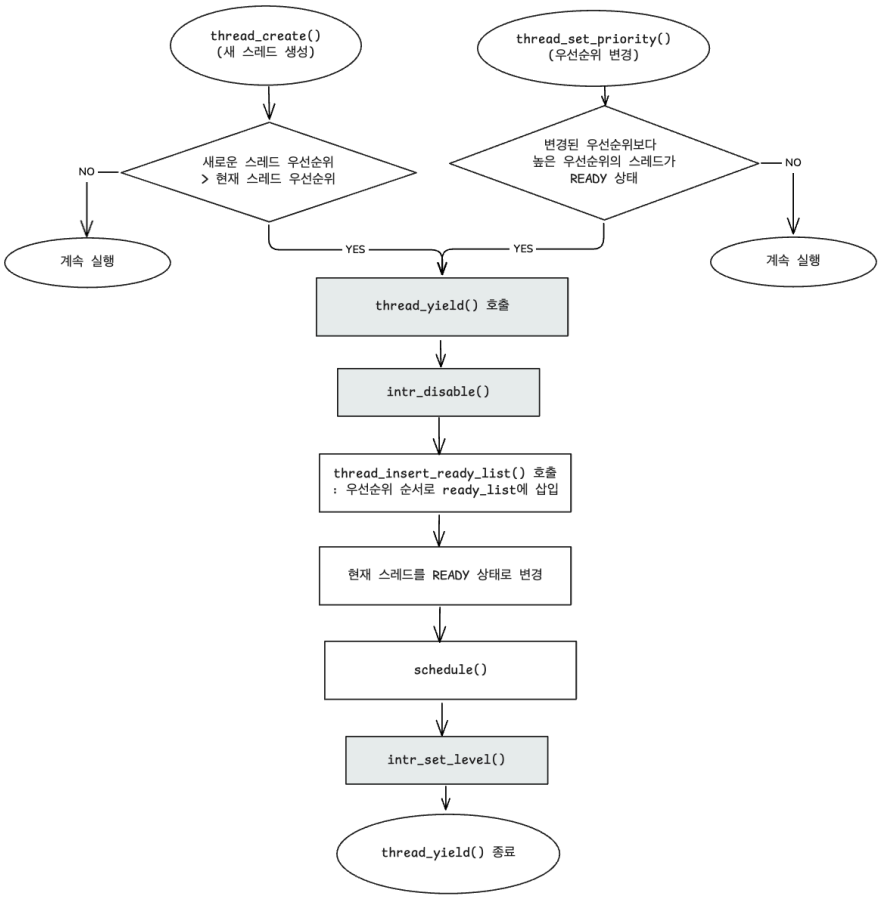
IV. 연구 결과

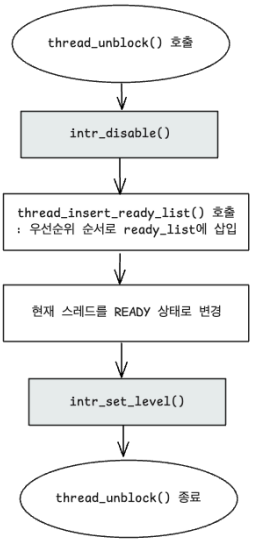
A. Flow Chart

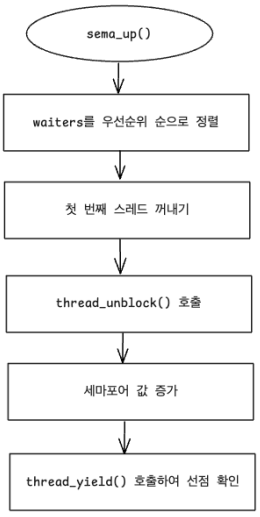
1. Alarm Clock

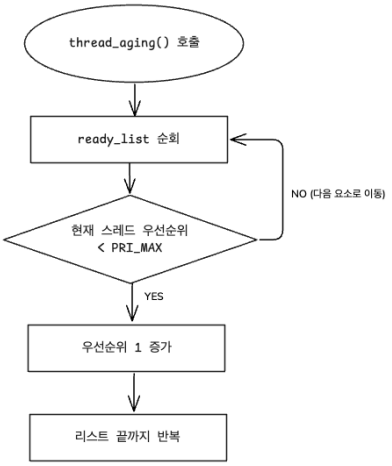


2. Priority Scheduling


thread_create()
(새 스레드 생성)
thread_set_priority()
(우선순위 변경)
새로운 스레드 우선순위
> 현재 스레드 우선순위
NO
계속 실행
변경된 우선순위보다
높은 우선순위의 스레드가
READY 상태
NO
계속 실행
YES
YES
thread_yield() 호출
intr_disable()
thread_insert_ready_list() 호출
: 우선순위 순서로 ready_list에 삽입
현재 스레드를 READY 상태로 변경
schedule()
intr_set_level()
thread_yield() 종료


thread_unblock() 호출
intr_disable()
thread_insert_ready_list() 호출
: 우선순위 순서로 ready_list에 삽입
현재 스레드를 READY 상태로 변경
intr_set_level()
thread_unblock() 종료


sema_up()
waiters를 우선순위 순으로 정렬
첫 번째 스레드 꺼내기
thread_unblock() 호출
세마포어 값 증가
thread_yield() 호출하여 선점 확인


thread_aging() 호출
ready_list 순회
NO (다음 요소로 이동)
현재 스레드 우선순위
< PRI_MAX
YES
우선순위 1 증가
리스트 끝까지 반복

B. 제작 내용

1. Alarm Clock

먼저 threads/thread.h의 struct thread에 wake_up_time 멤버 변수를 추가했다. 이는 int64_t 타입으로 선언하여 스레드가 깨어나야 할 시간을 저장한다.

```
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. */

    int64_t wake_up_time;
```

다음으로, devices/timer.c에 잠든 스레드들을 관리할 전역 리스트 sleep_list를 선언하고, timer_init() 함수에서 이를 초기화했다.

```
static struct list sleep_list;
```

```
void
timer_init (void)
{
  list_init(&sleep_list);
  pit_configure_channel (0, 2, TIMER_FREQ);
