**Pintos Project 3: Threads**

담당 교수 : 문의현 교수님

학번 / 이름 : 20181255 김기철

개발 기간 : 2022.12.1 ~ 2022.12.5

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

운영체제에서 thread들을 효율적으로 관리하기 위한 기능들을 구현한다. 이번 Proj 3에서 구현할 기능은 크게 Alarm Clock, Priority Scheduling 그리고 Advanced Scheduler이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock

운영체제에서 thread는 크게 Ready, Running, Blocked, Dying 4가지로 구분할 수 있다. 기존의 timer\_sleep 함수에서는 사용하지 않는 thread를 sleep 상태로 할당하기 위해, 시간마다 while문을 통해 반복적으로 할당된 시간이 지났는지를 확인한다. 이 과정에서 thread는 Ready 상태와 Running 상태를 반복적으로 오가는 비효율적인 작업이 진행된다. 이러한 비효율성을 해결하기 위해, 할당된 시간이 지날 때까지 thread를 Blocked 상태로 바꾸어 thread를 효율적으로 관리하도록 함수를 새로 구현해야 한다.

* 1. Priority Scheduling

기존 운영체제에서는 thread들을 round-robin 방식으로 스케줄링이 되어있어서 thread의 우선순위는 고려하지 않고, 무조건 ready queue의 마지막 부분에 insert된다. 따라서, 이번 프로젝트에서는 우선순위 스케줄링 방식으로 높은 우선순위를 가지는 thread들부터 먼저 ready queue의 앞 부분에 insert해주어야 한다. 하지만, 단순히 우선순위만을 고려하게 된다면 우선순위가 낮은 thread는 매우 오랫동안 ready queue에 남아있어야 하는 기아 현상이 발생할 수 있다는 단점이 있다. 따라서, 우선순위가 낮아도 ready queue에서 기다리는 시간이 길수록 우선순위를 증가시켜 ready queue의 앞 부분에 위치할 수 있도록 해주는 aging 기법 또한 사용해야 한다. 즉, 우선순위 스케줄링과 aging 기법을 적절히 구현하여 thread들을 효율적으로 관리해야 한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

위에서 구현한 우선순위 스케줄링 이외 추가적인 스케줄링 방식을 구현해야 하는데, 이번 프로젝트에서는 mlfq를 이용하는 BSD 스케줄링 방식을 구현한다. 여러 개의 ready queue 중 우선순위가 높은 thread를 선택하여 CPU에 할당되도록 구현해야 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

사용하지 않는 thread들을 할당된 시간이 지날 때까지 blocked 상태로 관리해야 하는데, 이를 구현하기 위해 새로운 자료구조와 멤버변수가 필요하다. 먼저, 할당된 시간을 나타내는 wakeup\_time을 thread 구조체 내에 선언해야 하고, blocked 상태인 thread들을 담기 위한 새로운 blocked\_queue 구조체를 선언해야 한다. 따라서, blocked queue에 저장되어 있는 thread들의 wakeup\_time을 검사하여 할당된 시간이 지났으면 blocked queue에서 ready queue로 옮겨주면 효율적으로 blocked 상태의 thread를 깨울 수 있다.

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

현재 running 상태의 thread보다 높은 우선순위의 thread가 ready queue에 들어온다면 현재 running 상태의 thread의 실행을 중단시킨 후에 높은 우선순위의 thread를 thread\_yeild 함수를 이용하여 높은 우선순위 thread를 running 상태로 바꾸어 실행시킨다. 또한, 중단된 기존의 thread는 다시 ready queue의 적절한 위치에 insert해준다.

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

BSD 스케줄링을 위해 다음과 같은 3개의 변수가 필요하다. 3개의 변수 중 nice와 recent\_cpu 변수는 thread 구조체 내에 선언해주고, load\_avg 변수는 전역 변수로 선언해 준다.

(1) nice

nice 변수는 -20에서 20사이의 값을 가지는 우선순위 값에 영향을 미치는 변수이다. 아래에서 좀 더 자세히 설명을 하겠지만, 새로운 우선순위 값은 다음과 같은 계산식을 통해 구한다.

