**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

학번 / 이름 : 20171637 박성우

개발 기간 :10.26~10.31

1. **개발 목표**

Busy waiting 방식으로 구현되어 비효율적인 Alarm clock(ready와 running 상태를 반복)을 non-busy waiting 방식(blocked state로 변경 후 시간이 다 되면 ready queue에 다시 추가)으로 변경하여 효율적인 방식으로 적용한다.

Priority를 고려하지 않는 round-robin 스케줄링 방식에서 priority를 고려하는 스케줄링 방식으로 변경하고, starvation 문제를 해결하기 위해 aging 기법을 추가한다.

Multi-Level Feedback Queue를 구현하고, nice, recent\_cpu, priority값을 알맞게 계산한다. 또한, floating-point arithmethic을 지원하지 않는 pintos를 위해 적절하게 floating-point 연산을 할 수 있도록 한다.

**개발 범위 및 내용**

* 1. **개발 범위**

1) Alarm Clock

타이머가 끝나지 않아도 thread\_yield 함수를 호출해서 thread가 RUNNING, READY 상태를 왔다 갔다 해야 하는 비효율적인 기존의 방식에서, 타이머가 끝나기 전에는 해당 thread를 BLOCKED 상태로 바꿔 깨어나야 할 때 UNBLOCK 하는 효율적인 방식으로 바꾼다.

2) Priority Scheduling

각각의 프로세스(쓰레드)들에 우선순위를 부여하여 우선순위가 높은 순서대로 실행 될 수 있도록 스케줄링 해준다. 여기서 starvation의 발생을 예방하기 위해 aging을 이용하여 구현해준다.

3) Advanced Scheduler (추가구현을 한 경우)

FIFO방식으로 스케줄링을 하던 기존의 Pintos를 수정해야한다. 각각의 쓰레드의 다른 스케줄링 요구를 균형되게 스케줄링 할 수 있도록 구현해준다. mlfqs로 구현한다. 이는 우선순위 별로 ready queue가 존재하고, 가장 높은 우선순위의 queue부터 쓰레드가 선택되게 하는 방식이다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

thread의 관리를 위해 thread들을 큐에 저장한다. 이때 wake up 시간을 함께 저장해준다. 시간을 체크할 때 설정했던 wake up 시간이 되지 않은 경우 thread를 blocked 상태로 만든다. timer\_interrupt()를 사용해서 매 tick마다 설정한 wake up 시간이 되었는지 확인하고 설정한 시간이 되었을 때, 이를 ready상태로 만들고 ready queue에 삽입하여 스레드를 깨우게 된다.

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

Running thread 를 잡아서 끌어내려버리고 높은 priority를 가진 thread 를 schedule 한다.

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

BSD scheduler에서는 nice 값이 사용된다. 각각의 thread는 -20~20의 nice값을 갖는데, 0인 경우 아무 영향을 미치지 않으며, 양수인 경우 다른 thread들이 CPU를 사용할 수 있도록 우선순위를 낮춰준다.

BSD스케줄러를 구현하는 데에는 실수 연산이 필요한데 기존의 pintos에서는 실수 연산이 되지 않는다. 따라서 실수 값인 recent\_cpu와 load\_avg를 계산하기 위해 fixed-point형식을 사용해 계산한다.

Recent\_cpu = ( 2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu + nice

이는 CPU시간을 추정하는데, 더 최근의 CPU time이 덜 최근의 CPU time보다 높은 가중치가 부여되어야 한다.

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

Ready상태의 thread수의 평균이다.

