**Pintos Project 3: Threads**

담당 교수 : 박성용

학번 / 이름 : 20151149 / 문상호

개발 기간 : 2021/11/25 ~ 2021/12/6

1. **개발 목표**

- 기존 busy waiting 방식의 Alarm Clock을 blocked status로 변경하는 방식을 사용하여 효율적으로 구현한다.

- 기존의 round-robin 기법으로 구현된 스케줄링을 우선순위가 높은 스레드를 먼저 실행하는 priority 기법으로 변경하여 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

|  |  |
| --- | --- |
| **/threads** | **init.c synch.c thread.c thread.h** |
| **/devices** | **timer.c** |

* 1. Alarm Clock

alarm clock을 busy waiting 방식에서, 스레드를 blocked status로 변경하고 매 틱마다 깨어나야할 스레드를 검사하여 unblocked 상태로 변경해줌으로써 무한루프를 돌지 않아도 되어 좀 더 효율적으로 구현이 가능하다.

* 1. Priority Scheduling

기존 round robin 방식에서 우선순위를 고려한 priority 스케줄링 기법으로 변경하여 구현한다. 스레드를 ready 상태로 만들어줄 때 list에 우선순위값을 기준으로 정렬되어 삽입되게끔 작성하여 priority가 가장 높은 스레드가 우선적으로 실행될 수 있도록 처리한다.

* 1. **개발 내용**

1. ~~Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.~~

~~Blocked 상태의 스레드들은 thread\_unblock 함수를 호출하여 unblock status로 변경할 수 있지만, 언제 어떤 스레드를 어떻게 unblock 처리해줄 것인지가 중요하다. 이를 위해서 구현해야 하는 것은 다음과 같다. 1. blocked 상태의 스레드들을 관리하는 하나의 큐가 있어야 하고, 2. 각 스레드들이 언제 깨어나야하는지에 대한 정보를 가지고 있고, 3. 매번 타이머 인터럽트가 발생할 때마다 스레드들을 unblock처리해줘야할 추가 함수 구현이 필요하다.~~

1. ~~Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.~~

가장 높은 우선순위의 스레드가 ready list의 어느 곳에 위치하는지와, 그 스레드를 언제 처리해줄 것인지가 중요하다. 먼저 전자를 위한 구현 방법은 ready\_list에 삽입할 때 해당 리스트를 우선순위 값으로 내림차순 정렬하는 것이라고 생각했다. ready\_list의 맨 앞에 항상 우선순위가 높은 스레드를 배치시키는 것이다. list.c의 list\_insert\_ordered 함수를 보면 parameter로 전달하는 less 비교 함수를 가지고 정렬과 함께 값을 삽입할 수 있는 함수이다. less 함수를 추가 작성하여 해당 삽입 함수를 사용한다면 비교 대상 a와 b 각 스레드의 priority를 비교하여 항상 내림차순 정렬 상태를 유지한 ready\_list를 만들 수 있을 것이다. 이제 우선순위로 정렬된 ready\_list에서 priority가 실행 중인 스레드의 priority보다 높은 스레드가 있을 경우, 먼저 스케줄링에 의해 실행되도록 핸들링이 필요하다. 이를 위해서 ready\_list의 스레드와 러닝 스레드 우선순위를 비교하여 thread\_yield 처리를 해주는 선점 작업 함수(thread\_preempt)를 구현하여 priority가 변경되는 함수인 thread\_create와 thread\_set\_priority에서 해당 함수를 호출하면 러닝 스레드에서보다 높은 우선순위를 가진 스레드를 스케줄링 시 먼저 처리할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 11/25 ~ 11/28 | proj3.pdf 메뉴얼 학습 및 코드 분석 |
| 11/29 ~12/1 | alarm clock 구현 |
| 12/2 ~ 12/4 | priority scheduling 구현 |
| 12/5 ~ 12/6 | 디버깅, 보고서 작성 |

* 1. **개발 방법**

1) Alarm Clock

기존의 busy waiting 방식의 alarm clock은 스레드가 running - ready status를 반복하여 갖게 되는 구조다. 잠들어야 하는 스레드들을 block status로 변경하여 깨어나기 전까지는 스케줄링에 영향을 받지 않도록 해야한다. 이를 위해서 스레드 구조체에 일어나야 하는 시간 정보를 담은 wake\_time 필드를 둔다. 그리고 sleep\_list(스레드 struct 배열)을 추가하여 block status를 가진 스레드들을 관리한다. 정리하면 thread\_sleep에서 스레드가 일어날 시간(wake\_time)을 저장해주고 sleep\_list에 삽입하고 스레드를 block 상태로 만들어주면 busy waiting 방법을 대신할 수 있다.

