**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

조 / 조원 : 20161595 배성현

개발 기간 : 2020/11/18~2020/11/27

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

현재 핀토스에서 busy-waiting으로 구현되어 비효율적인 Alarm Clock을 busy-waiting을 사용하지 않는 방법으로 작동할 수 있도록 구현하고, 현재 round-robin scheduling을 사용하는 scheduler를 threads의 priority를 고려하여 scheduling할 수 있도록 Priority Scheduling을 구현한다. 또한 Priority scheduling을 하게 되면 priority가 낮은 process가 scheduling되지 않는 starvation이 발생할 수 있기 때문에 이를 막기 위한 aging을 구현한다. 마지막으로 general purpose scheduler인 BSD Scheduler(Advanced Scheduler)를 추가구현으로써 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

기존의 pintos에서의 timer\_sleep()에서는 busy waiting을 하기 때문에 thread가 running state와 ready state 사이에서 반복되게 된다. 이는 비효율적이기 때문에 Alarm Clock부분을 busy waiting하지 않도록 바꾸어 주어야 한다. 따라서 Alarm Clock에 대한 부분을 busy waiting 하지 않도록 수정해주면 이러한 비효율성을 피할 수 있다.

* 1. Priority Scheduling

Priority Scheduling을 구현하게 되면 각 프로세스마다 우선순위를 부여하여 이러한 우선순위에 따라서 thread들이 scheduling될 수 있게 된다. 또한 aging을 통해 priority가 낮은 thread들이 점진적으로 priority가 높아지게 되어 결국에는 schedule되도록 하여 starvation을 막을 수 있게 된다.

* 1. Advanced Scheduler (추가구현을 한 경우)

I/O를 많이 수행하는 Thread의 경우에는 많은 CPU time을 필요로 하진 않지만 I/O device를 busy하게 유지하기 위해 fast response time을 필요로 한다. 또한 CPU bounded인 thread의 경우에는 fast response time을 필요로 하지 않지만, 많은 CPU time을 필요로 한다. 따라서 이를 balance하게 scheduling 할 수 있도록 Advanced Scheduler(BSD Scheduler)를 구현하여야 한다. 그리고 이를 구현하게 되면 CPU time이 우선순위 재계산에 이용되기 때문에 CPU사용 시간이 많았던 Thread는 다음 Scheduling시에 낮은 우선순위를 갖게 되고 I/O thread들은 상대적으로 priority가 높아져 fast response time을 가질 수 있게 되어 balance하게 scheduling이 가능해 진다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

먼저 block된 thread들을 저장하는 queue를 만들고 특정 thread가 block이 될 때 해당 thread가 일어나야 하는 시간을 같이 저장하여 queue에 넣는다. 그리고 timer\_interrupt함수가 매 tick마다 호출되기 때문에 timer\_interrupt함수가 호출될 때마다 queue 전체를 검사하여 일어날 시간이 다 된 thread를 unblock함으로써 ready\_list(ready queue)에 넣어 blocked 상태의 thread를 깨울 수 있다.

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

Priority Scheduling에서는 preemptive와 non-preemptive의 두 가지 방법 중 하나의 방법을 사용할 수 있다. 먼저 Non-preemptive의 경우에 Ready list에 Running thread보다 높은 priority를 가진 thread가 들어와도 running thread의 CPU점유를 바로 뺏는 것이 아닌 running thread를 다 돌리고나서 ready queue에서 priority가 가장 높은 것을 찾아 돌리게 되고, preemptive의 경우에는 Ready list에 Running thread보다 높은 priority를 가진 thread가 들어오면 Running thread가 높은 priority를 가진 thread에 CPU점유를 양보하게 된다. 따라서 해당 프로젝트에서는 preemptive를 사용하기 때문에, thread 생성, sema\_up등의 경우와 같이 thread가 ready list에 들어올 때 priority를 체크하여 새로운 thread가 running thread보다 priority가 높으면 running thread가 CPU를 새로운 thread에게 양보(yield)하도록 하여야 한다.

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

**nice(정수)**: 각각의 thread는 nice값을 가지며 nice값은 -20~20의 범위를 갖는다. nice값이 양수인 경우에 우선순위를 낮추어, 다른 thread들이 cpu를 차지할 수 있도록 하며, 가장 처음에 만들어진 초기 thread의 초기 nice값은 0을 가지고 그렇지 않은 경우에는 부모 thread의 nice값을 상속받는다.

**recent\_cpu(실수)**: thread의 cpu time을 추정하는 값으로 최근의 cpu time에 가중치를 두어 계산되게 된다(최근일수록 더 높은 가중치, 최근이 아닐수록 더 낮은 가중치). Time interrupt가 발생할 때마다 idle thread가 아닌 running상태에 있는 thread의 recent\_cpu값은 1씩 증가되게 되며, 1sec마다 running, ready, blocked상태에 있는 모든 thread들의 recent\_cpu값이 아래 공식에 의하여 재계산되게 된다.

