**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

학번 / 이름 : 20191626 / 이용욱

개발 기간 :2023. 11. 06 ~ 2023.11.18

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

thread와 관련된 추가적인 기능을 구현한다. alarm clock, priority scheduling (aging) 이 포함되며, 추가 구현으로 BSD scheduler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock 기존의 thread에서는 timer\_sleep()이 thread를 sleep시킨다. 이 때 적절한 timer 만료가 이루어지지 않음에도 불구하고 thread\_yield() 함수가 계속 호출되어서 thread의 state를 running, ready 에서 반복되게 한다. 이는 효율적이지 못한 sleep 방법이므로 이보다 더 효율적인 sleep 방법을 통해서 효율적으로 바꾼다.
  2. Priority Scheduling ㄴ기존 pintos의 scheduling은 RR scheduling이었다. 다시 말해서 thread의 우선순위와 상관없이 thread\_yield 함수나 thread\_unblock 함수가 호출되면 thread가 항상 ready list 의 마지막에 추가된다는 사실을 의미한다. 그러므로 이번 project에서는 thread의 우선순위를 고려한 priority scheduling을 구현한다. 이 때 priority scheduling은 낮은 우선순위를 가지는 thread 의 경우 starvation을 가진다는 취약점이 있기에 이를 해결하기 위해 aging 또한 구현한다. 그러므로 일정 시간 후에 우선순위를 높여주어 starvation 현상을 없앤다.
  3. Advanced Scheduler (추가구현을 한 경우)

BSD sheduler를 구현한다. MLFQ( Multi- Level Feedback Queue ) 를 사용하여 각 우선 순위마다 각자의 ready queue를 가지도록 한다. schedule() 함수가 호출되면 thread는 가장 높은 우선순위의 queue를 선택하고 여기서 각 priority 의 ready queue가 Round Robin을 따르게 된다. 핀토스 매뉴얼에 제시된 nice, recent\_cput, load\_avg값을 이용해서 priority 를 계산한다. 이 때 , 핀토스 커널은 부동소수점을 지원하지 않으므로 이를 고려하여 계산한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

blocked 상태의 스레드를 관리하는 list를 새로 만들어서 해당 상태의 thread를 모두 저장한다. 그리고 각 tick마다 이 리스트에 있는 thread를 검사해서 wak\_up\_tick이 된 thread 를 list에서 제거하여 그 상태를 un\_blocked로 바꾸고 ready queue에 넣어준다. 이를 통해 blocked 상태의 thread를 깨울 수 가 있게 된다.

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

ready list에 running thread 보다 높은 우선순위를 가진 thread가 들어오면 thread의 우선 순위 순서에 따라거 높은 priority를 가진 thread 가 실행 되어야 한다. 이 때 , running thread 를 ready로 바꾸고, ready list에 우선 순위 순서에 맞추어 정렬되도록 running thread를 넣는다. 새롭게 들어온 thread도 마찬가지로 priority에 맞추어 ready list에 넣는다. 그리고 ready list에서 가장 우선되는 thread를 실행한다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

nice, load\_avg, recent\_cpu 값들을 계산해서 이를 바탕으로 priority 값을 계산한다. 처음에는 세 가지 값 모두 0으로 initializing 한다.

최종 priority 계산 식은 다음과 같다.

priority = PRI\_MAX - (recent\_cpu / 4) - (nice \*2 )

여기서 priority 값은 0 ~ 63 / PRI\_MIN ~ PRI\_MAX 사이의 값을 가진다.

**- nice**

-20 ~ 20 의 값을 가지는데, nice 값이 클수록 우선 순위는 낮아진다. 그리고 자식 thread는 부모 thread의 nice 값을 상속받게 된다. 이 때 nice 값에 변경이 있다면 thread\_priority 값을 다시 계산해야한다.

**-recent\_cpu**

해당 thread의 CPU time을 추정하는 변수이다. timer interrupt가 발생하면 , running state의 recent\_cpu 값은 1씩 증가하고, 모든 thread ( running, ready, blocked)의 recent\_cpu 값이 다시 계산된다. 이 부분에서의 계산식은 아래와 같다.

recent\_cpu = nice + (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu

**-load\_avg**

ready state의 쓰레드의 평균 개수를 나타내는 값이다. 이 값은 매 초마다 다시 계산된다. 이 부분에서 사용하는 계산식은 아래와 같다.

ready\_threads는 idle thread를 제외하고, ready state나 running state에 있는 thread의 개수를 의미한다.

