**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

학번 / 이름 : 20192134 /김은원

개발 기간 :11/01 ~ 11/7

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

Alarm clock, Priority Scheduling, Advanced 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

Thread는 running state와 ready state만 있는데 이는 비효율적이다. 따라서 효율적으로 바꾸기 위해서 block state를 만들어야 한다. 매시간마다 스레드를 깨어날지 말지 확인하는 것이 아니라 확인 뒤 스레드를 block한다. 새로운 큐를 만들어 언제 스레드를 깨울지 구현해야 하고 깨울 시간이 되면 ready 큐에 넣는다

* 1. Priority Scheduling

현재는 RR으로 Scheduling이 되어있다. 스레드의 우선순위와는 관계없이 돌아간다. 이번 프로젝트에서는 우선순위에 따라 threads를 실행해야 한다. 숫자가 높을수록 더 높은 우선순위를 가지고, 높은 우선순위의 스레드를 cpu에서 실행시켜야 한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

Nice, recent cpu, load\_avg를 계산하여 다양한 방면을 고려하는 Scheduler를 만들 수 있다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Blocked 상태의 list를 만든다. Timer\_sleep에서 현재 스레드의 wake up 시간은 현재시간과 매개변수 ticks를 더해 저장한다. 그 뒤 list에 스레드를 추가한 뒤 block한다. 그리고 list를 반복문으로 돌면서 wake up 시간이 된 스레드를 찾는다. 만약 스레드의 깨어나는 시간이 현재 시간과 같거나 작으면 깨워야 한다. list에서 해당 스레드를 제거하고 thread\_unblock을한다.

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

현재 thread를 중지하고 높은 priority인 thread를 실행한다. 현재 thread를 ready list에 넣고 ready list에 넣을 때도 우선순위에 맞게 정렬해야 한다.

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

Nice, recent cpu, load\_avg를 계산해야 한다. Nice는 -20~20 값을 가진다. 0은 영향을 끼치지 않고 양의 nice 값은 우선순위를 낮춘다. 만약 스레드가 처음 만들어지는 것이라면 0이지만 부모 스레드에서 나왔다면 상속받는다.

Recent cpu는 스레드의 cpu 시간을 측정한다. 더 많은 recent cpu 시간은 더 무거워진다. 처음 스레드가 만들어졌다면 0, 부모 스레드가 있다면 상속받는다. Time interrupt가 발생했다면 recent cpu는 running state에서 1 증가한다. 하지만 idle thread 일 때는 제외한다. Recent cpu와 load avg는 다음과 같이 계산한다.

* + *recent\_cpu* = (2 \* load\_avg) / (2 \* load\_avg + 1 ) \* *recent\_cpu* + nice
  + *load\_avg* : average of the number of thread in READY state

load avg는 시스템이 시작될 때 0으로 초기화한다. 매 초마다 갱신된다.

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

이때 advanced scheduler는 fixed point arithmetic을 사용하기 때문에 변환을 해야 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
* Alarm clock: 11/1 ~11/2
* Priority Scheduling 11/2~11/5
* Advanced Scheduler: 11/5~11/6
  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
* Alarm clock:
* 우선 block상태에서 깨우는 시간을 추가해야 한다. Thread.h에 thread 구조체에 wakeup 추가한다. 그리고 time.c에 block하고 있는 strcut list를 추가해야 한다. 또한 timer\_init 함수에서 list를 초기화해야 하고 list에 wake up time을 넣기 위해 timer\_sleep 함수를 수정해야한다. 그 뒤 timer\_interrupt 함수도 수정한다. Thread\_wake\_up 함수를 만들어 block인 상태에서 ready로 스레드를 깨운다.
* Priority Scheduling:

