**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

조 / 조원 : 20211584 장준영

개발 기간 : 2023-11-8 ~ 2023-11-18

1. **개발 목표**

이번 프로젝트에서는 OS가 여러 프로세스를 제어하기 위해 사용하는 scheduling algorithm과 관련된 기능을 구현한다. 구현 이전 pintOS의 scheduler는 기본적인 RR scheduler로, 프로세스 간의 priority를 고려하지 않고 모두 공평하게 같은 time quantum을 부여하는 방법이다. 이를 priority를 도입하여 개선하고, starvation 등의 문제를 해결하기 위해 priority aging을 도입한다. 도입된 priority aging과 함께 MLFQ를 구현하여 최적의 scheduler를 pintOS에 탑재한다. 또, 코드 내에서 Sleep과 같은 system block command가 실행되었을 때, 그 프로세스를 정해진 시간 만큼 멈추고 다시 깨워주는 alarm clock의 개선 또한 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
   2. **Alarm Clock**

자고 있는 프로세스를 깨우는 alarm clock은 기존 pintOS 내에서 busy waiting 방식으로 구현되어 있다. 기존의 RR scheduler에 따라 자고 있는 프로세스에게 CPU 제어권을 주고, 만약 그 프로세스가 아직 깨울 시간이 되지 않았다면 다른 프로세스에게 yield하고 다시 ready queue로 돌아가는 방식이다. 하지만 이 방식은 상당히 비효율적인데, 프로세스 전환(context switch) 시의 overhead가 두 번이나 발생하고, 확인 과정에서 CPU의 clock tick을 사용하기 때문이다. 이를 개선하기 위해, 자고 있는 프로세스는 CPU의 권한을 얻을 수 없고, 특정 시간이 지나간 이후에만 다시 얻을 수 있도록 고쳐야 한다.

* 1. **Priority Scheduling**

앞서 언급한 대로 기존의 pintOS scheduler는 naïve FIFO RR scheduler로, 다른 요소는 고려하지 않고 ready queue에 들어온 순서대로 일정한 time quantum을 부여한다. 이 경우 다른 프로세스보다 먼저 실행되어야 할 긴급한 프로세스가 우선적으로 실행될 수 없기 때문에, 프로세스 간의 차이가 존재하는 요소가 추가적으로 필요하다. Priority scheduling에선 프로세스마다 priority가 존재하고, running state의 프로세스가 다른 프로세스(ready queue의 프로세스)보다 항상 높은 priority를 갖도록 관리한다. 이를 통해 실행될 필요가 있는 프로세스가 다른 프로세스보다 먼저 CPU를 점유하도록 관리할 수 있는 utility를 얻을 수 있다.

* 1. **Advanced Scheduler**

Priority scheduling을 도입했을 때, 만약 priority가 높은 프로세스만 계속해서 생성된다면 priority가 낮은 프로세스는 영영 CPU를 점유할 수 없다. 이렇게 컴퓨터의 특정 자원을 긴 시간 대기해도 얻을 수 없는 문제를 starvation이라고 한다. Starvation을 막기 위해선 priority를 일정 시간마다 특정 규칙에 의해 바꾸어주어야 한다. 이를 Priority aging이라고 한다. Priority aging은 프로세스가 실행되는 경향성에 따라 I/O bound process에게 favor를 주는 방향으로 진행되어야 priority가 높은 특정 프로세스가 오랜 기간 CPU를 독점하는 것을 막을 수 있을 것이다. 이를 통해 starvation를 해결하고 fairness를 얻을 수 있다.

* 1. **개발 내용**

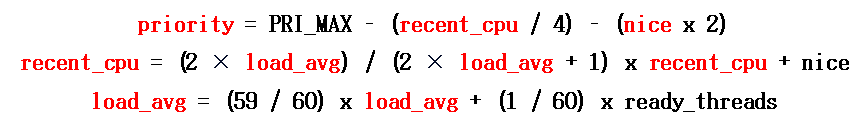
1. **Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.**

자고 있는 프로세스를 blocked state로 바꾸고, 특정 시간이 지나면 다시 깨워야 CPU 점유를 막을 수 있다. 그렇다면 특정 시간이 지났는지 어떻게 알 수 있을까? CPU는 clock tick마다 timer interrupt를 발생시킨다. timer interrupt가 발생할 때 마다 interrupt handler에서 자고 있는 프로세스가 깰 시간이 되었는지 확인하면 된다. 이를 위해, 1) 자고 있는 프로세스를 저장할 자료 구조를 선언하고, 2) 자게 된 프로세스가 그 자료 구조에 삽입될 때마다 PCB에 깨어나야 할 시간을 기록하고 block한다. 3) 이후 timer interrupt가 발생할 때마다 handler에서는 자료 구조를 순회하면서 깨어날 시간이 된 프로세스가 존재하는지 확인한다. 그런 프로세스가 존재할 경우, 다시 ready 상태로 바꿔준다.

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

앞서 말했듯이 Priority scheduling에선 프로세스마다 priority가 존재하고, running state의 프로세스가 다른 프로세스(ready queue의 프로세스)보다 항상 높은 priority를 갖도록 관리한다. PintOS 내에서 지원하는 list 자료 구조에는 list\_insert\_ordered라는 함수가 존재한다. list에 어떤 element를 삽입하면서 정렬된 상태가 유지되도록 위치를 찾아주는 함수인데, 이를 pintOS의 thread를 관리하는 여러 함수 및 자료 구조에서 사용할 것이다. 모든 자료 구조에 대해 항상 priority가 높은 프로세스가 가장 앞에 있도록 하는 것이다. Priority scheduling에선 1) 새로운 프로세스가 ready queue에 삽입되고, 2) ready queue가 priority에 따라 정렬된 이후, 3) 현재 running process와 ready queue의 first element의 priority를 비교해 더 큰 프로세스가 CPU를 점유한다. 이 과정을 list\_insert\_ordered 함수를 이용해 진행할 것이며, 자세한 code-level implementation은 후술한다.

1. **Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.**

****

강의 자료 내에서 priority 값은 위와 같이 계산된다. 쉽게 알 수 있는 상수와 변수를 제외하고, priority가 CPU 점유 시간에 따라 최적화되도록 돕는 세 개의 변수에 대한 설명은 다음과 같다.

[1] nice : 프로세스의 niceness를 나타내는 값이다. 친절하다는 의미 그대로 다른 프로세스에게 CPU 점유 권한을 많이 양도하는지를 의미하고, pintOS 내에서 -20부터 20까지의 값을 프로세스마다 매긴다. Priority는 nice가 클수록 작아지는 것을 확인할 수 있다. 이후 recent\_cpu 값에 영향을 준다.

[2] load\_avg : System-wide한 값으로 OS 내에 얼마나 많은 프로세스가 작동중인지를 표현한다. 매 초마다 갱신되는데, 식에서 알 수 있듯이 이전 스스로의 값의 영향을 받는다. 갱신 시에 ready\_threads 값이 이전보다 크다면 커지고, 그렇지 않다면 작아진다. 이 값은 이후 recent\_cpu 값에 영향을 준다.

