**Pintos Project 3: Threads**

담당 교수 : 김영재

조 / 조원 : 20161640 / 정동혁

개발 기간 : 2020/11/16~2020/12/03

1. **개발 목표**

이번 프로젝트의 목표로는 크게 3가지가 있다. 첫 번쨰는 현재 busy waiting방식으로 수행되는 것을 non-busy waiting방식으로 바꿔주는 것이고, 두 번째는 현재 RR(round-robin)방식으로 수행되고 있는 scheduling방식을 priority을 사용하는 scheduling방식으로 바꿔주는 것이다. 마지막은 추가구현으로, BSD Scheduler를 구현하는 것이다

1. **개발 범위 및 내용**
   1. **개발 범위**

**1). Alarm Clock**  
현재 제공받은 pintos는 busy-waiting방식으로 while loop를 통해 무한정으로 다음 차례를 기다리고 있는 방식으로 수행되고 있는데, alarm clock을 구현해주면 매tick마다 깨어나야 할 thread를 찾아보며 unblock을 해주게 되는 효율적인 구조로 수행된다.

**2). Priority Scheduling**  
현재 pintos내에서 scheduling을 round-robin방식으로 수행되고 있는데, 이 방식은 priority를 전혀 고려하지 않은 방식이므로 priority scheduling을 구현해주면 ready list내부에 우선순위가 높은 thread가 앞에 오게 되고, 현재 CPU를 점유중인 thread를 선점할 수 있게 된다.

**3). BSD Scheduler**이번 프로젝트에서 우리가 구현할 추가적인 scheduler는 Multi-Level Feedback Queue와 같이 동작한다. BSD Scheduler의 구현을 마치게되면 pintos상에서 command에 -mlfqs가 포함된다면 Multi-Level\_Feedback\_Queue방식으로 scheduling이 수행되게 된다.

* 1. **개발 내용**

**1). Alarm Clock**  
우선적으로 thread의 sleep, awake를 위해서 struct thread내부에 wakeup\_t이라는 element를 추가해준다. 이는 한 thread가 깨어나야할 time tick을 저장하는 element이다. 이후 전역변수로 next\_awake를 만들어준다. 이는 가장먼저 일어나야 할 thread의 wakeup\_t값을 지니고 있다. 이후 src/threads의 thread.c에 sleep\_thread() function을 만들어준뒤 내부에서 interrupt를 꺼주고, thread의 wakeup\_t와 next\_awake를 set해주고 sleep list에 추가해주며, thread\_block()을 통해 thread를 blocking한다. 이후 interrupt를 다시 켜준다. 이후에 src/thread의 thread.c에 awake\_thread() function을 만들어준뒤 내부에서 sleep\_list를 순회하며 현재 time\_tick보다 깨어나야할 time\_tick이 작거나 같은 thread들을 thread\_unblock() function을 통해 blocking을 해제해준뒤 next\_awake를 갱신해주고, sleep\_list에서 제거해준다.

**2). Priority Scheduling**  
만약 현재 ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 기존의 그저 ready list의 맨 뒤에 삽입하는 것이 아닌 맨 앞에 삽입될 수 있도록 하며, preemption이 가능하도록 설계해야 한다. 이를 위해서 priority를 고려한(priority value가 클수록 우선순위 높음) thread간 정렬기준을 함수로 작성하고, 기존에 src/threads에 구현되어 있던 thread\_unblock() function과 thread\_yield() function내부의 list\_push\_back함수를 앞서 언급한 정렬기준을 사용하는 list\_insert\_ordered() function으로 바꿔준다. 이를 통해 매번 unblock, yield시마다 ready\_list의 priority가 보장된다. 이후 preemption을 위해 기존에 구현되어 있던 thread\_create() function내부에서 마지막에 unblock이후 새로만든 thread의 priority가 기존에 CPU를 점유하던 thread보다 priority가 높다면 thread\_yield() function을 통해 양보를 하게 만들어주고, 현재 thread간의 priority를 set해주는 thread\_set\_priority() function도 새로운 priority가 실행되던 priority값보다 작아지면 thread\_yield() function을 통해 양보를 하게 만들어준다.

