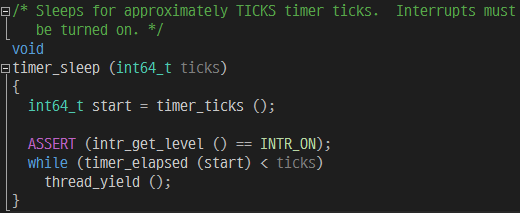
**Project 1 Final Report**

성해빈, 송창근

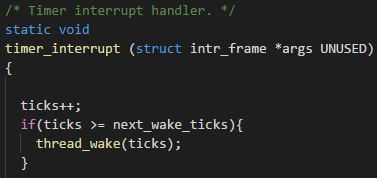
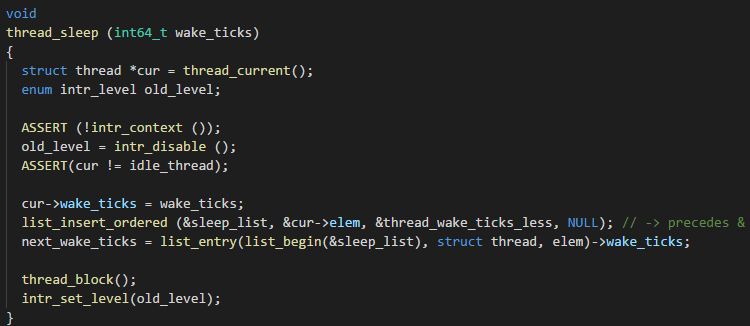
**1. 문제 해결**

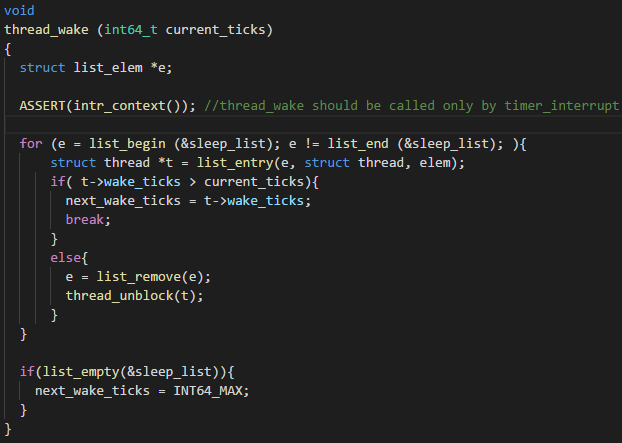
1. Alarm Clock
   1. 현재 구현의 문제



현재 timer\_sleep 함수는 매 틱마다 정한 틱이 지났는지 확인하여 지나지 않았으면 thread\_yield 함수를 호출해 자신에게 할당된 시간을 다른 스레드에게 돌린다. sleep 함수는 해당 스레드를 처리하지 않기 위해 존재하는 기능인데, 이와 같이 구현하면 스레드를 처리하지 않기 위해 계속해서 스레드에 접근해야 하는 모순이 생긴다. 이는 당연하게도 매우 비효율적이며, 성능에 악영향을 미친다.

* 1. 해결 방법 및 알고리즘

thread\_yield가 아니라 thread\_block을 이용하여 스레드를 아예 정지시키고, 시간이 되면 thread\_unblock을 이용하여 풀어주도록 한다. thread\_yield 는 계속 자동으로 ready\_list에 추가하기 때문에 상당히 비효율적이다. sleep\_list를 따로 만들어 일정 시간이 지날 동안 ready\_list에 넣지 않고 시간이 지나야 ready\_list에 넣는다. 이를 위해서는 정지한 스레드와 깨어날 시간을 저장해야 한다. timer\_sleep 함수가 호출되면 스레드 번호와 깨어나야 할 ticks을 저장한다. 여러 스레드가 동시에 sleep 될 수 있으니 깨어날 시간이 가까운 순서로 정렬하여 리스트에 삽입한다. 이후 해당 스레드를 block한다. thread\_tick 에 현재 시스템의 ticks을 확인하여 리스트의 가장 앞에 있는 스레드의 시간(next\_wake\_ticks)이 지났으면 해당 스레드에 thread\_unblock을 하는 함수를 넣는다.



sleep\_list

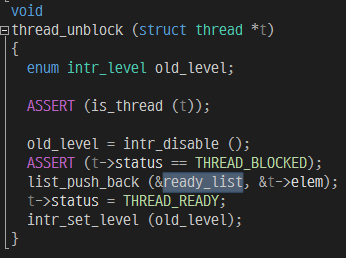
(sorted by next\_wake\_tick)

Thread\_wake

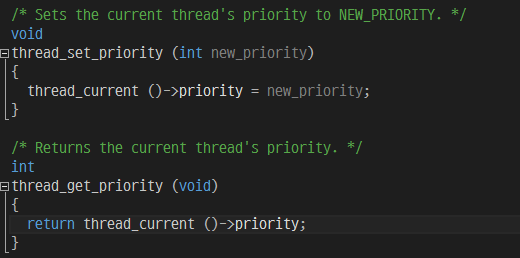
(external interrupt)

