**Pintos Project 3: Threads**

담당 교수 : 김영재

조 / 조원 : 20171666 이예은

개발 기간 : 12/4 ~ 12/7

1. **개발 목표**

이번 프로젝트의 목표는 block list를 활용한 alarm clock을 구현하고, 기존 round-robin 방식이었던 스케줄링을 priority가 있는 스케줄러로 바꾸는 것이다. 또한 aging 기법을 적용하고, 이를 이용해 mlfq를 사용한 advanced scheduler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
   2. Alarm Clock: 기존 timer는 설정된 expired time이 될 때까지 while loop를 이용해 계속 기다리고 확인하는 방식이었다. 하지만 이 방식은 매우 비효율적인 busy waiting 이기 때문에 block 상태를 이용한 새로운 alarm clock을 구현한다. 이를 이용해 cpu 낭비를 막고 더 효율적으로 timer를 세팅할 수 있다.
   3. Priority Scheduling: 기존 round robin scheduler는 우선순위가 없어 모든 thread를 돌며 동일한 time 마다 교체하는 방식이었다. 하지만 이 방식은 먼저 수행되어야 하는 thread를 고려하지 않기 때문에 priority를 적용한 새로운 scheduling을 구현한다. 또한 그냥 priority만 구현할 경우, priority가 낮은 threads는 계속 실행되지 않는 starvation 문제가 발생할 수 있기 때문에 aging 기법을 적용하여 이를 막는다.
   4. Advanced Scheduler (추가구현을 한 경우): priority scheduling에서 구현한 aging 기법을 적용해 새로운 BSD scheduler를 만든다. 핀토스 매뉴얼에 따라 MLFQ를 이용한 scheduler를 적용한다. 이를 통해 각 priority의 큐에서 round robin 방식으로, 높은 priority 부터 수행하도록 구현한다.
   5. **개발 내용**
2. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술

한번 time tick이 발생할 때 마다 핀토스는 timer\_interrupt 함수를 호출한다. 이를 이용해 timer\_interrupt 함수에서 전체 block된 thread 들이 저장된 block\_list를 순회한다. Block\_list를 돌며 저장된 wake\_time이 되어 wake up 해야 할 thread가 있으면 이를 block\_list에서 삭제하고 해당 thread를 thread\_unblock을 불러 wake up 한다.

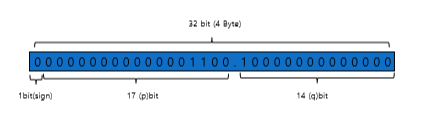
1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.

Priority scheduling에 따라 현재 실행중인 thread를 멈추고 높은 priority를 가진 thread가 실행되게 한다. 이를 구현하기 위해 list\_insert\_ordered를 사용하여 ready\_list가 priority에 따라 정렬되게 구현한다. 현재 실행중인 thread의 priority와 새로운 thread의 priority를 비교해 새로 들어온 thread가 priority가 더 높다면 thread\_yield 함수를 불러 해당 thread가 실행되게 해준다. 또한 semaphore를 동시에 이용할 경우 정렬이 깨질 수 있으므로 sema\_up, down 등의 함수에서도 priority에 따라 lock을 풀도록 synchronize 해준다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.

Advanced scheduler에서 priority 계산을 할 때 사용하는 계산식은 다음과 같다. *priority* = PRI\_MAX – (*recent\_cpu* / 4) - (*nice* \* 2). 이 때 PRI\_MAX는 priority의 최댓값이고, recent\_cpu는 최근 사용한 cpu time, nice는 priority를 높이는 방향으로 갈지, 낮추는 방향으로 갈지에 대한 변수이다. 그리고 recent\_cpu를 구하는 공식은 또한 다음과 같다. *recent\_cpu* = (2 \* load\_avg) / (2 \* load\_avg + 1) \* *recent\_cpu* + nice. 여기서 load\_avg는 ready state의 평균적인 thread 수이고, nice는 위와 같다. 그리고 load\_avg를 계산하는 식은 다음과 같다. *load\_avg* = (59/60) \* *load\_avg* + (1/60) \* *ready\_threads.*

여기서 ready\_threads는 현재 ready state의 thread 개수이다. 해당 공식들을 이용해 load\_avg부터 구한 후 recent\_cpu를 구하고, 이를 이용해 마지막 priority를 계산할 수 있다. 한가지 주의할 점은, pintos는 부동 소수점 연산을 지원하지 않기 때문에 4bit짜리 변수를 사용해 다음과 같이 fixed point 연산을 해야 한다.



1. **추진 일정 및 개발 방법**
   1. **추진 일정**

12/4: 명세서 확인 및 구현 내용 파악

12/5: alarm clock, priority 구현

12/6: aging 기법 적용, BSD scheduler 구현.

12/7: 보고서 작성

* 1. **개발 방법**
  2. Blocked 상태의 스레드를 어떻게 깨울 수 있는지.

이를 위해 먼저 block\_list라는 새로운 list를 선언하고, timer\_init에서 list를 초기화시킨다. 이후 timer.c의 timer\_sleep함수에서 sleep해야할 thread를 block\_list에 삽입하고 thread\_block 하도록 코드를 수정하고, timer\_interrupt함수에서는 block\_list를 순회하며 깨어나야 할 thread를 찾아 thread\_unblock을 불러 wake up 하도록 구현한다.

