**Pintos Project 3: Threads**

담당 교수 : 김영재

조 / 조원 : 20161598 손동현

개발 기간 : 2020.11.18~2020.12.08

1. **개발 목표**

기존에 busy waiting으로 구현되어 있던 Alarm Clock을 non-busy waiting 기법을 이용할 수 있게 수정한다. 그리고 각 프로세스에 priority를 부여하여 priority scheduling을 할 수 있게 구현한 뒤, 실수 연산을 구현하여 BSD Scheduler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
   2. Alarm Clock : 기존에 busy waiting으로 구현되어 있는 alarm clock은 특정 thread가 일정 부분을 계속 반복 수행하면서 대기를 하기 때문에 CPU 자원을 낭비하게 된다. 따라서 이를 non-busy waiting 기법을 이용할 수 있게 수정하여, CPU 자원을 낭비하지 않게 한다.
   3. Priority Scheduling : Priority Scheduling이란 각 프로세스에 우선 순위를 부과하여 우선 순위가 높은 프로세스를 먼저 스케줄링 하는 방법이다. 기존의 scheduling ready\_list 맨 끝에 새로운 thread를 추가하는 구조인데, 이것을 우선 순위에 맞게 ready\_list가 정렬된 상태를 유지하게 하는 방법으로 수정하여 더욱 효율적으로 scheduling을 할 수 있게 한다.
   4. Advanced Scheduler (BSD Scheduler) : 기존의 pintos가 사용하는 scheduling 방식은 FIFO 방식이다. 이를 MLFQ(Multi-Level Feedback Queue) 형식으로 구현한 General-Purpose scheduler를 구현하여 각 prioritys는 자신의 ready queue를 Round Robin 방식을 따르게 한다. 이를 통해 scheduling 시에 더욱 유연하고 효율적으로 scheduling을 할 수 있게 된다.
   5. **개발 내용**
2. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

* 깨어날 시간이 되지 않으면 thread는 Blocked 상태가 되어 있다. 이러한 thread들을 저장하기 위해 새로운 list(sleeping\_list)를 생성하고, 해당 thread 구조체 내부에 깨어날 시간을 함께 저장한다. 매 tick마다 timer\_interrupt 함수를 호출하여 깨어날 thread가 있는지 확인 후, 만일 있으면 해당 thread를 sleeping\_list에서 삭제하고 ready 상태의 thread들이 저장되는 list에 삽입하는 방식으로 blocked 상태의 thread를 깨울 수 있다.

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

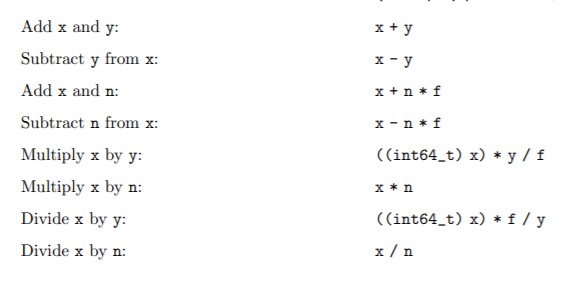
* Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우, 그 즉시 running thread는 CPU를 높은 priority를 가진 thread에게 양보하면서 running thread는 높은 priority를 가진 thread와 switching이 일어난다. 따라서 해당 running thread는 Ready list에 삽입되게 된다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.

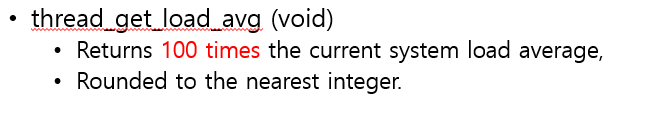
* Advanced Scheduler에서 priority 계산을 하기 위해서는 실수 연산(Ex. recent\_cpu, load\_avg : 실수)이 필요하다. 하지만 Pintos는 floating point 방식의 실수 연산을 지원하지 않는다. 따라서 이를 지원하기 위해 fixed point 방식을 사용한 실수 연산 함수가 필요하다. 총 8개의 함수 (“실수 \* 실수” , “실수 / 실수”, “실수 + 실수” , “실수 - 실수”, “정수 \* 실수” , “실수 / 정수”, “정수 + 실수” , “정수 - 실수”)가 필요하다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