**3). BSD Scheduler**  
추가적인 구현인 BSD Scheduler에서의 priority를 계산하기 위해서는 struct thread내부에 추가적으로 nice, recent\_cpu라는 element가 요구된다. 여기서 nice는 priority에 영향을 주는 element로, -20부터 20사이의 값을 가지며 양의 값을 가질 경우 다른 thread가 CPU를 차지할 수 있도록 한다. thread생성시에는 0으로, 혹은 부모 thread의 값을 물려받는다.  
recent\_cpu란 thread의 cpu\_time값이다. 즉, 해당 thread가 cpu를 점유하고 있던 시간을 의미한다. 이러한 recent\_cpu value는 load\_avg값과 특정한 수식을 통해 계산할 수가 있는데, 여기서 load\_avg란 read\_list에 들어가있는 thread의 개수를 의미한다. 이러한 load\_avg는 0으로 initial되며 매초마다 특정 수식을 통해 갱신이 되어진다. recent\_cpu값 또한 thraed생성시에는 0으로, 혹은 부모 thread의 값을 물려받으며 매 tick마다 1이 증가하게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| **날짜** | **구현 내용** |
| **2020/11/16** | **ppt 및 온라인강의 시청** |
| **2020/11/18** | **struct thread에 wakeup\_t element를 추가한뒤, sleep\_thread, awake\_thread function을 구현 및 timer.c의 timer\_sleep, timer\_interrupt function에서 호출, alarm관련 test case전부 pass** |
| **2020/11/23** | **thread간 priority비교를 위한 cmpr function, 이를 이용하여 list\_push\_back함수를 list\_insert\_ordered로 대체, priority scheduling구현.** |
| **2020/11/29** | **priority aging을 위해 -mlfqs, -aging case를 구분가능하도록 틀 완성** |
| **2020/11/30** | **BSD scheduler를 위해서 element추가뒤, ppt의 공식을 참고하여 함수제작 및 timer\_interrupt function완성, 모든 test case통과** |
| **2020/12/03** | **보고서 작성** |

