Pintos Project1 Design Report

20180038 박형규
20180480 성창환

1. Alarm Clock

1.1 Analysis of the current implementation

우선 naïve pintos의 alarm은 busy wait로 구현되어 있다.

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

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

Timer_sleep이 호출되면 우선 호출된 시점의 tick을 저장한다. 그리고 timer_sleep이 호출되고 몇tick이 지났는지를 timer_elapsed를 이용해 계산하고 ticks보다 작다면 thread_yield 함수를 이용해서 현재 쓰레드의 cpu점유를 반환하고 ready_list의 맨 뒤로 이동하게 된다. 이 구현은 스레드가 완전히 멈추어 있다가 특정 시간에 다시 작동하는 것이 아니라(스케쥴링에서 제외되지 않는다)는 문제점이 있다.

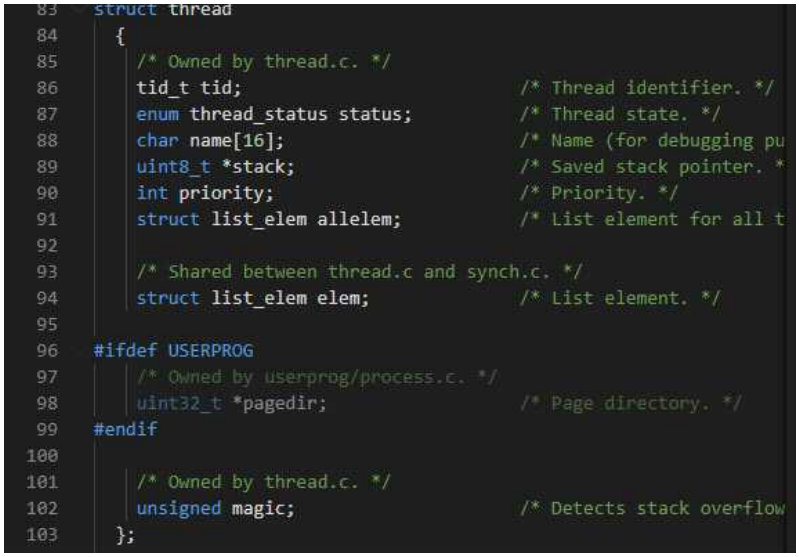
1.2 Solution

Busy waiting 방식이 아닌 sleep/wakeup 방식으로 변경하면 된다. 구체적으로는 스레드 자체를 정지시키고, 스케쥴링이 되지 않도록 처리한 뒤, 특정 tick에 다시 스케쥴링될 수 있도록 하는 것이다. 즉, 스레드를 ready 상태로 놔두지 않고, block상태로 만들어서 별도의 리스트로 보관하고 있다가, 8254칩의 매 tick 신호에 맞추어서 특정 tick에 도달하면 스레드를 다시 ready상태로 변경하고, 스케쥴링 리스트에 넣어주면 된다.

2. priority scheduling

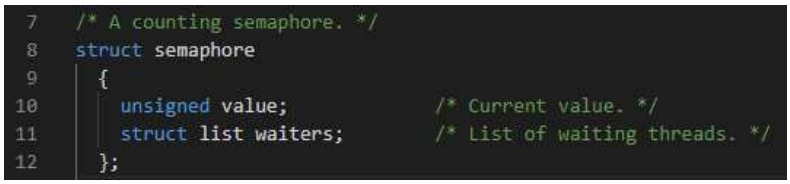
2.1 Analysis of the current thread system and synchronization

먼저 thread.h와 synch.h에 나와있는 구조체들부터 살펴보자. 다음은 thread.h에 있는 thread 구조체이다.



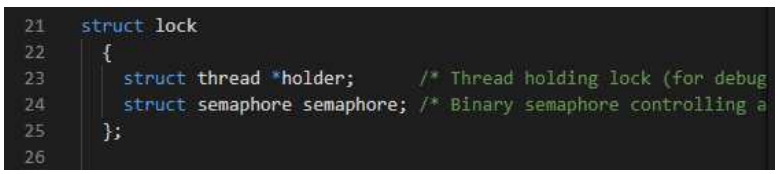
```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging pu
    uint8_t *stack;                     /* Saved stack pointer.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

    /* 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. */
    unsigned magic;                     /* Detects stack overflow
  };
```

