Operating System (Fall 2014) Project 1
Wait Queue, Priority Scheduling

Jongwook Choi

Introduction

이 프로젝트의 목표는 기본적인 부분만 구현되어 있는 Pintos의 thread 기능에 새로운 방식의 구현을 추가하고 이를 통하여 thread의 구현방식이나 이들의 priority scheduling을 직접 수정, 구현하여 보는 과제로 크게 두 가지가 있다.

1. **wait queue 구현** : thread가 sleep될 때, 기존의 busy wait 방식을 제거하고 wait queue에 들어가도록 하고, 특정 시간이 지나면 다시 실행될 수 있도록 ready queue에 들어가도록 구현한다. alarm-multiple을 실행했을 때 Idle tick수를 체크해서 busy wait여부를 알 수 있다.
2. **priority scheduling 구현** : 우선순위가 높은 thread 가 먼저 실행될 수 있도록 하는 것이 기본 목적이다. 쓰레드가 새로 생성되거나, priority가 변경되거나 등의 여러 시나리오에 따라 항상 우선순위대로 실행이 되게끔 해야 한다. lock (critical section)이 있을 때 priority inversion 문제가 발생할 수 있는데, priority donation을 이용하여 이 문제를 해결해야 한다.

Task 1. Wait Queue

Design

기존에 실행될 쓰레드들을 관리하는 큐인 ready queue가 있는데, busy wait을 막기 위해 sleeping 상태인 쓰레드들을 관리하는 wait queue를 추가한다.

- Wait queue에 추가되는(push) 조건: `timer_sleep` 호출한 쓰레드를 큐에 추가함
- Wait queue에서 빠지는(pop) 조건
  - 매 tick (타이머에서 발생) 마다, 깨어날 쓰레드(지정한 시간만큼 sleep을 완료한)가 있는지를 검사한다.

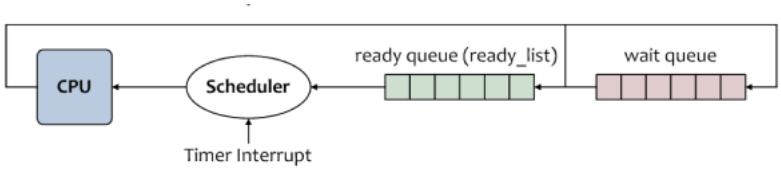

Figure 1: Wait Queue 개념도

– 검사하여 깨어난 쓰레드는 wait queue에서 dequeue 되고, ready queue로 다시 enqueue 된다.

자료 구조

각 쓰레드가 sleep 되는 경우, 어느 tick까지 sleep되어야 하는지에 대한 정보를 저장해야 한다. 이 정보는 `struct thread`의 `sleep_endtick` 에 쓰레드가 sleep된 경우 깨어날 tick로서 저장된다.

```
@@ -90,6 +90,7 @@ struct thread
     int priority;                       /* Priority. */
     struct list_elem allelem;           /* List element for all threads list. */
+    struct list_elem waitelem;          /* List element, stored in the wait_list queue */
+    int64_t sleep_endtick;              /* The tick after which the thread should awake (if the thread is in sleep) */

     /* Shared between thread.c and synch.c. */
     struct list_elem elem;              /* List element, stored in the ready_list queue */
```

Wait queue 는 `ready_list`와 유사하게 `wait_list` 라는 linked list로 관리된다. 이 리스트에 저장되는 element는 `struct thread`의 `waitelem`이다.

```
 /* List of processes in THREAD_READY state, that is, processes
    that are ready to run but not actually running. */
 static struct list ready_list;

+/* List of processes in sleep (wait) state (i.e. wait queue). */
+static struct list wait_list;
+
```

구현: Enqueue

자세한 것은 commit `d795340`를 참고하라.

먼저, sleep 을 위해서 `thread_sleep_until(int64_t ticks_end)` 함수가 추가되었는데, 현재 쓰레드를 `ticks_end` 틱까지 sleep 시키는 역할을 한다. 위에서 설명한 바와 같이 쓰레드에 `sleep_endtick` 정보를 저장하고 enqueue 한뒤, `thread_block()`를 통해 쓰레드가 block 되도록 한다. 이제 `timer_sleep` 함수에서는 `thread_sleep_until()` 함수를 이용하여 (현재 시각 + sleep할 tick 수) 쓰레드를 sleep 시키면 된다.

```
@@ -92,8 +92,13 @@ timer_sleep (int64_t ticks)
   int64_t start = timer_ticks ();

   ASSERT (intr_get_level () == INTR_ON);
-  while (timer_elapsed (start) < ticks)
-    thread_yield ();
+  enum intr_level old_level = intr_disable ();
+
+  // sleep the thread for `ticks` seconds,
+  // until the tick becomes [start + ticks]
+  thread_sleep_until (start + ticks);
+
+  intr_set_level (old_level);
```

참고로, timer sleep시 인터럽트를 disable하는 것이 필요했다. 이는 timer sleep 도중에 인터럽트가 발생하게 되면 wait queue에 넣는 과중에서 data race가 발생할 수 있기 때문이다.