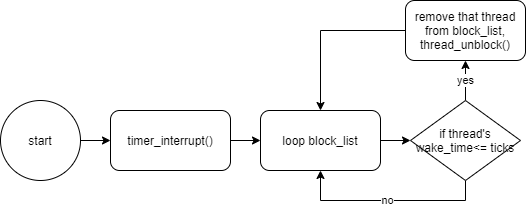
* 1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지.

이를 위해 thread를 ready\_list에 넣는 함수들, thread\_unblock과 thread\_yield 함수를 수정한다. 이전에는 해당 함수에서 그냥 list\_push\_back을 통해 넣어주었다면 이제 list\_insert\_ordered를 이용해 priority 순으로 정렬되도록 넣어준다. 또한 새로운 thread가 더 큰 경우 thread\_create에서 마지막에 현재 실행중인 thread의 priority와 새로운 priority와 비교해 바꿔야 한다면 thread\_yield를 부르도록 구현한다. 정렬이 깨지지 않기 위해 sema\_down 함수에서도 waiter list를 list\_insert\_ordered로 넣어서 정렬을 맞춰준다.

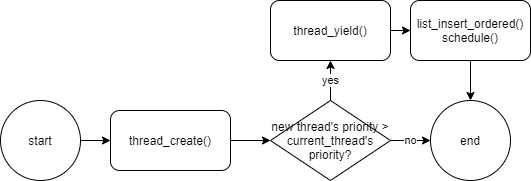
* 1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.

먼저 pintos에서 소수점 연산을 하기 위해 fixed point 연산을 위한 함수들을 구현한다. 또한 thread 구조체에 priority 연산에 필요한 정보를 저장할 변수 cpu, nice를 추가하고, thread.c에서 사용할 load\_avg 변수를 추가한다. 이후 위에서 나온 식을 이용해 recent\_cpu와 load\_avg를 갱신하는 함수를 추가한다. 이후 해당 함수를 사용해 새로운 priority를 계산하고 이를 모든 thread에 대해 갱신하도록 구현한다. 그리고 timer.c에서 매 1 tick 마다 cpu와 load\_avg를 갱신하고, 매 4 tick 마다 priority를 갱신 후 scheduling 하도록 코드를 추가한다.

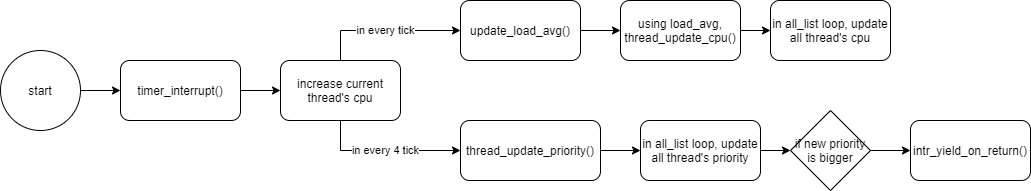
1. **연구 결과**
   1. **Flow Chart**
2. Blocked 상태의 스레드를 어떻게 깨울 수 있는지



