**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

학번 / 이름 : 20190258/김혜린

개발 기간 : 2021.12.03-2021.12.05

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

효율적인 시스템을 위해 기존의 Round-Robin scheduling 방식을 쓰는 pintos의 스케줄러를 Priority scheduling 방식으로 바꾸고 busy-waiting을 구사하는 비효율적인 Alarm-clock을 처리하며 나아가 BSD scheduler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock

기존의 pintos의 경우, timer sleep을 통해 sleep 상태로 들어간 thread에 대해 while문을 통해 반복적으로 wake up time이 될 때까지 계속 RUNNING status와 READY status를 왔다갔다하며 thread의 상태를 확인한다. 따라서 굉장히 많은 연산 비용이 소모되고 이를 BLOCKED status와 blocked\_queue를 이용해 효율적으로 관리하여 연산 시간을 줄인다.

* 1. Priority Scheduling

Round-Robin에 추가로 thread가 priority 순으로 실행될 수 있게 되고, preempted scheduling과 semaphore 상에서의 priority를 바탕으로 한 release를 구현할 수 있게 된다. 또한 추가로 aging함수를 통해 starvation을 해결할 수 있게 된다.

* 1. Advanced Scheduler (추가구현을 한 경우)

2)의 Priority Scheduling의 경우, aging을 하지 않을 경우, starvation이 발생하는 문제가 생기는데 Advanced Scheduler를 이용해 따로 aging을 하지 않아도 priority가 낮은 thread의 priority를 특정 주기에 따라 높이면서 이 현상을 해결할 수 있게 된다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

thread 구조체에 추가로 wakeup이라는 변수를 추가해 Blocked 상태로 넘어가기 전에 wakeup 시간을 계산해 저장한 뒤, blocked\_list에 저장한다. 그리고 time interrupt가 발생할 때마다 그때의 ticks와 blocked thread의 wakeup을 비교해 wakeup 시간이 지났을 경우, 해당 thread를 깨워 다시 ready 상태로 만들어준다.

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

Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우, preempted scheduling이 실행되며 이에 따라 running thread는 ready상태로 넘어가고 해당 thread가 즉시 running thread가 된다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

Nice, Recent CPU, load avg 세 개의 요소가 필요한데 nice는 thread의 첫 priority 값을 정해주는 -20에서 20 사이의 정수 상수값이며 recent cpu는 바로 직전에 사용한 cpu time을 저장하고 있으며 load avg는 최근 1분간 수행가능한 프로세스의 평균 개수이다. 이때 프로세스는 ready state에 있는 process로만 판단한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2021.12.03 pintos manual 확인 및 구현 내용에 대한 이해

2021.12.04 Alarm Clock, Priority Scheduling 구현

2021.12.05 BSD scheduling 구현

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수
  1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술

Alarm Clock 파트를 수정하기 위해서는 device/timer.c의 timer\_sleep함수와 timer\_interrupt 함수를 수정해야한다. 또한 struct thread에 wakeup이라는 int64\_t 타입의 변수를 추가해준다.

timer\_sleep에서 wake up time이 될 때까지 while문을 계속 돌던 것에서 blocked\_list라는 list 변수를 추가해 해당 리스트에 thread를 추가하고 thread를 blocked 상태로 만들어준다. 그냥 리스트에 삽입만 되면 언제 깨어나는지를 알 수 없으므로 struct thread에 추가한 wakeup 변수에 깨어나야하는 시간을 저장해주고 리스트에 push한다.

thread를 깨울지 확인하는 건 time interrupt가 발생해 커널모드로 진입했을 때로 blocked\_list를 돌며 thread의 wakeup 값이 ticks보다 작은지 확인한 뒤, 작다면 해당 thread를 list에서 제거하고 다시 ready 상태로 만들어준다.