  intr_register_ext (0x20, timer_interrupt, "8254 Timer");
}
```

thread_sleep() 함수에서 기존의 while 루프와 thread_yield() 반복 호출을 제거했다. 인터럽트 컨텍스트가 아닌지 확인하는 ASSERT문을 추가하여 안전성을 보장하고자 했고, 인터럽트를 비활성화한 상태에서 현재 스레드의 wake_up_time을 start + ticks로 설정하고 sleep_list에 스레드를 추가한 후 thread_block()을 호출하여 BLOCKED 상태로 전환한다. 작업이 끝나면 인터럽트 레벨을 복원한다.

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

  ASSERT (intr_get_level () == INTR_ON);

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

  ASSERT (!intr_context());

  old_level = intr_disable();
  if (ticks > 0) {
    cur->wake_up_time = start + ticks;
    list_push_back(&sleep_list, &cur->elem);
    thread_block();
  }

  intr_set_level(old_level);
}
```

thread_interrupt() 함수에서 매 tick마다 sleep_list를 확인하고 깨울 스레드를 처리하도록 했다. 먼저 리스트를 순회하면서 각 스레드의 wake_up_time을 현재 ticks와 비교한다. 순회 중 리스트를 수정하므로 다음 요소를 미리 next 변수에 저장해두도록 구성했고, 조건을 만족하는 스레드는 list_remove()로 제거하고 thread_unblock()을 호출하여 READY 상태로 전환한다.

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

  if (!list_empty(&sleep_list)) {
    struct list_elem *e = list_begin(&sleep_list);

    while (e != list_end(&sleep_list)) {
      struct thread *t = list_entry(e, struct thread, elem);
      struct list_elem *next = list_next(e);

      if (ticks >= t->wake_up_time) {
        list_remove(e);
        thread_unblock(t);
      }
      e = next;
    }
  }

  thread_tick ();
}
```

