**Pintos Project 3 : Thread**

**(설계 프로젝트 수행 결과)**

과목 명 : OS

담당 교수 : 김영재 교수님

조 / 조원 : 장지훈,안시현

개발 기간 : 2018/11/20~12/3

**프로젝트 제목 : pintos project 3 Thread**

**제출일 : 2018/12/05**

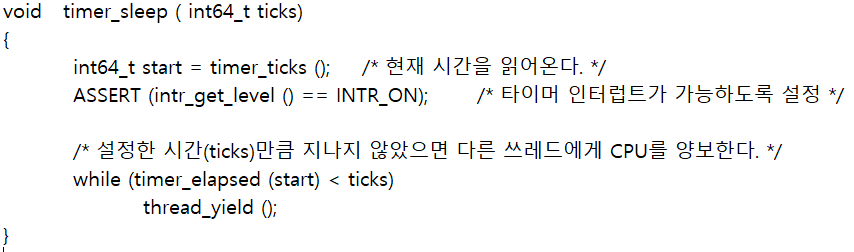
**참여 조원 : 안시현/장지훈**

1. **개발 목표**

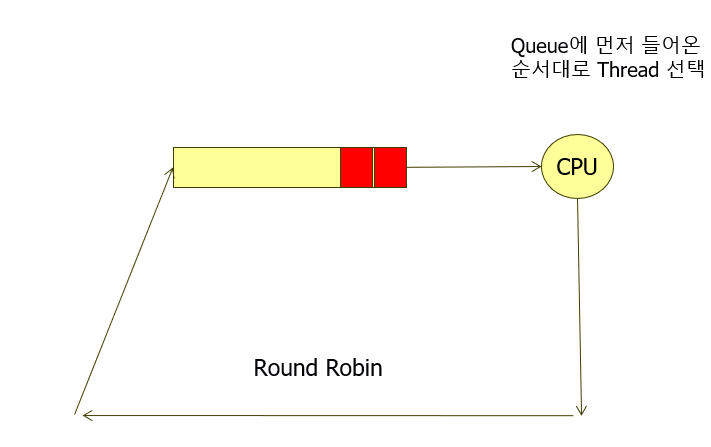
* 이번 프로젝트는 크게 2가지를 구현하는 것이다. 첫번째는 현재 Alarm clock이 timer\_sleep 함수에서 busy-waiting을 이용하는 방식으로 구현되어 있는데 이를 busy-waiting을 이용하지 않고 더 효과적인 방식으로 작동 할 수 있게 수정하는 것이다. 두번째는 Thread scheduling을 구현하는 것이다. Priority scheduler를 구현하여 각 Process가 갖는 priority에 따라 우선순위에 따른 scheduling이 적절하게 이루어질 수 있도록 해야 한다. 또한 여기에서 priority가 낮은 process들의 starvation 현상을 방지하기 위해 aging도 구현 하여야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 우선 alarm call을 busy-waiting -> non busy-waiting 방식으로 수정하기 위해서는 device/ timer.c의 timer\_sleep() 함수를 수정해야 한다. 현재 코드는 busy-wating 기반으로 timer\_sleep을 호출한 thread는 thread가 깰 시간이 될 때 까지 while문을 돌며 계속 기다리고 있다.



이를 non-busy-waiting방식으로 수정하여야 한다.

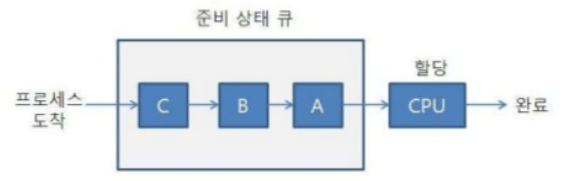


**<초기 pintos scheduler>**

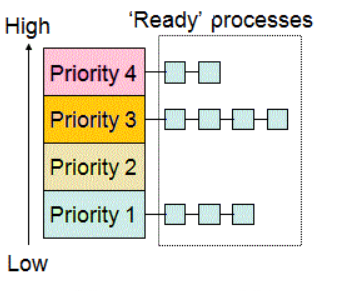
* Alarm call구현 후에는 Thread scheduler를 구현하여야 한다. 초기 pintos는 priority를 따로 두지 않고 새로운 thread가 입력되면 ready\_list의 맨 끝에 추가하는 구조로 구현되어 있다. 이를 각 thread에 우선순위를 부여하여 scheduling이 우선순위에 의해 수행되도록 수정할 것이다. 또한 이 과정에서 aging기법을 이용하여 우선순위가 낮은 thread들의 우선순위를 점진적으로 상승시켜서 결국엔 실행될 수 있도록 할 것이다. 이 구현을 위해 thread/thread.c를 수정해야 한다.
* 현재 thread/synch.c에 sema\_down과 sema\_up 함수에서 sema\_waiter list가 priority에 상관 없이 list\_push\_back 함수를 통해 list의 제일 뒤에 push 되는 방식으로 구현되어 있는데 이 부분을 sema\_waiter list도 priority에 기반해서 정렬되도록 수정해야 한다.
  1. **개발 내용**

