**Pintos Project 3: Threads**

**(설계 프로젝트 수행 결과)**

과목 명 : 운영체제

담당 교수 :김영재 교수님

조 / 조원 : 58조 / 20151601(장진영), 20151600(장승민)

개발 기간 :2019.11.25 ~2019.12.07

**프로젝트 제목 : Pintos Project 3 Threads**

**제출일 :2019.12.08**

**참여 조원 :20151601 장진영 / 20151600 장승민**

1. **개발 목표**

이번 프로젝트의 목표는 pintos에서의 특정 Thread 관련 기능인 Alarm Clock, Scheduler를 구현해야 한다. Thread가 일정 부분을 반복 수행하면서 대기하는 상태인 Busy Waiting을 하면서 CPU time을 낭비하는 문제점을 해결하기 위해 Busy waiting 방식으로 구현되어 있는 함수를 설정된 시간이 지나지 않으면 thread를 BLOCK상태로 만들어 놓은 후, wake up시간이 되면 READY상태로 만들어 ready queue에 삽입하는 방식으로 개선하고, 또한 새로 입력되는 thread를 차례로 ready queue의 맨 끝에 추가하고 각 프로세스마다 우선순위를 부여하여 우선순위에 따라 수행하는 priority scheduling을 BSD scheduler 방법으로 구현한다. 이 때, starvation 현상을 막기 위하여 process 생성 후 흐른 시간에 비례하여 prioirty를 높여주는 aging 기법 또한 사용한다. 이러한 방법들을 통해 thread를 효율적으로 관리하는 방법에 대해서 이해한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **1. Alarm Clock**
* **2. Priority Scheduling & Aging**
* **3. BSD Scheduler** 
  1. **개발 내용**
     1. Alarm Clock
     2. 원래 pintos의 코드 중에서 timer.c의 timer\_sleep()함수에서 구현되어 있는 busy-waiting방식은 while문을 이용하여 특정 thread가 일정 부분을 계속해서 반복 수행하면서 설정된 시간이 지나가기를 기다리도록 구현되어 있어 CPU time의 낭비를 초래한다. 그러므로 Busy Waiting을 해결하기 위해 Alarm Clock을 구현한다. Alarm Clock 방법은 시간 체크 후 설정한 시간이 되지 않았으면 Thread를 BLOCK상태로 만들고, 추후 설정된 시간이 되었을 때 interrupt를 발생시켜 해당 thread를 unblock시켜서 READY 상태로 만들어 Ready Queue(ready\_list)에 삽입해주는 방법을 사용한다.
     3. Priority Scheduling & Aging
        + 각 프로세스마다 우선순위를 부여하여 우선순위가 높은 thread부터 수행하는 scheduling 기법이다. 현재 pintos는 FIFO 형식으로 구현되어있고 각 thread마다 priority를 설정해주어야 한다. 우선 ready\_list에 우선순위에 맞게 list에 push해주기 위하여 list\_insert\_ordered() 함수를 이용하여 우선순위에 따라 ready\_list에 insert 해주어야한다. semaphore 사용시, semaphore list가 우선순위에 따라서 push 또는 remove가 되어야 한다. 그리고 우선순위가 낮은 thread들은 우선순위가 높은 thread들에 밀려 실행이 되지 않을 경우인 starvation 현상을 방지하여야 한다. 따라서 thread 생성 후 흐른 시간에 비례해서 우선순위를 높여주는 aging 기법을 사용하여야 한다. 또한 새로운 thread가 생성될 때마다 thread를 생성하는 thread의 priority와 비교하여 다시 schedule한다.
     4. BSD Scheduler
        + BSD Scheduler의 목적으로는 thread의 다른 scheduling needs를 밸런스있게 스케쥴링하는 스케쥴러이다. Thread의 CPU의 사용시간을 추정하는 값으로, 최근의 cpu 사용시간에 가중치를 두어 계산하여 time1(recent\_cpu)에 저장하고, time interrupt가 발생될 때 마다 idle thread를 제외한 running상태의 thread들의 time1 값을 1씩 증가시킴으로써 cpu time이 많은 thread는 다음 스케쥴링 결정 시 낮은 우선순위를 할당한다. 그리고 ready 상태의 thread 수의 평균을 추정함으로써 사용하는 load\_avg를 설정하여 TIMER\_FREQ마다 모든 thread들의 recent\_cpu와 load\_avg값을 재계산 해야 한다. load\_avg는 booting시 0으로 초기화시킨다. recent\_cpu와 load\_avg를 구하는 식은 아래와 같다.

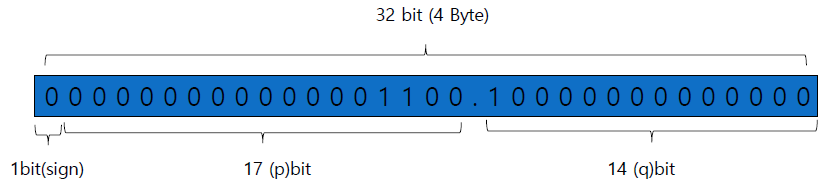
• *load\_avg* = (59/60) \* *load\_avg* + (1/60) \* *ready\_threads*