load\_avg = (59 / 60) \* load\_avg + (1 / 60) \* ready\_threads

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

23.11.06 ~ 23.11.09 : Alarm Clock 구현

23.11.09 ~ 23.11.12 : Priority Scheduling 구현

23.11.13 ~ 23.11.18 : Advanced Sheduler 구현 및 보고서 작성

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수

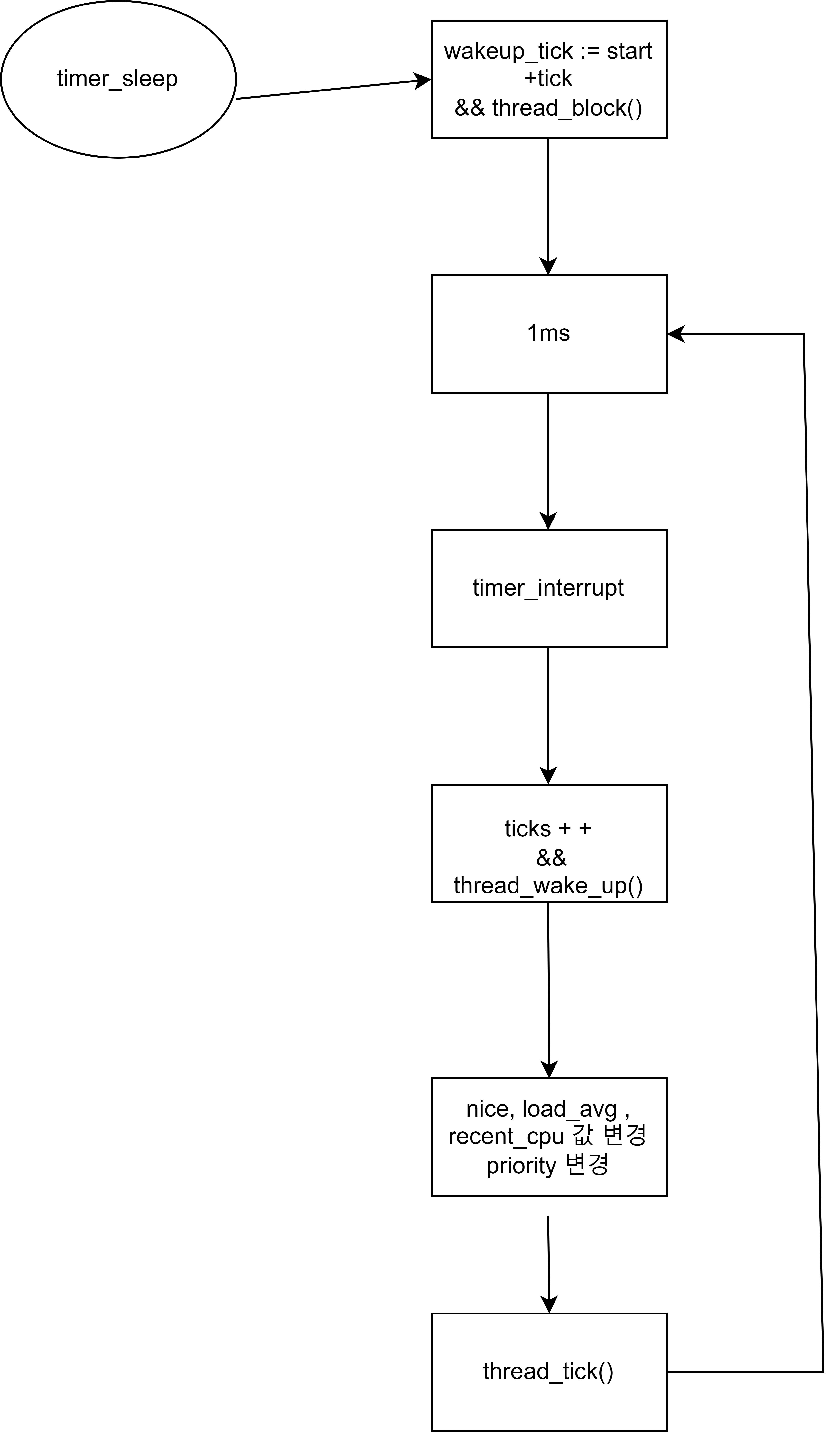
blocked state의 thread를 관리하는 자료구조인 sleep\_list를 추가한다. 또한 timer\_sleep() 함수에서 언제 쓰레드를 깨울 것인지를 저장하는 wakeup\_tick 변수를 thread 구조체에 추가한다. 쓰레드를 깨우는 것은 timer\_interrupt() 함수 내부에서 호출되는 thread\_wake\_up() 함수를 통해 이루어지도록 한다. 여기서 timer\_interrupt() 함수는 매 tick마다 call한다.

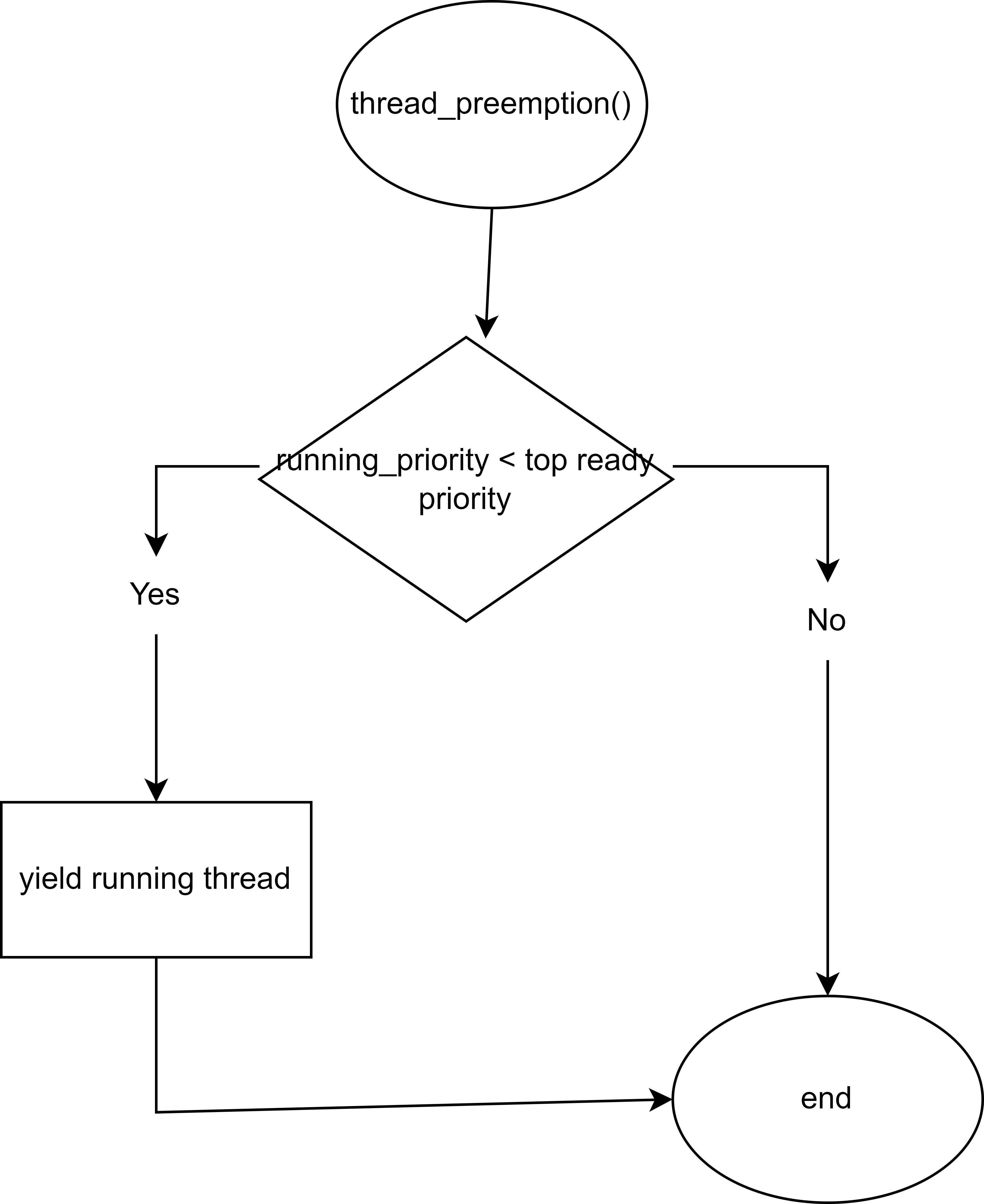
현재 실행중인 running thread보다 높은 priority를 가지는 thread가 들어오는 경우 ready\_list에 우선순위를 고려하여 추가하여야한다. 기존에 list\_push\_back() 함수가 호출되어 단순히 추가만 되는 상황과 달리, list\_insert\_ordered() 함수로 변경하여 순서에 맞게 추가하도록 바꾼다.. 이 때, 우선 순위를 비교하기 위해 priority\_comp 함수를 추가한다. 이렇게 ready\_list가 priority에 맞게 관리된다.

thread 구조체에 nice와 recent\_cpu 변수를 추가한다. 또한 update\_recent\_cpu() 함수와 update\_priority() 함수를 추가한다. nice 의 변경은 thread\_set\_nice() 함수에서, load\_avg 값과 recent\_cpu 값의 변경은 update\_recent\_cpu() 함수에서 실행하도록 하고, 최종priority 값의 변경은 update\_priority() 함수에서 실행하도록 한다.