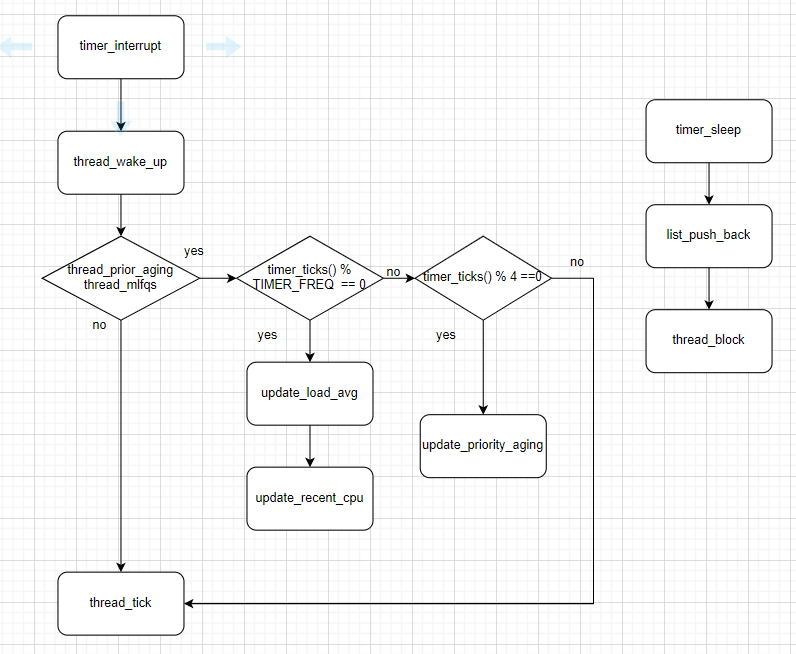
우선순위를 고려하여 thread를 실행해야 한다. Thread\_crete()함수는 스레드가 만들어질 때 호출된다. 이 함수의 parameter에는 priority가 있다. Thread\_create 함수를 수정하여 현재 스레드의 우선순위보다 parameter의 priority가 더 크면 thread\_yield를 호출하도록 한다. 즉 현재 실행하는 스레드를 교체해야 한다. 우선순위를 비교하기 위해서 thread\_priority\_compare함수를 만들고 sema에서는 우선순위가 없기 때문에 sema\_up 함수도 수정한다. Sema 구조체인 waiters에서 semaphore를 기다리는 스레드 중 가장 우선순위가 높은 스레드를 찾아 sema up을 한다. 또한 우선순위를 바꾸어야 하기 때문에 thread\_set\_priority 함수를 수정한다. Thread\_yield와 thread\_unblock 함수에서도 리스트를 넣을 때 우선순위에 맞게 정렬한다. 그리고 aging 작업을 위해 update\_priority\_aging 함수를 추가한다.

* Advanced Scheduler:

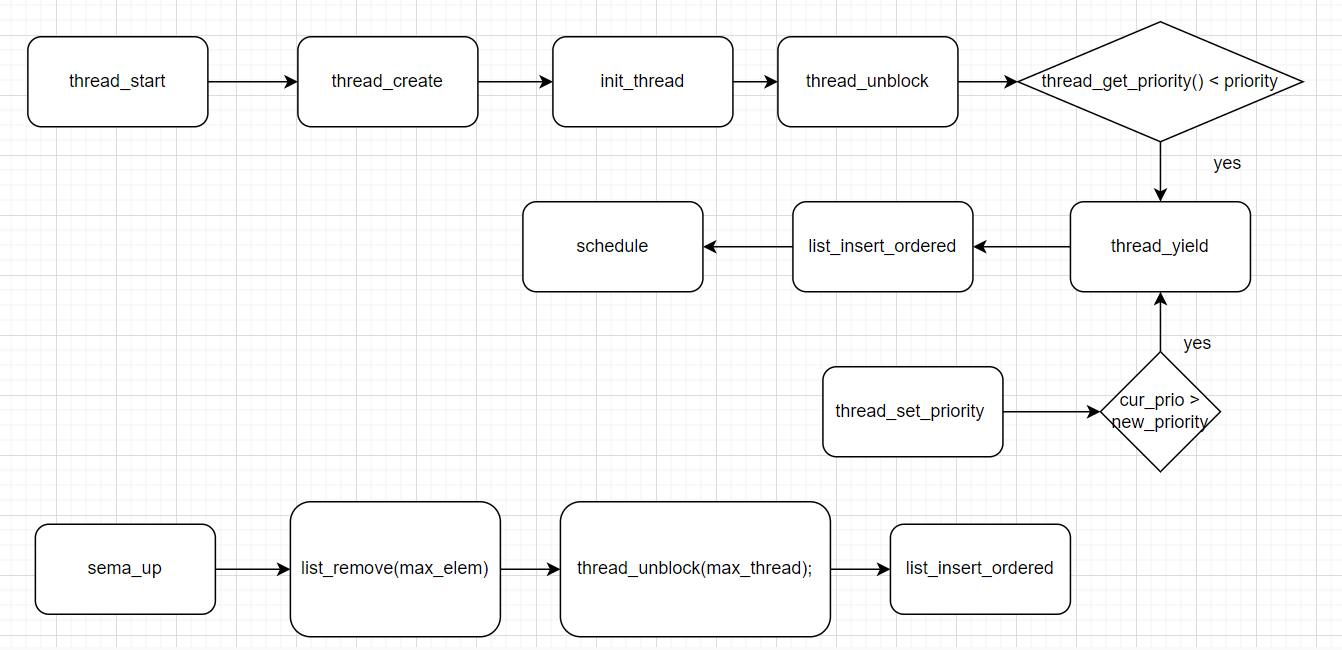
Thread.c에서 필요한 변수를 정의한다. 또한 적절한 초기화를 해야 한다. Thread\_get\_nice, thread\_get\_load\_avg, thread\_get\_recent\_cpu 함수를 구현한다. Update\_load\_avg, update\_recent\_cpu함수를 추가한다. Timer.c의 Timer\_interrupt 함수에서 값을 갱신한다.