최종적으로 priority를 계산하는 방식이다.

priority = PRI\_MAX – (recent\_cpu / 4) - (nice \* 2)

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
* 2021.10.26. ~ 2021.10.30. : Alarm clock, Priority scheduling, BSD scheduler 구현
* 2021.10.31 ~ 2021.10.31 : 보고서 작성
  1. **개발 방법**

Alarm clock

Block 상태의 스레드들을 관리하기 위한 sleep list를 thread.h 에 struct list sleep\_list 구조체를 추가하고 thread.c 에서 초기화를 해주었다. Thread 구조체에 tiqui\_taka 라는 스레드가 자다가 일어나야 하는 시간을 저장하는 변수를 추가 해주었다.

src/devices/timer.c 에 있는 timer\_sleep 함수와 timer\_interrupt. 함수를 수정한다. Timer\_sleep 에서 시간이 될 때까지 무지성으로 thread\_yield 함수만 주구장창 호출하던 것을 tiqui\_taka 라는 변수에 일어나야 하는 시간을 저장하고 sleep\_list 에 스레드를 넣고 그 스레드를 block 상태로 바꾸어준다. Timer\_interrupt 에서 1틱마다 sleep\_list 를 돌면서 일어나야 할 시간이 된 스레드들을 unblock 해준다.

Priority Scheduling

Thread\_create 에서 생성한 스레드가 실행중인 스레드보다 priority가 높으면 yield 해준다.

기존 thread\_yield 함수에서는 thread\_yield 함수가 호출되면 현재 스레드를 ready\_list 맨뒤로 보내버렸지만 이제 ready\_list에 스레드들의 priority 순서대로 저장하고 맨 뒤로 보내는 게 아닌 priority 를 비교해 나가다 적절한 순서에 저장하게 된다. 따라서 ready\_list 는 priority 순으로 정렬된다.! Thread\_unblock 함수도 똑같이 고친다.

기존에 뼈대만 있던 Thread\_set\_priority 함수와 thread\_get\_priority 함수를 채워준다. get\_priority 함수는 현재 스레드의 priority를 반환하고 set\_priority 함수는 새로운 priority 를 설정하고 만약 현재 스레드보다 priority 가 작으면 thread\_yield 한다.

Starvation 을 막기위한 thread\_prior\_aging 전역변수와 thread\_aging() 함수를 추가한다. 명령어에 -aging 이 들어오면 thread\_prior\_aging 을 true 로 바꾸고 thread\_aging 함수를 실행하게 된다.

Semaphore 로 block 된 스레드들이 sema\_up 될 때 priority를 고려해서 제일 높은 priority를 가진 스레드부터 unblock 되고 가장 priority 가 높은 스레드부터 실행되어야 한다. 따라서 sema\_up 할 때 sema\_waiters 리스트를 정렬한 후 맨 앞, 즉 priority 가 가장 높은 스레드를 unblock 한다. 그리고 현재 실행되는 priority 보다 높을 수 있으니 thread\_yield 함수를 통해 priority 가 제일 높은 스레드가 실행되게 해주어야 하는데 여기서 가장 중요한 점이 sema\_up에서 thread\_yield 함수로 다시 schedule 을 하게 되면 page가 만들어지기도 전에 schedule 을 먼저 하게 되는 경우가 발생한다! 그래서 userprog test case 들이 다 터지게 된다! 이를 방지하기 위해 sema\_up 만을 위한 sema\_\_thread\_yield() 함수를 만들고 thread\_started 전역변수를 만들고 thread\_start() 함수가 실행되자마자 true 로 세팅한다.

BSD Scheduler

Mlfq scheduler 를 위해 필요한 변수들을 계산하고 반환하기 위해 load\_avg 전역변수와

Thread 구조체에 변수 nice, 와 recent\_cpu 를 추가해 주고 thread\_init 함수에서 변수들을 초기화 해준다..

