**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

조원 : 정성엽

개발 기간 : 11.23 ~ 12.8

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

이번 프로젝트에서는 pintos 상에서 비효율적인 Busy waiting으로 구현된 Alarm Clock의 구현을 Sleep, Wake up 방법을 사용해 효율적으로 만드는 것과 FIFO 구조로 Scheduling 되는 Pintos의 Schedule 방법을 Priority를 적용해 Scheduling 할 수 있도록 만드는 것이 목표입니다. 또한 추가구현으로 MLFQ를 사용하며 더 많은 인자를 고려하여 priority로 표현 가능한 BSD Scheduler를 구현하는 것이 목표입니다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock
  2. Priority Scheduling
  3. Advanced Scheduler (추가구현을 한 경우)

1) Alarm Clock

: 기존 Pintos 상의 Alarm 기능은 Busy Waiting 방법을 사용해 구현되어 있습니다. 하지만 Busy Waiting 방법은 CPU 자원을 많이 사용하는 비효율적인 방법입니다. 때문에 Alarm Clock 기능의 구현을 Sleep, Wake Up 방법을 사용하여 더 효율적으로 만들려 합니다.

2) Priority Scheduling

: 기존 Pintos 상의 Scheduling 방법은 FIFO 방법을 사용하여, 먼저 Ready Queue에 들어온 쓰레드를 먼저 실행시킵니다. 하지만 이 방법은 각 쓰레드의 우선순위를 반영할 수 없습니다. 때문에 우선순위에 따라 우선순위가 높은 쓰레드를 먼저 실행 가능한 Scheduling 기능을 만들고자 합니다. 또한 Priority가 낮은 쓰레드가 영영 실행되지 않는 상황을 방지하고자 Aging또한 구현하려 합니다. 이러한 구현을 통해 기존의 단순한 Scheduling보다 더 복잡하고 많은 기능을 수행가능한 즉, 우선순위의 반영이 가능한 Scheduler의 구현이 가능할 것으로 기대합니다.

3) Advanced Scheduler

: 위의 2)에 해당하는 Priority Scheduling을 구현하면 해당 Scheduler는 각 쓰레드마다 설정되는 Priority만을 기반으로 Scheduling 합니다. 하지만 실제 Scheduling 과정에는 Priority뿐만 아니라, CPU 점유율, 다른 thread와의 상관관계 (Niceness) 등 다양한 요소를 고려할 수 있어야 합니다. 이를 위해 구현한 것이 Advanced(BSD) Scheduler입니다. 구현을 통해 위에 언급한 다양한 요소를 고려한 더 좋은 Scheduling이 가능할 것으로 기대합니다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.
3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

1) Blocked 상태의 Thread를 어떻게 깨울 수 있는지 서술.

: Block 상태의 Thread가 깨어나야 할 시간이 된다면, Wake up을 통해 다시 Ready Queue에 넣어주어야 합니다. 각 Thread마다 언제 깨어날지를 저장하는 변수를 추가하고, Sleep 상태인 thread를 저장할 list를 새로 만든다. 특정 함수를 통해 주기적으로 Sleep상태인 Thread들이 저장된 list를 순회하면서 thread의 wakeup\_time이 현재까지 진행된 tick(time)보다 작은지 확인하고, 작다면, 해당 thread는 이제 깨어나야 하는 thread라는 의미임으로, Sleep List에서 삭제, thread\_unblock() 함수를 통해 thread의 state를 ready로 변경, 그리고 Ready Queue에 추가해준다. 이러한 역할을 하기에 적절한 함수는 timer\_interrupt() 함수인데, 이 함수는 주기적으로(매 tick마다) 자동으로 호출되고, 따라서 주기적으로 위의 검사를 수행하기에 적합하다. 해당 함수내부에 위의 검사방법을 구현하여 Block 상태의 Thread를 주기적으로 깨울 수 있다.

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

: Priority Scheduling에 따르면, Running Thread는 Ready List 중 가장 Priority가 높은 Thread일 것이다. 따라서 현재 Running Thread보다 높은 Priority를 가진 Thread가 Ready Queue에 들어온다면 Scheduling 정책에 따라 Ready Queue의 가장 앞에 위치하게 될 것이다. 이후 thread\_yield()를 통해 현재 Running 중인 Thread의 수행을 멈추고 Ready Queue의 가장 앞에 있는(가장 Priority가 높은) Thread를 수행한다면, 가장 높은 Priority인 Thread를 Running Thread로 만들 수 있다. 이때 이전에 수행중인 Thread의 수행이 아직 끝나지 않았다면 다시 Ready Queue에 적절한 위치에 삽입될 것이다.

3) Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.

: Advanced Scheduler에서 Priority는 CPU 점유율, 다른 thread와의 상관관계 (Niceness) 등 다양한 요소를 포함하여 계산됩니다. 명세서에 제시된 계산식은 아래와 같습니다.

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

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

load\_avg = (59/60) \* load\_avg + (1/60) \* number\_of\_ready\_threads

따라서 매번 각 Thread의 Priority를 위와 같은 계산을 통해 산출해야 하는데, Advanced Scheduler상에서는 Float point arithmetic을 지원하지 않는다 대신 Fixed-Point real arithmetic을 사용해야한다. 때문에 대신 Fixed-Point real을 32bit int data type로 저장하지만 실제 연산 방법은 달라진다. 이러한 연산을 위해 int와 Fixed-Point real간의 형 변환 함수, Fixed-Point real의 곱셈 나눗셈 함수가 필요하다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

1) 11.27 ~ 11.28: 프로젝트 분석, 코드 분석 및 설계 구상

2) 11.29: Alarm clock 구현

3) 12.04: 디버깅 및 Priority Scheduler 구현

4) 12.05: 디버깅 및 Advanced Scheduler 구현

5) 12.06: 코드 검증 및 디버깅

6) 12.07 ~ 12.08 : 보고서 작성

* 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) **Alarm Clock** (Blocked 상태의 Thread를 어떻게 깨울 수 있는지 서술.)

: 위와 같은 문제는 사실상 Alarm Clock의 구현과 직결된다. 기존의 Pintos 구현은 timer\_sleep() 함수를 통해 thread를 Sleep 상태로 만든다면 이를 Busy Waiting으로 기다린다. 하지만 이번 구현에서 이러한 비효율적인 방법을 Sleep, Wake Up 방법을 사용해 구현한다. Block 상태의 Thread가 깨어나야 할 시간이 된다면, Wake up을 통해 다시 Ready Queue에 넣어주어야 한다. 각 Thread마다 언제 깨어날지를 저장하는 변수를 추가하여야 함으로 src/threads/thread.h 코드의 thread 자료구조에 해당 변수를 추가해야 한다. Sleep 상태인 thread를 저장할 list를 새로 만들어야 함으로, src/devices/timer.c 코드에 자료구조를 새롭게 선언한다. 특정 함수를 통해 주기적으로 Sleep상태인 Thread들이 저장된 list를 순회하면서 thread의 wakeup\_time이 현재까지 진행된 tick(time)보다 작은지 확인하고, 작다면, 해당 thread는 이제 깨어나야 하는 thread라는 의미임으로, Sleep List에서 삭제, thread\_unblock() 함수를 통해 thread의 state를 ready로 변경, 그리고 Ready Queue에 추가해준다. 이러한 역할을 하기에 적절한 함수는 timer\_interrupt() 함수인데, 이 함수는 주기적으로(매 tick마다) 자동으로 호출되고, 따라서 주기적으로 위의 검사를 수행하기에 적합하다. 해당 함수내부에 위의 검사방법을 구현하여 Block 상태의 Thread를 주기적으로 깨울 수 있다. 따라서 src/devices/timer.c 코드상의 timer\_interrupt() 함수를 수정한다. 그리고 각 Thread의 Sleep 방법을 변경해야 함으로, src/devices/timer.c 코드의 timer\_sleep() 함수를 수정해야 한다.

2) **Priority Scheduling** (Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.)