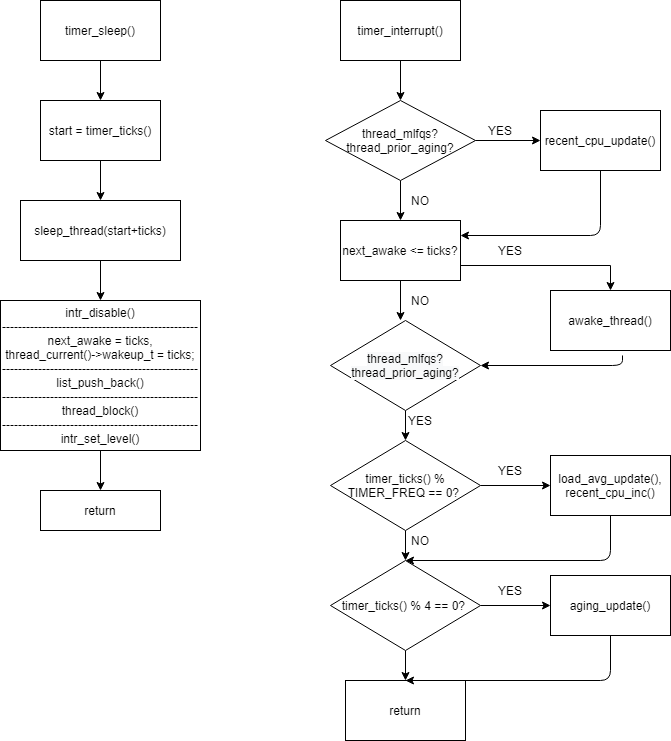
* 1. **개발 방법**

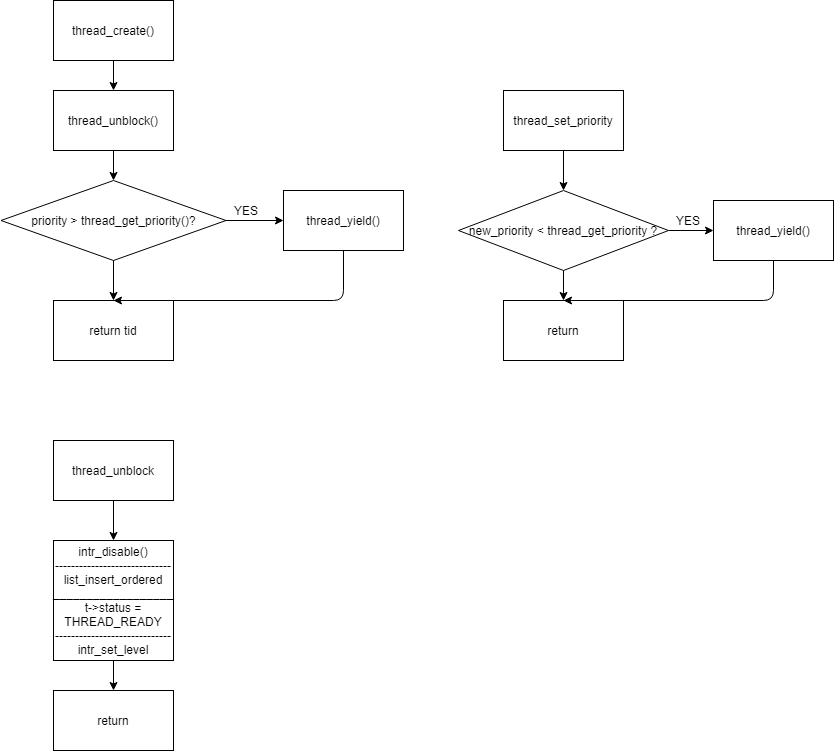
**1). Alarm Clock**우선적으로 Alarm Clock을 구현하기 위해서는 /src/threads의 thread.h에 존재하는 struct thread에 wakeup\_t element를 추가해줘야 한다. 이는 해당 thread객체의 wake up time tick값을 의미한다. 또한 현재 자고있는 thread들을 관리할 수 있는 수단이 존재하지 않으므로 새로운 list sleep\_list를 추가해준다. 이후 thread\_init() function내부에서 list\_init(&sleep\_list)를 통해서 list를 초기화 해준다.   
thread의 sleep과 awake를 위해서는 thread.c내부에 sleep\_thread, awake\_thread function을 만들어주었다. 두 함수의 동작은 앞서 개발내용에서 언급하였다. 이렇게 두 함수를 구현한 뒤에는 현재 busy-waiting방식으로 동작하고있는 /src/devices의 timer.c의 timer\_sleep function을 수정해줘야 한다. 여기서는 기존의 while loop을 제거해준뒤 이전에 구현했던 sleep\_thread(start+ticks)를 호출함으로써 효율적인 방식으로 작동하게 한다.

**2). Priority Scheduling**priority scheduling을 위해서는 현재 ready\_list에 thread가 쌓이는 방식을 수정해줘야 한다. 이를 구현하기 위해서는 기존의 방식처럼 list\_push\_back을 사용하는 것이 아닌, list\_insert\_ordered function을 사용해야 하는데 이 function을 사용하기 위해서는 thread만의 정렬기준이 필요하다. 이를 위해 bool cmpr이라는 function을 src/threads의 thread.c에 만들어주고, 1st thread객체의 priority element가 크면 true, 아니면 false를 반환해주는 구조로 만들어준다. 이후 thread\_unblock() function과 thread\_yield() function내부의 list\_push\_back을 전부 list\_insert\_ordered function으로 대체해줌으로써 priority가 보장된 ready\_list를 구현할 수 있다.   
또한 preemption을 위한 처리가 필요한데, 이는 thread\_create() function, thread\_set\_priority function을 수정해주면 된다. 만약 지금 CPU를 차지하고 있는 thread의 priority값이 정렬된 ready\_list의 첫 번쨰 thread의 priority값보다 작으면 CPU를 양보해야하므로 thread\_create()의 마지막 부분에서는 현재 실행중인 thread의 priority와 parameter로 받은(만들어준 thread의) priority를 비교해서 보다 작으면 thread\_yield() function을 통해 CPU를 넘겨주게 만들어준다. thread\_set\_priority function에서는 priority를 바꿔준 뒤에 바꿔준 값이 이전의 priority값보다 작을 경우 thread\_yield를 통해 CPU를 넘겨주게 된다.  
이외에도 sema\_down, sema\_up에 대한 추가적인 구현이 필요한데, sema\_down에서는 waiter list에 block된 element를 넣어줄 때 priority를 고려해서 넣어줘야 하므로 list\_insert\_ordered function을 사용한다. sema\_up에서는 waiter list에서 가장 큰 priority를 갖는 thread를 찾아내기 위해 waiter list를 loop를 통해 돌면서 가장 높은 priority의 thread를 찾아내고 unblock을 해준다.