* 1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.

threads/thread.c의 thread\_create에서 생성되는 thread에 priority를 저장하는데 이때 해당 priority와 thread\_current()의 priority를 비교해 생성되는 thread의 priority가 클 경우, thread\_yield 함수를 실행해 생성되는 thread가 바로 실행될 수 있도록한다.

thread\_yield함수는 마찬가지로 threads/thread.c에 구현되어 있는데 매번 ready\_list의 가장 앞 thread를 cpu에 할당한다. 따라서 priority 순으로 ready\_list를 정렬해줄 필요가 있다. 따라서 list\_insert\_ordered 함수에서 priority를 기준으로 정렬할 수 있도록 비교 함수를 구현해야한다.

* 1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

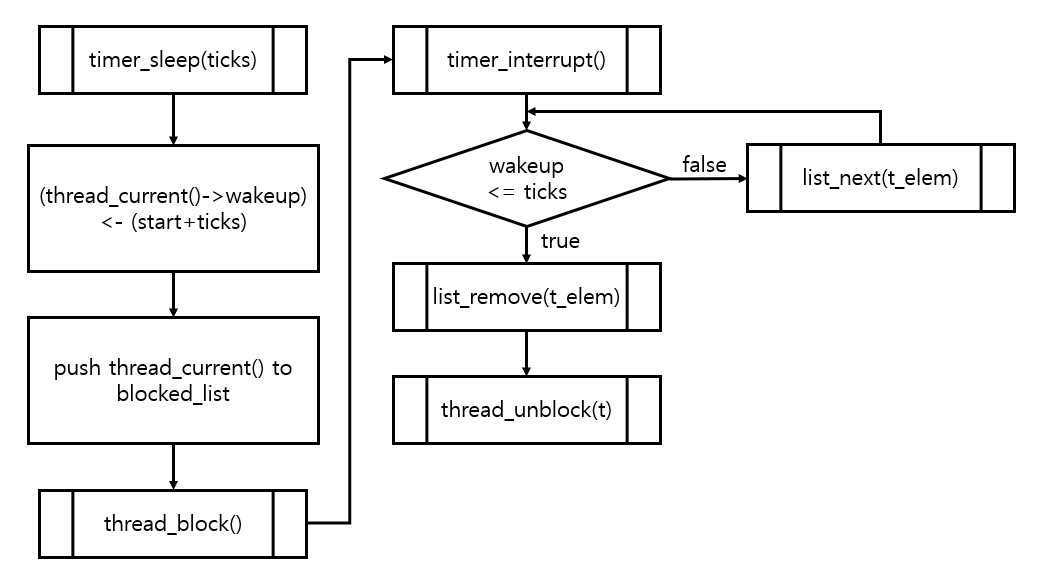
thread마다 nice와 recent\_cpu값을 고유하게 가지기 때문에 struct thread 구조체에 int 타입의 nice와 recent\_cpu를 멤버 변수로 추가한다. 또한 load\_avg도 pintos에 전역변수로 추가해주어야한다.

nice, recent\_cpu, load\_avg값을 set하고 get하기 위한 thread\_get\_nice, thread\_set\_nice, thread\_get\_load\_avg 그리고 thread\_get\_recent\_cpu 값을 구현해야한다. 또한 load\_avg, thread의 recent\_cpu, priority를 주기적으로 업데이트해주어야 하기 때문에 device/timer.c의 timer\_interrupt함수가 호출될 때마다 현재 thread의 recent\_cpu 값을 1추가하고 매초마다 load\_avg와 모든 thread의 recent\_cpu 값을 업데이트 해주어야 하므로 해당 함수도 구현해야한다. 또한 4 ticks마다 priority도 업데이트 해주어야 하기때문에 기존의 priority aging 함수를 바꾸어 priority를 업데이트 해주는 함수를 구현한다.