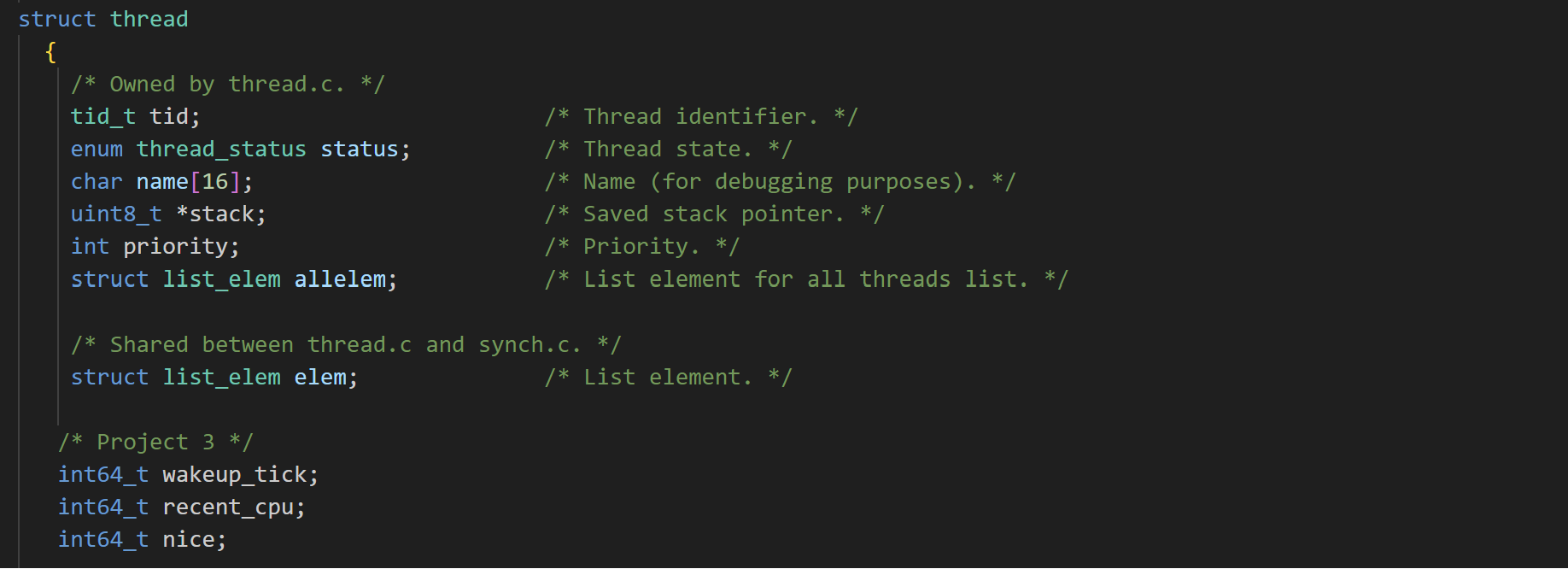
이 때 중요한 점은 부동소수점 계산을 실행해야 한다. pintos 명세서에 나와있는 계산 방법에 맞게 적절히 사용한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)
* Alarm Clock
* ****
* Priority Scheduling

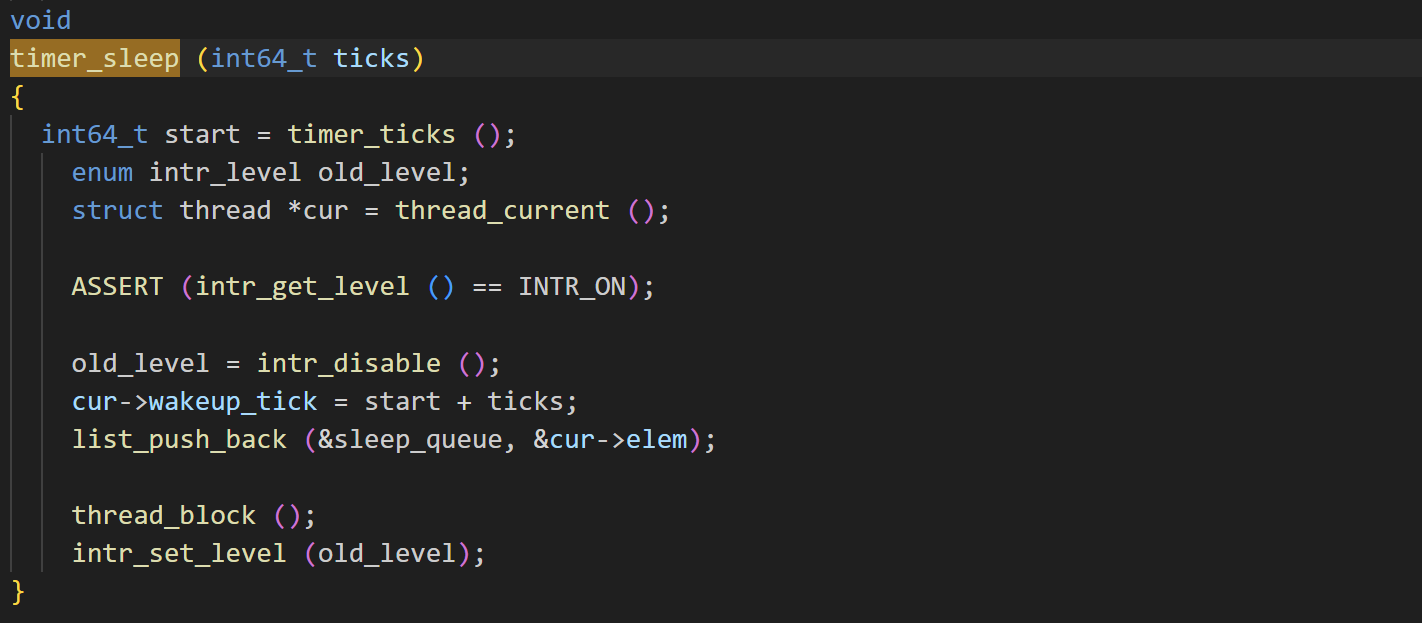
****

* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술thr할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