thread구조체 내부에는 가장 먼저 thread의 상태 정보를 저장하는 thread_status가 있다. 이는 thread.h 윗부분에서 보이는 것과 같이 THREAD_RUNNING, THREAD_READY, THREAD_BLOCKED, THREAD_DYING의 상태를 가질 수 있다. 다음 주목해야 할 정보로는 int형 변수인 priority이다. 이는 thread의 우선순위를 나타내는데, 이 변수를 통해 우리는 priority scheduling을 구현 할 것이다. 그리고 allelem과 elem은 각각 list_elem형의 변수로써 모든 thread list와 list element를 저장한다. 다음으로는 synch.h에 define 되어 있는 semaphore struct이다.



```
/* A counting semaphore. */
struct semaphore
  {
    unsigned value;             /* Current value. */
    struct list waiters;        /* List of waiting threads. */
  };
```

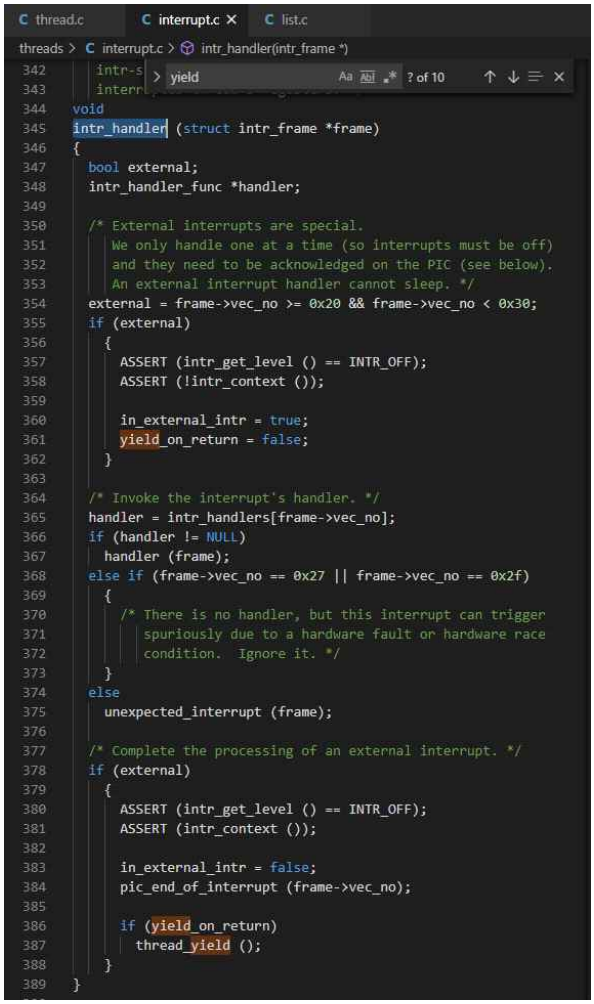
semaphore struct는 value와 list waiters의 두 가지 변수를 가진다. 먼저 value는 해당 sema를 사용하는 thread의 수를 나타낸다. 그리고 list형의 변수인 waiters는 해당 semaphore을 기다리는 thread들의 list를 나타낸다. 다음으로는 lock struct를 살펴보자.



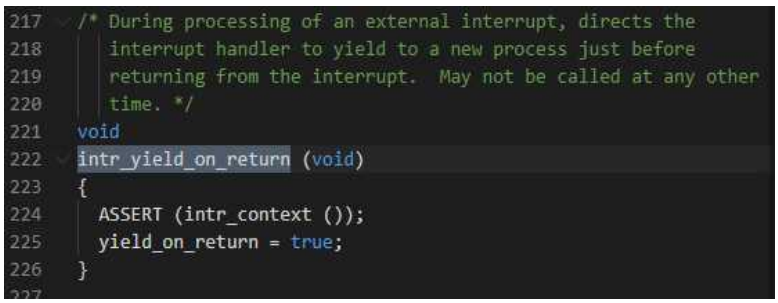
```
struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debug
    struct semaphore semaphore; /* Binary semaphore controlling a
  };
```

lock 구조체에는 현재 lock을 소유하고 있는 thread포인터 형의 변수인 holder가 존재한다. 그리고 semaphore형의 변수인 semaphore가 존재하는데 이는 lock이므로 binary semaphore로 동작한다.