이렇게 잠든 스레드들은 일어날 시간이 되면 다시 ready 상태로 변경해줘야 하는데, 이를 위해서 thread\_wake 함수를 작성한다. 이 함수에서는 sleep\_list를 iterate하며 일어날 시간이 된 스레드들을 sleep\_list에서 제거하고 unblock 처리해준다. 이는 매 tick마다 실행되는 timer\_interrupt 함수에서 호출하도록 하면 매 틱마다 깨워줘야 할 스레드들을 핸들링할 수 있다.

2) Priority Scheduling

우선순위 스케줄링으로 구현하기 위해 ready\_list에 삽입할 때 해당 리스트를 우선순위 값으로 정렬하기로 하였다. list.c의 list\_insert\_ordered 함수를 보면 parameter로 전달하는 less 비교 함수를 가지고 정렬과 함께 값을 삽입할 수 있는 함수이다. 이를 사용하고, 내림차순(값이 클수록 우선순위가 높으므로) 정렬을 위해 less 함수를 추가 작성하여 비교 대상 a와 b 각 스레드의 priority를 비교한다. 이제 ready\_list에 스레드를 삽입하는 부분이 작성된 thread\_yield와 thread\_unblock 함수에 list\_push\_back 대신 list\_insert\_ordered를 작성하면 list에서 우선순위대로 정렬된다. 동기화(synchronization)를 위한 semaphore와 lock에서도 같은 방식의 스케줄링을 구현해줄 수 있다. sema\_down 시에도 waiters 배열에 스레드를 추가해줄 때 list\_insert\_ordered로 변경하고, sema\_up에서는 스레드를 unblocking 하기전 waiters 배열을 우선순위 내림차순으로 정렬(list\_sort)한다.