Project를 진행하는데 필요한 기본 개념들에 대하여 짧게 설명하고 어떤 방식으로 이용했는지를 서술해 보았다**.**

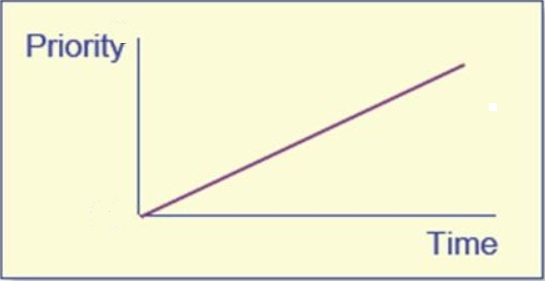
* + 1. **Round-Robin scheduling**



* 라운드 로빈 스케줄링(Round Robin Scheduling, RR)은 선점형 스케줄링의 하나로서, 프로세스들 사이에 우선순위를 두지 않고, 순서대로 시간단위(Time Quantum/ Slice)로 CPU를 할당하는 방식의 CPU 스케줄링 알고리즘이다. CPU(Processor)을 사용할 수 있는 기회를 프로그램 프로세스들에게 공정하게 부여하기 위한 한 방법으로서 각 프로세스에 일정시간을 할당하고, 할당된 시간이 지나면 그 프로세스는 잠시 보류한 뒤 다른 프로세스에게 기회를 주고, 또 그 다음 프로세스에게 하는 식으로, 돌아가며 기회를 부여하는 운영방식이라 할 수 있다. 이번 Project에서는 각각의 priority에 따른 ready\_queue를 만들고, 동일한 priority를 가지는 하나의 queue 안에서는 RR 방식을 이용한 sheduling을 할 것이다.
  + 1. **Priority thread scheduling**



* Priority scheduling은 우선순위에 따른 scheduling 방식으로 , context switching이 일어날 때 ready 상태인 thread들 중에 우선순위가 높은 thread가 cpu를 잡을 기회를 얻는 방식의 scheduling을 말한다. 이번 프로젝트 에서는 각각의 thread가 가질 수 있는 priority를 0~63으로 정해 놓았으므로, 64개의 ready\_queue를 만들어서 높은 우선순위의 thread가 먼저 실행 될 수 있도록 할 것이다.
  + 1. **Starvation and aging**



* 기아현상(Starvation)이란 낮은 등급에 있는 ready queue에 있는 프로세스가 영원히 실행 기회를 얻지 못하고 무한정 기다리는 현상이다. 이를 해결하기 위해 Aging 기법을 이용한다. Aging기법은 기아현상을 해결하기 위한 기법으로서 오랫동안 기다린 프로세스에 우선순위를 점진적으로 높여 줌으로써 언젠가는 실행 기회를 얻을 수 있도록 해주는 기법이다. 이번 프로젝트에서는 기아현상을 해결하기 위해 ready queue에 있는 모든 thread들의 우선순위를 timer interrupt가 있을 때 마다(thread tick이 증가할 때 마다) 1씩 증가시켜 줄 것이다. 그렇게 함으로써 언젠가는 실행 기회를 얻게 될 것을 보장 할 수 있다.
  + 1. **Priority preemption**
* 선점 스케줄링(preemptive scheduling)은 시분할 시스템에서 타임 슬라이스가 소진되었거나, 인터럽트나 시스템 호출 종료 시에 더 높은 우선 순위 프로세스가 발생 되었음을 알았을 때, 현 실행 프로세스로부터 강제로 CPU를 회수하는 것을 말한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

11월 20 ~ 22 : 프로그램, 매뉴얼 분석

11월 23 ~ 25 : thread 관련 부분 구현 (alarm -, priority-)

11월 26 ~ 30 : mlfqs 관련 구현

12월 01 ~ 03 : 보고서 작성

* 1. **개발 방법**

**ALARM CALL 부분**

* busy-waiting 기반의 timer 함수를 non busy-waiting 방식으로 구현하기 위하여 thread.h에 sleep\_node라는 구조체를 새로 정의할 것이다. sleep\_node에는 thread와 그 thread가 깨어나야 하는 시간인 wakeup time 변수를 둘 것이며 이를 list로 만들어 sleep thread들을 관리할 것이다.
* timer.c에 timer\_sleep 함수 내에서 while문으로 busy-waiting하게 기다리던 방식을 sleep\_list에 그 thread를 추가하고 그 thread를 thread\_block함수를 이용하여 block시켜 non-busy-waiting 방식으로 작동하게끔 수정할 것이다.
* Timer.c에 timer\_interrupt 함수를 수정하여 timer interrupt가 걸릴때마다 sleep\_list를 검사하여 wakeup time에 도달한 thread가 있으면 그 thread를 unblock 시켜 줄 것이다
* Timer.c에 sleep\_list를 wakeup\_time으로 정열하기 위해서 less function을 정의할 것이다. 이 함수를 통해 thread는 wakeup time으로 정렬 될 수 있다.
* Thread.c에 Priority scheduling을 위해 thread.c의 ready\_list를 ready\_queue[64]로 수정하여 64개의 우선순위마다 각각의 queue를 따로 두어 관리할 것이다.
* Synch.c에 sema\_down과 sema\_up의 sema\_waiter를 수정하는 과정에서 우선순위에 기반한 정렬이 이루어질 수 있도록 할 것이다. 정렬을 위해 high\_func라는 함수를 정의하여 이용할 것이다.