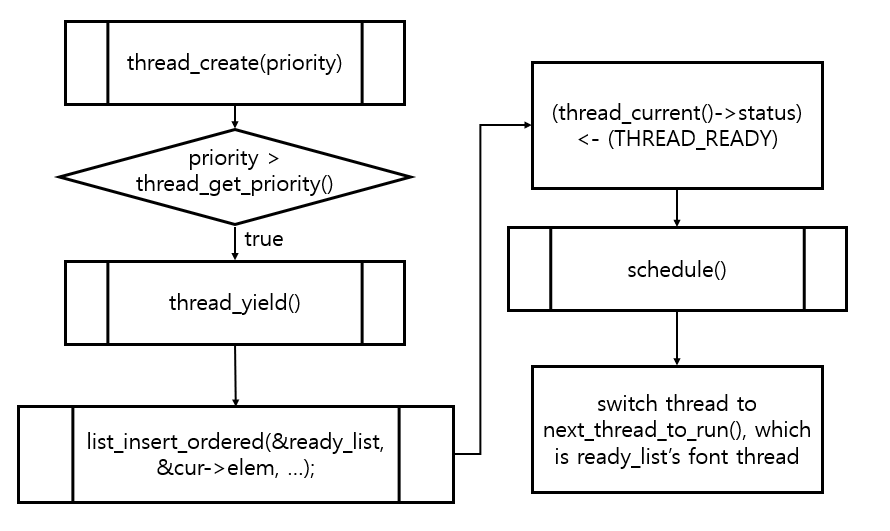
요소들의 업데이트 과정에서는 float 연산이 들어가는데 pintos는 float 연산을 지원하지 않으므로 pintos 매뉴얼에 나온 fixed point 연산을 매크로 함수로 구현해 요소들을 계산하는 함수를 구현한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)
  1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술



* 1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.



* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명
  1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술

blocked 상태의 thread를 따로 관리해주어야 하기 때문에 device/timer.c에 전역변수로 struct list blocked\_list를 선언해준 뒤, timer init에 이를 초기화해주는 함수를 추가한다.

|  |
| --- |
| void timer\_init (void)  {  pit\_configure\_channel (0, 2, TIMER\_FREQ);  intr\_register\_ext (0x20, timer\_interrupt, "8254 Timer");  // 3: 20190258  list\_init(&blocked\_list);  } |

이제 timer\_sleep 함수가 호출되면 현재 thread를 blocked\_list에 삽입하고 thread\_block을 호출해 해당 thread를 block 상태로 만들어주어야 하는데 이때 list.h에 정의된 list\_push\_back함수를 이용해 blocked\_list에 넣는 부분을 구현하였다.

|  |
| --- |
| void  timer\_sleep (int64\_t ticks)  {  //printf("timer sleep start\n");  int64\_t start = timer\_ticks ();  enum intr\_level old\_level;  ASSERT (intr\_get\_level () == INTR\_ON);  // 3: 20190258  old\_level = intr\_disable();  thread\_current()->wakeup = start+ticks;  list\_push\_back(&blocked\_list, &thread\_current()->elem);  thread\_block();  intr\_set\_level(old\_level);  } |

이때 blocked\_list에 현재 thread를 삽입하기 전에 해당 thread의 wakeup에 wake up time을 계산해 저장해주어야 한다. 이는 thread\_current()->wakeup = start+ticks; 을 통해 구현하였다.

* + 개발 과정에서 발생한 이슈

개발 과정에서 wakeup을 처음에 int 타입으로 하여 구현하였는데 ticks와 start의 타입이 int64\_t이고 연산 과정에서 32bytes를 넘는 경우가 발생하였는지 테스트 프로그램을 돌려본 결과 wake가 제대로 된 시간에 일어나지 않는 이슈가 발생하였다.