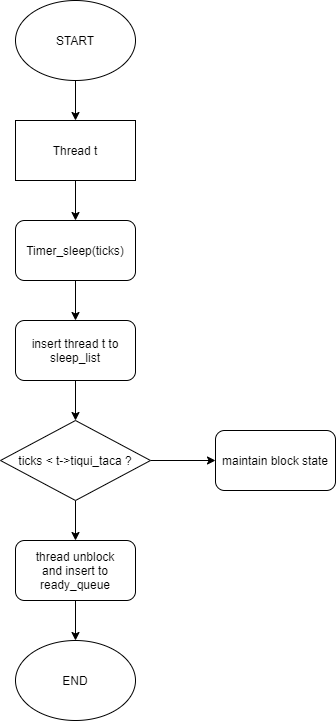
thread\_get\_nice/set\_nice, thread\_get\_recent\_cpu/load\_avg, thread\_increase\_recent\_cpu, recal\_load\_avg\_and\_recent\_cpu, recal\_priority 함수를 만든다. Src/device 의 timer\_interrupt 에서 -mlfqs 명령어가 붙었을 때 적절한 시간마다 위의 함수들을 호출해서 각 변수들을 변경해준다.

주의할 점은 pintos 는 실수계산을 지원해주지 않기에 pintos manual 에 따른 실수 계산하는 법을 이용해서 계산을 해야한다. 이를 위해 #define F (1<<14) 를 정의해두고 이 F를 이용한다.

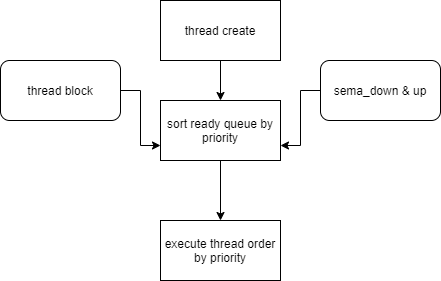
1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)

Alarm clock



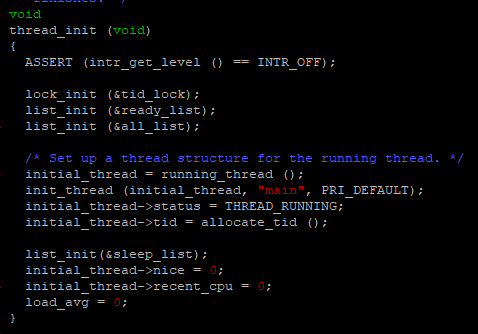
Priority scheduling



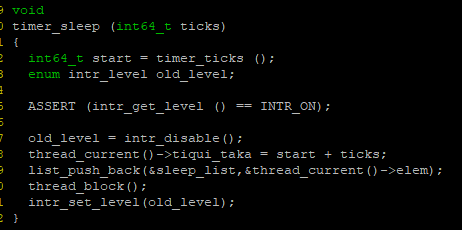
* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

**Alarm Clock**

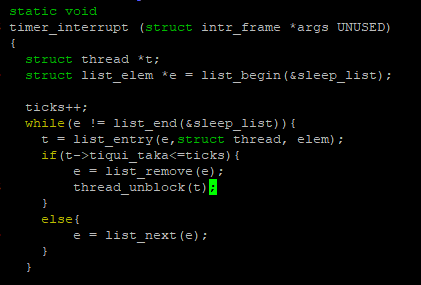
Block list 를 관리하기 위한 sleep list 를 thread.h 에 만들고 thread.c 의 thread\_init 에서 초기화를 해주었다.



그 후 timer\_sleep 함수에서 스레드가 일어나야 하는 시간을 tiqui\_taka 변수에 저장하고 sleep list에 넣어준 후 block 해준다.



Timer\_interrupt 함수에서 sleep list 를 돌면서 일어날 시간이 된 스레드 들을 unblock 하고 sleep list에서 지워준다.



**Priority Scheduling**

Thread\_create 에서 생성한 스레드가 실행중인 스레드보다 priority가 높으면 yield 해준다.

tid\_t

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

{ ……

if(priority > thread\_get\_priority())

