**Pintos Project 3: Threads**

담당 교수 : 문의현

학번 / 이름 : 20191251/박효리

개발 기간 : 11/21 – 11/4

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술

1) Alarm Clock

* 사용하지 않는 Thread를 timer ticks의 시간만큼 sleep 상태에 놓기 위해서 시간이 다 지났는지를 반복문 안에서 확인해준다. 이때 thread가 running 상태였다가 ready 상태였다가 하기 때문에 비효율적이다. 따라서 thread에 block 상태를 부여해준다. 즉 timer ticks가 지나지 않은 thread는 block state로 바꿔서 block 상태인 thread만 큐에 저장하고 wake up time이 되면 다시 ready큐로 옮겨준다.

2) Priority Scheduling

* 프로젝트 진행 전 코드를 보면 thread\_yield나 thread\_unblock함수가 불리면 current thread나 unblocked thread가 우선순위에 상관 없이 그냥 ready 큐의 맨 뒤에 놓이게 된다. 이때 우선순위 스케줄링을 추가하면 우선순위가 높은 thread의 경우 바로 CPU에 yield해줄 수 있으며 ready 큐에 thread들이 우선순위대로 정렬되어 있게 된다. 우선순위는 최소 0부터 최대 63의 숫자로 표현되며 default값은 31이다. 숫자가 높을수록 높은 우선순위를 의미한다. 그러나 우선순위 스케줄링의 문제는 계속해서 우선순위에 밀리는 thread가 생길 수 있다는 것이다. 이러한 starvation의 해결을 위해 thread가 ready큐에서 대기하는 시간이 길어질수록 우선순위를 증가시키는 aging기법을 포함한다.

3) Advanced Scheduler (추가구현을 한 경우)

* Advanced Scheduler 또는 BSD Scheduler는 여러 가지 큐를 사용하는 multilevel feedback queue나 multilevel ready queue를 사용해서 스케줄링하는 기법이다. 기본적으로 round robin을 따른다.
  1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

앞에서 언급했듯 blocked 상태 thread를 저장하는 큐를 만든다. 시간이 지날 때마다 이 queue에 저장된 thread들의 wake up time을 확인한다. 만일 wake up해야한다면 다시 해당 thread를 ready queue로 옮겨준다. 따라서 깨운다는 건 blocked thread queue에서 ready queue로 옮기는 작업이 될 것이다.

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

현재 running중인 thread 보다 더 높은 우선순위를 갖는 thread를 찾아 CPU를 yield해야 한다. 따라서 running thread를 ready queue에 넣어 다시 대기 상태로 만들어주고, 해당 thread는 ready queue에서 꺼내어 실행시켜준다.

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

nice, load\_avg 그리고 recent CPU 이 3가지의 변수가 필요하다. 또한 이 값들을 불러오고 변경하는 thread\_get\_nice, thread\_set\_nice 등 함수들을 만들어준다.

nice는 -20부터 20 사이의 값을 갖고 처음에는 0의 값을 갖는다. child thread는 부모 thread의 nice를 물려받으며, nice가 양수라면 thread의 우선순위를 감소해준다. thread\_get\_nice와 thread\_set\_nice함수는 각각 thread의 nice값을 반환하고 nice값을 current thread의 nice로 바꿔서 우선순위를 다시 계산해주는 일을 하도록 구현한다.

load\_avg는 ready thread의 개수의 평균을 저장하며 전역 변수로 선언한다. 시스템이 처음으로 부트되면 0으로 초기화가 된다. 또한 이 값은 thread\_get\_load\_avg함수 안에서 매 초 아래 식에 의해 업데이트된다.

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

recent CPU는 thread에 의해 소모된 CPU 시간을 저장하는 변수이다. 이때 더 최근에 실행된 CPU시간이 더 나중에 실행된 CPU시간보다 크게 측정이 된다. recent CPU는 thread가 만들어지면서 0의 값을 가지게 되고, 자식 thread는 부모의 recent CPU를 물려받는다. Time interrupt가 발생하면 running thread는 recent CPU가 1 증가하게 된다. 또한 thread들은 매 초 아래 식에 의해 recent CPU값이 업데이트 된다.

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

이제 BSD스케줄링에서 우선순위는 이 세가지 값을 이용해서 아래 식과 같이 계산된다. (PRI\_MAX는 최대 우선순위 값인 63이다.)