기존의 Pintos에서는 thread scheduling을 Round Robin 방식을 이용하여 구현하였다. Round Robin 방식이란 CPU에 thread를 할당할 때 priority가 높은 thread를 우선 순위로 할당하는 것이 아니라 단지 ready_list에 있는 thread를 순서대로 pop out하여 할당하는 것을 의미한다. thread를 바꾸는 과정은 context switching 하는 과정이므로 먼저 thread/interrupt.c에서 context switching을 진행하는 함수인 intr_handler(struct intr_frame *frame) 함수를 살펴보았다.



위의 코드에서 thread가 바뀌는 부분을 찾아보면 가장 아래에 yield_on_return이라는 값이 true일 경우에 thread_yield() 함수를 통해 context_switching이 일어나고 있는 모습을 볼 수 있다.



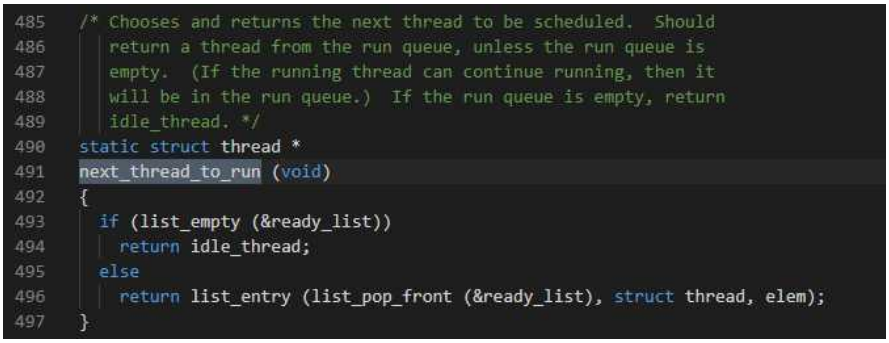
yield_on_return 값은 위 코드에서 보는 것과 같이 intr_yield_on_return() 함수에서 true로 설정을 해 주고 있었다. 다음으로 yield_on_return 값이 true일 때 이동하는 thread_yield() 함수를 살펴 보았다.

```
299   /* Yields the CPU.  The current thread is not put to sleep and
300      may be scheduled again immediately at the scheduler's whim. */
301   void
302   thread_yield (void)
303   {
304     struct thread *cur = thread_current ();
305     enum intr_level old_level;
306
307     ASSERT (!intr_context ());
308
309     old_level = intr_disable ();
310     if (cur != idle_thread)
311       list_push_back (&ready_list, &cur->elem);
312     cur->status = THREAD_READY;
313     schedule ();
314     intr_set_level (old_level);
315   }
```

이 함수는 현재 진행되고 있는 thread를 ready_list에 추가하고 현재 thread의 status를 THREAD_READY로 변경하여 schedule() 함수를 실행하고 있었다. 이 때, ready_list에 현재의 thread를 추가할 때 list_push_back() 함수를 이용하여 추가하는 모습을 보며 단지 ready_list에 priority를 기준으로 sort하여 삽입하는 것이 아니라 push_back하는 모습을 볼 수 있었다. 그렇다면 status를 THREAD_READY로 변경한 이후 schedule()을 실행하는데 schedule()에서는 어떤 기능을 하는지 살펴 보기 위하여 이동하였다.

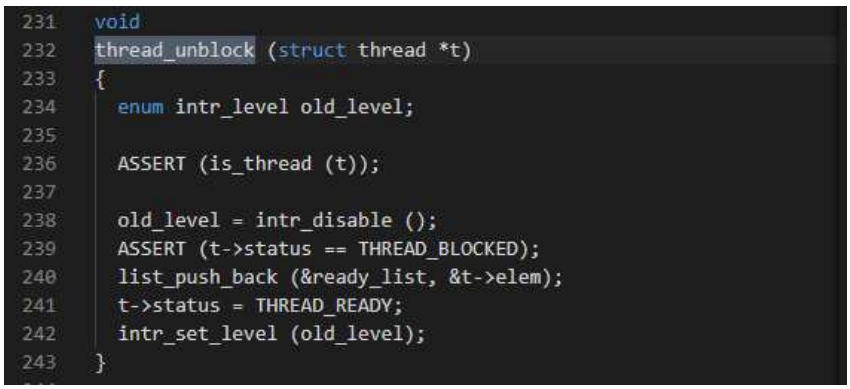
```
545   /* Schedules a new process.  At entry, interrupts must be off and
546      the running process's state must have been changed from
547      running to some other state.  This function finds another
548      thread to run and switches to it.
549
550      It's not safe to call printf() until thread_schedule_tail()
551      has completed. */
552   static void
553   schedule (void)
554   {
555     struct thread *cur = running_thread ();
556     struct thread *next = next_thread_to_run ();
557     struct thread *prev = NULL;
558
559     ASSERT (intr_get_level () == INTR_OFF);
560     ASSERT (cur->status != THREAD_RUNNING);
561     ASSERT (is_thread (next));
562
563     if (cur != next)
564       prev = switch_threads (cur, next);
565     thread_schedule_tail (prev);
566   }
```