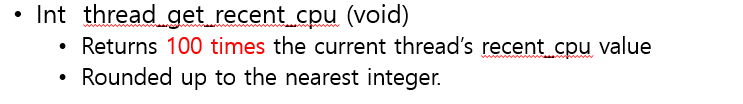
* 11/18 ~ 11/25 – proj3 동영상 시청 및 명세서 분석   
  11/26 ~ 12/02 – Alarm Clock, Priority Scheduling 구현   
  12/03 ~ 12/06 – BSD Scheduler 구현   
  12/07 ~ 12/08 – 보고서 작성
  1. **개발 방법**
* threads/thread.h : Struct thread 구조체 내에 깨어날 시간을 저장할 sleeping\_time 변수, BSD Scheduler의 priority 계산을 위해 필요한 recent\_cpu, nice 변수를 선언해주어야 한다.
* devices/timer.c : Blocked되어 자고 있는 thread들을 저장하고 있는 List가 필요하므로 struct list 구조체를 선언해주어야 한다.
* devices/timer.c / timer\_init : timer가 초기화될 때, sleeping\_list도 초기화 시켜주어야 한다.
* devices/timer.c / timer\_sleep : Thread를 재우는데 사용되는 함수로 기존에는 busy waiting 기법이 적용되어 있었으나, 이를 삭제하고 non-busy waiting 기법을 사용할 수 있게 코드를 수정해야 한다.
* devices/timer.c / timer\_interrupt : timer 하드웨어에 의해 매 tick마다 timer interrupt가 걸리는데 이 때 timer\_interrupt 함수가 호출된다. 해당 함수는 sleeping\_list를 확인하여 깨어날 thread가 있으면 이를 깨울 수 있게 코드를 수정해야 한다.
* threads/synch.c : sema\_up 함수에서 sema->waiters에서 priority가 가장 큰 값을 추출해 내기 위해 두 개의 thread를 priority로 비교하는 함수가 필요하다.
* threads/synch.c / sema\_up : 기존에는 sema를 기다리는 thread list 중 제일 앞에 있는 thread를 가져왔지만 여기에 Priority scheduling을 적용할 수 있게 코드를 수정해야 한다.
* threads/init.c / parse\_options : Parse\_options 함수에서 pintos가 넘기는 aging 플래그를 받는 경우, thread\_prior\_aging을 참으로 하여 aging 옵션이 실행되게 코드를 수정해야 한다.
* threads/thread.c : BSD scheduler를 위해 필요한 load\_avg와 실수 연산 함수에 사용하기 위해 많이 사용되는 2^14을 정의해두어 후에 사용할 수 있게 하였고, aging 옵션과 mlfqs 옵션을 위한 bool type flag도 정의한다. 또한 ready\_list를 priority가 높은 순으로 정렬하기 위해 사용되는 함수도 정의한다.
* threads/thread.c / thread\_init : Thread가 초기화될 때, 해당 thread 구조체 내부에 있는 recent\_cpu와 nice 값도 0으로 초기화 할 수 있게 코드를 수정한다.
* threads/thread.c / thread\_create : 새로 생성한 thread가 현재 실행 중인 thread의 우선 순위보다 높다면, 재스케줄링할 수 있게 구현한다.
* threads/thread.c / thread\_unblock : Thread가 unblock 되면, 해당 thread는 ready\_list에 삽입하게 되는데, 이 때 ready\_list는 항상 우선순위에 의해 정렬된 상태를 유지할 수 있게 구현한다.
* threads/thread.c / thread\_yield : 현재 실행 중인 thread를 ready\_list에 삽입하고 ready\_list는 항상 priority에 의해 정렬된 상태를 유지할 수 있게 하게 구현한다. 그 후 다시 스케줄링할 수 있게 한다.
* threads/thread.c / thread\_set\_priority : 우선 thread\_mlfqs가 참, 즉 MLFQ scheduler를 사용 중인 경우, thread\_set\_priority 함수 내용이 실행이 안되게 한다. 만일 thread\_mlfqs가 거짓인 경우, 현재 수행 중인 thread는 가장 높은 priority를 갖도록 보장되게 구현한다.
* threads/thread.c / thread\_get\_priority : 현재 수행중인 thread의 priority를 반환하게 구현한다.
* threads/thread.c / set\_nice : 만일 현재 수행중인 thread의 nice 값을 새로 받아 수정이 필요한 경우, 현재 수행중인 thread의 nice 값을 수정하고, 우선 순위를 다시 계산할 수 있게 구현한다. 그 후, 현재 ready\_list에 있는 thread 중 가장 높은 우선 순위 값을 얻어서 만일 현재 수행중인 thread의 우선 순위가 해당 우선 순위보다 낮다면 재스케줄 하게 구현한다.
* threads/thread.c / thread\_get\_nice : 현재 수행중인 thread의 nice 값을 반환하게 구현한다.
* threads/thread.c : 다음의 8가지 실수 연산을 수행할 수 있는 함수를 선언한다.



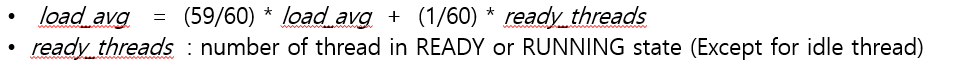
* threads/thread.c / thread\_get\_load\_avg : proj3 명세서에 나온 아래의 수식을 계산할 수 있는 함수를 구현한다.



* threads/thread.c / thread\_get\_recent\_cpu : proj3 명세서에 나온 아래의 수식을 계산할 수 있는 함수를 구현한다.



* threads/thread.c : proj3 명세서에 나온 아래의 load\_avg, recent\_cpu, priority 수식을 계산할 수 있는 함수를 구현한다.

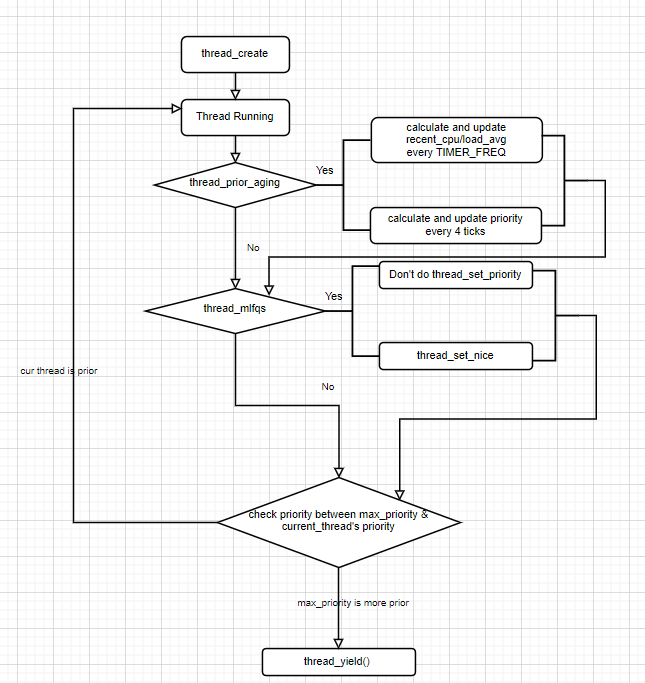




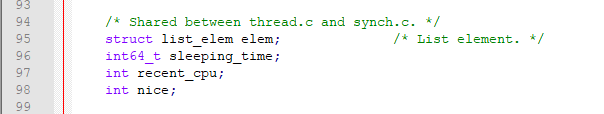


* threads/thread.c : recent\_cpu와 load\_avg 값은 매 초(TIMER\_FREQ)마다 업데이트되야 한다. 따라서 이를 해주는 함수를 구현한다. 또한, priority는 매 4 tick마다 업데이트되야 한다. 따라서 이를 해주는 함수도 구현한다.

