**Pintos Project 3 : Threads**

**(설계 프로젝트 수행 결과)**

과목 명 : 운영체제

담당 교수 : 김영재 교수님

조 / 조원 : 72조 20141515 김연후 20141362 천지환

개발 기간 : 11.17 ~ 12.8

**프로젝트 제목 : Pintos Project 3 Threads**

**제출일 : 2019.12.8**

**참여 조원 : 20141515 김연후, 20141362 천지환**

1. **개발 목표**

* 기존 pintos의 경우 Alarm Clock이 구현되어 있긴 하지만 이가 busy waiting 기법을 통해 구현된 상태이다. 동작의 문제는 없지만 busy waiting 기법의 경우 CPU를 많이 잡아 먹기 위해 busy waiting을 해결한다.
* 기존 pintos의 경우 schedule을 할 때 thread 들의 priority가 고려 되지 않은 round-robin schedule 기법을 사용한 상태이다. 이를 thread 들의 priority를 고려할 수 있는 priority scheduler로 고쳐준다.
* 앞의 priority scheduler을 작성했을 경우 priority가 낮은 thread의 경우 아무리 시간이 지나도 실행되지 않는 오류가 있을 수 있다. 이를 해결하기 위해 aging 기법을 추가해준다.
* 추가 구현으로 multi-level-feedback-queue를 이용하는 bsd scheduler 기법을 추가해준다.

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

* 1의 개발 목표에서도 설명하였 듯 현재 Alarm clock의 동작 자체는 정상적으로 동작하도록 구현되어있는 상태이다. 하지만 기존 핀토스는 구현 방식에서 busy waiting을 기법을 사용하고 있다.

|  |
| --- |
|  |

* 위의 사진을 보면 알 수 있듯이 devices/timer.c 의 timer\_sleep함수에서 while문을 돌면서 timer가 tick보다 커지는지 계속해서 확인하며 timer가 주어진 tick보다 커질떄까지 현재 thread를 thead\_yield() 함수를 통해 계속 다른 thread에게 실행 권한을 넘겨 준다.
* 이 경우 busy waiting 상태가 되어 계속 CPU time의 낭비가 초래된다. 따라서 busy waiting 이 아닌 다른 방법을 사용해서 alarm clock이 작동하도록 구현해주어야 한다.
* 따라서 주어진 tick이 되기 전까지는 thread를 block상태로 만든다. 그 후 thread를 sleep 되어 있는 상태인 thread들을 저장하는 queue를 새로 만들어 이 queue에 thread를 추가해주고 후에 timer가 주어진 tick보다 크거나 같아질 때 상태를 ready로 바꾸며 ready queue로 옮겨준다. 이는 개발 내용에서 더 자세히 설명하겠다.

1. Priority Scheduler

* 1번의 개발 목표에서도 설명 하였 듯 기존 pintos에서는 thread 간의 priority가 고려되지 않았다. 기존의 scheduler는 새로운 thread가 ready queue에 추가되거나 ready queue에서 running 할 thread를 선택할 떄 priority가를 따지지 않고 ready queue에 추가 할 떄는 들어온 순서대로 ready queue의 뒤에 추가해주는 구조이다.(아래 사진의 list\_push\_back() 함수 이용) running 할 thread는 ready queue에 있는 것을 순서대로 선택하여 사용했다.

|  |
| --- |
|  |

* Priority scheduler에서는 이와 다르게 thread들 모두 각자의 priority 값을 갖게 한 다음에 이 priority를 고려해서 scheduling하도록 구현해준다. Ready queue에 넣을떄 priority를 고려해서 넣어 ready queue에서 pop 해줄때는 priority의 order에 맞게 pop 되도록 하여 구현한다. 이는 개발 내용에서 더 자세히 설명하겠다.

1. Aging

-앞의 priority scheduler를 구현한 이후 schedule을 하게 되면 priority를 고려하여 priority가 높은 thread를 계속 우선시 하여 진행하기 떄문에 priority 낮은 thread의 경우 계속 실행이 안되는 starvation 문제가 일어날 수 있다(아래의 그림의 priority 13을 갖는 thread의 경우 계속해서 rady list로 priority가 13보다 큰 thread가 추가되면 진행되지 않는다.)

|  |
| --- |
|  |

이를 해결하기 위해 시간이 지남에 따라 priority 값이 계속 증가하게 설정하여 priority가 처음에는 낮은 상태였어도 시간이 지난 후에는 priority가 점점 커져서 실행 될 수 있게하는 aging 기법을 이용한다. 이 방법은 아래의 개발 방법에서 더 자세히 설명한다.