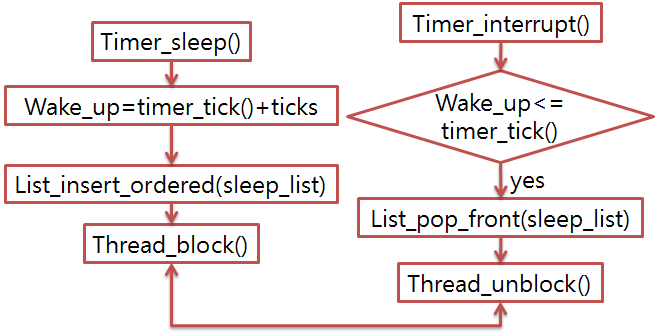
**AGING 부분**

* 각 thread를 대기시간에 비례해서 priority를 상승시켜 주기 위해 thread\_aging함수를 정의할 것이다. 이 함수에서는 모든 ready\_queue에 있는 thread들의 우선순위를 1씩 증가시키고 모든 ready\_queue를 재배열 해줄 것이다.

**MLFQS 관련 부분**

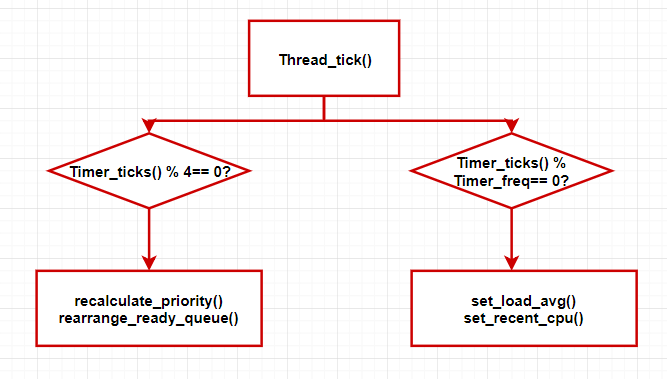
* Thread.h에 mlfqs를 구현하기 위해서 thread 구조체에 nice, recent\_cpu 변수를 추가하고 생성된 thread의 nice를 parent의 nice로 초기화 하기 위해 parent thread를 알아야 하므로 thread \* 변수를 추가해서 parent도 저장 할 수 있도록 할 것이다. 또한 같은 맥락으로 thread.c에 load\_avg 변수를 추가 할 것이다.
* Thread.c에 매 thread\_tick 함수를 수정할 것이다. 이는 매 4 ticks 마다 모든 thread의 priority가 재계산 되어야 하고 매 seconds 마다 load\_avg와 모든 thread의 recent\_cpu 값이 재계산 되어야 하기 때문이다. 이를 위해 thread\_tick 함수에서 timer\_ticks() 함수를 호출하여 그 시간만큼이 지났을 때 적절하게 재계산 될 수 있도록 할 것이다. 그리고 이를 위해 각 thread의 priority를 새로 계산해주는 함수인 recalculate\_thread\_priority와 바뀐 우선순위에 따라 새롭게 ready\_queue를 정렬하기 위해서 rearrange\_ready\_queue 함수를 정의할 것이다.
* Thread.c에 fixed floating point 계산을 위해서 필요한 함수를 추가할 것이다. 이를 통해 실수 연산을 지원하지 않는 pintos에서도 recent\_cpu와 load\_avg를 계산해 낼 수 있다.
* Thread.c에 새로운 load\_avg와 recent\_cpu를 계산하기 위해 필요할 함수들을 추가하여 새롭게 계산된 load\_avg와 recent\_cpu를 통해 new priority를 적절히 구할 수 있도록 할 것이다..
* Thread.c에 project3 명세서에서 구현하라고 명시한 나머지 함수들 또한 구현 할 것이다.
  1. **연구원 역할 분담**
* 장지훈 : pintos manual 분석, alarm call , priority scheduling , 보고서
* 안시현 : pintos manual 분석, mlfqs 기반 함수, priority scheduling, 보고서

1. **연구 결과**
   1. **합성 내용**

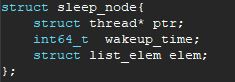


thread가 sleep상태로 들어갈 때에 sleep\_node의 wake\_up변수에 thread가 일어나야 할 시간을 저장해준 뒤 sleep\_list에 저장해준다.