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

**load\_avg(실수)**: ready상태에 있는 thread의 평균 개수를 나타내며 시스템 boot시 0으로 초기화 된다. 또한 매 초마다 아래 공식에 의하여 load\_avg값이 재계산되게 된다. (ready\_threads: ready또는 running 상태에 있는 thread의 개수)

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

**Priority(정수):** 0(PRI\_MIN)~63(PRI\_MAX)의 범위를 가지며 매 4번째 TICK마다 아래 공식에 의하여 재계산되게 된다.

* + - 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

|  |  |
| --- | --- |
| 일정 | 내용 |
| 11/18 | 숙제 강의 자료 및 강의 확인 |
| 11/19 | 관련 Pintos Manual 확인 |
| 11/20~11/21 | Alarm Clock 구현 |
| 11/22~11/23 | Priority Scheduling 구현 |
| 11/24~11/26 | Advanced Scheduler 구현 |
| 11/27 | Document 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
* Alarm Clock

먼저 threads/thread.h에 현재 block된 thread들을 나타내는 list인 sleepingList라는 자료구조를 추가해줄 것이고, thread구조체에 block이 시작된 시간과, 어느 시간정도 block이 될지를 저장해주는 변수인 sleepStartTime과 sleepTime이라는 변수를 추가해 줄 것이다. 이후 threads/thread.c의 thread\_init함수에 위의 sleepingList를 init해주는 코드를 넣어 줄 것이다. 또 block될 thread의 sleepStartTime과, sleepTime을 업데이트해주고 sleepingList에 넣어주는 함수인 thread\_sleep이라는 함수를 추가해 줄 것이며, sleepingList를 순회하며 체크하여 wake할 thread이면 unblock해주는 thread\_awake라는 함수를 추가해 줄 것이다. 이 후 devices/timer.c의 timer\_sleep함수에서 이전의 busy-waiting기반의 코드를 지우고 위의 thread\_sleep함수를 호출해 주도록 하는 코드를 추가해줄 것이고, timer\_interrupt함수에서 thread\_awake함수를 호출해 주도록 하여 Alarm Clock을 구현할 수 있도록 할 것이다.

* Priority Scheduling

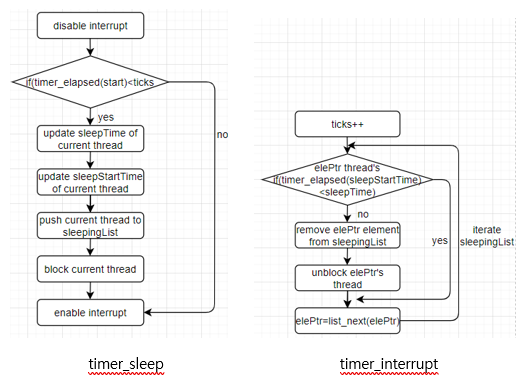
먼저 threads/thread.c에 thread들의 priority를 비교하여 주는 함수인 priority\_greater함수를 추가하여 줄 것이다. 그리고 thread가 ready\_list에 추가되어야 하는 상황인 thread\_unblock과 thread\_yield함수에서 list\_insert\_ordered함수를 이용하여 thread가 priority에 따라 내림차순으로 정렬된 채로 ready\_list에 들어갈 수 있도록 할 것이다. 또한 preemptive하게 구현하기 위하여 thread\_set\_priority함수와 thread\_create함수와 같이 thread의 priority가 정해지는 순간에 현재 돌고 있는 current\_thread와 ready\_list의 가장 첫 thread의 priority를 비교하는 코드를 넣어주어 ready\_list의 첫 thread의 priority가 더 높다면 yield할 수 있도록 하는 코드를 추가하여 줄 것이다(ready\_list는 priority에 따라 정렬되어 있을 것이므로 가장 첫 thread가 우선순위가 제일 높음). 그리고 semaphore의 waiting list를 나타내는 waiters에 thread를 추가하는 threads/synch.c의 sema\_down함수에서 list\_insert\_ordered함수를 이용하여 thread를 추가하도록 하고 sema\_up함수에 의하여 waiters에서 깨어날 때 가장 priority가 높은 thread가 먼저 깨어나도록 하는 코드를 넣어주고 현재 돌고 있는 current\_thread와의 priority를 비교하고 yield하는 코드를 넣어주어 sema\_up상황에서도 preemptive하게 작동할 수 있도록 할 것이다. 마지막으로 priority가 낮은 thread의 starvation이 일어나지 않도록 매 틱마다 ready\_list안의 thread들의 priority를 1씩 증가시켜주는 thread\_aging함수를 threads/thread.c에 추가하여 줄 것이고 thread\_set\_priority, thread\_get\_priority등의 함수를 완성시켜 줄 것이다.