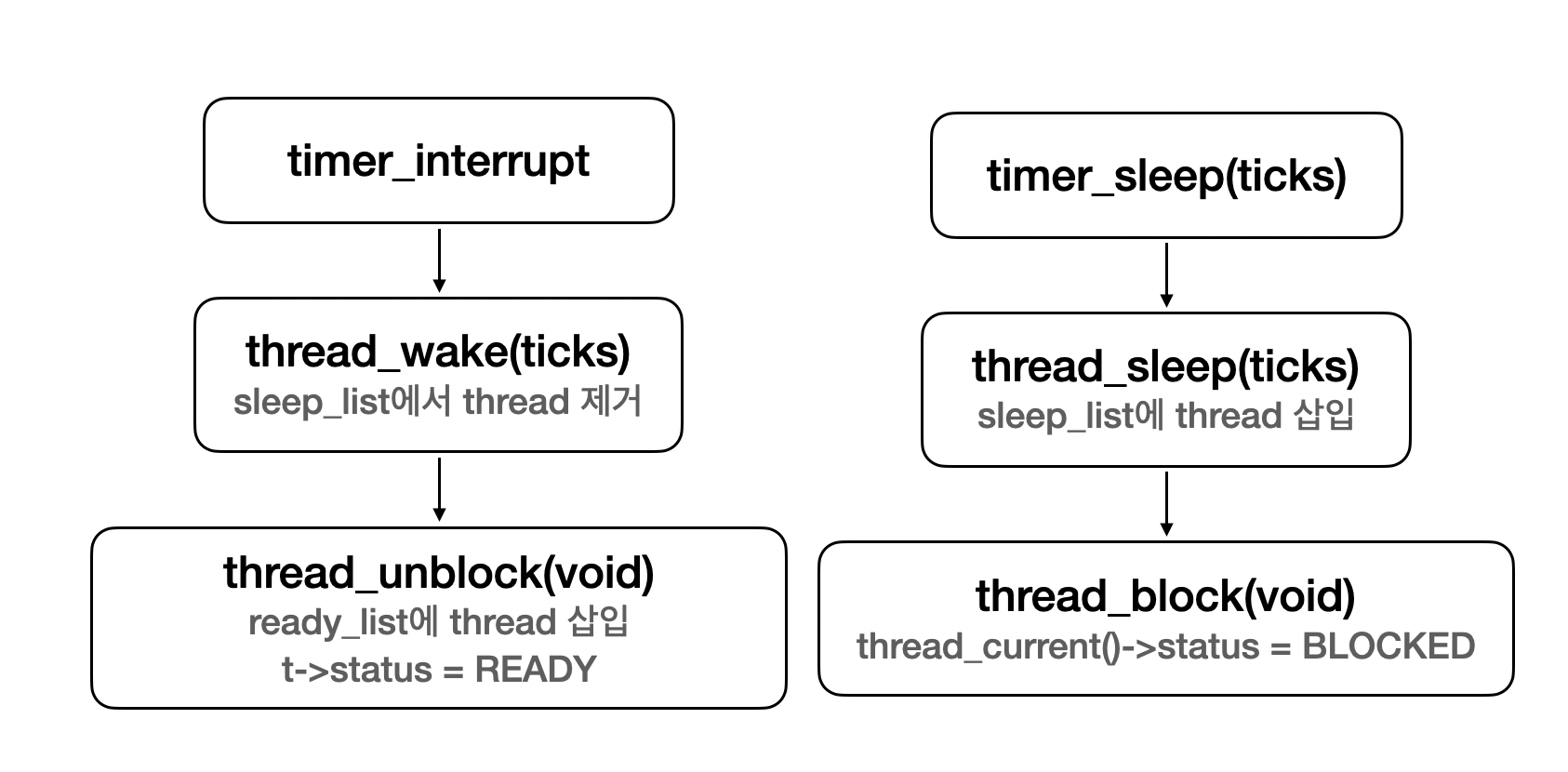
현재 ready\_list 스레드의 priority가 실행 중인 스레드의 priority보다 높을 경우, 먼저 스케줄링에 의해 실행되도록 핸들링이 필요하다. 이를 위해서 ready\_list의 스레드와 러닝 스레드 우선순위를 비교하여 thread\_yield 처리를 해주는 선점 작업 함수(thread\_preempt)를 구현하여 priority가 변경될 수 있는 thread\_create와 thread\_set\_priority에서 해당 함수를 호출한다.

이외의 priority\_condvar 테스트가 계속 실패하여 코드를 살펴보니 cond\_wait으로 처리한 스레드를 cond\_signal로 깨울 때 가장 높은 우선순위의 스레드를 깨워야하는 방식이었다. 지금까지는 다루지 않았던 condition struct를 다룰 때에도 우선순위 스케줄링을 구현해주어야 했다. 이전 작업들과 크게 다를 것 없이 wait(down)에는 list\_insert\_ordered로 변경해주고, signal 에서는 sema\_up전에 waiters를 정렬해주었다. 그러나 위의 semaphore struct를 다룰 때와의 다른 점은 condition의 waiters는 스레드가 아닌 세마포어들의 리스트라는 것이다. condition의 waiters 리스트 내의 semaphore 들 중 우선순위가 높은 세마포를 깨워야 한다. 이전 작업에서 semaphore struct의 waiters(스레드 리스트) 필드 정렬은 이미 처리하였으므로, condition에서의 우선순위 처리는 semaphore.waiters의 맨 앞 스레드 우선순위들끼리 비교하면 된다. 이를 위한 비교 함수를 따로 추가 작성해주어 cond\_wait과 cond\_signal에서 사용하였다.