그 뒤에 time interrupt가 걸렸을 때 sleep\_list안의 thread들의 wake\_up값을 현재의 시간과 비교하여 일어날 시간이 됐으면 thread를 unblock해준다.



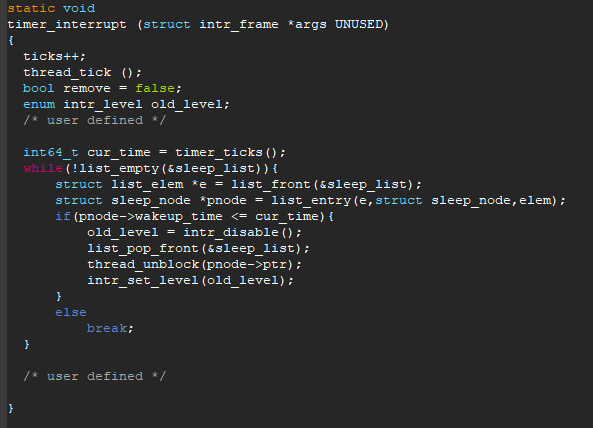
* Thread tick에서 매 4 tick마다 priority를 재계산 해주고 ready\_queue를 다시 정렬한다. 또한 매 1 초마다 load\_avg와 recent\_cpu를 재계산 해준다.
  1. **제작 내용**



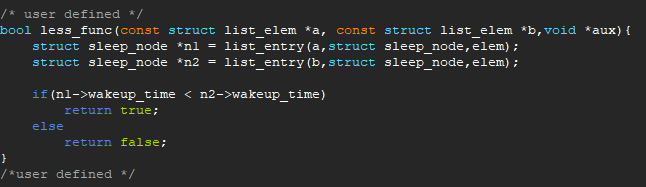
* 먼저 busy-waiting 기반의 방식을 바꾸기 위해 block된 thread들을 저장할 sleep\_node 구조체를 정의하였다. 이 구조체는 thread를 pointing할 수 있는 thread \* 형 변수와, 각각의 thread가 깨어나야 하는 wake\_up\_time 변수, 그리고 list에 저장하기 위한 list\_elem type의 elem 변수로 구성하였다.



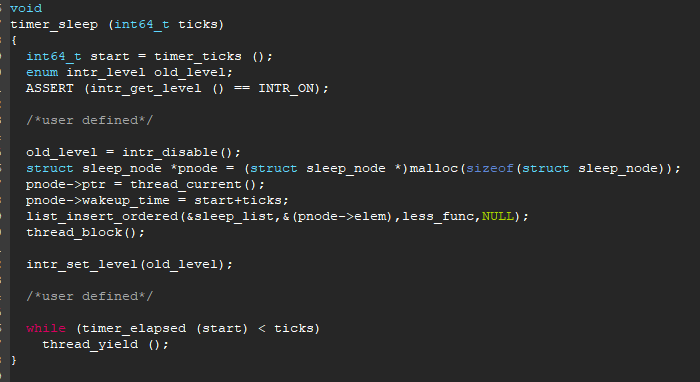
* Mlfqs 구현을 위해 nice와 recent\_cpu 변수를 추가하였다. Thread \* type의 parent변수는 nice와 recent\_cpu을 부모 thread로부터 물려받기 위한 변수이다.



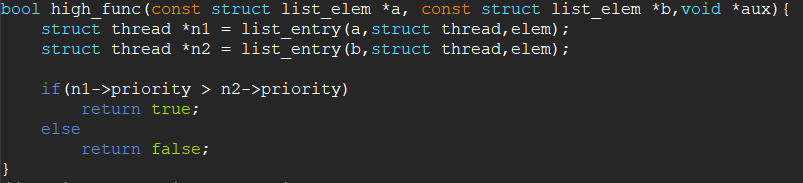
* timer\_interrupt 함수도 수정 하였는데, 기존에 thread\_tick 함수만 호출하던 부분에서 현재 시간을 timer\_ticks() 함수를 이용해 구하고 sleep\_list에 저장된 thread들 중에서 그 시간에 꺠어나야 할 thread가 있으면 thread\_unblock함수를 이용하여 깨워줄 수 있도록 했다. 또한 이 과정에서 interrupt가 발생하면 안되므로 interrupt 발생은 막아두었다.