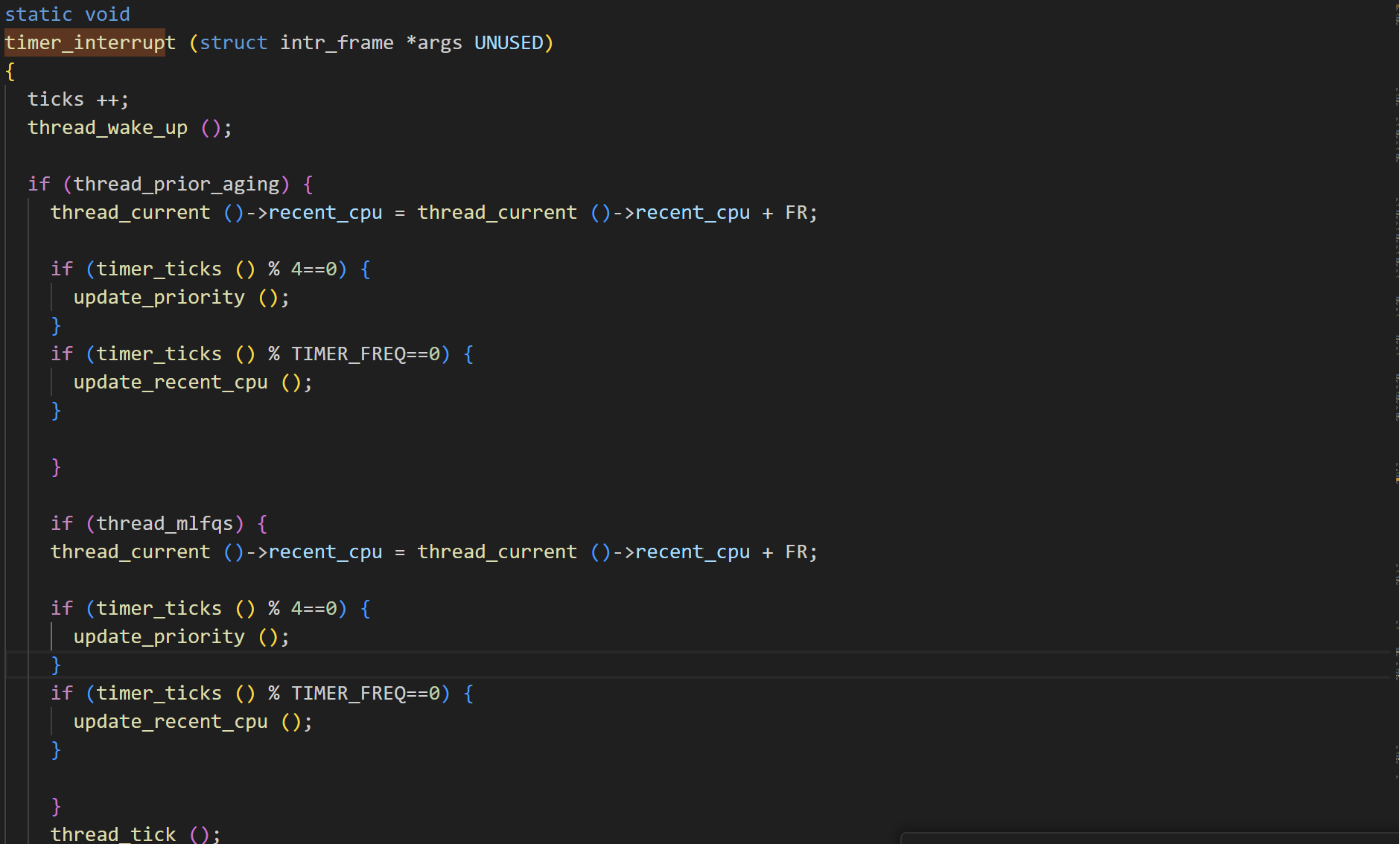
****

thread 구조체에 wakeup\_tick 변수와 추가 구현을 위한 recent\_cpu, nice 변수 등을 추가한다.

1. Blocked 상태의 thread를 깨우는 방법

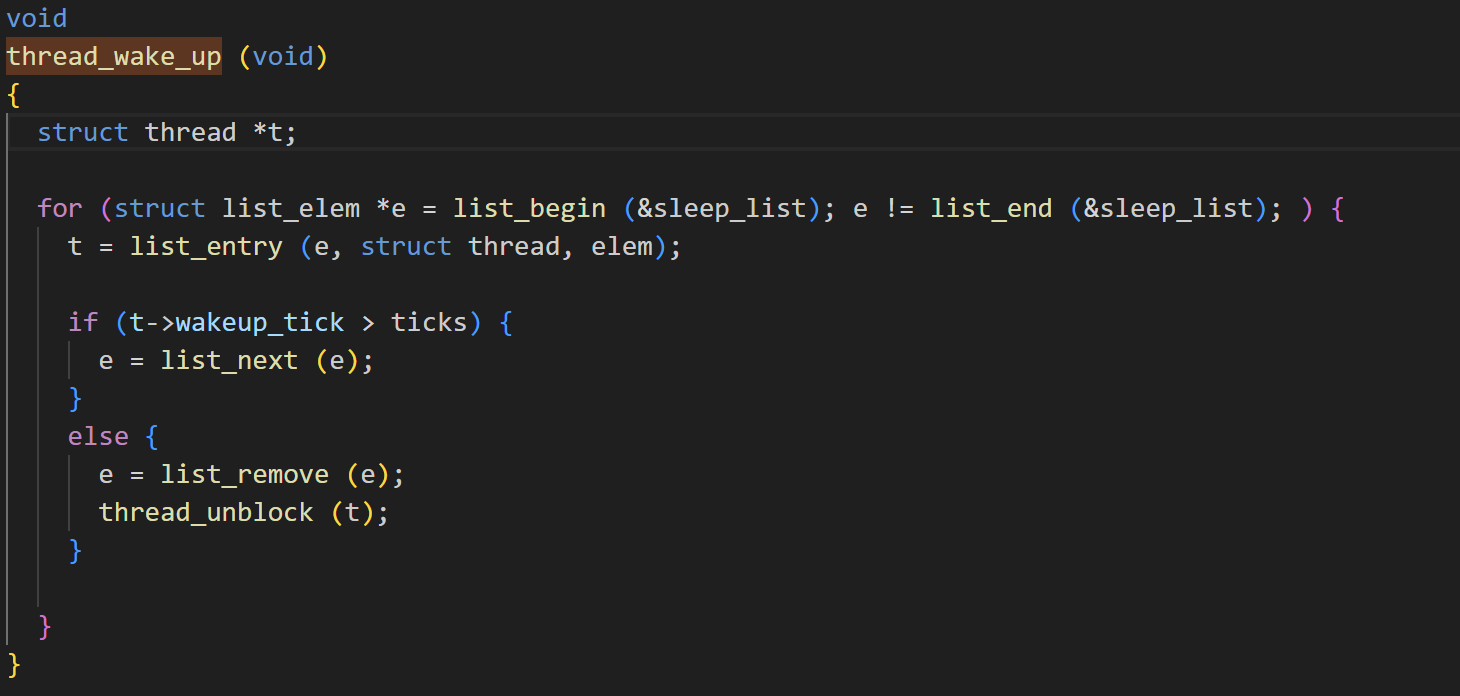


기존의 busy waiting을 지우고 현재 thread의 wakeup tick에 ticks , start를 합한 값으로 하고 다음 sleep\_list에 list\_push\_back() 함수를 사용해서 추가하고, thread\_block() 함수를 call 한다.