* Advanced Scheduler

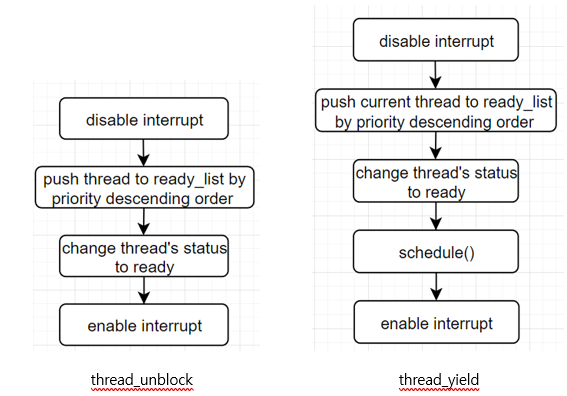
먼저 각각의 thread가 자신의 nice값과 recent\_cpu값을 가지고 있을 수 있도록 threads/thread.h의 thread구조체에 nice와 recent\_cpu변수를 추가하여 줄 것이다. 또한 threads/thread.c에 load\_avg라는 static 전역 변수를 선언하여 ready상태에 있는 thread의 평균 개수를 저장하고 있을 수 있도록 할 것이다. 그리고 thread\_init함수에 이러한 load\_avg변수와, 초기 생성되는 thread의 nice값과 recent\_cpu값을 0으로 초기화해 주는 코드를 추가해 줄 것이고, thread\_create함수에서는 새로 생성된 thread가 parent의 nice값과 recent\_cpu값을 상속받을 수 있도록 하는 코드를 추가해줄 것이다. 이어서 Pintos 커널에서는 실수 연산(Floating Point 연산)을 지원하지 않고, 우리가 구현하고자 하는 advanced scheduler를 구현하기 위해서는 실수 연산이 필요하기 때문에, 핀토스 매뉴얼을 참고하여 Fixed-Point연산을 할 수 있도록 하는 함수들을 만들어 threads/thread.c에 추가하여 줄 것이다. 또한 advanced\_scheduling함수를 만들고 이를 thread\_tick함수에서 thread\_mlfqs가 true일 때 호출해주도록 하여 advanced scheduler를 thread\_mlfqs가 true일 때 사용할 수 있도록 할 것이다. 그리고 이 advanced\_scheduling 함수에는 idle thread가 아닌 running상태에 있는 thread의 recent\_cpu값을 1씩 증가시키는 코드와, 1초에 한번씩 load\_avg, recent\_cpu공식에 따라 load\_avg와 모든 thread들의 recent\_cpu를 재계산하는 코드, 그리고 매 4틱마다 모든 thread의 priority를 재계산하는 코드를 추가하여 줄 것이다. 그리고 이렇게 재계산된 priority에 따라 current thread와 ready\_list의 가장 처음에 있는 thread의 priority를 비교하여 yield하는 코드를 추가하여 preemptive하게 작동될 수 있도록 할 것이다. 마지막으로 thread\_set\_nice, thread\_get\_nice, thread\_get\_load\_avg, thread\_get\_recent\_cpu등의 함수를 완성시켜 줄 것이다.

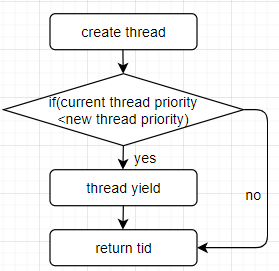
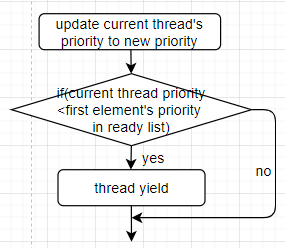
1. **연구 결과**
   1. **Flow Chart**

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

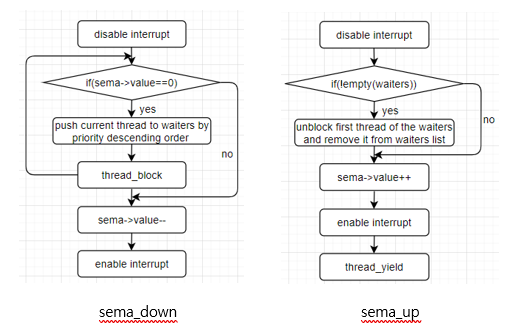


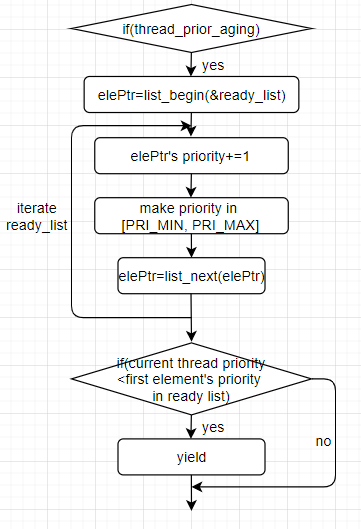
* Priority Scheduling



for preemption in thread\_create and thread\_set\_priority





thread\_aging

* 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



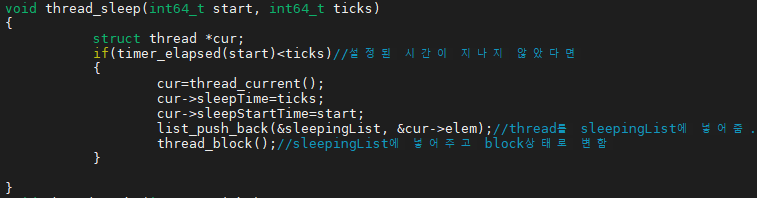
먼저 threads/thread.h에 위와 같이 현재 block된 thread들을 저장할 sleepingList라는 자료구조를 추가해주었고,



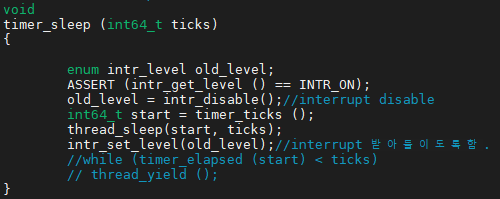
또 thread구조체에 block이 시작된 시간과, 어느 시간정도 block이 될지를 저장해주는 변수인 sleepStartTime과 sleepTime이라는 변수를 추가해주었다.