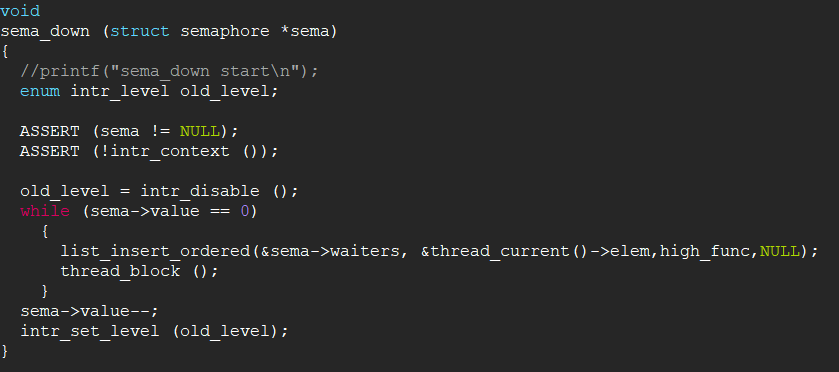


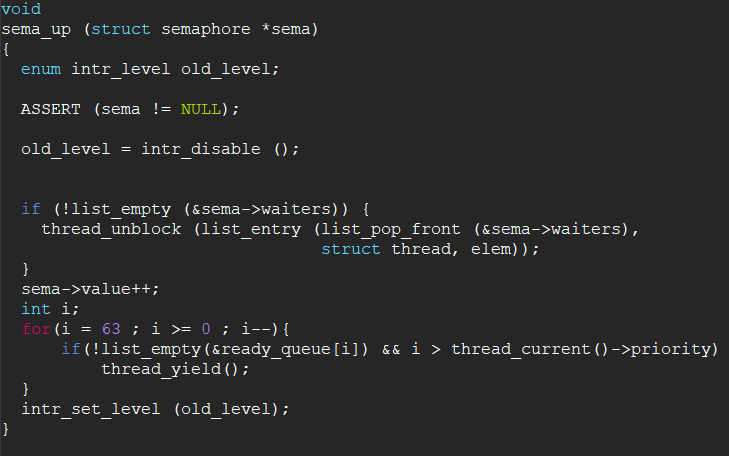
* Sleep\_list를 wakeup\_time으로 정렬하기 위해 이용되는 함수이다.

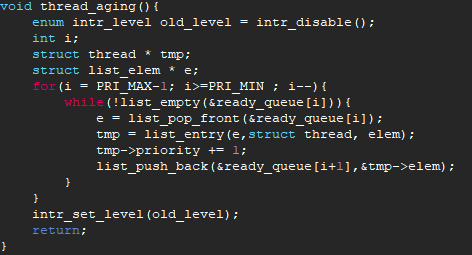


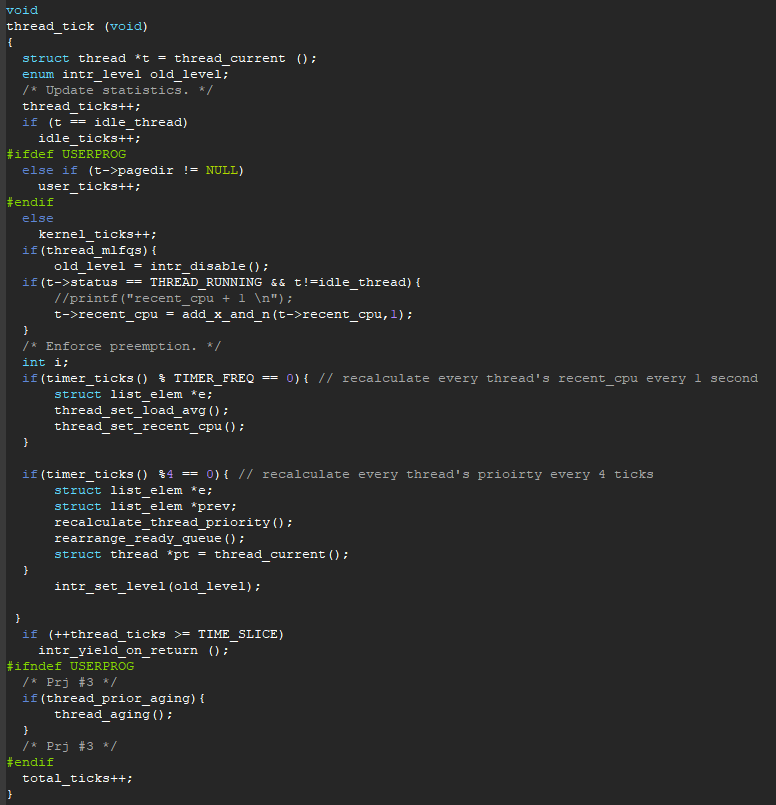
* 기존에 busy-waiting하게 구현되던 timer\_sleep 함수를 non-busy-waiting하게 수행되게 하기 위해 sleep\_node를 동적으로 생성하여 wake\_up time을 저장해 놓고 sleep\_list에 추가한 뒤 그 thread를 block해 주었다. Block된 thread는 wake\_up time이 되면 thread\_unblock 함수에 의해 깨게 된다.