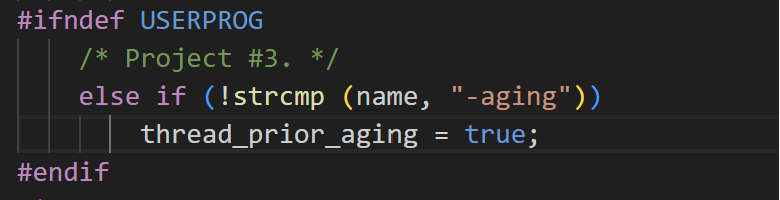
timer\_interrupt는ticks 증가 후 , thread\_wake\_up() 함수를 call.

ticks 변경시마다 thread를 찾아서 깨우는 역할을 한다.

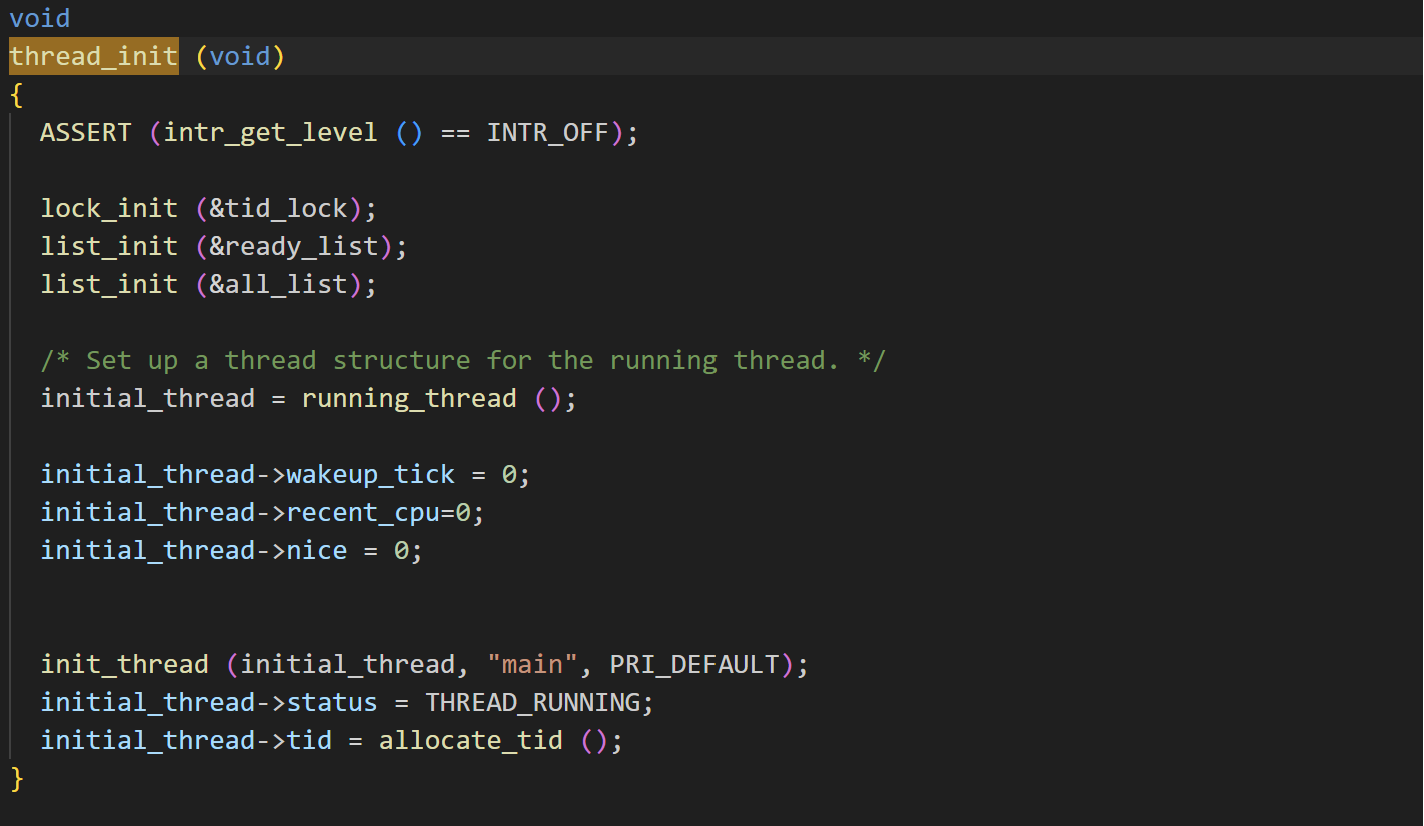


timer\_interrupt() 함수에서 호출되는 thread\_wake\_up 함수이다. sleep\_list를 전부 탐색해서 wakeup\_ticks(깨워야할 시간) 이 ticks보다 크면 다음 리스트에 추가하고,