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

**11/21-11/23: Alarm Clock 구현**

**11/25-11/29: Priority Scheduling 구현**

**11/30-12/5: Advanced Scheduler 구현 및 보고서 작성**

* 1. **개발 방법**

1) Alarm Clock

thread 구조체에 wake up time을 저장하는 정수형 필드 wakeUp을 추가해주고, 이를 thread\_init함수에서 0으로 초기화한다. sleep\_list라는 이름의 blocked 상태 thread를 관리하는 큐를 timer.c에서 선언하고 timer\_init함수에서 이 리스트를 초기화해준다. 또한 timer\_sleep함수와 timer\_interrupt함수도 수정해서 alarm clock을 구현해준다.

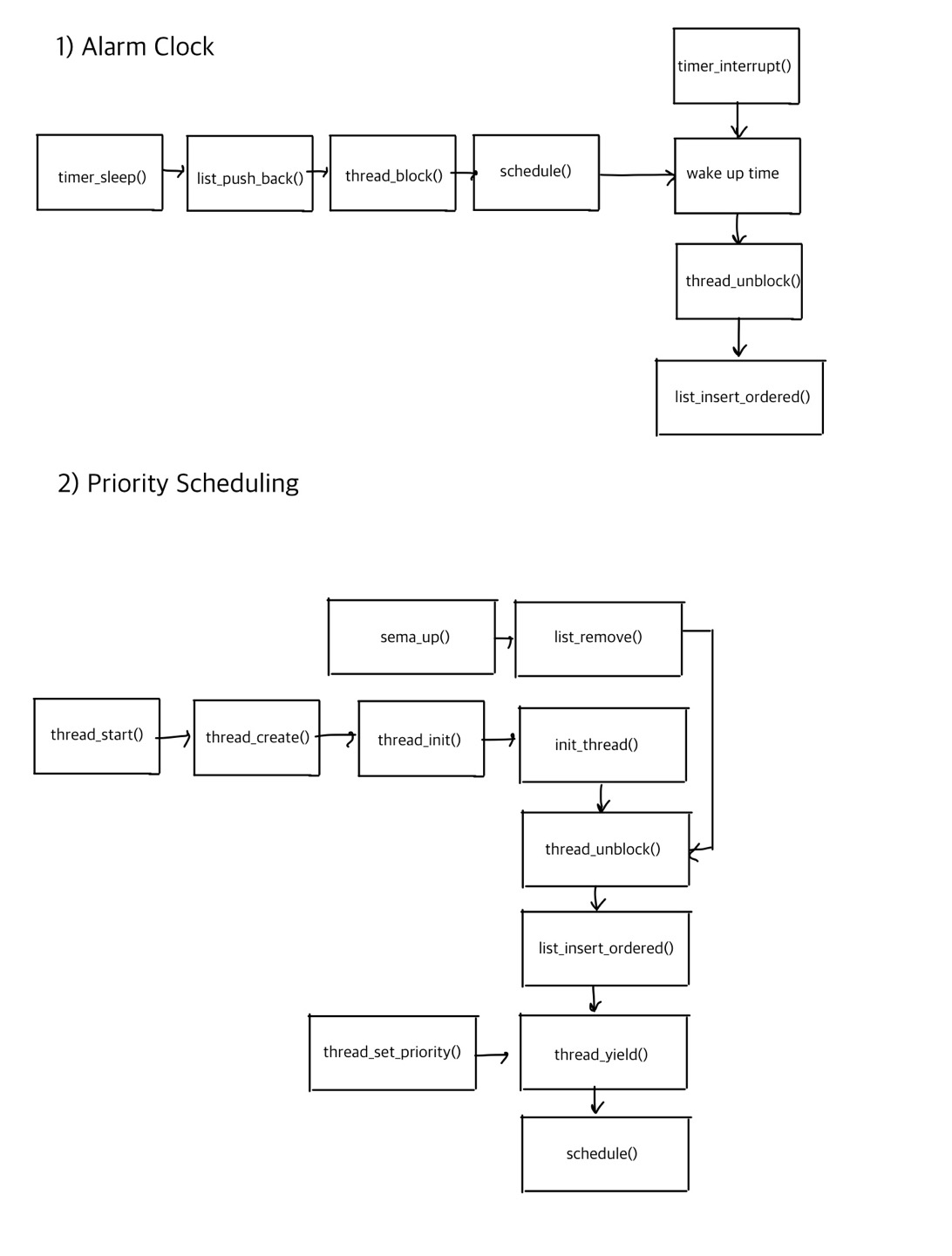
2) Priority Scheduling

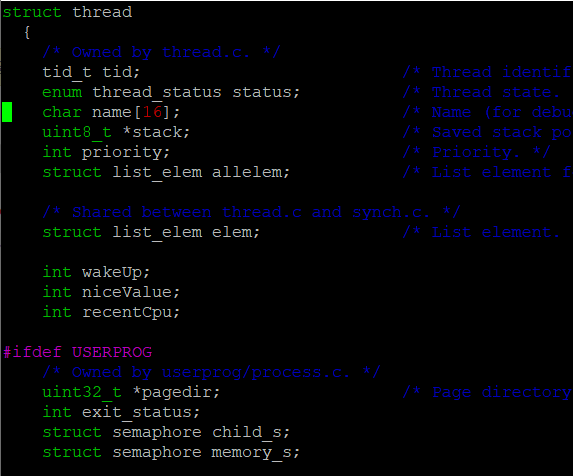
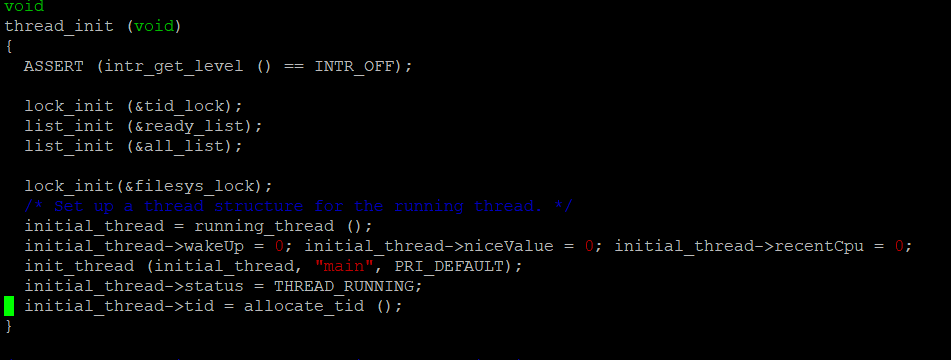
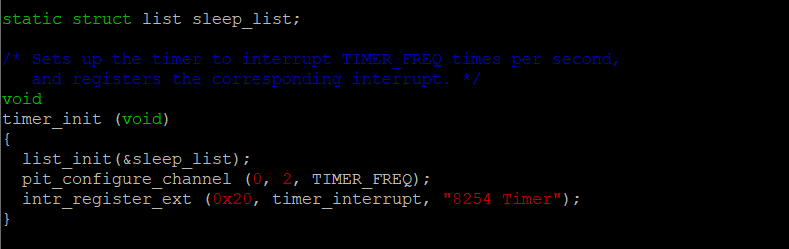
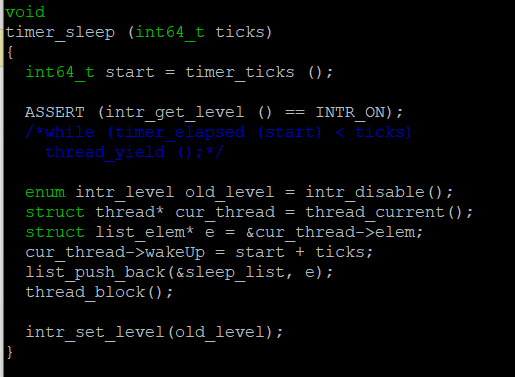