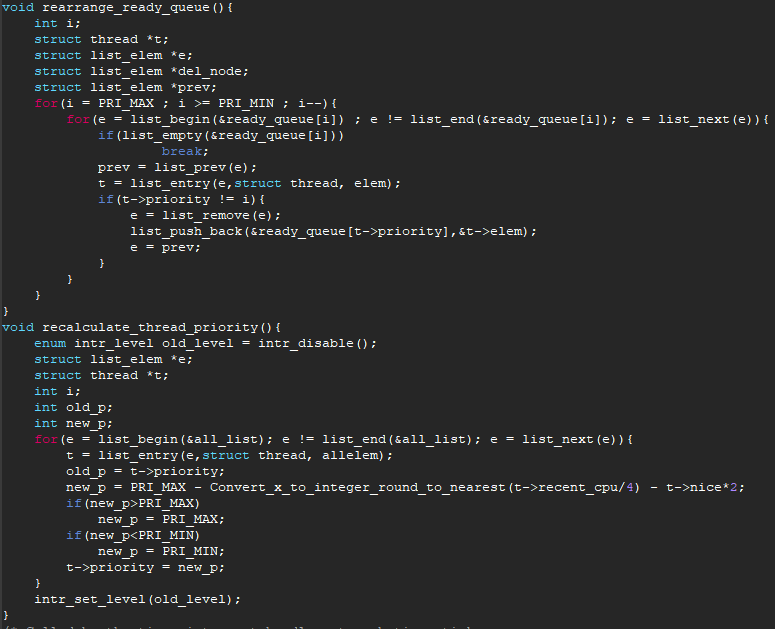
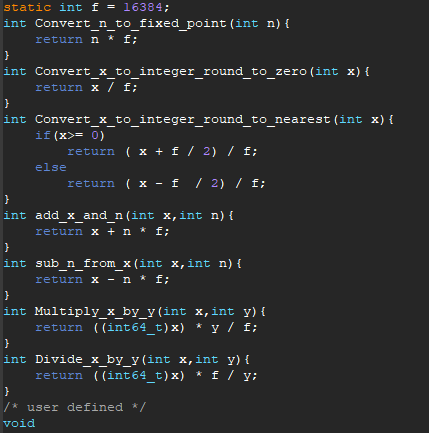


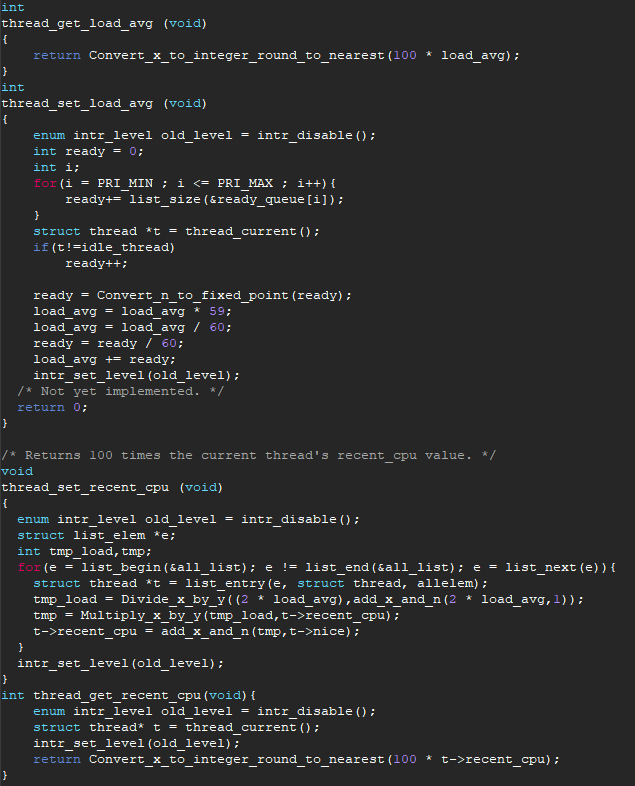
* 기존에 들어오는 순서대로 저장되었던 sema\_waiter list를 우선순위에 기반하여 정렬하기 위해 추가한 함수이다.
* 
* 기존의 list\_push\_back 함수로 제일 뒤에 추가되던 부분을 list\_insert\_ordered함수를 이용해서 자기 자리를 찾아서 삽입되도록 수정하였다.



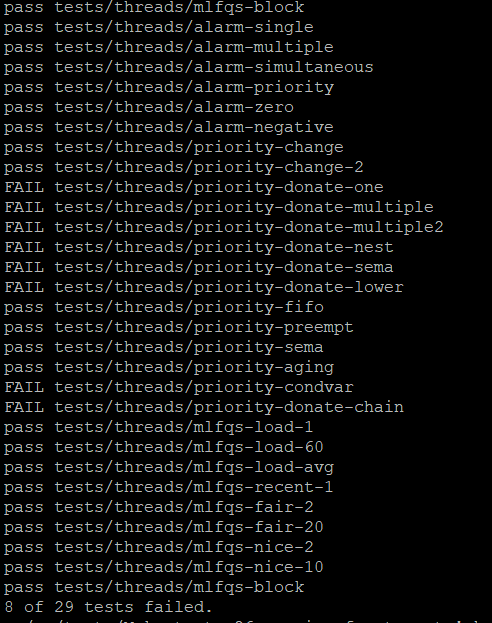
* Thread가 unblock 되면 그 unblock된 thread의 priority가 현재 수행되고 있는 thread의 priority보다 높을 수가 있으므로 그럴 경우 preemption해 주는 code를 추가하였다.
* 
* Starvation 현상을 막기 위해 timer\_interrupt가 걸려 thread\_tick() 함수가 호출 될 때마다 ready queue에 있는 모든 thread의 우선순위를 1씩 늘려주었다. 그 후 각 thread들을 우선순위에 맞는 ready queue에 다시 삽입 해 주었다. 이를 통해 낮은 우선순위의 thread도 언젠가는 수행 될 수 있다.