: 새로운 Thread가 Ready Queue에 들어온다면 해당 Thread의 Priority에 따라 적합한 위치의 ready\_list상에 저장해야 한다. 이러한 저장은 기존에 list에 구현되어있는 list\_insert\_ordered() 함수를 사용해 구현되는데, order의 기준인 Priority를 기준으로 Compare를 수행할 Compare 함수를 src/thread/thread.c 코드상에 새롭게 구현해야 한다. 위에서 언급한 것처럼 Priority에 따라 현재 수행중인 Thread를 조정해야 함으로, src/thread/thread.c 코드상에 아래와 같은 각 기능을 수행하는 함수를 구현 또는 수정해야 한다. 먼저 thread\_set\_priority() 함수의 수정을 통해 새롭게 ready\_list에 들어온 thread의 Priority가 현재 수행중인 thread보다 더 높은지 검사를 하고, 높다면 thread\_yield()를 호출하여 Running thread의 변경을 수행해야 한다. 이때 호출되는 thread\_yield() 함수 또한 적절한 동작을 하도록 수정되어야 하는데, 현재 수행중인 thread를 적절한 priority에 따라 다시 ready\_list에 삽입해야 한다. 또한 Thread가 unblock될 때 Priority에 따라 ready\_list에 삽입되도록 thread\_unblock() 함수를 수정해야 한다. 추가적으로 sema\_up() 함수 상에서 semaphore를 waiting 하는 thread를 다시 ready queue에 넣게 되면, 이때도 Priority에 따라 ready\_list에 정렬되어 삽입되어야 함으로 src/threads/synch.c 코드의 waiting list 상에서 가장 높은 priority를 가지는 waiting thread를 찾고, 이 thread를 ready상태로 바꾸어야 한다.

3) **Advanced Scheduler** (Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.)

: Advanced Scheduler에서 Priority는 CPU 점유율, 다른 thread와의 상관관계 (Niceness) 등 다양한 요소를 포함하여 계산된다. 명세서에 제시된 계산식은 아래와 같다.

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

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

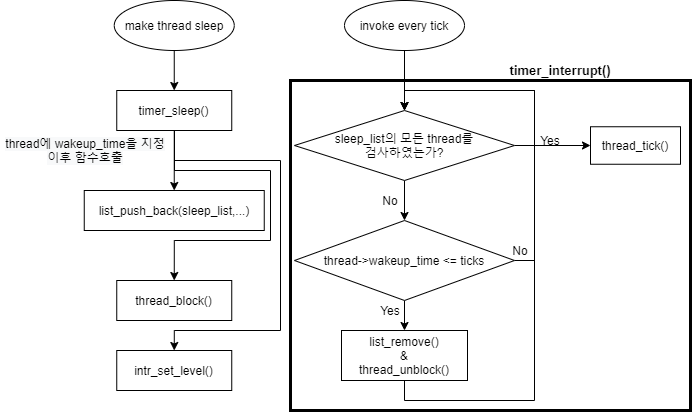
load\_avg = (59/60) \* load\_avg + (1/60) \* number\_of\_ready\_threads

따라서 매번 각 Thread의 Priority를 위와 같은 계산을 통해 산출해야 하는데, Advanced Scheduler상에서는 Float point arithmetic을 지원하지 않는다 대신 Fixed-Point real arithmetic을 사용해야한다. 때문에 대신 Fixed-Point real을 32bit int data type로 저장하지만 실제 연산 방법은 달라진다. 이러한 연산을 위해 int와 Fixed-Point real간의 형 변환 함수, Fixed-Point real의 곱셈 나눗셈 함수가 필요하다. 때문에 위에서 말한 4 종류의 함수를 새롭게 src/threads/thread.c 코드에 추가로 구현한다. 또한 계산의 인자인 recent\_cpu와 nice가 저장될 변수를 각 thread에 추가해야 한다. 때문에 src/threads/thread.h 코드의 thread 구조체에 해당 변수를 추가한다. 이제 추가한 변수를 초기화 해야 함으로 src/threads/thread.c 코드의 thread\_init() 함수에 각 변수의 초기화 기능을 추가한다. 또한 init\_thread() 함수에서 실행되는 thread에 추가된 변수 정보를 초기화 하는 기능을 추가한다. 이제 기존에 선언되어 있는 thread\_get\_nice(), thread\_get\_load\_avg(), thread\_get\_recent\_cpu(), thread\_set\_nice() 함수를 구현하고, 계산식에 따라 load\_avg와 recent\_cpu를 갱신해주는 함수를 새롭게 선언하여 구현한다. 갱신된 load\_avg와 recent\_cpu 정보를 사용해 priority를 계산, 갱신해주는 update\_priority() 함수를 구현한다. ‘-aging’ 조건이나 ‘-mlfqs’ 조건이 입력된다면 timer\_interrupt() 함수가 호출될 때 마다 priority 조건에 맞게 각 변수를 갱신해야 한다. 이때 각 조건여부를 확인하기 위해 조건 명령어를 확인해야 함으로 src/threads/init.c 코드상의 각 명령어를 parsing하는 parse\_options()함수를 변경한다. 이때 각 flag값을 새롭게 src/threads/thread.h, src/threads/thread.c 코드에 변수로 할당하고 각 flag값에 따라 추가적인 동작을 수행하도록 src/threads/thread.c 코드의 timer\_tick() 함수에 기능을 추가한다.

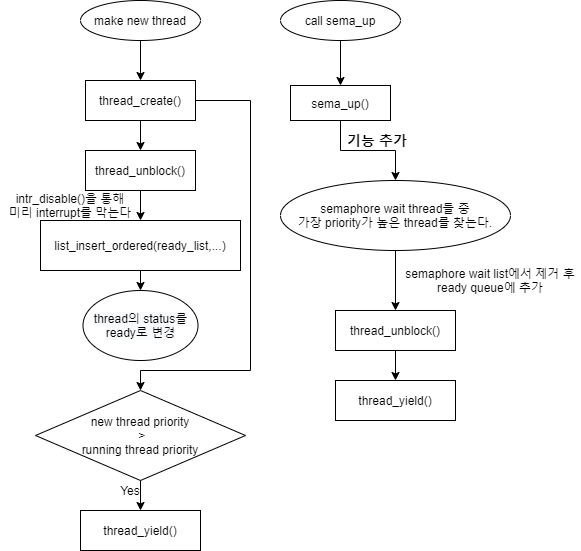
1. **연구 결과**
   1. **Flow Chart**

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

1) Alarm Clock



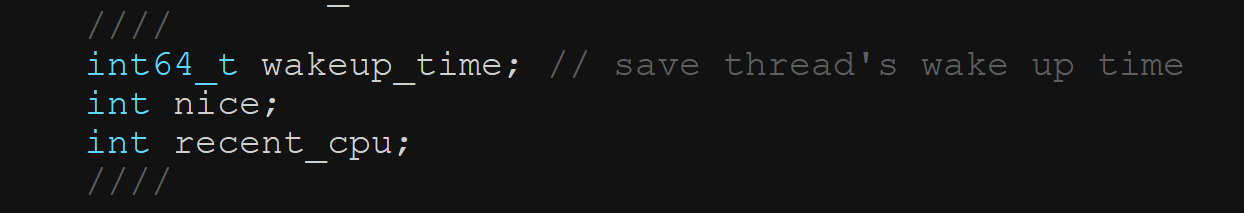
2) Priority Scheduling

****

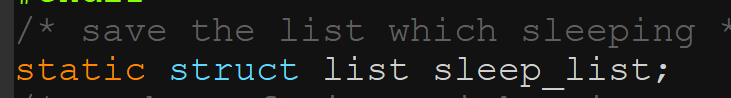
* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

1) **Alarm Clock** (Blocked 상태의 Thread를 어떻게 깨울 수 있는지 서술.)