이후 threads/thread.c의 thread\_init함수에 위와 같이 sleepingList를 init해주는 코드를 추가해주었다.

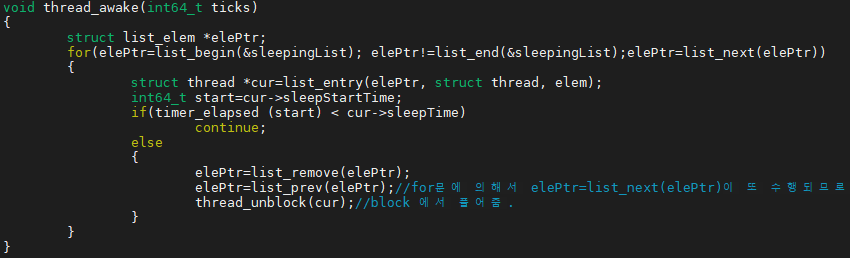


Threads/thread.c의 thread\_sleep함수

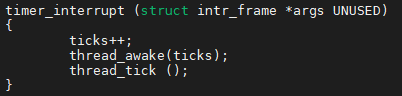


Devices/timer.c의 timer\_sleep함수

이어서 threads/thread.c에 thread\_sleep이라는 함수를 만들고 이 함수에서 start로부터의 걸린 시간을 체크해주는 함수인 timer\_elapsed함수를 이용해 시간을 체크해주어 ticks만큼의 시간이 지나지 않았다면 위와 같이 block될 thread의 sleepStartTime과, sleepTime을 업데이트해주고 sleepingList에 넣어주도록 하였다. 그리고 이를 devices/timer.c의 timer\_sleep함수에서 호출하여 주도록 하였다.



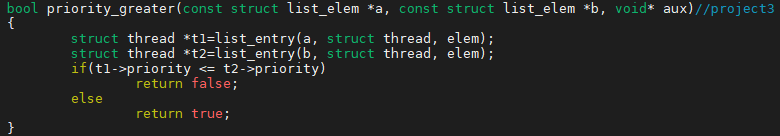
Threads/thread.c의 thread\_awake함수



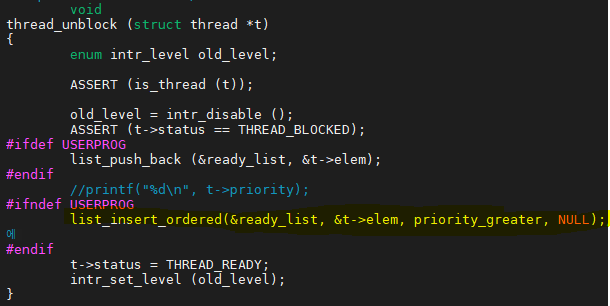
Devices/timer.c의 timer\_interrupt함수

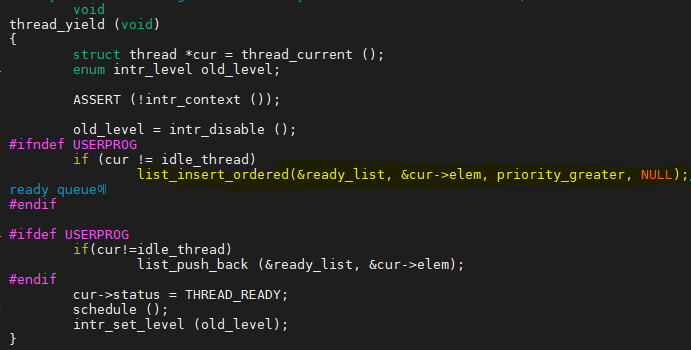
마지막으로 위와 같이 threads/thread.c에 thread\_awake라는 함수를 만들고 이 함수에서 sleepingList를 순회하며 block된 thread들이 일어날 시간이 지났는지를 체크하도록 하여 일어날 시간이 된 thread들을 unblock해주고 sleepingList에서 제거해 줄 수 있도록 하였다. 또한 매 틱마다 이를 체크해 주어야 하므로 이 thread\_awake함수를 매 틱마다 호출되는 devices/timer.c의 timer\_interrupt함수에서 호출하여 주도록 하였다.

* Priority Scheduling

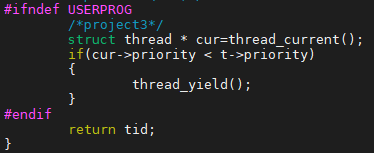


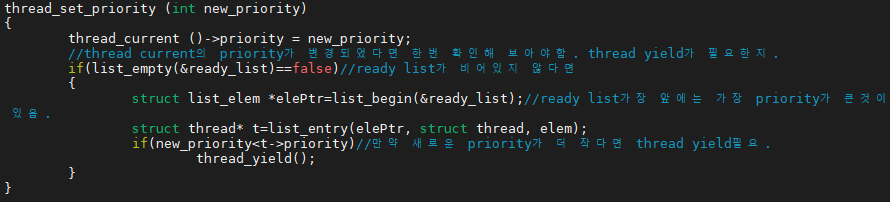
먼저 위와 같이 threads/thread.c에 두 list\_element를 받아 thread로 바꾸어 priority를 비교하여 주는 priority\_greater함수를 만들었다.