1. Fixed-point real arithmetic

* 기존 핀토스의 경우 float 의 연산이 설정되어 있지 않아서 int의 연산만이 가능하다. 하지만 aging 기법이나 bsd scheduler를 구현할 때 load\_avg나 recent\_cpu를 계산하기 위해서는 float의 연산이 필요하다. 이를 위해서 float 연산을 아래의 그림을 참고해서 추가로 구현해준다. 이는 아래의 개발 방법에서 더 자세히 설명한다.

|  |
| --- |
|  |

1. BSD scheduler

* Scheduler에 일종으로 아래 그림과 같이 multi level queue를 이용하는 scheduler이다. 각각의 queue는 priority를 갖고 이 priority가 높은 queue부터 진행되며 각 queue는 round-robin의 schedling을 이용하도록 추가 구현해준다. 이는 아래의 개발 방법에서 더 자세히 설명한다.

|  |
| --- |
|  |

* 1. **개발 내용**

1. Alarm clock

* 앞에서도 설명하였듯 busy waiting 기법을 해결하기 위해 주어진 tick이 되기 전까지는 thread를 block상태로 만들고 새로 만든 queue에 넣어준다. 이를 위해 deviecs/timer.h에 sleep\_list를 static 변수로 추가해준다 그 뒤에 timer\_sleep의 앞의 busy wait으로 구현되어 있는 부분을 thread block이후 sleep list에 그 thread를 추가하는 방식으로 바꿔준다.

|  |
| --- |
|  |
|  |

* 그 뒤에 timer\_interrupt에서 이 sleep list를 돌면서 대기 시간이 모두 끝나고 새로 unblock 해줘야 될 thread가 있는지 확인하여준다.

|  |
| --- |
|  |

1. Priority scheduling

* 앞에서도 설명하였듯이 ready queue에 넣어줄 때 priority를 고려하여 priority 가 큰순서로 넣어 queue안에는 자동으로 정렬이 되어있게 해준다 그 후 출력할떄는 순서대로 출력하여 자동적으로 priority가 큰 thread 부터 나오게 해준다. Priority가 큰 순서로 thread를 넣는 것은 list\_insert\_ordered() 함수를 이용하며 이떄 elem들의 priority를 비교할 연산이 필요하기에 priority\_cmp로 elem의 priority들을 비교해주는 연산자를 설정하여 준다.
* 그 후 thread\_yield와 thrad\_unblock에서 list\_insert\_ordered로 list insert함수를 바꿔준다. Thread\_crate에서도 새로 생성할 thread의 priority가 현재 수행되고 있는 thread의 크기보다 크면 thread\_yield를 하게 설정하여준다.
* 이때 sema작업에서 priority 의 순서에 맞지 않게 thread가 실행 될 수 있기에Sema의 작업에서도 waiter를 골라줄떄 priority를 고려하도록 코딩해주어야한다.

|  |
| --- |
|  |
|  |

1. Aging

-위에서 설명하였듯이 time이 지남에 따라 함수의 priority가 커지게 해주어야 한다. 이때 priority를 계산하는 과정에서 float 계산이 필요하다. 기존 핀토스는 float 계산을 지원하지 않기에 새로 헤더파일을 만들어 이를 구현해야하며 이는 아래에서 설명하고있다. Priority를 계산해주는 식은 아래 와 같다.



aging에서는 저 priority의 계산에서 필요한 변수인 recent\_cpu와 nice 그리고 recent\_cpu의 계산에 필요한 load\_avg를 추가로 구현해주었다. Nice 변수의 경우 -20에서 20사이를 갖으며 cpu의 priority를 낮춰주는 역할(nice값이 양수일떄)을 하며 이는 priority식에는 있지만 bsd에서 사용되고 priority scheduling aging에서는 계속 0의 값을 갖는다. Recent\_cpu의 값은 각 thread 들이 사용할 cpu time을 측정 한 것으로써 더 많은 cpu time을 사용할수록 더 큰 recent cpu값을 갖게 되고 이 경우 더 낮은 priority를 갖게 된다. 마지막으로 load\_avg의 경우 ready state에 있는 thread 수의 평균 값으로 기존 load\_avg값과 ready\_threads 값을 이용해서 설정해준다. Ready\_threads 값은 idle thread를 제외한 ready 상태이거나 running 중인 thread 수이다. 이 세 변수를 구하는 식은 아래의 bsd scheduler에서 설명하였다. 이 변수들을 바탕으로 thread\_aging() 함수를 만들어서 timer가 timer가 interrupt 될떄 마다 thread가 wake\_up 인지 확인후 aging이 true이면 priority가 커지도록 구현하였다. 코드는 아래에서 서술하였다.