2. Priority Scheduling

먼저 threads/thread.h에 thread_priority_compare() 함수와 thread_insert_ready_list() 함수의 프로토타입을 선언하고, threads/thread.c에 해당 함수들의 내용을 작성했다.

thread_priority_compare() 함수는 두 개의 list_elem 포인터를 받아 각각 thread 구조체로 변환한 후 priority 값을 비교하여 첫 번째 스레드의 우선순위가 더 높으면 true를 반환한다. thread_insert_ready_list() 함수는 struct thread 포인터를 받아 list_insert_ordered()를 사용하여 ready_list에 우선순위 순서대로 스레드를 삽입한다. 이때 thread_priority_compare() 함수를 비교 함수로 사용했다.

```
bool thread_priority_compare (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED);
void thread_insert_ready_list (struct thread *t);
```

```
/* 스레드 우선순위 비교 함수 */
bool
thread_priority_compare (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
{
  struct thread *ta = list_entry(a, struct thread, elem);
  struct thread *tb = list_entry(b, struct thread, elem);
  return ta->priority > tb->priority;
}

/* ready_list에 스레드 우선순위가 높은 순서로 삽입 */
void
thread_insert_ready_list (struct thread *t)
{
  list_insert_ordered(&ready_list, &t->elem, thread_priority_compare, NULL);
}
```

thread_create() 함수를 수정하여 새로운 스레드의 우선순위가 현재 실행 중인 스레드보다 높은지 확인하고, 높다면 thread_yield()를 호출하여 현재 스레드는 READY 상태로 변경되도록 했다.

```
  /* Add to run queue. */
  thread_unblock (t);

  // 새로운 스레드의 우선순위가 더 높다면, 현재 스레드는 ready 상태로 변경
  if (priority > thread_current()->priority)
    thread_yield();

  return tid;
}
```

thread_yield() 함수에서 현재 스레드를 ready_list에 삽입할 때 우선순위 순서를 유지하도록 했다. ready_list의 끝에 스레드를 넣는 list_push_back() 대신 앞서 작성한 thread_insert_ready_list() 함수를 사용하여 우선순위에 맞는 적절한 위치에 삽입한다.

```
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)
    thread_insert_ready_list(cur);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

thread_unblock() 함수도 이와 동일하게 수정했다. BLOCKED 상태의 스레드가 다시 실행 가능한 상태가 될 때, list_push_back() 대신 thread_insert_ready_list()를 사용해 ready_list에 우선순위 순서로 삽입되도록 했다.

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  thread_insert_ready_list(t);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

thread_set_priority() 함수를 수정하여 우선순위 변경 후 선점이 필요한지 확인하도록 했다.

먼저 현재 스레드의 우선순위를 새로운 값으로 설정한다. 그 다음 인터럽트를 비활성화하고 ready_list가 비어있지 않은지 확인한다. ready_list가 비어있지 않다면 list_front()를 사용하여 ready_list의 첫 번째 요소를 가져온다. ready_list는 항상 우선순위 순서로 정렬되어 있으므로, 첫 번째 스레드가 ready_list에서 가장 높은 우선순위를 가진 스레드이다. 만약 이 스레드의 우선순위가 방금 설정한 새로운 우선순위보다 높다면 현재 스레드는 더 이상 가장 높은 우선순위를 가지지 않는다는 의미이므로 thread_yield()를 호출하여 CPU를 양보한다. 해당 작업이 끝나면 인터럽트 레벨을 복원한다.

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

  enum intr_level old_level = intr_disable ();

  // 더 높은 우선순위의 스레드가 있다면 양보
  if (!list_empty(&ready_list)) {
    struct thread* highest_ready = list_entry(list_front(&ready_list), struct thread, elem);

    if (highest_ready->priority > new_priority)
      thread_yield ();
  }

  intr_set_level (old_level);
}
```

threads/synch.c에서 sema_up() 함수는 세마포어를 해제하는 함수로, 대기 중인 스레드 중 하나를 깨워야 한다. waiters 리스트가 비어있지 않다면 그중에서 가장 높은 우선순위를 가진 스레드를 선택해야 한다. 이를 위해 먼저 list_sort()를 호출하여 waiters 리스트 전체를 우선순위 순서로 정렬한다. 이때 thread_priority_compare() 함수를 비교 함수로 사용했다.