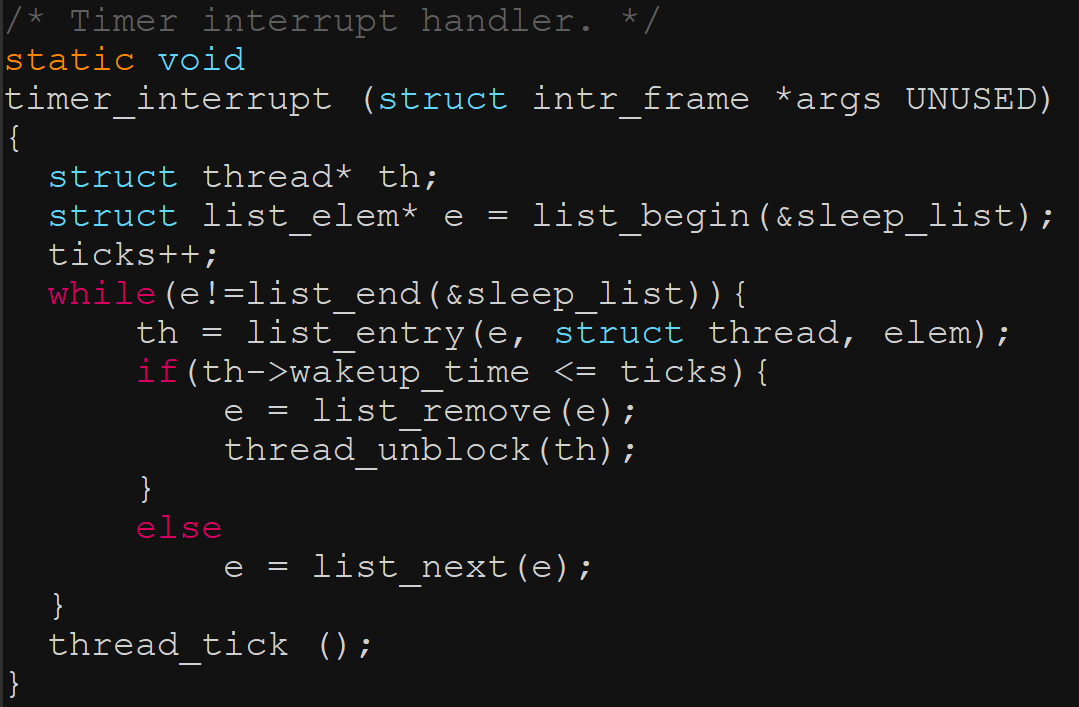
: Block 상태의 Thread가 깨어나야 할 시간이 된다면, Wake up을 통해 다시 Ready Queue에 넣어주어야 한다. 각 Thread마다 언제 깨어날지를 저장하는 변수를 추가하여야 함으로 src/threads/thread.h 코드의 thread 자료구조에 해당 변수를 추가해야 한다.



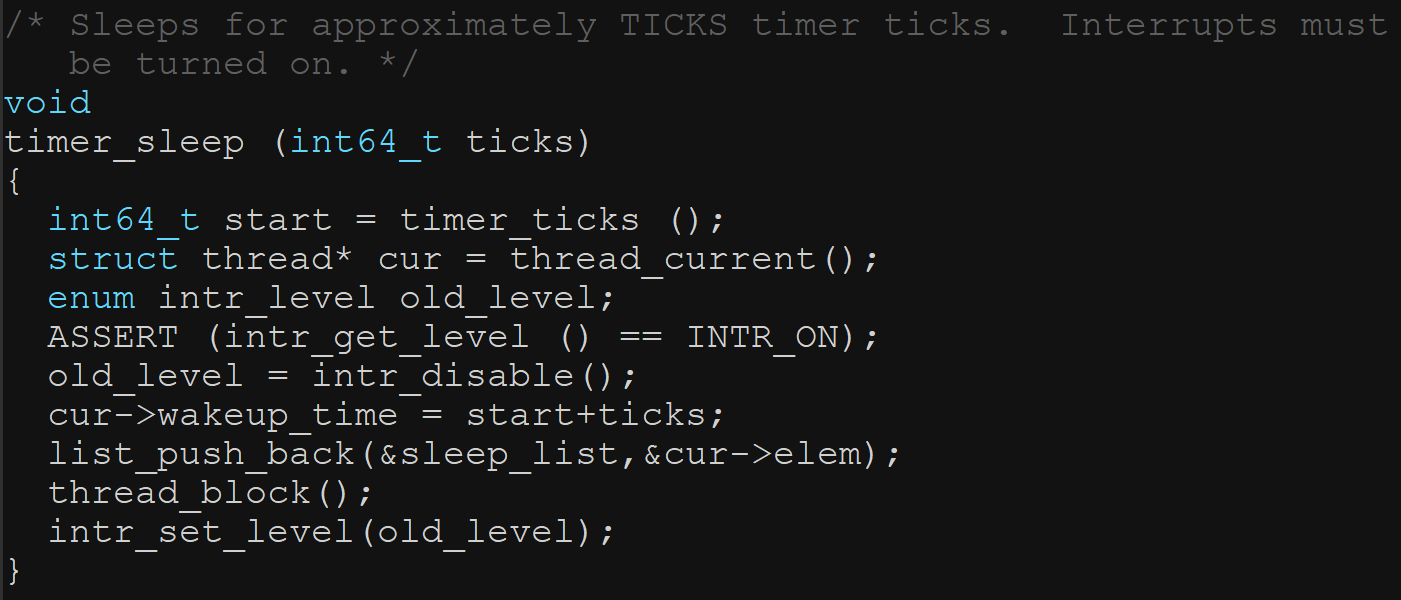
Sleep 상태인 thread를 저장할 list를 새로 만들어야 함으로, src/devices/timer.c 코드에 자료구조를 새롭게 선언한다.



timer\_interrupt() 함수를 통해 주기적으로 Sleep상태인 Thread들이 저장된 list를 순회하면서 thread의 wakeup\_time이 현재까지 진행된 tick(time)보다 작은지 확인하고, 작다면, 해당 thread는 이제 깨어나야 하는 thread라는 의미임으로, Sleep List에서 삭제, thread\_unblock() 함수를 통해 thread의 state를 ready로 변경, 그리고 Ready Queue에 추가해준다. 이루 thread\_tick() 함수를 호출해 tick을 진행시킨다.



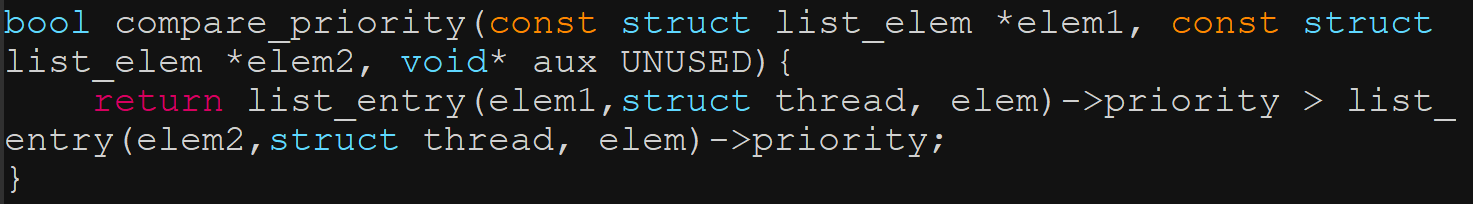
그리고 각 Thread의 Sleep 방법을 변경해야 함으로, src/devices/timer.c 코드의 timer\_sleep() 함수를 수정해야 한다.



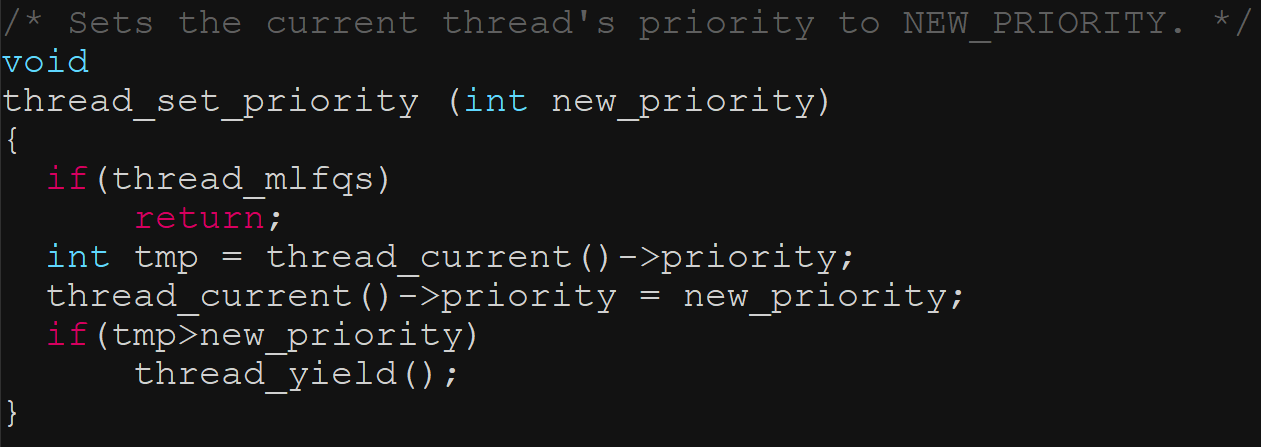
위의 순서도에서 설명한 것과 같이 sleep할 thread의 wakeup\_time을 지정하고 sleep\_list에 list\_push\_back() 함수를 사용해 삽입한다. 이후 thread\_block() 함수를 호출하여 현재 thread를 THREAD\_BLOCK 상태로 바꾸고 schedule() 함수를 사용해 다시 scheduling한다.