1. Fixed-point real arithmetic

* 위에서도 설명하였듯이 기존 pintos는 float연산이 불가능하기에 개발 범위에서 보여준 사진의 32bit가 float를 구성하는 방법을 이용한다. 그림을 보면 32bit 이진수의 경우 most significant bit인 1bit이 sign bit이며 그 뒤의 17bit가 integer부분을 그리고 그 뒤의 14bit가 floating point 부분을 나타낸다. 이는 결국 소수점에 14자리를 설정해 준다는 것이므로 수를 입력 받아서 2^14 를 곱해줄 경우 float가 32bit의 형태로 나타나게 된다. 이 점을 이용해 헤더 파일을 추가로 만들어서 #define FRACTION (1<<14) 를 설정 후 이를 곱해준다. 여기서 <<는 비트연산자로 1<<14는 1에 2^14승을 곱한 형태이다. 이를 이용해 헤더 파일을 만들어 주면 아래와 같다.

|  |
| --- |
|  |

1. BSD Scheduler

* 이는 위에서도 설명하였 듯 multi level queue를 이용하는 scheduler로 각 queue가 priority를 갖는데 이 priority를 계산해주는 식이 필요하다. 이때 사용되는 변수가 aging에서 정의한 nice와 recent\_cpu, load\_avg이다. 세 변수의 내용은 앞에서 언급한 봐야 같고 nice에 새로운 값이 주어질 떄 마다 load\_avg와 recent\_cput의 값을 새로 고쳐주어야 한다. 구현 방법은 thread\_mlfqs를 flag로 설정해 이 flag가 true일 경우 먼저 앞의 priority scheduler에서 필요한 thread\_set\_priority 수행을 막아주고 ppt에서 말한 바와 같이 interrupt가 발생할 때마다 recent\_cpu에 1을 추가 해준다. 또 TIMER\_FREQ마다 recent\_cpu와 load\_avg값을 최신화 해주고 priority의 경우 이를 이용해서 4tick 마다 최신화 하도록 구현한다. Recent\_cpu, load\_avg, priority를 계산해 주는 식은 아래 사진과 같다.

|  |
| --- |
|  |
|  |
|  |

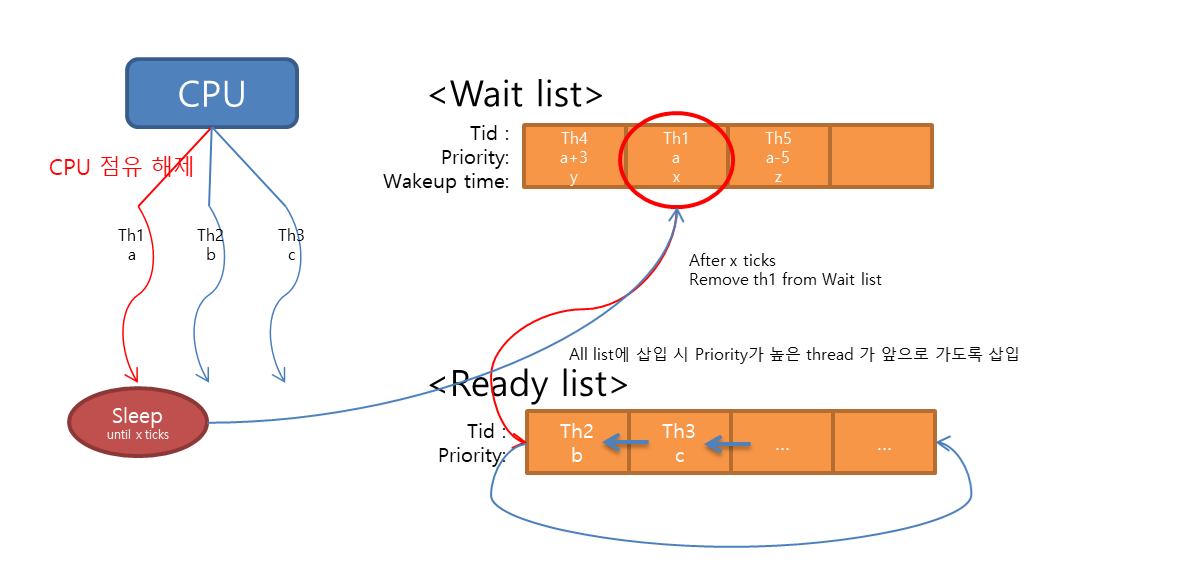
* 이를 바탕으로 앞에서 설정한 fixed-point 연산을 이용해 코드를 구현해준다. 코드의 구현은 뒤에 첨부되어있다.