• *time1 = (2\*load\_avg)/(2\*load\_avg+1)\*time1+ time2(nice)*

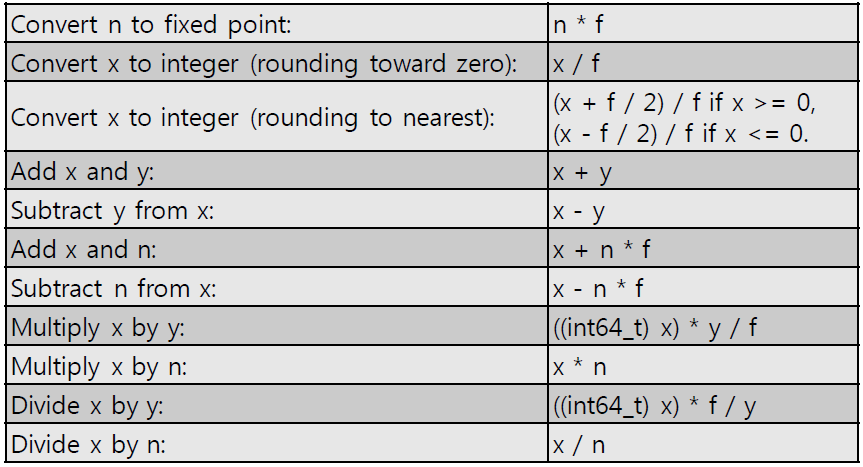
1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 1. 2019.11.25 : 강의자료 이해
* 2. 2019.11.25 ~ 2019.11.28 : alarm clock 및 priority scheduling 구현
* 3. 2019.11.29 ~ 2019.12.02 : aging 구현
* 4. 2019.12.03 ~ 2019.12.06 : BSD 스케줄러 구현 및 코드 리팩토링
* 5. 2019.12.07 : 코드 리팩토링 및 보고서 작성
  1. **개발 방법**
     1. **Alarm clock**
        + - devices/timer.c 안에서의 timer\_sleep()은 현재 busy waiting 방식으로 구현되어 있다. 이를 alarm clock 방식으로 구현하여야 한다. parameter로 넘어온 현재 thread의 tick에다가 os가 부팅된 후 부터의 시간을 알 수 있는 함수인 timer\_ticks()를 호출하여 return 값을 start에 저장한 후 더해서 현재 thread가 깨어나야 하는 시간을 구한 후 wakeTime이라는 변수에 저장을 한다. 즉 wakeTime 변수는 현재 thread가 깨어나야 하는 시간을 설정한다. 그리고 thread를 block하기 전에 sleep이라는 list에 list\_push\_back함수를 이용하여 넣어주는데 이때 기존의 sleep list안에 있는 thread들과 t->elem을 넣어준다.
     2. **Priority Scheduling & Aging**
        + - 원래의 pintos는 FIFO 형식으로 스케쥴링 되어있었기 때문에 thread가 ready상태가 될 때 ready list의 맨 뒤에 insert되는 구조였다. 그러나 우선순위 기법을 적용하기 위하여 thread 구조체의 priority 변수를 사용하여야 한다. 각 thread에 priority를 적용함으로써 우선순위에 대하여 내림차순으로 ready\_list에 insert 해야한다. 그래서 list\_push\_back()대신 list\_insert\_ordered()를 사용하여 priority가 큰 순서대로 insert한다. thread.c의 thread\_unblock() thread\_yield()에서도 기존에 사용한 list\_push\_back대신에 list\_insert\_ordered()로 priority순서대로 insert되게끔 설정하였다. 또한 thread\_create()에서 priority를 비교를 통해 priority가 제일 큰 thread가 수행 되기 위해 thread\_yield()가 호출 된다.
          - semaphore를 사용하는 경우를 위하여, thread/synch.c안의 sema\_down()함수에서 list\_push\_back을 활용하여 삽입했던 ready\_list를 list\_insert\_ordered()를 사용하여 prioirty를 비교하여 삽입해주었고, sema\_up() 함수에서는 sema up을 할 경우 priority가 큰 thread를 실행시키기 위해 re-scheduling 해주는 thread\_yield()함수를 사용하였다.
          - aging은 명세서에 적혀있는 대로 pintos에서 -aging 옵션을 주었을 때만 aging flag를 true로 set하여 thread\_aging()을 호출하는 기능을 추가하였다. thread\_aging()에서는 thread tick마다 ready\_list에 존재하는 모든 thread의 priority를 일정량 증가시켜준다.
     3. **BSD Scheduler**
        + thread\_mlfqs가 false일 때는 기존에 존재하는 round-robin scheduler방식으로 진행하고, true로 set될 때는 multi-level feedback queue scheduler방식으로 진행해야 하므로 이 때만 BSD Sceduler가 실행되도록 구현하였다. mlfqs를 수행하기 위해서는 priority, time1(recent\_cpu), load\_avg, time2(nice)를 적절하게 설정해야 한다. 따라서 initial\_thread 구조체 안에 time2(nice)와 time1(recent\_cpu)를 추가했고, 전역변수로 load\_avg을 추가하였다. Nice값은 time2 변수로 할당하였고, recent\_cpu 변수는 time1 변수로 할당하였다.priority는 주어진 수식 *PRI\_MAX - (recent\_cpu(time1)/4) - (nice(time2)\*2)*을 사용하여 재계산을 진행해야 한다. recent\_cpu(time1)는 TIMER\_FREQ마다 주어진 수식 *recent\_cpu(time1) = (2\*load\_avg) / (2\*load\_avg+1) \* recent\_cpu(time1) + nice(time2)* 재계산해야하며, recent\_cpu(time1)를 계산할 때 필요한 ready 상태의 thread들 수의 평균을 나타내는 load\_avg 또한 주어진 수식 *load\_avg* = (59/60) \* *load\_avg* + (1/60) \* *ready\_threads*를 통해 TIMER\_FREQ마다 업데이트 해야 한다. pintos kernel에서는 실수 연산을 지원하지 않기 때문에 BSD scheduling에서 필요한 실수연산은 fixed-point형식을 사용해서 계산하였다. fixed-point 형식은 맨 앞 1bit는 sign값을 17bit는 정수부를 14bit는 소수부를 나타낸다. 뒤의 소수부의 크기만큼을 16진수로 나타내면 0x4000이기 때문에 recent\_cpu(time1)와 load\_avg를 계산할 때 pintos manual에 나와 있는 방식 그대로 0x4000을 곱하고 나눈다.

아래의 그림을 보면, fixed-point 형식에 대해 더 잘 이해할 수 있다.



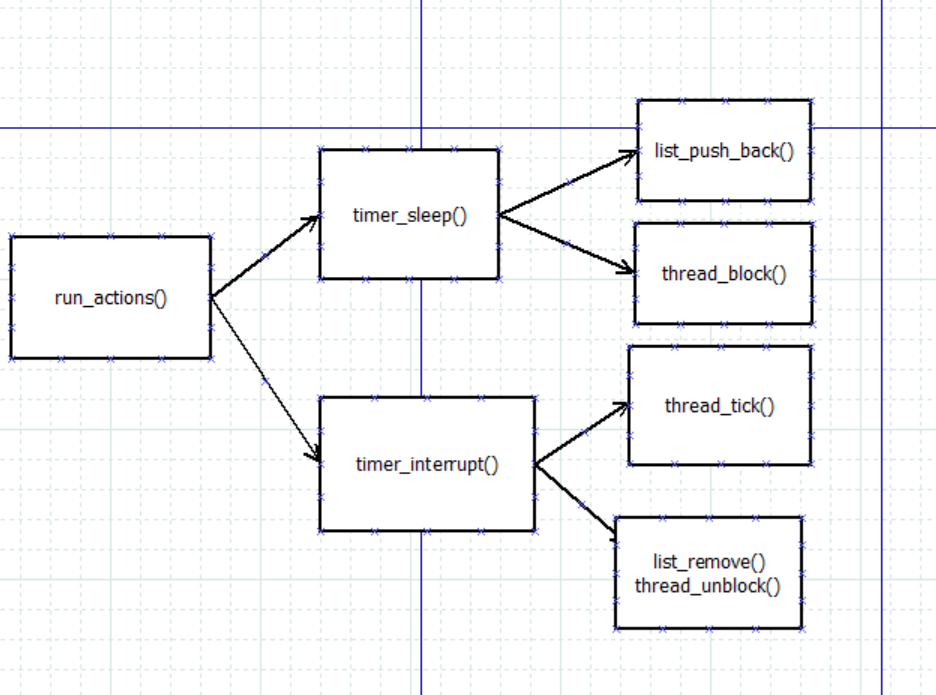
연산에 대한 방법은 아래의 그림과 같다.



