**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

이름 / 학번 : 이나연 / 20171662

개발 기간 : 11/25 ~ 12/8

1. **개발 목표**

이번 프로젝트에서 구현할 내용은 Alarm Clock, Priority Scheduling, Advanced Scheduler (BSD Scheduler)이다. 기존의 pintos에선 thread가 단순 busy-waiting 기반으로 대기를 하였다면, 이번 프로젝트에선 해당 부분을 Alarm-clock 방법으로 구현한다. 또한 기존의 pintos에선 RR scheduling 방식을 사용하여 thread를 실행하였었는데, 이 부분을 thread의 priority를 고려하는 방법으로 바꾸어 구현한다. 추가적으로 BSD scheduler를 사용하여 thread간의 balance를 조절할 수 있도록 해준다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

기존의 pintos에서, timer\_sleep()함수는 busy-waiting 기법을 사용하여 CPU의 자원을 낭비하였다. 이를 방지하기 위해 해당 부분을 context switching 방식과 결합하여 구현한다. ‘tick’의 값과 비교를 계속 하며 tick만큼의 시간이 가지 않았다면 thread를 block하고, 이런 상태로 변환된 thread들을 관리하기 위해 새로운 list를 만든다. 해당 list에 저장될 때, 해당 thread의 wake up time역시 저장해둔 후 해당 시간이 되었을 때 thread를 다시 ready queue에 삽입한다.

* 1. Priority Scheduling

기존의 pintos에선 새로운 thread가 들어오면 단순히, 우선순위 등의 아무런 고려 없이 ready list의 맨 뒤에 삽입된다. priority scheduling을 구현하면 각 thread의 우선순위에 따라 scheduling이 우선순위에 의해 수행되도록 바뀔 수 있다. 새로운 thread가 들어왔을 때 현재 수행되고 있는 thread보다 priority가 더 높다면, 그 즉시 CPU를 새로운 thread에게 yield해주도록 한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

기존의 pintos에선 scheduling을 FIFO의 방식으로 한다. 이를 BSD scheduler를 통해 각 thread의 다른 scheduling 요구를 balance있게 scheduling하는 scheduler로 구현한다. schedule()이 실행되면, 제일 높은 우선순위의 priority queue에 있던 thread부터 실행이 된다. 각 priority queue들은 RR scheduling에 기반하여 동작한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

timer\_sleep()에서 while문을 반복적으로 수행하며 시간을 체크하는 방법 대신 alarm clock 기법으로 시간을 체크 후 설정된 시간이 아직 되지 않았다면 thread를 blocked 상태로 변환하여 sleep list에 저장한다. 이 때, 해당 thread가 언제 깨야 하는지에 대한 정보를 wakeup\_tick으로 저장한다. 매 tick마다 timer\_interrupt()가 호출되면, 해당 시간에 깨어나야 할 thread가 있는지 확인하고, 있다면 해당 thread를 다시 ready list에 삽입한다.

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

만약 ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 기존의 running thread가 preempt되고 새로운 thread로 yield하게끔 한다. 여기서 priority가 높은 순서대로 ready list에 정렬하기 위해 thread가 들어올 때 ready list에 추가될 시점에 단순히 기존의 방법인 list\_push\_back()함수로 맨 뒤에 삽입하는 대신 그 때마다 priority를 check하여 list\_insert\_ordered()함수를 사용한다.

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

초기의 priority는 thread\_create()에서 정해지지만, 4 tick마다 모든 thread의 priority는 recalculate되어야 한다. 해당 priority의 계산식은 다음과 같다.

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

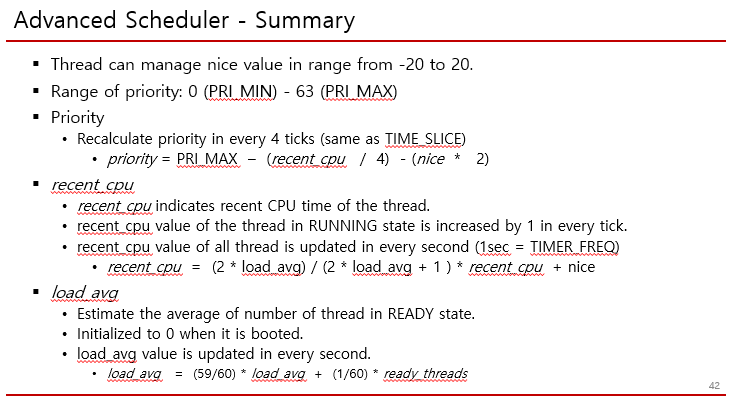
이 때, recent\_cpu는 최근에 사용된 CPU time의 추정값이고, nice는 thread의 nice value이다. PRI\_MAX는 maximum priority를 의미하며, 63의 값을 가진다. minimum priority는 0의 값을 가진다. 그러나 여기서 recent\_cpu값 역시 매초 recalculate된다. 해당 값의 계산 식은 다음과 같다.

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

위 식에서 사용되는 load\_avg는 ready state인 thread의 수의 평균값이다. 그러나 이 값 역시 매 초 update된다.

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

load\_avg의 초기값은 0으로 설정되고, 이는 system내에서 전역으로 쓰인다. 위 식에서 ready\_threads는 ready 또는 running state에 있는 thread의 수이다.



1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

11/25~11/29: 알고리즘 구상, alarm clock 구현

11/30~12/5: priority scheduling, bsd scheduler 구현

12/6~12/8: 보고서 작성 및 프로그램 보완

* 1. **개발 방법**

1. Alarm Clock

<devices/timer.c>

timer\_sleep(): busy waiting 부분을 alarm clock역할을 하는 thread\_sleep()함수로 대체함

timer\_interrupt(): 매 tick마다 thread\_wake\_up()함수를 호출하여 현재 block되어 있는 thread들 중 깨어나야 하는 thread가 있는지 확인후 있다면 깨워서 ready\_list에 삽입

<threads/thread.h>

struct thread내의 wakeup\_tick 멤버 추가: 각 thread의 wakeup time을 저장하기 위하여 추가하였다.

<threads/thread.c>

thread\_sleep(): 인자로 전달받은 해당 thread가 깨어나야 할 시간을 해당 thread내에 저장하고 sleep list에 push하고 block한다.

thread\_wake\_up(): 매 tick마다 수행된다. 해당 시점에 sleep list안의 thread들을 확인하며 wake up time이 된 thread들은 unblock을 시켜주며 ready list에 다시 삽입될 수 있게끔 한다.

thread\_init(): block된 thread들을 저장하기 위한 list인 sleep list를 초기화해주는 부분을 추가한다.

2. Priority Scheduling

<threads/thread.c>

thread\_tick(): thread\_prior\_aging이 true라면 thread\_aging()함수를 호출하여 priority aging 기능을 구현하는 부분을 추가한다.

thread\_create(): 만약 새로 들어온 thread의 priority가 running thread의 priority보다 높다면 running thread에서 thread\_yield()를 실행하게끔 하는 부분을 추가한다.

thread\_unblock(): 기존의 ready list에 priority를 고려하지 않고 맨 뒤에 push하는 부분을 priority를 고려한 list\_insert\_ordered()함수를 사용하여 ready list에 thread를 삽입하는 부분으로 대체한다.

thread\_yield(): 기존의 ready list에 priority를 고려하지 않고 맨 뒤에 push하는 부분을 priority를 고려한 list\_insert\_ordered()함수를 사용하여 ready list에 thread를 삽입하는 부분으로 대체한다.

thread\_set\_priority(): running thread와 ready list의 제일 앞에 있는 thread의 priority를 서로 비교하여 ready list에 있던 thread의 priority가 더 높으면 그 즉시 preempt하여 running thread에서 thread\_yield()를 호출하는 부분을 추가한다.

thread\_aging(): 호출될 때마다 ready list에 있는 thread의 priority를 1씩 증가시킨다.

compare\_priority(): 두 thread의 priority를 비교하여 결과를 반환하는 함수를 추가한다.

<threads/synch.c>

sema\_down(): 기존에 sema->waiters 리스트에 현재 thread가 단순히 맨 뒤에 push되던 부분을 compare\_priority의 결과값에 따라 priority를 비교하여 list\_insert\_ordered()함수를 사용하여 waiters list에 현재 thread를 삽입하는 부분으로 대체한다.

sema\_up(): 우선 thread\_yield()를 하여 priority를 체크하게 하는 부분을 추가한다.