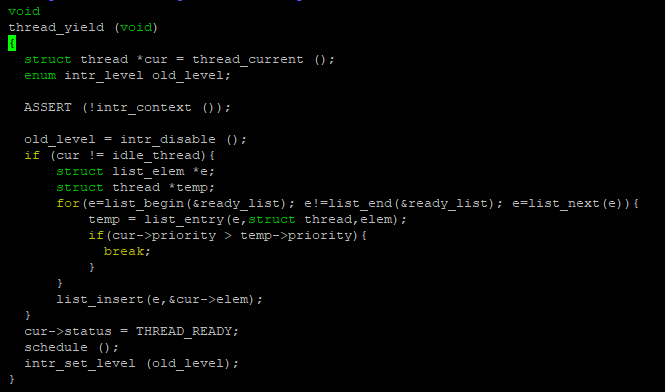
thread\_yield()

return tid;;

}

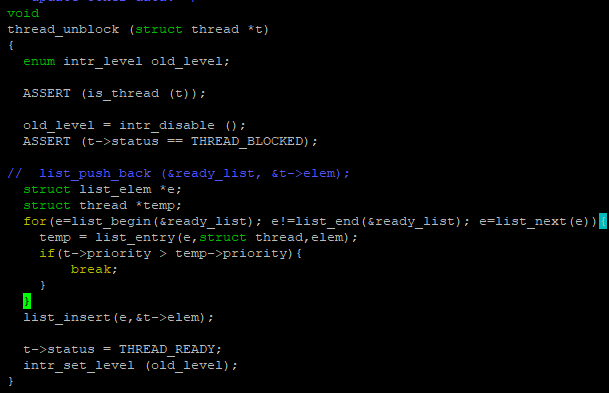
Thread\_yield()

실행중인 스레드를 ready list에 우선순위 순서에 맞게 ready list 에 삽입한다.



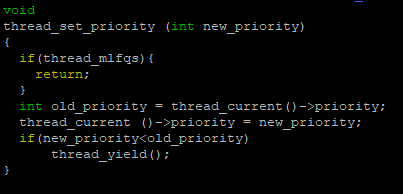
Thread\_unblock()

Thread\_yield 와 마찬가지로 ready list 를 정렬해서 넣는다



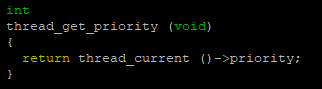
Thread\_set\_priority()

Priority 를 새로 설정하고 만약 새로운 priority 가 수행 중이던 thread의 기존 priority 보다 작으면 thread\_yield()를 호출해준다.



Thread\_get\_priority()

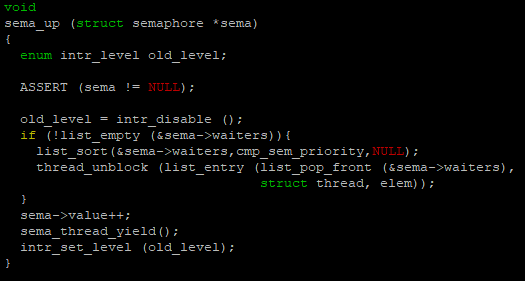
Priority 내놔



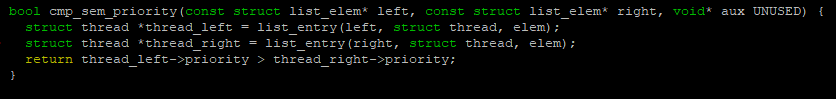
Sema\_up()

Sema\_waiters list를 우선 priority에 맞게 정렬을 해준다음에 unblock 해준다.

그리고 특별한 sema\_thread\_yield() 함수를 호출하여 우선순위가 높으면 먼저 실행시켜 준다.



여기서 sem\_priority는 우선순위를 비교해주는 함수이다.



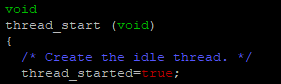
Sema\_thread\_yield()

기본적으로 thread\_yield 함수랑 똑같으나 이 sema\_thread\_yield 함수는 thread\_started 라는 변수가 참이고 intr\_context() 변수가 거짓일 때만 실행된다. Sema\_up 함수에서 thread\_yield 를 사용하게 되면 pagedir.c 가 실행이 되기도 전에 schedule 을 하게 되는 상황이 발생할 수 있어서 userprog 의 test case 80개가 다 터진다!