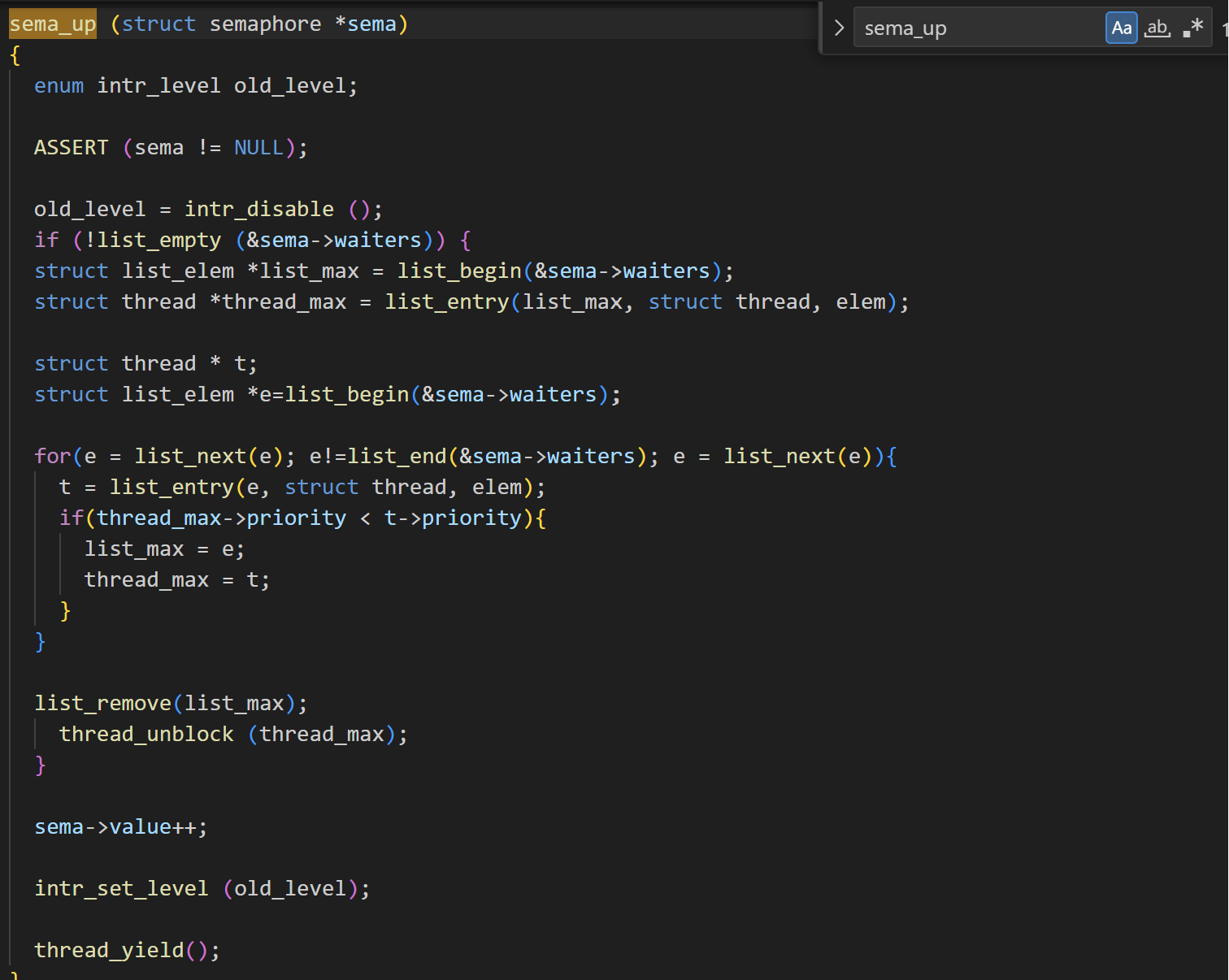
그렇지 않으면 sleep\_list에서 제거 후에 thread\_unblock함수를 호출해서 해당 thread를 unblocked상태로 바꿈으로써 그 thread를 깨운다.



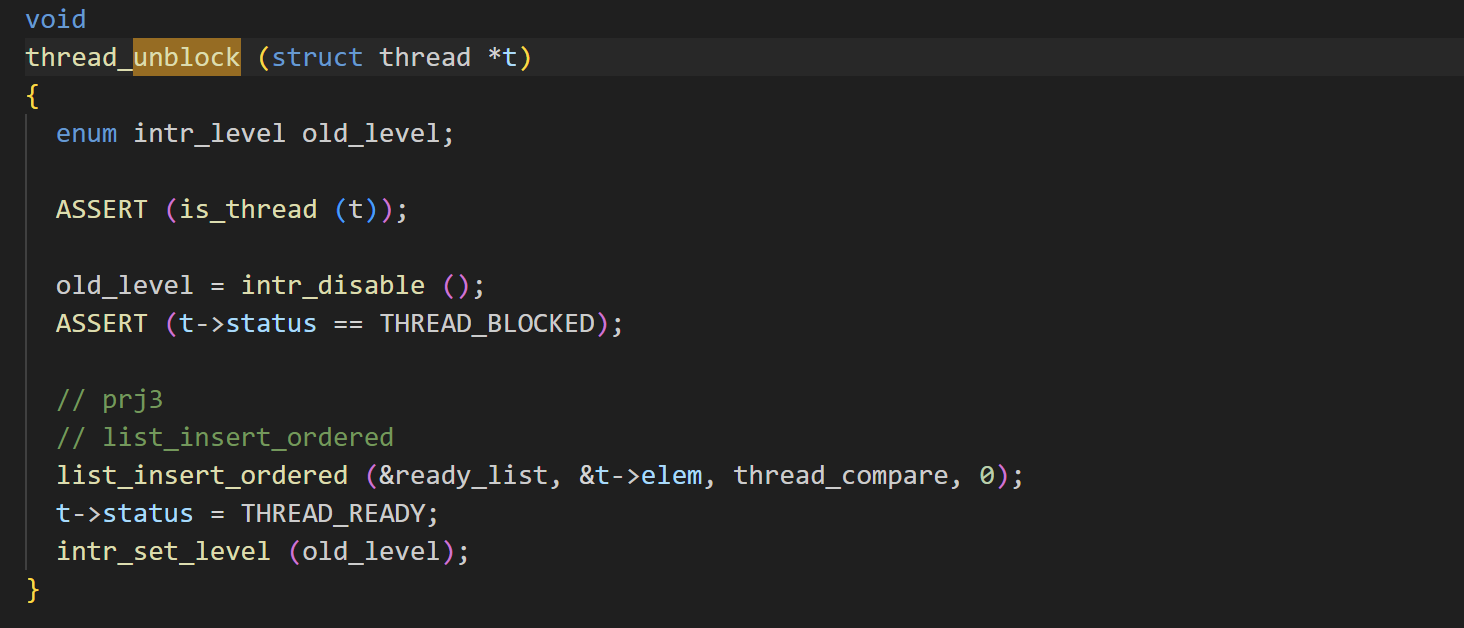
thread\_prior\_aging을 true로 설정합니다.