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

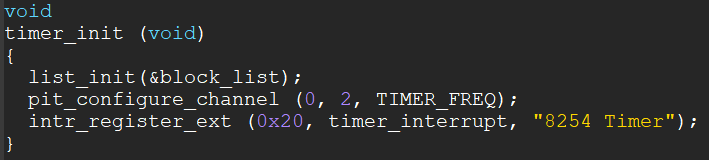


1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술



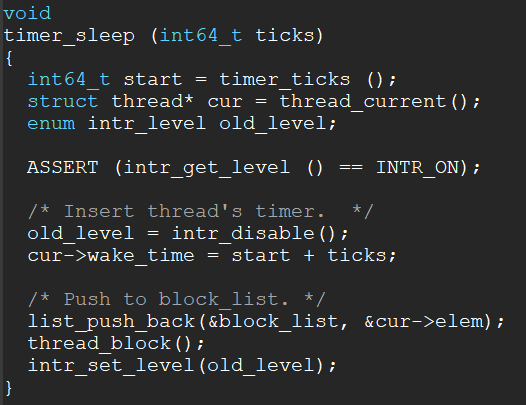
* 1. **제작 내용**

1. Timer\_init



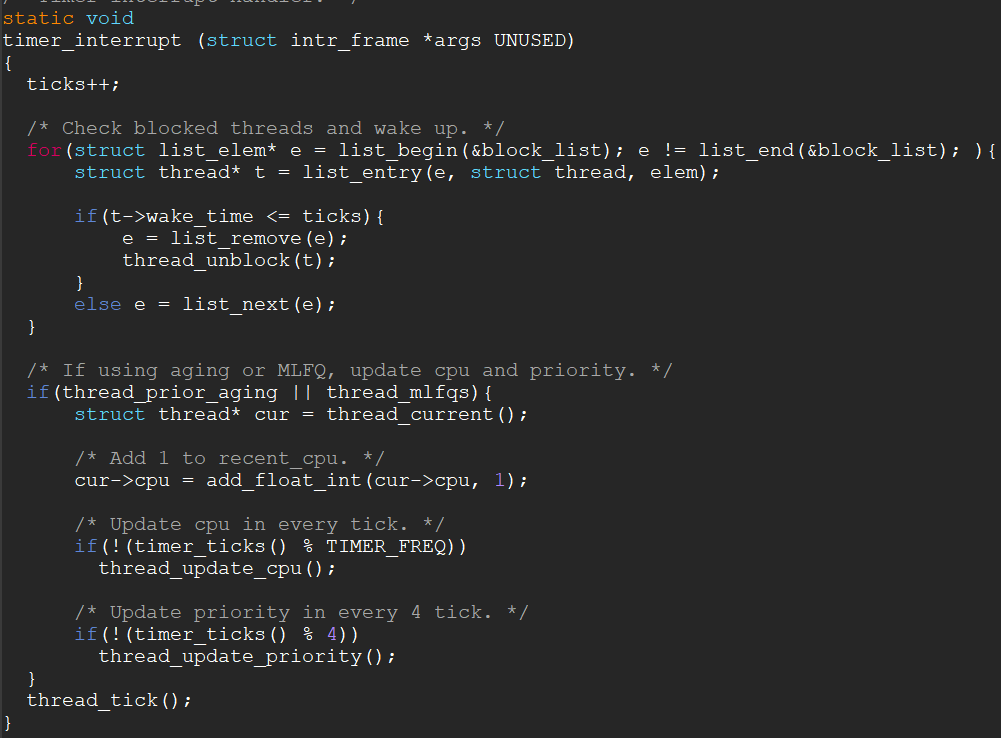
새로 추가한 block\_list를 list\_init을 이용해 초기화해준다.

1. Timer\_sleep



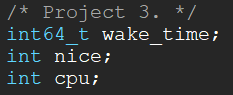
Timer\_sleep 함수에서는 while을 이용한 busy waiting 기법을 버리고 block\_list를 사용한다. Thread의 wake\_time 변수에 깨어나야 할 시간을 저장하고 해당 thread를 block\_list에 list\_push\_back을 이용하여 넣는다.

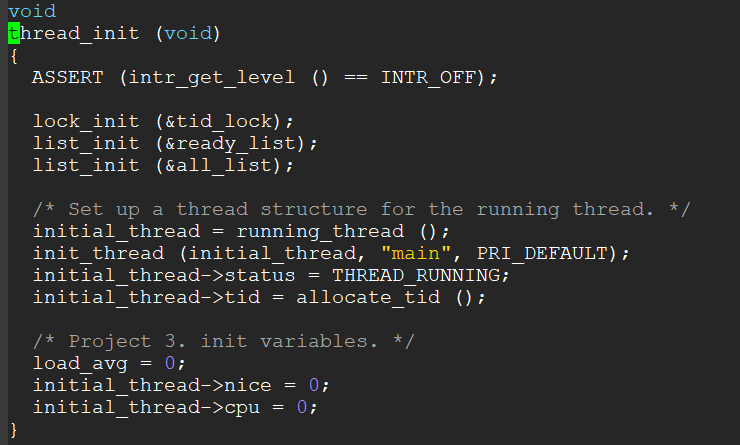
1. Timer\_interrupt



Thread를 wake up 하는 코드는 위의 for문으로, block\_list의 모든 thread를 순회하며 wake\_time이 ticks 이하인 thread를 찾는다. 해당 thread는 wake 시간이 되어 이제 깨어나야 하므로 block\_list에서 삭제하고 thread\_unblock을 불러 ready\_list에 등록되게 해준다. 아래는 aging 기법과 MLFQ를 위한 부분으로, 만약 둘 중 하나의 flag가 true로 세팅되어 있다면 아래 코드를 실행해 aging을 적용시킨다. 먼저 현재 실행중인 cpu이 recent\_cpu를 증가시키고, 매 tick 마다 thread\_update\_cpu 함수를 불러 cpu와 load\_avg를 갱신한다. 그리고 매 4번의 tick 마다 priority를 업데이트 해야 하므로 thread\_update\_priority 함수를 부른다. 이 모든 과정이 끝났다면, thread\_tick 함수를 불러 알맞은 thread가 실행되도록 한다.