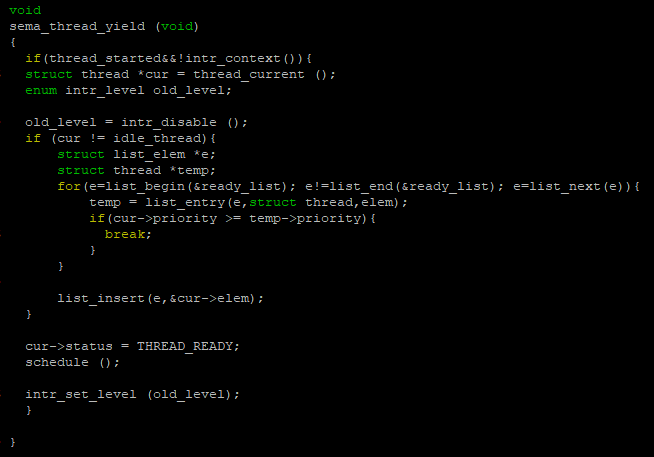
이런 일을 막기 위해서 thread\_started 전역변수를 만들고



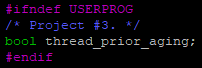
Thread\_start() 함수 즉 thread 가 시작될 때 true 로 설정한다.

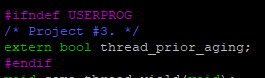


그리고 이 thread\_started 가 참이고 외부 interrupt 가 아닐 때 thread\_yield 기능을 수행하는 sema\_thread\_yield 를 실행한다.



Test case priority-aging 을 위해 thread.c 와 thread.h 에 thread\_prior\_aging 변수를 선언한다.

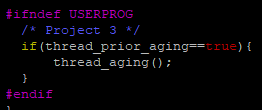
thread.c

thread.h

그리고 threads/init.c 에서 aging 명령어가 들어올 경우 true 로 세팅해준다.



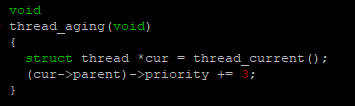
Thread.c 의 thread\_tick 함수에 thread\_prior\_aging 변수가 참이면 thread\_aging() 함수를 실행시켜서 aging test case를 처리해준다.



Thread\_aging()

Priority\_aging test case 를 보면 main thread 의 priority 를 2 줄이고 새로운 thread 를 만들어서 그 스레드에서 무한루프를 도는데 시간이 지나면 aging 이 되어 main 스레드가 실행이 되게 해야한다. 하지만 project3 ppt 자료에는 BSD Scheduler 의 priority recalculate 설명하기 전에 aging 이 먼저 나와있고 설명도 시간에 따라 잘 알아서 구현하면 된다고 하고 넘어갔기 때문에

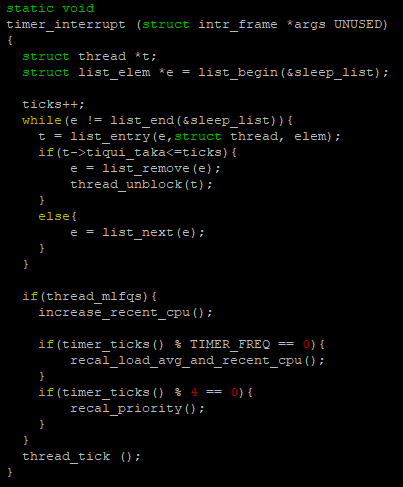
Project 3의 추가구현 없이도 이 priority-aging test case 가 통과되어야 한다고 판단하여 무한루프 도는 thread 의 부모 thread 인 main thread 의 우선순위를 3 높여주어 이를 해결했다.



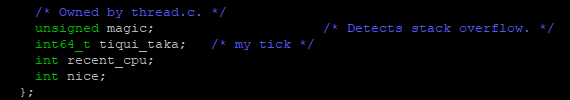
**BSD Scheduler**

Timer\_interrupt()

-mlfqs 명령어가 들어오면 BSD scheduler 의 규칙에 따라 priority를 계산해주어야 한다. 따라서 timer\_interrupt 에서 thread\_mlfqs 변수가 참이면 적절한 시간마다 priority를 계산하는 데에 있어 필요한 변수들을 재설정 해주는 함수를 호출한다.