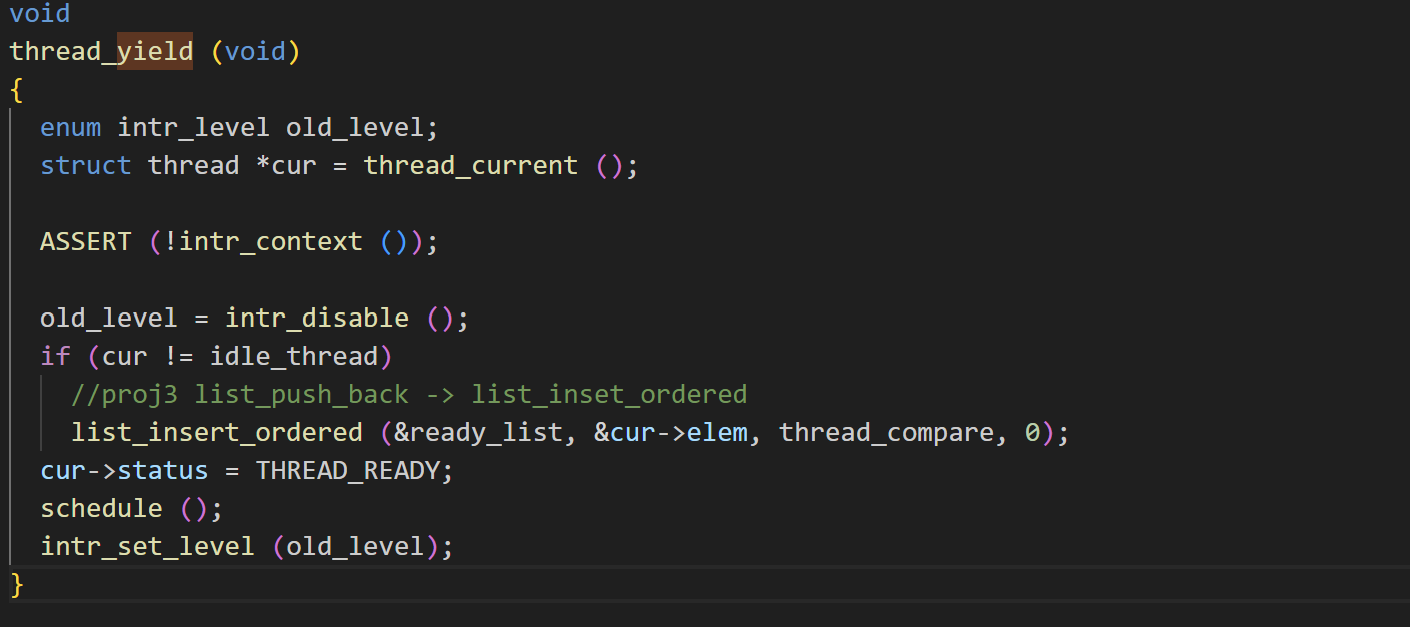


thread\_init 에서 wakeup\_tick , nice , recent\_cpu 를 모두 0으로 초기화 합니다.

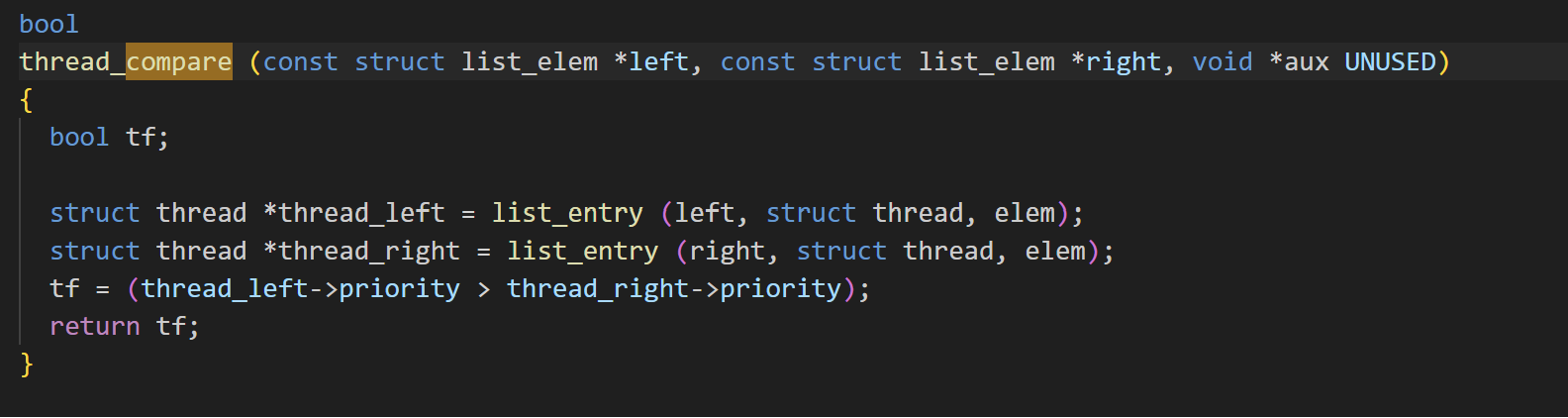


sema\_up 함수에서도 우선순위 관련 처리를 한다. 가장 큰 우선순위를 가진 thread를 찾아서 list\_remove 로 제거하고, thread\_unblock() 함수를 통해 unblocked 상태로 변경한다. 그러고 thread\_yield() 함수를 통해 re - scheduling 한다.





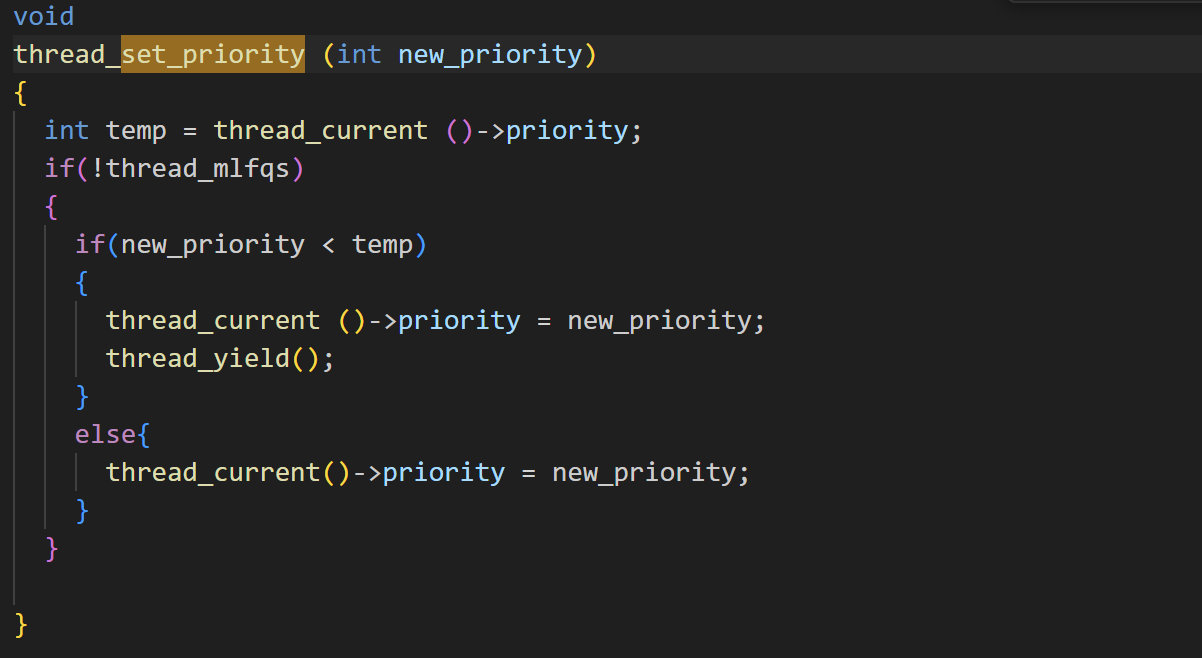
unblock , yield 함수에서 llist\_insert\_ordered 함수를 통해 넣는것으로 변경한다. 이때 우선순위 비교를 위해 thread\_compare함수를 구현한다.