정렬된 리스트의 첫 번째 요소가 가장 높은 우선순위를 가진 스레드이므로, list_pop_front()로 이 요소를 꺼낸다. 꺼낸 list_elem을 list_entry()로 struct thread로 변환하여 thread_unblock()을 호출하면 해당 스레드가 READY 상태로 전환되어 다시 실행될 수 있다.

이후 세마포어 값을 1 증가시킨 후 thread_yield()를 호출한다. 이는 방금 깨어난 스레드의 우선순위가 현재 실행 중인 스레드보다 높을 수 있기 때문에 스케줄러에게 다시 스케줄링할 기회를 주기 위함이다.

```
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters)) {
    // waiters 리스트를 우선순위 순으로 정렬
    list_sort(&sema->waiters, thread_priority_compare, NULL);
    // 가장 높은 우선순위 스레드를 꺼내서 깨움
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  }
  sema->value++;

  // 현재 스레드보다 높은 우선순위 스레드가 깨어났으면 양보할 수 있도록 함
  thread_yield();
  intr_set_level (old_level);
}
```

Aging 기법을 구현하기 위해 threads/thread.h에 thread_aging() 함수 프로토타입을 작성하고, threads/thread.c에 함수 내용을 작성했다. 이 함수는 ready_list를 순회하면서 각 스레드의 우선순위를 1씩 증가시킨다. 단, 우선순위가 이미 PRI_MAX(63)에 도달한 스레드는 더 이상 증가시키지 않도록 했다.

```
void thread_aging (void);
```

```
void thread_aging (void)
{
  struct list_elem *e = list_begin (&ready_list);
  while (e != list_end (&ready_list)) {
    struct thread *t = list_entry (e, struct thread, elem);
    if (t->priority < PRI_MAX) {
      t->priority++;
    }
    e = list_next (e);
  }
}
```

3. Advanced Scheduler

먼저 threads/thread.h에 고정 소수점 연산을 위한 매크로들을 정의했다. F를 (1<<14)로 정의하여 고정 소수점 변환 상수로 사용한다. INT_TO_FP(n)는 정수를 고정 소수점으로 변환하고, FP_TO_INT(x)는 고정 소수점을 정수로 변환한다. FP_TO_INT_ROUND(x)는 반올림하여 정수로 변환한다. ADD_FP, SUB_FP, ADD_MIX, SUB_MIX는 고정 소수점 덧셈과 뺄셈을, MULT_FP, MULT_MIX, DIV_FP, DIV_MIX는 고정 소수점 곱셈과 나눗셈을 수행한다.

```
#define F (1 << 14)
#define INT_TO_FP(n) ((n) * F)
#define FP_TO_INT(x) ((x) / F)
#define FP_TO_INT_ROUND(x) ((x) >= 0 ? ((x) + F/2) / F : ((x) - F/2) / F)
#define ADD_FP(x,y) ((x) + (y))
#define SUB_FP(x,y) ((x) - (y))
#define ADD_MIX(x,n) ((x) + (n) * F)
#define SUB_MIX(x,n) ((x) - (n) * F)
#define MULT_FP(x,y) (((int64_t)(x)) * (y) / F)
#define MULT_MIX(x,n) ((x) * (n))
#define DIV_FP(x,y) (((int64_t)(x)) * F / (y))
#define DIV_MIX(x,n) ((x) / (n))
```

threads/thread.h의 struct thread에 nice와 recent_cpu를 int 타입 멤버 변수로 추가했다. threads/thread.c의 init_thread() 함수에서 initial_thread인 경우 nice와 recent_cpu를 각각 0으로 초기화했고, 그렇지 않으면 부모 스레드의 nice와 recent_cpu 값을 모두 상속받도록 했다.

```
int64_t wake_up_time;
int nice;
int recent_cpu;
```

```
if (running_thread() == initial_thread) {
  t->nice = 0;
  t->recent_cpu = 0;
} else {
  t->nice = thread_current()->nice;
  t->recent_cpu = thread_current()->recent_cpu;
}
```