[3] recent\_cpu : 해당 프로세스가 이전에 CPU를 점유할 때 얼마나 오래 사용했는지를 나타낸다. load\_avg와 마찬가지로 매 초 갱신되면서 이전 값의 영향을 받는다. CPU를 오래 사용하는 CPU bound job은 priority가 낮아져야 convoy effect 등의 문제를 막을 수 있으므로, recent\_cpu 값이 클수록 priority는 낮아진다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

우선 프로젝트를 진행하기 전 항상 읽어보는 pintOS 매뉴얼부터 읽고 이번 프로젝트 구현의 핵심 개념과 방법을 이해하였다. 또, userprog에서 진행되던 이전 프로젝트와 다르게 threads 내의 코드를 직접 조작해야 했으므로, threads 디렉토리 내의 코드 파일을 모두 읽어보면서 계획을 구상하였다.

코드는 pintOS 내의 CPU clock tick과 친숙해지기 위해 alarm clock부터 구현을 시작하였는데, 구현 중 마주친 문제로 인해 C의 외부 변수 참조 개념을 이해하는데 시간을 썼다. 다음으로 priority scheduling을 구현하였고, scheduler에 대한 이해를 바탕으로 priority aging / BSD scheduler를 구현하였다. 이 부분이 가장 까다로웠는데, pintOS가 소수점 연산을 지원하지 않았기 때문이다. PintOS 매뉴얼을 읽어보면 평소에 쓰던 방법과는 다르게 bitwise한 소수점 연산을 진행해야 하는데, 이 부분이 상당히 헷갈렸다. 또 구현을 수정하면서 make check를 계속 돌려야 하니 시간 투자도 많이 하게 되었다. 요구되는 Test case가 모두 pass인 것을 확인한 이후에는 보고서를 작성하였다.

* 1. **개발 방법**

**1) Alarm Clock**

우선, Sleep 등을 통해 생기는 blocked thread를 저장하고 관리하기 위한 자료구조인 slept thread queue를 선언하고 사용할 것이다. 앞서 설명한대로 Sleep하게 된 프로세스의 PCB에서 state를 blocked로 바꾸고, 이 queue에 삽입한다. 또, 깨어날 시간을 알아야 하기 때문에 thread structure에 wakeup field를 추가하여 기록해준다. 이 과정은 우리가 주로 수정해야 하는 timer\_sleep 함수 내에서 이루어진다. 이후, slept thread queue에 있는 프로세스들을 timer interrupt가 발생할 때마다 체크하면서 깨어날 시간이 된 프로세스가 있다면 깨워준다. timer interrupt 발생 시마다 프로세스가 해야 할 동작이 있는 thread.c의 thread\_tick 함수에서 이를 수행한다.

**2) Priority Scheduling**

Priority scheduling을 위해, ‘ready queue 내 프로세스의 max priority update가 있을 가능성이 있는 행위가 발생하면 list\_insert\_ordered 또는 reschedule’이 필요하다. 밑줄 친 동작은 다음과 같다.

[1] thread\_create : 쓰레드가 생성될 때는 당연히 max priority가 변할 수 있다.

[2] thread\_unblock : 쓰레드가 unblock될 때 다시 ready queue로 들어가기 때문에 max priority가 변할 수 있다.

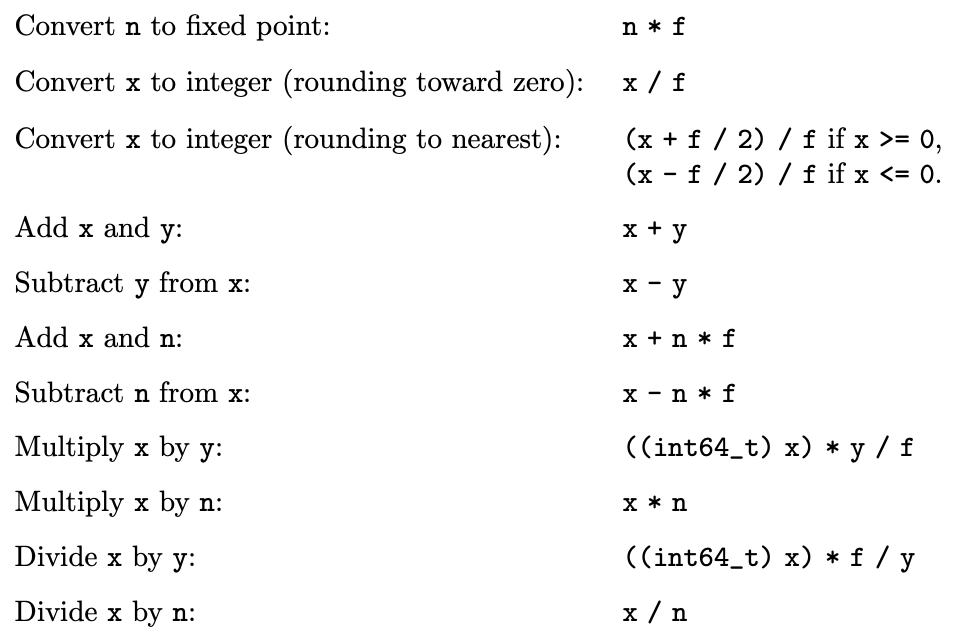
[3] thread\_yield : 현재 실행중인 쓰레드가 CPU 제어권을 넘길 때 다시 ready queue로 들어가기 때문에 max priority가 변할 수 있다.

[4] thread\_set\_priority, thread\_set\_nice : priority나 그 값에 영향을 주는 값이 변할 경우 당연히 max priority가 변할 수 있다.

[5] semaphore up : 어떤 세마포가 up되어 쓰레드들이 race를 하는 경우 그 중 가장 priority가 높은 쓰레드가 우승하여 ready queue로 들어가야 한다. 이 과정에서 max priority가 변할 수 있다.

이 routine 안에서 list\_insert\_ordered를 통해 가장 priority가 높은 동작이 실행될 수 있도록 한다. 사실 thread\_yield를 이대로 구현하고 나면 이를 ‘priority를 기준으로 한 running process / ready queue refreshing routine’으로 사용할 수 있기 때문에, priority를 수정한 이후 thread\_yield를 호출해 쉽게 구현할 수 있다. 즉, 갱신이 필요할 때 thread\_yield를 호출하면 되는 것이다.