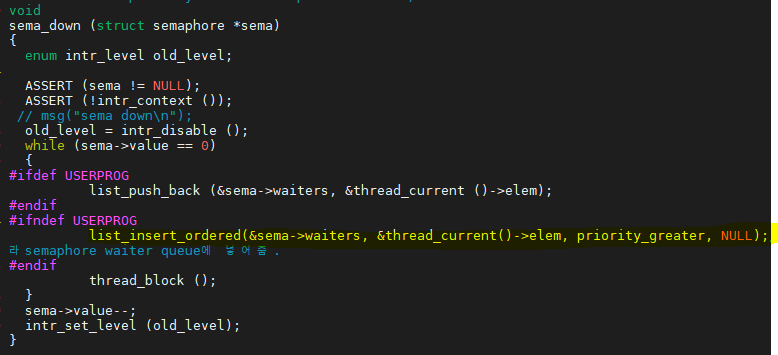


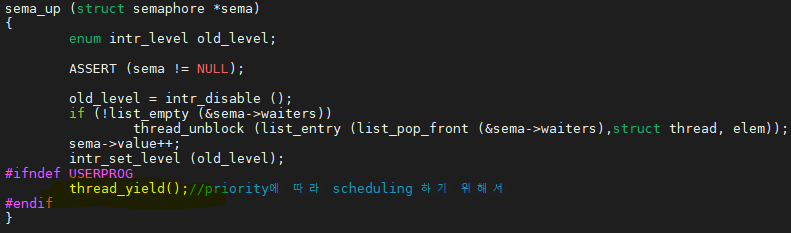
그리고 ready\_list가 항상 priority에 따라 정렬되어 있도록 하기 위하여 ready\_list에 thread를 추가하는 threads/thread.c의 thread\_unblock과 thread\_yield함수에서 위와 같이 list\_insert\_ordered함수를 이용하도록 하여 thread가 priority에 따라 내림차순으로 정렬된 채로 ready\_list에 들어갈 수 있도록 하였다.

Threads/thread.c의 thread\_create함수의 일부

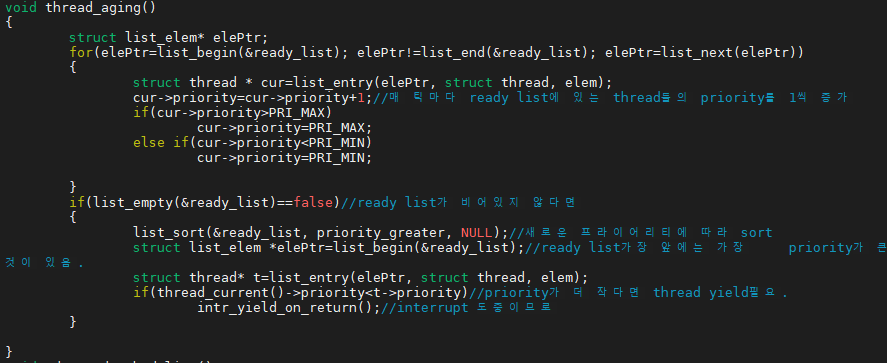


또한 위와 같이 threads/thread.c의 thread\_create함수의 마지막 부분에 current\_thread와 새로 만들어진 thread의 priority를 비교하는 코드를 넣어주고 또 thread\_set\_priority함수에서는 현재 thread의 priority를 새로운 priority로 설정해주고 이를 ready\_list의 가장 첫 thread와 비교해주는 코드를 넣어주어 새로 만들어진 thread 또는 ready\_list의 첫 thread의 priority가 더 높다면 current\_thread가 yield할 수 있도록 하는 코드를 추가해주어 preemptive하게 작동할 수 있도록 하였다.



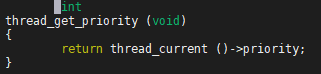


그리고 위와 같이 threads/synch.c의 sema\_down함수에서 semaphore의 waiters에 thread가 추가될 때 list\_insert\_ordered함수를 이용하여 thread를 추가하도록 하여 waiters가 thread의 priority에 따라 정렬되어 있도록 하였고 sema\_up함수에서는 마지막에 thread\_yield함수를 호출해 주도록 하여 waiters에서 나와 ready\_list에 들어간 thread와 현재 thread가 preemptive하게 작동될 수 있도록 하였다.



thread\_tick함수에서 thread\_aging 호출

마지막으로 priority가 낮은 thread들의 starvation이 일어나지 않도록 threads/thread.c에 위와 같이 ready\_list안의 모든 thread들의 priority를 1씩 증가시켜주는 thread\_aging함수를 만들었고 thread\_tick함수에서 thread\_prior\_aging이 true이면 이 함수를 호출하도록 하여 매 틱마다 aging이 일어날 수 있게 하였다. 또한 thread\_aging함수의 마지막 부분에 ready\_list의 첫 thread와 current\_thread의 priority를 비교하고 yield할 수 있게 하여 aging후에 preemptive하게 작동될 수 있도록 하였다.



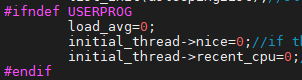
thread\_get\_priority함수는 위와 같이 현재 thread의 priority를 반환할 수 있도록 하였다.

* Advanced Scheduler

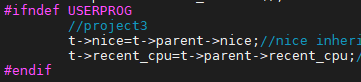
threads/thread.h thread구조체의 일부

먼저 각각의 thread가 자신의 nice값과 recent\_cpu값을 가지고 있을 수 있도록 위와 같이threads/thread.h의 thread구조체에 nice와 recent\_cpu변수를 추가하여 주었다.