1. **연구 결과**
   1. **Flow Chart**

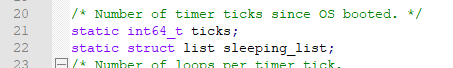
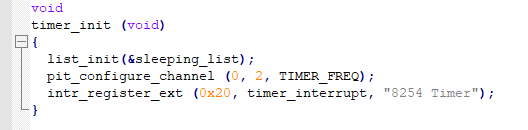


* 1. **제작 내용**
* threads/thread.h

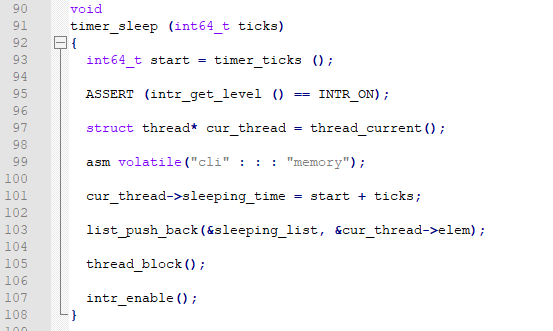


Struct thread 구조체 내에 깨어날 시간을 저장할 sleeping\_time 변수, BSD Scheduler의 priority 계산을 위해 필요한 recent\_cpu, nice 변수를 선언하였다.

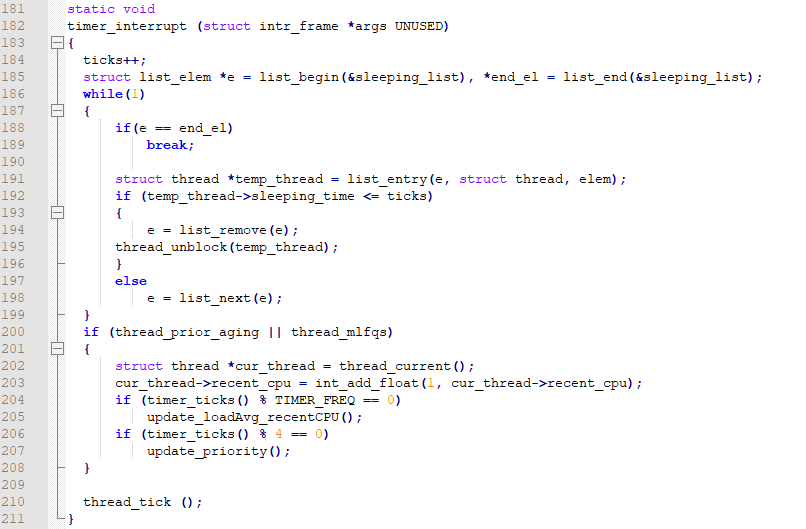
* devices/timer.c

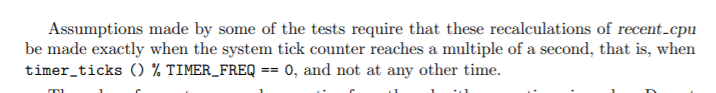
Blocked되어 자고 있는 thread들을 저장하고 있는 List가 필요하므로 struct list 구조체를 선언하였다. 그리고 timer가 초기화될 때, sleeping\_list도 초기화 시켜주었다.

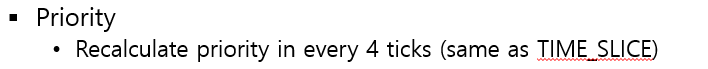


Thread를 재우는데 사용되는 함수이다. 기존에는 busy waiting 기법이 적용되어 있었으나, 이를 삭제하고 non-busy waiting 기법을 사용할 수 있게 코드를 수정하였다. 어셈블리 명령어로 메모리에 대한 접근을 허용시켜 주었다. Timer\_ticks 함수를 통해 현재 tick의 값을 반환한다. 여기서 tick이란 컴퓨터가 켜지고 1ms에 1씩 증가하는 값이다. 각 thread는 자신의 구조체 내부에 sleeping\_time이라는 시간에 처음 CPU를 잡은 시간부터 메모리 상에 존재한 시간을 더한 값을 갖은 후, blocked된 thread를 저장하는 sleeping\_list에 삽입된다. 그리고 해당 thread를 thread\_block() 함수를 사용하여 block하였다. 그리고 마지막으로 interrupt를 enable로 set하는 함수를 호출하였다.