**3) Advanced Scheduler**

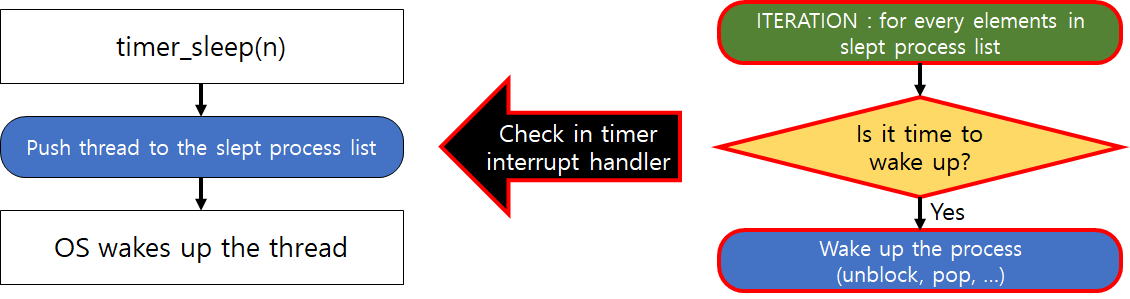
****

PintOS 매뉴얼을 읽어보면, 보통 실수 연산을 할 때 사용하는 floating point 방식이 아닌 fixed point 방식을 사용하는 것을 알 수 있다. 따라서, 위에서 설명한 priority aging 계산 구현을 위해 fixed point 실수 연산부터 구현해야 한다. integer bit과 fraction bit를 따로 두어 계산하기 때문에, 이를 유념하여 구현하여야 한다.

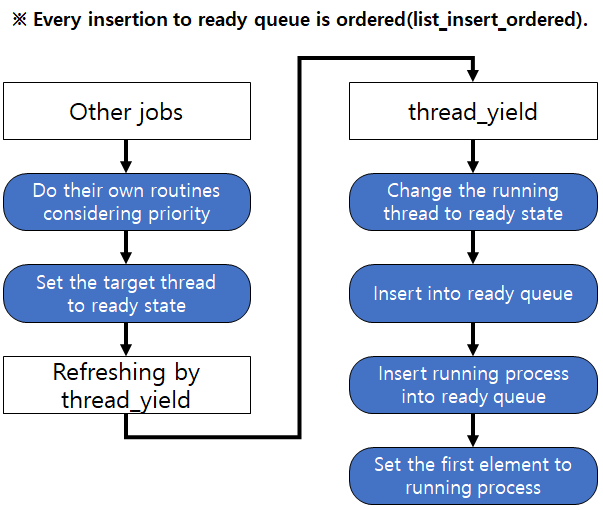
Fixed point arithmetic이 구현되었다면 이제 aging을 위한 variable을 선언해야 한다. priority, nice, recent\_cpu는 쓰레드마다 갖는 값이기 때문에 늘 하던 대로 thread structure에 값을 선언하고, load\_avg는 System-wide한 값이므로 모두가 참조할 수 있도록 global로 선언한다. 이제 이 값을 토대로, priority, recent\_cpu, load\_avg를 다시 계산하는 루틴을 작성하고, 이를 각 변수의 정해진 주기마다 실행하면 된다. 주의할 점은, priority를 다시 계산하는 루틴을 실행한 이후에는 당연히 yield를 해주어야 한다는 것이다. 이때 앞서 설명한 thread\_yield를 사용하지 않고 intr\_yield\_on\_return을 사용하는데, 이는 timer interrupt handler 과정 중에 실행되지 않고 handling을 마쳤을 때 yield하도록 하기 위함이다. 이렇게 priority와 priority aging을 모두 구현하면 BSD scheduler가 완성된다.

1. **연구 결과**
   1. **Flow Chart**

**1) Alarm Clock**

****

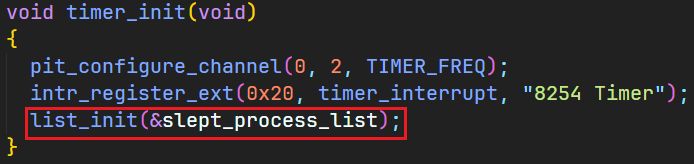
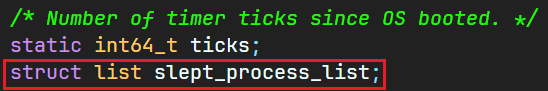
**2) Priority Scheduling**



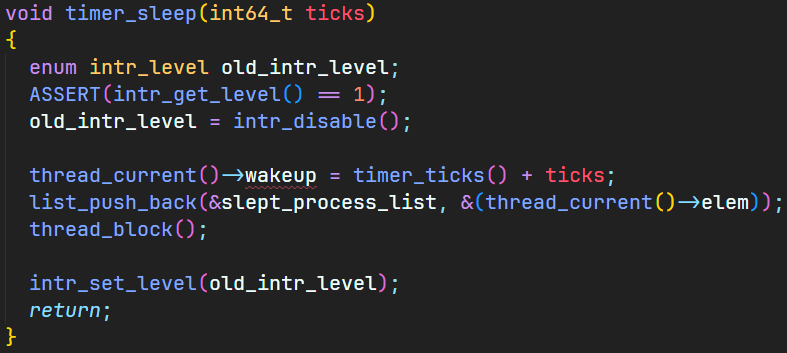
* 1. **제작 내용**

**1) Alarm Clock**

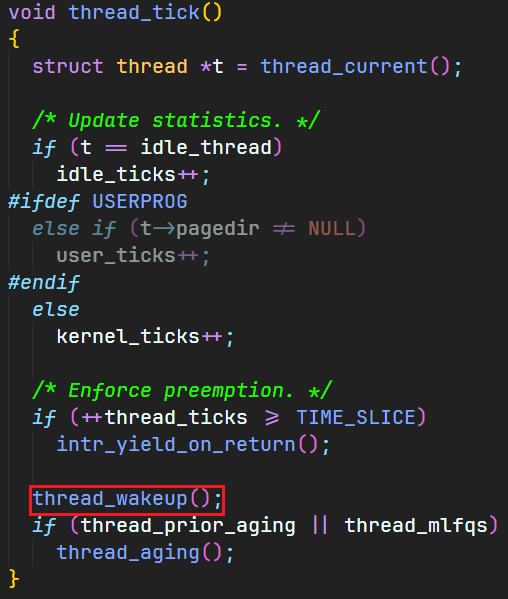
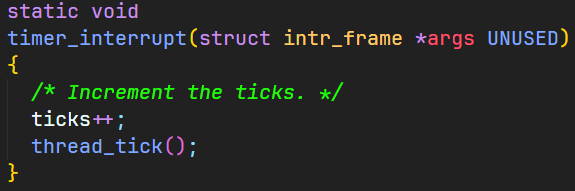
우선 앞서 이야기했듯 자고 있는 프로세스를 관리하는 자료구조가 필요하다.



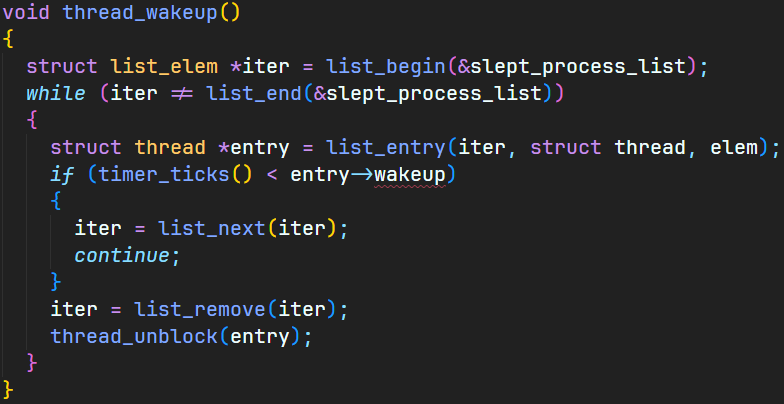
devices/timer.c의 global variable로 slept\_process\_list를 선언하였다. 이 리스트는 타이머가 시작되면서 동시에 초기화된다.