3. BSD Scheduler

<threads/thread.h>

typedef int32\_t fixpoint: int형과 차이를 두기 위하여 헷갈리지 않도록 fixpoint라는 자료형을 생성하였다. real number인 recent\_cpu와 load\_avg와 floating point arithmetic을 지원하지 않는 pintos를 위하여 생성하였다.

struct thread내의 recent\_cpu, nice: 최근에 thread에 주어진 CPU time과 niceness를 저장하기 위하여 추가하였다.

<threads/thread.c>

thread\_init(): 전역변수로 선언된 load\_avg를 0으로 초기화한다. 초기 thread의 nice, recent\_cpu값도 0으로 초기화한다.

thread\_tick(): thread\_mlfqs가 true일 때도 priority aging기법이 실행될 수 있도록 하는 부분을 추가한다.

thread\_set\_priority(): thread\_mlfqs가 true일 땐 thread\_set\_priority()를 수행하지 않게끔 하는 부분을 추가한다.

thread\_set\_nice(): 현재 thread의 nice value를 parameter로 주어진 nice value로 바꾼다. 바뀐 nice값에 따라 priority도 업데이트한 후, ready list의 제일 앞 thread와 현재 thread의 priority를 비교하여 ready list에 있던 thread의 priority가 더 크면 thread\_yield()를 수행하게끔 한다.

thread\_get\_nice(): 현재 thread의 nice value를 반환하게끔 한다.\

thread\_get\_load\_avg(): system load average의 100배를 반환한다. 이 때 floating point를 int형으로 변환하여 반환한다.

thread\_get\_recent\_cpu(): 현재 thread의 recent\_cpu value의 100배를 int형으로 변환하여 반환한다.

init\_thread(): 현재 running thread의 recent\_cpu값과 nice값을 물려받아 저장하는 부분을 추가한다.

thread\_aging(): 매초마다 recent\_cpu value를 업데이트하고 매 4 tick마다 priority를 다시 계산하여 업데이트해주는 부분을 추가한다.

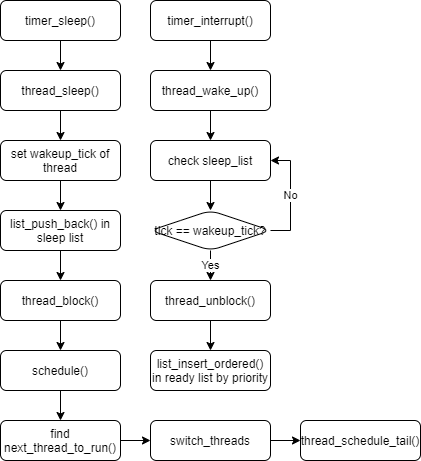
priority\_update(): recent\_cpu value를 int형으로 변환하여 priority를 다시 계산하여 현재 thread의 priority를 업데이트 해주는 함수를 추가한다.

recent\_cpu\_update(): 인자로 전달받은 recent\_cpu값에 추가적으로 nice를 더한 값을 recent\_cpu값으로 업데이트 해주는 함수를 추가한다.

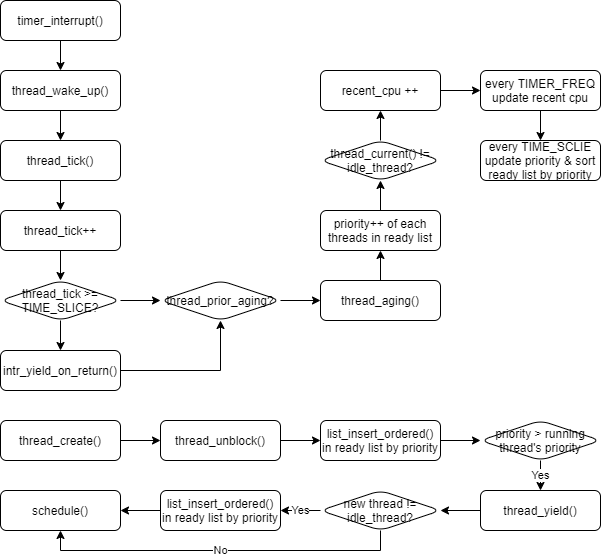
1. **연구 결과**
   1. **Flow Chart**

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

1. Alarm Clock



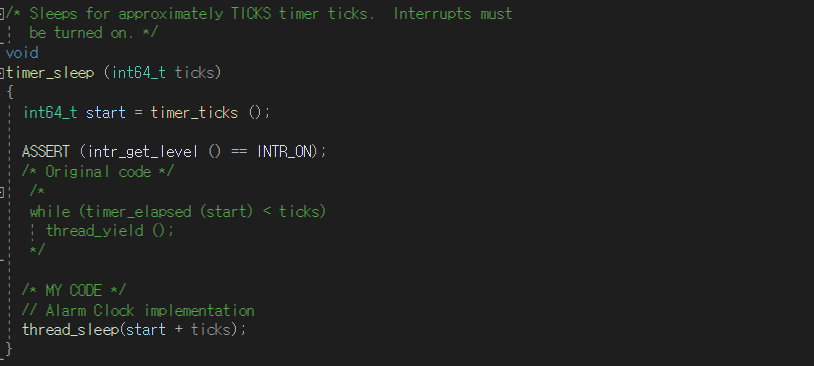
2. Priority Scheduling



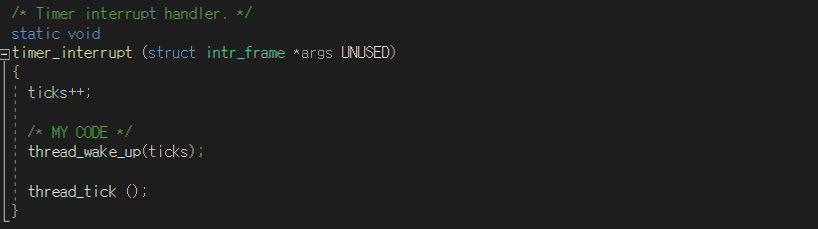
* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

1. Alarm Clock

<devices/timer.c>

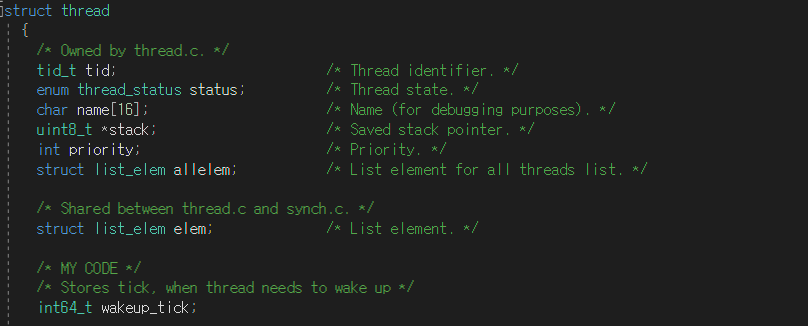


timer\_sleep(): busy waiting 부분을 alarm clock역할을 하는 thread\_sleep()함수로 대체함. thread\_sleep() 함수의 인자로는 해당 thread의 wake up tick의 값인 start+ticks를 넘겨주었다.



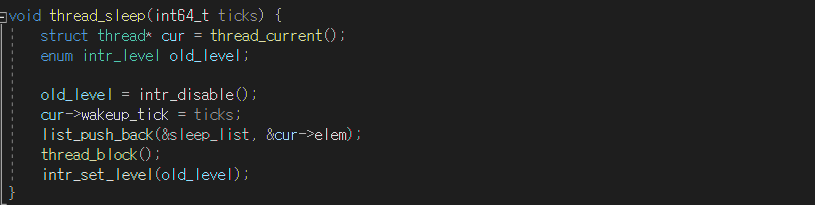
timer\_interrupt(): 매 tick마다 thread\_wake\_up()함수를 호출하여 현재 block되어 있는 thread들 중 깨어나야 하는 thread가 있는지 확인후 있다면 깨워서 ready\_list에 삽입하였다.