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

여기서 nice 변수에 2를 곱한값을 빼주는 것을 볼 수 있는데, nice 변수가 양수라면 우선순위 값이 작아지게 되고 nice 변수가 0이라면 우선순위 값에 영향을 미치지 않는다는 뜻이다. nice 변수는 thread가 처음 생성될 때, 0으로 초기화된다.

(2) recent\_cpu

recent\_cpu 변수는 thread가 CPU에서 소모한 시간을 나타내느 값으로 thread가 최근일수록 더 많은 시간을 소모한다. recent\_cpu 변수도 nice 변수와 마찬가지로 처음 thread가 생성될 때, 0으로 초기화되고 thread가 running 상태일 때, 값이 1 증가하게 된다. recent\_cpu는 매초 다음과 같은 계산식을 통해 구해진다.

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

(3) load\_avg

위에서 recent\_cpu 값을 구할 때, load\_avg 변수가 필요하다. load\_avg 변수는 thread의 평균 개수를 나타내는 변수로, 처음에 0으로 초기화되는 전역변수이다. load\_avg 변수도 recent\_cpu와 같이 매초 다음과 같은 계산식을 통해 구해지는데, 다음 계산식에서 ready\_thread는 idle thread를 제외하고 ready 또는 running 상태의 thread의 개수를 나타내는 변수이다.

**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

12.1~12.2 온라인 강의 학습 및 강의자료 이해

12.2~12.4 Alarm clock 및 Priority Scheduling 기능 구현

12.5~12.5 Advanced Scheduler 기능 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

1) Alarm Clock

현재 thread에 할당된 시간을 나타내는 wakeup\_time을 thread 구조체 내에 새로 선언한 후, 0으로 초기화해준다. 또한, blocked 상태의 thread들을 관리하기 위한 새로운 자료구조인 blocked\_queue를 새로 선언한다. 선언을 한 후, timer\_sleep 함수에서 기존의 thread\_yield 함수를 이용한 방식이 아닌 새롭게 구현을 해주어야 하는데, 현재 thread의 wakeup\_time을 할당해주고 list\_push\_back 함수를 이용하여 blocked\_queue에 thread를 넣어준 후, thread\_block 함수를 이용하여 현재 thread를 blocked 상태로 바꿔준다.

2) Priority Scheduling

강의자료에 나와있는 대로 aging 기법을 위한 thread\_prior\_aging 변수를 선언하고 aging 기법이 이용될 수 있도록, init.c의 parse\_option 함수에서 thread\_prior\_aging을 true로 바꿔주는 부분을 추가한다.

thread\_create 함수 내에 새로운 우선순위 값이 현재 thread의 우선순위 값보다 크다면 thread\_yield 함수를 이용하여 running 상태로 바꿔준다.

우선순위 비교를 위한 함수 thread\_cmp\_priority를 구현한다. 정의하기 전에 헤더파일에 먼저 선언을 한다. 인자로 받은 elem 변수 a와 b를 먼저 list\_entry 함수를 이용하여 thread에 접근하고, 2개의 thread의 우선순위를 비교하여 높은 우선순위 값을 가진 thread일수록 ready\_queue의 앞 부분에 정렬되도록 한다. 이 우선순위 비교 함수는 아래에 설명할 thread\_unblock 함수와 thread\_yield 함수에 이용된다.

3) Advanced Scheduler

thread 구조체 내에 nice 변수와 recent\_cpu 변수를 선언한다. 그리고 핀토스는 floating-point arithmetic을 지원하지 않으므로 32-bit 중 하위 14-bit 소수부분인 것을 이용하여 FLOATING\_POINT 값을 (1<<14)로 정의하고, 이를 이용하여 nice 변수와 recent\_cpu 변수 그리고 우선순위 값 연산을 float자료형처럼 연산할 수 있게 된다. 또한, thread의 평균 개수를 나타내는 load\_avg를 전역 변수로 선언한다.

thread\_get\_priority, thread\_set\_priority, thread\_new\_priority, thread\_get\_nice, thread\_set\_nice, thread\_get\_recent\_cpu, thread\_get\_load\_avg, thread\_set\_rcpu\_and\_ lavg 함수를 구현하여 현재 thread의 우선순위, nice, recent\_cpu, load\_avg 값을 위에서 설명한 계산식을 이용하여 새로 연산하거나 현재 thread의 값들을 적절히 반환해주도록 함수를 구현한다.