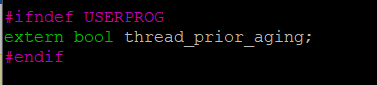
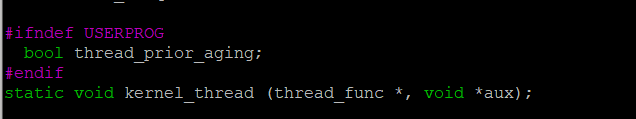
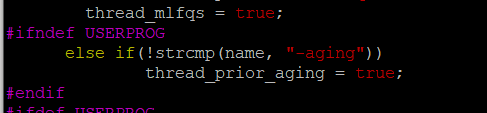
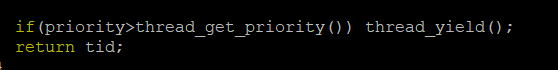
thead.h에서 bool타입의 thread\_prior\_aging을 선언한다. 이 변수를 init.c에서 true로 할당해서 aging이 잘 작동할 수 있게 한다. thread.c에서도 bool 타입의 thread\_prior\_aging을 선언한다. thread\_yield함수와 thread\_unblock함수에서 list\_push\_back부분을 list\_insert\_ordered함수를 사용하게끔 수정해서 단순히 ready 큐의 뒤에 삽입되는 것이 아닌, 우선순위대로 정렬되어 있게 한다. 그리고 thread\_create함수 역시 current thread보다 새로 생성하는 thread의 우선순위가 높을 경우를 확인해서 이 경우 새 thread를 yield하게끔 수정한다. 마지막으로 thread\_set\_priority함수 내에서 인수로 받은 new\_priority값으로 current thread의 우선순위를 바꿔주도록 수정한다. 이때 new\_priority가 원래 값보다 작다면 thread\_yield함수로 다시 스케줄링하도록 한다.