<threads/thread.h>

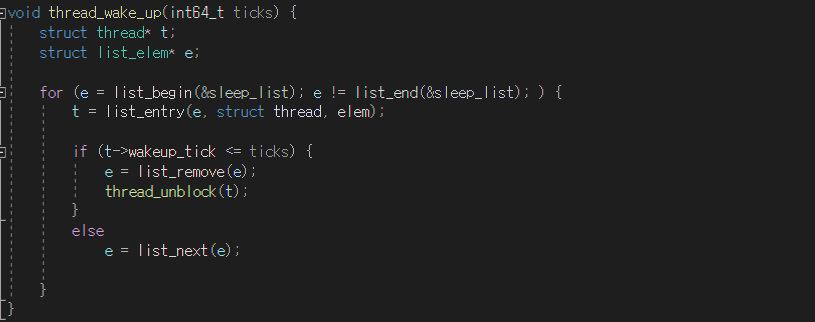


struct thread내의 wakeup\_tick 멤버 추가: 각 thread의 wakeup time을 저장하기 위하여 추가하였다.

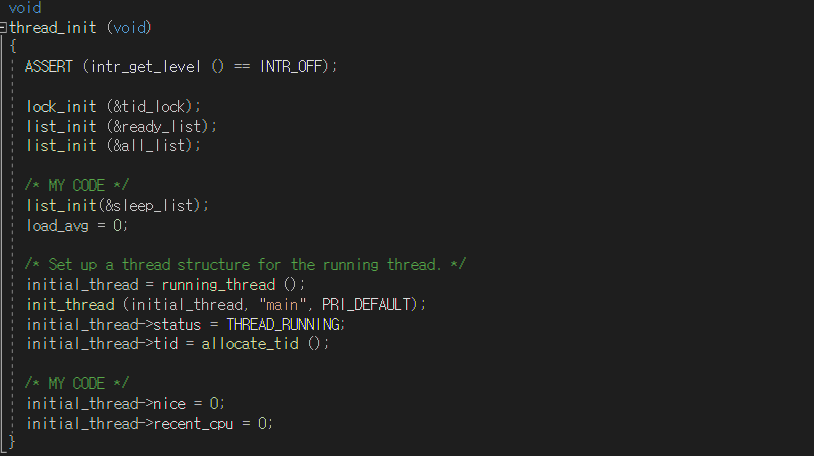
<threads/thread.c>



thread\_sleep(): 인자로 전달받은 해당 thread가 깨어나야 할 시간을 해당 thread내에 저장하고 sleep list에 list\_push\_back()함수를 이용해 맨 뒤에 push하고 state를 block으로 바꾸게끔 thread\_block()함수를 호출하였다. 해당 쓰레드는 wake\_up함수에 의해 unblock되기 전까지 sleep의 상태가 유지된다.



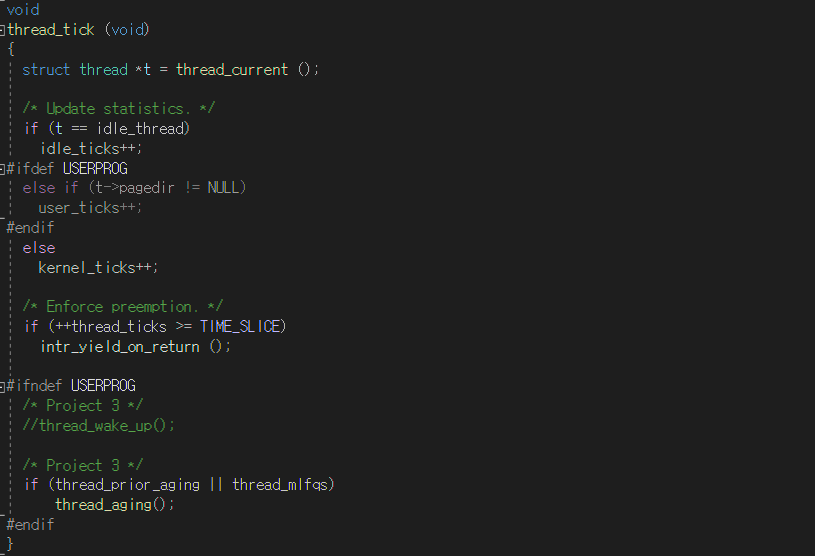
thread\_wake\_up(): 매 tick마다 수행된다. 해당 시점에 sleep list안의 thread들을 list\_next로 확인하며 현재 tick과 thread내에 저장되어 있던 wakeup\_tick의 값을 비교하여 현재 tick이 크거나 같다면 wake up time이 되었다는 뜻이므로, 해당 thread들은 unblock을 시켜주며 ready list에 다시 삽입될 수 있게끔 한다. 아니라면 sleeplist내의 다음 thread를 확인한다. 해당 작업을 sleep list의 끝까지 반복한다.



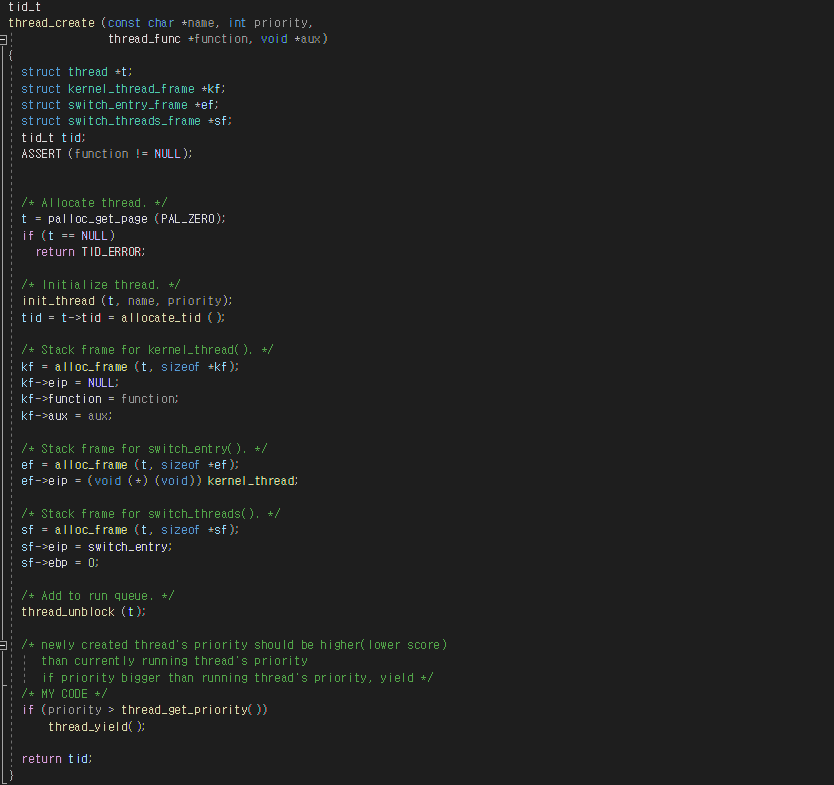
thread\_init(): block된 thread들을 저장하기 위한 list인 sleep list를 초기화해주는 부분을 추가한다.

2. Priority Scheduling

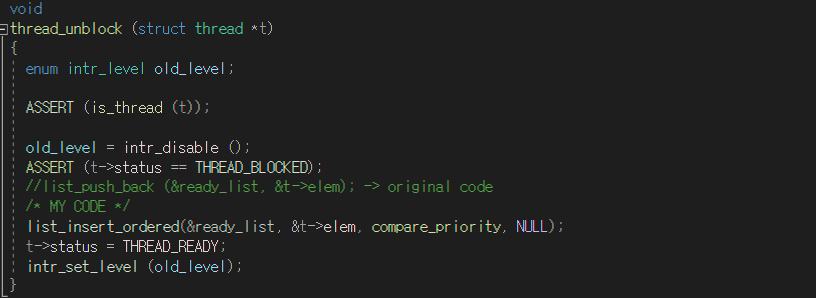
<threads/thread.c>



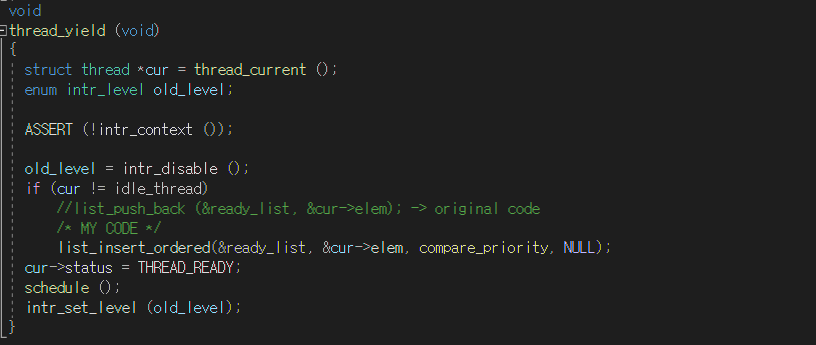
thread\_tick(): thread\_prior\_aging이 true라면 thread\_aging()함수를 호출하여 priority aging 기능을 구현하는 부분을 추가한다.