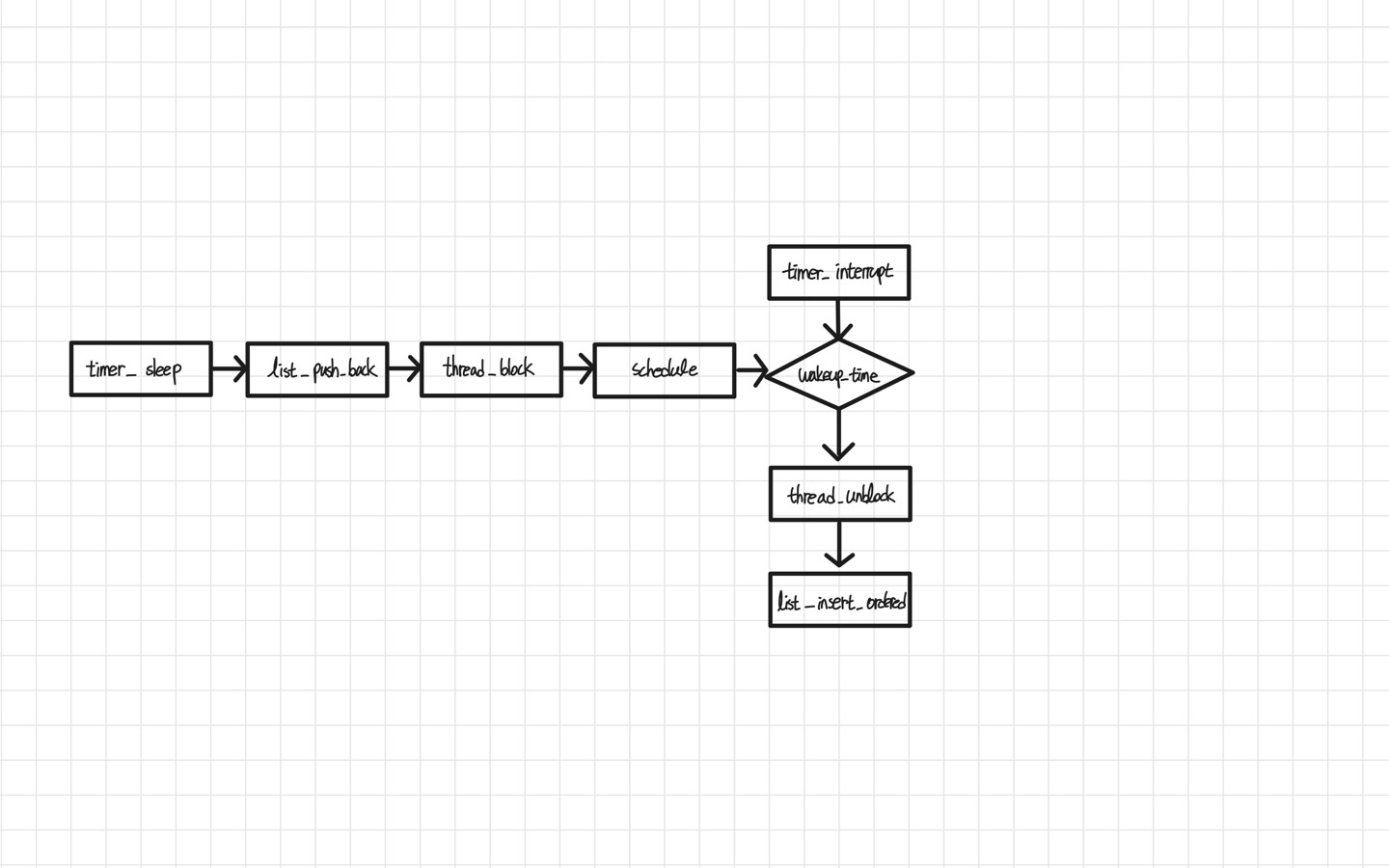
매 초마다 현재 thread의 우선순위 값을 초기화한 후, ticks에 따라 thread\_new\_ priority 또는 thread\_set\_rcpu\_and\_lavg 함수를 호출한다.

sema의 waiters에 존재하는 thread들 중 우선순위가 가장 큰 thread를 sema\_up를 해주도록 한다.