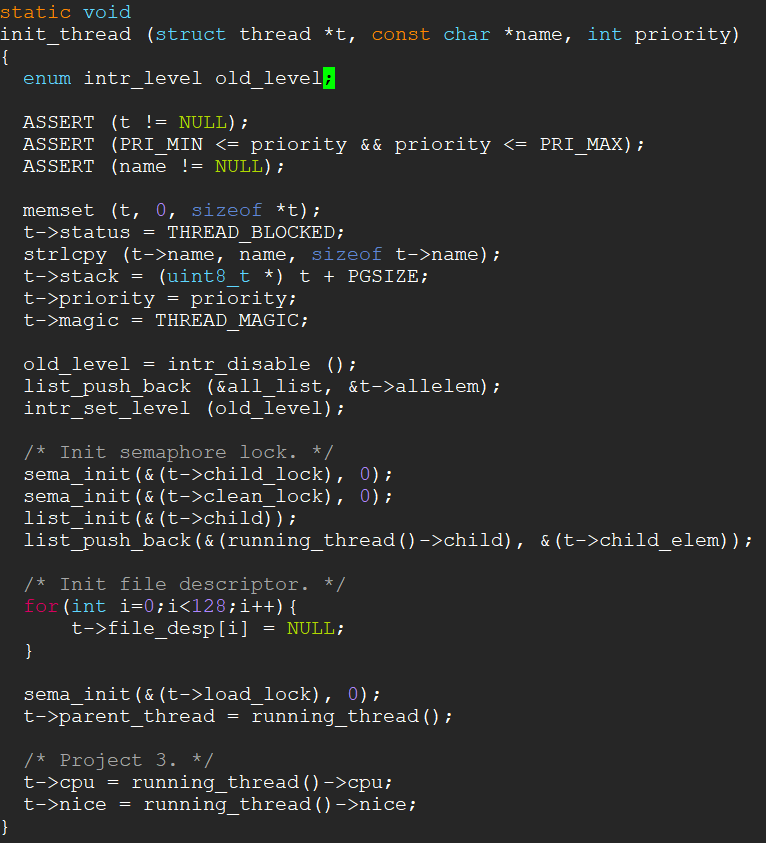
1. Thread\_init





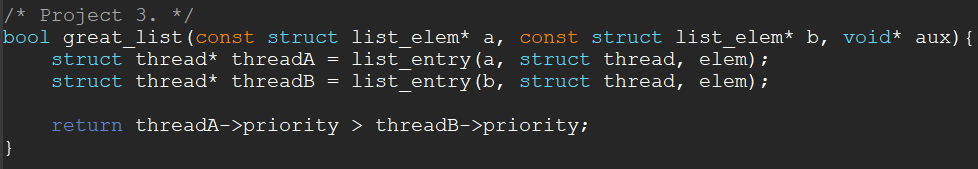
Thread 구조체에 위와 같은 새로운 변수를 추가하고 thread\_init 함수에서 이들을 초기값인 0으로 초기화해준다.

1. Init\_thread



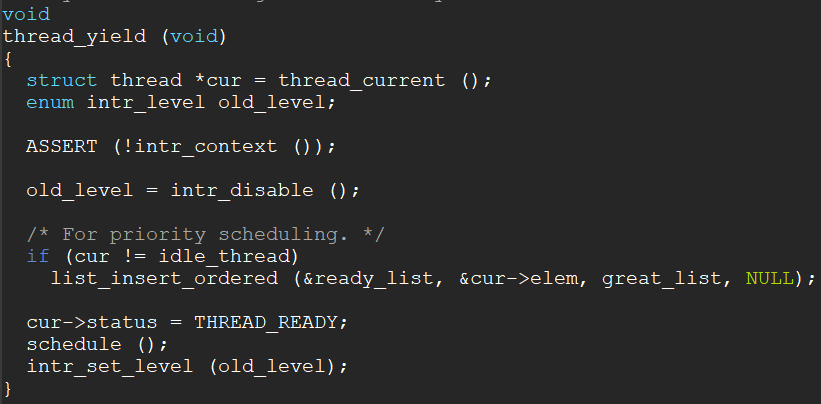
Thread의 recent\_cpu와 nice 값을 초기화해준다.

1. Great\_list



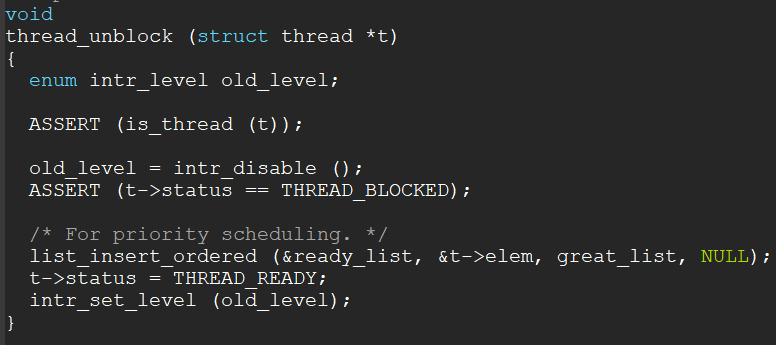
List\_insert\_ordered를 위해 필요한 함수로, 인자로 받은 thread의 priority를 비교해 큰 것이 앞으로 가도록 Boolean 값을 반환한다.

1. Thread\_yield



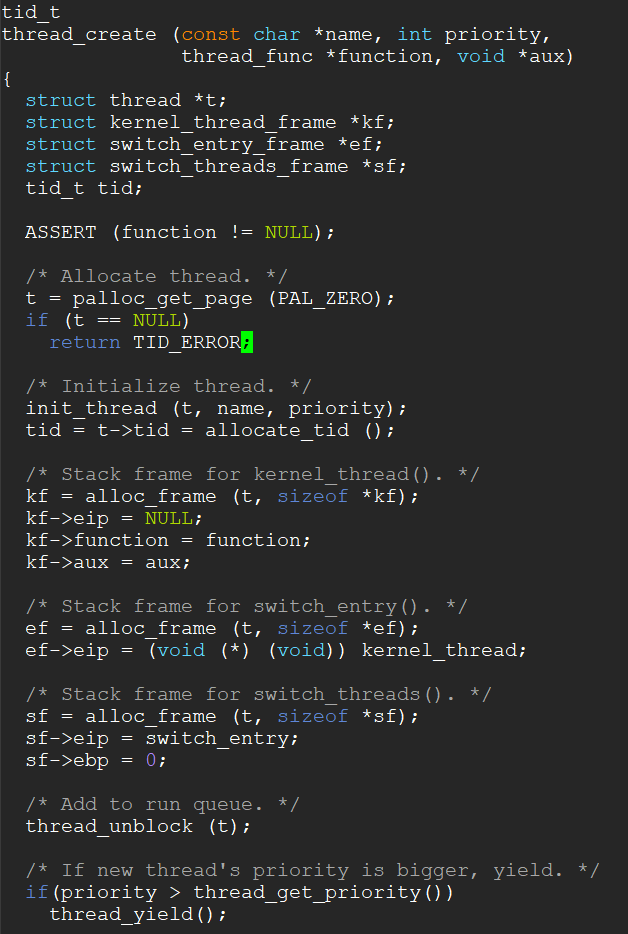
기존 list\_push\_back이었던 부분을 priority 적용을 위해 list\_insert\_ordered로 수정한다.