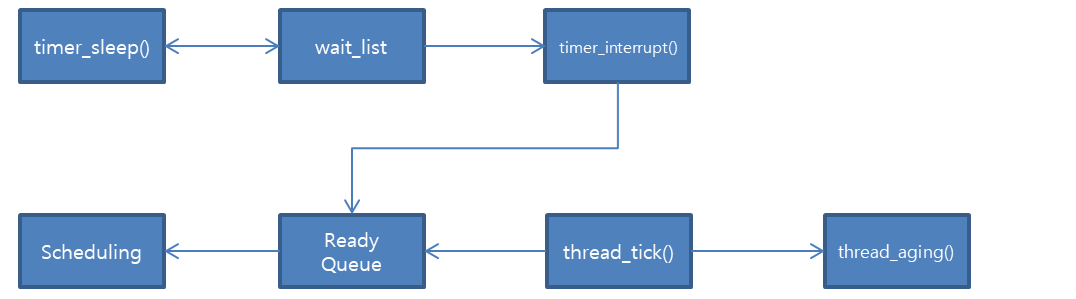
1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 기간 | 구현 내용 |
| 2019.11.17.~20. | Project ppt 이해 및 내용 숙지 |
| 2019.11.21.~23. | Alarm Clock 부분 이해 및 코드 작성 |
| 2019.11.24.~27 | Priority Scheduling 부분 이해 및 코드 작성 |
| 2019.11.28~12.01 | Priority aging 부분 이해 및 코드 작성 |
| 2019.12.02~4 | 기본 구현 에러 체크 및 코드 수정 |
| 2019.12.5~6 | BSD scheduler 코드 작성 및 오류체크 |
| 2019.12.8 | Pintos 보고서 작성 |

* 1. **개발 방법**
* 개발이 필요한 이유와 구현 원리 및 방법에 대해서는 A의 2번에서 자세히 설명하였다. Cspro9서버에서 VI editor를 이용해 ppt과 pintos manual을 바탕으로 개발하였다. 개발 환경은 linex, 컴파일 환경은 gcc이다.
  1. **연구원 역할 분담**
* 각자 project 3를 구현한 다음 만나서 더 코드 구성이 좋은 프로그램 제출, 보고서는 나눠서 작성.

1. **연구 결과**
   1. **합성 내용**





* 1. **제작 내용**

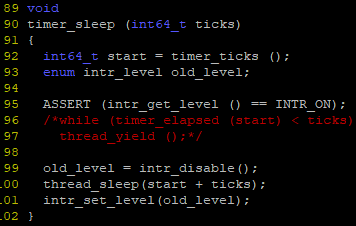
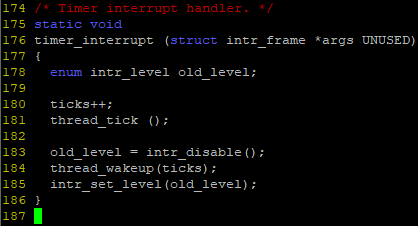
1. 자료구조

* struct thread에 wakeup\_tick을 추가 
* thread.c에 wait\_list를 추가해주고, thread\_init() 함수에서 wait\_list에 대해 초기화한다.

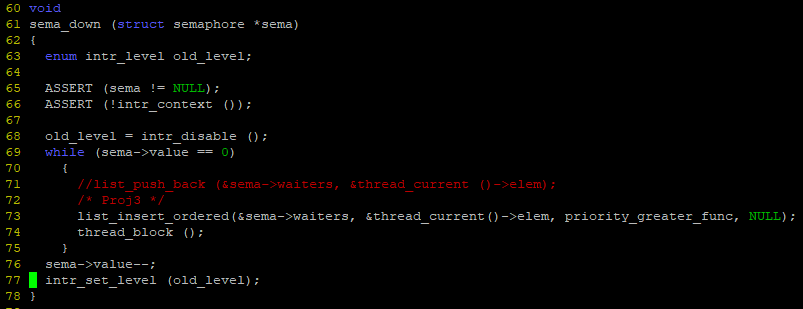
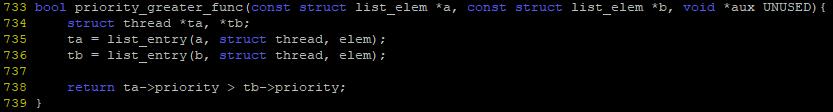