이제 blocked\_list에 저장된 thread들을 깨워야할지 확인하는 부분을 구현해야하는데 이 부분은 device/timer.c의 timer\_interrupt 함수에 구현되어 있다. 따라서 os는 timer interrupt가 발생할 때마다 blocked\_list의 각 thread를 돌며 해당 thread의 wakeup이 현재 ticks보다 작은지 확인한다. 현재 ticks보다 작다면 해당 thread는 blocked status에서 벗어나 ready status로 바뀌어야 하므로 해당 thread를 list에서 제거한 뒤, thread\_unblock 함수를 호출해 해당 thread를 다시 ready\_list에 넣어준다.

|  |
| --- |
| static void  timer\_interrupt (struct intr\_frame \*args UNUSED)  {  ticks++;  // 3: 20190258  struct list\_elem \*t\_elem = list\_begin(&blocked\_list);  // check wake time of thread in blocked list  while(t\_elem != list\_end(&blocked\_list)){  struct thread \*t = list\_entry(t\_elem, struct thread, elem);  int64\_t wakeup = t->wakeup;  if(wakeup <= ticks){  t\_elem = list\_remove(t\_elem);  thread\_unblock(t); // push t into ready\_list  }  else{  t\_elem = list\_next(t\_elem);  }  }  … |

* 1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.

priority preempt를 구현하기 위해 threads/thread.c의 thread\_create함수에 생성되는 thread의 priority가 현재 thread보다 클 경우 thread\_yield함수를 호출하는 코드를 추가하였다. 이때 현재 실행되는 thread의 priority는 therads/thread.c에 구현된 thread\_get\_priority함수를 이용해 가져왔다.

|  |
| --- |
| tid\_t  thread\_create (const char \*name, int priority,  thread\_func \*function, void \*aux)  {  …  /\* Add to run queue. \*/  thread\_unblock (t);  // printf("thread unblock finish\n");  // 3: 20190258  if(priority > thread\_get\_priority()) thread\_yield();  return tid;  } |

thread\_yield 함수가 실행되고 schedule 함수가 호출되면 next\_thread\_to\_run 함수가 다음에 실행되어야할 thread를 가져오는데 ready\_list의 가장 앞 element를 pop해온다. 따라서 ready\_list를 priority 순서로 정렬해 저장해줄 필요가 있어 thread\_yield와 thread\_unblock에서 ready\_list에 thread를 삽입할 때 priority를 기준으로 내림차순으로 정렬될 수 있도록 list\_insert\_ordered 로 list\_push\_back 함수를 변경해주었다.

|  |
| --- |
| void  thread\_yield (void)  {  struct thread \*cur = thread\_current ();  enum intr\_level old\_level;    ASSERT (!intr\_context ());  old\_level = intr\_disable ();  // 3: 20190258  if (cur != idle\_thread){  list\_insert\_ordered(&ready\_list, &cur->elem, thread\_priority\_cmp, NULL);  }  cur->status = THREAD\_READY;  schedule ();  intr\_set\_level (old\_level);  } |

|  |
| --- |
| 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);  // 3: 20190258  list\_insert\_ordered(&ready\_list, &t->elem, thread\_priority\_cmp, NULL);    t->status = THREAD\_READY;  intr\_set\_level (old\_level);  } |

또한 semaphore나 lock이 걸려 waiter에 thread가 들어가게 됐을 때도, 후에 thread release시 priority가 높은 순으로 release를 진행해주어야 시스템 전체에서 priority scheduling이 유지될 수 있다. 따라서 threads/synch.c의 sema\_down과 sema\_up에 각각 기존의 list\_push\_back과 list\_pop\_front에서 list\_push\_back은 list\_inserted\_ordered로 변경하는 것으로 sema\_up의 경우 혹시 priority가 바뀌었을 경우를 대비해 list\_sort 함수를 호출해 priority 내림차순으로 thread가 정렬될 수 있도록 코드를 수정하였다.