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 서술할 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
* **Alarm clock:**

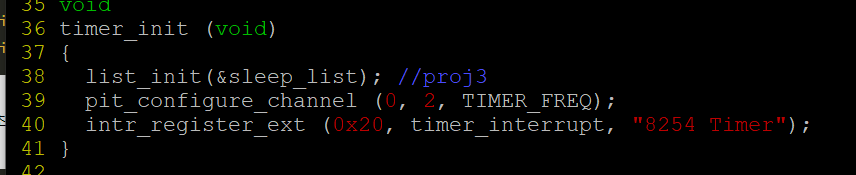
threads/thread.h thread 구조체에 깨우는 시간을 저장해야 하므로

 를 추가한다.

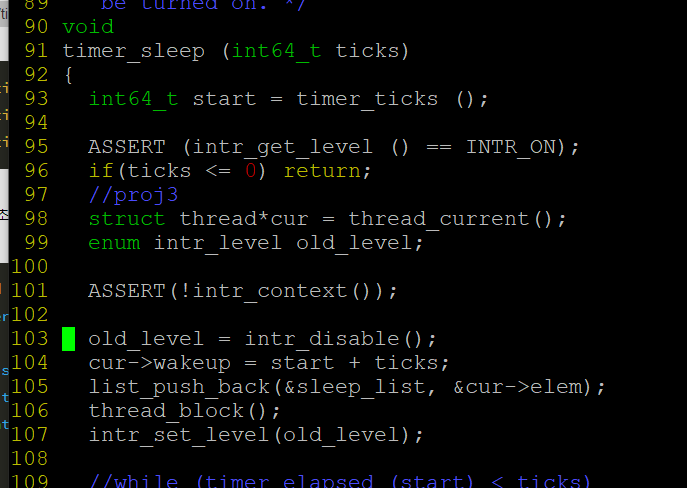
그리고 devices/timer.c에 sleep\_list를 추가했다. Sleep\_list는 block인 상태의 스레드를 저장해 놓는 list이다.



Timer.c의 Sleep\_list를 초기화한다.

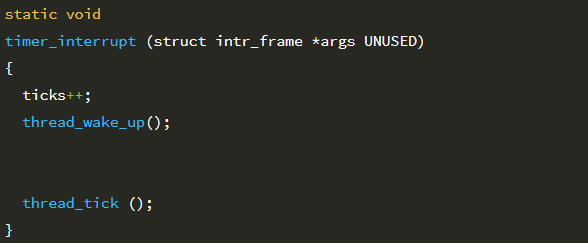
****

Timer.c의 Timer\_sleep 함수를 수정한다.

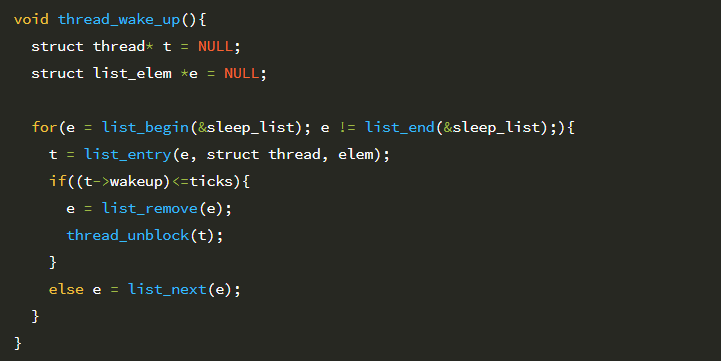


Start는 현재 시간이다. Timer\_sleep은 block에 있을 시간을 저장해야 하는 함수이기 때문에 wakeup 시간은 현재 시간과 얼마나 block해 있을 시간을 더한 것이다. 그 전에 ticks가 음수이거나 0이면 바로 return해 다음 작업을 수행하지 않도록 한다. wakeup시간을 저장했다면 바로 sleep\_list에 저장한다. 그 뒤 스레드를 block한다. Thread\_block함수는 thread.c에 있다. 현재 스레드의 상태를 block하고 schedule 함수를 호출한다.