1. 알고리즘 또는 구현한 함수

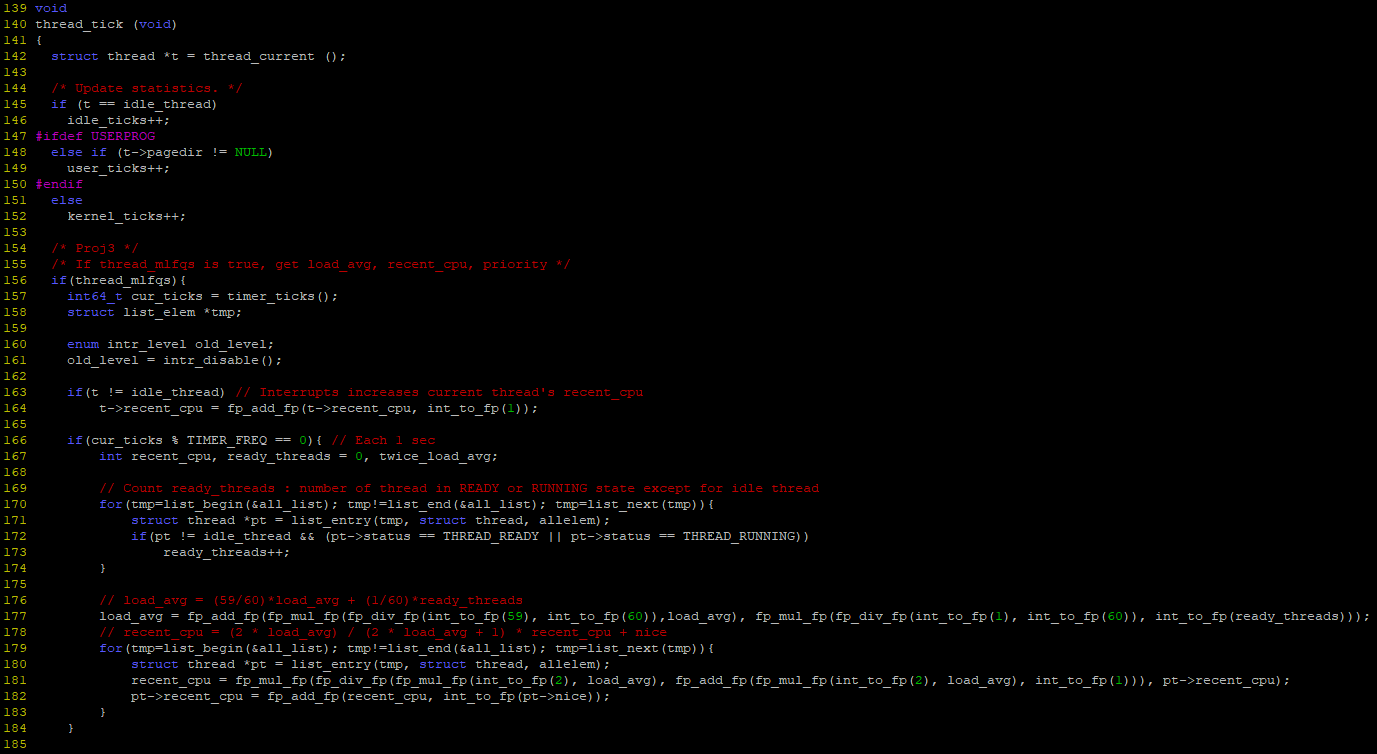
<timer.c>

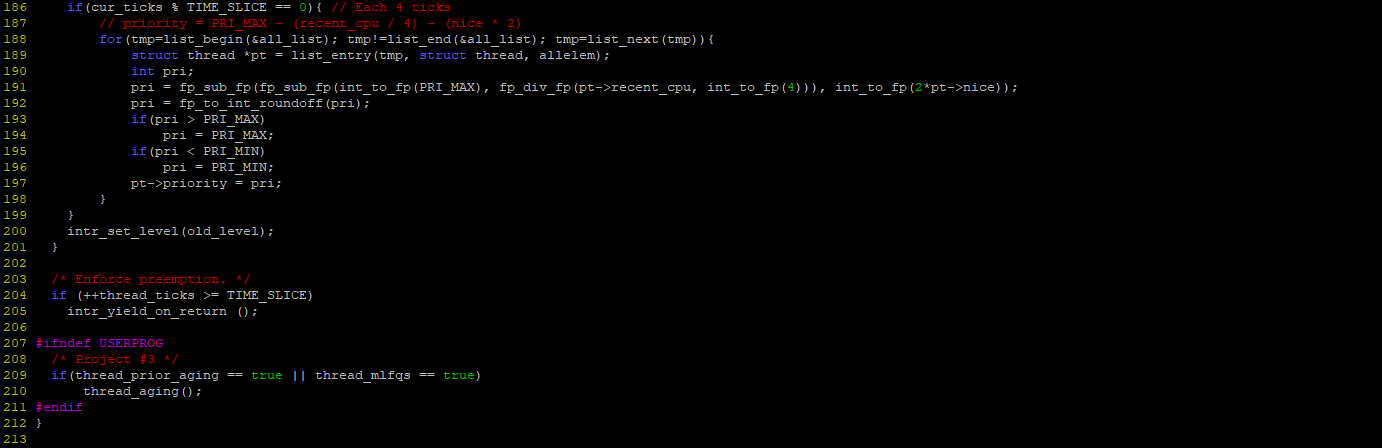
* timer\_sleep() : 받아온 ticks(sleep상태 시간)와 timer\_ticks()값 (현재 시간) 을 더해서 wakeup time을 thread\_sleep()함수와 이용한다.
* timer\_interrupt() : tick 증가에 따라 thread\_tick() 호출, wait\_list 탐색해서 thread의 wakeup\_tick과 tick을 비교하여 현재 tick이 같거나 크면, list\_pop\_front()로 제일 앞부분의 thread를 깨운다.

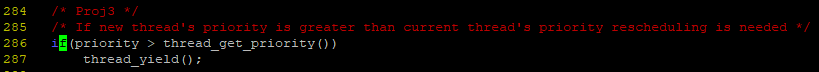
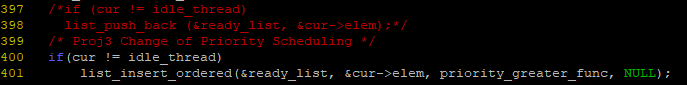
<synch.c>

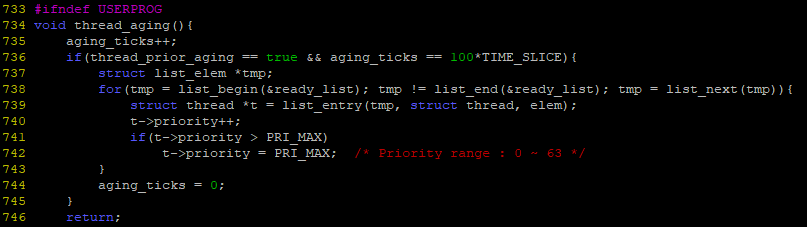