3) Advanced Scheduler

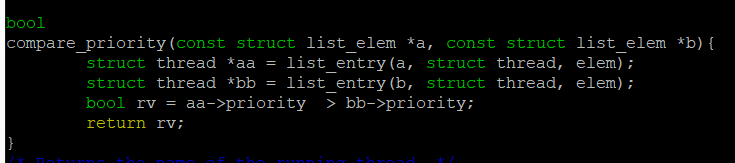
thread.h의 thead구조체에서 스케줄러 구현에 필요한 정수형 nice와 recent\_cpu 필드를 추가한다. 뿐만 아니라 load\_avg와 fixed point read arithmetic 연산을 위한 FRACTION 매크로도 thread.h에서 선언한다. thread.c에서는 thread\_init에서 nice와 recent\_cpu를 0으로 초기화한다. 만일 부모 스레드가 있다면 init\_thread함수 안에서 자식이 부모의 값을 물려받게끔 코드를 작성해준다. 그 다음, thread\_get\_nice, thread\_set\_nice, thread\_get\_recent\_cpu, thread\_get\_load\_avg, thread\_set\_priority, thread\_get\_priority를 전부 thread.c안에서 작성해준다. 이 작업은 위에서 적은 식들을 사용해서 구현해주면 된다. 하지만 이때 주의해야 할 것은 fixed point read arithmetic 연산을 잘 구현해야 한다는 것이다. timer.c에서는 매 tick마다 호출되는 timer\_interrupt 함수에서 thread의 우선순위를 업데이트 하는 부분을 작성한다. 이를 위해 함수 내부에서 thread\_prior\_aging이나 thread\_mlfqs가 true면 current thread의 nice, priority, recent\_cpu가 전부 식을 통해 업데이트 되도록 코드를 적어준다. 마무리 작업은 synch.c에서 이뤄진다. sema\_up함수를 우선순위에 sema\_up이 적용되게끔 수정한다. waiters는 sema를 기다리는 thread의 리스트이므로 가장 높은 우선순위를 갖는 thread를 반복문을 통해 찾고 sema\_up을 해준 다음, 다시 스케줄링 하게끔 thread\_yield함수를 불러준다.

1. **연구 결과**
   1. **Flow Chart**

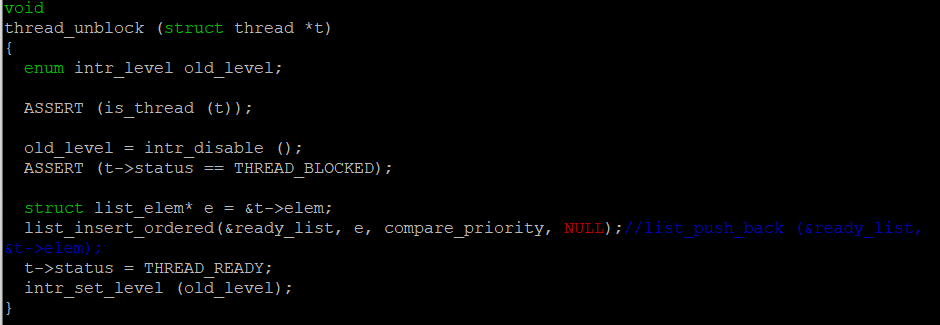


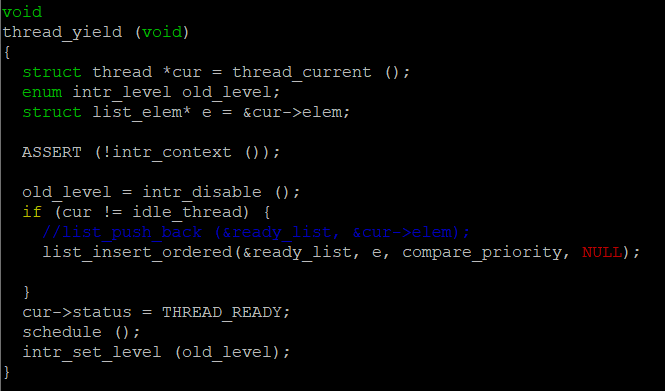
* 1. **제작 내용**
* 1) Alarm Clock
* 
* 
* thread 구조체에 wake up time을 저장하는 정수형 필드 wakeUp을 추가해주고, 이를 thread\_init함수에서 0으로 초기화한다.
* 
* sleep\_list라는 이름의 blocked 상태 thread를 관리하는 큐를 timer.c에서 선언하고 timer\_init함수에서 이 리스트를 초기화해준다.
* 
* timer\_sleep함수가 current thread의 wakeUp값을 인수로 받은 tick에 start를 더한 값으로 변경해주고 sleep\_list에 추가한 뒤 thread\_block()호출하게끔 수정해준다.
* 2) Priority Scheduling
* 
* 
* 
* thread.h와 thread.c에서 각각 thread\_prior\_aging 변수를 선언하고, init.c의 pasrse\_option함수에서 aging의 작동을 위해 thread\_prior\_aging에 true를 할당한다.
* 