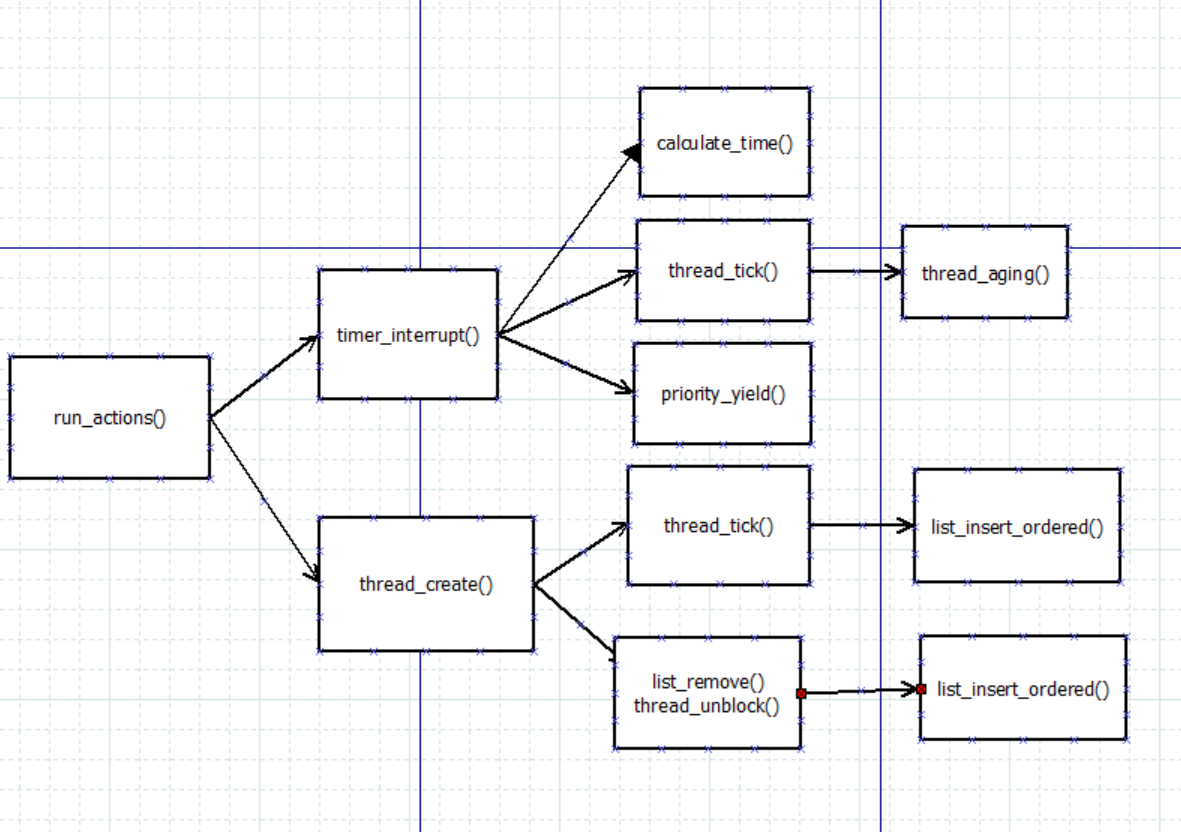
* 1. **연구원 역할 분담**

같은 장소에서 서로 개념을 이해하고 함께 소통하면서 진행하였다. 따로 명확한 역할을 나눈 것이 아니고 서로에게 도움을 주며 모르는 부분에 대해서 피드백하고 대화하면서 진행하였다.

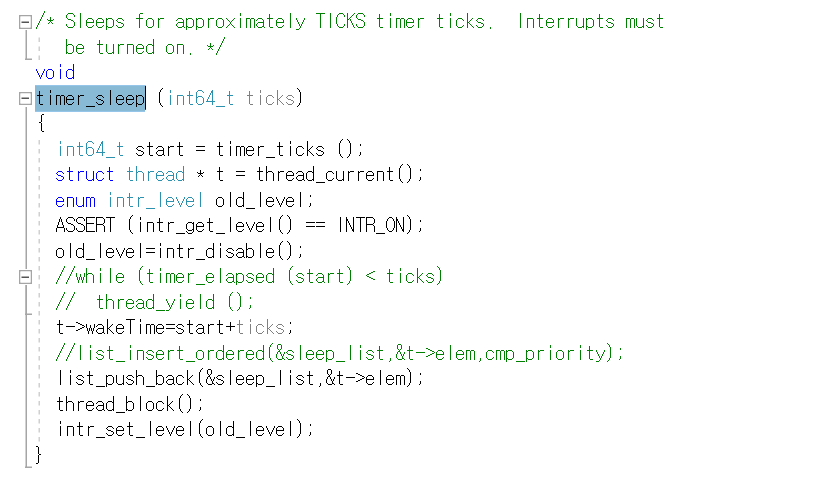
1. **연구 결과**
   1. **합성 내용**

****

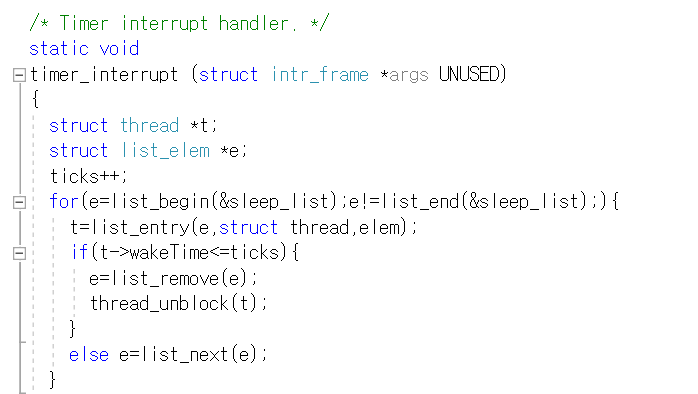
**-Alarm Clock 구현 흐름도**

****

* **Priority Scheduling & BSD scheduler & Priority aging 구현 흐름도** 
  1. **제작 내용**
     1. **Alarm Clock - <src/devices/timer.c>**
        + - **void timer\_sleep(int64\_t ticks)**

****

* \* 함수 설명 : Thread마다 실행 시간을 계산하여 wakeTime 변수에 저장하고 thread를 sleep시키는 함수이다.
* \* 사용 변수
* int64\_t start : 현재 시간을 읽어 저장하는 함수이다.
* enum intr\_level old\_level : interupt on, off를 저장하는 변수
* \* 내용 : t구조체 안에 wakeTime이라는 변수를 설정하였다. 이 변수는 Thread마다의 일어나야 하는 시간을 저장해두는 변수이므로, 현재 시간인 start와 잠들어있을 시간인 ticks를 더해 저장해주었다. 그리고 sleep 상태에 있는 sleep list에 wakeTime 순서대로 저장해준다. 이 때, list\_push\_back 함수를 활용하였다.
  + - * **struct void timer\_interrrupt ( struct intr\_frame \*args UNUSED)**

****

\* 함수 설명 : timer\_interupt handler이다. 매 tick마다 실행되며 wakeTime 시간이 된 thread를 찾아 깨워주는 역할을 한다.

\* 사용 변수 : 없음

\* 내용 : tick을 하나씩 더해주며 sleep 상태에 있는 thread의 list인 sleep list에서 wakeTime 시간이 ticks보다 큰경우 list에서 remove해주고 thread의 block을 풀어준다.

**2.Priority Scheduling & Aging**

* + - * + **void thread\_aging()**

****

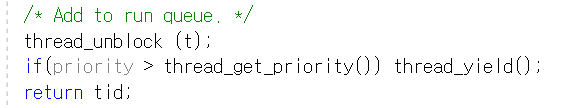
\* 함수 설명 : process 생성 후 흐른 시간에 비례하여 우선순위를 높여주는 기법인 aging을 구현한 함수이다.

\* 사용 변수

struct list\_elem \*e : thread를 list형식으로 이은 list\_elem을 임시 저장해놓기 위한 변수

\* 내용 : 현재 ready\_list(Ready Queue)에 있는 thread의 priority를 1씩 늘려줌으로써 우선순위를 높여주었다.