구현: Dequeue

자세한 것은 commit `349f854`를 참고하라.

깨어날 쓰레드를 검사하기 위해서는 timer의 매 틱마다 검사를 하는데, 이를 위해서는 타이머의 interrupt service rountine을 사용하면 된다. 즉, 타이머 틱이 변할 때마다 호출되는 ISR (`timer.c: timer_interrupt`)가 있는데, 여기서 현재 tick 정보를 통해 깨어날 쓰레드들을 알 수 있다.

wait queue의 전체를 순회하면서, 깨울 쓰레드가 있는 경우 리스트에서 제거하는 `thread.c: thread_awake(int64_t current_tick)` 함수에 구현되어 있다. 깨어난 쓰레드는 `thread_unblock()`를 통해 다시 ready queue에 추가될 수 있다.

Test Results: `alarm-multiple`

wait queue가 구현된 이후 alarm-multiple 테스트 케이스의 실행 결과는 아래와 같다.

```
Loading..........
Kernel command line: -q run alarm-multiple
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  314,163,200 loops/s.
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 2: duration=30, iteration=1, product=30
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
(alarm-multiple) thread 3: duration=40, iteration=1, product=40
(alarm-multiple) thread 1: duration=20, iteration=2, product=40
(alarm-multiple) thread 0: duration=10, iteration=4, product=40
(alarm-multiple) thread 4: duration=50, iteration=1, product=50
(alarm-multiple) thread 0: duration=10, iteration=5, product=50
(alarm-multiple) thread 2: duration=30, iteration=2, product=60
(alarm-multiple) thread 1: duration=20, iteration=3, product=60
(alarm-multiple) thread 0: duration=10, iteration=6, product=60
(alarm-multiple) thread 0: duration=10, iteration=7, product=70
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 3: duration=40, iteration=3, product=120
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 4: duration=50, iteration=3, product=150
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
```

```
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
Timer: 582 ticks
Thread: 550 idle ticks, 32 kernel ticks, 0 user ticks
Console: 2955 characters output
Keyboard: 0 keys pressed
Powering off...
```

아래에 보면 각 tick이 소모된 쓰레드 정보를 알 수 있는데, busy wait은 idle tick이 없음에 반해 wait queue를 구현한 이후에는 550개의 idle tick이 생겨서 올바르게 동작했음을 확인할 수 있다. idle tick이란 idle thread가 실행된 tick의 수이므로, 각 alarm 쓰레들이 sleep 되는 동안 tick (CPU 자원)을 소모하지 않고 block 되어 있다가 적당한 시점에 깨어났다는 의미이다.

```
-Timer: 580 ticks
-Thread: 0 idle ticks, 580 kernel ticks, 0 user ticks
+Timer: 582 ticks
+Thread: 550 idle ticks, 32 kernel ticks, 0 user ticks
```

Task 2. Priority Scheduling and Donation

Design: Priority Scheduling

쓰레드의 priority가 올바르게 동작하는 것이 요구사항인데, 결국 현재 쓰레드가 실행되고 있는데 우선순위와 관련한 특정 조건이 발동되면 현재 쓰레드를 yield 시켜서 다시 schedule 될 수 있도록 하는 것이 핵심이다. `ready_list`에 있는 쓰레드들 중에서 가장 높은 우선순위의 쓰레드가 실행되면 되므로, 조건들을 생각해보면

- 쓰레드가 새로 생성되어 큐에 들어오는 경우 (`thread_create`)
- block 되어 있던 쓰레드가 unblock 되어 큐에 들어오는 경우 (`thread_unblock`)
- 실행되고 있던 쓰레드가 yield 되어 큐에 들어오는 경우
- 현재 쓰레드의 우선순위가 변경된 경우 (`thread_set_priority`)

가 있다.