|  |
| --- |
| void  sema\_down (struct semaphore \*sema)  {  enum intr\_level old\_level;  ASSERT (sema != NULL);  ASSERT (!intr\_context ());  old\_level = intr\_disable ();  while (sema->value == 0)  {  // list\_push\_back (&sema->waiters, &thread\_current ()->elem);  // 3: 20190258  list\_insert\_ordered(&sema->waiters,&thread\_current()->elem, thread\_priority\_cmp, NULL);  thread\_block ();  }  sema->value--;  intr\_set\_level (old\_level);  } |
| void  sema\_up (struct semaphore \*sema)  {  enum intr\_level old\_level;  ASSERT (sema != NULL);  old\_level = intr\_disable ();  //printf("waiter number: %d\n", list\_size(&sema->waiters));  if (!list\_empty (&sema->waiters)) {  list\_sort (&sema->waiters, thread\_priority\_cmp, NULL);  thread\_unblock (list\_entry (list\_pop\_front (&sema->waiters),  struct thread, elem));  }  sema->value++;  intr\_set\_level (old\_level);  thread\_yield();  } |

list\_inserted\_order과 list\_sort 함수를 호출할 때에는 각 element를 비교할 수 있는 비교함수를 인자로 넣어주어야하기 때문에 list\_elem가 임베디드된 struct thread를 찾아 priority를 비교해주는 thread\_priority\_cmp 함수를 threads/thread.c에 추가로 구현해주었다.

|  |
| --- |
| bool  thread\_priority\_cmp(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux)  {  struct thread \*t\_a = list\_entry(a, struct thread, elem);  struct thread \*t\_b = list\_entry(b, struct thread, elem);  return t\_a->priority > t\_b->priority;  } |

해당 함수의 프로토타입은 list.h에 list\_less\_func으로 정의되어 있다.

priority 스케줄링만 구현할 경우, aging 함수를 추가로 구현해주어야 하는데 Advanced Scheduler까지 구현하였기 때문에 aging 대신 nice, recent\_cpu, load\_avg를 바탕으로 priority 함수를 업데이트 하는 방식으로 최종 구현되었다. 따라서 이에 대한 설명은 3)에서 설명할 예정이다.

* 1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

load\_avg, recent\_cpu, nice 값을 저장해주어야 하기 때문에 load\_avg는 threads/thread.c에 전역변수로 선언해주고 recent\_cpu, nice의 경우 thread 단위로 관리 되어야 하기 때문에 struct thread에 멤버 변수로 추가해주었다.

|  |
| --- |
| struct thread  {  // 3: 20190258  int64\_t wakeup; /\* wake up time when thread go to sleep \*/  int recent\_cpu;  int nice;  … |

pintos 매뉴얼에 따르면 initial\_thread의 경우, recent\_cpu와 nice를 0으로 설정하고 부모가 있는 프로세스의 경우 부모 프로세스의 값을 물려받는다고 설명되어 있어서 thread\_init함수에 initial\_thread의 recent\_cpu와 nice를 0으로 설정하는 코드를 삽입하고 init\_thread에서 parent에 생성되는 thread를 삽입한뒤, 해당 thread가 부모의 recent\_cpu와 nice값을 물려받을 수 있도록 코드를 추가하였다.

|  |
| --- |
| void  thread\_init (void)  {  ASSERT (intr\_get\_level () == INTR\_OFF);  lock\_init (&tid\_lock);  list\_init (&ready\_list);  list\_init (&all\_list);  /\* Set up a thread structure for the running thread. \*/  initial\_thread = running\_thread ();  // 3: 20190258  init\_thread (initial\_thread, "main", PRI\_DEFAULT);  initial\_thread->recent\_cpu = 0;  initial\_thread->nice = 0;  initial\_thread->status = THREAD\_RUNNING;  initial\_thread->tid = allocate\_tid ();  } |
| static void  init\_thread (struct thread \*t, const char \*name, int priority)  {  …  // 3: 20190258  t->recent\_cpu = running\_thread()->recent\_cpu;  t->nice = running\_thread()->nice;  … |

threads/thread.c의 아직 구현되어 있지 않은 thread\_get\_nice, thread\_set\_nice, thread\_get\_load\_avg 그리고 thread\_get\_recent\_cpu 함수들을 구현해 thread의 recent\_cpu 값과 nice 값 그리고 시스템의 load\_avg 값을 가져올 수 있도록 하였다.