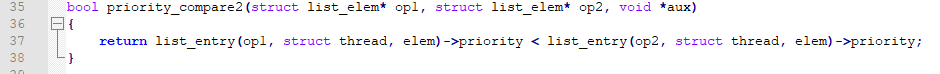
Timer 하드웨어에 의해 매 tick마다 timer interrupt가 걸리는데 이 때 timer\_interrupt 함수가 호출된다. 해당 함수는 sleeping\_list를 확인하여 깨어날 thread가 있으면 이를 깨우는 역할을 해준다. 따라서 while 반복문을 통해 sleeping list를 돌면서 만일 sleeping\_list의 thread의 sleeping\_time이 ticks 이하이면, 즉 깨어날 thread이면 해당 thread를 sleeping\_list에서 제거한 뒤, thread\_unblock 함수를 통해 unblock 시켜주었다. 또한 thread\_prior\_aging이 참, 즉 Aging 기법이 적용되거나, thread\_mlfqs이 참, 즉 MLFQ scheduler를 사용하는 경우, 현재 실행중인 thread의 recent\_cpu(실수)에 1(정수)를 정수와 실수를 더해주는 함수 int\_add\_float를 사용하여 더해주었다. 그리고, 아래와 같은 pintos 명세서에 나와 있는 기준에 따라 timer\_ticks()가 TIMER\_FREQ의 배수이면 recent cpu를 다시 계산하고, 매 4 tick마다 우선순위를 새로 계산한다.



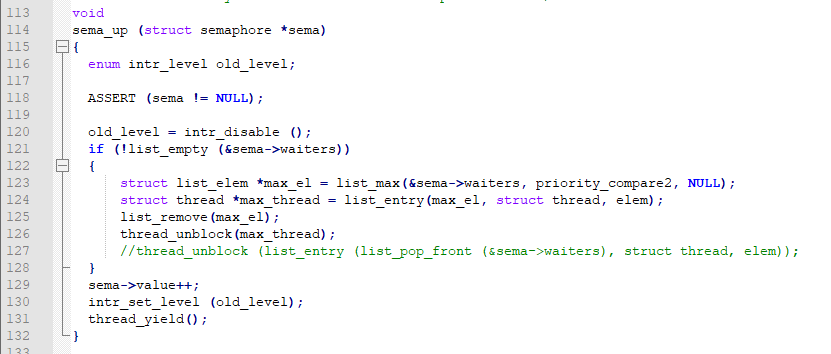


그리고 마지막으로 thread\_ticks() 함수를 호출하여 thread\_tick를 1 증가시킨다.

* threads/synch.c



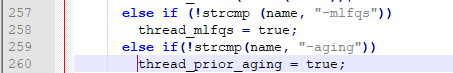
아래의 sema\_up 함수에서 sema->waiters에서 priority가 가장 큰 값을 추출해 내기 위해 두 개의 thread를 priority로 비교하는 함수이다.



기존의 sema\_up 함수의 내용을 수정하였다. 기존에는 sema를 기다리는 thread list 중 제일 앞에

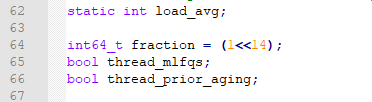
있는 thread를 가져왔다. 여기에 Priority scheduling을 적용하기 위해서 sema를 기다리는 thread 중 priority가 가장 높은 thread를 list에서 추출해야 한다. 위에서 정의한 priority\_compare2 함수를 사용하여 가장 우선 순위가 높은 thread를 추출한 뒤, 해당 thread를 list에서 제거하고, unblock 시켜주었다.

* threads/init.c

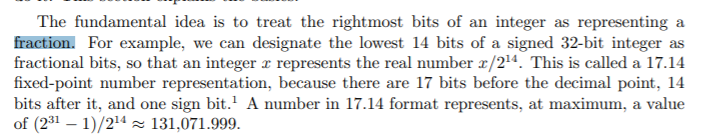


Parse\_options 함수에서 pintos가 넘기는 aging 플래그를 받는 경우, thread\_prior\_aging을 참으로 하여 aging 옵션이 실행되게 한다.

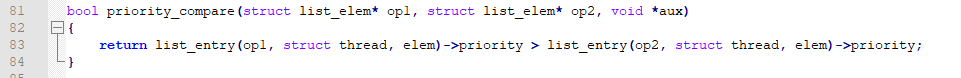
* threads/thread.c



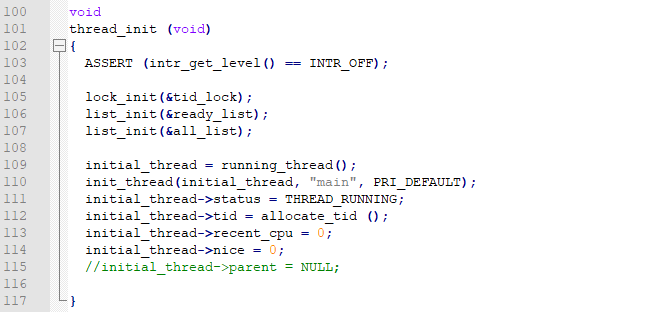
BSD scheduler를 위해 필요한 load\_avg를 선언한다.



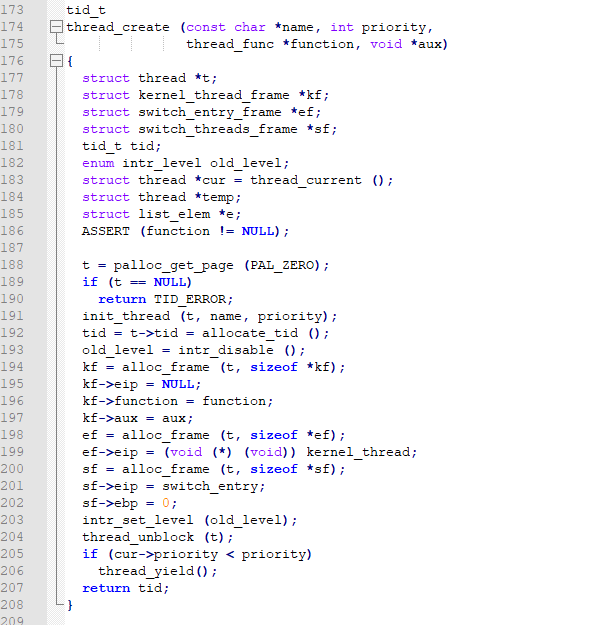
실수 연산 함수에 사용하기 위해 많이 사용되는 2^14을 정의해두어 후에 사용할 수 있게 하였고, aging 옵션과 mlfqs 옵션을 위한 bool type flag도 정의하였다.