이후 timer\_sleep이 호출되면 위와 같이 작동한다. thread structure에 추가된 field인 wakeup에 해당 프로세스가 깨어나야 할 시간을 기록한다. 이후, slept\_process\_list에 프로세스를 기록하고 CPU를 점유할 수 없도록 block한다. 이렇게 하면 정해진 시간까지는 CPU를 점유할 수 없는 프로세스가 된다.



timer\_interrupt의 동작에 따라 깨어날 시간이 되었는지 확인하기로 계획했으므로, 이를 구현해야 한다. timer\_interrupt를 보면, 쓰레드들에게 시간이 지나갔으므로 필요한 동작을 하라고 알리는 thread\_tick 함수가 호출되고 있는 것을 볼 수 있다. timer\_interrupt에서 구현해도 되지만, 값 갱신 등을 통일감 있게 구성하기 위해 thread\_tick에서 깨우는 동작 또한 구현하였다. thread\_tick의 내부를 보면, thread\_wakeup이라는 routine이 호출되는 것을 볼 수 있다.



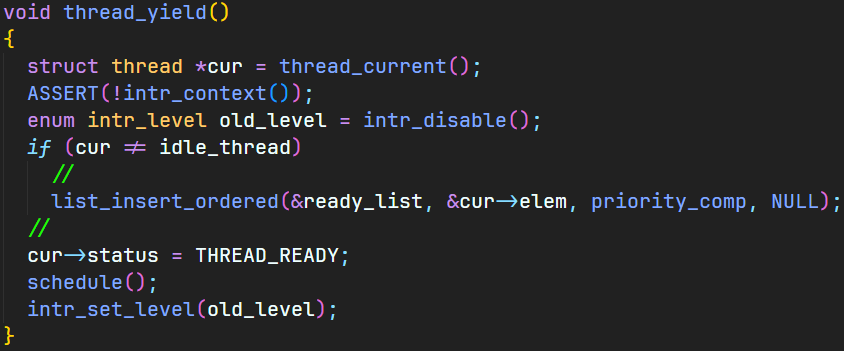
thread\_wakeup 내에선 slept\_process\_list를 처음부터 끝까지 순회하며 깨울 프로세스가 있는지 찾는다. 현재 tick이 wakeup보다 낮으면 넘어가고, 높거나 같다면 리스트에서 제거한 후 unblock한다.



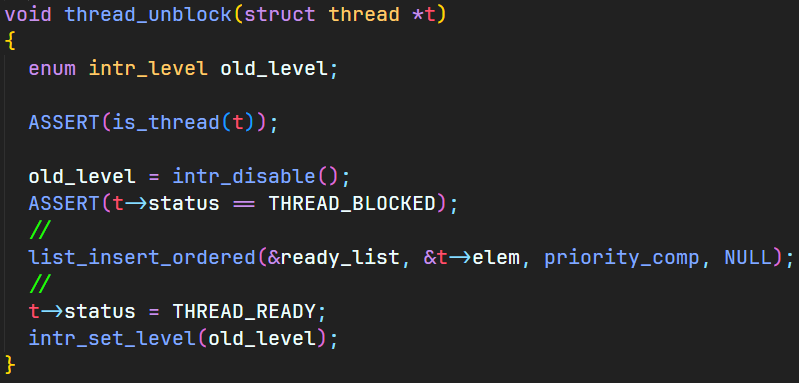
주의할 점은, 다른 디렉토리의 다른 코드끼리 자료 구조를 공유하기 위해 C의 외부 변수 참조를 사용해야 한다. 이는 extern 키워드로 쉽게 가능하다.

**2) Priority Scheduling**

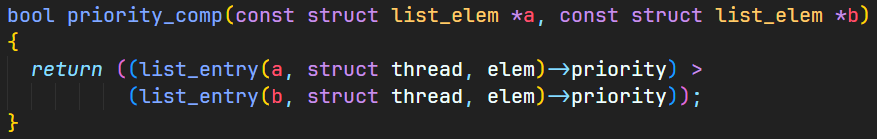
앞서 작성한 priority scheduling 구조 설명과 flow chart에서 흐름을 모두 계획했다. 그에 맞게 코드를 작성하면 된다.



Ready queue에 프로세스를 삽입해야 하는 thread\_yield에선 list\_insert\_ordered를 사용한다. 코드를 잘 읽어보면, list\_push\_back에서 list\_insert\_ordered로 바뀐 것 말고는 원본 코드와 차이가 없음을 알 수 있다. priority 기준 정렬된 상태로 ready queue를 채우고, schedule만 하면 되는 간단한 과정이다.



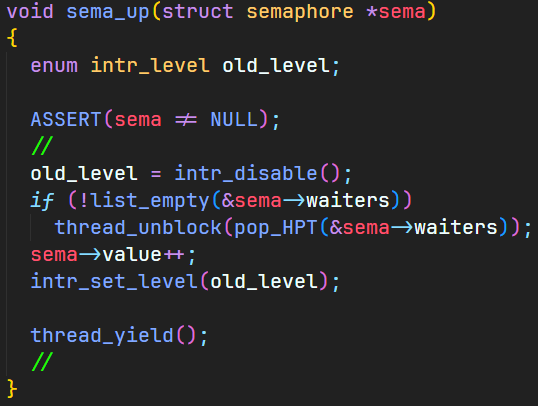
마찬가지로, thread\_unblock에서도 list\_push\_back이 list\_insert\_ordered로 변했다. 해당 함수의 인자로 꾸준히 보이는 priority\_comp가 있는데, 이는 정렬 기준을 알려주는 compare 함수이다. C의 라이브러리 함수 중 하나인 qsort의 인자로 들어가는 compare 함수를 생각하면 쉽다.