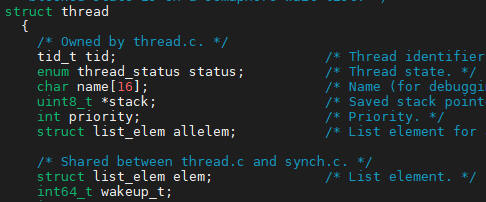
**3). BSD Scheduler**BSD방식의 scheduler에는 앞서 언급하였듯이 thread객체에 nice, recent\_cpu element의 추가가 필요하다. 이외에는 load\_avg변수가 필요한데, 이는 thread에 귀속된 element가 아닌 매초마다 갱신되는 공용변수이기 때문에 /src/threads의 thread.c에 전역변수로 선언을 해주었다. 또한 ppt의 pintos의 숫자계산방식을 위해 FRACTION (1<<14)이라는 수도 thread.h에 #define 해주었다. 이후 load\_avg를 1초마다 업데이트해주는 load\_avg\_update, recent\_cpu값을 매 tick마다 증가시켜주는 recent\_cpu\_inc, recent\_cpu값을 update해주는 recent\_cpu\_update, BSD방식의 priority를 계산해주는 aging\_update function등을 thread.c내부에 만들어주었다.   
추가적으로 이미 thread.c에 함수원형만 구현되어있는 thread의 priority를 set해주는 thread\_set\_priority, 현재 수행중인 thread의 priority값을 return해주는 thread\_get\_priority, 입력받은 nice값으로 현재 thread의 nice값을 바꿔주는 thread\_set\_nice, 현재 thread의 nice값을 return해주는 thread\_get\_nice, 현재system의 load\_averge값의 100배의 가장인접한 정수값을 return해주는 thread\_get\_load\_avg, 현재 recent\_cpu값의 100배의 가장 인접한 정수값을 return해주는 thread\_get\_recent\_cpu function또한 구현해주었다.

1. **연구 결과**
   1. **Flow Chart**

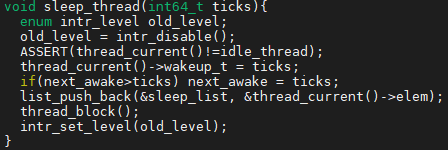
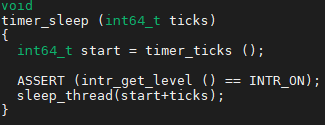
 alarm clock

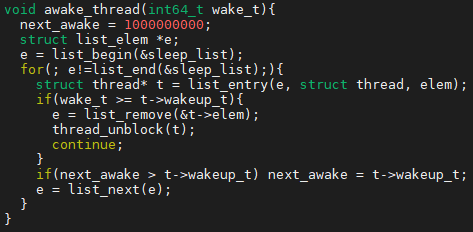
****priority scheduling

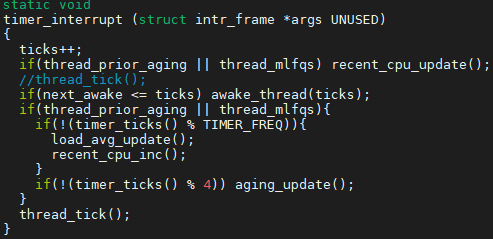
* 1. **제작 내용**

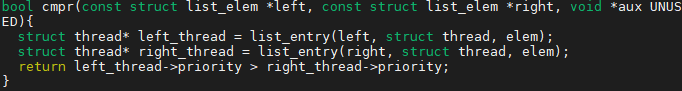
**1). Alarm Clock**  
위에서 언급한대로 다음과 같이 struct thread에 wakeup\_t element를 추가해주었다.  


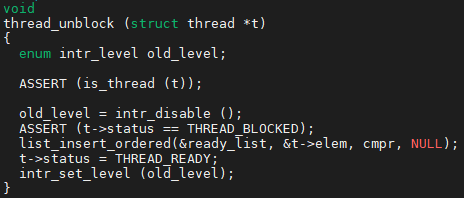
이후 다음과 같이 sleep\_list를 thread.c에 만들어주고 init\_thread function내부에서 list\_init을 통해 초기화를 진행해주었다.  
  