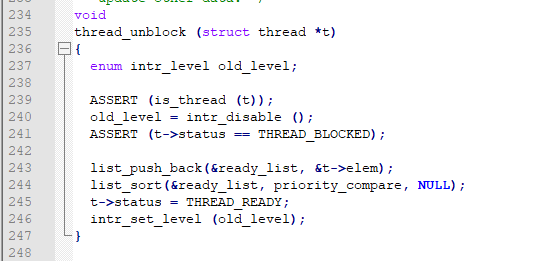
ready\_list를 priority가 높은 순으로 정렬하기 위해 사용되는 함수이다.



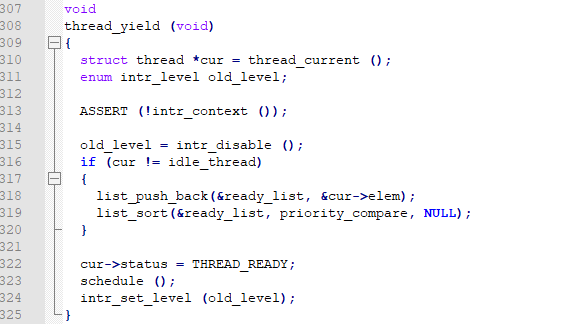
Thread가 초기화될 때, 해당 thread 구조체 내부에 있는 recent\_cpu와 nice 값도 0으로 초기화 시켜준다.



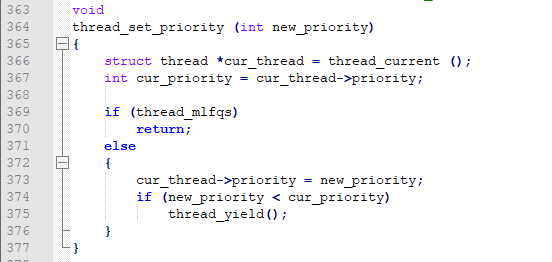
새로 생성한 thread가 현재 실행 중인 thread의 우선 순위보다 높다면, thread\_yield 함수를 호출하여 현재 실행 중인 thread를 ready\_list에 삽입하고 다시 스케줄링한다.



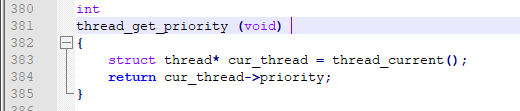
Thread가 unblock 되면, 해당 thread는 ready\_list에 삽입하게 되는데, 이 때 ready\_list는 항상 우선순위에 의해 정렬된 상태를 유지하게 하기 위해 앞서 선언한 priority\_compare 함수를 사용하여 list\_sort 함수로 ready\_list를 정렬한다.



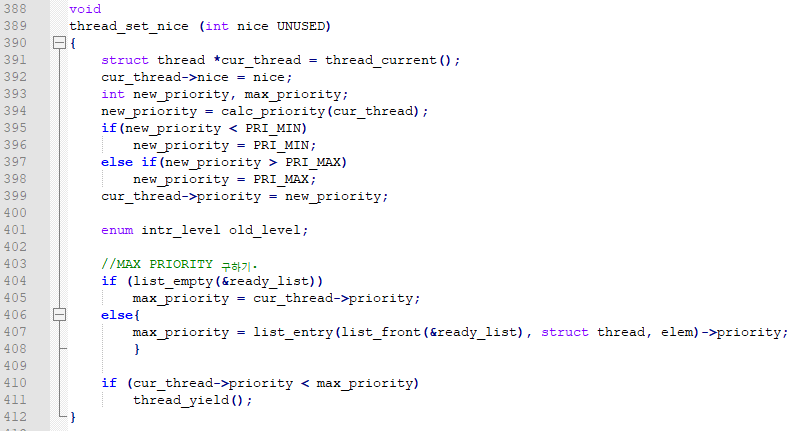
현재 실행 중인 thread를 ready\_list에 삽입한다. 이 때 ready\_list는 항상 priority에 의해 정렬된 상태를 유지할 수 있게 하기 위해, 삽입 후, list\_sort함수를 사용하여 정렬하여 준다. 그 후 다시 스케줄링한다.



우선 thread\_mlfqs가 참, 즉 MLFQ scheduler를 사용 중인 경우, thread\_set\_priority 함수가 실행이 안되야 하기 때문에 아래 내용을 실행하지 않고 바로 return을 한다. 만일 thread\_mlfqs가 거짓인 경우, 현재 수행 중인 thread는 가장 높은 priority를 갖도록 보장되야 한다. 따라서 파라미터로 넘겨 받은 priority를 현재 수행 중인 thread의 priority 변수에 대입 후, 기존의 실행 중이던 thread의 priority가 파라미터로 넘겨 받은 priority보다 큰 경우, thread\_yield 함수를 호출하여 다시 스케줄링한다.



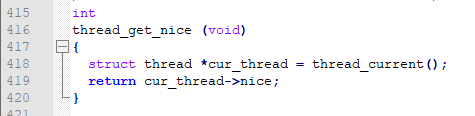
현재 수행중인 thread의 priority를 반환한다.