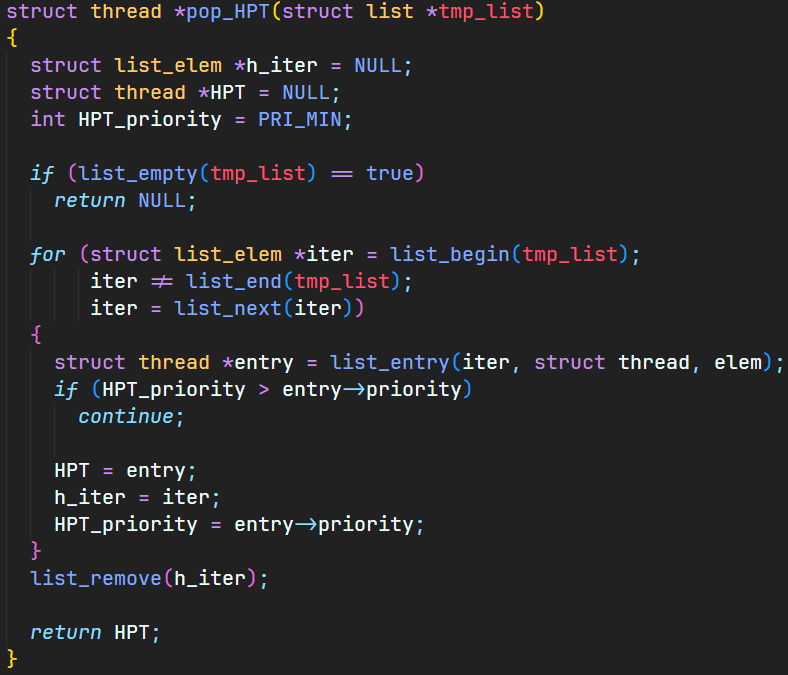
아주 단순하게 priority의 대소 관계를 비교하여 정렬한다. 이를 통해 ready queue가 항상 priority를 기준으로 정렬된 상태로 놓이게 되고, 이 상태에서 기본적인 FIFO로 schedule을 하면 그게 바로 priority scheduling이 되는 것이다. 이제, 이 두 함수를 이용해 ready queue에 새로운 프로세스가 발생할 때마다 이용하면 된다.



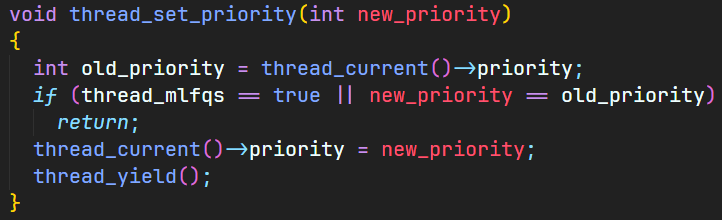
먼저, thread\_create 함수의 마지막 부분에 단 두 줄이 추가되었다. 현재 CPU를 점유하고 있는 프로세스의 priority보다 지금 생성되어 ready queue에 들어간 프로세스의 priority가 더 크다면 CPU를 포기하고 다시 schedule한다.



다음은 thread/synch.c의 sema\_up이다. 앞서 priority가 각기 다른 waiter들의 race condition에 대해 설명했듯, 가장 priority가 높은 프로세스에 대해 lock을 해제하고 yield해야 한다. 코드를 보면, sema->waiters 중 HPT를 pop해 그것만 unblock 하고 yield하는 것을 확인할 수 있다.



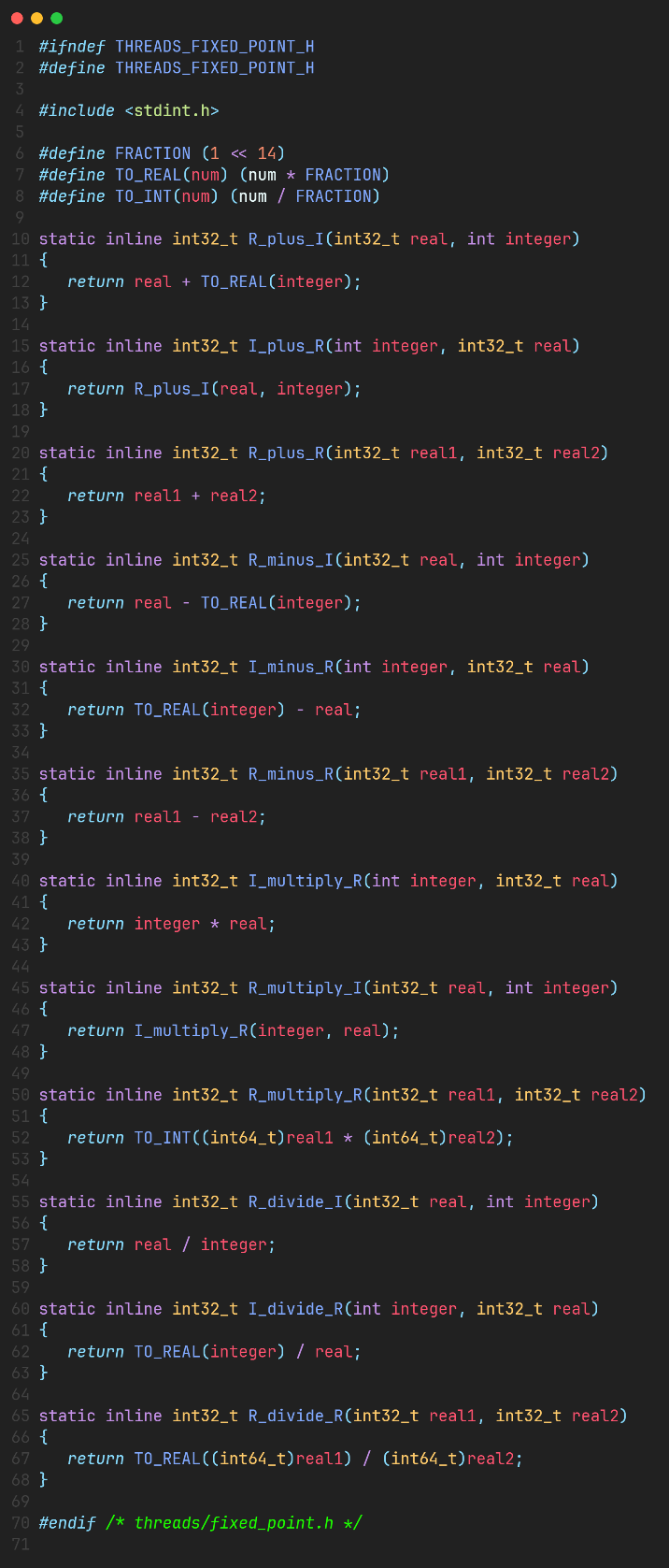
pop\_HPT는 프로세스를 담고 있는 어떤 리스트에 대해 가장 높은 priority를 갖는 쓰레드(Highest Priority Thread)를 pop하는 함수이다. 리스트를 순회하면서 가장 priority가 높은 entry를 찾아, 리스트에서 제거한 후 반환해준다.



마지막으로 thread\_set\_priority이다. 기존의 old\_priority와 바꾸려는 new\_priority가 같으면 넘어가고, 아니라면 업데이트한다. 이후 yield하면 된다.