또한, lock 구현 (semaphore) 에서도 priority 대로 동작해야 한다. `semaphore` 구조체에는, 진입하지 못하고 대기 (block) 하는 thread를 관리하는 `waiters` 라는 list가 있는데, 이미 세마포어를 점유하고 있는 쓰레드가 빠져나갈 때 (`sema_up`) 하나의 쓰레드가 다시 실행될 수 있으므로 unblock 될 것이다. 이 때 빠져나오는 쓰레드가 우선순위가 가장 높은 것이어야 한다.

마지막으로, monitor를 구현하기 위한 conditional variables 에서도 semaphore와 비슷한 요구조건이 있다. conditional variable에서 대기하고 있는 semaphore들도 (`struct condition`의) `waiters`라는 list로 관리된다.

이를 위해 각각의 경우에서 — (i) ready queue인 `ready_list`를 (ii) semaphore 대기열 `waiters`를 (iii) condition 대기열 `waiters` 를 — 정렬된 상태로 유지한다. 각 queue는 일종의 priority queue 라는 자료구조로 추상화해서 생각할 수 있는데, 특별히 복잡한 우선순위 큐 구현을 쓰지 않고, (시간복잡도의 비효율성은 존재하지만) 그냥 기존처럼 linked list로 다루되 정렬된 상태를 유지함으로써 올바른 스케쥴링이 가능하도록 한다.

Implementation

간단히 수정사항을 언급하면, 일단 `ready_list`에 추가될 때마다 정렬된 순서를 유지하도록 한다. `thread_unblock()` 및 `thread_yield()` 에 존재하는, `ready_list`에 추가하는 부분을, `list_insert_ordered()` 함수를 사용하여 `thread`의 priority 대로 정렬되어 list에 삽입되도록 수정해준다. (`comparator_greater_thread_priority` 비교 함수)

높은 우선순위의 쓰레드가 스케쥴(실행)되기 위해서는

- 새로운 쓰레드 t가 생성되어 추가되었는데 (`thread_create`) 현재 쓰레드 t_0보다 우선순위가 높은 경우
- 쓰레드 t가 unblock 되고 (`thread_unblock`), 현재 쓰레드 t_0보다 우선순위가 높은 경우
- 쓰레드 t가 세마포어 대기열에서 깨어날 때 (`sema_up`), 현재 쓰레드 t_0보다 우선순위가 높은 경우
- 현재 쓰레드 t_0의 우선순위가 변경되는 경우 (donation 받는 경우도 포함), 바로 다음에 실행될 쓰레드 t가 존재하여 t의 우선순위가 더 높은 경우

바로 `thread_yield`를 하여 재스케쥴링이 일어날 수 있도록 처리해준다.

Priority Inversion Problem

Lock이 있는 경우 priority inversion 문제가 발생할 수 있다. 각각 높은/중간/낮은 우선순위 쓰레드 H, M, L이 있는데, 어떤 lock에 의해 L가 실행되고 있고 H가 block인 상태이며 (lock과 독립인) M가 ready list에 있으면, M가 L보다 항상 먼저 스케쥴되어 H가 (M보다 먼저) 실행되지 않는 현상이다. 이를 막기 위해 priority donation을 해야 하는데 가장 간단한 시나리오를 생각해보면

- 현재 쓰레드 H가 lock을 acquire 할 때, 이미 lock을 물고 있는 holder 쓰레드 L이 있고 H의 우선순위가 L보다 높은 경우,
  - L의 우선순위를 H의 우선순위로 기부함
  - 이 때 L의 원래 우선순위는 기억하고 있어야 함 (`struct thread: original_priority`)
- (우선순위를 기부받은) L가 lock을 release하려고 할 때,
  - L의 우선순위를 기억해둔 `original_priority`로부터 복구함

와 같은 메커니즘으로 동작한다. 그런데 한 쓰레드가 하나 이상의 쓰레드로부터 우선순위를 donate 받거나, 중첩으로 donation이 일어날 수도 있으므로 이를 고려한 자료구조를 디자인 해야한다. 그 내용은 아래와 같다.