* + - * **tid\_t thread\_create(const char \*name,int priority,thread\_func \*function, void \*aux)**

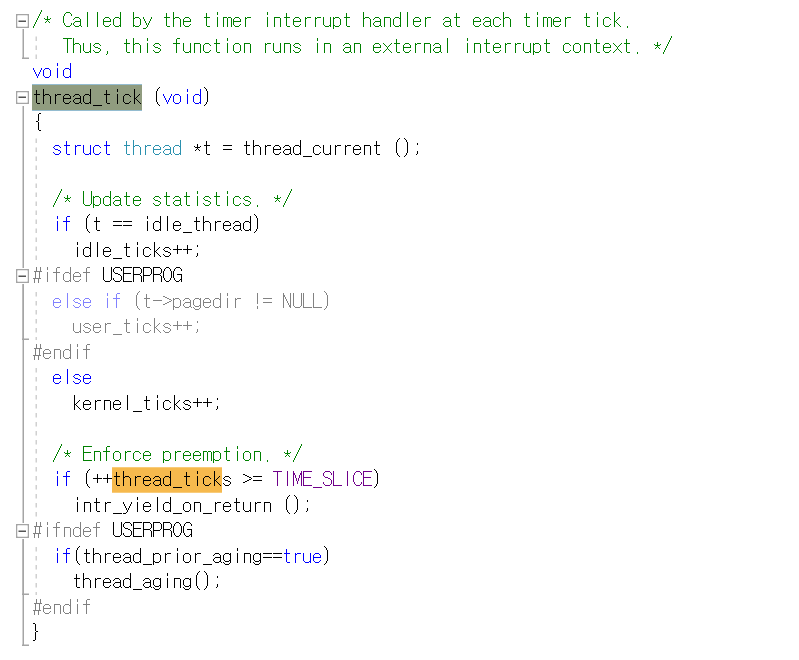


\* 함수 설명 : thread를 생성할 때 BSD 스케쥴링으로 priority를 설정을 해주는데, 현재 실행되고 있는 우선순위와 비교하여 다른 Thread에게 CPU를 양보한다.

\* 사용 변수 : 없음

\* 내용 : 위의 코드는 project 3에 추가한 함수이고, 생성한 thread와 현재 실행중인 thread의 우선순위를 비교하여 생성한 thread의 priority가 더 큰 경우 reschedule을 위해 thread\_yield() 함수를 호출해준다.

* + - * **void thread\_tick(void)**



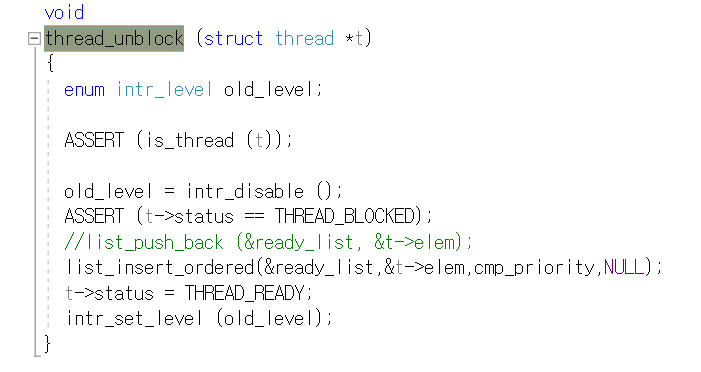
\* 함수 설명 : timer tick마다 timer interrupt handler에 의해 호출되는 함수이다. 매 tick마다 aging 기법을 도입하였다.

\* 사용 변수

struct thread \* t : thread를 임시 저장해놓기 위한 변수

\* 내용 : timer tick마다 tick을 ++해주며, 명세서에 적혀있듯이 aging 기법을 활용한 testcase에 한해 thread\_aging() 함수를 호출해주었다

* + - * **void thread\_unblock(struct thread \*t)**



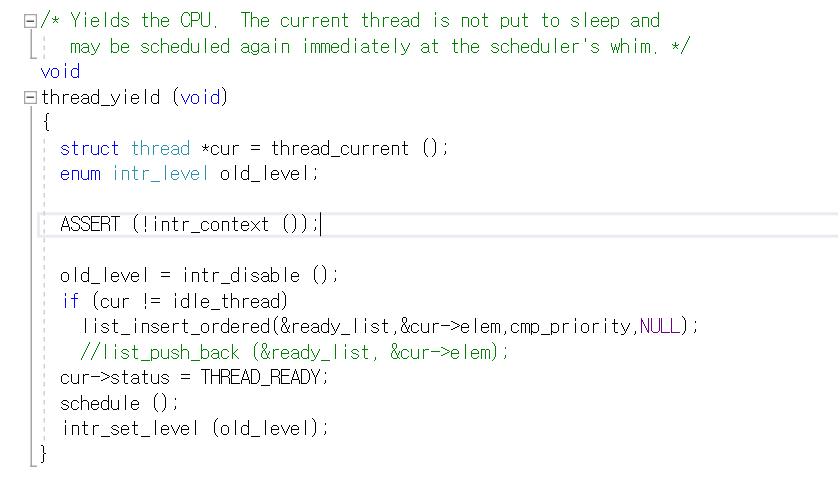
\* 함수 설명 : thread를 BLOCK상태에서 READY상태로 바꾸어주는, 즉 unblock해주는 함수이다.

\* 사용 변수

enum intr\_level old\_level : interupt on, off를 저장하는 변수

\* 내용 : 원래는 list\_push\_back()함수로 list 삽입을 했던 코드를 우선순위를 도입함으로써 우선순위에 따라서 insert하기 위해 list\_insert\_ordered() 함수로 바꿔주었다.

* + - * **void thread\_yield(void)**

****

\* 함수 설명 : 현재의 Thread를 ready\_list에 넣고 다시 schedule하는 함수이다.

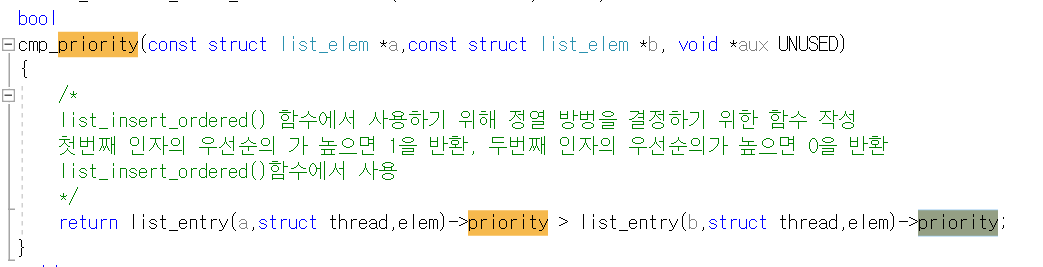
\* 사용 변수

struct thread \*cur : 현재의 thread를 포인팅하는 변수

enum intr\_level old\_level : interupt on, off를 저장하는 변수

\* 내용 : 원래는 list\_push\_back()함수로 list 삽입을 했던 코드를 우선순위를 도입함으로써 우선순위에 따라서 insert하기 위해 list\_insert\_ordered() 함수로 바꿔주었다.