thread.c의 thread\_create함수에서 현재 실행되고 있는 thread의 우선순위가 새로 생성하는 thread의 우선순위가 더 높을 때 thread\_yield를 다시 호출해서 CPU를 새로 생성된 thread에게 yield될 수 있게 한다.



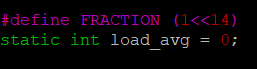
이 함수는 새로 선언한 함수로, 두 thread의 우선순위를 비교한 결과를 반환한다.

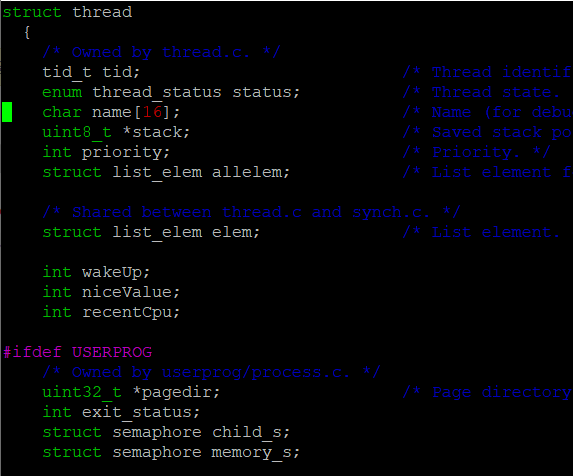




thread\_yield함수와 thread\_unblock함수에서 list\_push\_back부분을 list\_insert\_ordered함수를 사용하게끔 수정해서 단순히 ready 큐의 뒤에 삽입되는 것이 아닌, 우선순위대로 정렬되어 있게 한다.