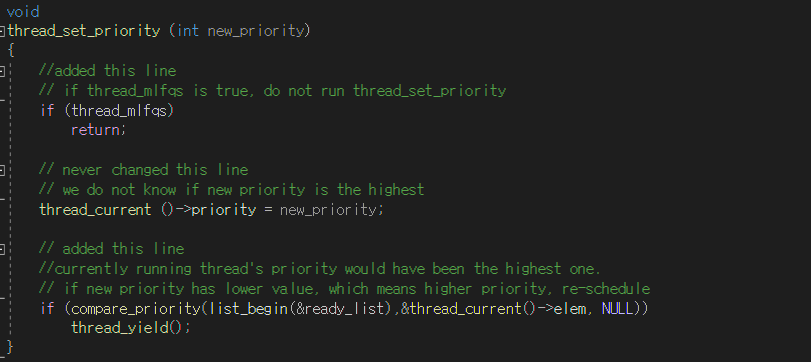
thread\_create(): 만약 새로 들어온 thread의 priority가 running thread의 priority보다 높다면 running thread에서 thread\_yield()를 실행하게끔 하는 부분을 추가한다.



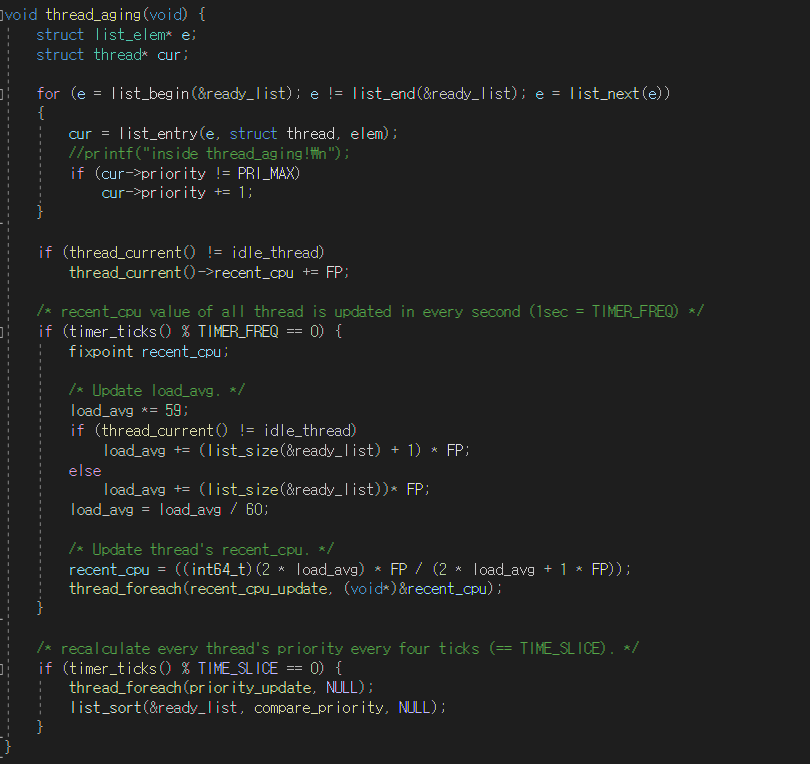
thread\_unblock(): 기존의 ready list에 priority를 고려하지 않고 맨 뒤에 push하는 부분을 priority를 고려한 list\_insert\_ordered()함수를 사용하여 ready list에 thread를 삽입하는 부분으로 대체한다.



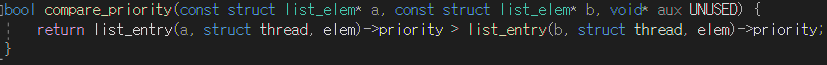
thread\_yield(): 기존의 ready list에 priority를 고려하지 않고 맨 뒤에 push하는 부분을 priority를 고려한 list\_insert\_ordered()함수를 사용하여 ready list에 thread를 삽입하는 부분으로 대체한다.



thread\_set\_priority(): running thread와 ready list의 제일 앞에 있는 thread의 priority를 서로 비교하여 ready list에 있던 thread의 priority가 더 높으면 그 즉시 preempt하여 running thread에서 thread\_yield()를 호출하는 부분을 추가한다.

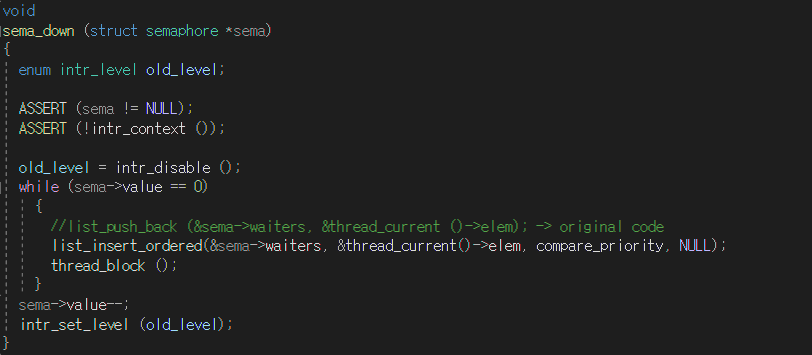


thread\_aging(): 호출될 때마다 ready list에 있는 thread들을 돌며 각 thread의 priority를 1씩 증가시킨다. 이 때 priority가 maximum값인 PRI\_MAX에 도달했는지를 확인한 후, 도달했다면 증가시키지 않는다.

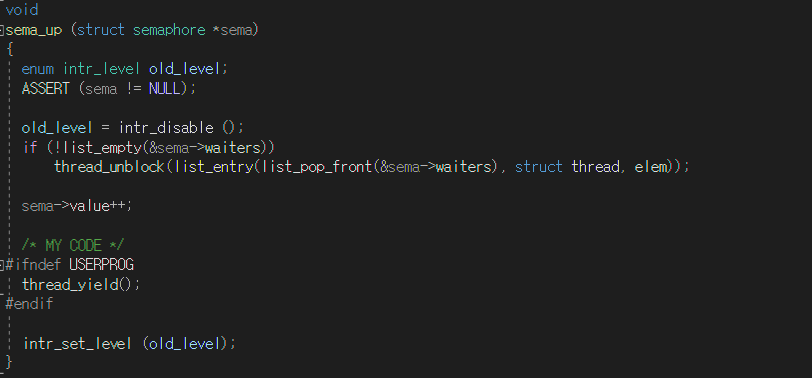


compare\_priority(): 두 thread의 priority를 비교하여 첫번째 thread의 priority가 더 크면 true, 두번째 thread의 priority가 더 크면 false를 return 한다.

<threads/synch.c>



sema\_down(): 기존에 sema->waiters 리스트에 현재 thread가 단순히 맨 뒤에 push되던 부분을 compare\_priority의 결과값에 따라 priority를 비교하여 list\_insert\_ordered()함수를 사용하여 waiters list에 priority순으로 현재 thread를 삽입하는 부분으로 대체한다.



sema\_up(): 우선 thread\_yield()를 하여 priority를 체크하게 하는 부분을 추가한다. 앞에서 thread\_unblock으로 waiting하던 thread를 다시 데려왔지만, priority를 확인하여 지금 run시킬지 아니면 ready list에 있어야 할 지 확인해야 하기 때문이다.