**-\* bool cmp\_priority(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)**



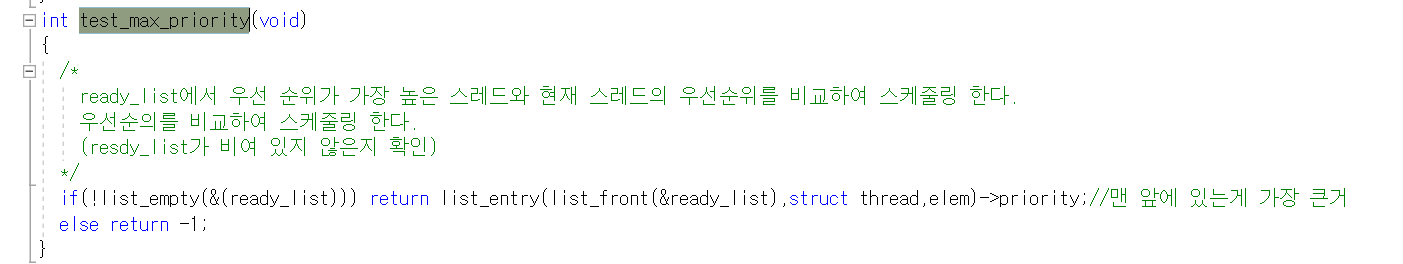
\* 함수 설명 : thread 2개를 parameter로 전달받아 a, b의 priority을 비교한다.

\* 내용 : list\_inserted\_ordered() 함수에서 사용하기 위해 정렬 방법을 결정하기 위한 함수이다.

첫번쨰 인자의 우선순위가 높으면 1을 반환, 두번째 인자의 우선순위가 높으면 0을 반환한다.

List\_inserted\_ordered() 함수에서 사용 가능하다.

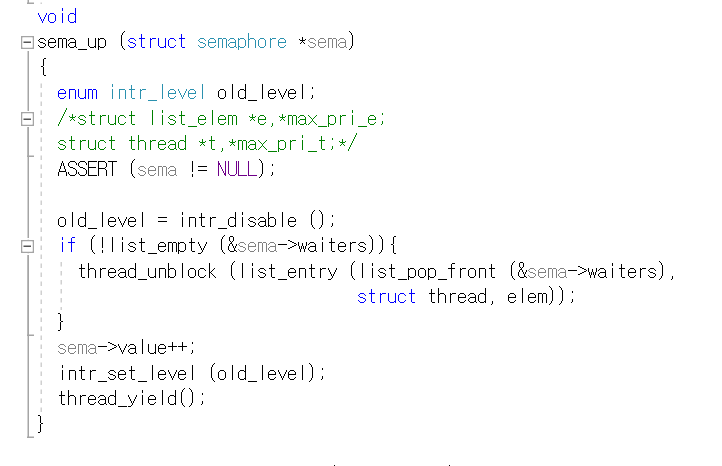
* + - * + **Int test\_max\_priority(void)**

****

\* 함수 설명 : ready\_list에서 우선 순위가 가장 높은 스레드와 현재 스레드의 우선순위를 비교하여 스케줄링한다. .

\* 내용 : ready\_list에서 우선 순위가 가장 높은 스레드와 현재 스레드의 우선순위를 비교하여 스케줄링한다. .그 후, 비교를 통해 맨 앞에 있는 가장 큰 값을 리턴한다. 만약, ready\_list가 비어 있으면 -1을 return 한다.

* + - * + **void sema\_up(struct semaphore \*sema)**

****

\* 함수 설명 : semaphore up을 구현한 함수이다. 기존 sema\_up 함수에서 맨 아래 thread\_yield()를 추가해주었다.

\* 사용 변수

enum intr\_level old\_level : interupt on, off를 저장하는 변수

\* 내용 :semaphore와 priority scheduling을 동시에 사용하기 위해서 기존 sema\_up에서 thread의 priority에 따라서 ready queue에 삽입하고 reschedule하기 위해 thread\_yield()를 추가하였다.

* + - * **void sema\_down(struct semaphore \*sema)**

****

\* 함수 설명 : semaphore down을 구현한 함수이다. 기존 sema\_down 함수에서 list\_push\_back 함수를 list\_insert\_ordered 함수로 변경하였다.

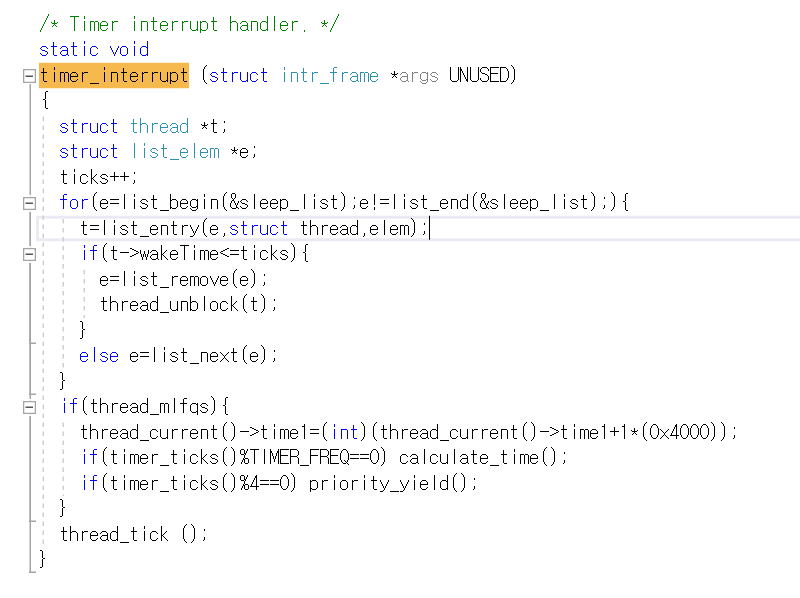
\* 사용 변수

enum intr\_level old\_level : interupt on, off를 저장하는 변수

\* 내용 : 현재 우선순위 스케쥴링을 적용하기 위하여 우선순위에 따라 semaphore wait list에 thread를 삽입해주기 위하여 기존 sema\_down 함수에서 list\_push\_back 함수를 list\_insert\_ordered 함수로 변경하였다.

**3.. BSD Scheduler**

* + - * **struct void timer\_interrrupt ( struct intr\_frame \*args UNUSED)**

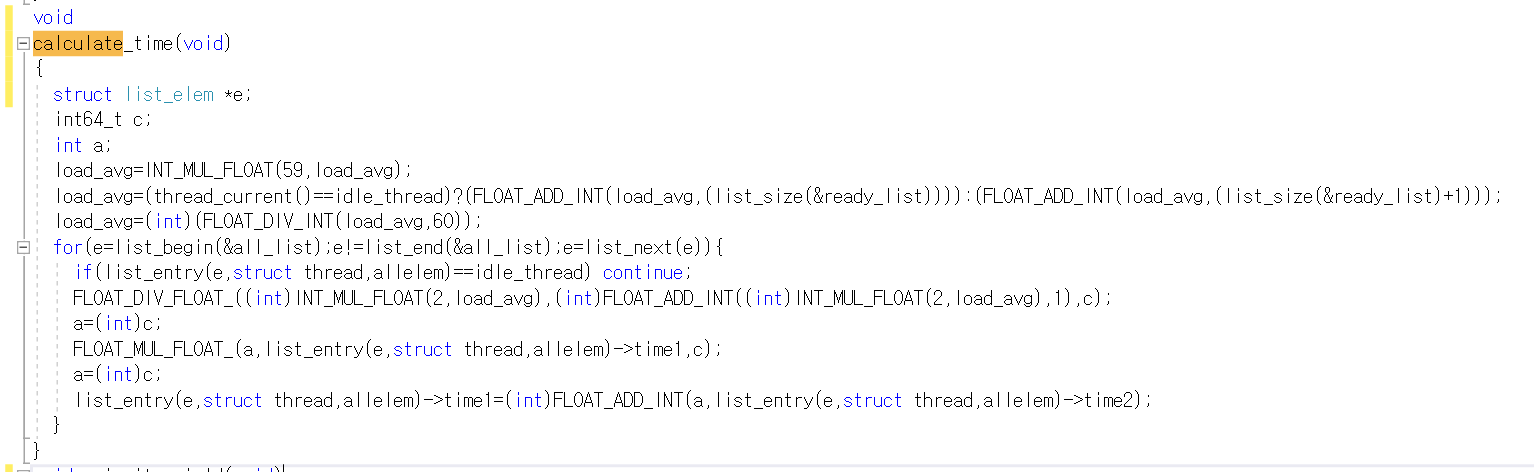
****

\* 함수 설명 : thread\_mlfqs가 true일때 BSD scheduler가 구현되도록 한다.