1. Thread\_unblock



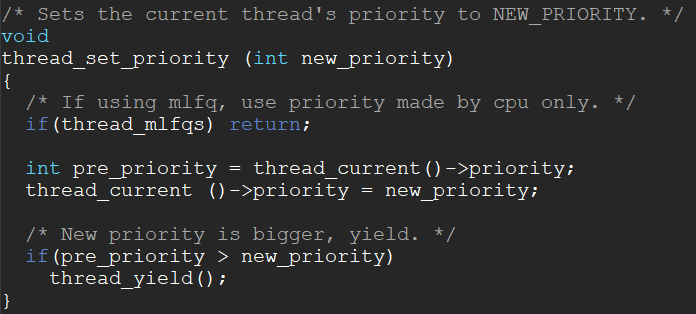
기존 list\_push\_back이었던 부분을 priority 적용을 위해 list\_insert\_ordered로 수정한다.

1. Thread\_create



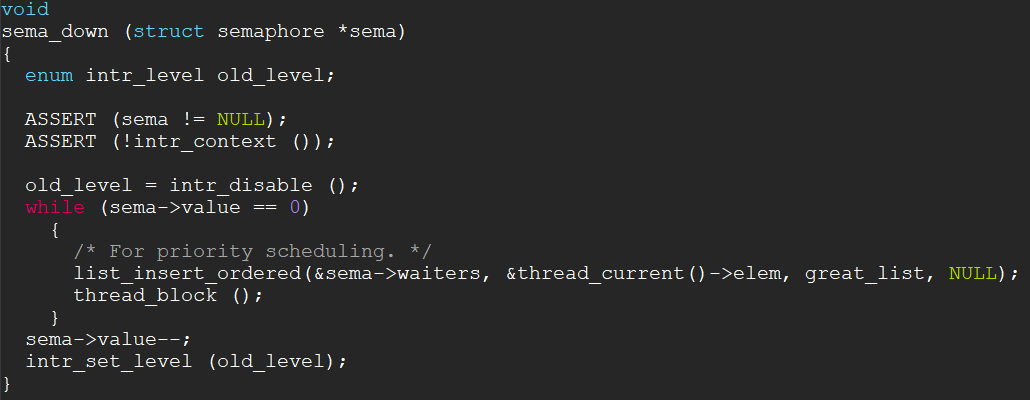
새로운 thread를 만드는 함수의 가장 마지막에 priority 검사하는 부분을 추가한다. 새로 들어오는 thread의 priority가 더 크다면 thread\_yield를 불러 새로운 thread가 바로 실행되도록 한다.

1. Thread\_set\_priority



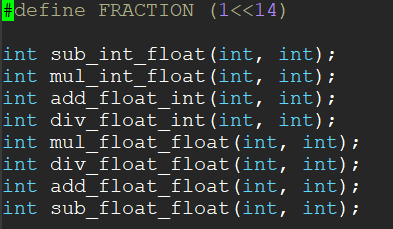
새로운 priority를 설정하는 함수로, 만약 MLFQ를 사용한다면 해당 scheduler는 cpu time에 의한 priority만 존재하므로 임의로 세팅할 수 없기에 return을 하여 함수를 종료한다. 현재 thread의 priority를 새로 받은 priority로 갱신하고, 기존 priority와 비교한다. 현재 실행중인 thread의 priority가 가장 크다는 것은 당연하므로 만약 새로운 priority가 기존 priority보다 작다면, 실행중인 thread가 다른 thread보다 priority가 작아질 수 있다. 그러므로 thread\_yield를 불러 새로운 가장 높은 priority의 thread가 실행되게 해준다.

1. Sema\_down

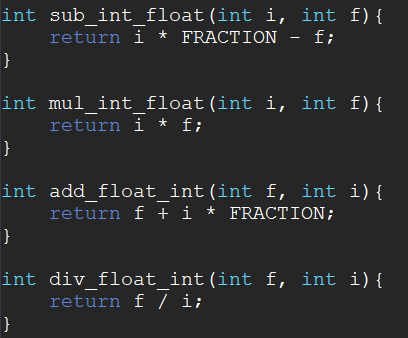


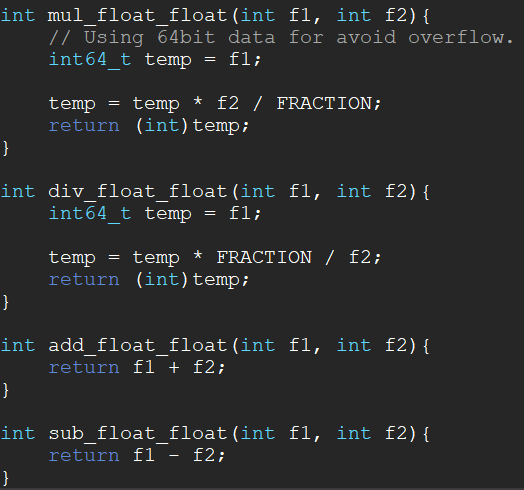
이전 semephore는 waiter thread의 priority를 고려하지 않고 먼저 들어온 thread 부터 block을 해제했다. 하지만 이 경우 priority의 정렬이 깨질 수 있으므로 list\_insert\_ordered 함수를 사용해 waiters list도 priority로 정렬되게 한다. 이를 통해 synchronization을 맞춘다.