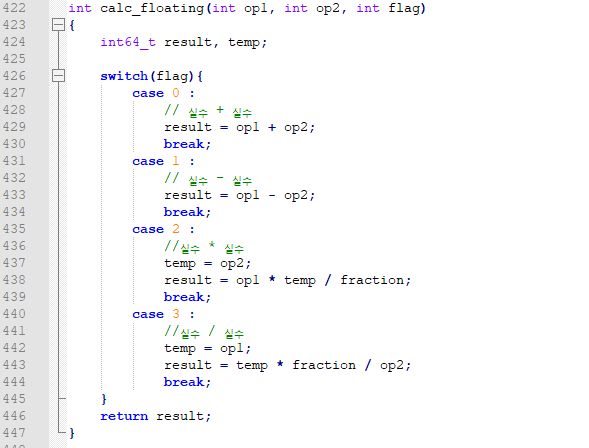
만일 현재 수행중인 thread의 nice 값을 새로 받아 수정이 필요한 경우, 현재 수행중인 thread의 nice 값을 수정하고, 우선 순위를 다시 계산해야 한다. 따라서 우선 순위를 계산하는 calc\_priority 함수를 호출하여 새로운 우선 순위를 계산한 뒤, 해당 우선 순위가 아래의 우선 순위 범위 내에 들어오는지 확인한다. 만일 넘어가는 경우, 양 끝 값으로 설정하였다.



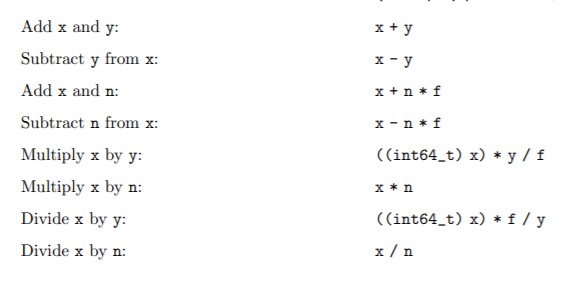
그 후, 현재 ready\_list에 있는 thread 중 가장 높은 우선 순위 값을 얻어서 만일 현재 수행중인 thread의 우선 순위가 해당 우선 순위보다 낮다면 thread\_yield 함수를 호출하여 재스케줄 한다.



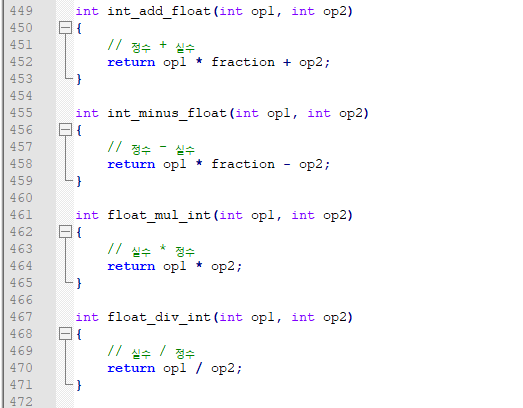
현재 수행중인 thread의 nice 값을 반환한다.



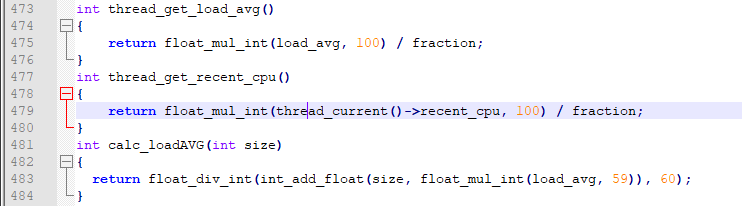
필요한 총 8개의 실수 연산 중, 실수끼리의 연산을 지원하는 함수이다. 원래는 총 8개의 연산 모두 이 함수에 넣으려 했으나, 정수와 실수의 연산은 파라미터의 순서가 중요해서 코드를 짜기에 헷갈리는 부분이 있어 뒤에 함수 이름으로 구별할 수 있도록 따로 구현하였다. 8개의 실수 연산은 아래 그림의 pintos 명세서에 따라 구현하였다.

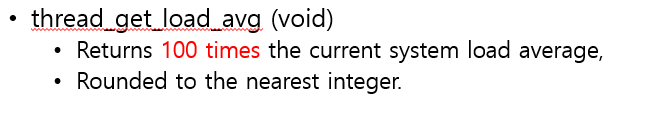


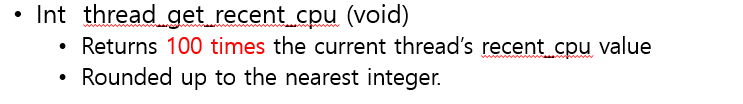
실수 + 실수, 실수 – 실수는 위와 같이 +, - 연산자로 구현하였고, 실수 \* 실수, 실수 / 실수의 경우 overflow가 일어날 수 있기 때문에 int64\_t 타입 변수 temp를 사용하여 pintos 명세서에 따라 구현하였다.



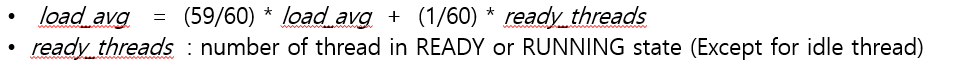
정수와 실수의 연산을 지원하는 함수이다. 위에서 첨부한 pintos 명세서에 따라 구현하였다.