\* 사용 변수 : 위에서 설명함.

\* 내용 : BSD Scheduler에 대한 test case를 수행하기 위해 thread\_mlfqs가 true로 set 될 때 진행되는 code이다. 우선 현재 thread의 time1(recent\_cpu)에 1에 (0x4000) 값을 곱한 값을 더해준 값을 다시 time1에 넣어준다. 그 후에, Timer\_FREQ마다 calculate\_time()을 호출하고, 그 함수에서 모든 thread의 time1을 재계산한다. 그리고 4 마다, priority\_yeild를 호출해 모든 thread의 Priority를 계산한다.

* + - * + **void calculate\_time(void)**



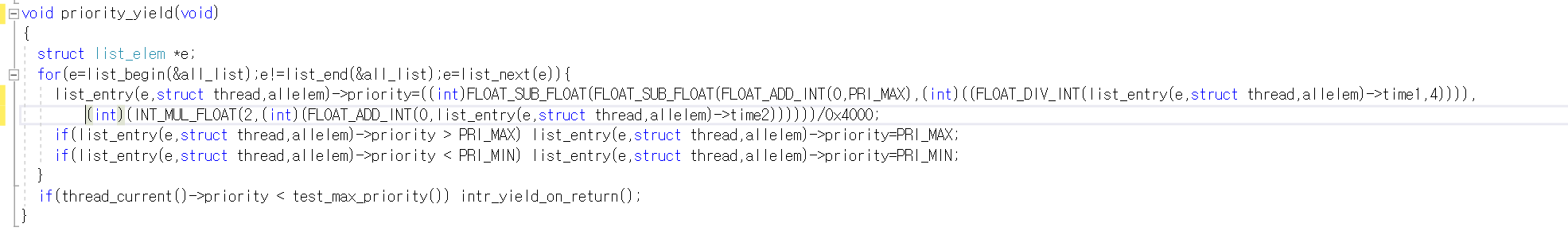
\* 함수 설명 : recent\_cpu를 구하는 공식인 recent\_cpu(time1) = (2\*load\_avg) / (2\*load\_avg+1) \* recent\_cpu(time1) + nice(time2)를 계산한다. 또한, load\_avg를 구하는 공식인 load\_avg = (59/60)\*load\_avg + (1/60)\*ready\_thread를 계산한다.

\* 사용 변수 : 없음

\* 내용 : 식이 길기 때문에 곱셈을 기준으로 3단계로 나눠서 진행하였다. 계산은 pintos manual을 참조하여 하였다. 제일 처음에 load\_avg을 구하는 공식인 59를 곱하는 것을 float.h에 선언된함수인 INT\_MUL\_FLOAT를 통해 곱해준다. 그 이유는, 실수 \* 정수 이기 때문이다. 그리고,

현재 thread가 idle thread라면, load\_avg와 ready\_list의 list\_size를 더해주고, idle thread가 아니라면, ready\_list의 list\_size 값에 1을 더해준 값을 더해준다. 그 후에 , 실수 / 정수 이므로, Float\_div\_Int 를 통해 load\_avg를 나눠준다. 그 후에, list\_elem e가 계속 다음 리스트로 넘어가는 반복문을 진행하여, 실수 / 실수이므로, Float\_div\_Float를 통해 Load\_avg 값에 2를 곱한 것을 load\_avg 값에 2를 곱한 값과 1을 더해준 값으로 나눠주고, 그 값을 c에 넣어준다. 그리고 a에 int 형 변환을 해준 c 값을 넣어주고, a와 현재 list allelem값의 time1(recent\_cpu)값과 곱해준 후, c에 다시 할당한다. 그리고 이 값을 다시 a에 넣어주고, 마지막으로, time1(recent\_cpu) 값에 a 와 현재 list의 allelem의 time2(nice) 값을 더해준다. 위의 2가지 공식을 합쳐서 계산해서, 상당히 복잡한 식이 나오게 되었다.

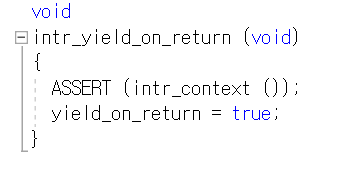
* + - * + **void priority\_yield(void)**

****

\* 함수 설명 : priority를 구하는 공식인 priority = PRI\_MAX - (recent\_cpu/4) - (nice\*2)를 계산한다.

\* 내용 : 위의 식인 recent\_cpu(time1) 값을 4로 나눠주고, PRI\_MAX에 그 값을 빼준다. 그 후에 time2(Nice) 값을 Ox4000으로 나눠준 값을 2와 곱한 값을 다시 한 번 빼주면서, 위의 공식을 충족한다. 0x4000으로 나눠주는 이유는, 실수를 integer로 convert하려면 2^14를 나눠줘야 하기에 2^14의 16진수 표현인 0x4000을 나눈다. 그 후에, 현재 list\_entry의 priority 값이 PRI\_MAX 보다 큰 경우에는 PRI\_MAX 값을 할당하고, 그 반대의 경우에는 PRI\_MIN 값을 할당한다. 그리고 만약, 현재 thread의 priority가 test\_max\_priority 값보다 작은 경우, intr\_yield\_on\_return 함수를 호출한다.

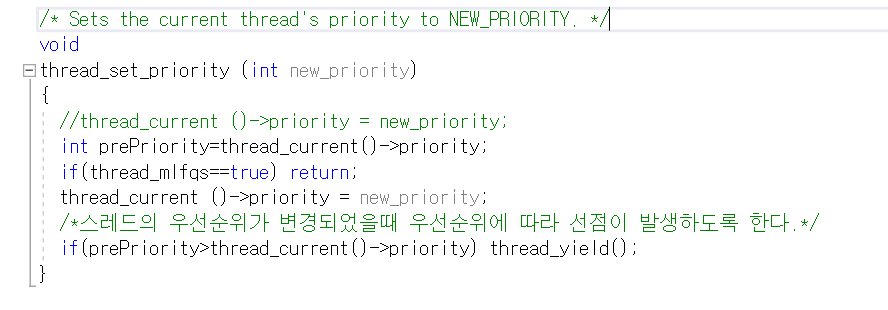
* + - * + \* **void intr\_yield\_on\_return(void)**



\*함수 : yield\_on\_return 변수를 true로 셋팅해준다. 이 함수는 interrupt.c에 만들어 두었다.

Yield\_on\_return 변수는 interrupt.c에 선언되어 있는 전역변수이다.