1. Arithmetic.c



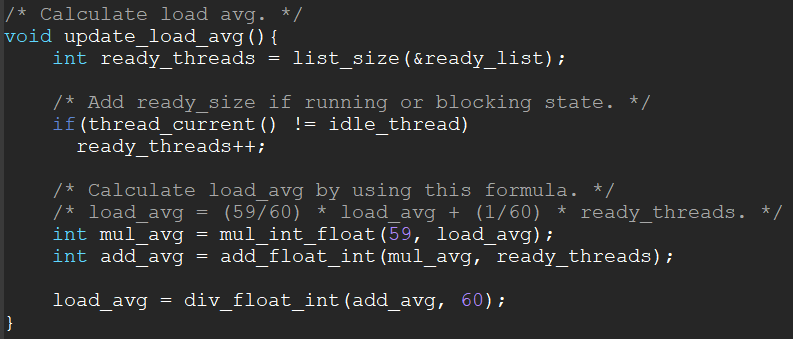
Fixed point 연산을 위한 함수들을 선언한다.





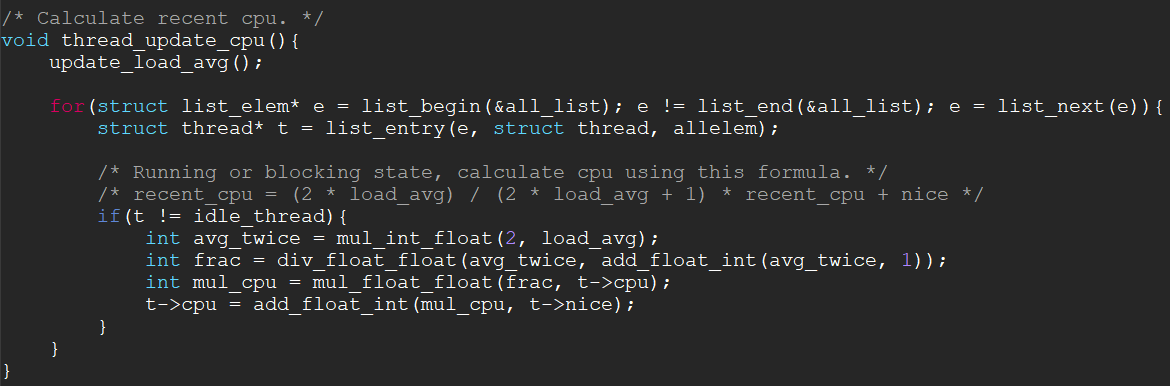
Pintos 매뉴얼에 기재된 fixed point 연산 방법을 참고해 각 연산을 구현한다. 그리고 float float 연산의 경우, float를 곱하는 과정에서 overflow가 발생할 수 있으므로 64bit짜리 data를 임의로 사용해 overflow를 막는다.

1. Update\_load\_avg



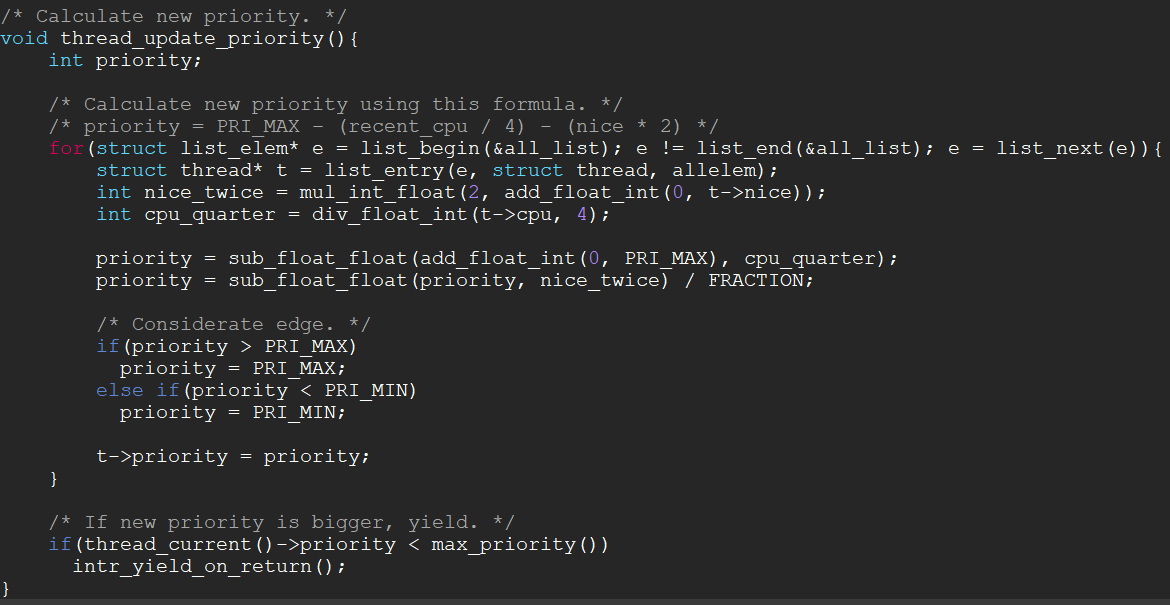
Load\_avg를 갱신하는 함수이다. 위에 언급한 공식을 사용하며 ready\_threads는 ready\_list에 있는 전체 thread 개수와 같으므로 이를 구한다. 그리고 현재 thread가 running 상태가 아니라면 ready\_thread를 1 증가시켜 올바른 ready\_threads를 구한다. 이후 해당 ready\_threads를 사용해 공식에 적용한다.