threads/thread.c에 전역 변수로 static int load_avg를 추가했고, load_avg 공식에 따라 계산에 사용하기 위한 상수로 LOAD_AVG_NUMERATOR는 59로, LOAD_AVG_DENOMINATOR는 60으로 정의했다. 그리고 thread_init() 함수에서 load_avg를 0으로 초기화했다.

```
static int load_avg;

#define LOAD_AVG_NUMERATOR 59
#define LOAD_AVG_DENOMINATOR 60
```

```
void
thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);

  load_avg = 0;

  lock_init (&tid_lock);
  list_init (&ready_list);
  list_init (&all_list);
```

thread_tick()함수에서 thread_mlfqs가 true인 경우에 BSD Scheduler 관련 계산을 수행하도록 했다. 매 tick마다 현재 실행 중인 스레드가 idle 스레드가 아니라면 recent_cpu를 1 증가시킨다. 매 초마다 (timer_ticks() % TIMER_FREQ == 0) update_load_avg()와 update_recent_cpu()를 호출하여 load_avg와 모든 스레드의 recent_cpu를 재계산한다. 그리고 4 tick마다 (timer_ticks() % 4 == 0) update_priorities()를 호출하여 모든 스레드의 우선순위를 재계산한다.

```
if (thread_mlfqs) {
  if (t != idle_thread)
    t->recent_cpu = ADD_MIX(t->recent_cpu, 1);

  if (timer_ticks () % TIMER_FREQ == 0) {
    update_load_avg ();
    update_recent_cpu ();
  }

  if (timer_ticks () % 4 == 0)
    update_priorities ();
}
```

nice 값 관련 함수들은 다음과 같다. 먼저 thread_get_nice()는 단순히 현재 스레드의 nice 값을 반환한다. thread_set_nice() 함수는 인터럽트를 비활성화한 상태에서 현재 스레드의 nice 값을 설정하고, calculate_priority()를 호출하여 우선순위를 재계산한다. 이후 ready_list가 비어있지 않고, ready_list의 첫 번째 스레드의 우선순위가 현재 스레드보다 높으면 thread_yield()를 호출하여 선점이 발생하도록 한다.

```c
/* Sets the current thread's nice value to NICE. */
void
thread_set_nice (int nice)
{
  enum intr_level old_level = intr_disable ();

  struct thread *current = thread_current ();
  current->nice = nice;

  calculate_priority (current);

  if (!list_empty (&ready_list))
  {
    struct thread *highest_thread = list_entry (list_front (&ready_list), struct thread, elem);
    if (highest_thread->priority > current->priority)
      thread_yield ();
  }

  intr_set_level (old_level);
}

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

recent_cpu 값 관련 함수들은 다음과 같다. 먼저 thread_get_recent_cpu() 함수는 현재 스레드의 recent_cpu에 100을 곱한 후 FP_TO_INT_ROUND() 매크로를 사용하여 반올림하여 정수로 반환한다. calculate_recent_cpu() 함수는 idle_thread가 아닌 스레드에 대해 *recent_cpu = (2 * load_avg) / (2 * load_avg + 1) * recent_cpu + nice* 공식을 사용하여 recent_cpu를 계산한다. 먼저 load_avg에 2를 곱한 값(load_avg2)을 구하고, load_avg2를 (load_avg2+1)로 나눈 계수(coef)를 구한다. 이 계수를 현재 recent_cpu에 곱하여 가중치를 적용한 후(weighted_cpu), nice를 더하여 최종 recent_cpu 값을 얻는다. 그리고 update_recent_cpu() 함수를 호출하여 all_list를 순회하면서 각 스레드에 대해 calculate_recent_cpu()를 호출하도록 하였다.