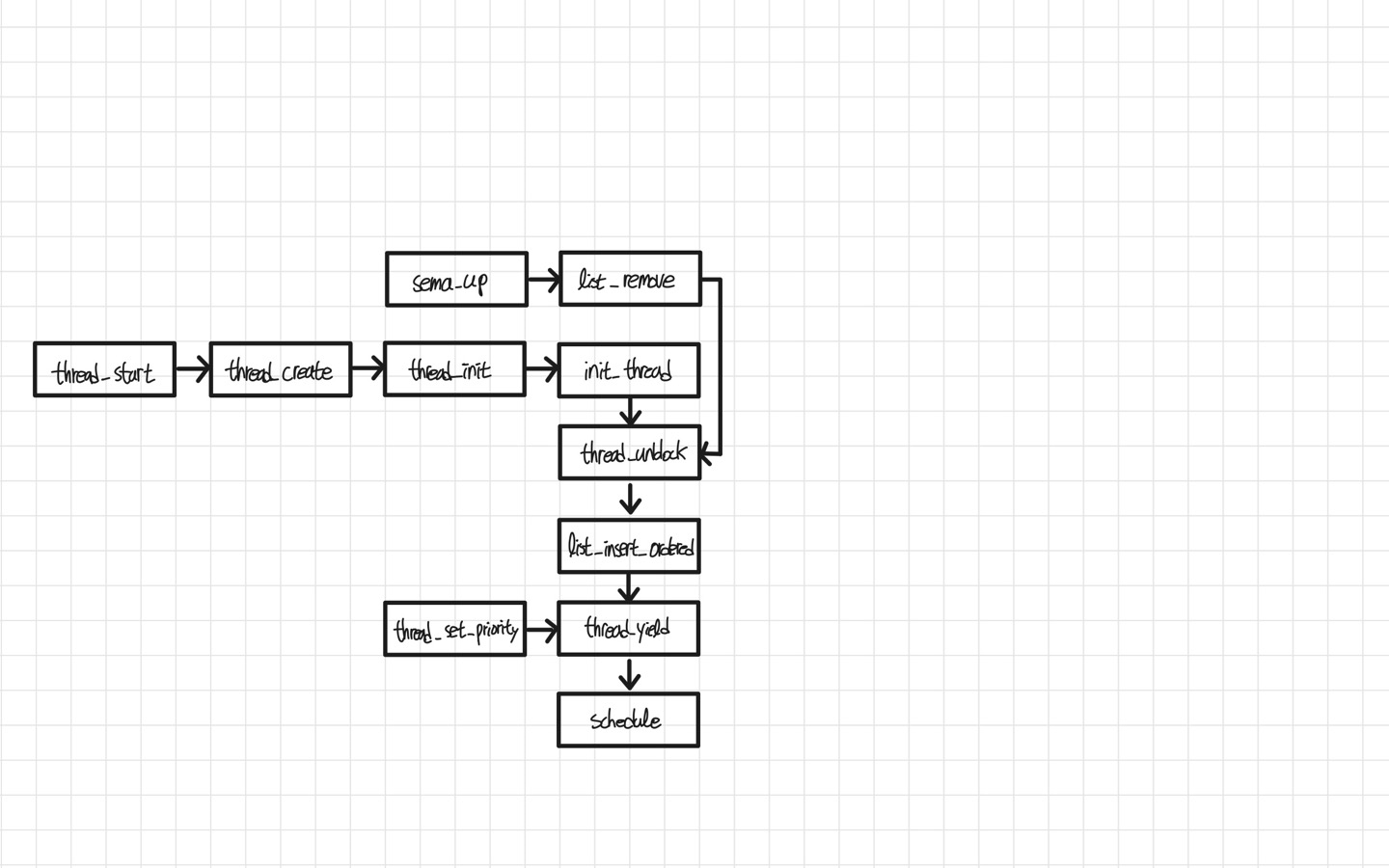
1. **연구 결과**
   1. **Flow Chart**

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

1) Alarm Clock



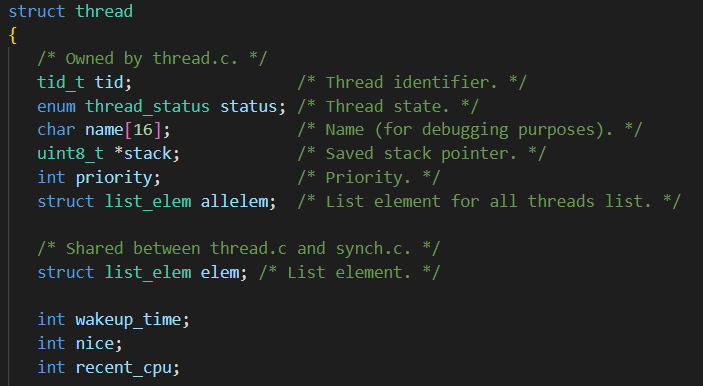
2) Priority Scheduling



* 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

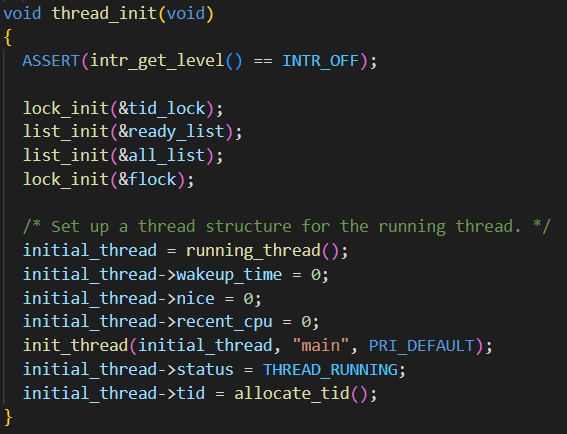
1) Alarm Clock

(1) threads/thread.h



thread 구조체 내에 할당된 시간을 나타내는 wakeup\_time 변수를 선언한다.

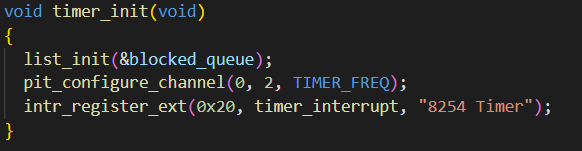
(2) threads/thread.c