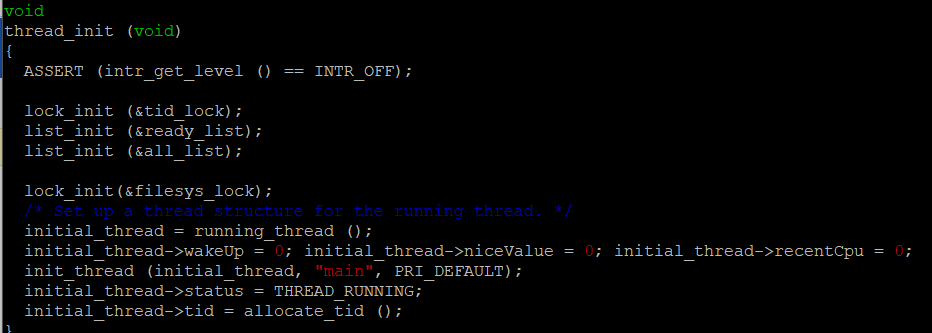
3) Advanced Scheduler

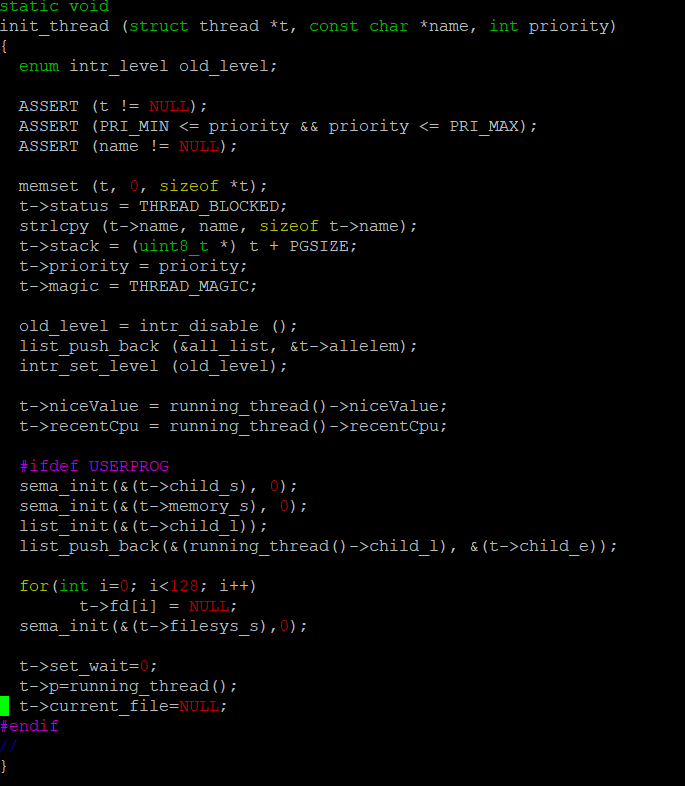




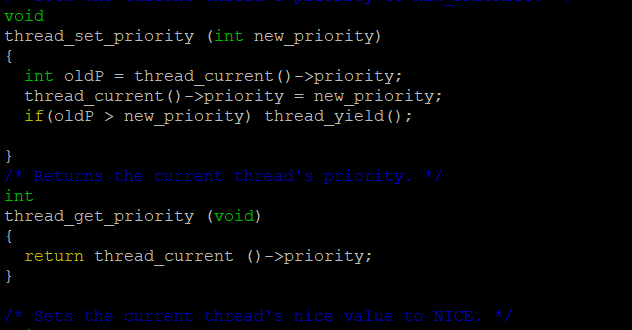
Advanced Scheduler 구현에 필요한 FRACTION, load\_avg을 thread.h에서 선언 해

주고 정수형 niceValue 필드와 recentCpu 필드를 thread 구조체에 추가해준다.

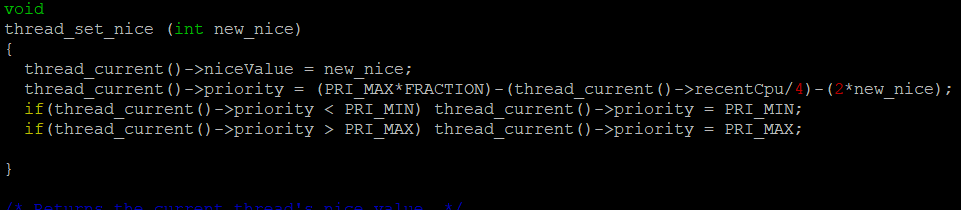


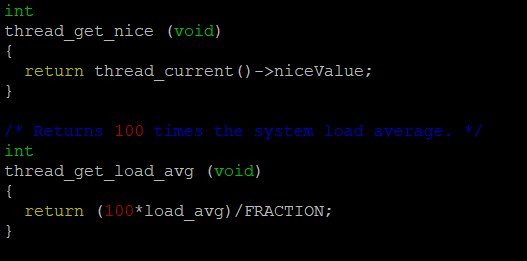


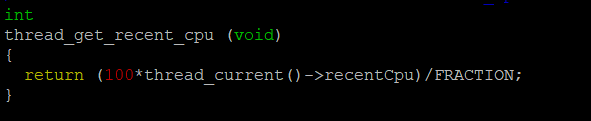
thread.c에서는 thread\_init에서 nice와 recent\_cpu를 0으로 초기화한다. 만일 부모 스레드가 있다면 init\_thread함수 안에서 자식이 부모의 값을 물려받게끔 코드를 작성해준다.