schedule() 함수에서는 next_thread_to_run() 함수를 이용하여 next 포인터에 다음으로 실행될 thread를 찾아 넣는다. 그렇다면 next 포인터에 어떤 thread를 할당하는지 살펴보기 위하여 next_thread_to_run() 함수로 이동하였다.



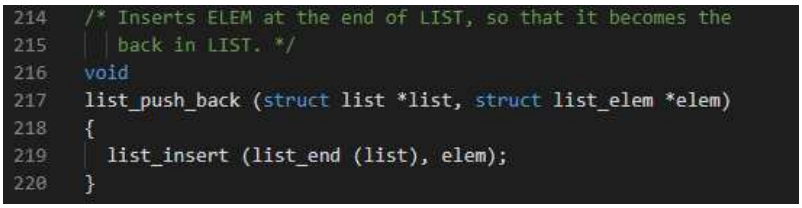
```
/* Chooses and returns the next thread to be scheduled.  Should
   return a thread from the run queue, unless the run queue is
   empty.  (If the running thread can continue running, then it
   will be in the run queue.)  If the run queue is empty, return
   idle_thread. */
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
    return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
```

next_thread_to_run() 함수에서는 ready_list가 empty가 아니라면 ready_list의 가장 앞의 element를 pop하여 return 해주고 있었다. 그렇다면 ready_list에 thread가 insert되는 방식은 어떨까? 만약 thread가 priority의 우선 순위를 고려하여 sort되어 insert된다면 단지 pop_front의 방식을 통해서도 priority scheduling이 가능 할 것이다. 그렇다면 naive Pintos에서는 어떤 방식을 사용하여 구현했는지 살펴보기 위해 ready_list에 insert하는 경우를 모두 살펴보자.



```
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_push_back (&ready_list, &t->elem);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

thread_unblock() 함수에서 list_push_back() 함수를 이용하여 ready_list에 thread를 삽입하고 있었다. 그렇다면 list_push_back() 함수가 어떻게 동작하는지 알아보기 위하여 lib/kernel/list.c로 이동하여 함수를 찾아 보았다.



```
/* Inserts ELEM at the end of LIST, so that it becomes the
   back in LIST. */
void
list_push_back (struct list *list, struct list_elem *elem)
{
  list_insert (list_end (list), elem);
}
```

list_push_back() 함수에서는 단순히 list의 마지막에 element를 삽입하는 방식을 사용하고 있었다. 즉, priority를 전혀 고려하지 않고 있었다.

따라서 기존의 Pintos에서는 thread를 ready_list에 삽입할 때 priority를 기준으로 sorting하지도 않고, ready_list에서 다음 실행할 thread를 pop out할 때 priority가 높은 것 먼저 pop하는 것이 아니라 단지 front element를 pop 하기 때문에 Round Robin 방식을 사용하고 있는 것을 확인 할 수 있었다.

다음은 동기화 부분이다. 동기화와 관련된 부분에는 semaphore과 lock이 존재한다. semaphore은 critical section에 한 번에 여러개의 thread가 접근 가능하지만 lock은 하나

의 thread만 접근이 가능하다는 차이가 있다. 즉 lock은 semaphore의 특수 케이스라고 할 수 있다. 그렇다면 naive Pintos에서 어떻게 동기화를 구현하고 있는지 살펴보자.