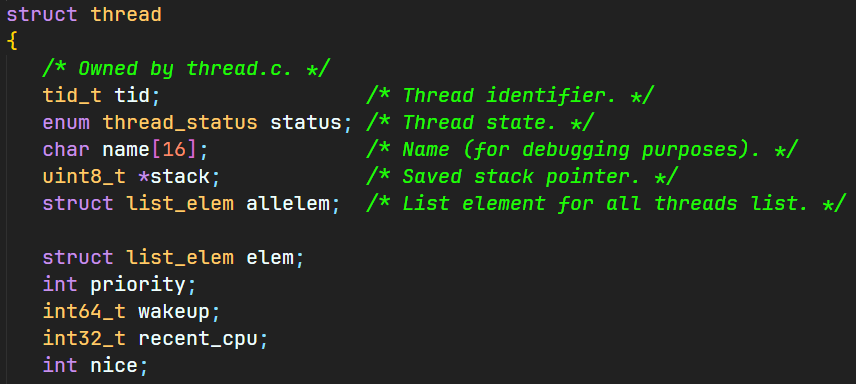
**3) Advanced Scheduler**

Fixed point arithmetic을 따로 구현해야 한다고 설명했었다. 이 부분이 꽤 구현량이 많아서 코드 정리에 난항을 겪다가, 다른 사람들의 코드를 참고해보니 fixed\_point.h라는 헤더 파일을 따로 만든 것을 보고 나도 그렇게 하였다. 아래는 fixed\_point.h에 정의된 fixed point arithmetic 함수이다.



코드를 확인하면, 32비트 정수 자료형에 대해 14비트는 fraction으로, 나머지는 integer로 사용하는 것을 확인할 수 있다. 또, multiply와 divide에 대해선 overflow가 발생할 수 있으므로 int64\_t로 casting하고 값을 계산한다. 단순한 bitwise 연산이고, pintOS 매뉴얼 그대로 구현해 과제의 문맥과 어울리진 않는다고 생각하여 자세한 설명은 생략한다.

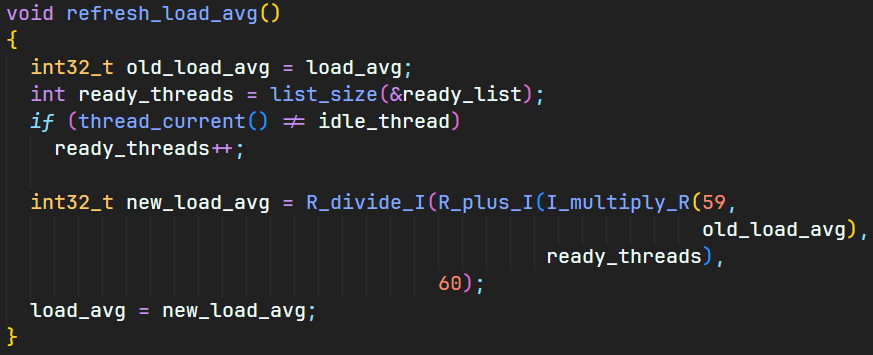
정의한 함수를 이용해, 시간에 따라 다시 계산되는 priority와 그 요소를 구할 것이다. 가장 먼저, load\_avg, nice, recent\_cpu 등의 field를 미리 선언해야 한다.



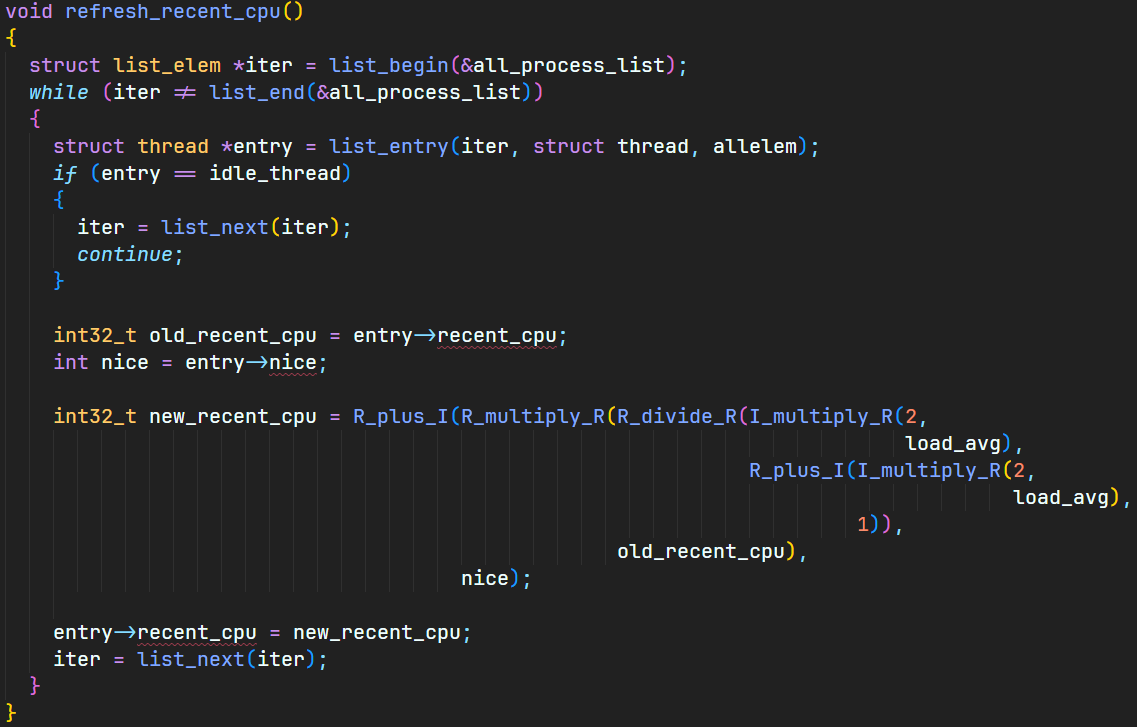
먼저, recent\_cpu와 nice는 프로세스마다 고유한 값이므로 threads/thread.h의 thread structure에 선언한다.



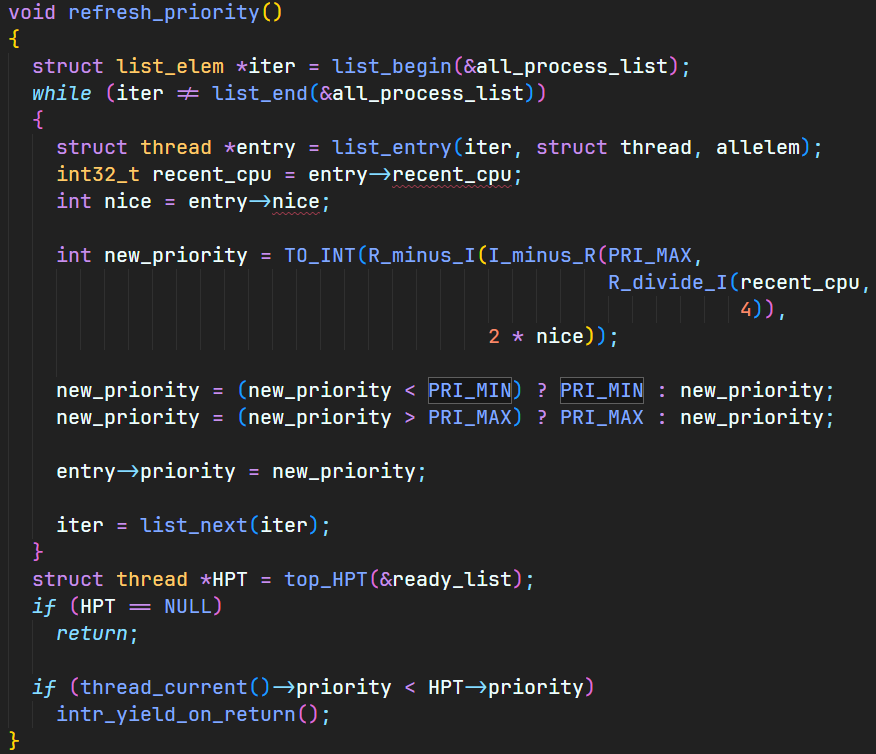
load\_avg는 system-wide한 값이므로 threads/thread.c에 global static variable로 선언한다.