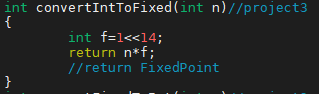
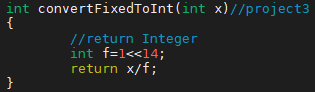
threads/thread.c의 load\_avg변수 선언 부

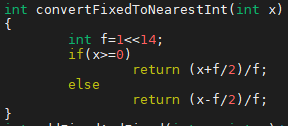
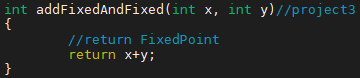
thread\_init함수의 일부

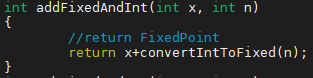
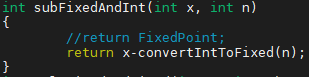
또한 위와 같이 threads/thread.c에 load\_avg라는 static 전역 변수를 선언하여 주고 thread\_init함수에 이러한 load\_avg변수와, 초기 생성되는 thread의 nice값과 recent\_cpu값을 0으로 초기화해 주는 코드를 추가해 주었다.

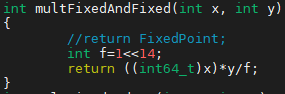
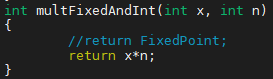
thread\_create함수의 일부

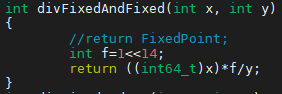
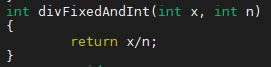
그리고 새로 생성되는 thread의 nice값과 recent\_cpu값의 경우에는 parent의 nice값과 recent\_cpu값을 상속받기 때문에 위와 같이 thread\_create함수에서는 새로 생성된 thread가 부모의 nice값과 recent\_cpu값을 가질 수 있도록 하였다.

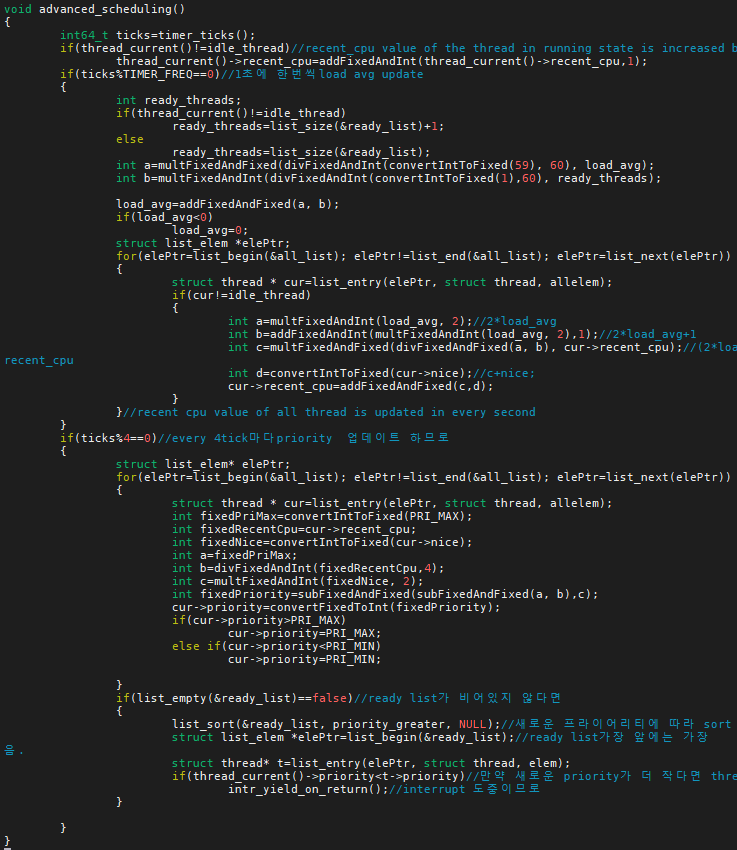
 

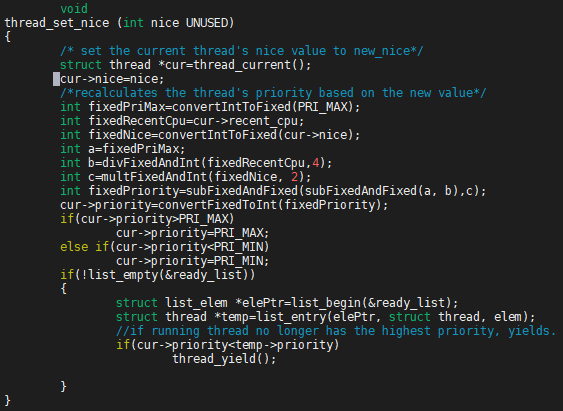
그리고 핀토스 매뉴얼을 참고하여 위와 같이 Fixed-Point연산을 할 수 있도록 하는 함수들을 만들어 threads/thread.c에 추가하여 주었다.

threads/thread.c thread\_tick함수의 일부

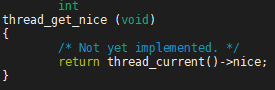
이어서 thread\_tick함수에서 thread\_mlfqs가 true일 때 advanced\_scheduling함수를 호출해주도록 하여 이 경우에 advanced scheduler를 사용할 수 있도록 하였다.