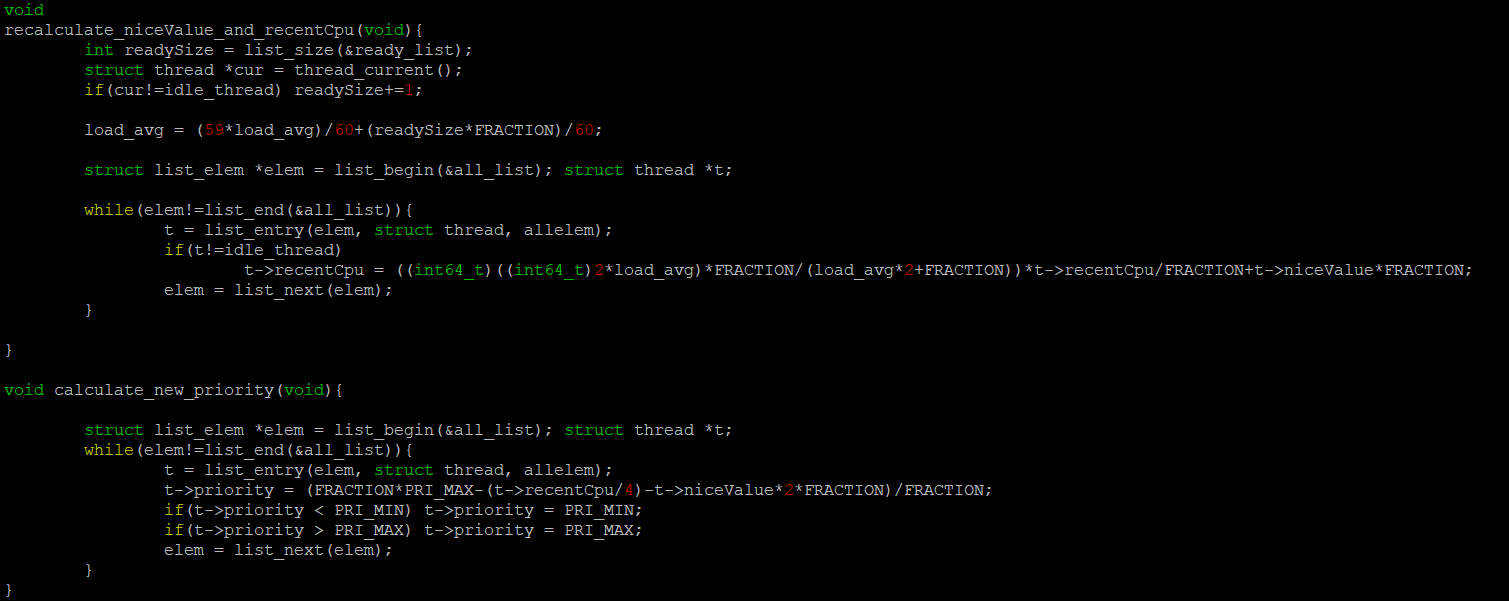
우선 thread\_set\_priority함수 내에서 인수로 받은 new\_priority값으로 current thread의 우선순위를 바꿔주도록 수정한다. 이때 new\_priority가 원래 값보다 작다면 thread\_yield함수로 다시 스케줄링하도록 한다. thread\_get\_priority는 그냥 현재 우선순위를 반환한다.



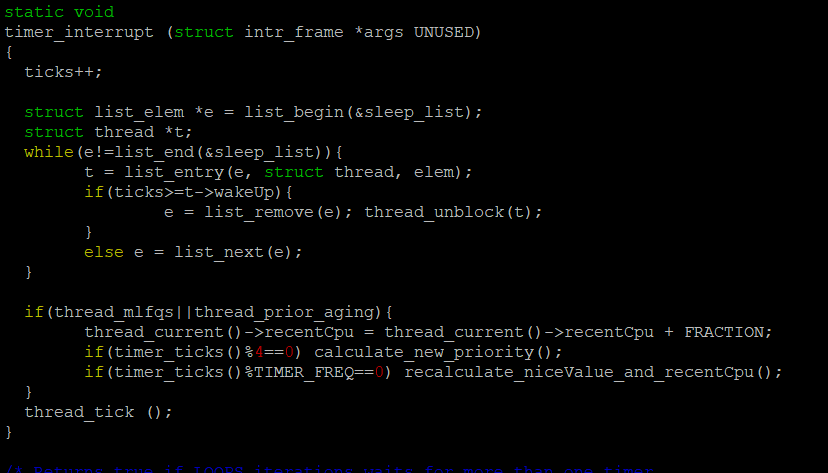




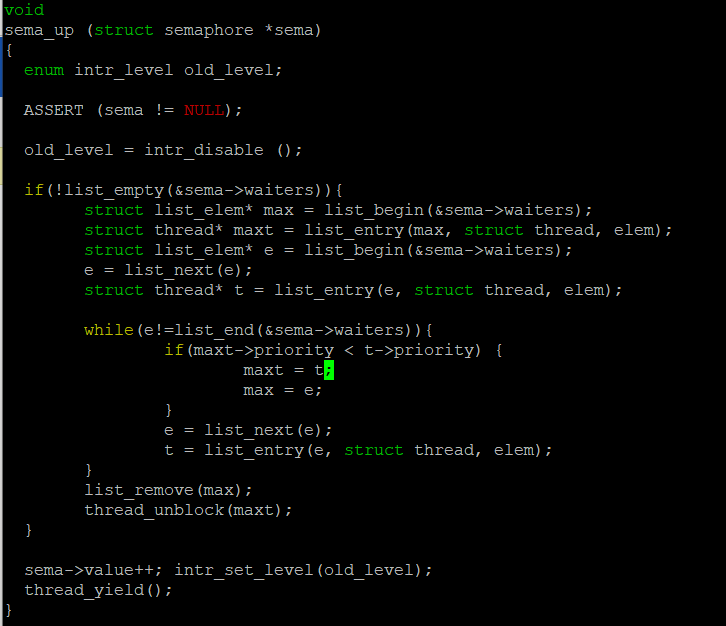
이 함수들은 위에서 언급한 식대로 3가지 변수들을 식을 통해 계산한 값을 반환하도록 하였다. thread\_set\_nice 함수는 인자로 받은 new\_nice의 값으로 current thread의 nice값을 갱신한다. 이때 이 new\_nice값을 우선순위에 반영해주고 만일 갱신된 우선순위가 PRI\_MAX나 PRI\_MIN을 넘어가게 되면 범위를 넘지 않게 각각의 값으로 갱신한다.