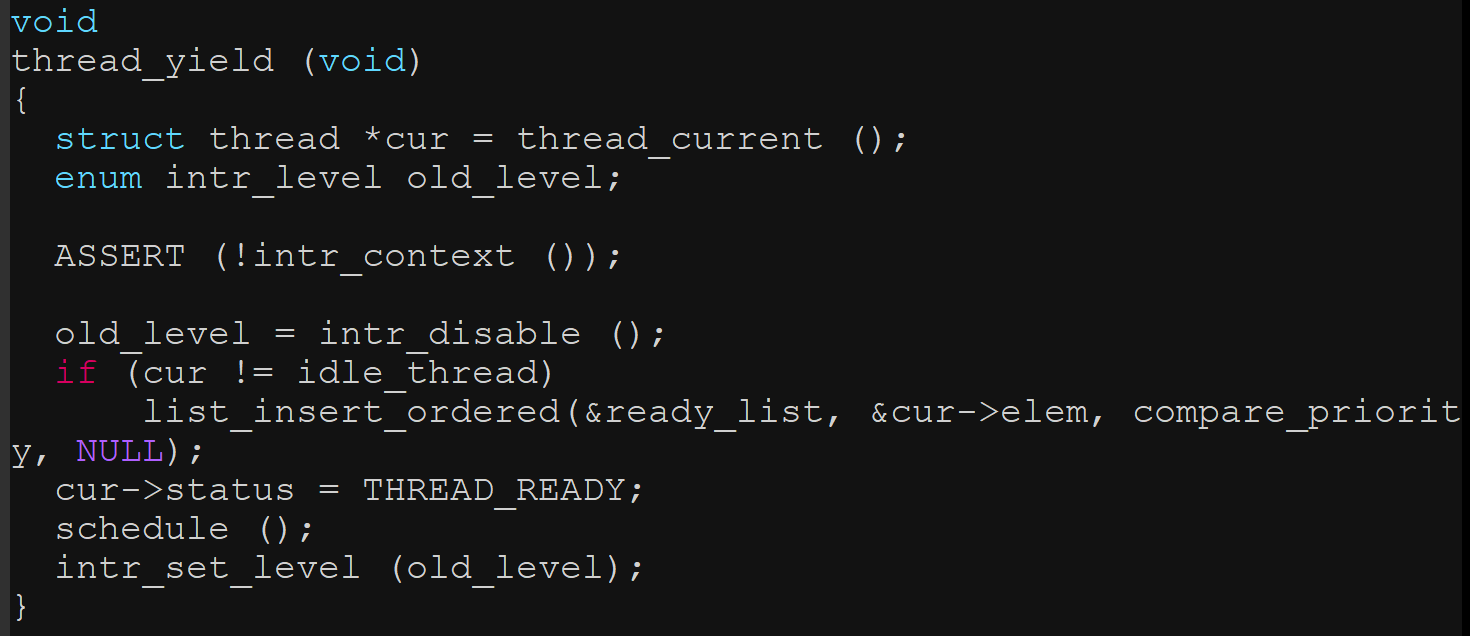
2) **Priority Scheduling** (Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.)

: 새로운 Thread가 Ready Queue에 들어온다면 해당 Thread의 Priority에 따라 적합한 위치의 ready\_list상에 저장해야 한다. 이러한 저장은 기존에 list에 구현되어있는 list\_insert\_ordered() 함수를 사용해 구현되는데, order의 기준인 Priority를 기준으로 Compare를 수행할 Compare 함수를 src/thread/thread.c 코드상에 새롭게 구현해야 한다.

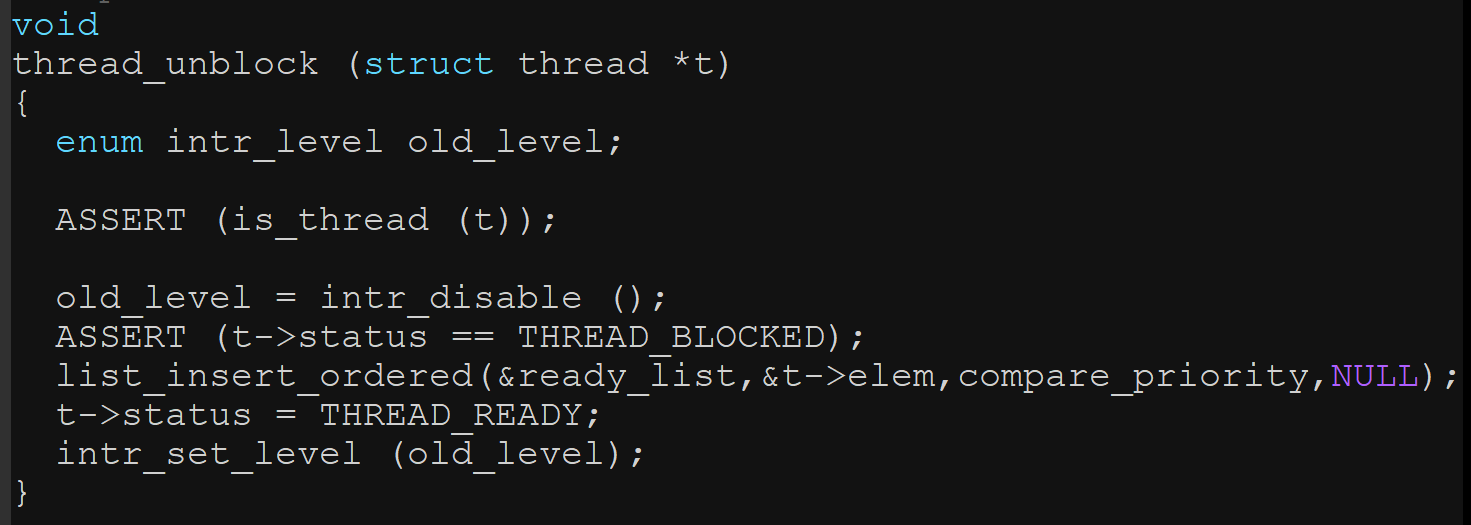


위에서 언급한 것처럼 Priority에 따라 현재 수행중인 Thread를 조정해야 함으로, src/thread/thread.c 코드상에 아래와 같은 각 기능을 수행하는 함수를 구현 또는 수정해야 한다. 먼저 thread\_set\_priority() 함수의 수정을 통해 새롭게 ready\_list에 들어온 thread의 Priority가 현재 수행중인 thread보다 더 높은지 검사를 하고, 높다면 thread\_yield()를 호출하여 Running thread의 변경을 수행해야 한다.

이때 호출되는 thread\_yield() 함수 또한 적절한 동작을 하도록 수정되어야 하는데, 현재 수행중인 thread를 적절한 priority에 따라 다시 ready\_list에 삽입해야 한다.