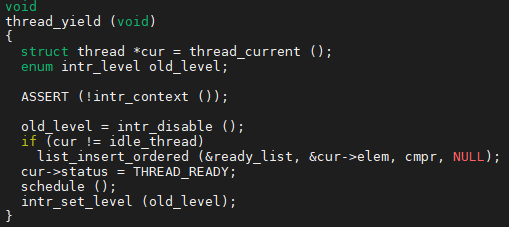

이후 다음과 같은 sleep\_thread function을 만들어주었다.  
  
이 함수의 동작과정을 살펴보면 먼저 interrupt를 막아준뒤(atomic한 operation처럼 진행되기 위하여) 현재 thread의 wake time에 입력받은 ticks를 저장시켜준다. 그리고 처음에는 0으로 초기화된 다음에 깨어나야할 thread의 wake up time(next\_awake)보다 현재 입력받은 ticks이 더 작은 값을 가지게 된다면 그 녀석을 다음에 깨어나야할 thread로 만들어준다. 이후 sleep\_list에 그러한 thread를 추가해주고 thread\_block()을 통해 잠근다. 이후 다시 interrupt를 허용해준다.  
위와 같은 함수를 완성한 뒤에 다음과 같이 /src/devices의 timer.c의 timer\_sleep function에서 호출해주었다.  
  
이를 통해 thread가 sleep\_list에 들어가는 과정을 전부 구현하였다.

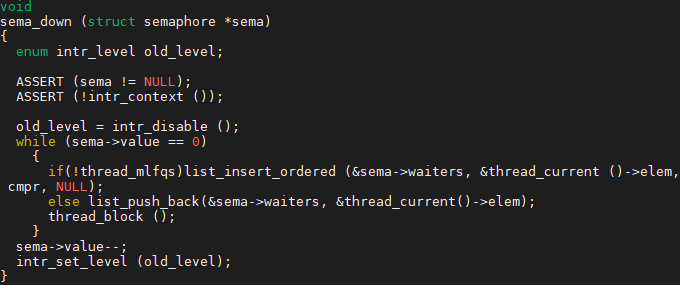
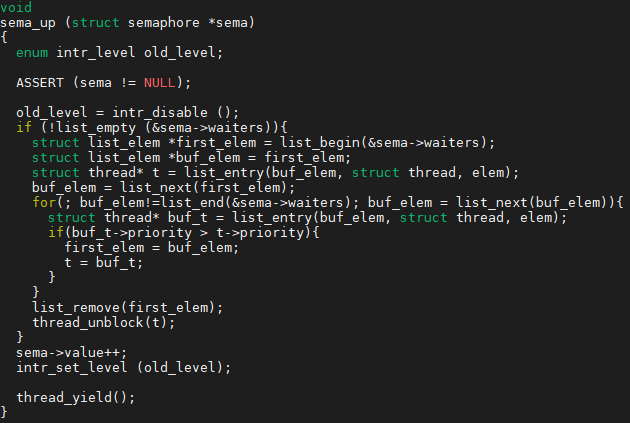
이후에 구현할 사항은 자고있는 thread를 깨우는 것이다. 이는 void awake\_thread function을 통하여 구현하였다. awake\_thread function은 다음과 같다.  
  
이 함수의 동작과정을 살펴보면 먼저 next\_awake의 값을 엄청나게 크게 설정한다. 이후 잠들어있는 thread들이 보관된 sleep\_list를 for loop를 통해 순회하며 각 thread의 일어나야할 시간이 현재 time tick보다 작으면 해당 thread의 block을 해제해주고 sleep\_list에서 제거해준다. 또한 for loop를 통해 가장작은 wake up time을 찾아내 next\_awake변수또한 설정해준다.   
위와 같은 함수를 완성한 뒤에 다음과 같이 /src/devices의 timer.c의 timer\_interrupt function에서 호출해주었다. (일단은 if(next\_awake<=ticks) awake\_thread(ticks) line만 살펴보겠다)

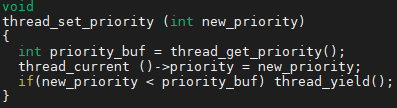
  
위의 함수를 보면 알 수 있다시피 만약 다음 thread가 깨어나야할 시간이 현재 time tick보다 작으면 awake\_thread(ticks) function을 통해 해당 thread를 깨워준다.