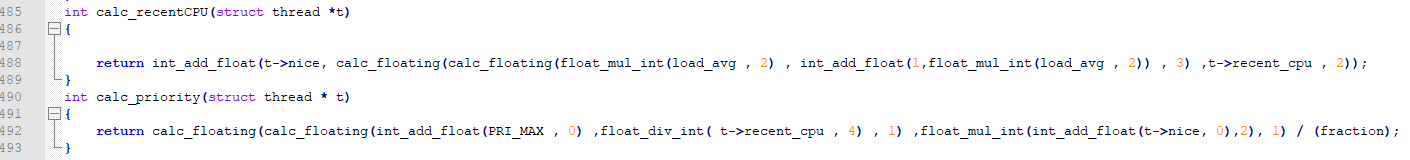




Thread\_get\_load\_avg와 thread\_get\_recent\_cpu 함수는 proj3 명세서에 주어진 것에 맞게 실수 연산 함수를 사용하여 구현하였다.



calc\_loadAVG 함수는 proj3 명세서에 주어진 것에 맞게 실수 연산 함수를 사용하여 구현하였다. 파라미터로 온 size는 현재 READY/RUNNING state인 thread의 개수를 의미한다. Load\_avg 값은 READY 상태의 thread 수의 평균으로 매 초(TIMER\_FREQ)마다 업데이트된다.

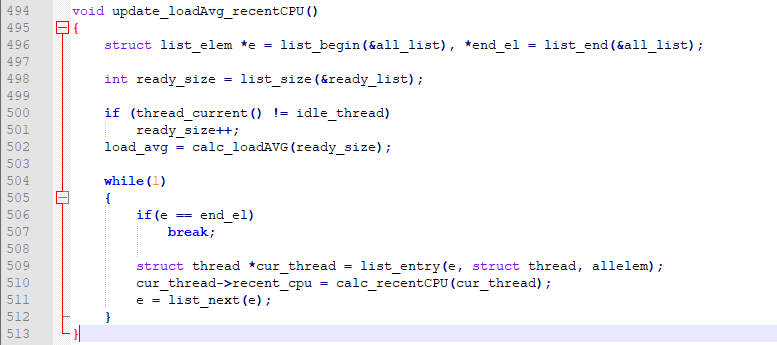




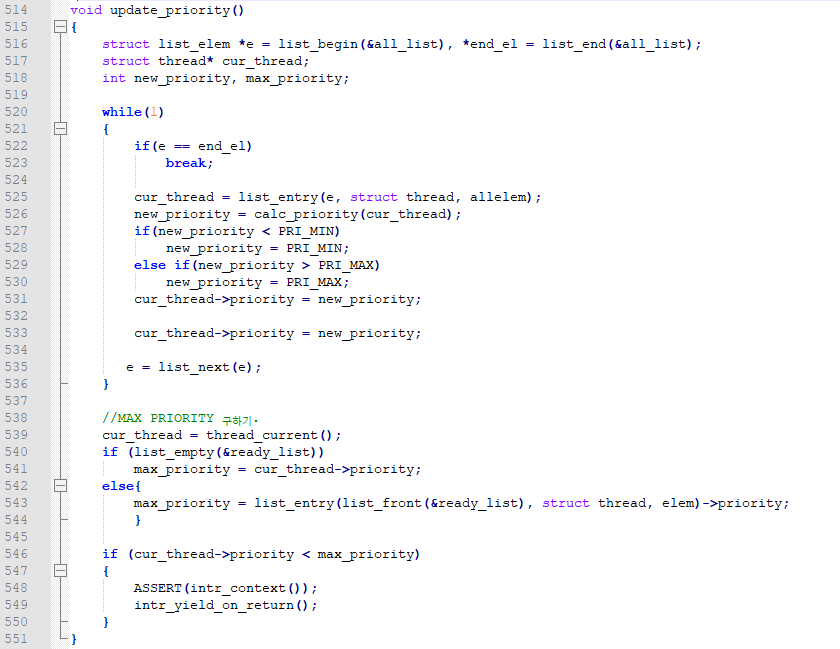
calc\_recentCPU 함수는 proj3 명세서에 주어진 것에 맞게 실수 연산 함수를 사용하여 구현하였다. 파라미터로 온 thread t는 현재 수행중인 thread이다. Recent\_cpu 값은 thread의 최근 CPU 시간을 나타내는 값으로 RUNNING 상태의 thread는 매 tick마다 1씩 증가하고, 모든 thread는 매 초(TIMER\_FREQ)마다 업데이트된다.



calc\_priority 함수는 proj3 명세서에 주어진 것에 맞게 실수 연산 함수를 사용하여 구현하였다. 파라미터로 온 thread t는 현재 수행중인 thread이다. 모든 thread의 priority는 매 4 tick 마다 재계산된다.

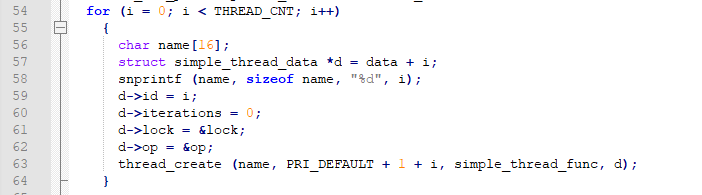


위에서 언급했듯이, recent\_cpu와 load\_avg 값은 매 초(TIMER\_FREQ)마다 업데이트되야 한다. 따라서 이를 해주는 함수이다. 우선 list\_size 함수를 통해 ready\_list의 size를 구한 뒤, 현재 수행되는 thread가 idle이 아닌 경우, 해당 thread도 메모리에 있는 thread로 간주하고 1을 추가하여 준다. 그리고 calc\_loadAVG 함수를 통해 load\_avg 값을 계산한다. Recent\_cpu는 모든 thread가 매 초마다 재계산되어야 하므로, while문을 통해서 all\_list를 돌면서 calc\_recentCPU 함수를 통해 모든 thread 들의 recent\_cpu 값을 업데이트해주었다.