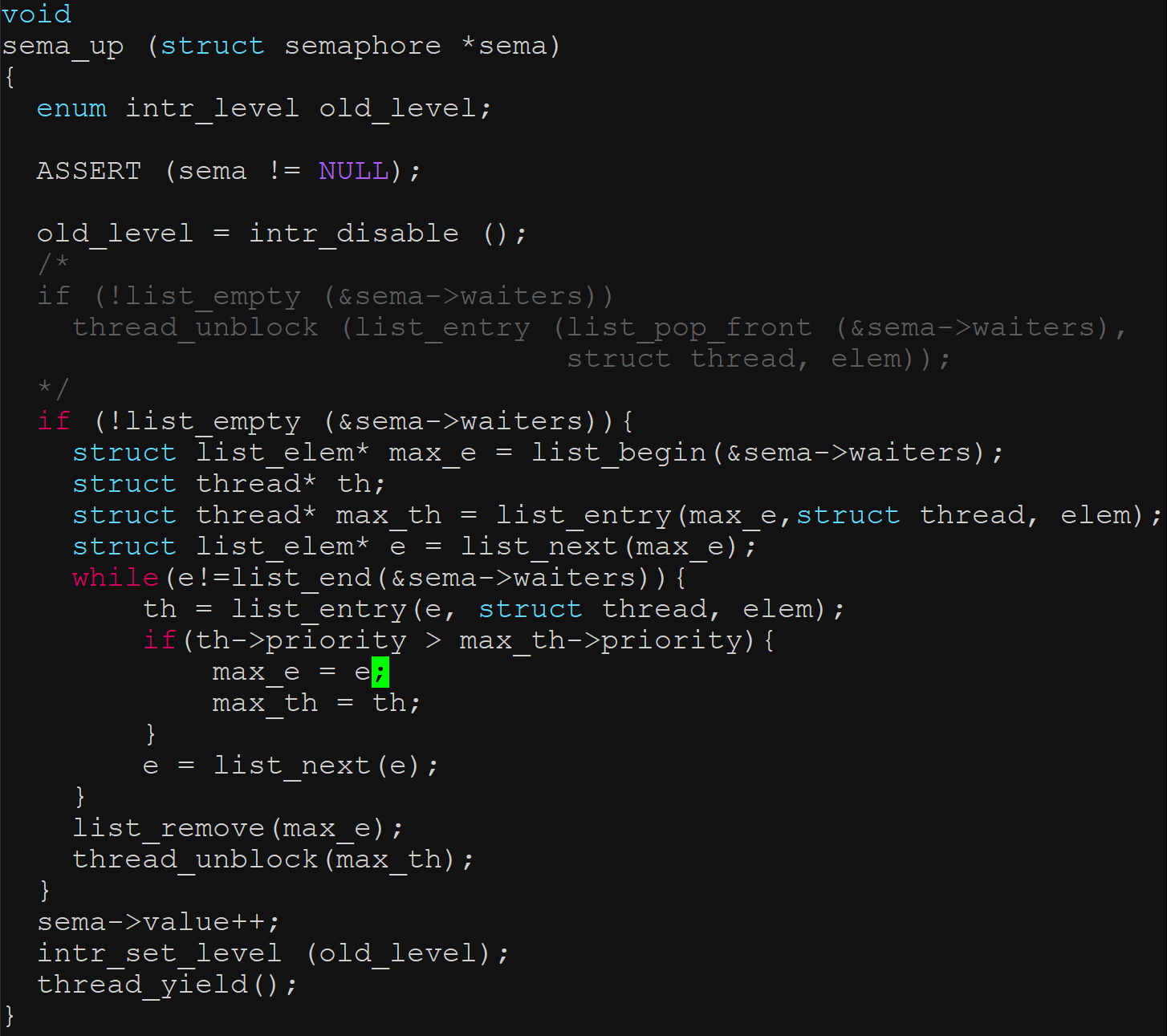


또한 Thread가 unblock될 때 Priority에 따라 ready\_list에 삽입되도록 thread\_unblock() 함수를 수정해야 한다.



list\_insert\_ordered() 함수를 사용해 priority에 따라 ready\_list의 적절한 위치에 삽입하고, block되어있던 thread의 상태를 ready로 바꾸어준다.

추자적으로 sema\_up함수에도 기능을 추가해야 하는데, if문과 내부 구현이 그 기능이다.



추가적으로 sema\_up() 함수 상에서 semaphore를 waiting 하는 thread를 다시 ready queue에 넣게 되면, 이때도 Priority에 따라 ready\_list에 정렬되어 삽입되어야 함으로 src/threads/synch.c 코드의 waiting list 상에서 가장 높은 priority를 가지는 waiting thread를 찾고, thread\_unblock() 함수를 사용해 이 thread를 ready상태로 바꾸어야 한다. 그리고 ready\_list의 변화가 있었음으로, thread\_yield() 함수의 호출을 통해 다시 Priority에 따라 실행되는 tread를 조정한다.

3) **Advanced Scheduler** (Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.)

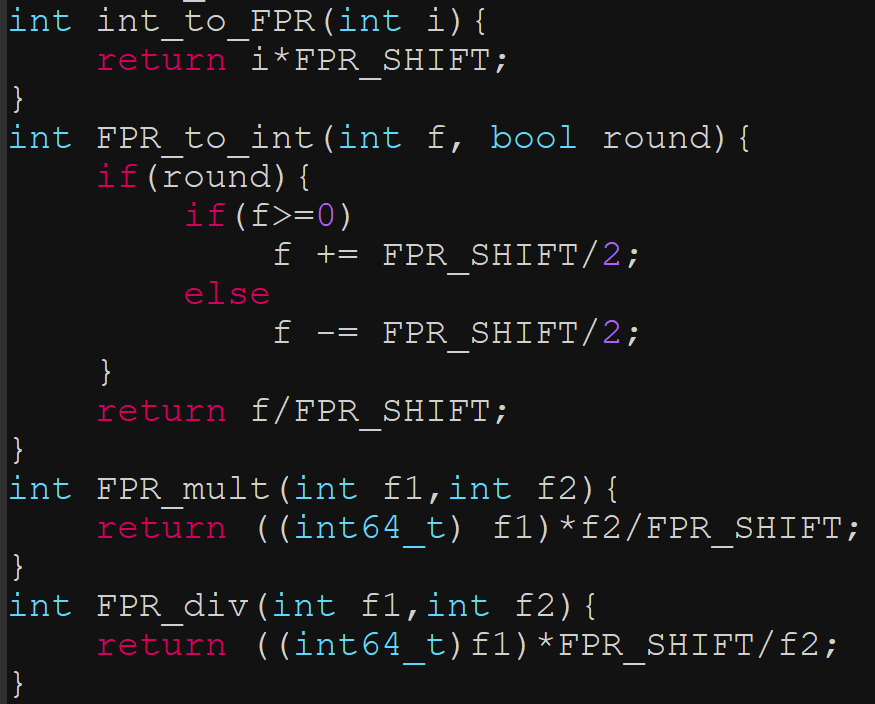
: Advanced Scheduler에서 Priority는 CPU 점유율, 다른 thread와의 상관관계 (Niceness) 등 다양한 요소를 포함하여 계산된다. 명세서에 제시된 계산식은 아래와 같다.

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

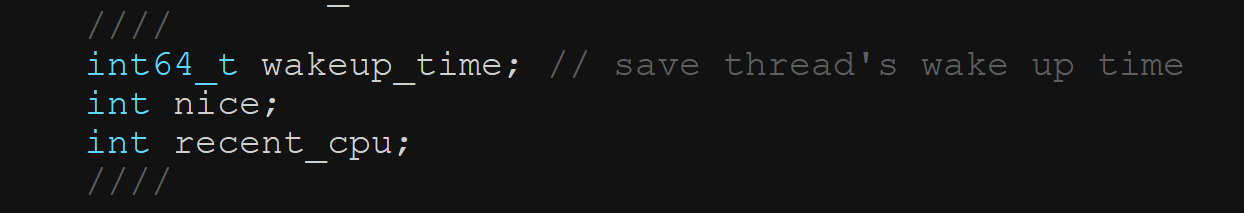
recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu + nice

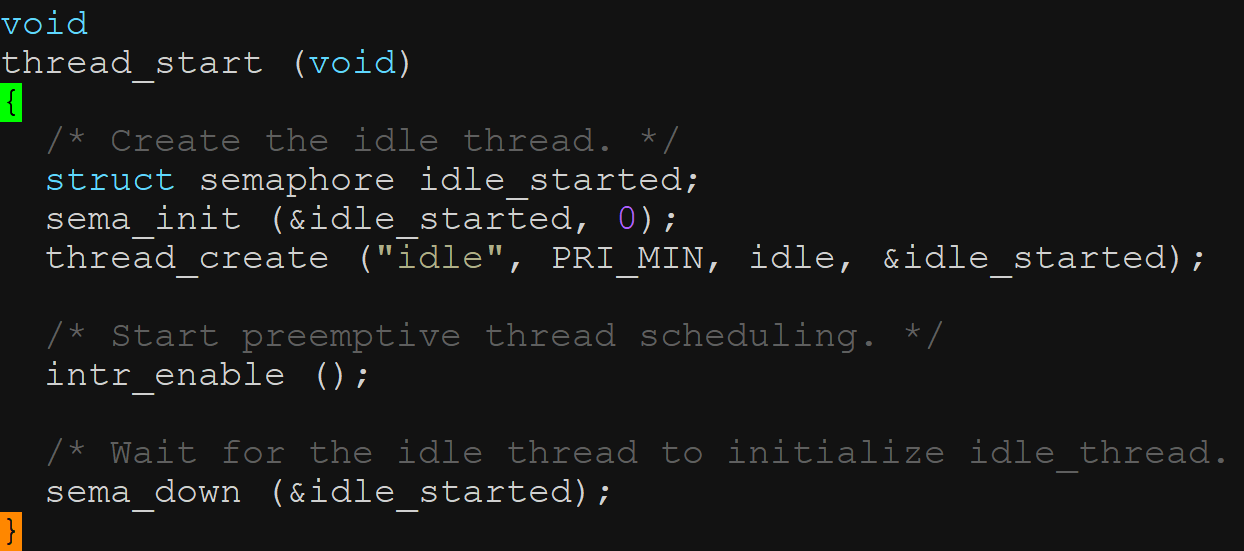
load\_avg = (59/60) \* load\_avg + (1/60) \* number\_of\_ready\_threads

이때 사용되는 Fixed-Point real arithmetic 연산을 위해 int와 Fixed-Point real간의 형 변환 함수, Fixed-Point real의 곱셈 나눗셈 함수가 필요하다. 때문에 위에서 말한 4 종류의 함수를 새롭게 src/threads/thread.c 코드에 추가로 구현한다.