Timer\_interrupt 함수에서 ticks를 하나씩 증가시킨다. 그 뒤 스레드를 깨우는 함수를 호출한다.

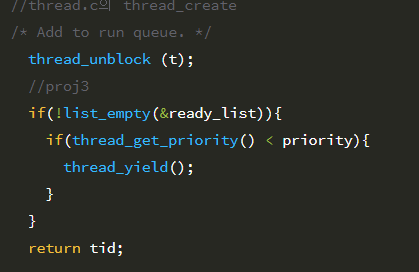


Thread\_wake\_up함수는 sleep list 중 깨울 시간이 된 스레드를 리스트에서 제거하고 block에서 해제한다.

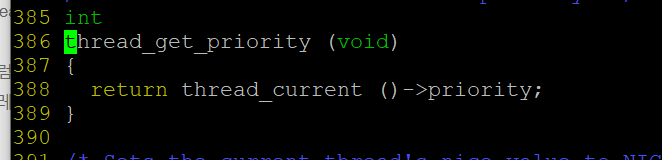


**Priority Scheduling**:

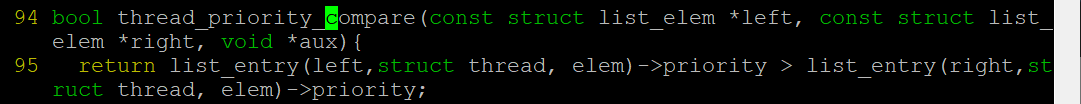
Thread\_create 함수에서 현재 만든 스레드의 우선순위보다 cpu에서 실행하고 있는 우선순위가 더 크면 현재 스레드를 다시 ready queue에 넣는다.



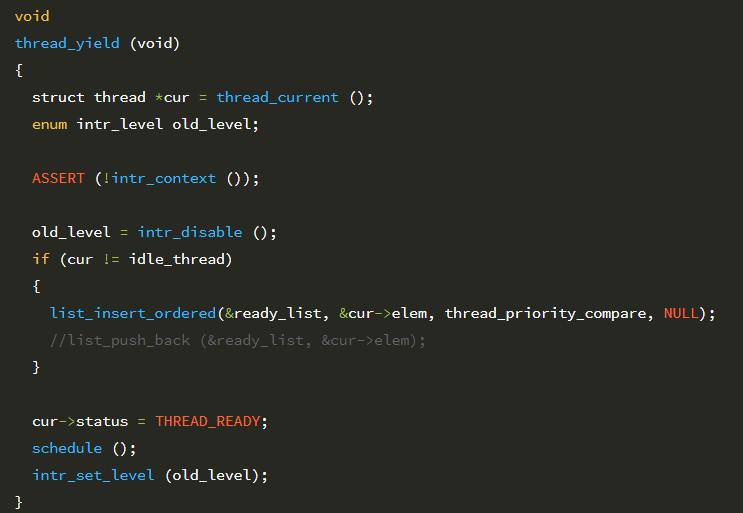
현재 스레드의 우선순위를 가져오는 함수인 thread\_get\_priority는 다음과 같다.



스레드의 우선순위를 비교한 뒤 ready list에 넣어야 한다. 즉 스레드의 우선순위를 비교하는 함수를 추가한다.



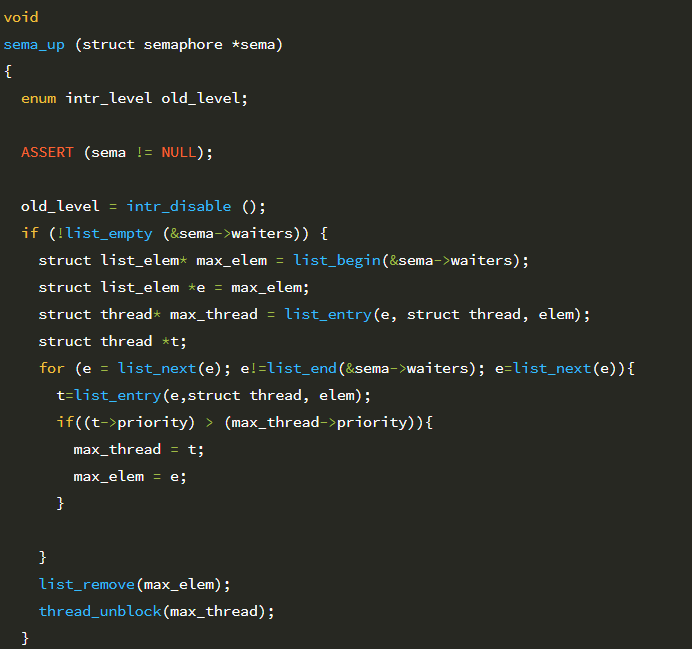
Thread\_yield()에서 현재 스레드를 다시 ready\_list에 정렬하며 넣는다.