* ready\_list, wait\_list를 priority 반영하여 만들기 위해 기존의 list\_push\_back()함수 대신에 list\_insert\_ordered()함수를 이용
* sema\_down에서 priority에 따라 wait\_list를 만들기 위해 priority를 비교하여 정렬되도록 하는 priority\_greater\_func() 함수를 이용하였다. 이 함수는 두 priority a,b 중 a가 크면 true, b가 크면 false를 return 한다.
* sema\_up에서도 priority에 따라 ready\_list를 만들기 위해 priority를 비교하여 정렬되도록 하는 priority\_greater\_func() 함수를 이용하였다.

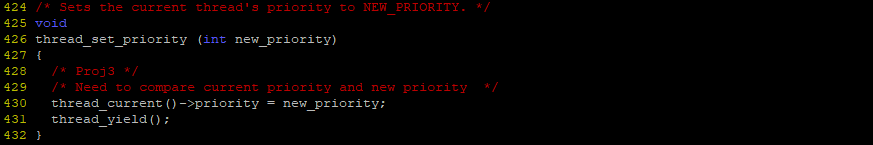
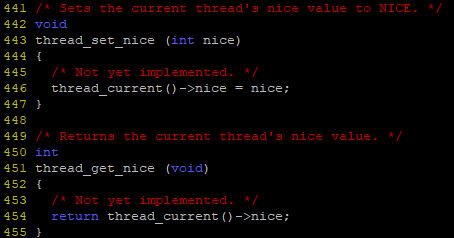
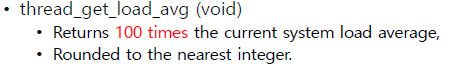
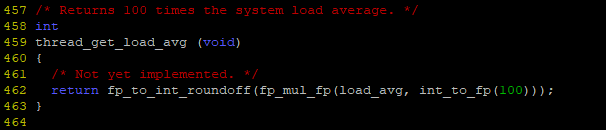
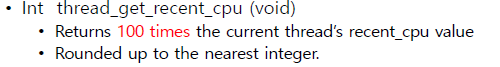
<thread.c>