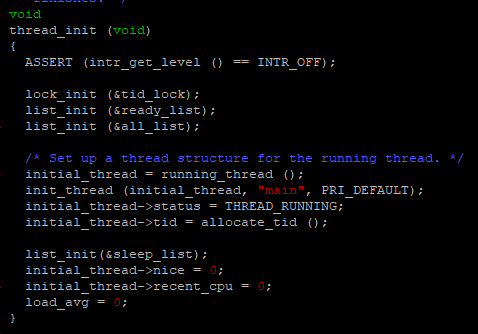


Priority 를 계산하는데 있어서 필요한 변수들은 thread.h 의 thread 구조체에 추가해 주거나 thread.c 의 전역변수로 추가를 해주었다.

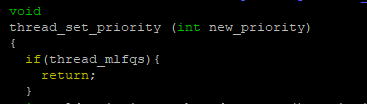
thread.h thread 구조체

thread.c 전역변수

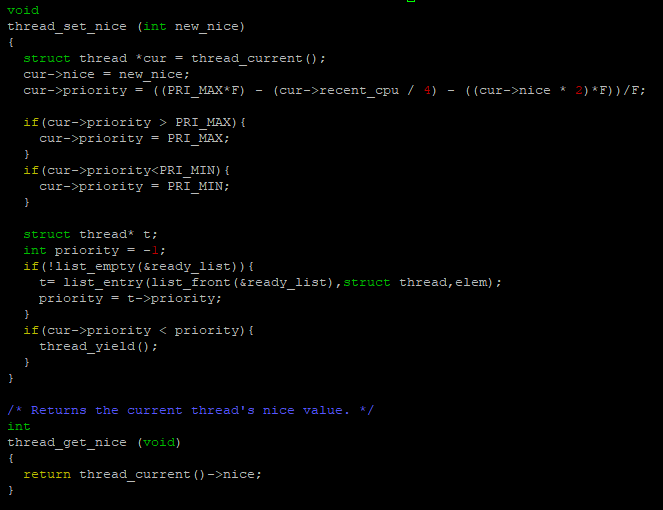
그리고 이들을 초기화 해준다.