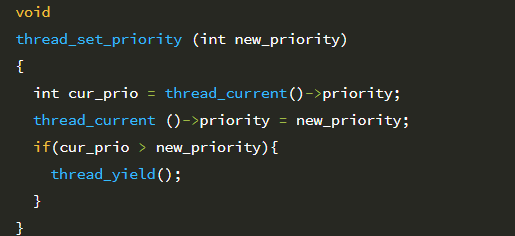
Block 상태에 해제되고 다시 ready\_list에 스레드를 넣기 위해 thread\_unblock도 수정한다.



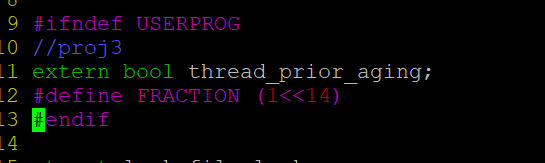
Sema up도 waiters 중에서 thread priority가 가장 높은 thread를 찾아 sema up을 해야한다.



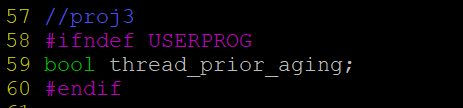
현재 스레드 우선순위가 새로운 우선순위보다 크면 다시 스레드 우선순위 조정을 위해 thread\_yield를 호출한다.



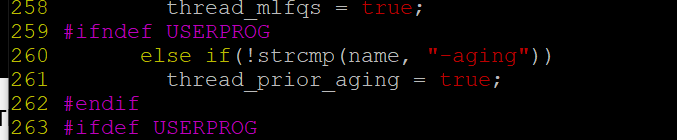
Priority aging: ppt에 따라 aging을 위해서는 bool 타입의 thread\_prior\_aging을 선언하고 threads/init.c도 수정한다.



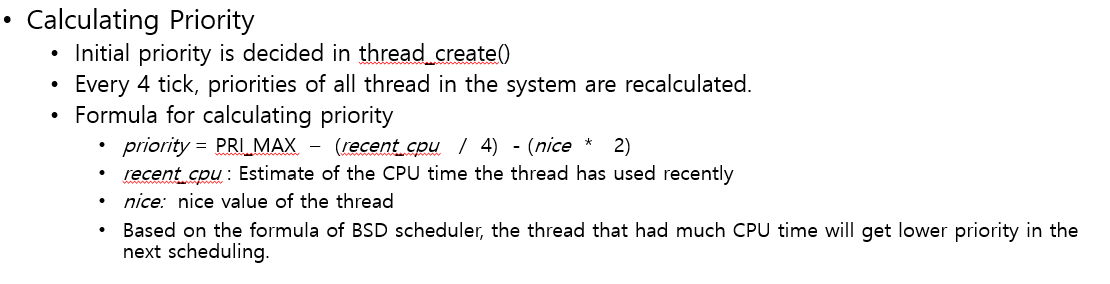
Thread.c 에도 선언한다.



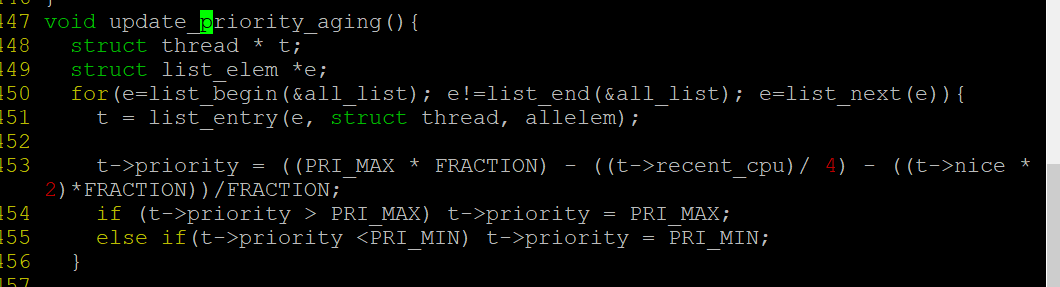
Init 파일을 수정하여 test 가 가능하도록 한다.