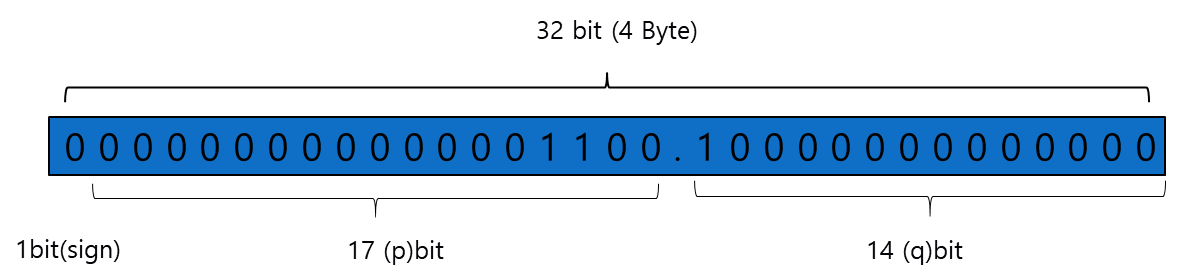
3. BSD Scheduler

<threads/thread.h>

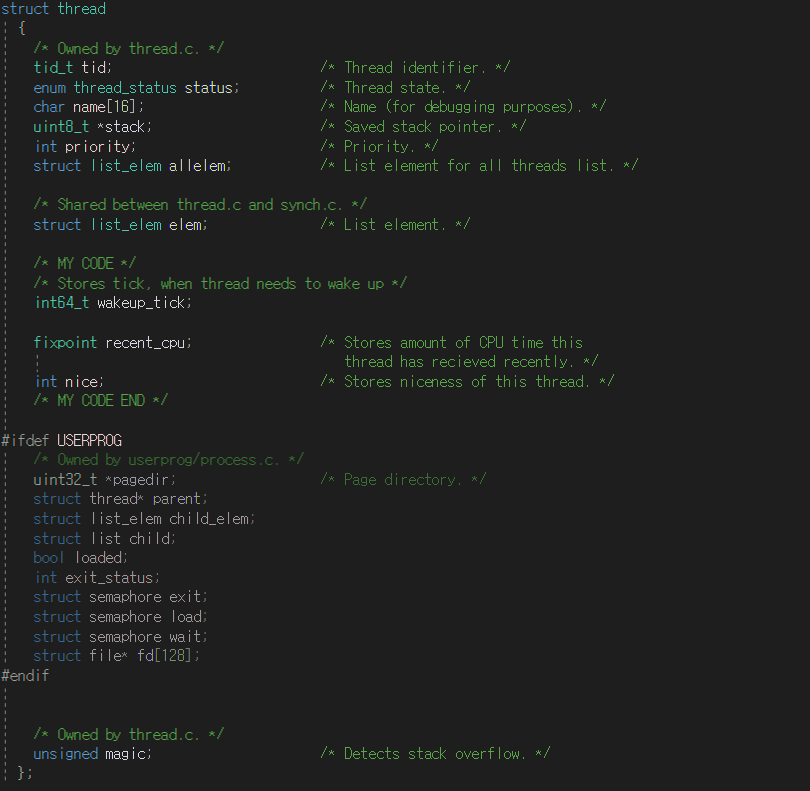


typedef int32\_t fixpoint, FP: int형과 차이를 두기 위하여 헷갈리지 않도록 fixpoint라는 자료형을 생성하였다. real number인 recent\_cpu와 load\_avg와 floating point arithmetic을 지원하지 않는 pintos를 위하여 생성하였다.

pintos 커널은 floating point arithmetic을 지원하지 않지만, recent\_cpu와 load\_avg는 real number들인데 해당 값들이 pintos내에서 쓰여야 한다. 따라서 해당 값들을 int형으로 바꾸어 계산에 포함하여야 한다. 이럴 때 해당 real number값들을 int형으로 바꾸는 과정이 BSD scheduler 관련 코드를 짤 때 매우 많이 나올 것이다. 해당 부분을 간소화하는 변수, 상수가 필요하다고 판단하였다.

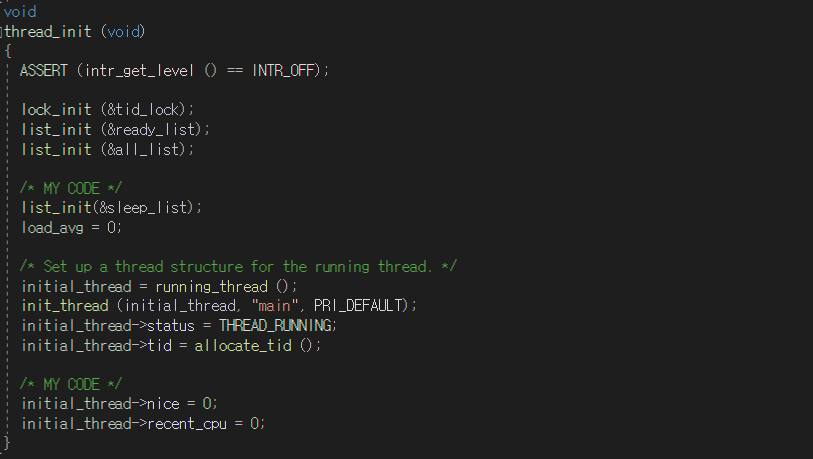


q bit쪽을 한 번에 표현하기 위해 FP라는 수를 define하였다. 해당 상수를 통해 q bit쪽을 아예 생각하지 않고 바로 p bit를 int형으로 표현할 수 있게끔 하는 식을 간소화할 수 있다.

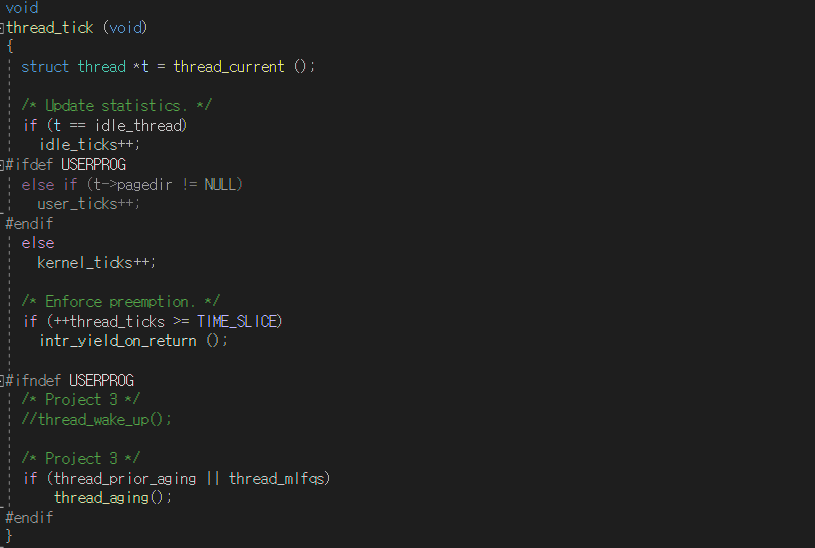


struct thread내의 recent\_cpu, nice: 최근에 thread에 주어진 CPU time과 niceness를 저장하기 위하여 추가하였다.

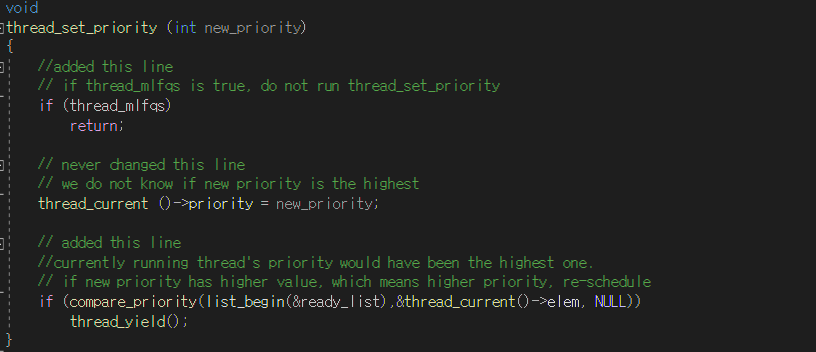
<threads/thread.c>



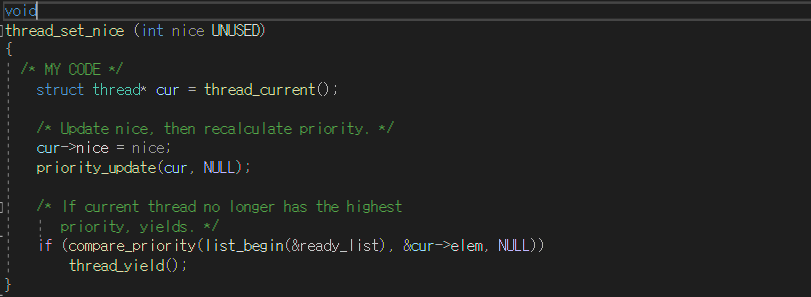
thread\_init(): 전역변수로 선언된 load\_avg를 0으로 초기화한다. 초기 thread의 nice, recent\_cpu값도 0으로 초기화한다.



thread\_tick(): thread\_mlfqs가 true일 때도 priority aging기법이 실행될 수 있도록 하는 부분을 추가한다. thread\_aging()함수 내에서 aging과 함께 recent\_cpu, load\_avg를 update하기 때문이다.