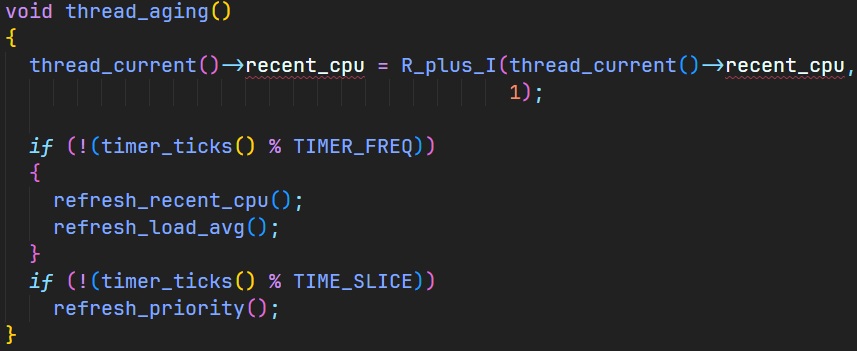
load\_avg를 주어진 공식에 따라 다시 계산하는 함수인 refresh\_load\_avg이다. 단순한 연산 과정이므로 설명은 필요가 없을 것으로 보인다. idle\_thread의 경우 세지 않는 것을 유념하자.



다음으론 refresh\_recent\_cpu이다. 현재 OS에 존재하는 모든 프로세스를 관리하는 all\_process\_list를 순회하며, 모든 element에 대해 recent\_cpu를 새로 고침 한다. 역시 idle\_thread의 경우 세지 않는다.

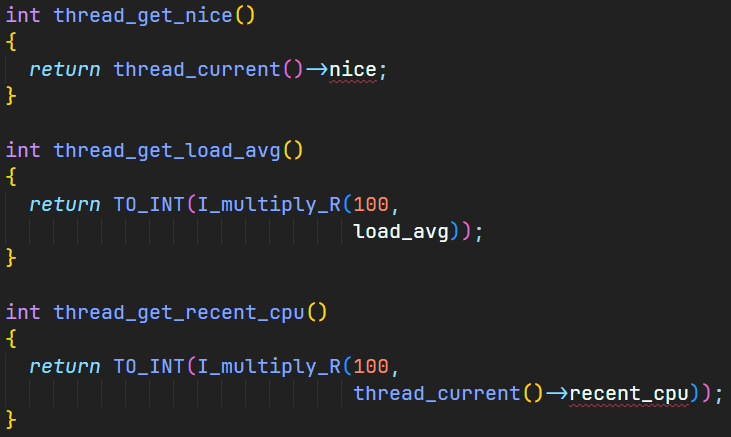


마지막으로 refresh\_priority이다. 역시 all\_process\_list에 대해 순회하며 해당 entry의 priority를 모두 갱신해주고 있다. 또, 상방 최댓값과 하방 최솟값을 초과하면 조정한다. 모든 프로세스에 대해 priority를 갱신하고 나서, priority가 현재 구동중인 프로세스보다 높은 프로세스가 발생한다면 intr\_yield\_on\_return을 호출하여 handling 이후 yield할 수 있도록 한다.

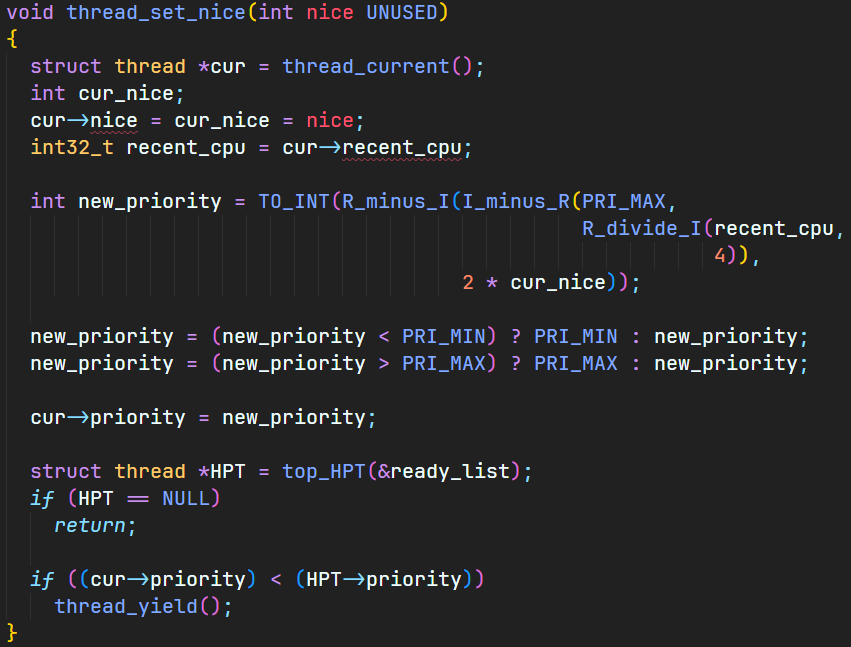


이제 이 함수들을 이용해 실질적인 aging routine을 만들어주기만 하면 된다. recent\_cpu, load\_avg를 갱신하는 주기와 priority를 갱신하는 주기가 다르다는 점을 유념해야 한다. thread\_tick에서 thread\_aging이 불리고, 위 routine이 실행되면 priority aging이 동작한다.

이 외에도, pintOS는 형식 정의만 되어있고 내용이 구현되지 않은 함수가 몇 개 있다. 이 또한 구현했다.



보통의 경우 위와 같이 단순하게 해당하는 값을 return하면 된다. 예외적으로 thread\_set\_nice는 복잡한 구현 과정을 거치는데, 아래를 참고하자.