* + - * **void thread\_set\_priority(int new\_priority)**

****

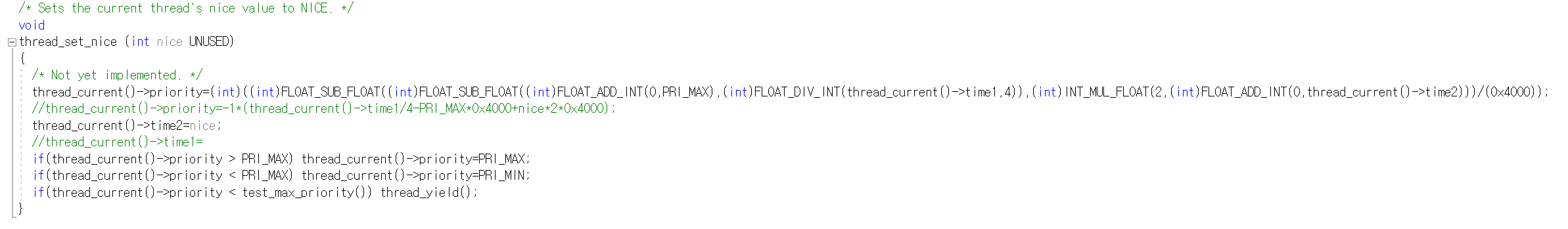
\* 함수 설명 : 현재 thread의 priority를 new\_priority로 set한다

\* 사용 변수 :

Int prePriority : 현재 thread의 priority를 저장하는 변수.

\* 내용 : prePriority에 현재 thread의 priority를 저장하고, parameter로 받은 new\_priority를 현재 thread의 priority로 prePriority와 새로 할당된 new\_priority를 비교해서 더 작을 경우 thread\_yield함수를 이용하여 reschedule한다.

* + - * **void thread\_set\_nice(int nice)**

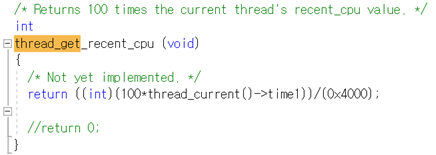
****

\* 함수 설명 : 현재 thread의 time2를 parameter로 받은 nice로 설정한다.

\* 사용 변수 : 없음.

\* 내용 : 현재 threa의 Priority 값을priority = PRI\_MAX - (recent\_cpu/4) - (nice\*2) 공식에 맞춰 계산하여 할당한다. 0x4000으로 나눠주는 이유는, 실수를 integer로 convert하려면 2^14를 나눠줘야 하기에 2^14의 16진수 표현인 0x4000을 나눈다. 그 후에, 현재 thread의 time2에 nice값을 할당한다. 그 후, 현재 thread의 priority 값이 PRI\_MAX 보다 크면, PRI\_MAX로 할당해주고, 반대라면 PRI\_MIN으로 할당한다. 또한, test\_max\_priority보다 작다면, thread\_yeild() 함수를 호출해 reschedule 한다.

* + - * **int thread\_get\_recent\_cpu(void)**

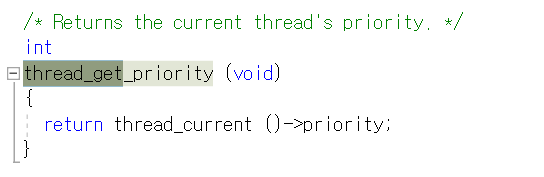
****

\* 함수 설명 : 현재 thread의 time1(recent\_cpu)값의 100을 곱한 값을 return한다.

\* 사용 변수 : 없음.

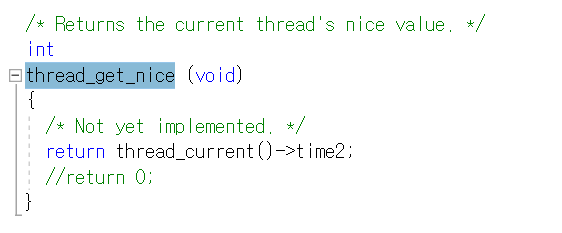
\* 내용 : recent\_cpu는 실수 값이므로 return시에 fixed-point 형식을 사용해야 한다. pintos manual에 따르면 실수\*정수는 그냥 그대로 계산하기 떄문에 100\*thread\_current()->time1(recent\_cpu)를 하고 실수를 integer로 convert하려면 2의 14승을 나눠야 하므로 2^14의 16진수 표현인 0x4000을 나눠서 return한다.

* + - * **int thread\_get\_priority(void)**



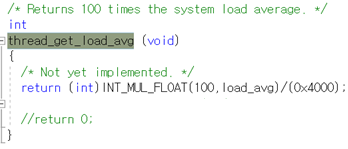
\* 함수 설명 : 현재 thread의 priority를 return한다.

* + - * **int thread\_get\_nice(void)**

****

\* 함수 설명 : 현재 thread의 time2(nice)를 return한다.

* + - * **int thread\_get\_load\_avg(void)**

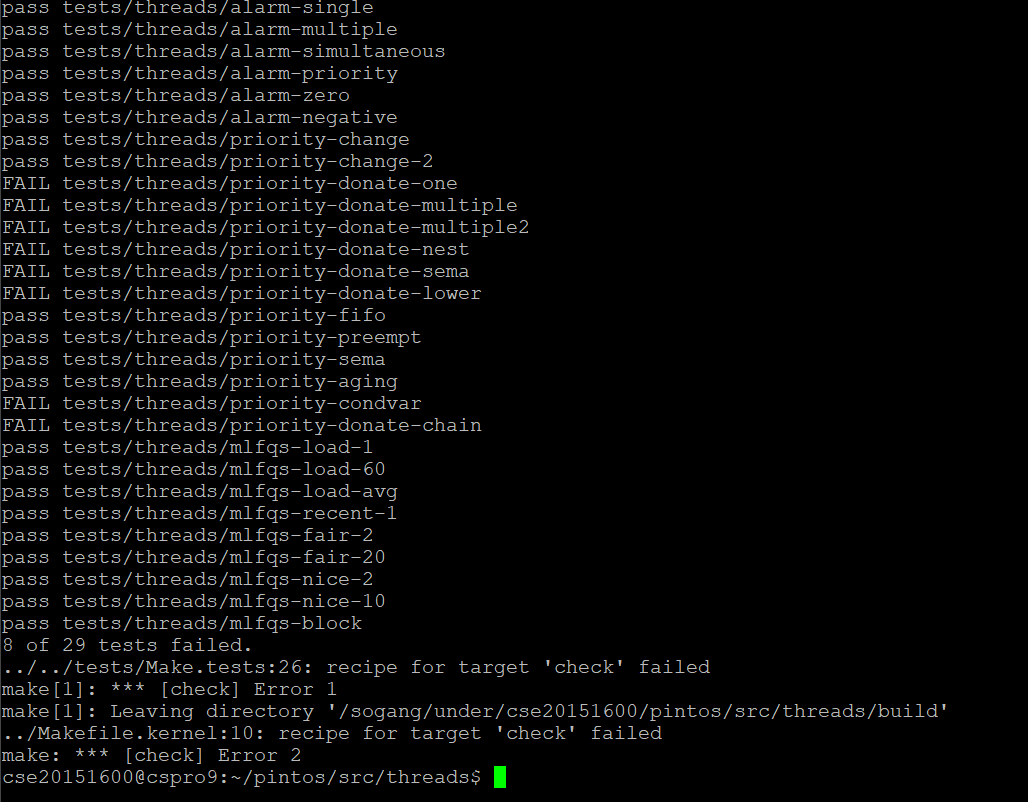
****

\* 함수 설명 : load\_avg값의 100을 곱한 값을 return한다.