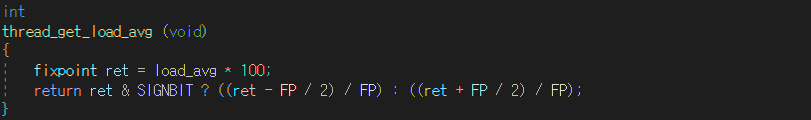
thread\_set\_priority(): thread\_mlfqs가 true일 땐 thread\_set\_priority()를 수행하지 않게끔 한다.



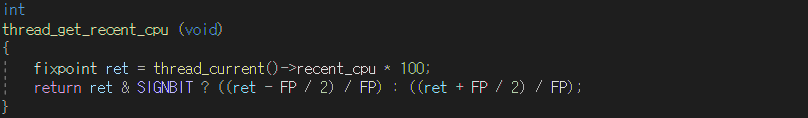
thread\_set\_nice(): 현재 thread의 nice value를 parameter로 주어진 nice value로 바꾼다. 바뀐 nice값에 따라 priority도 업데이트한 후, ready list의 제일 앞 thread와 현재 thread의 priority를 비교하여 ready list에 있던 thread의 priority가 더 크면 thread\_yield()를 수행하게끔 한다.



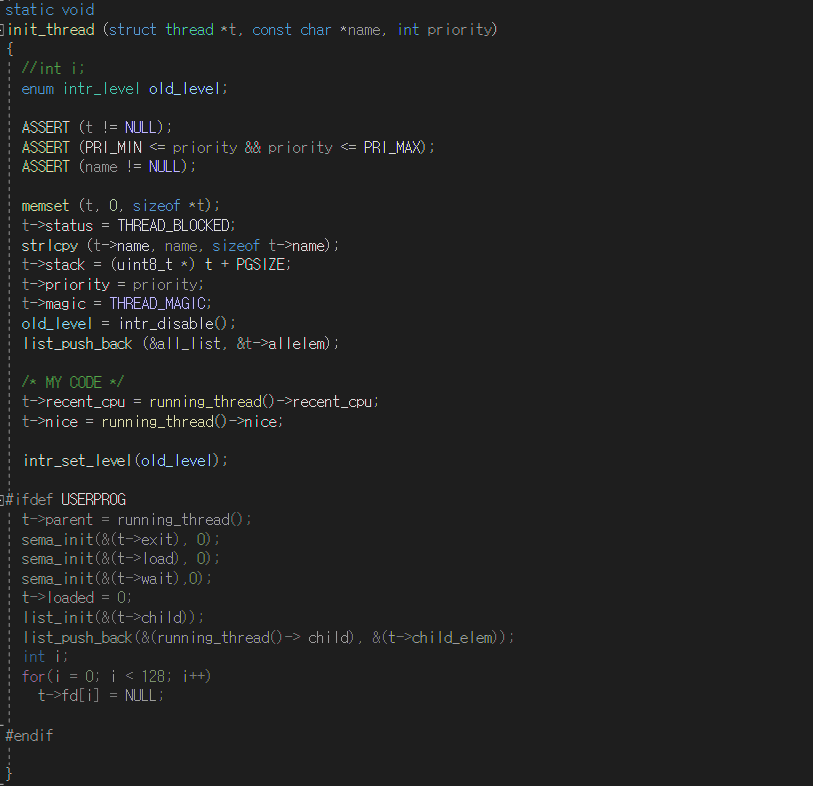
thread\_get\_nice(): 현재 thread의 nice value를 반환한다.



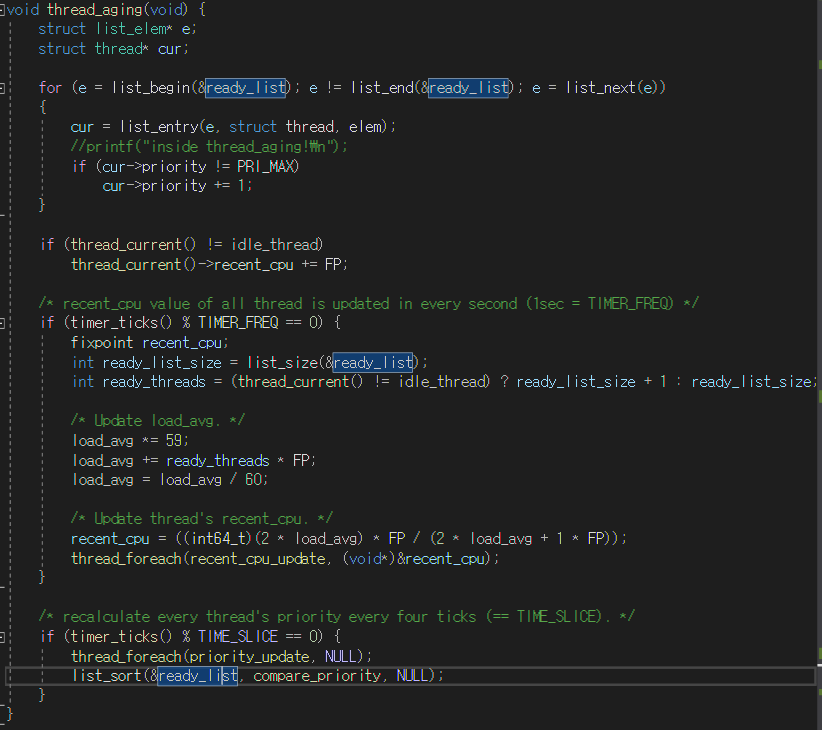
thread\_get\_load\_avg(): system load average의 100배를 반환한다. 이 때 floating point를 int형으로 변환하여 반환한다. fixpoint 형 load\_avg에 100배를 한 다음, 해당 수의 signbit를 고려하여 signbit가 1이라면 -0.5를 하여 반올림하고, 0이라면 +0.5를 하여 반올림하게끔 하였다.



thread\_get\_recent\_cpu(): 현재 thread의 recent\_cpu value의 100배를 int형으로 변환하여 반환한다. fixpoint형 recent\_cpu에 100배를 한 다음, 해당 수의 signbit를 고려하여 signbit가 1이라면 -0.5를 하여 반올림하고, 0이라면 +0.5를 하여 반올림하게끔 하였다.



init\_thread(): 현재 running thread의 recent\_cpu값과 nice값을 물려받아 저장하는 부분을 추가한다.



thread\_aging(): 매초마다 recent\_cpu value를 업데이트하고 매 4 tick마다 priority를 다시 계산하여 업데이트해주는 부분을 추가한다.

우선 현재 thread가 running state라면 매 tick마다 recent cpu값을 1 증가시켜준다. 현재 recent\_cpu, load\_avg 모두 fixpoint형이기 때문에 1을 더해준다는 것은 사실 14번째 bit를 1 증가시켜주어야 하는 경우이므로 1 << 14가 저장되어 있는 상수값 FP를 더하는 것으로 recent\_cpu의 1증가를 표현하였다.

모든 thread의 recent\_cpu 역시 매초 (every TIMER\_FREQ) 업데이트가 되어야 한다. 해당 값의 계산 식은 다음과 같다.

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

위 식에서 사용되는 load\_avg는 ready state인 thread의 수의 평균값이다. 그러나 이 값 역시 매 초 update된다.