|  |
| --- |
| void  thread\_set\_nice (int nice)  {  //printf("set nice: %d\n", nice);  struct thread \*cur = thread\_current ();  cur->nice = nice;  int new\_priority = get\_new\_priority(cur);  thread\_set\_priority(new\_priority);  } |
| int  thread\_get\_nice (void)  {  return thread\_current()->nice;  } |
| int  thread\_get\_load\_avg (void)  {  return ROUND\_TO\_N(MUL\_FI(load\_avg, 100));  } |
| int  thread\_get\_recent\_cpu (void)  {  return ROUND\_TO\_N(MUL\_FI(thread\_current()->recent\_cpu, 100));  } |

thread\_set\_nice 함수의 경우 인자로 넘어온 nice 값을 현재 thread에 할당한 뒤, 이를 바탕으로 다시 priority를 계산해 thread\_set\_priority 함수를 호출해 새 priority를 thread에 할당한다.

thread\_get\_nice 함수의 경우 현재 thread의 nice 값을 return해주고 thread\_get\_load\_avg와 thread\_get\_recent\_cpu 함수의 경우 강의자료와 핀토스 매뉴얼에 나온대로 fixed point를 계산하는 매크로 함수를 구현하여 해당 값들을 계산해 반환해주었다.

이제 load\_avg와 recent\_cpu를 주기적으로 업데이트해주어야 하는데 주기는 매초마다이며 따라서 timer\_interrupt에 매초마다 두 요소를 업데이트해주는 함수를 구현해 실행시켜주었다. 또한 현재 실행되고 있는 thread의 경우 timer interrupt가 발생할 때마다 recent\_cpu를 추가해주어야 하기 때문에 이도 device/timer.c의 timer\_interrupt함수에 추가로 구현해주었다. thread priority의 경우 4 ticks마다 업데이트 해주어야 하기 때문에 이또한 timer\_ticks() % 4의 조건문을 이용해 확인한 뒤, thread\_aging\_all()이라는 priority update 함수를 구현해 모든 thread의 priority를 주어진 식에 따라 계산해주었다.

|  |
| --- |
| static void  timer\_interrupt (struct intr\_frame \*args UNUSED)  {  …  // priority aging & thread\_mlfqs  if(thread\_prior\_aging || thread\_mlfqs){  // increase recent\_cpu one  thread\_current()->recent\_cpu = ADD\_FI(thread\_current()->recent\_cpu, 1);  if(timer\_ticks() % TIMER\_FREQ == 0){  // update load\_avg  update\_load\_avg();  // recalculate recent\_cpu  recal\_recent\_cpu();  }  // every 4 ticks, aging  if(timer\_ticks() % 4 == 0){  thread\_aging\_all();  }  }  thread\_tick(); |
| void  update\_load\_avg(void)  {  int ready\_threads = list\_size(&ready\_list);  if (thread\_current() != idle\_thread) ready\_threads++;  load\_avg = DIV\_FI(ADD\_FI(MUL\_FI(load\_avg, 59), ready\_threads),60);  } |
| void  recal\_recent\_cpu(void)  {  struct list\_elem \*elem = list\_begin(&all\_list);  while(elem != list\_end(&all\_list)){  struct thread \*t = list\_entry(elem, struct thread, allelem);  t->recent\_cpu = ADD\_FI(MUL\_FF(DIV\_FF(MUL\_FI(load\_avg, 2), ADD\_FI(MUL\_FI(load\_avg, 2), 1)), t->recent\_cpu), t->nice);  elem = list\_next(elem);  }  } |
| void  thread\_aging\_all()  {  //printf("thread aging\n");  struct thread \*t;  struct list\_elem \*t\_elem = list\_begin(&all\_list);  while(t\_elem != list\_end(&all\_list)){  t = list\_entry(t\_elem, struct thread, allelem);  t->priority = get\_new\_priority(t);  t\_elem = list\_next(t\_elem);  }  if(thread\_current()->priority < list\_entry(list\_begin(&ready\_list), struct thread, elem)->priority)  intr\_yield\_on\_return ();  } |
| int get\_new\_priority(struct thread \*t)  {  int new\_priority = ROUND\_TO\_Z(SUB\_FF(SUB\_FF(CONV\_TO\_F(PRI\_MAX), DIV\_FI(t->recent\_cpu, 4)), MUL\_FI(CONV\_TO\_F(t->nice), 2)));  if(new\_priority > PRI\_MAX)  new\_priority = PRI\_MAX;    if (new\_priority < PRI\_MIN)  new\_priority = PRI\_MIN;  return new\_priority;  } |