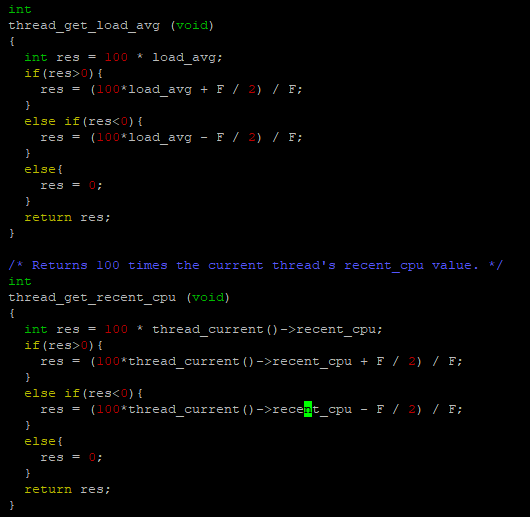


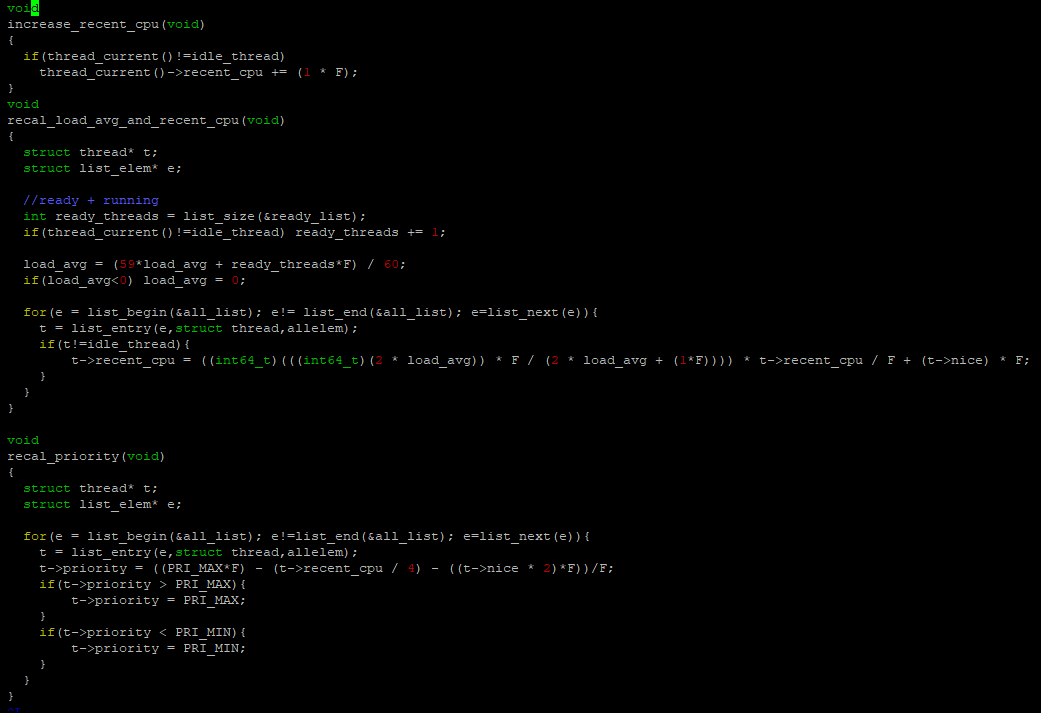
-mlfqs 명령어가 들어오면 각 스레드들은 자기 맘대로 priority를 바꿀 수 없기 때문에 thread\_set\_priority 함수를 수정해준다. Thread\_mlfqs 변수가 참이면 아무것도 하지 않게 만든다.



그 후 각 변수를 시간에 따라 알맞게 변경시키는 함수들을 만들어 주면 된다.

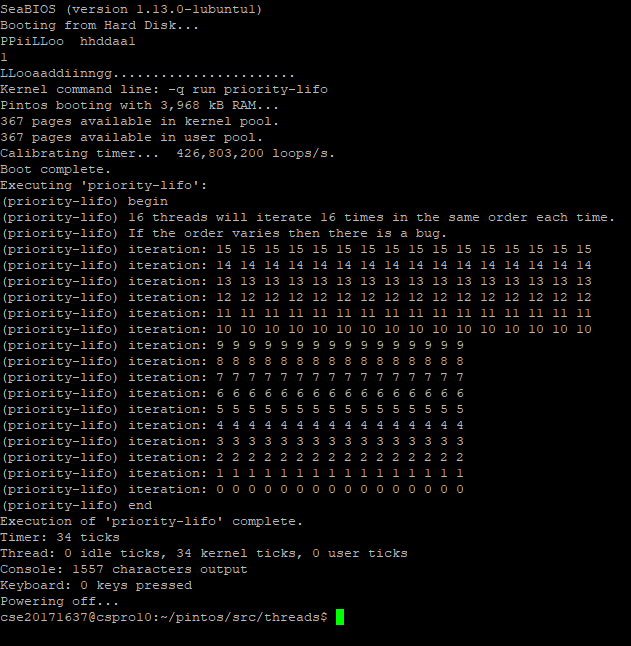






Project 3 ppt 와 pintos manual 에 따라 함수를 만들고 또한 pintos 는 실수 연산을 지원하지 않기에 #define F (1<<14) 를 통해 F 를 만들고 이 F를 이용해 실수 연산을 하였다.

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



올바른 결과를 뿜뿜하고 있다.

* make check 수행 결과를 캡처하여 첨부
* 