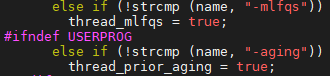
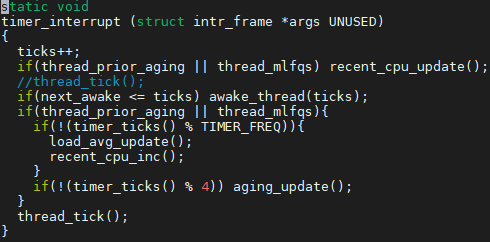
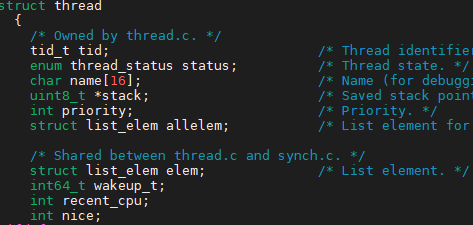
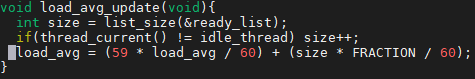
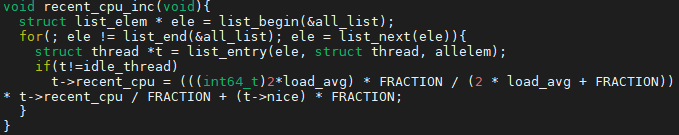
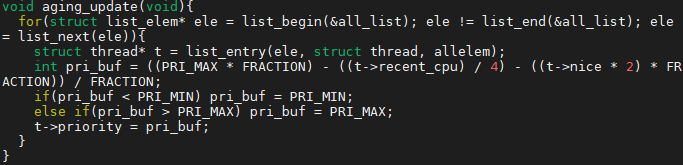
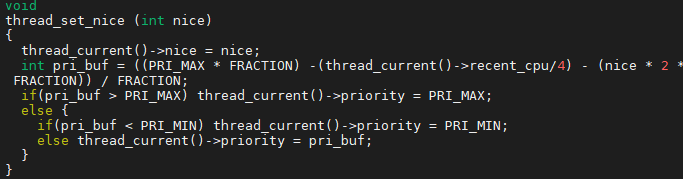
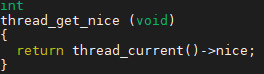
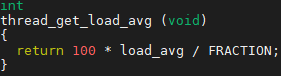
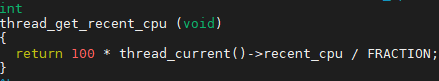
**2). Priority Scheduling**우선적으로 list에 thread를 넣을 때 thread객체의 element인 priority가 가장 높은 순서대로 list에 들어가야 하므로 list\_insert\_ordered function을 사용해줘야 한다. 이를 위해서는 thread객체간 대소비교를 해줘야한다. 그리하여 다음과 같은 cmpr function을 구현하였다.  
  
이러한 cmpr함수를 통해 list\_insert\_ordered사용시 priority가 높은 thread순서로 list에 들어가게 된다.

이후 다음과 같이 thread\_unblock함수의 list\_push\_back을 list\_inser\_ordered로 바꿔줌으로써 thread\_unblock시 priority에 맞게 ready\_list에 들어가게 된다.  


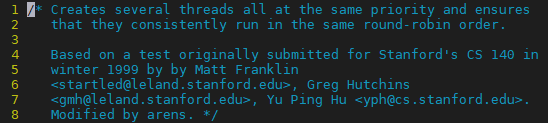
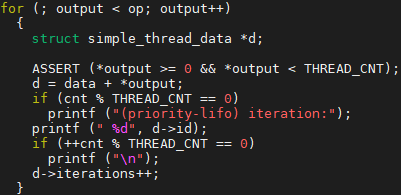
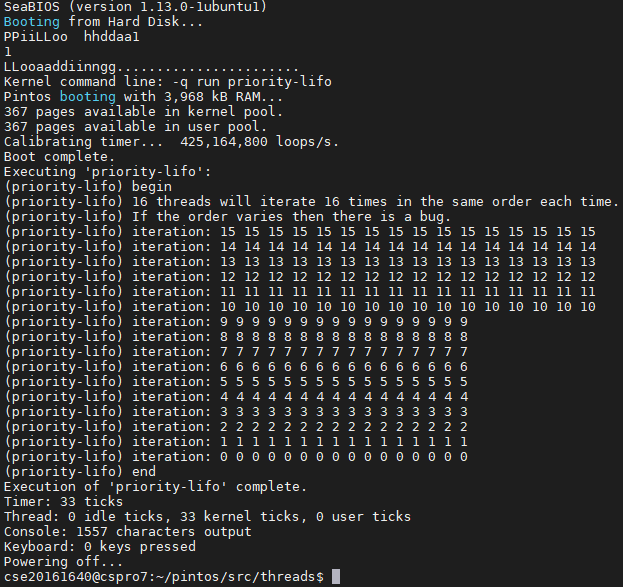
또한 이미 구현되어 있는 thread\_yield함수에서도 ready\_list에 thread가 들어갈 때마다 list\_insert\_ordered를 통해 priority에 맞게 ready list에 넣어준다.  