update\_load\_avg, recal\_recent\_cpu 와 thread\_aging\_all 함수의 경우 threads/thread.c에 구현되어 있다. 세 함수 모두 pintos 매뉴얼과 강의자료에 나와있는 수식을 fixed points 연산을 구현한 매크로 함수를 이용해 계산하였다.

recal\_recent\_cpu와 thread\_aging\_all함수는 호출될 때마다 전체 thread의 값을 변경해주어야 하기 때문에 all\_list를 이용해 각 thread의 recent\_cpu와 priority를 바꾸어 주었다.

priority의 경우 get\_new\_priority 함수를 추가로 구현해 강의자료와 매뉴얼대로 계산한 뒤, PRI\_MIN보다 작을 경우 PRI\_MIN으로 PRI\_MAX보다 클 경우 PRI\_MAX로 바꾸어주는 코드까지 하나의 함수 안에 구현하였다.

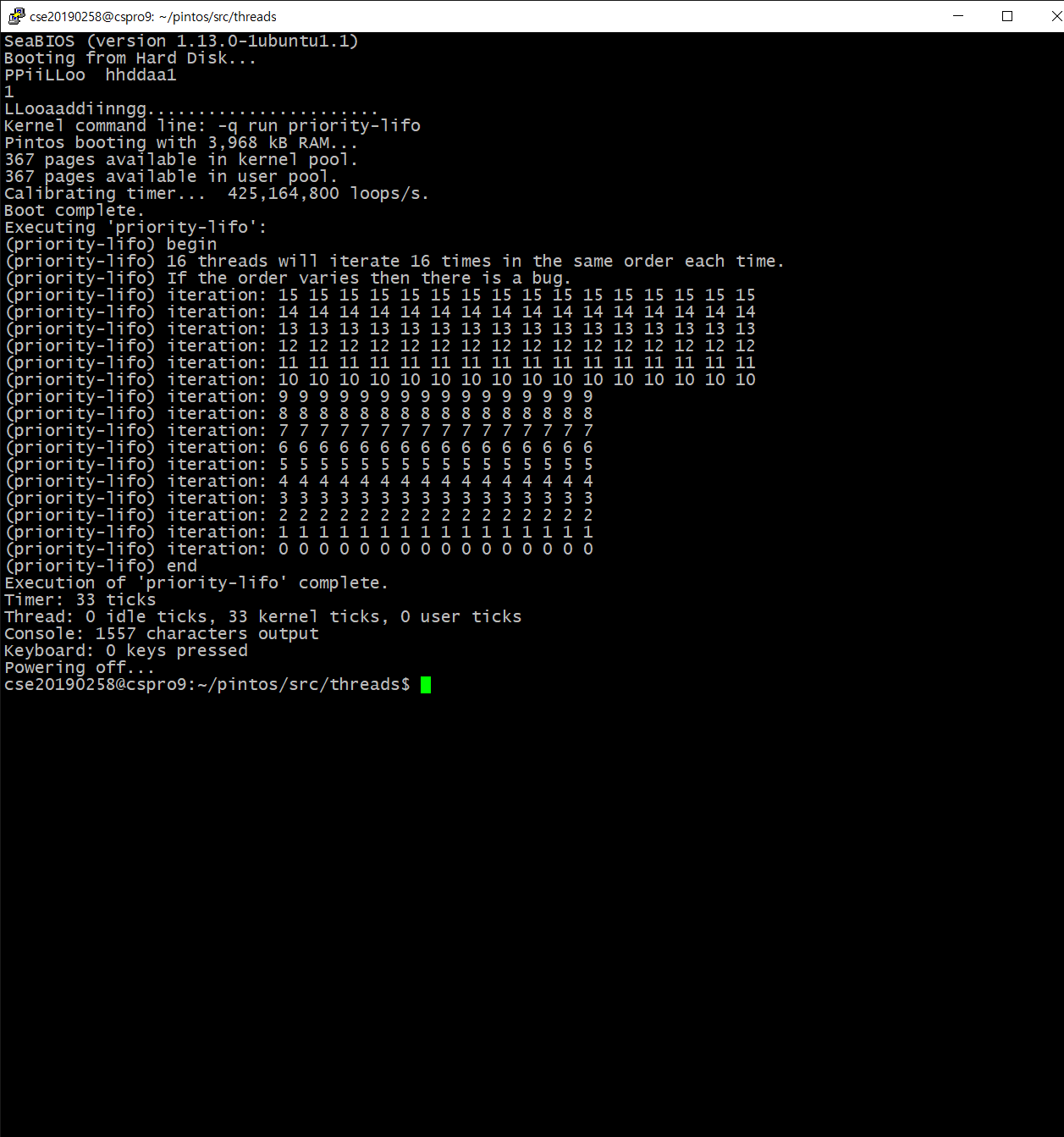
fixed points 연산을 구현한 매크로 함수들은 아래와 같다.

|  |
| --- |
| #define FRACTION (1<<14)  #define CONV\_TO\_F(n) ((n)\*(FRACTION))  #define ROUND\_TO\_Z(x) ((x)/(FRACTION))  #define ROUND\_TO\_N(x) ((x>=0)?