Ready\_list

1. Priority Scheduling
   1. 현재 구현의 문제



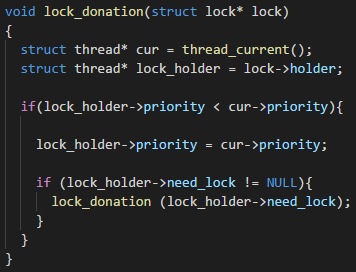
스레드는 ready\_list에 저장된 순서로 실행된다. 그런데 현재 구현은 스레드를 ready\_list에 넣을 때, list\_push\_back 함수를 이용해서 그냥 리스트의 맨 마지막에 넣는다. 즉, 스레드는 계속 순서대로 실행된다. 그러나 실제 OS에서는 먼저 처리해야 하는 스레드들이 있다. 예를 들어, 작업 관리자 등은 우선적으로 처리해야 할 것이다. 이러한 기능을 위해 스레드에 우선순위가 있다. 우선 순위를 처리할 때 다양한 문제들이 생길 수 있는데, 이를 전부 해결해야 한다. 예를 들어, 높은 우선순위 작업을 처리하기 위해 필요한 자원을 낮은 우선순위 작업이 가지고 있다면(lock했을 때), 그 때는 낮은 우선순위 작업의 우선도를 일시적으로 올려줘야 한다.(donation)

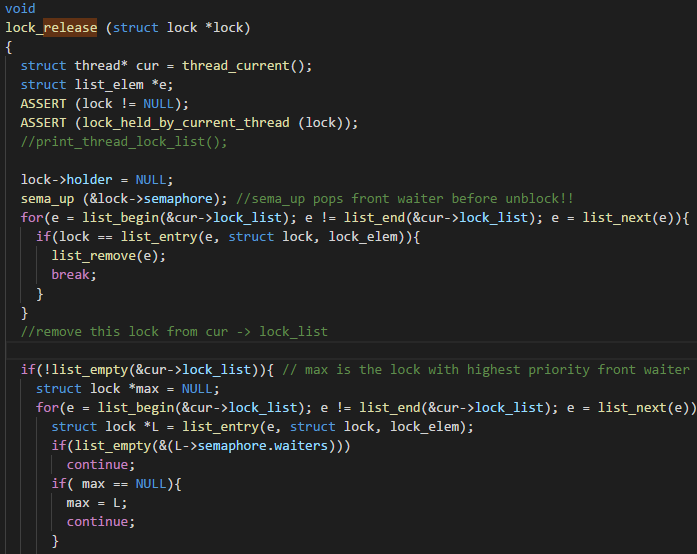


또한, 우선순위를 제대로 구현하기 위해 우선순위를 설정하는 함수와 우선순위를 알아내는 함수가 있는데, 우선순위를 설정하는 함수의 경우, 우선순위가 바뀜으로 인해 현재 실행 중인 스레드가 최고 우선순위가 아니게 된 경우 그 스레드를 yield해야 하고, 우선순위를 받아오는 함수의 경우 donation까지 고려하게 함수를 수정해야 한다.

* 1. 해결 방법 및 알고리즘

스레드가 가질 수 있는 상태는 RUNNING, READY, BLOCK, DYING이다. 여기서 DYING는 스레드가 삭제되므로 READY 될 수 없음을 감안하면 ready\_list에 스레드가 삽입되는 경우는 thread\_yield와 thread\_unblock 때임을 알 수 있다. 해당 함수들의 list\_push\_back 부분을 list\_insert\_ordered로 새로 만들어준다. 이를 위해서는 대소비교를 하는 함수(list\_less\_func)를 만들어주어야 하는데, 스레드의 우선순위를 비교하게 하면 된다.





Donation을 구현하기 위해 thread 구조체를 수정하여 현재 가지고 있는 lock들의 리스트 lock\_list와 현재 필요로 하는 lock인 need\_lock을 구조체의 원소로 추가한다. 또한 스레드의 원래 priority를 담는 priority\_orig도 추가한다. 이 원소들은 걸맞는 함수들에서 수정한다.

sema\_down에서 waiter에 스레드를 집어넣을 때 list\_push\_back가 아니라 우선순위별로 넣도록 list\_insert\_ordered를 이용하여 넣는다. lock을 해 줄 때는 donation을 진행해야 한다. 해당 lock의 holder의 우선순위를 현재 스레드의 우선순위와 비교하여 현재 스레드의 우선순위가 더 크다면 lock->holder의 우선순위를 현재 스레드의 우선순위로 바꾼다.

release의 경우에는 donation한 우선순위를 다시 되돌려야 하는데, 현재 스레드가 가진 lock\_list를 돌면서 lock의 waiter\_list 를 확인하여 최고의 우선순위를 찾아 우선순위를 바꿔준다. 만약 더 이상 lock\_list에 lock이 없다면 priority\_orig로 원상복구하면 된다.