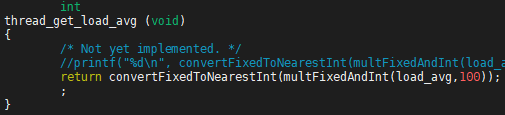
마지막으로 위와 같이 advanced\_scheduling 함수를 만들고 위에서 구현한 fixed point연산을 이용하도록 하였는데, 이 advanced\_scheduling함수는 매 틱마다 호출되므로 매 틱마다 idle thread가 아닌 running상태에 있는 thread의 recent\_cpu값을 1씩 증가시켜주도록 하였고, 또 ticks%TIMER\_FREQ를 이용하여 1초에 한번씩 load\_avg를 공식에 따라 업데이트 할 수 있도록 하였다. 또한 recent\_cpu공식에 따라 모든 thread들이 1초에 한 번씩 recent\_cpu를 재계산할 수 있도록 하였고 ticks%4를 이용하여 매 4틱마다 모든 thread들의 priority가 공식에 따라서 재계산이 되도록 하고 이렇게 재계산된 priority에 따라 ready list를 sort후 current thread와 ready\_list의 가장 처음에 있는 thread의 priority를 비교하고 yield하도록 하여 preemptive하게 작동될 수 있도록 하였다.



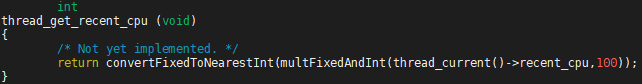
thread\_set\_nice함수에서는 위와 같이 current thread의 nice값을 새로 들어온 nice값으로 바꾸어 주도록 하고, 이렇게 바뀐 nice값을 기준으로 priority를 재계산한 후에, ready\_list의 가장 첫 thread와 priority를 비교하고 yield하여 preemptive하게 작동할 수 있도록 하였다.



thread\_get\_nice함수에서는 위와 같이 current thread의 nice값을 반환하여 주도록 하였다.

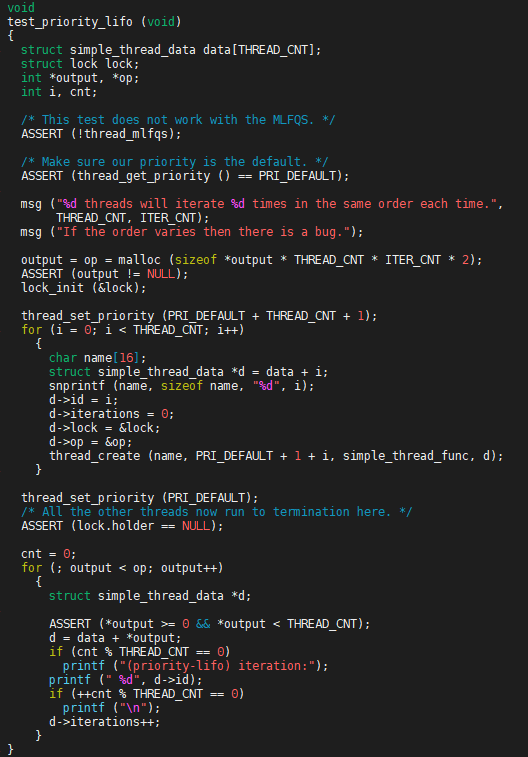


Thread\_get\_load\_avg함수에서는 위와 같이 load\_avg값에 100을 곱한 후에 이 fixed point로 표현된 수를 nearest integer형태로 바꾸어 반환하도록 하였다.

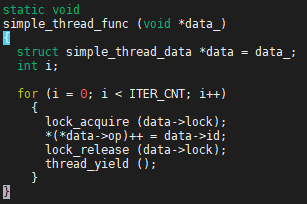


Thread\_get\_recent\_cpu 함수에서도 위와 같이 current thread의 recent\_cpu값에 100을 곱한 후에 이 fixed point로 표현된 수를 nearest integer형태로 바꾸어 반환하도록 하였다.

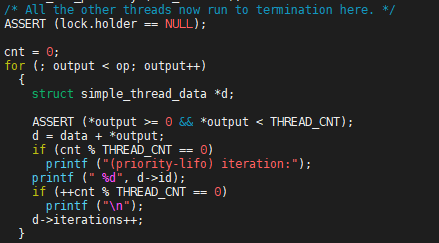
* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석