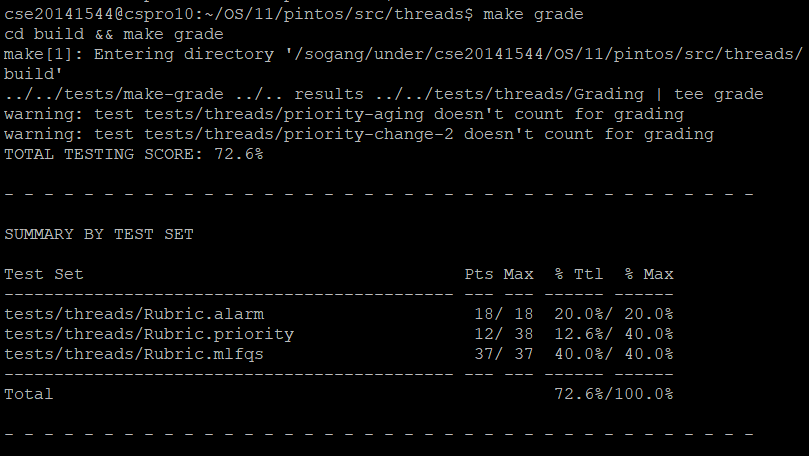


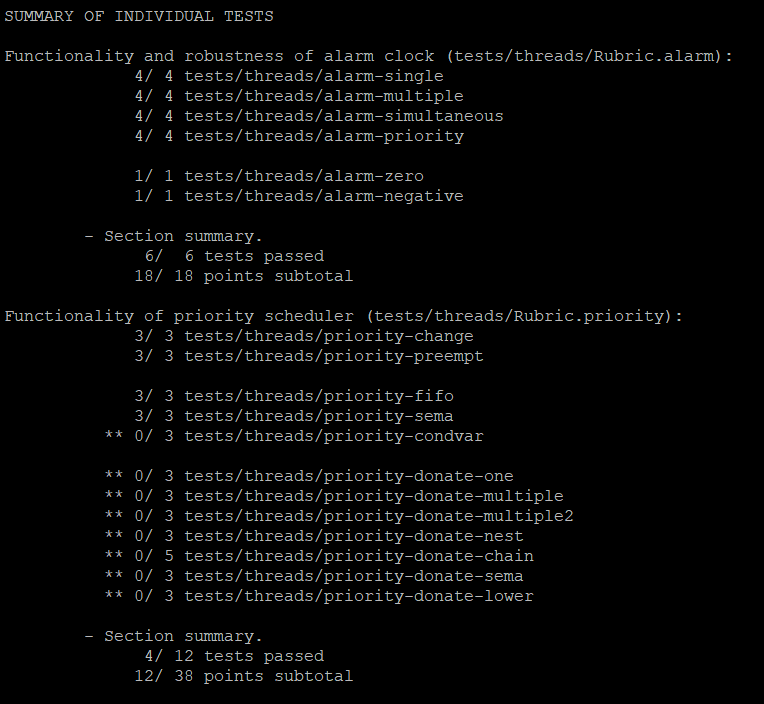
* Thread\_tick 함수에서 timer\_ticks() 함수를 이용해 매 4tick마다 우선순위를 재계산 하는 부분을 추가하였고 매 초마다 load\_avg와 recent\_cpu를 재계산 하는 부분도 추가하였다. 또한 aging옵션이 붙었으때는 thread\_aging 함수를 호출하여 priority aging 부분이 추가되도록 했다. 그리고 idle threa가 아닌 Running thread에 대해선 각각의 tick마다 recent\_cpu를 1씩 증가시켜 주어 정확한 priority 계산이 이루어질 수 있도록 하였다.
* 
* 매 4 ticks 마다 우선순위가 재계산된다. 이를 위해 recalculate\_thread\_priority 함수를 추가했다. Project 명세서에 있는 공식대로 new\_priority를 계산해 주고 0보다 작거나 63보다 큰 값이 나오는 경우에는 0과 63으로 만들어 주어 범위를 벗어나지 않게 하였다.
* MLFQS 구현을 위해 ready\_queue를 64개로 구성하였기 때문에, 우선순위가 재계재 되었을 때 각각의 ready\_queue를 재배열 해야한다. Rearrange\_ready\_queue는 이를 위한 함수이다. 각각의 ready\_queue를 돌면서 priority에 맞지 않는 ready\_queue에 있는 thread들의 제자리를 찾아준다.
* 
* Fixed-Point Real Arithmetic을 위해 추가한 부분이다. Pintos kernel에서는 실수 연산을 지원하지 않으므로 Fixed-point 연산 방식을 통해 load-avg와 recent\_cpu 값을 계산하여야 하는데, 이 과정에서 필요한 함수들을 pintos manual을 보고 구현했다.