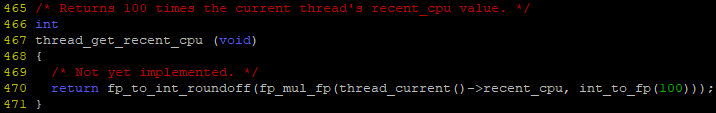
* thread\_tick() : 매 번의 tick 마다 timer interrupt에 의해 불려지는 함수로 thread\_mlfqs가 수행될 때는 매초마다 ready\_threads 개수와 load\_avg, recent\_cpu를 계산해주고, 4초에 한 번씩 priority를 구해준다. Thread에 aging이 필요할 때는 thread\_aging() 함수를 이용한다.

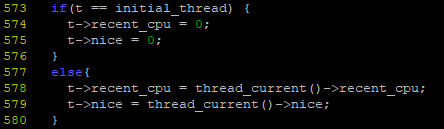
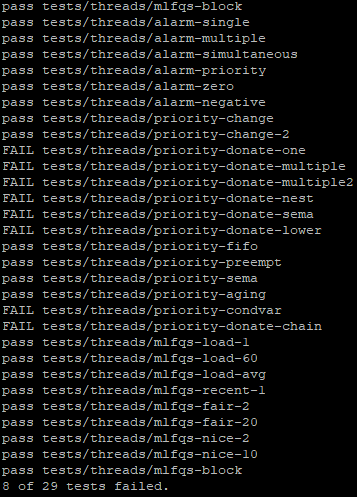


* thread\_create() : priority를 비교하여 ready\_list에 넣어주는 기능으로 thread\_yield()함수를 사용한다.
* thread\_yield() : idle\_thread가 아니면 priority를 비교하는 방식으로 ready\_list에 넣어주는 기능 추가
* thread\_aging() : priority scheduling에서 starvation문제를 해결하기위한 방법 구현



* thread\_set\_priority() : 현재 thread의 priority를 받아온 new\_priority값으로 설정하고, thread\_yield() 함수로 순서 정리
* thread\_set\_nice() : 현재 thread의 nice값을 받아온 nice 값으로 설정
* thread\_get\_nice() : 현재 thread의 nice값 return
* thread\_get\_load\_avg() : load\_avg를 ppt 참고로 구현
* thread\_get\_recent\_cpu() : thread의 recent\_cpu를 ppt 참고로 구현