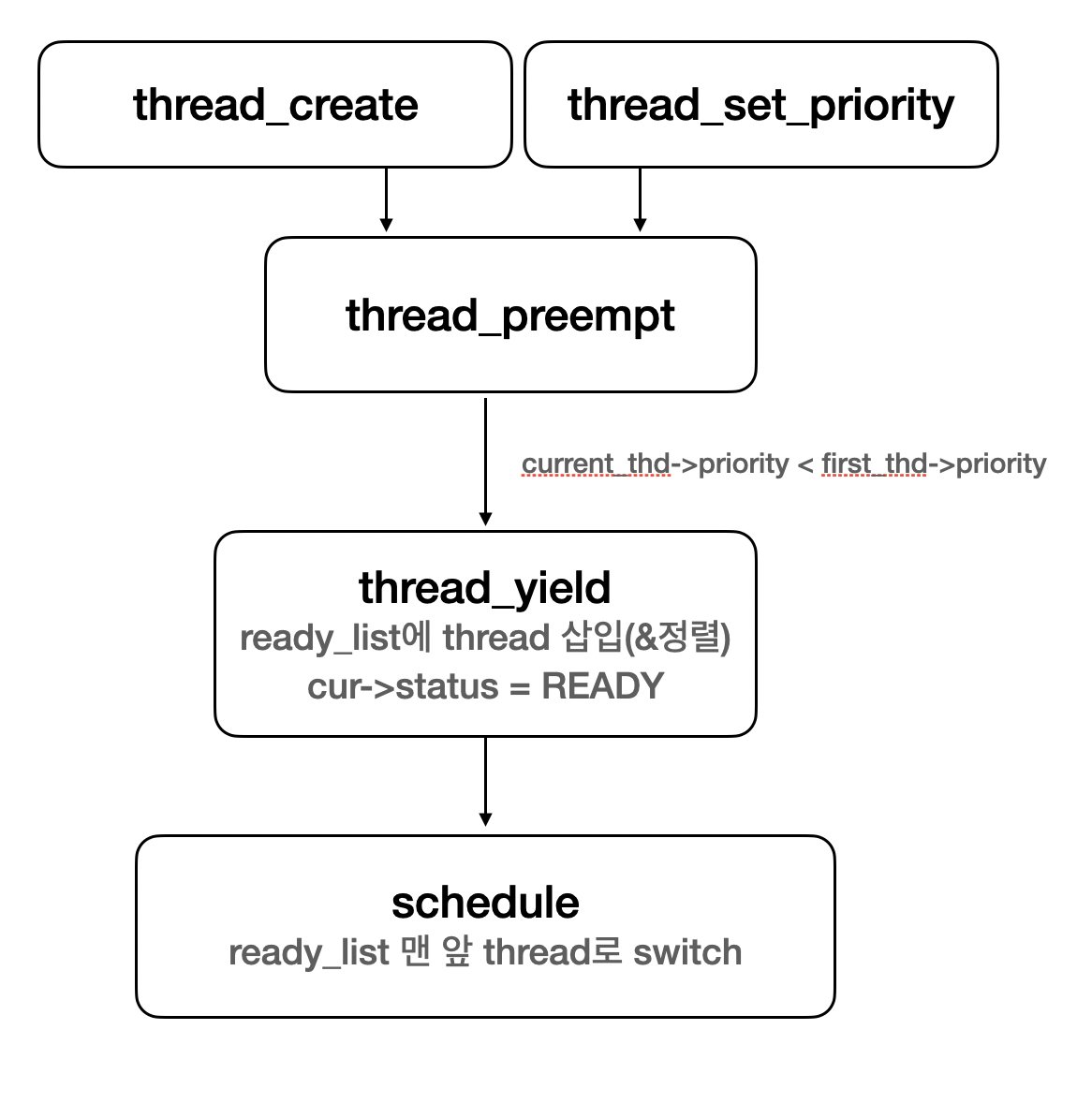
추가 구현 사항으로 스레드 starvation을 막기 위해 aging 처리를 해주는 함수를 구현하였다. timer\_interrupt마다 호출되는 thread\_tick 함수에서 호출되어 ready\_list 내의 스레드들의 priority를 틱마다 1씩 올려준다.