순서에 맞게 thread를 관리하는 처리부분은 /src/threads의 synch.c의 sema관련 function에서도 중요하다.   
sema\_down의 경우에도 waiter list에 priority에 맞게 insert를 해줘야 하므로 다음과 같디 sema->value가 0일 때 list\_insert\_ordered를 통해 순서에 맞게 waiter list에 넣어주고 thread\_block을 통해 thread를 block시킨다.  
  
sema up의 경우에도 현재 waiter list에서 가장 priority가 높은thread를 unblock시켜줘야 하므로 for loop를 통해 waiter list를 순회하며 가장 큰 priority를 가진 thread를 찾은 뒤 waiter list에서 제거해주고, 해당 thread를 unblock시켜준다.  


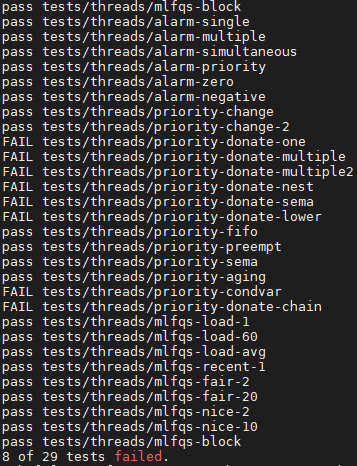
이후에는 priority가 높은 thread의 preemption을 위해 thread\_create, thread\_set\_priority function의 수정이 필요하다.  
우선적으로 thread\_create를 살펴보면 thread를 만들어줄때마다 만들어준 thread의 priority값이 현재 CPU를 선점하고 있는 thread보다 큰 값을 가질 경우 앞서 구현한thread\_yield를 통해 CPU를 넘겨준다.  
 thread\_create  
이후 한 thread의 priority값을 set해주는 thread\_set\_priority function에서도 현재 thread의 priority를 set해줬을 때 만약 그 값이 이전의 priority보다 작아졌을 경우 thread\_yield를 통해 CPU를 넘겨준다.  
   
priority에서 중요한 점은 aging개념도 존재하는데 이는 아래의 BSD scheduler와 연관지어 언급하겠다.

**3). BSD Scheduler**우선적으로 BSD Scheduler를 위해서는 priority aging이 구현되어 있어야 한다. 이를 위해 ppt에 명시된대로 thread.h에 extern bool thread\_prior\_aging을 추가해 aging기법을 활용할 건지 정해준다.  
이후 /src/threads의 init.c의 parse\_options function에서 thread\_prior\_aging case를 추가해준다.  
  
결국 pintos상에서 command에 -aging이 들어가면 aging기법을 활용하여 priority를 고려하겠다는 의미이고, 우리가 추가로 구현해야하는 Scheduler또한 aging기법을 활용하므로 다음과 같이 timer\_interrupt function을 구현할 수 있다.  
  
여기서 사용된 부가적인 함수들은 아래에서 언급한다.  
추가적인 scheduler를 구현하기 앞서 thread에 새로운 element들을 추가해줘야하는데 바로 nice와 recent\_cpu이다. 이들은 priority를 계산할 떄 사용된다.  
  
ppt에 명시된대로 이들은 init시 0으로 초기화 해준다.  
이외에도 read\_list에 들어가 있는 thread의 개수를 알기위한 변수(load\_avg)가 필요한데, 이는 공용변수이므로 thread.c에 전역변수로 선언해준다.  
  
이 값 또한 초기에는 0의 값을 가지고 있다.  
이제 ppt에서 사용된 공식들을 활용하기 앞서 FRACTION이라는 전역변수가 필요하므로 이또한 #define을 통해 thread.h에 선언해준다.  
  