Design: Priority Donation

- 각 `thread`가 새롭게 저장하는 자료들
  - donation 받기 전의 원래의 우선순위(`original_priority`),
  - 어떤 lock에 의해 block되어 있는지(`waiting_lock`) : 이 lock을 앎으로써 lock의 holder를 참조하면 priority 기부의 중첩 구조를 모두 알 수 있음 (for nested donation)
  - 이 쓰레드가 잡고 있는 lock 객체들의 리스트(`locks`) : 이 쓰레드에게 우선순위를 기부한 다른 쓰레드들을 알 수 있음 (for multiple donation)

```
@@ thread.h (struct thread)
     int priority;                       /* Priority. */
+    int original_priority;              /* Priority, before donation */
@@ thread.h (struct thread)
+    // needed for priority donations
+    struct lock *waiting_lock;          /* The lock object on which this thread is waiting (or NULL if not locked) */
+    struct list locks;                  /* List of locks the thread holds (for multiple donations) */
```

- 각 `lock`이 저장하는 자료들
  - `lockelem` : `thread::locks`에 저장하기 위한 list element
  - `priority` : 현재 lock의 priority는 이 lock을 소유한 쓰레드 (`holder`)의 priority로 정의한다.

```
@@ synch.h (struct lock)
     struct thread *holder;      /* Thread holding lock (for debugging). */
     struct semaphore semaphore; /* Binary semaphore controlling access. */
+
+    struct list_elem lockelem;  /* List element for the thread's 'locks' list. */
+    int priority;               /* priority of the the thread holding the lock (for priority donation) */
```

Lock을 잡을 때의 처리

쓰레드 `t`가 lock `l`에 대해 `lock_acquire`를 부르면 `waiting_lock`이 `l`로 설정된다. 그러면 priority donation chain 의 다음 쓰레드인 `l->holder` 의 우선순위가 `t` 보다 낮으면, `l->holder` 가 `t` 로부터 우선순위를 donate 받고 `l`의 priority를 기부받은 우선순위인 `t`의 우선순위로 세팅한다(위의 정의에 따라).

그런데 nested donation이 되려면, `l->holder`의 다음 쓰레드 (`l->holder`가 잡고 있는 lock의 holder) 의 우선순위가 여전히 낮다면, 이 우선순위가 다음 쓰레드에게도 donate 되어야 한다. 따라서 우리는 위의 priority

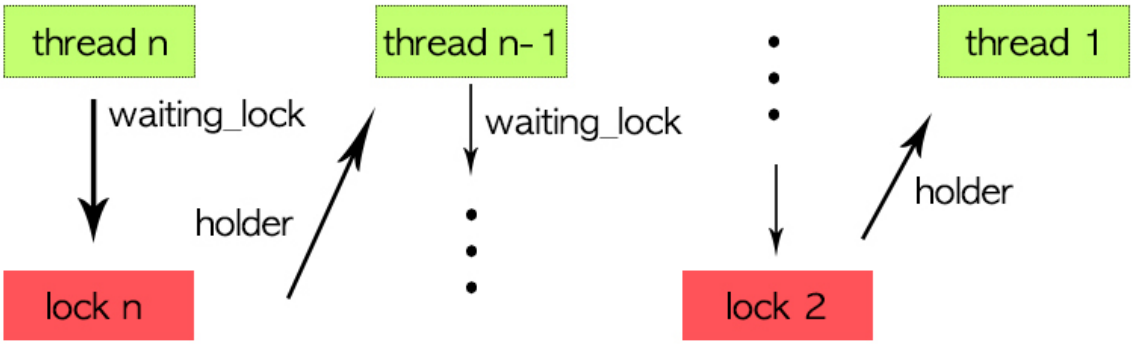

Figure 2: Priority Donation Chain

donation chain를 따라서 `t`와 `l`을 이동하며, **`t`의 우선순위가 `t->waiting_lock->holder`의 우선순위보다 높을 때까지 priority를 기부하면** 된다. 이렇게 하면 lock을 점유하는 running 쓰레드가, (chain의 앞쪽에서) 락을 얻고자 하는 쓰레드의 priority를 기부받게 되므로 priority inversion 을 해결할 수 있다.[1]

최초의 lock `l`에 대해서 `sema_down`를 통해 semaphore count가 줄어들고, 마지막으로 `l`의 holder 쓰레드의 locks 를 업데이트 함으로써[2] priority를 기부한 쓰레드들의 리스트를 최신으로 업데이트해야 한다 (이것은 나중에 lock을 해제할 때, priority를 어디로 복구해야 하는지를 알기 위함이다.

이런 일련의 과정이 `lock_acquire()` 함수에 구현되어 있다.