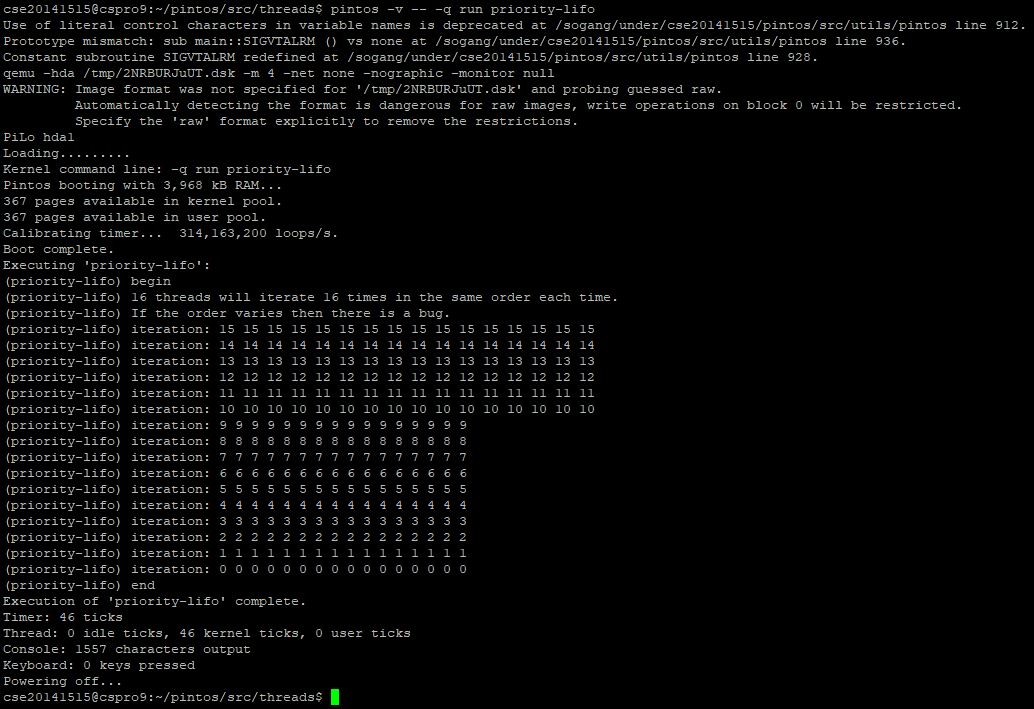


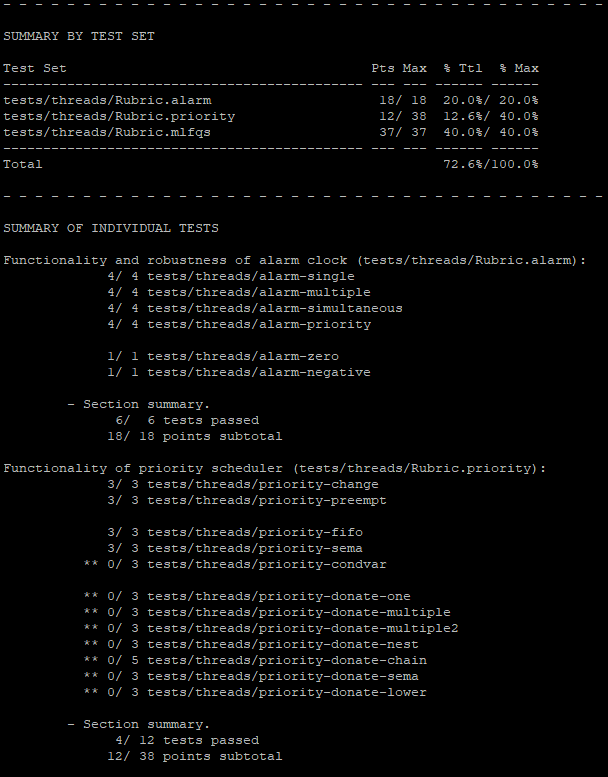
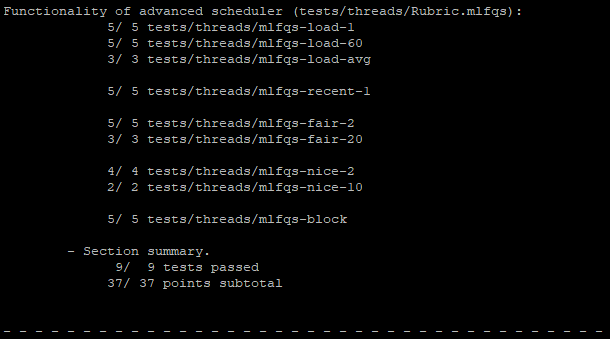
* init\_thread() : thread t가 init\_thread인지 확인하여, 맞으면 nice, recent\_cpu값을 0으로, 아니면 parent\_thread의 nice값과 recent\_cpu값을 가져와 같게 해주도록 추가하였다.
  1. **시험 및 평가 내용**
* make check 수행화면

기본 12개의 tests pass

추가 구현 9개의 tests pass

* priority-lifo 수행화면



* make grade 수행화면

1. **기타**
   1. **연구 조원 기여도**

* 김연후 20141515 : 50% 천지환 20141362 : 50%
  1. **소감**
* **김연후** : 수업시간에 배웠던 thread를 priority를 이용하여 scheduling 해주는 것을 직접 구현해보니 흥미로웠다. 특히 thread\_aging()의 필요성에 대해 이번 과제를 하면서 크게 체감했고, #define 에서도 무조건 괄호() 를 치는 것에 대한 중요성을 알 수 있었던 과제였다.
* **천지환 :** 마지막 bsd를 구현할 때 필요한 fixed-point 연산 구현에서 #define을 이용하였다. 여기서 FRACTION을 (1<<14) 가 아닌 1<<14로 DEFINE해서 연산자 우선순위 상 사칙연산이 비트연산보다 먼저 되어 bsd의 load\_avg값에서 계속 오류가나는 문제가 발생하였는데 이 문제를 몇 시간 동안 찾지 못하는 상황이 있었다. 앞으로는 연산자 우선순위와 ( )에 주의해야 한다는 점을 상기시키게 되었고 그래도 pintos 프로젝트를 무사히 마치게 되어 만족스러웠다.