```c
int
thread_get_recent_cpu (void)
{
  enum intr_level old_level = intr_disable();
  int recent_cpu = FP_TO_INT_ROUND(MULT_MIX(thread_current()->recent_cpu, 100));
  intr_set_level(old_level);
  return recent_cpu;
}
```

```
static void
calculate_recent_cpu (struct thread *t)
{
  if (t != idle_thread) {
    int load_avg2 = MULT_FP(load_avg, INT_TO_FP(2));
    int coef = DIV_FP(load_avg2, ADD_FP(load_avg2, INT_TO_FP(1)));
    int weighted_cpu = MULT_FP(coef, t->recent_cpu);
    t->recent_cpu = ADD_FP(weighted_cpu, INT_TO_FP(t->nice));
  }
}

static void
update_recent_cpu (void)
{
  struct list_elem *e;
  for (e = list_begin (&all_list); e != list_end (&all_list); e = list_next (e)) {
    struct thread *t = list_entry (e, struct thread, allelem);
    calculate_recent_cpu (t);
  }
}
```

load_avg 값 관련 함수들은 다음과 같다. 먼저 thread_get_load_avg() 함수는 load_avg에 100을 곱한 후 반올림하여 정수로 반환한다. update_load_avg() 함수는 get_ready_threads()를 호출하여 ready_threads를 구한 후 *load_avg = (59/60) * load_avg + (1/60) * ready_threads* 공식을 사용하여 load_avg를 업데이트한다. 여기서 get_ready_threads() 함수는 ready_list의 크기를 list_size()로 계산하고, 현재 실행 중인 스레드가 idle_thread가 아니면 1을 추가하여 READY 또는 RUNNING 상태인 스레드의 개수를 반환하는 함수이다.

```
int
thread_get_load_avg (void)
{
  enum intr_level old_level = intr_disable();
  int load = FP_TO_INT_ROUND(MULT_MIX(load_avg, 100));
  intr_set_level(old_level);
  return load;
}
```

```
static void
update_load_avg (void)
{
  int ready_threads = get_ready_threads();
  load_avg = DIV_FP(MULT_FP(INT_TO_FP(LOAD_AVG_NUMERATOR), load_avg) + INT_TO_FP(ready_threads), INT_TO_FP(LOAD_AVG_DENOMINATOR));
}
```