Lock을 해제할 때의 처리

반대로, 현재 쓰레드 `t`가 락 `l`을 해제한다고 가정하자. 그러면 일단 `l->holder` 및 lock list 등의 자료구조를 올바르게 세팅된다. 이제 semaphore count가 증가된 이후, priority donation 을 복구해야 한다. `t`가 priority를 기부받았을 가능성이 있는데 이것이 chain을 따라 중첩으로 있을 수가 있다. 따라서 `locks` list를 보면서 `t`에게 priority를 기부한 다른 쓰레드들을 파악하는데 (위에서 이 정보를 저장했다)

[1] 귀납법을 통해 엄밀한 증명이 가능하지만 생략한다.

[2] lock의 priority를 정의하여 이 순서대로 정렬하여, 중첩 구조에서 lock release 발생 시 새롭게 priority를 무엇으로 업데이트해야 하는지 정보를 유지할 수 있다. 아래 release 쪽 설명 참고.

- priority donor (기부한 쓰레드)가 없다면, donation chain의 끝이므로 `original_priority`를 통해 원래의 priority로 복구하면 된다.
- priority donor 들이 있다면, 이 중 highest priority 가 `t`의 새로운 (donated) priority 가 될 것이다. 따라서 이에 맞도록 donation을 다시 수행하여 `t`의 priority를 변경해준다[3].

이런 일련의 과정이 `lock_release()` 함수에 구현되어 있다. 참고로 앞에서도 언급했듯이 thread donation을 통해 우선순위가 변경되면, 적절한 yield를 통해서 쓰레드들이 높은 우선순위를 갖는 것부터 스케쥴링되어 실행될 수 있도록 해주어야 한다.

Test Results: `alarm-priority`

Priority scheduling 구현이 완료된 이후 alarm-priority 테스트 케이스의 실행 결과는 아래와 같다.

```
Loading..........
Kernel command line: -q run alarm-priority
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  314,163,200 loops/s.
Boot complete.
Executing 'alarm-priority':
(alarm-priority) begin
(alarm-priority) Thread priority 30 woke up.
(alarm-priority) Thread priority 29 woke up.
(alarm-priority) Thread priority 28 woke up.
(alarm-priority) Thread priority 27 woke up.
(alarm-priority) Thread priority 26 woke up.
(alarm-priority) Thread priority 25 woke up.
(alarm-priority) Thread priority 24 woke up.
(alarm-priority) Thread priority 23 woke up.
(alarm-priority) Thread priority 22 woke up.
(alarm-priority) Thread priority 21 woke up.
(alarm-priority) end
Execution of 'alarm-priority' complete.
Timer: 525 ticks
Thread: 490 idle ticks, 35 kernel ticks, 0 user ticks
Console: 840 characters output
Keyboard: 0 keys pressed
Powering off...
```

[3] 이것도 우선순위 큐 등으로 관리해야 하지만 구현상으로 maximum lookup을 하는 식으로 구현되어 있다(개선이 필요함).

이 테스트 케이스는 쓰레드 우선순위를 25, 24, ···, 21, 30, 29, ···, 26 순서대로 생성하여 스케쥴링한다. 하지만 쓰레드는 항상 우선순위가 높은 순서대로 실행되기 때문에 30 부터 21까지 우선순위에 따라 스케쥴링되어 순차적으로 실행된 것을 확인할 수 있다.

결론 및 추가 과제

모든 조원이 함께 내용을 이해하고, 토론하고, 코딩 및 디버깅/문서화에 참여하였다. 기본적인 요구사항 만족 및 구현은 완료하여, 모든 테스트 케이스를 성공시켰다.

```
SUMMARY BY TEST SET

Test Set                                        Pts Max  % Ttl  % Max
----------------------------------------------- --- --- ------ ------
tests/threads/Rubric.alarm                       18/ 18  30.0%/ 30.0%
tests/threads/Rubric.priority                    38/ 38  70.0%/ 70.0%
----------------------------------------------- --- --- ------ ------
Total                                                   100.0%/100.0%
```

몇가지 개선사항들은 많이 있다. 가장 먼저 우선순위 힙 구현을 linked list가 아닌 heap 또는 64개의 list로 구현하면 (thread priority는 64종류밖에 안되므로) 더 효율적일 수 있는데, 런타임 성능이 critical 하지 않으므로 일단은 간단한 구현에 초점을 맞추었다. 이와 같은 부분에 대한 개선도 필요할 것이고, starvation을 방지하기 위한 BSD Scheduler 등을 구현하는 작업이 더 필요할 수 있는데 이런 과제들은 상당히 재미있는 과제가 될 것이다.