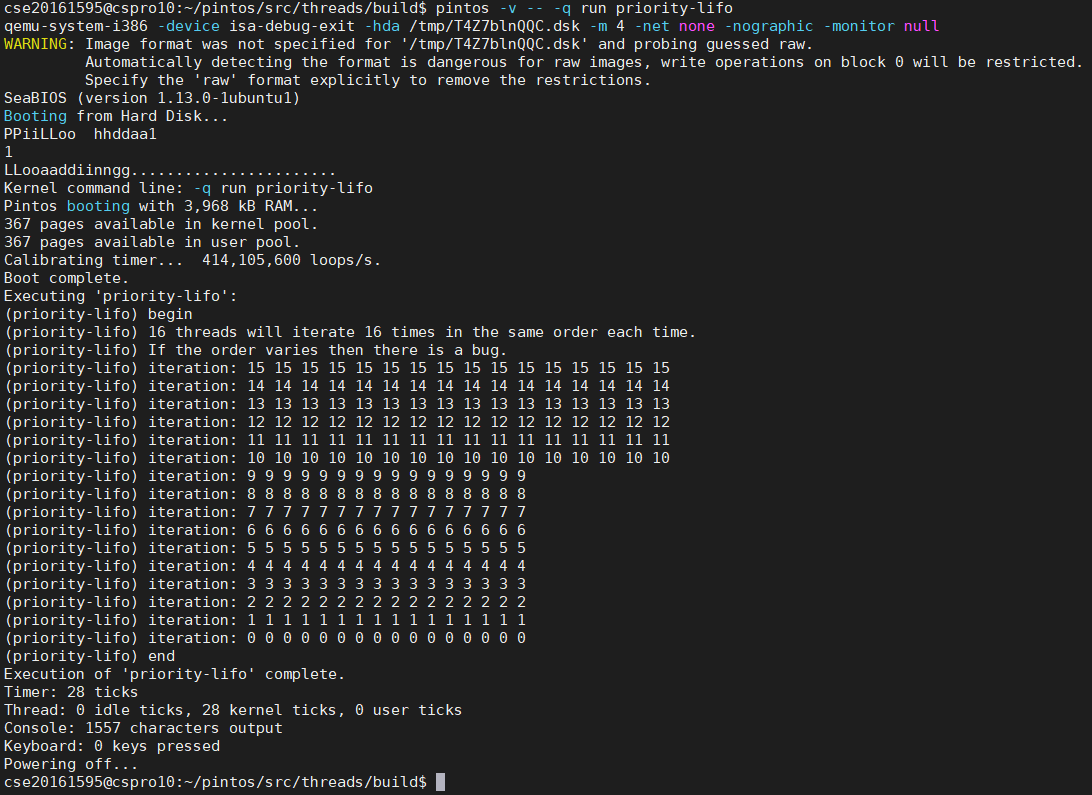
Priority-lifo.c의 코드를 보게 되면 위와 같이 먼저 초기 thread의 priority를 PRI\_DEFAULT\_THREAD\_CNT+1로 설정해줌으로써 초기 thread의 priority가 앞으로 만들 thread들 보다 높도록 되어 있다. 따라서 초기 thread가 THREAD\_CNT개수만큼의 thread들을 연속하여 만들어 ready queue에 넣게 된다는 것을 알 수 있다. 이 때 만들어진 thread들은 PRI\_DEFAULT+1+i의 priority와 i라는 data->id를 갖게 되는데 이는 늦게 만들어질수록 thread priority가 높다는 것을 의미하고 각 thread들의 data->id는 0~15값을 가진다는 것을 의미한다(가장 마지막에 만들어진 thread data->id=15). 따라서 나중에, 이렇게 생성된 thread들이 scheduling 될 때 늦게 생성되어 늦게 ready queue에 들어간 thread들부터 scheduling되어 실행되어야 하는 것을 알 수 있다. 그리고 이렇게 thread들을 16개를 만든 이후에, 초기 thread의 priority를 다시 PRI\_DEFAULT로 set함으로써 초기 thread의 priority가 가장 낮아지게 되고 ready queue에 가장 마지막에 들어간 thread에 의하여 preemption되게 된다.



따라서 가장 마지막에 생성된 thread의 priority가 가장 높으므로 이 thread부터 가장 처음에 생성된 thread(초기 thread를 제외하고는 priority 가장 낮음)까지의 순서(만들어진 순서의 역순)대로 수행이 되게 되는데 각각의 thread들은 위의 simple\_thread\_func()을 수행하게 된다. 그리고 simple\_thread\_func()이 수행됨에 있어서 ITER\_CNT(16번)만큼 반복을 하게 되고, 반복되면서 output buffer에 해당 thread의 data id(몇 번째로 생성된 thread인지)를 쓰게 된다. 이후 반복함에 있어 매번 thread\_yield를 호출하지만 수행되고 있는 thread가 그 시점에 있어서 가장 우선순위가 높은 thread이기 때문에 해당 thread가 다시 scheduling되어 수행되므로 해당 thread가 끝나기 전까지는 thread\_yield를 호출하더라도 해당 thread가 진행되고 따라서 output buffer에는 position의 위치를 옮겨가면서 해당 thread의 data id가 ITER\_CNT만큼(16번) 써지게 된다. 즉 최종적으로 output buffer에는 가장 마지막에 생성된 thread의 data->id(15)가 16번씩 써진 것부터 시작해서 가장 처음에 생성된 thread의 data->id(0)가 16번 써져 있게 된다(15(16번), 14(16번), 13(16번),…, 2(16번), 1(16번), 0(16번).



이렇게 생성된 모든 thread가 수행이 끝나게 되면 다시 초기 thread가 scheduling되어 위의 내용을 진행하게 되는데 여기서 output buffer를 통해 수행된 순서, iteration에 따른 thread의 data->id를 출력하게 된다.



실행 결과인 위의 내용을 보게 되었을 때, output buffer를 통해 수행된 순서, iteration에 따른 thread의 data->id가 찍힌 부분을 보면 나중에 생성되어 ready queue에 들어간 thread(15)부터 처음에 생성된 thread(0)까지의 순서대로, 즉 Last In First Out의 순서대로 잘 실행이 되었다는 것을 확인 할 수 있다.

* make check 수행 결과를 캡처하여 첨부