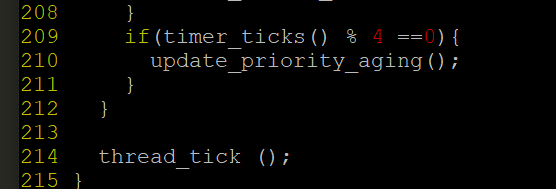
시간에 따라 우선순위를 높여야 해당 스레드가 cpu에서 실행될 수 있다. 따라서 update\_priority\_aging 함수를 만든다.



우선순위는 매 4 tick 마다 다시 계산된다. 그리고 우선순위는 priority = PRI\_MAX – (recent\_cpu / 4) - (nice \* 2)를 따른다. 따라서 다음과 같이 함수를 구현할 수 있다.

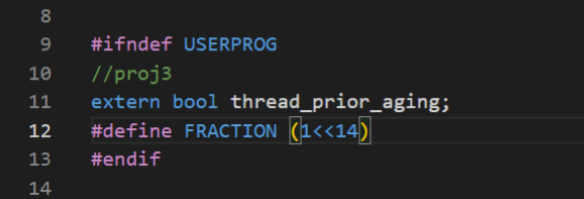


4ticks마다 우선순위를 갱신해야 하므로 timer interrupt함수도 수정한다.

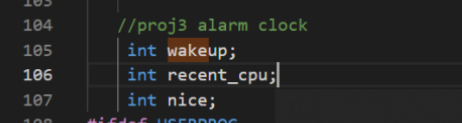


* **Advanced Scheduler:**

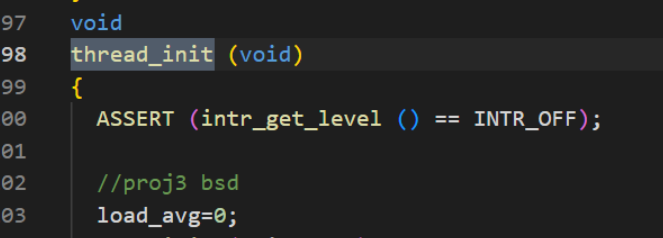
Fixed 로 구현하기 위해 FRACTION을 정의한다. Thread.h에 정의했다.



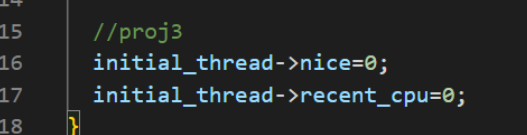
Thread.h에 thread 구조체에 recent\_cpu와 nice를 추가했다.



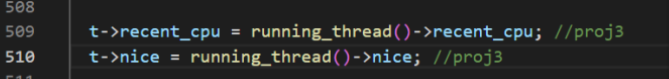
Load\_avg를 thread.c에 선언했다. Thread\_init에서 초기화하였다.



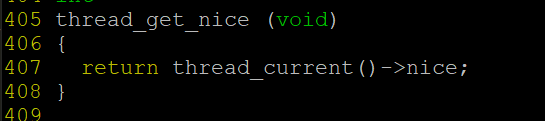
Thread\_init에서 nice, recent\_cpu도 초기화했다.



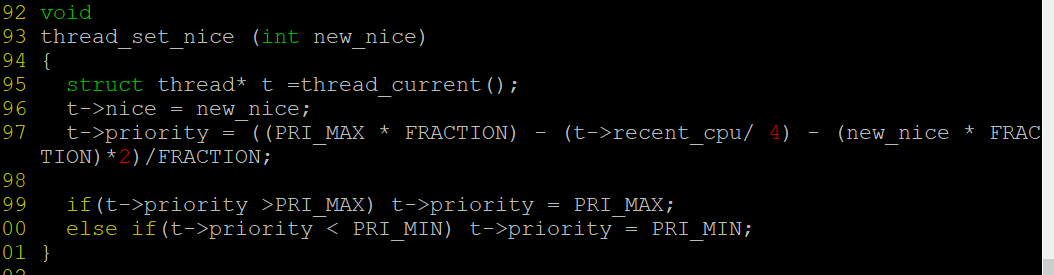
그리고 부모 스레드에서 상속받도록 구현했다.



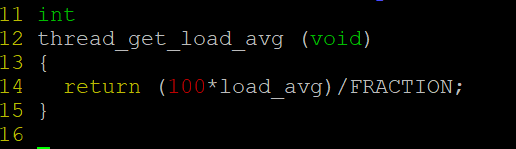
Thread\_get\_nice 는 현재 스레드의 nice 값을 반환한다.