이 함수들은 새로 정의한 함수들이다. 먼저 recalculate\_niceValue\_and\_recentCpu함수는 계산식을 통해 nice와 recent\_cpu값을 갱신한다. calculate\_new\_priority 함수는 우선순위를 계산식을 통해 갱신하는 함수이다.

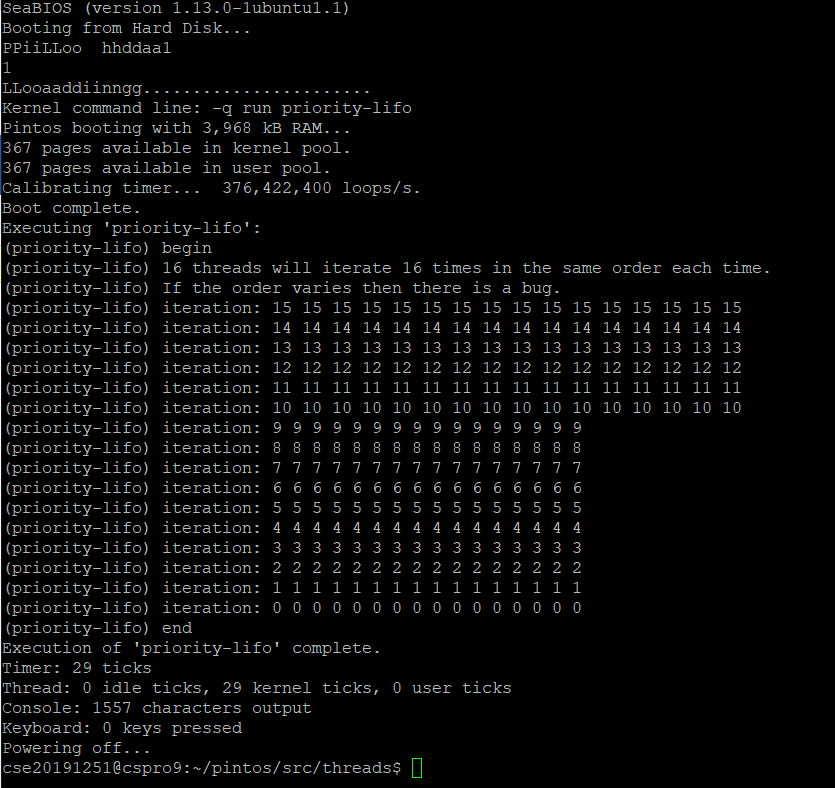


위 두 함수는 timer.c의 timer\_interrupt함수에서 호출된다. timer\_interrupt함수는 매 tick때 마다 thread의 우선순위를 갱신한다. 함수 내부에서 thread\_prior\_aging이나 thread\_mlfqs가 true면 current thread의 nice, priority, recent\_cpu가 전부 식을 통해 업데이트 되도록 작성해주었다.



마무리 작업은 synch.c에서 이뤄진다. sema\_up함수를 우선순위에 sema\_up이 적용되게끔 수정한다. waiters는 sema를 기다리는 thread의 리스트이므로 가장 높은 우선순위를 갖는 thread를 반복문을 통해 찾고 sema\_up을 해준 다음, 다시 스케줄링 하게끔 thread\_yield함수를 불러준다.

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



이는 ‘pintos -v -- -q run priority-lifo’ command를 실행한 결과이다. priority-lifo test는 같은 우선순위 값을 갖는 여러 개의 thread를 만들고 동일한 round robin의 순서에 따라 지속적으로 run을 한다. 이때 총 16개 thread가 lifo의 순서에 따라 모두 16번씩 반복 실행돼야 한다. 실행 결과를 보면 같은 숫자가 0부터 15까지 16번 출력되는 것을 볼 수 있다.

make check 수행 결과)

* 