1. Thread\_update\_cpu



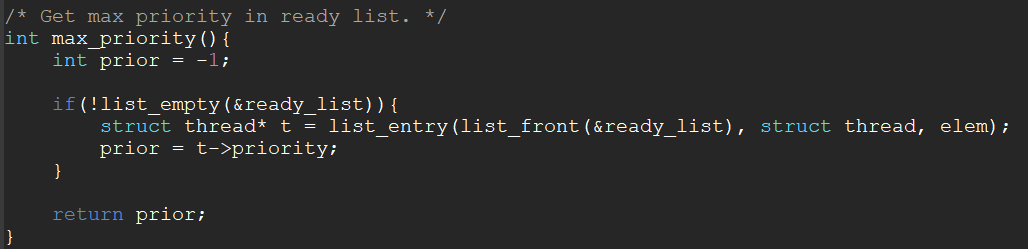
전체 thread의 recent\_cpu를 갱신한다. All\_list를 순회하며 현재 running 상태가 아닌 thread의 recent\_cpu를 갱신한다. 이전에 update\_load\_avg 함수를 불러 미리 load\_avg를 갱신하고 이를 위에 언급한 공식에 넣어 사용한다.

1. Thread\_update\_priority



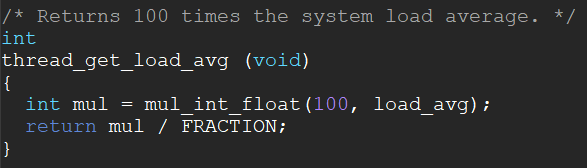
전체 thread의 priority를 갱신한다. 위에 언급된 공식을 사용하며, 만약 계산한 priority가 최대, 최소 priority를 넘는 경우 해당 edge에 따라 최대, 최소 값을 가지도록 한다. 만약 priority를 update했는데 현재 실행중인 priority보다 큰 것이 있다면 intr\_yield\_on\_return 함수를 불러 새롭게 가장 높은 priority의 thread를 실행하게 한다.

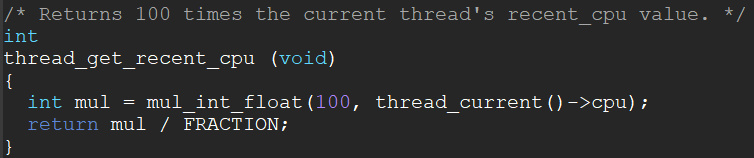
1. Max\_priority



현재 ready\_list에 있는 thread 중 priority에 따라 정렬되어 있으므로 front의 thread’s priority를 반환해준다.

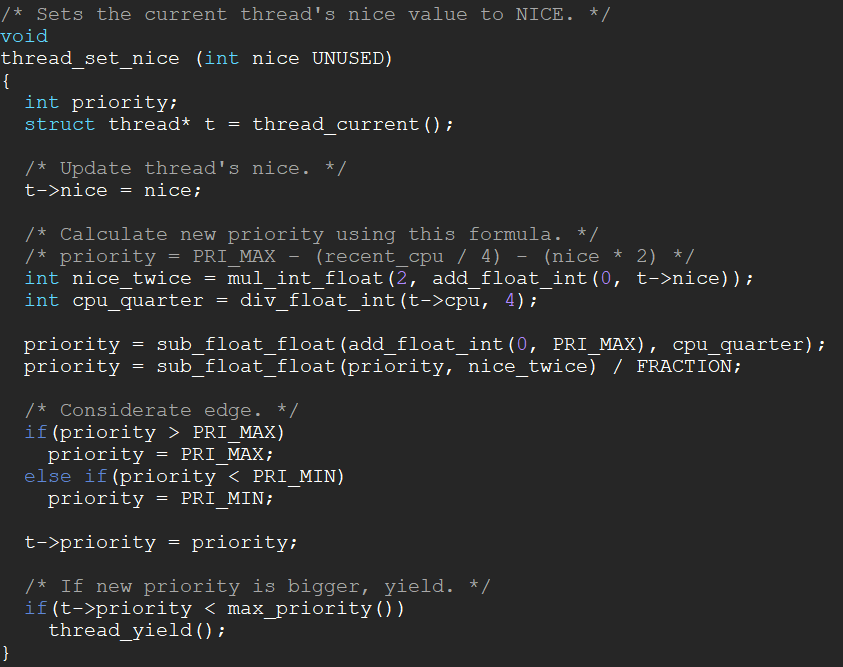
1. Get\_load\_avg, get\_recent\_cpu





주석에 따라 현재 load\_avg와 recent\_cpu를 100배 한 값을 반환한다.