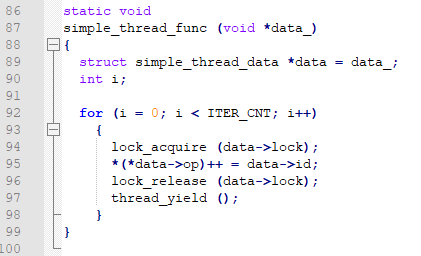


위에서 언급했듯이, priority는 매 4 tick마다 업데이트되야 한다. 따라서 이를 해주는 함수이다. priority는 모든 thread가 매 4 tick마다 재계산되어야 하므로, while문을 통해서 all\_list를 돌면서 calc\_priority 함수를 통해 모든 thread 들의 priority 값을 업데이트해주었다. 여기서 priority 값을 계산할 때, 주의할 점은 해당 값이 우선 순위 범위를 넘어선 경우, 해당 범위 안에 들어오게 양 끝 값 중 하나로 맞추어준다. 그 후, 현재 ready\_list의 가장 높은 우선 순위 값을 구한다. 현재 수행 중인 thread의 우선 순위가 가장 높아야 하는데, 만일 그 조건이 만족되지 않는다면 이미 interrupt 도중이므로 intr\_yield\_on\_return 함수를 호출하여 준다.

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

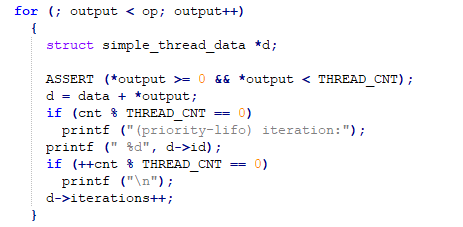


위 코드를 통해 for문을 통해 THREAD\_CNT(=16)개의 thread를 만든다는 것을 알 수 있다. 위의 코드는 해당 thread들의 구조체 변수들을 초기화하고, thread를 create할 때 simple\_thread\_func를 파라미터로 넘기는 것을 알 수 있다.

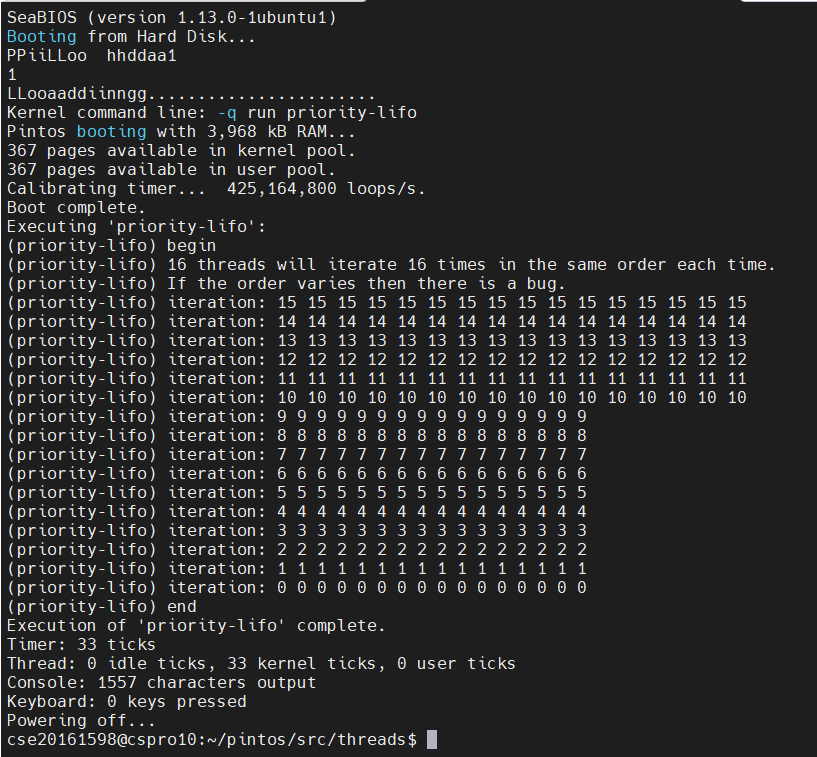


Simple\_thread\_func는 위와 같으며, 이를 통해 각 thread는 lock을 통해 동기화를 함을 알 수 있다.

처음 for 문에서 thread\_create에서 두 번째 파라미터는 thread를 생성할 때, priority가 되는데, 총 16개의 thread가 차례대로 만들어질 때, PRI\_DEFAUT+1+i가 priority의 파라미터로 넘어가게 되므로, 처음에는 1의 priority를 갖고 계속 1씩 증가하여 마지막 16번째 thread는 16의 priority를 갖음을 알 수 있다. 제일 처음 priority 1의 thread가 수행 중일 때, 2의 thread가 생성되면, 현재 수행중인 thread의 priority(1)보다 새로 만들어진 thread의 priority(2)가 더 크므로 thread\_yield가 수행되어 priority 1의 thread는 ready\_list에 들어가게 된다. 이렇게 계속 더욱 큰 priority를 갖는 thread가 생성되므로 1~15까지의 thread는 ready\_list에 순서대로 들어가게 된다. 그 후 아래의 for문을 실행하게 된다. 각 thread는 simple\_thread\_func의 함수의 내용처럼 ITER\_CNT(=16)번 반복하여 lock을 acquire과 release를 반복하게 된다.



그 때 매번 위 반복문의 “ %d”, d->id에 의하여 본인의 id인 priority보다 1 낮은 값을 출력하게 된다. 해당 testcase는 마지막에 생성된 testcase일수록 priority가 높으므로 더욱 마지막에 생성된 thread순으로 thread가 실행되게 된다. 따라서 LIFO(Last In First Out) 형태를 띄게 된다고 할 수 있다. 아래 실행 결과, priority가 높은 thread부터 ITER\_CNT(=16)번 반복해서 수행됨을 확인할 수 있으므로 제대로 프로그램이 돌아간다는 것을 알 수 있다.



* make check 수행 결과