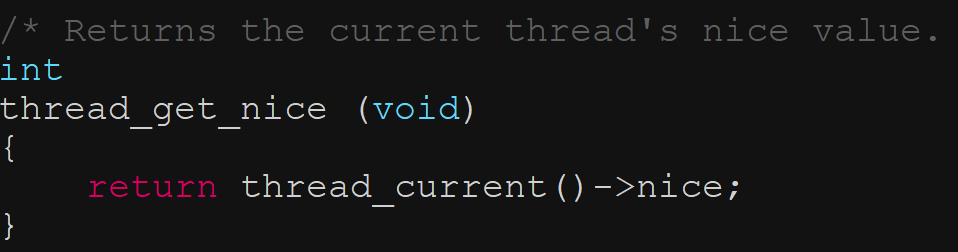


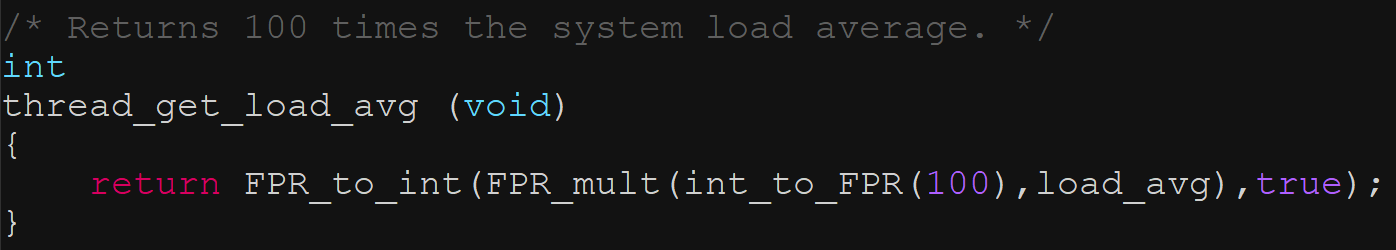
또한 계산의 인자인 recent\_cpu와 nice가 저장될 변수를 각 thread에 추가해야 한다. 때문에 src/threads/thread.h 코드의 thread 구조체에 해당 변수를 추가한다.

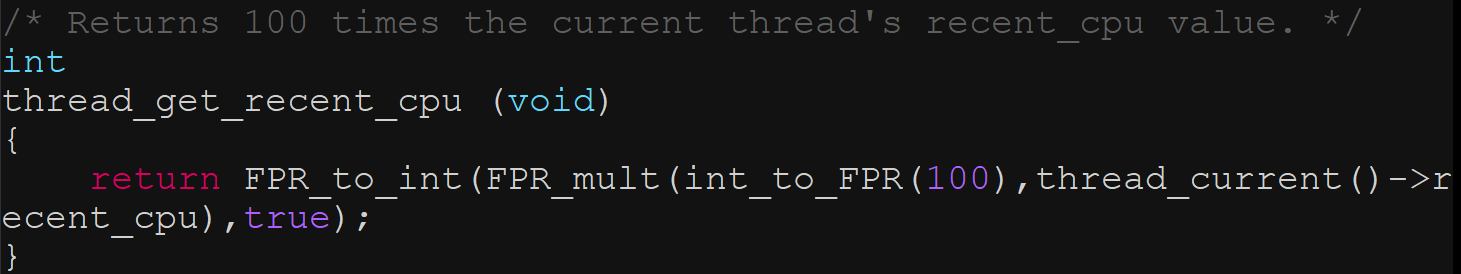
이제 추가한 변수를 초기화 해야 함으로 src/threads/thread.c 코드의 thread\_init() 함수에 각 변수의 초기화 기능을 추가한다. 또한 init\_thread() 함수에서 실행되는 thread에 추가된 변수 정보를 초기화 하는 기능을 추가한다.

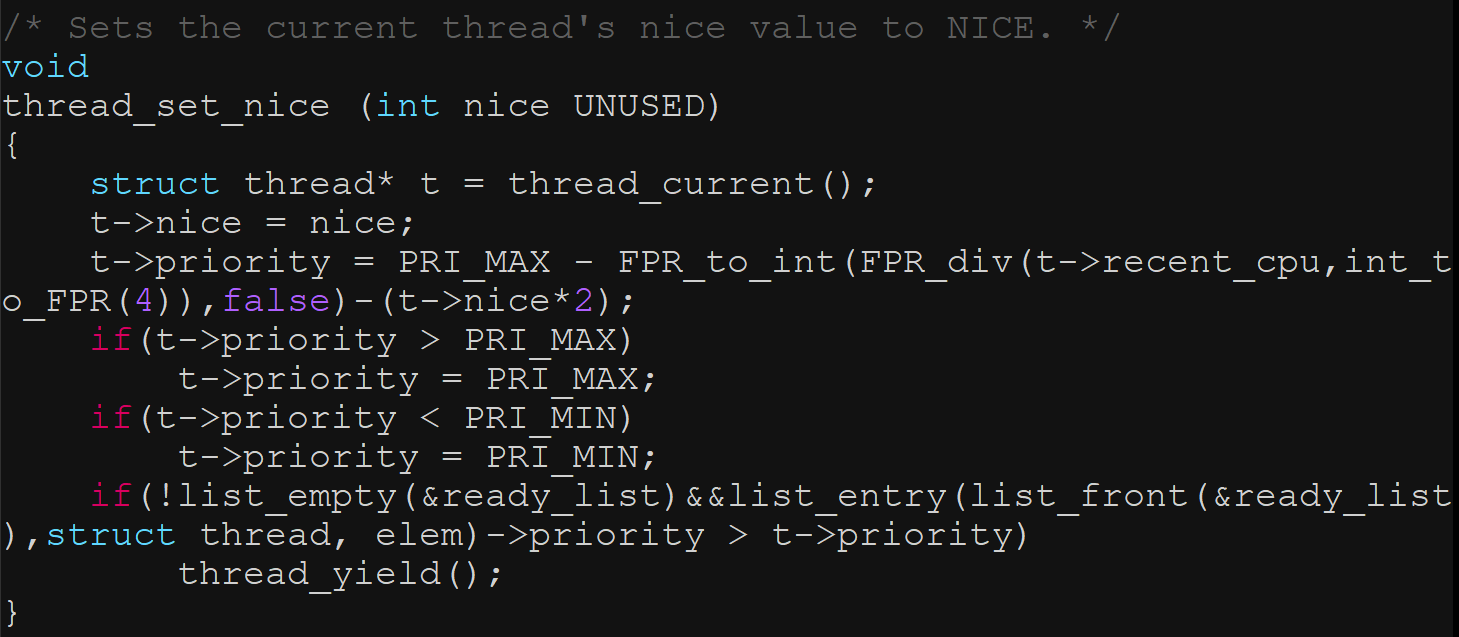


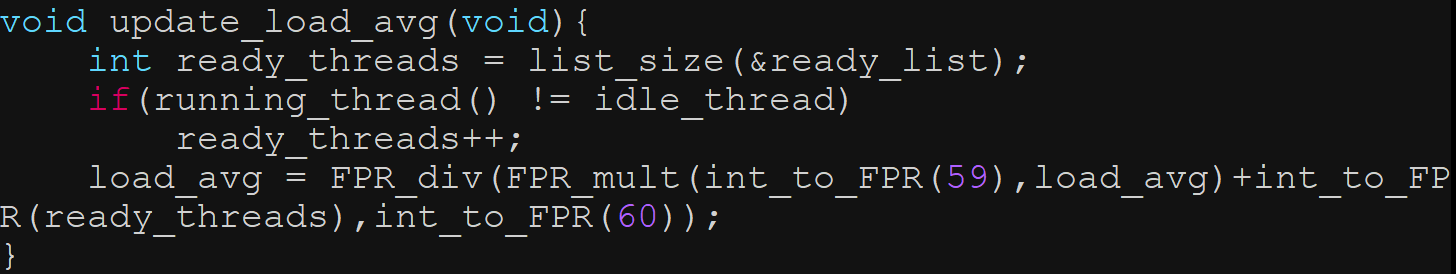
이제 기존에 선언되어 있는 thread\_get\_nice(), thread\_get\_load\_avg(), thread\_get\_recent\_cpu(), thread\_set\_nice() 함수를 구현하고, 계산식에 따라 load\_avg와 recent\_cpu를 갱신해주는 함수를 새롭게 선언하여 구현한다.