((x+FRACTION/2)/FRACTION) :((x-FRACTION/2)/FRACTION))  #define ADD\_FF(x, y) (x)+(y)  #define SUB\_FF(x, y) (x)-(y)  #define MUL\_FF(x, y) (((int64\_t)x)\*(y)/(FRACTION))  #define DIV\_FF(x, y) (((int64\_t)x)\*(FRACTION)/(y))  #define ADD\_FI(x, n) ((x)+(n)\*(FRACTION))  #define SUB\_FI(x, n) ((x)-(n)\*(FRACTION))  #define MUL\_FI(x, n) ((x)\*(n))  #define DIV\_FI(x, n) ((x)/(n)) |

* + 개발 과정에서 발생한 이슈

개발 과정에서 테스트 프로그램을 돌려봤을 때 이론상 통과해야하는데 자꾸 실패하여 printf를 이용해 로그를 찍어보았는데 wakeup 값이 갑자기 변하는 것을 확인할 수 있었다. 왜 이런 일이 발생하는지 에러를 찾아본 결과 recent\_cpu를 계산하는 과정에서 list와 잘못된 list\_elem 변수를 이용해 list\_entry를 찾아 연산을 진행해 결과적으로 다른 변수의 값을 건드리는 일이 발생하였다.

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



thread의 priority를 점점 높여가며 16개의 thread를 실행시켰을 때 나중에 실행시킨 thread가 priority scheduling에 따라 먼저 실행되어 iteration을 돌고 종료하는지 확인하는 테스트 프로그램이다.

* make check 수행 결과를 캡처하여 첨부