```
static int
get_ready_threads (void)
{
  int ready_threads = list_size (&ready_list);
  if (thread_current () != idle_thread)
    ready_threads++;
  return ready_threads;
}
```

priority 값 관련 함수들은 다음과 같다.

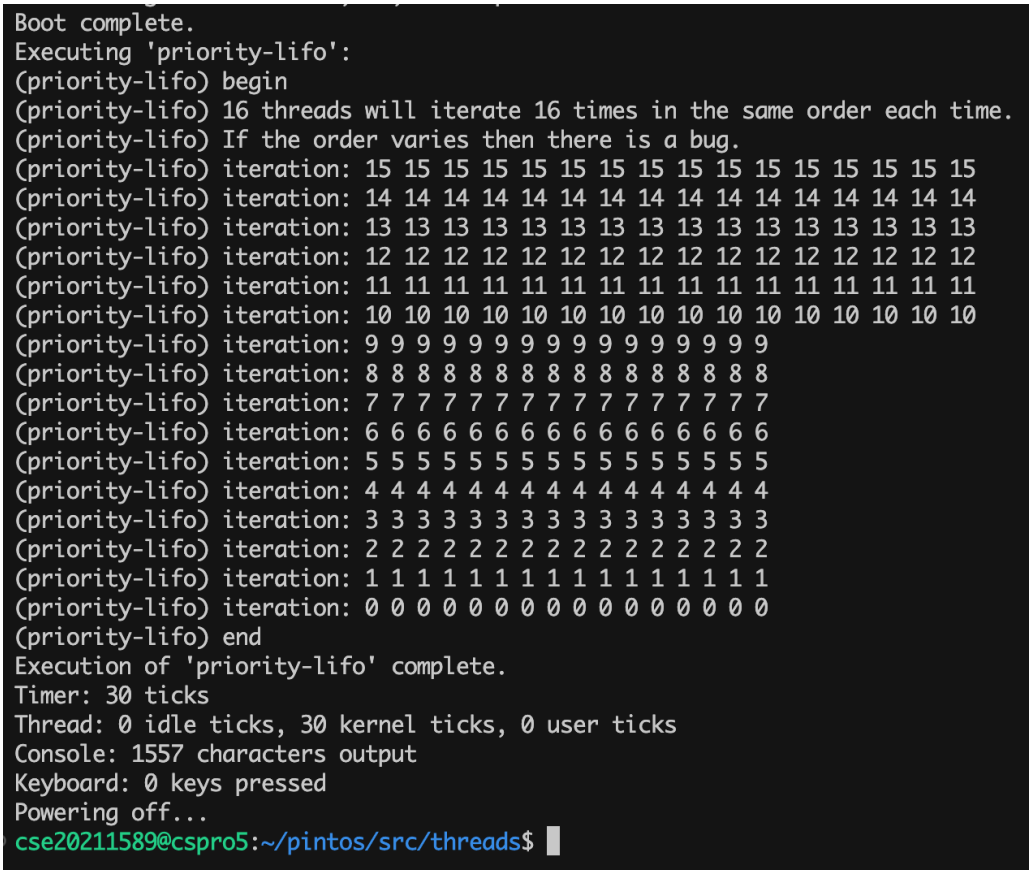
먼저 calculate_priority() 함수는 idle_thread가 아닌 스레드에 대해 *priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)* 공식을 사용하여 우선순위를 계산한다. recent_cpu를 4로 나눈 값을 정수로 변환하고, PRI_MAX에서 이 값과 nice에 2를 곱한 값을 빼서 새로운 우선순위를 구한다. 계산 결과가 PRI_MIN보다 작으면 PRI_MIN으로, PRI_MAX보다 크면 PRI_MAX로 제한한다.

update_priorities() 함수는 인터럽트가 비활성화 상태인지 확인한 후, all_list를 순회하면서 각 스레드에 대해 calculate_priority() 함수를 호출한다. 모든 우선순위 재계산이 끝난 후 ready_list가 비어있지 않고 현재 스레드보다 더 높은 우선순위의 스레드가 있으면 intr_yield_on_return()을 호출하여 인터럽트 복귀 시 스케줄링이 발생하도록 한다.

```
static void
calculate_priority (struct thread *t)
{
  if (t != idle_thread) {
    int recent_cpu_term = FP_TO_INT(DIV_MIX(t->recent_cpu, 4));
    int new_priority = PRI_MAX - recent_cpu_term - (t->nice * 2);

    if (new_priority < PRI_MIN) new_priority = PRI_MIN;
    if (new_priority > PRI_MAX) new_priority = PRI_MAX;

    t->priority = new_priority;
  }
}

static void
update_priorities (void)
{
  ASSERT (intr_get_level () == INTR_OFF);

  struct list_elem *e;
  for (e = list_begin (&all_list); e != list_end (&all_list); e = list_next (e)) {
    struct thread *t = list_entry (e, struct thread, allelem);
    calculate_priority (t);
  }

  if (!list_empty (&ready_list)) {
    struct thread *cur = thread_current ();
    struct thread *highest_ready = list_entry (list_front (&ready_list), struct thread, elem);
    if (highest_ready->priority > cur->priority)
      intr_yield_on_return ();
  }
}
```

C. 시험 및 평가 내용

다음은 src/threads에서 'pintos -v -- -q run priority-lifo' 명령어를 실행시켰을 때의 결과를 캡처한 것이다. priority-lifo 테스트는 동일한 우선순위를 가진 스레드 여러 개를 생성한 후 이들이 라운드 로빈 방식으로 일관되게 실행되도록 설계되었다. 16개의 스레드를 생성하며, 각 스레드는 16번 반복 실행된다. 실행 결과를 보면 0부터 15까지 지정된 16개의 스레드가 LIFO 순서로 16번 반복적으로 실행되는 것을 확인할 수 있었다.



다음은 src/threads에서 'make check'를 실행한 결과로, 이번 프로젝트의 평가 기준에 해당하는 모든 테스트 케이스에 대해 pass한 것을 확인할 수 있다.