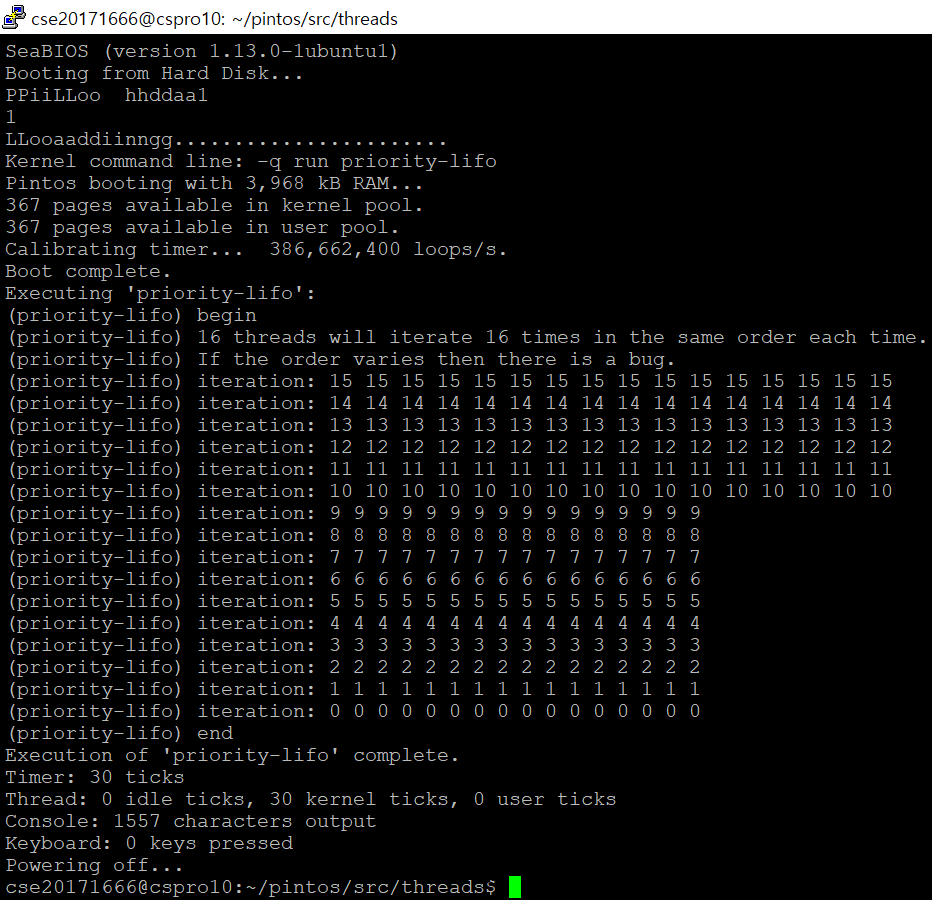
1. Set\_nice



Nice를 새로 갱신하는 함수이다. 새로 갱신했다면 priority 또한 달라지므로 위의 priority 식을 이용해 새로운 priority를 계산한다. 이 또한 최대 최소를 넘지 않도록 조정한 후 thread에 넣어준다. 여기서도 만약 새로운 priority가 ready\_list의 가장 큰 priority보다 작아진다면, thread\_yield를 불러 현재 thread가 아닌 가장 큰 priority thread가 실행되게 한다.

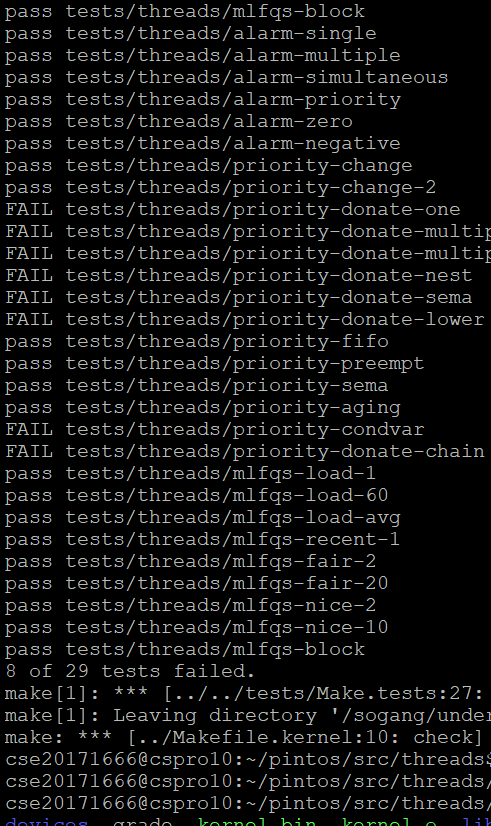
* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

priority-lifo 테스트 결과



Priority-lifo 테스트는 같은 priority를 가지는 16개의 thread를 16번 반복하는 코드로, 같은 priority이기 때문에 순서가 바뀌면 안된다. 이에 대한 결과는 위와 같으며, 순서가 바뀌지 않고 올바르게 된 것을 확인할 수 있다.

make check 수행 결과



Ppt의 모든 테스트에 대해 성공적으로 pass된 것을 확인할 수 있다.