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

load\_avg의 초기값은 0으로 설정되고, 이는 system내에서 전역으로 쓰인다. 위 식에서 ready\_threads는 ready 또는 running state에 있는 thread의 수이다.

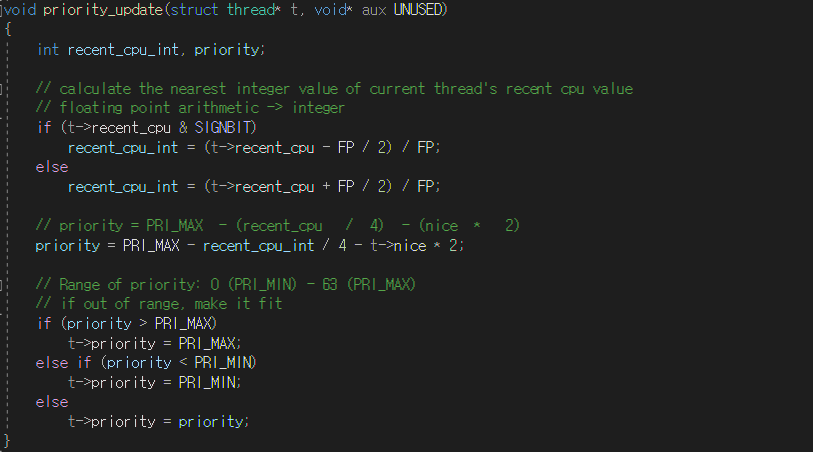
따라서 우선 현재 tick가 TIMER\_FREQ로 나누었을 때 0이 되는지를 확인한 후, 맞다면 TIMER\_FREQ의 단위에 있다는 뜻이므로 recent\_cpu와 load\_avg를 recalculate한다. recent\_cpu를 계산하려면 우선 load\_avg의 값이 먼저 계산되어야 하므로 위에서 언급된 식에 의하여 load\_avg의 값을 계산한다. 이 때 ready\_threads는 ready 또는 running state에 있는 thread의 수여야 하므로 우선 ready\_list의 크기가 곧 ready state에 있는 thread의 수이므로 그 값에다 만약 현재 thread가 running state라면 running state thread의 수 1을 더해서 ready\_threads의 값으로 넣고, 아니라면 그냥 ready\_list의 크기만큼을 ready\_threads로 설정한다. 분자를 다 계산하였다면 60으로 최종적으로 나누어주면 된다.

recent\_cpu의 값 역시 위의 식대로 계산한다. 다만 nice값은 각 thread마다 값이 다 다르기 때문에 thread\_foreach()함수로 존재하는 모든 thread(all\_list에 저장)에 대해서 recent\_cpu\_update함수를 수행해주게끔 하였다. 인자로는 새로 계산한 recent\_cpu의 값을 전달하였다.

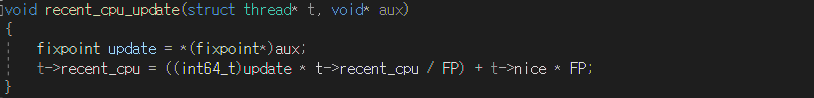
또한 TIME\_SLICE, 즉 4 tick마다 priority를 update해주어야 한다. 계산식은 아래와 같다.

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

해당 계산 역시 모든 thread에 대하여 계산을 해야 하므로, thread\_foreach()함수로 존재하는 모든 thread에 대하여 prority\_update를 수행해주게끔 하였다. recent\_cpu와 nice값 모두 각 thread안에 저장되어 있으므로 따로 전달하는 인자는 없게끔 하였다. 그렇게 모든 thread의 priority를 update한 후 priority에 대하여 다시 ready list에 있는 thread를 sort하였다.



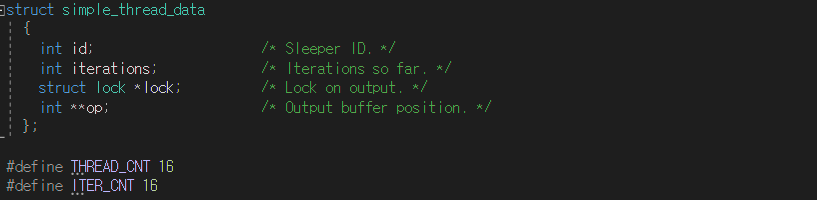
priority\_update(): recent\_cpu value를 int형으로 변환하여 priority를 다시 계산하여 현재 thread의 priority를 업데이트 해주는 함수를 추가한다. 우선 recent\_cpu은 fixpoint형 변수이기 때문에 int형인 priority를 계산하려면 int형으로 변환한 후 계산을 수행해주어야 한다. recnt\_cput의 signbit가 1이라면 -0.5를 한 후 반올림을 수행하고, 0이라면 +0.5를 한 후 반올림을 수행한다. 그런 다음 FP로 나누어 정수 부분만 남기게끔 하고 recent\_cpu\_int에 저장하였다. priority = PRI\_MAX – (recent\_cpu / 4) – (nice \* 2)의 식을 사용하여 recalculation을 수행한다. 계산된 priority가 PRI\_MAX또는 PRI\_MIN 범위 밖에 있을 수 있기 때문에 각 경우에 대하여 PRI\_MAX를 넘었다면 PRI\_MAX로 설정해주고, PRI\_MIN 보다 작다면 PRI\_MIN으로 설정해주었다.

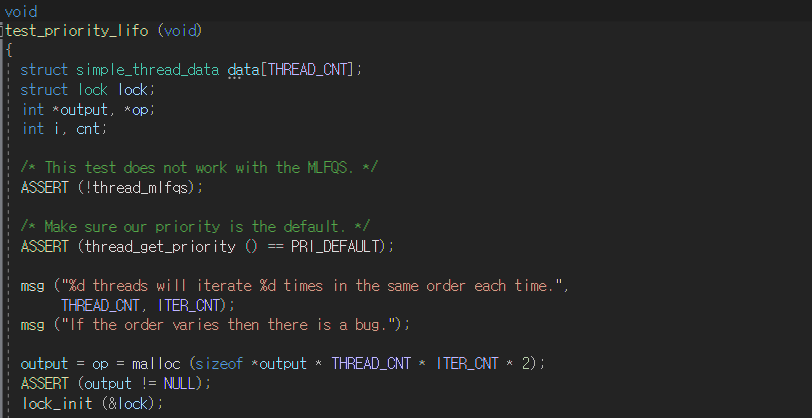


recent\_cpu\_update(): 인자로 전달받은 recent\_cpu값에 추가적으로 nice를 더한 값을 recent\_cpu값으로 업데이트 해주는 함수를 추가한다. t->nice가 int형이기 때문에 recent\_cpu의 값을 FP로 나누어 정수부분만 남겨 연산을 수행한 후 다시 FP를 곱하여 fixpoint형으로 다시 돌아가게끔 하였다.

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

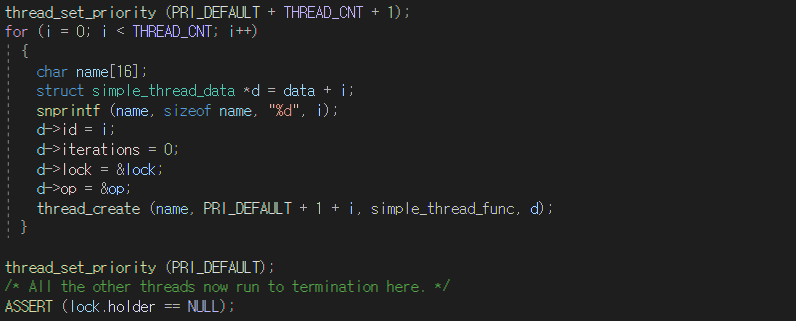
priority-lifo.c의 코드는 다음과 같다.





우선 초기화하는 부분을 보면 sleep thread의 id, 현재까지의 iteration 수, output에서의 lock, output buffer의 position을 저장하는 simple\_thread\_data라는 구조체를 16개를 담고 있는 data라는 list 및 아래에 쓰일 변수들이 선언되어 있다. 우선 해당 test에선 thread\_mlfqs를 사용하지 않게끔 설정하고 각 thread들의 priority가 default priority로 설정되어 있는지 확인한다.