1. **연구 결과**
   1. **Flow Chart**

~~Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.~~



~~Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.~~

~~~~

* 1. **제작 내용**

**1-1. /threads/threads.h**

- blocked 상태의 스레드를 unblock처리할 때, 언제 깨어나야하는지를 판단하기 위해 wakeup\_time 필드를 추가하였다.

|  |
| --- |
| struct thread  {  ...  int64\_t wakeup\_time;  } |

- 추가 구현한 함수들의 프로토타입을 헤더 파일에 선언하였다.

|  |
| --- |
| ...  void thread\_wake (int64\_t ticks);  void thread\_sleep (int64\_t ticks);  bool compare\_priority (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED);  void thread\_aging ();  void thread\_preempt ();  ... |

**1-2. /threads/threads.c**

- blocked 상태의 스레드들을 관리하기 위해 sleep\_list 구조체를 전역으로 선언하였다.

|  |
| --- |
| ...  static struct list sleep\_list;  ... |

- 스레드를 block 상태로 만드는 함수를 추가 작성하였다. wakeup\_time을 저장하고, sleep\_list에 스레드를 삽입하고 thread\_block 함수를 호출하여 block 처리한다.

- 모든 스레드들의 공통 부모 스레드인 idle 스레드는 sleep 처리가 되지 않도록 assetion 처리하였다.

|  |
| --- |
| /\*\*  \* @brief thread를 재우고 blocked status로 만드는 함수  \*  \* @param ticks  \*/  void thread\_sleep (int64\_t ticks) {  enum intr\_level old\_level;  // off interrupt  old\_level = intr\_disable();  struct thread\* cur = thread\_current();  // idle 스레드 sleep 방지  ASSERT(cur != idle\_thread);  cur->wakeup\_time = ticks;  list\_push\_back(&sleep\_list, &cur->elem);  thread\_block();  // on interrupt  intr\_set\_level(old\_level);  } |

- sleep\_list에서 깨워줘야할 스레드를 unblock 상태로 만드는 함수를 추가 작성하였다. sleep\_list를 돌며 wakeup\_time이 현재 tick보다 같거나 작아지는 경우, sleep\_list에서 제거하고 ready status로 바꿀 수 있도록 thread\_unblock을 호출한다.

|  |
| --- |
| /\*\*  \* @brief sleep\_list에서 깨워야할 스레드들을 unblock(ready status) 처리  \*  \* @param ticks  \*/  void thread\_wake (int64\_t ticks) {  struct list\_elem \*e = list\_begin(&sleep\_list);  while (e != list\_end(&sleep\_list)) {  struct thread\* thd = list\_entry(e, struct thread, elem);  if (thd->wakeup\_time <= ticks) {  // 일어나야할 시간이 되면 sleep\_list에서 제거 및 unblock(to ready status) 처리  e = list\_remove(e);  thread\_unblock(thd);  } else {  // 아닐 경우 다음 element로 포인터 변경  e = list\_next(e);  }  }  } |

- list\_insert\_ordered 함수에 less parameter로 전달할 우선순위 비교 함수를 추가 작성하였다. 비교할 element 두 개를 받고 두 비교군의 우선순위를 판단하여 boolean 값을 리턴한다.