제일 먼저 ppt의 load\_avg를 update하는 공식을 살펴보면 다음과 같다.  
  
그러나 위와 같이 C코드를 작성할 경우 버림, 반올림등의 문제가 발생하였다. 이를 해결하기 위해 먼저 분자를 뒤의 큰값(load\_avg, ready\_threads)에 곱해준뒤에 나중에 60으로 나눠줌으로써 우리가 원하는 load\_avg값을 얻어낼 수 있었다.  
  
다음으로 recent\_cpu값을 증가시키는 공식을 구현하였는데 공식을 살펴보면 다음과 같다.  
  
이는 이전과 같은 버림, 반올림 문제가 없어보이지만 real number problem이 존재한다. 이를 위해 우선적으로 load\_avg앞의 2의 값을 int64\_t로 더 큰 자료형으로 형변환을 해준뒤에 FRACTION값을 곱해주었고, 분모에 존재하는 값중에 1도 FRACTION을 곱해줌으로써 pintos에서의 위의 수식을 구현하였다. 이후 뒤에 곱해지는 recent\_cpu를 FRACTION으로 나눠주고, 더해지는 nice의 값에 FRACTION을 곱해줌으로써 pintos의 real number problem을 해결해주었다.  
  
또한 recent\_cpu값도 주기적으로 update가 필요한데 이는 recent\_cpu에 FRACTION을 더해주는 방식으로 구현된다.  
  
마지막으로 위의 값들을 사용하여 thread의 priority를 구해줘야하는데 그 공식은 다음과 같다.  
  
여기서 사용된 것중에 FRACTION처리가 필요없는 경우는 real\_number/int case뿐이므로 recent\_cpu/4를 제외하고 전부 FRACTION을 곱해준뒤 마지막에 FRACTION으로 나눠준다.  
만약 이렇게 구한 priority값이 PRI\_MIN보다 작은 경우, PRI\_MIN으로 set해주고, PRI\_MAX보다 큰 경우, PRI\_MAX로 set해준다. 만약 괜찮은 값인 경우 thread의 priority로 set해준다.  
  
추가구현인 test case들을 통과하기 위해서는 추가적인 function들의 구현이 필요한데 이는 현재 프로토타입으로만 존재하는 thread\_set\_nice, thread\_get\_nice, thread\_get\_load\_avg, thread\_get\_recent\_cpu등이 있다.  
우선적으로 thread\_set\_nice부터 살펴보면, 입력받은 new\_nice값으로 현재 수행중인 thread의 nice값을 바꿔주고 aging\_update함수와 동일하게 aging과정을 진행한다.  
  
thread\_get\_nice함수는 현재 실행중인 thread의 nice값을 반환하는 형태로 구현하였다.  
  
thread\_get\_load의 경우 test case수행시 ppt에 명시된대로(소수점 두자리까지 비교를 해야한다) 현재 load\_avg값에 100을 곱해주고 FRACTION으로 나눠준뒤 반환해준다.  
  
thread\_get\_recent\_cpu또한 위의 경우와 마찬가지로 현재 thread의 recent\_cpu값에 100을 곱해준뒤 FRACTION으로 나눠주고 반환해준다.  


* 1. **시험 및 평가 내용**

**priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석**  
  
  
  
위의 코드들은 /ser/tests/threads내부에 존재하는 priority\_lifo.c의 일부분을 캡쳐한 것이다. 위에서 보면 알 수 있듯이 위의 code는 같은 priority를 가지는 여러thread가 같은 RR order로 동작하는지를 확인하는 code이다.  
출력방식을 보면 기대되는 출력값은 매 라인별로 (priority-lifo) iteration: 이후에 수행되는 thread의 id가 출력되고(RR방식으로 반복) 마치면 개행이 되는 반복출력이다. 아래는 출력결과이다.  


위의 출력결과에서 알 수 있듯이 같은 priority를 가지는 thread들이 정확히 같은 routine으로 RR방식처럼 동작하고 있음을 알 수 있다. 그러므로 정상적으로 priority-lifo test case를 수행했다고 할 수 있다.

**Make Check결과**  
위의 결과는 /src/threads에서 make check를 수행한 결과이다. 위의 결과를 보면 알 수 있듯이 ppt에 명시된 모든 case들에 pass하였으므로 정상적으로 thread project를 완료하였다.