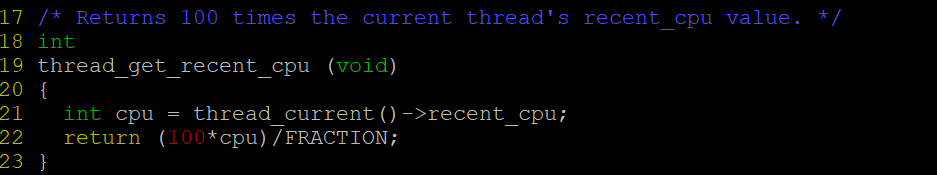
Thread\_set\_nice에서 현재 스레드의 nice값은 new nice값으로 한다. New nice값을 기반으로 스레드의 우선순위를 다시 계산해야 한다. 계산 후 우선순위가 max보다 크다면 max값으로, 우선순위가 min보다 작다면 min 값으로 조정한다.



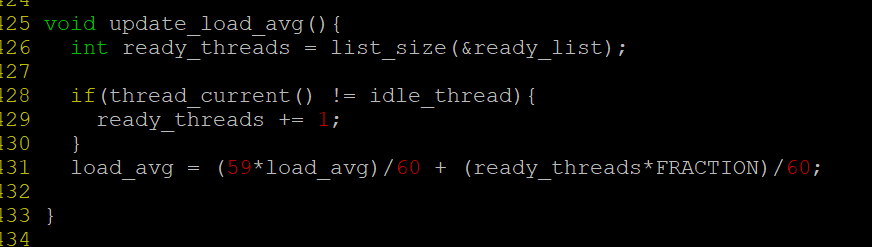
thread\_get\_load\_avg는 현재 시스템의 load avg값에 100을 곱한 값을 반환한다.



Thread\_get\_recent\_cpu는 현재 스레드의 recent cpu의 100배값을 반환한다.



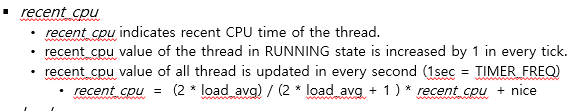
Update\_load\_avg는 시간에 따라 갱신되는 load\_avg값을 계산하는 함수이다.

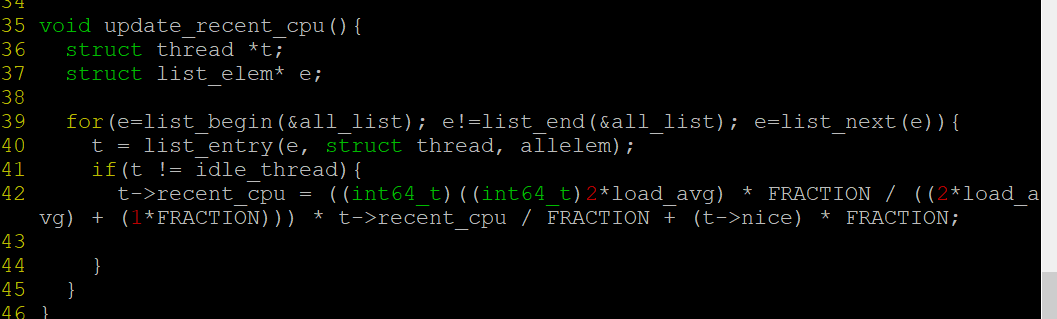


Ready thread는 Idle thread가 아니면 1을 더한다. load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_threads 로 load\_avg 함수를 계산한다.

Recent cpu 또한 마찬가지로 갱신하는 함수가 필요하다.