|  |
| --- |
| /\*\*  \* @brief 우선순위를 비교하는 함수  \*  \* @param a 첫번째 비교 대상  \* @param b 두번째 비교 대상  \* @param aux additional params  \* @return true  \* @return false  \*/  bool compare\_thread\_by\_priority(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED) {  return list\_entry(a, struct thread, elem)->priority > list\_entry(b, struct thread, elem)->priority;  } |

- 스레드들의 starvation을 방지하기 위한 aging 함수를 추가 작성하였다. 매 틱 마다 ready\_list의 스레드들의 priority를 1씩 올려준다.

|  |
| --- |
| /\*\*  \* @brief 스레드 starvation을 방지하기 위해 priority를 높여주는 함수  \*  \*/  void thread\_aging () {  struct list\_elem\* e;  for (e = list\_begin(&ready\_list); e != list\_end(&ready\_list); e = list\_next(e)) {  list\_entry(e, struct thread, elem)->priority++;  }  } |

- 실행 중인 스레드와 ready\_list의 맨 앞 스레드의 우선순위를 비교하여, 후자의 우선순위가 높을시 재스케줄링을 위해 thread\_yield를 호출해주는 함수를 작성하였다.

|  |
| --- |
| /\*\*  \* @brief 실행 중인 스레드와 ready\_list 맨 앞의 thread priority를 비교하여 선점 처리를 도와주는 함수  \*  \*/  void thread\_preempt () {  if (list\_empty(&ready\_list)) return;  struct thread\* first\_thd = list\_entry (list\_front (&ready\_list), struct thread, elem);  if (thread\_current ()->priority < first\_thd->priority) {  thread\_yield ();  }  } |

- 스레드 초기화 시에 sleep\_list를 함께 초기화시켜준다.

|  |
| --- |
| void  thread\_init (void)  {  ...  list\_init(&sleep\_list);  ...  } |

- 매 timer\_interrupt마다 호출되는 thread\_tick 함수에 aging 함수를 추가하였다.

|  |
| --- |
| void  thread\_tick (void)  {  ...  if (thread\_prior\_aging == true)  list\_init(&sleep\_list);  } |

- 새로운 커널 스레드가 만들어질 때 또는 러닝 스레드의 priority를 변경할 때, ready\_list에 담긴 스레드의 priority가 러닝 스레드보다 높을 수 있을 경우를 고려하여 선점 처리를 해줄 수 있는 함수를 추가한다.

|  |
| --- |
| tid\_t  thread\_create (const char \*name, int priority,  thread\_func \*function, void \*aux)  {  ...  /\* prj#3 \*/  // 새 priority를 가진 스레드가 만들어질 때, 더 높은 우선순위를 가진 스레드의 선점을 위해 비교  thread\_preempt();  ...  }  void  thread\_set\_priority (int new\_priority)  {  ...  /\* prj#3 \*/  // 새 priority를 설정할 때, 더 높은 우선순위를 가진 다른 스레드의 선점을 위해 비교  thread\_preempt();  } |

- ready\_list에 스레드를 삽입하는 부분이 작성된 thread\_yield와 thread\_unblock 함수에 list\_push\_back 대신 list\_insert\_ordered를 작성하여 list를 우선순위대로 정렬하며 엘리먼트를 삽입한다.