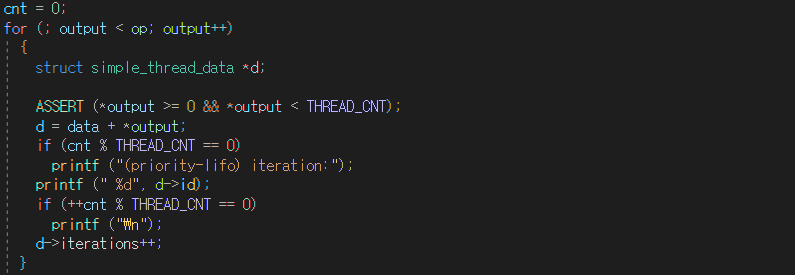
해당 함수에선 16개의 thread들이 각각 16번 iterate할 것이라고 말해주는 메시지를 출력한다. 그런 다음 output을 저장할 int\*형 변수들에 16\*16\*2의 int 자리를 할당해준 다음 lock을 초기화한다. 현재 상태론 output과 op의 주소값이 동일한 상태인것이다.

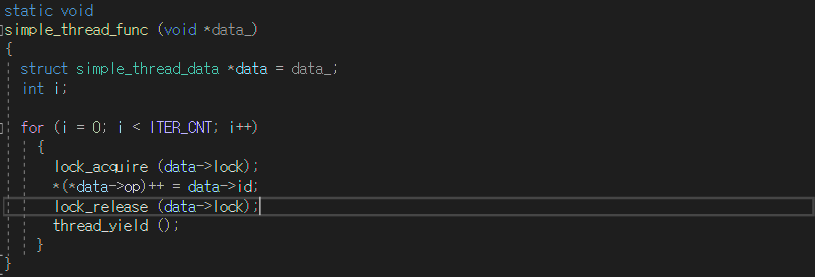


우선 main thread (지금 running thread)의 priority를 default+16+1로 설정한다. 이후 생겨날 다른 thread들은 해당 thread의 priority보다 작을 것이기 때문에 main thread 후 다른 thread들이 실행될 것이다.

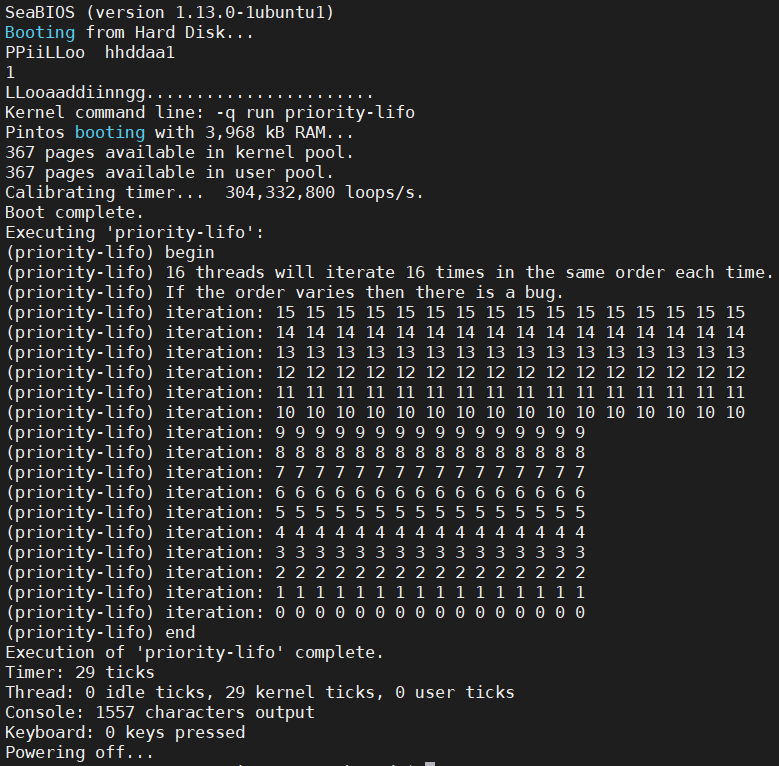
앞서 선언한 simple\_thread\_data의 배열 변수 data의 각 index에 새로운simple\_thread\_data를 저장한다. name이라는 char배열 변수에 현재 i를 저장한다. 0~15의 값이 각 iteration에서 저장될 것이다. 즉 16개의 simple\_thread\_data 변수들로 이루어질 thread들의 이름은 각각 0~15가 될 것이다. 각 simple\_thread\_data의 id 역시 i로 설정하고, iteration은 우선 0으로 초기화하고, lock에는 앞서 선언하였던 lock의 주솟값을 저장하고, 마지막에 thread\_create()로 각 simple\_thread\_data에 대응하는 thread들을 생성한다. thread\_create에선 생성할 thread의 이름, priority를 전달받는데, 전달할 때 name은 앞서 설정한 name, priority는 default priority + 1 + i로 한다. 즉 배열의 인덱스가 증가함에 따라 priority도 증가하게끔 설정한다. thread\_create()에선 우선 생성할 thread를 thread\_unblock()으로 run queue에 add한다. thread\_unblock()에선 ready list에 thread를 삽입할 때에도 priority순으로 삽입한다. 따라서 thread들이 create될 때마다 ready list엔 순차적으로 priority가 큰 thread부터 저장될 것이다. 다시 thread\_create()로 돌아와 만약 현재 running thread와 전달받은 priority를 비교했을 때 전달받은 priority가 더 크다면 새로운 thread로 run을 바꿔주기 위해 thread\_yield()를 호출한다. 그러나 지금까지는 running thread의 priority가 더 크기 때문에 thread\_yield()가 아직까지는 일어나지 않는다.

그렇게 thread들의 creation이 완료되면 현재 running thread의 priority를 default priority로 다시 바꾸어 방금 생성한 thread들보다 priority가 작게 한다. 그런 다음 lock.holder를 NULL로 ASSERT하여 이제 다른 thread들이 그 지점부터 run하게 된다. run하는 순서는 ready list안에 있던 순서대로 먼저 나온 다음 각 thread에 대하여 또 priority를 내부에서 계속 check할 것이다.





thread\_create()를 할 때 simple\_thread\_func가 thread\_func자리로 들어갔었다. 이는 kernel\_thread\_frame의 function자리에 저장되는데, kernel\_thread()함수 내에서 function(aux)가 수행되게 된다. 따라서 ready list에 각 id를 가지는 thread가 순서대로 있었는데, 각 thread안에서 simple\_thread\_func가 수행된다. simple\_thread\_func내에선 ITER\_CNT, 즉 16번 iteration이 수행되는데, 각 iteration내에서 우선 op를 증가시킨다. 처음에 d->op에 op의 주솟값을 넣었었기 때문에 이 함수 내에서 data->op의 값을 증가시킨다는 것은 원래의 test\_priority\_lifo내의 op에도 영향을 끼친다. 각 thread내에서 op를 16번 증가시키고 그 thread가 16개 있었기 때문에 op는 16\*16만큼 증가했을 것이다. 그런 다음 각 iteration에서 thread\_yield()를 수행하는 것을 알 수 있다. thread\_yield()함수에선 idle thread가 아니라면 priority에 의하여 ready list에 삽입한 다음 schedule()을 수행한다. 따라서 각 id를 가지는 thread가 다시 16번의 iteration에 의하여 16개 ready list에 priority에 따라 맞는 자리에 삽입될 것이다. 그렇게 ready list에 thread를 넣은 다음 thread들이 run되기 시작한다. thread들이 run할 때 cnt=0에서 시작하지만 thread들이 run하기 시작했기 때문에 cnt는 안의 과정에 의해서 계속 증가할 것이다. ready list에는 simple\_thread\_func에 의해 각 0~15의 값을 id로 가지는 thread들이 ITER\_CNT개 만큼 다 있을 것이기 때문에 printf (" %d", d->id); 부분이 ITER\_CNT번 출력될 것이다. 그것도 priority는 id에 비례했던 것이 앞의 코드에서 이미 설명되었기 때문에 id가 15인 thread 16개, 14인 thead 16개, … , id가 0인 thread 16개가 위 과정을 거칠 것이다. 그렇게 프로그램이 다 실행되고 나면 아래와 같은 결과값이 출력된다.



* make check 수행 결과를 캡처하여 첨부