```
60  void
61  sema_down (struct semaphore *sema)
62  {
63    enum intr_level old_level;
64
65    ASSERT (sema != NULL);
66    ASSERT (!intr_context ());
67
68    old_level = intr_disable ();
69    while (sema->value == 0)
70      {
71        list_push_back (&sema->waiters, &thread_current ()->elem);
72        thread_block ();
73      }
74    sema->value--;
75    intr_set_level (old_level);
76  }
```

```
104 /* Up or "V" operation on a semaphore.  Increments SEMA's value
105    and wakes up one thread of those waiting for SEMA, if any.
106
107    This function may be called from an interrupt handler. */
108 void
109 sema_up (struct semaphore *sema)
110 {
111   enum intr_level old_level;
112
113   ASSERT (sema != NULL);
114
115   old_level = intr_disable ();
116   if (!list_empty (&sema->waiters))
117     thread_unblock (list_entry (list_pop_front (&sema->waiters),
118                                 struct thread, elem));
119   sema->value++;
120   intr_set_level (old_level);
121 }
```

thread가 sema_down() 함수를 통해 동기화 부분에서 작업을 실행하고 sema->value가 0이 될 때, 즉 어떤 thread가 이미 작업 중일때 이를 waiters list에 list_push_back() 함수를 통해 넣어둔다. 그리고 sema_up() 함수를 통해 thread_unblock을 하게 되는데 이때 list_pop_front() 방식을 통해 thread를 뽑아오는 것을 볼 수 있다. 여기서도 thread의 priority는 고려하고 있지 않고 단지 FIFO의 방식을 활용하고 있는 것을 볼 수 있다.

2.2 Solutions

기존의 naive Pintos에서는 thread_create() 함수를 통해 thread가 만들어질 때 단지 thread_unblock()을 통해 thread를 run queue에 넣으며 create을 종료했다. 하지만 우리는 이제 priority를 고려해야 하므로 thread_create() 함수의 마지막에 새로 만들어지는 thread의 priority와 현재 실행중인 thread의 priority를 비교하여 만약 새로 만들어지는 thread의 priority가 더 크다면 thread_yield()가 발생할 수 있도록 해야 할 것이다.

다음으로 ready_list에서 priority가 적용될 수 있게 하는 방법에는 두 가지가 존재한다. 먼저 ready_list에 thread를 insert할 때 priority를 적용하여 list_push_back() 함수를 쓰는 대신에 list_insert_ordered() 함수를 사용하여 insert하는 방법이 있고, insert할 때는 단지 list_push_back() 함수를 사용하지만 ready_list에서 pop 하기 전에 next_thread_to_run() 함수에서 list_sort() 함수를 이용하여 list를 sort하고 pop 하는 방법이 존재한다. 이 방법을 이용하기 위해서는 list_sort() 함수의 인자로 비교 함수가 필요하기 때문에 priority를 비교하는 함수를 추가로 구현해야 한다.

다음으로는 thread_donation을 구현하여 synchronization을 해결하는 방법이다. sema_down() 함수나 sema_up() 함수를 수정하여 이를 해결 할 수 있다. 먼저 sema_down() 함수를 수정하려면 waiters list에 element를 insert할 때 list_push_back() 함수 대신에 list_insert_ordered() 함수를 이용하여 priority를 기준으로 정렬하면서 넣는 방법이 있다. sema_up() 함수를 수정한다면 thread_unblock()을 진행할 때 waiters list에서 priority를 기준으로 가장 높은 element부터 pop을 하면 된다.

다음으로는 priority inversion 현상을 해결하기 위한 방법이다. priority inversion을 해결하기 위해서는 priority donation이 필요하다. 가장 simple한 donation 뿐만 아니라 multiple donation과 nested donation 모두를 구현해야 하므로 이를 모두 고려해야 한다. 먼저 thread 구조체에 추가로 donation이 끝났을 때 이전 priority로 돌아가기 위해 필요한 이전의 priority를 기억하는 변수가 추가로 필요 할 것이다. 또한, 현재 thread가 획득 하기