|  |
| --- |
| void  thread\_unblock (struct thread \*t)  {  ...  /\* prj#3 \*/  // as-is  // list\_push\_back (&ready\_list, &t->elem);  list\_insert\_ordered (&ready\_list, &t->elem, compare\_thread\_by\_priority, 0);  ...  }  void  thread\_yield (void)  {  ...  if (cur != idle\_thread) {  /\* prj#3 \*/  // as-is  // list\_push\_back (&ready\_list, &cur->elem);  list\_insert\_ordered (&ready\_list, &cur->elem, compare\_thread\_by\_priority, 0);  }  ...  } |

**1-3. /devices/timer.c**

- timer\_sleep 함수에서 busy waiting 방식 대신 blocked status를 활용하여 스레드를 재울 수 있도록 구현한 함수를 호출한다.

|  |
| --- |
| void  timer\_sleep (int64\_t ticks)  {  ...  /\* as-is \*/  // while (timer\_elapsed (start) < ticks)  // thread\_yield ();  // busy waiting 대신 sleep(blocked) 처리  thread\_sleep(start + ticks);  } |

- 매 틱마다 호출되는 timer\_interrupt에서 깨워줘야할 스레드들을 처리하기 위해 구현한 함수를 호출한다.

|  |
| --- |
| /\* Timer interrupt handler. \*/  static void  timer\_interrupt (struct intr\_frame \*args UNUSED)  {  ...  // timer interrupt가 걸릴 때마다 스레드들을 검사하여 깨우기  thread\_wake(ticks);  } |

**1-4. /threads/synch.c**

- condition의 waiters 리스트 내의 semaphore 들 중 우선순위가 높은 세마포를 깨우기 위한 작업을 구현하기 위해, list\_insert\_ordered 함수 내에서 사용할 비교 함수를 추가 구현하였다.

|  |
| --- |
| /\*\*  \* @brief condition간의 우선순위를 비교하는 함수  \*  \* @param a 첫번째 비교 대상  \* @param b 두번째 비교 대상  \* @param aux additional params  \* @return true  \* @return false  \*/  bool compare\_sema\_by\_priority (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED) {  // condition의 waiter (semaphore)  struct semaphore\_elem \*sema\_a = list\_entry(a, struct semaphore\_elem, elem);  struct semaphore\_elem \*sema\_b = list\_entry(b, struct semaphore\_elem, elem);  // semaphore의 waiters (thread)  struct list \*threads\_a = &(sema\_a->semaphore.waiters);  struct list \*threads\_b = &(sema\_b->semaphore.waiters);  return list\_entry(list\_begin(threads\_a), struct thread, elem)->priority > list\_entry(list\_begin(threads\_b), struct thread, elem)->priority;  } |

- sema\_down을 호출하는 스레드가 cs 영역에 진입하지 못할 시, sema->waiters에 삽입되는 로직을 우선순위를 고려하여 정렬 및 삽입하는 list\_insert\_ordered로 변경하였다.

|  |
| --- |