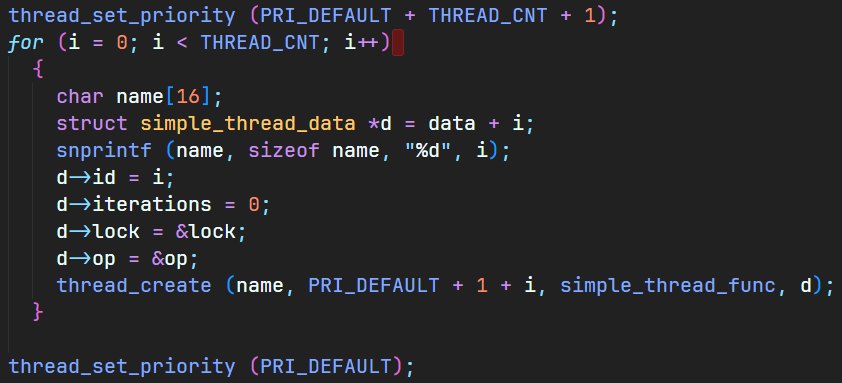


Nice 값은 priority에 관여하기 때문에, priority가 변할 때와 동일하게 설정해주어야 한다. 변한 nice에 대해 새로운 priority를 구하고 변경해준다. 이후 refresh\_priority와 동일하게 현재 CPU를 점유중인 프로세스보다 priority가 클 경우 yield한다. handler 내부에서 실행되지 않기 때문에 thread\_yield를 바로 호출해도 된다는 점의 차이가 있다.

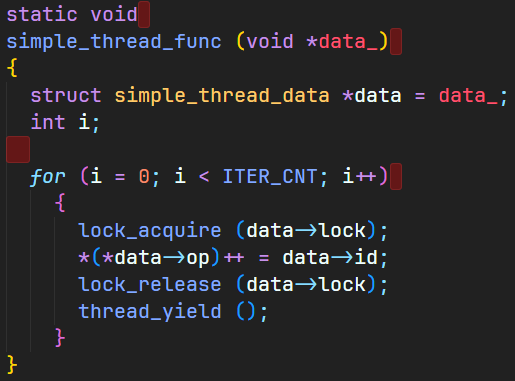
이렇게 하면 Alarm Clock, Priority Scheduler, BSD Scheduler가 모두 구현된다.

* 1. **시험 및 평가 내용**

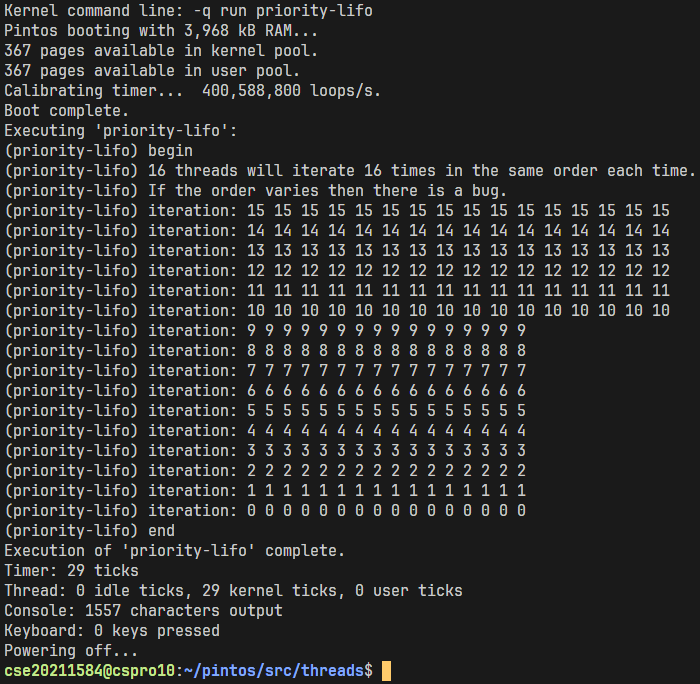
**1) priority-lifo test**



priority-lifo.c의 주요 파트이다. 위 사진은 생성할 쓰레드의 priority를 설정하는 부분인데, 우선 본인의 priority를 가장 높게 해 쓰레드를 생성하는 도중 우선 순위를 빼앗기지 않도록 한다. 이후, THREAD\_CNT(=16)만큼 반복하여 priority가 점점 커지는 쓰레드들을 만든다. 즉 LIFO라는 이름에서 알 수 있듯이, 마지막에 생성된 쓰레드의 priority가 가장 높게 만들어져 먼저 실행되도록 한다.

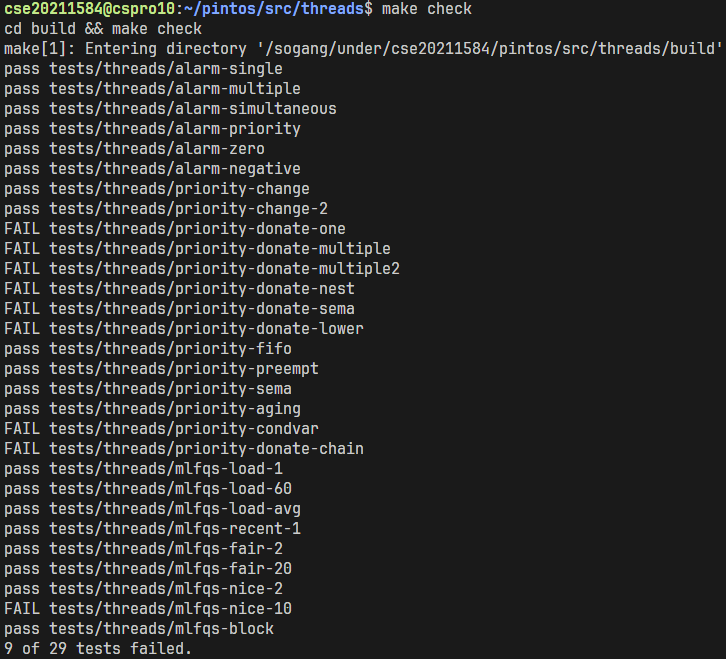


이러한 방식으로 생성된 쓰레드들은 위의 동작을 한다. 사실상 의미 있는 동작은 없고, 몇 번의 반복 이후 yield한다. 공유된 자료구조인 data에 접근해 작동한 쓰레드의 순서를 기록하기 때문에, ready queue에서 나와 running된 순서를 알 수 있다. Priority scheduling이 제대로 설계되어 있다면, 마지막에 만들어진 id가 15인 쓰레드부터 0인 쓰레드까지 순차적으로 일을 마치고 data에 기록할 것이다. 정말로 그러한지 결과를 확인하면 된다.



‘pintos -v -- -q run priority-lifo’의 실행 결과이다. 예상대로 높은 priority의 쓰레드가 마지막에 생성되었다고 하더라도 가장 먼저 실행되는 모습을 볼 수 있다. 또, 단순 priority scheduling이기 때문에 높은 priority의 job이 terminate 되기 전에 그보다 낮은 priority의 job이 CPU을 점유하지 않는다. Priority scheduler가 잘 설계되었다고 볼 수 있다.

**2) make check**



BSD Scheduler까지 모두 포함하여 요구하는 test case 중 mlfqs-nice-10을 제외하고 전부 pass하였다. mlfqs-nice-10은 아마 사소한 최적화의 문제로 FAIL을 받은 것 같다.