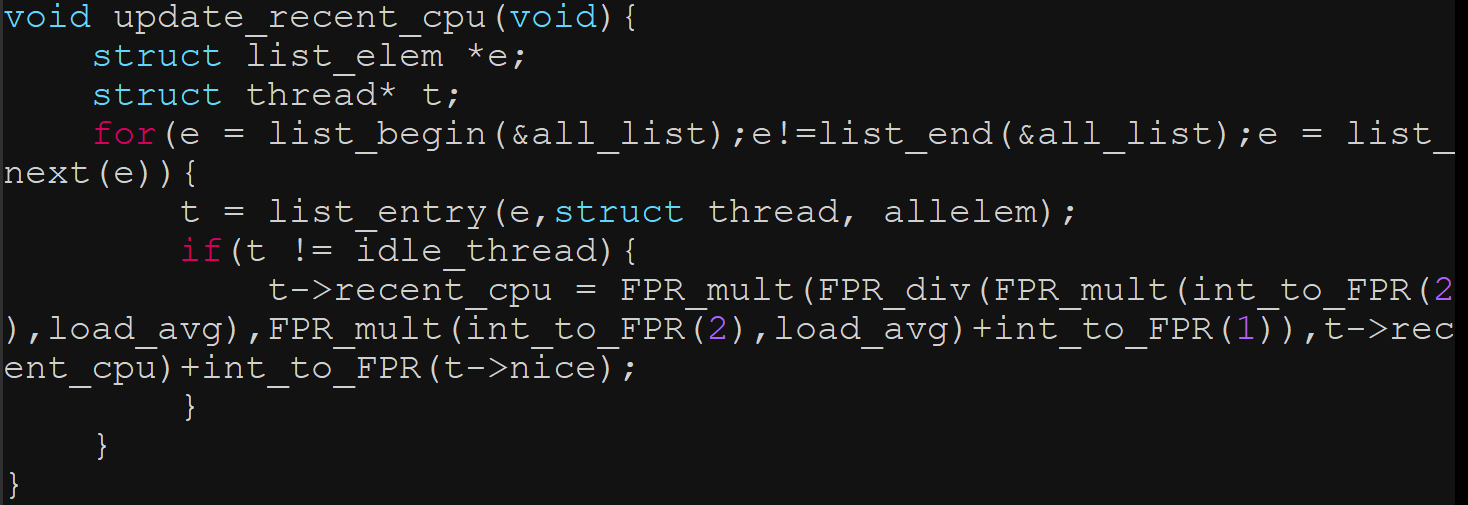




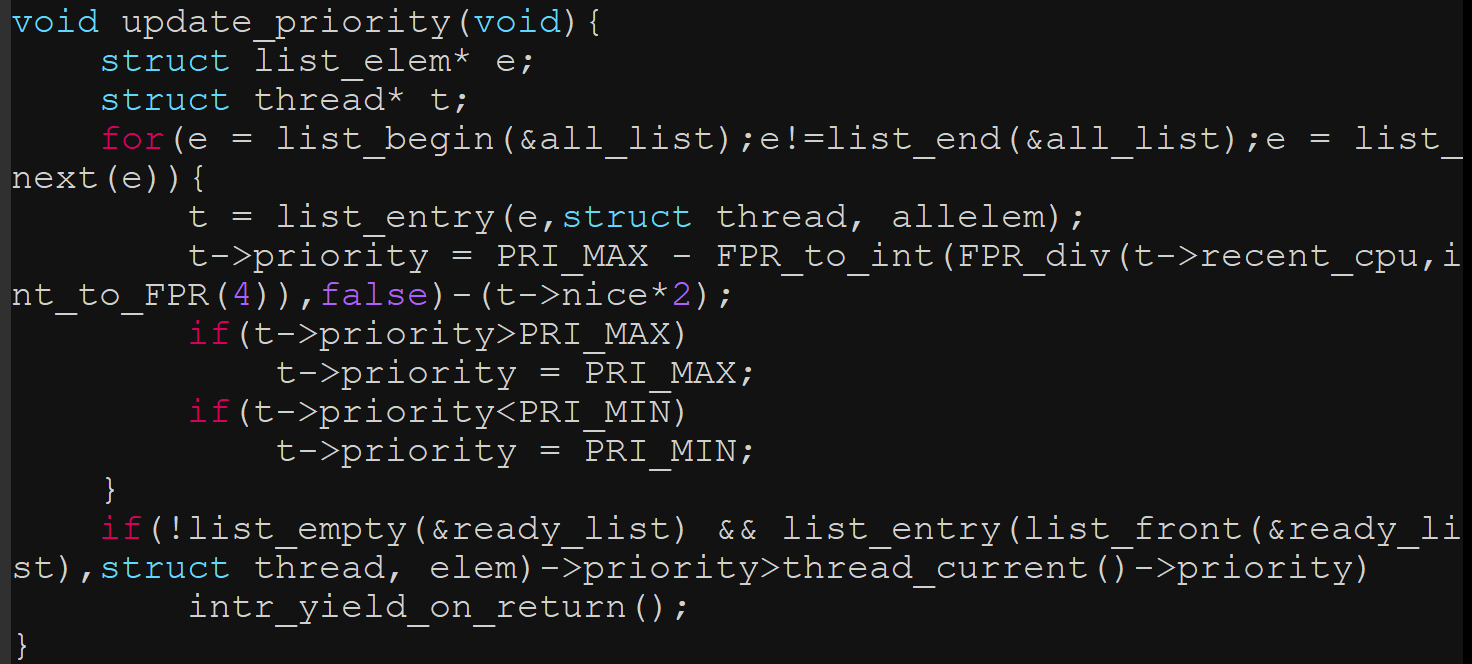




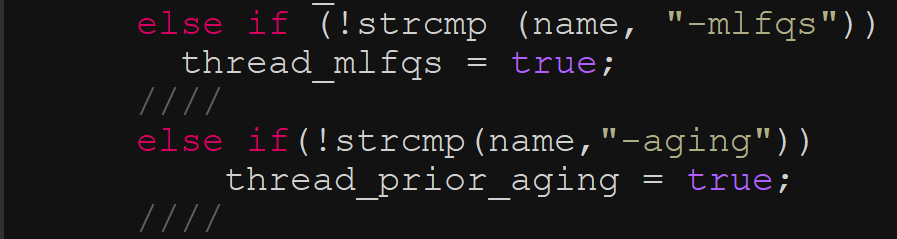




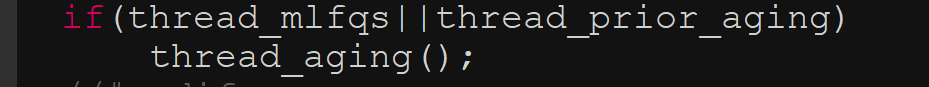
갱신된 load\_avg와 recent\_cpu 정보를 사용해 priority를 계산, 갱신해주는 update\_priority() 함수를 구현한다.



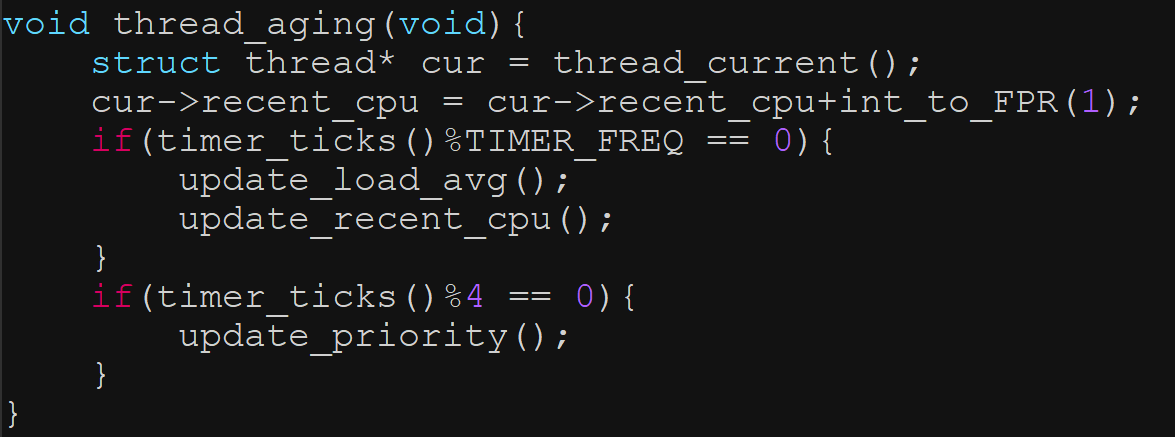
‘-aging’ 조건이나 ‘-mlfqs’ 조건이 입력된다면 timer\_interrupt() 함수가 호출될 때 마다 priority 조건에 맞게 각 변수를 갱신해야 한다. 이때 각 조건여부를 확인하기 위해 조건 명령어를 확인해야 함으로 src/threads/init.c 코드상의 각 명령어를 parsing하는 parse\_options()함수를 변경한다.