그 내용은 위와 같다.

추가 구현에서는 nice, load\_avg와 관련된 함수들을 구현한다.

thread\_set\_priority()



현재 thread의 우선 순위를 변경한다.

thread\_get\_priority

현재 thread의 우선 순위를 return 한다.



thread\_set\_nice()

현재 thread의 nice 값을 매개변수로 받은 nice로 설정하는 함수이다.

현재 thread 의 nice값을 매개변수로 받은 nice 값으로 설정하고 필요한 계산식에 따라 계산후 재 설정 한다.

그 값은 pintos 매뉴얼에 나온 연산 방법을 사용한다.

priority = PRI\_MAX - (recent\_cpu/4) - (nice \*2)

priority 재설정 할 때, PRI\_MIN PRI\_MAX 조건을 확인한다.

thread\_get\_nice()

현재 thread의 nice 값을 return 한다.

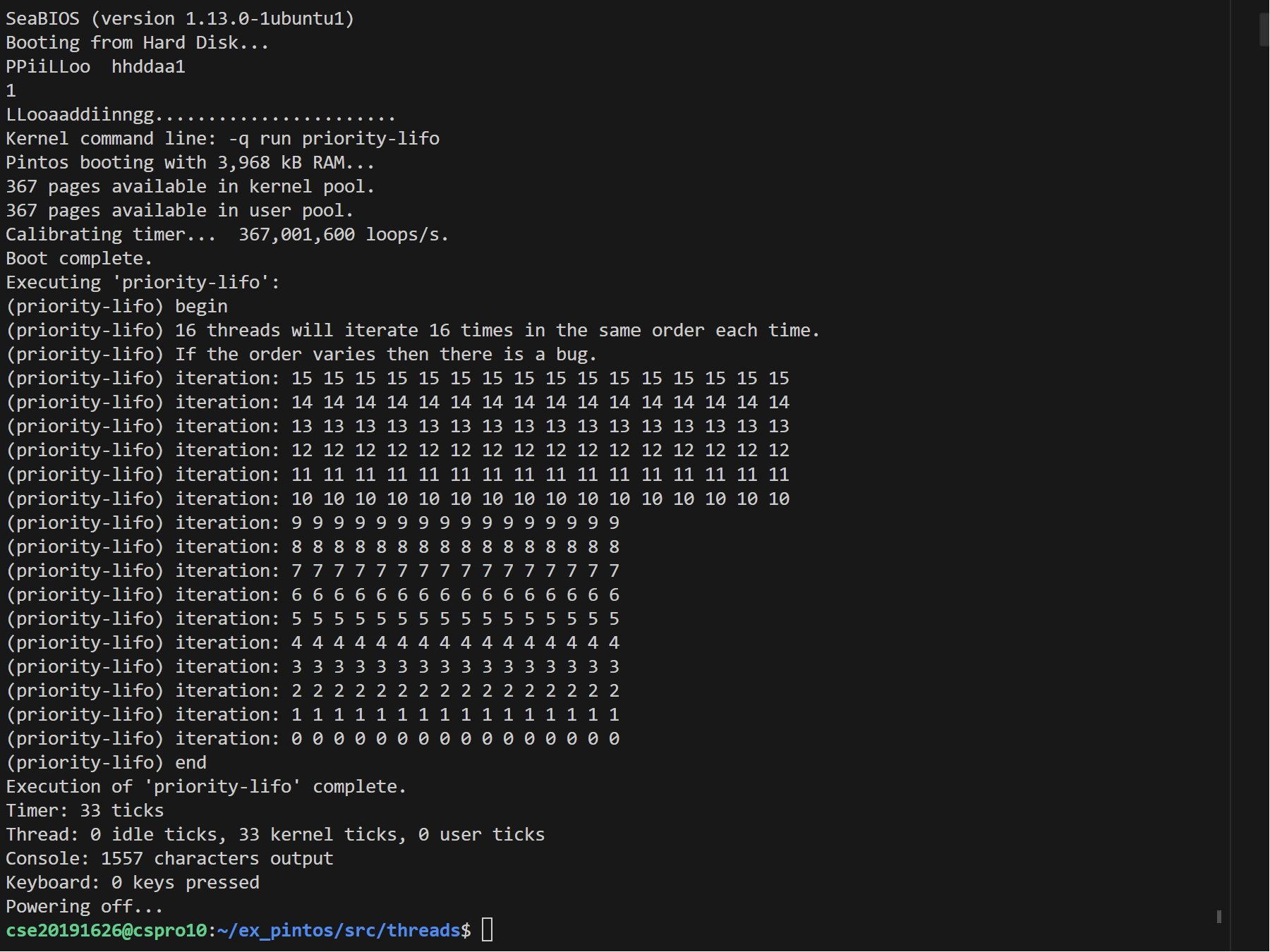
thread\_get\_load\_avg()

system load average 값에 100을 곱한 값을 return 한다.

thread\_get\_recent\_cpu()

현재 thread의 recent\_cpu값에 100을 곱한 값을 return 한다.

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



priority-lifo 코드를 분석해보자 이 코드에서는 0~15의 16개 쓰레드를 만든다.

이 때 각각의 우선 순위는 PRI\_DEFAULT + i + 1이다. 그 후에, main thread의 우선순위를 PRI\_DEFAULT로 설정하고 우선순위가 15인 thread부터 차례로 실행한다.

이를 통해 우선 순위가 높은 thread가 나중에 들어오더라도 먼저 수행을 하게 된다는 뜻이다.

위 결과 화면 처럼 출력이 된다. 0 부터 15까지 차례로 readylist에 들어가더라도, 가장 높은 15의 우선순위를 가진 thread가 먼저 실행되고 종료됨을 알 수 있다.

* make check 수행 결과를 캡처하여 첨부