| void  sema\_down (struct semaphore \*sema)  {  ...  while (sema->value == 0)  {  /\* prj#3 \*/  // list\_push\_back (&sema->waiters, &thread\_current ()->elem);  list\_insert\_ordered (&sema->waiters, &thread\_current ()->elem, compare\_thread\_by\_priority, 0);  thread\_block ();  }  ...  } |

- cs 영역 수행을 마치고 sema\_up을 호출하는 스레드가 sema->waiters에서 맨 앞 스레드를 꺼내어 unblock 처리하는 로직을 변경하였다.

- thread\_unblock 전 waiters를 우선순위 내림차순으로 정렬한다.

- unblock 되는 스레드가 러닝 스레드보다 priority가 높을 경우를 고려하여 선점 작업을 도와주는 함수를 호출한다.

|  |
| --- |
| void  sema\_up (struct semaphore \*sema)  {  ...  if (!list\_empty (&sema->waiters)) {  list\_sort (&sema->waiters, compare\_thread\_by\_priority, 0);  thread\_unblock (list\_entry (list\_pop\_front (&sema->waiters), struct thread, elem));  }  ...  /\* prj#3 \*/  thread\_preempt();  ...  } |

- semaphore 엘리먼트를 waiting 배열에 삽입할 때, condition struct의 waiters(semaphore elements)들을 우선순위에 따라 정렬하고 삽입하는 코드로 변경하였다.

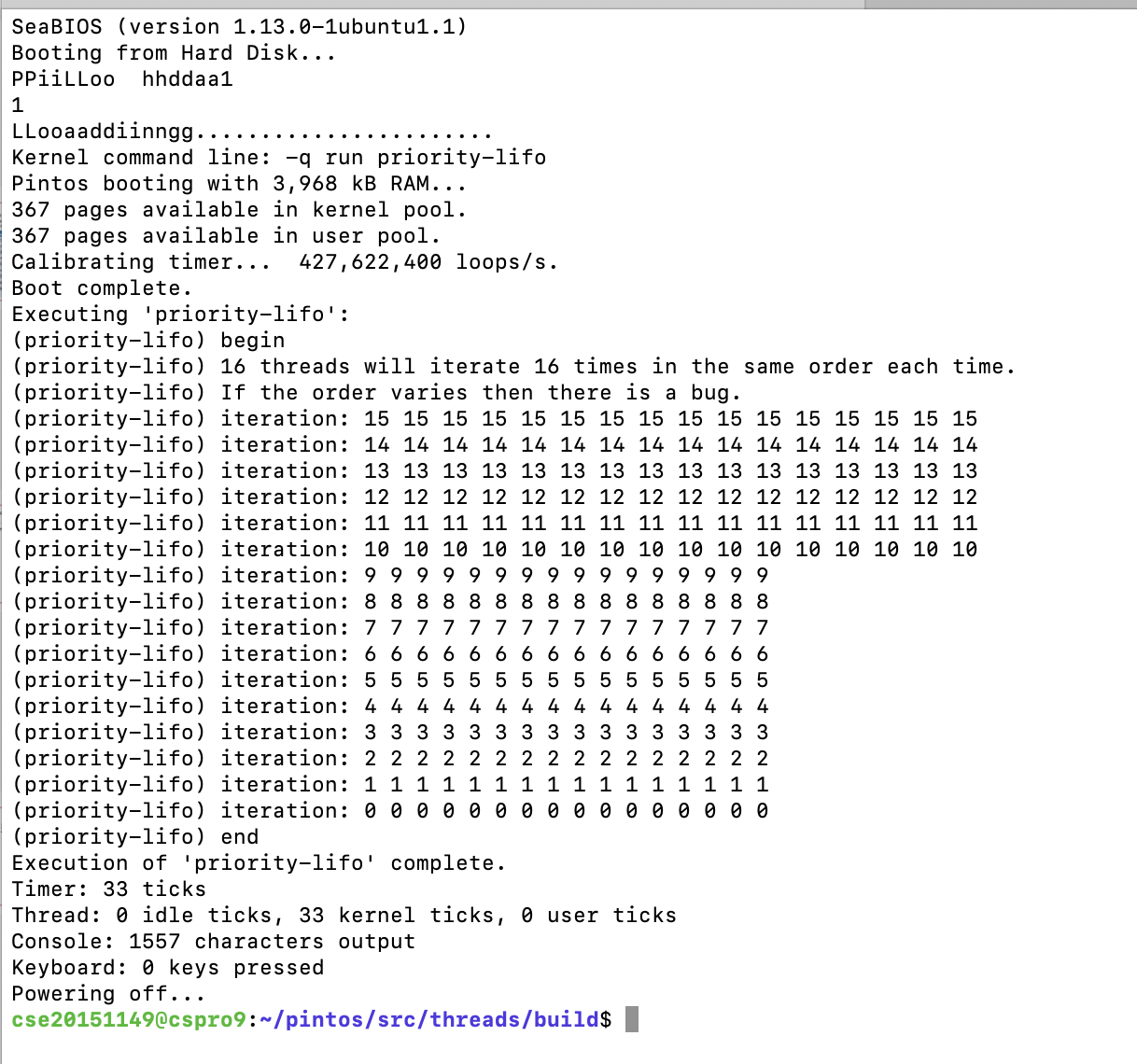
|  |
| --- |
| void  cond\_wait (struct condition \*cond, struct lock \*lock)  {  struct semaphore\_elem waiter;  ASSERT (cond != NULL);  ASSERT (lock != NULL);  ASSERT (!intr\_context ());  ASSERT (lock\_held\_by\_current\_thread (lock));    sema\_init (&waiter.semaphore, 0);  // list\_push\_back (&cond->waiters, &waiter.elem);  list\_insert\_ordered (&cond->waiters, &waiter.elem, compare\_sema\_by\_priority, 0);  lock\_release (lock);  sema\_down (&waiter.semaphore);  lock\_acquire (lock);  } |

- condition struct의 waiters(semaphore elements) 중 맨 앞의 세마포를 wake up 시켜주기 전에, 우선순위에 따라 정렬하는 함수를 추가한다.

|  |
| --- |
| void  cond\_signal (struct condition \*cond, struct lock \*lock UNUSED)  {  ASSERT (cond != NULL);  ASSERT (lock != NULL);  ASSERT (!intr\_context ());  ASSERT (lock\_held\_by\_current\_thread (lock));  if (!list\_empty (&cond->waiters)) {  list\_sort (&cond->waiters, compare\_sema\_by\_priority, 0);  sema\_up (&list\_entry (list\_pop\_front (&cond->waiters), struct semaphore\_elem, elem)->semaphore);  }  } |

* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

16개의 스레드를 우선순위를 증가시키며 실행하고, 나중에 실행시킨 thread가 가장 우선순위가 높으므로 스케줄링에 따라 가장 먼저 실행된다. priority scheduling가 정상적으로 잘 구현되었다면 아래와같이 우선순위가 높은 스레드들부터 실행되어 iteration을 돌며 스레드들이 순차적으로 종료될 것이다.



- 테스트 코드 수행 결과