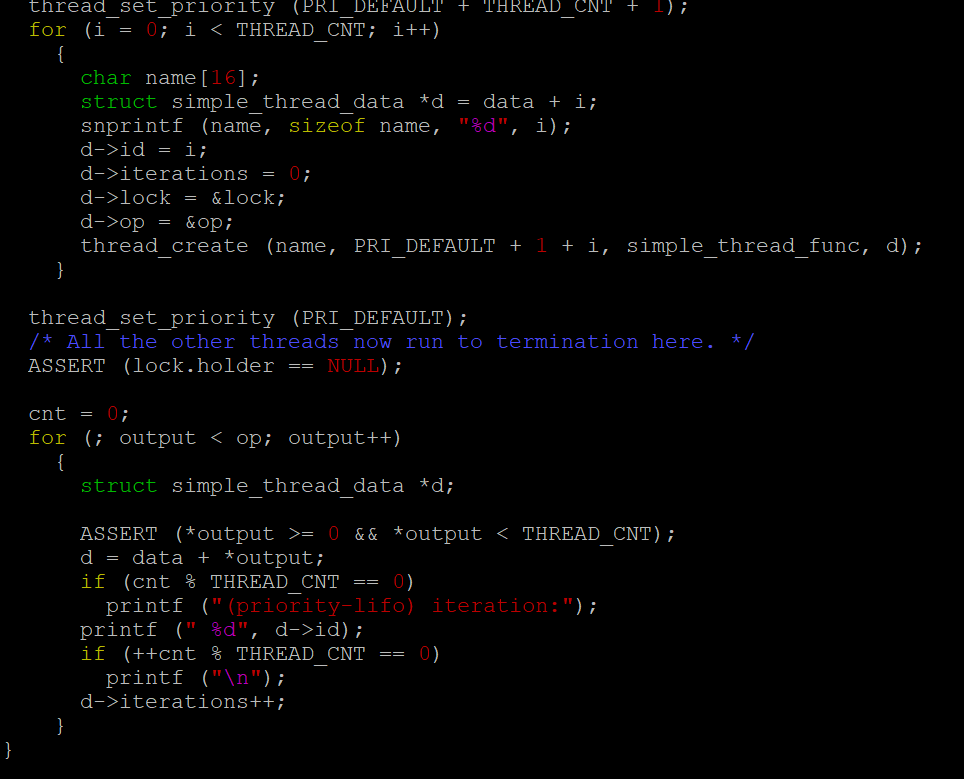
스레드 마다 recent cpu가 있다. 따라서 반복문을 통해 스레드의 recent cpu를 계산한다.

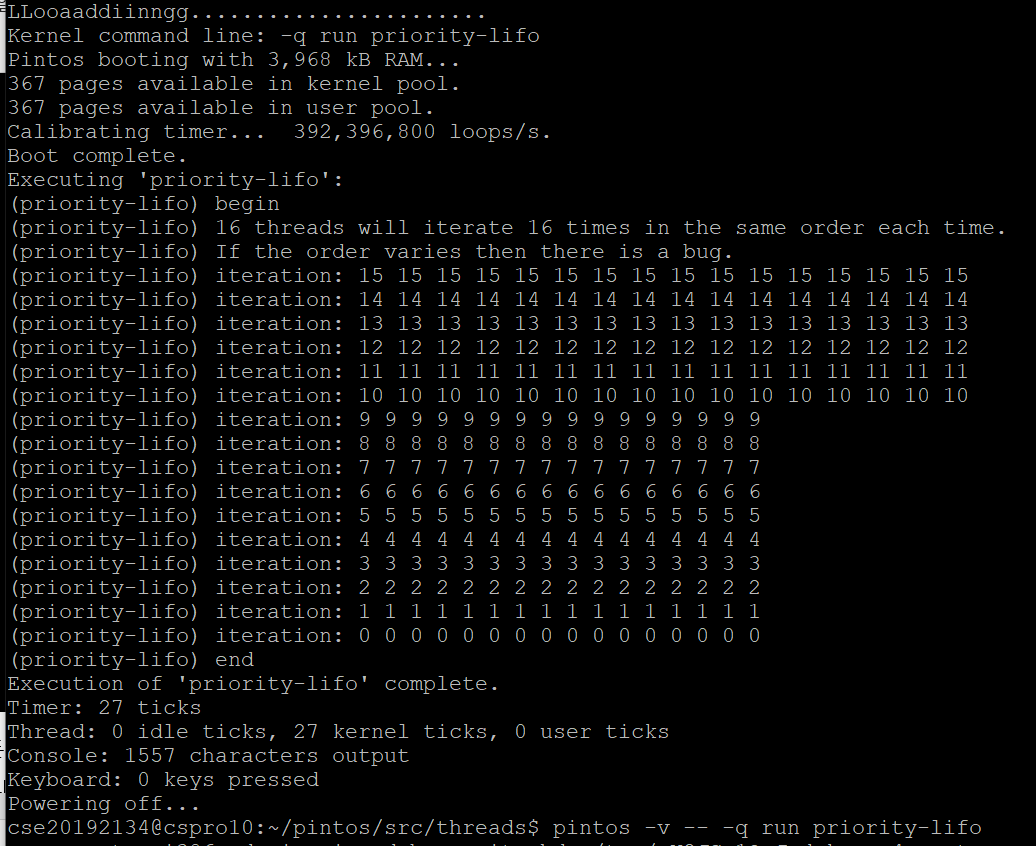




* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

priority lifo 는 동일한 우선순위 스레드를 여러 개 만들고 출력되는 우선순위가 만약 다양하게 나온다면 실패, 동일한 우선순위라면 성공이다.



* 

우선순위가 일정하게 15에서 0으로 같은 숫자로 줄어들었다는 것을 알 수 있다.

* make check 수행 결과를 캡처하여 첨부
* 