Lock\_list

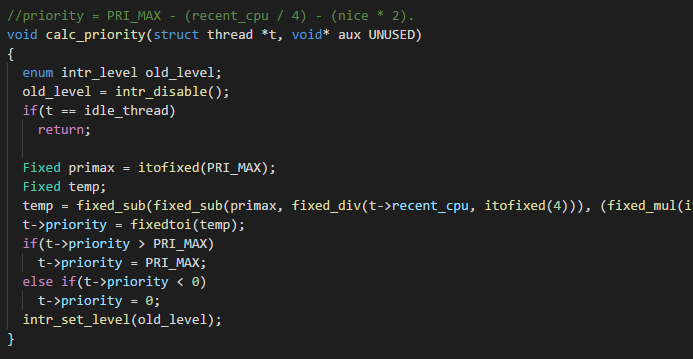
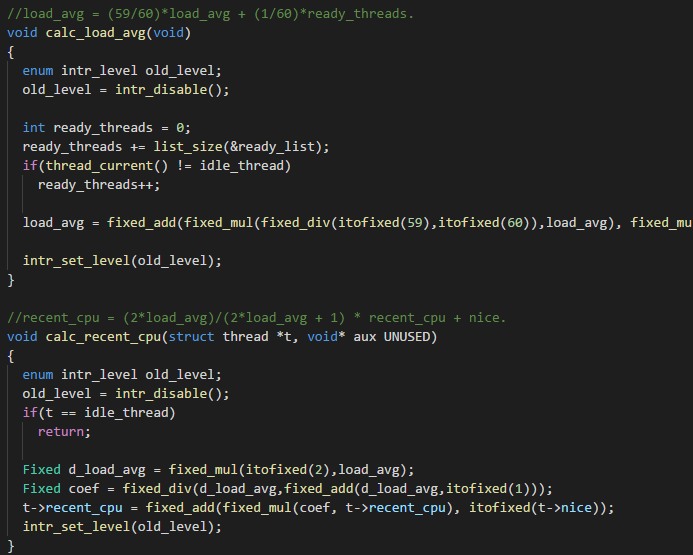
Semaphore.waiters

(sorted by priority)

1. Advanced Scheduler
   1. 현재 구현의 문제

새로운 스케줄러(MLFQS)를 구현한다. 현재 구현으로는 우선순위가 자동으로 고쳐지지 않는다. 또한 –mlfqs 커널 옵션을 통해 스케줄러를 선택할 수 있도록 한다.

* 1. 해결 방법 및 알고리즘



priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2) 가 우선순위를 결정하는 공식이다. recent\_cpu는 해당 스레드가 최근에 사용한 CPU 시간이며, nice는 해당 스레드의의 nice값이다. 자세한 것은 부록에 적혀있지만, MLFQS에서는 우선순위가 4틱마다 자동으로 계산되고 임의로 수정할 수 없는데, nice값을 조정함으로서 간접적으로 우선순위를 수정한다. nice는 크면 클수록 스레드에서 점유율이 낮아진다. priority 계산을 위하여 Fixed Point 연산을 구현한 뒤, load\_avg, recent\_cpu를 계산하고 nice를 수정 및 출력하는 함수를 구현하고, timer\_interrupt에 부록의 규칙에 맞게 함수를 넣는다. 여기서는 ready\_list를 그대로 사용해 ready\_list를 새로 계산한 priority에 맞게 sort상태로 유지하는 것으로 구현하였다. List를 각 priority마다 64개로 구현하는 방법도 있지만, 우리의 전 코드를 최대한 활용하는 방법이라서 이 방법을 채택했다.

**2. Discussion**

스레드의 존재 및 OS가 그것을 관리한다는 사실은 이미 알고 있었지만 직접 구현하는 것은 매우 새롭고 신선한 경험이었다. 프로젝트 1에서는 우선순위의 개념과 그 중요성을 알게 되었고, 우리가 편하게 쓰는 기능들이 내부적으로는 정말 많은 과정을 통해 실현된다는 것을 배울 수 있었다. C언어로 OS를 어떻게 만드는지, 멀티 쓰레드 구현은 어떻게 하는지 궁금했는데, 결국 어셈블리 언어를 이용해 eip와 스택 상태를 저장해서 구현하는 것이었다. 맨 처음에는 전혀 코드를 이해 못했고, 무엇을 요구하는지도 잘 몰랐다. 하지만 인터넷에 검색도 해보고, 핀토스의 코드를 보고, 테스트에서 부르는 함수들을 보면서 서서히 이해가 되기 시작했다. 또 디버깅 툴로 printf나 ASSERT를 주로 활용했는데, 코드의 thread\_list\_stats 같이 현재 리스트 상태를 쭉 보여주는 함수를 만들어 디버깅에 사용하였다. 하지만 이 방법도 아무래도 한계가 있어 다음 프로젝트부터는 gdb를 사용할 계획이다. 또 list에 넣기 전에 반드시 remove를 해줘야 하는데, 이것을 간과해서 무려 2번이나 list가 망가져버려 이유도 모르고 헤맸던 기억이 있다. 기본으로 구현된 함수가 remove작업에 대한 코드가 하나도 없던 걸 모르고 막연히 될 것이라고 생각했던 게 이유다.