를 기다리는 lock의 주소를 저장하는 변수 또한 필요하다. 그리고 multiple donation을 구현할 때 필요한 donation을 받은 thread 구조체의 list들이 필요 할 것이다. 다음으로는 함수의 수정이다. 먼저 init_thread() 함수에서 새롭게 추가한 priority donation과 관련된 변수들의 초기화가 필요하다. 다음으로 lock_aquire() 함수에서는 donation을 진행하는 부분을 추가해야 한다. 만약 lock_aquire()의 인자로 받은 lock의 holder가 이미 존재한다면 아까 새롭게 추가했던 lock의 주소를 저장하는 변수에 획득 하기를 기다리는 lock의 주소를 저장한다. 그리고 multiple donation을 고려하기 위해서 이전 상태의 priority를 변수에 저장하고 donation을 받은 thread들을 아까 추가한 list를 이용하여 관리한다. 그 후에 donation을 위한 함수를 호출하고 마지막에 lock을 획득하게 되면 lock holder을 갱신해 준다. donation을 위한 새롭게 추가된 함수에서는 현재 thread가 기다리는 lock에 연결 된 모든 thread들을 순회하며 현재 thread의 priority를 lock을 hold하고 있는 thread에게 donation하는 작업이 필요하다. lock_release() 함수에서는 추가로 lock을 해지했을 때 이전의 상태로 돌아가게 하는 함수의 실행이 추가로 필요하다. 이는 두 개의 새로운 함수에서 나누어서 실행 될 것인데, 첫 번째 함수에서는 lock을 해지 했을 때 donation list에서 더이상 해당 entry가 필요 없기 때문에 해당 entry를 삭제 해주고 현재 thread의 donation list를 확인해서 해지 할 lock을 보유하고 있는 entry를 삭제한다. 다음으로 두 번째 함수에서는 thread의 priority가 변경되었을 때 donation을 고려해서 priority를 다시 결정하는 작업을 한다. 먼저 lock_release()가 일어났기 때문에 현재 thread의 priority를 donation받기 이전의 priority로 변경하고, 가장 priority가 높은 donation list의 thread와 현재 thread의 priority를 비교하여 더 높은 값을 현재 thread의 priority로 설정한다. 마지막으로 thread_set_priority() 함수와 thread_get_priority()의 수정이 필요하다. 먼저 thread_set_priority() 함수에서는 바로 직전에 언급했던 함수로 인해 priority가 변경되므로 donation 관련 정보를 갱신하는 작업이 추가로 필요하다. 이 작업에서 current thread가 더이상 highest priority를 가지고 있지 않담녀 thread_yield()함수를 호출하는 작업이 필요하다. 마지막으로 thread_get_priority() 함수에서는 해당 thread에 이전에 donation이 발생했다면 thread의 priority 대신 donate 받은 priority를 return 해주어야 한다.

3. Advanced Scheduler

우선 Pintos 커널은 floating point연산을 지원하지 않아 fixed point연산을 구현해야한다. 이번 구현에서는 appendix에 설명된 방식인 17.14방식을 사용해서 구현을 할 것이다. 32bit를 1 17 14로 나누어 사용하게 된다. Fixed point연산을 위한 함수들을 별도로 fixed_point.h/fixed__point.c에 구현해서 사용할 것이다.

Priority의 연산에는 appendix에 설명된 다음과 같은 식들이 사용된다.
- priority = PRI_MAX - (recent_cpu / 4) - (nice * 2).
- recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice.
- load_avg = (59/60)*load_avg + (1/60)*ready_threads.

우선 load_avg는 최근 1분동안 수행 가능했던 프로세스의 평균 개수를 나타내고, timer_ticks() % TIMER_FREQ == 0일 때 업데이트 된다.

다음으로 recent_cpu는 스레드가 최근 얼마나 많이 cpu를 점유했는지를 나타낸다. Init은 0으로 초기화되고, 자식 스레드는 부모의 값으로 초기화 된다. 그리고 매 timer interrupt마다 1이 증가하고 timer_ticks () % TIMER_FREQ == 0마다 재계산된다.

마지막으로 priority는 위의 주어진 식을 이용해서 매 4tick마다 재계산된다.

추가적으로, mlfqs는 0~63까지의 priority를 가지고 있고, 따라서 64개의 queue를 선언해서 각 priority마다 스레드를 추가할 수 있다. 또한 63에서 priority가 증가, 0에서 priority가 감소하는 경우에는 라운드 로빈으로 처리한다.

추가적으로 thread_mlfqs가 true인 경우 thread_set_priority가 수행되지 않도록 구현하고 priority의 변경은 set_nice함수안에서 구현한다.