이때 각 flag값을 새롭게 src/threads/thread.h, src/threads/thread.c 코드에 변수로 할당하고 각 flag값에 따라 추가적인 동작을 수행하도록 src/threads/thread.c 코드의 timer\_tick() 함수에 기능을 추가한다.

****

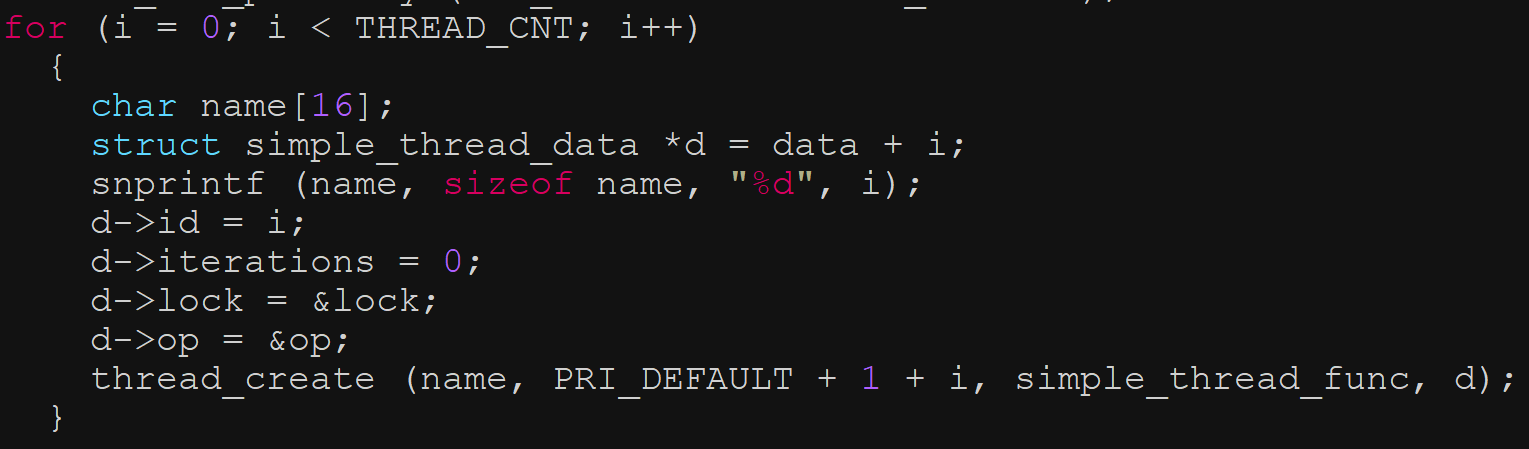
이때 호출되는 thread\_aging() 함수는 각 tick마다 호출되어 위의 계산식 대로 thread의 load\_avg, recent\_cpu, priority를 update한다.



* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석
* make check 수행 결과를 캡처하여 첨부

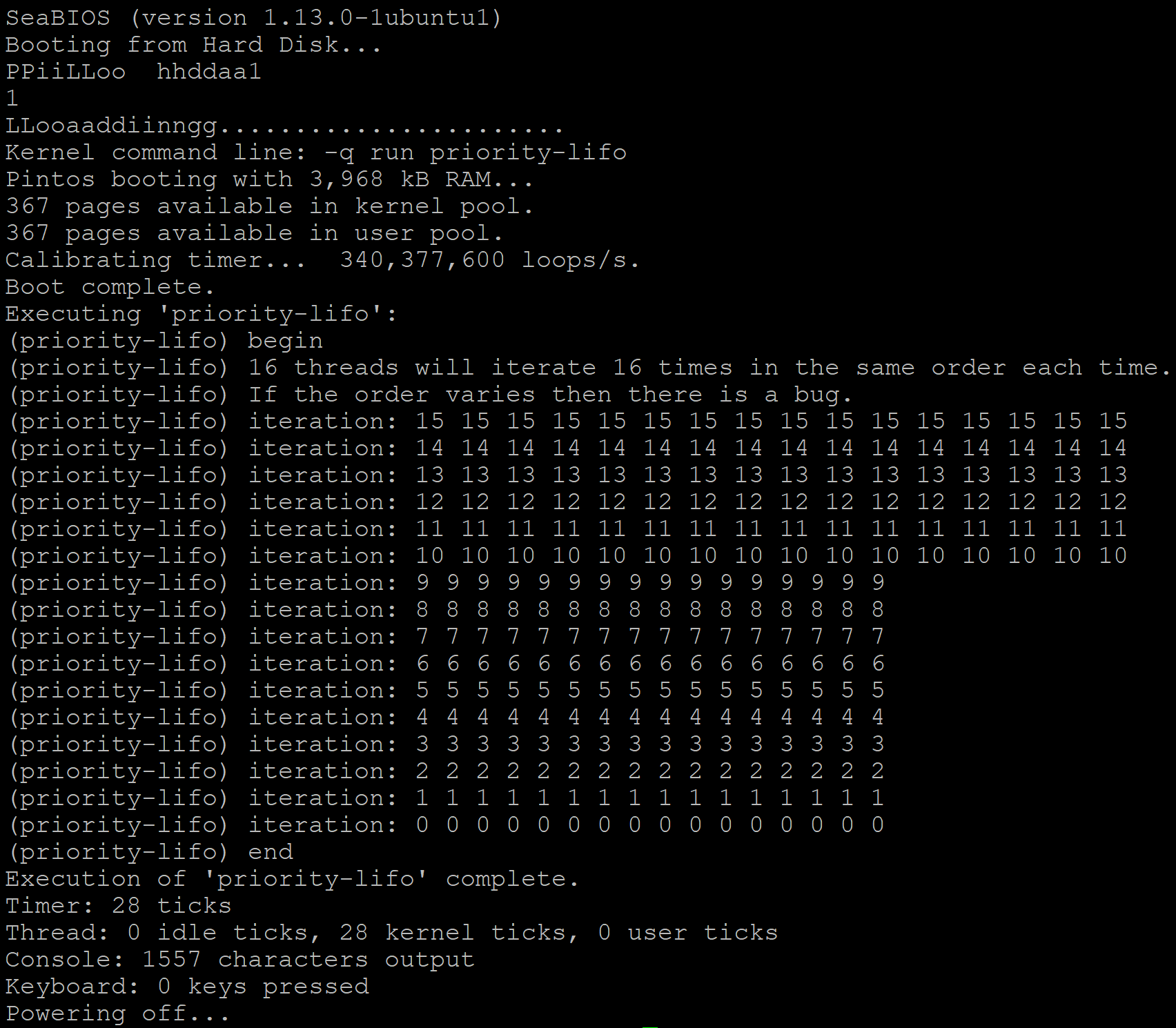
<priority-lifo.c 코드 분석>

코드를 확인해 보면, 코드상에서 가장 중요한 동작을 수행하는 부분은 아래와 같다.



코드를 보면, priority를 1씩 증가시키면서 thread를 만들고 있음을 확인할 수 있다. 따라서 우리가 이러한 코드의 실행 후 기대할 수 있는 동작은 priority가 더 높은 thread를 계속 만들기 때문에 이후에 추가된 thread가 priority가 더 높을 것이고 더 먼저 끝날 것이다.

<priority-lifo 테스트 결과 분석>



위의 코드 분석으로 예상한 것과 같이 이후에 추가된 더 높은 priority를 가지는 thread가 더 먼저 실행되고 더 먼저 끝난다.

<make check 수행 결과>