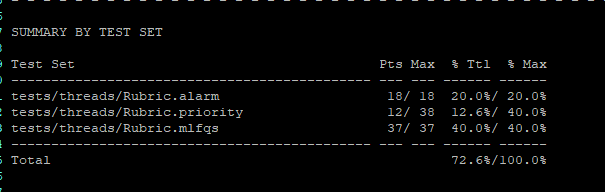
\* 내용 : load\_avg는 실수 값이므로 return시에 fixed-point 형식을 사용해야 한다. 실수 \* 정수이기 떄문에, float.h에 구현해놓은 INT-MUL\_FLOAT 함수를 사용하여 100과 load\_avg값을 곱한다. 그 후에,0x4000으로 나눠준다. 실수를 integer로 convert하려면 2의 14승을 나눠야 하므로 2^14의 16진수 표현인 0x4000으로 나눈다. 그 후 그 값을 return 해준다.

* 1. **시험 및 평가 내용**

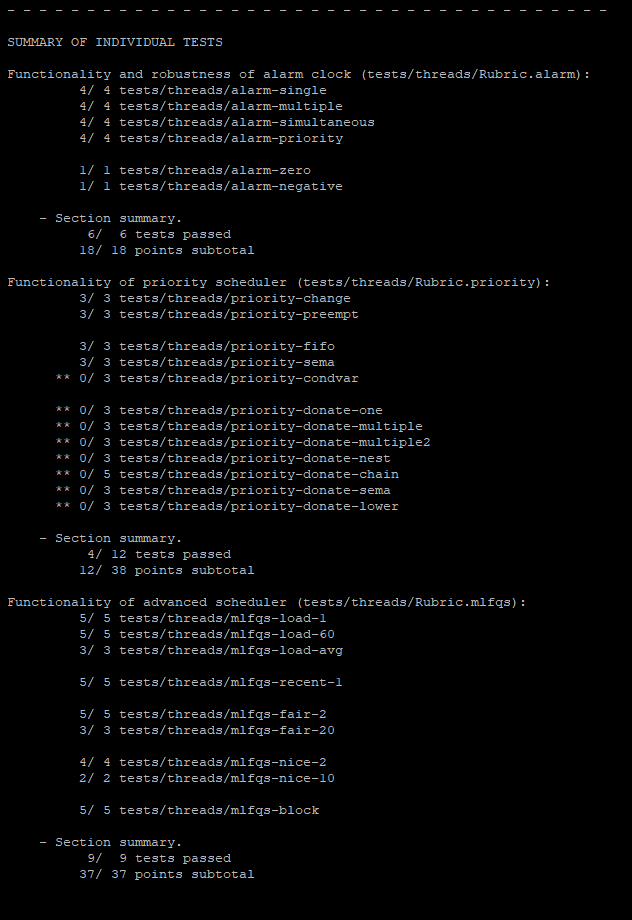
**< make check >**

****

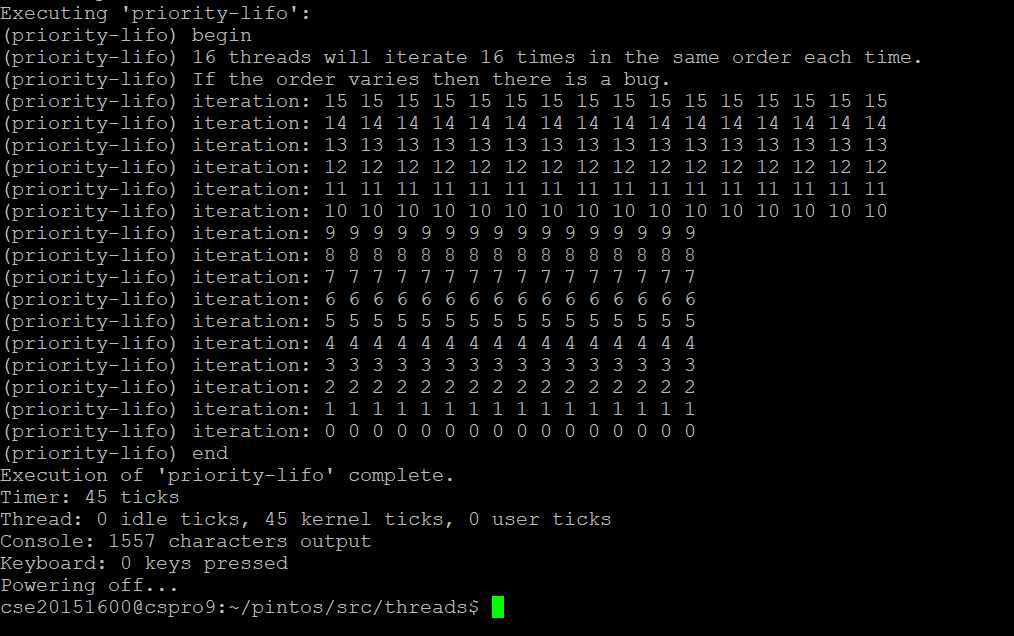
**< make grade -1 >**



**< make grade – 2>**

****

**<priority-lifo 결과>**

****

test/priority-lifo.c 안에서, THREAD\_CNT를 16으로 설정하고 16개의 thread의 우선순위를 차등적으로 부여하여 갈수록 우선순위를 증가시키며 thread를 생성하였다. 우선순위는 PRI\_DEFAULT + 1 + i로 설정하는데, i는 0부터 THREAD\_CNT만큼 증가하는 변수이다. 이 경우, thread 마다 우선순위가 다르므로 ready list에 삽입될 때 우선순위에 맞게 삽입될 것이다. 따라서 priority가 PRI\_DEFAULT+16인 thread부터 PRI\_DEFAULT + 1까지 우선순위 기준으로 thread가 실행될 것이다.

1. **기타**
   1. **연구 조원 기여도**

* **장진영 : 50 %, 장승민 : 50 %**
  1. **소감**

장진영: Thread project의 목표는 Thread 관련 기능인 Alarm clock, Scheduler를 구현하는 것이었다. 우리는 semaphore와 busy waiting에 관하여 project 2에서 다루었었고, 따라서 busy waiting을 해결하기 위하여 Alarm Clock을 구현하는 것은 큰 어려움이 없었다. 수정해야 하는 부분 또한 명세서에 상세하게 작성되어 있어 명세서를 따라서 그대로 코딩하니 실행되는 것을 보았고 성취감이 있었다. 역시 먼저 한번 경험을 해본 후에 같은 내용을 활용하여 코딩하니 훨신 원활하게 진행된다는 것을 깨달았다.

장승민 : 이번 프로젝트에서는 무작정 기다리는 방법인 busy waiting이 cpu\_time을 낭비하게 되므로 이를 개선하기 위한 방법인 alarm clock과 priority schedule & aging에 대해서 자세히 알 수 있었다. 그리고 multi level feedback queue schedule을 진행하기 위한 BSD scheduler에 대해서도 알 수 있었다. 그리고 pintos안에서는 실수 연산이 불가능 해서 fixed-point를 사용하여 계산해야 한다는 사실도 알게 되었다.