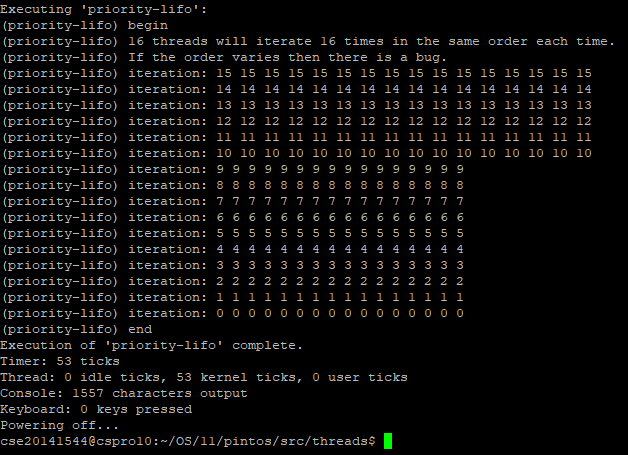
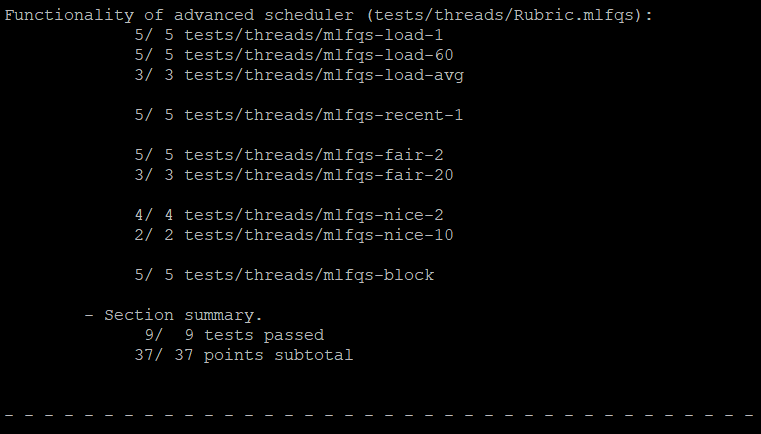


* Recent\_cpu와 load\_avg를 계산해 주는 함수이다. 프로젝트 명세서에 있는 공식을 이용하여 계산하였고, recent\_cpu는 load\_avg와 달리 thread별로 있는 변수이므로 all\_list를 이용하여 모든 thread에 대해 계산해 주었다.
  1. **시험 및 평가 내용**
* **코드캡쳐**









* Priority-lifo test 실행 결과이다. 우선순위가 높은 15부터 0까지 차례로 제대로 실행 되었음을 확인 하였다.

1. **기타**
   1. **연구 조원 기여도**

* 안시현(50%)
* 장지훈(50%)
  1. **소감**

안시현 : 이번 프로젝트 진행을 하면서 개념적으로만 알고 있던 Thread scheduling에 대하여 직접 구현해보며 scheduling에 대한 이해도를 높일 수 있었던 것 같다. 특히 Multi-level-feedback-queue관련 testcase를 통과 시키면서 pintos의 fixed floating point관련 연산 과정이 이해가 되지 않아 헷갈렸지만 부족했던 MFQS관련 개념을 완벽히 이해 할 수 있어서 뿌듯했고, nice와 recent cpu, load\_avg란 변수를 통해 cpu 사용시간과 대기중인 thread개수에 따른 새로운 priority를 구해낼 수 있다는 것이 상당히 흥미로웠다. 다음 프로젝트인 가상 메모리 부분도 상당히 기대가 된다.

장지훈 : 사실 처음에는 MLFQ관련 구현을 포기하고 priority scheduling만 구현하려 했으나, 프로젝트 진행 과정에서 욕심이 생겨 전부 다 구현하게 되었다. 프로젝트 초반에는 프로젝트 초반에는 thread\_yield 함수의 역할을 잘못 이해 하고 있어 조금 갈피를 잡지 못하였다. 그렇지만 매뉴얼을 반복해서 읽고 프로젝트 명세서를 여러 번 참조한 결과 느리지만 모든 testcae를 진행 할 수 있었다. 또한 aging관련 testcase는 thread\_tick마다 ready\_queue에 있는 모든 thread의 priority를 1씩 증가시켜 ready\_queue에 있는 시간에 비례하여 priority가 증가하도록 하였는데, 실제 OS에서 구현된 방식이 궁금해 져서 인터넷에 검색을 해보기도 하였다. 힘들었지만 다 끝내고 나니 보람차고 OS에 대한 흥미를 유발해주는 프로젝트 였던 것 같다.