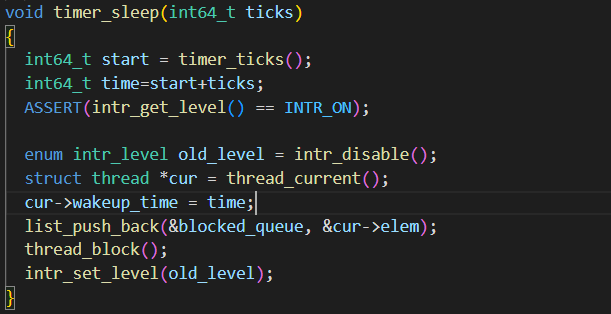
thread\_init 함수에서 위에서 선언한 wakeup\_time 변수를 0으로 초기화한다.

(3) device/timer.c





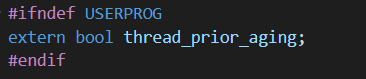
blocked 상태의 thread들을 담기 위한 새로운 blocked\_queue 구조체를 선언한 후, timer\_init 함수에서 list\_init 함수를 이용하여 blocked\_queue를 초기화한다.



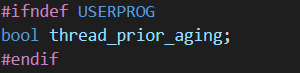
timer\_sleep 함수에서 현재 thread의 wakeup\_time을 할당해주고, list\_push\_back 함수를 이용하여 blocked\_queue에 thread를 넣어준 후, thread\_block 함수를 이용하여 현재 thread를 blocked 상태로 바꿔준다.

2) Priority Scheduling

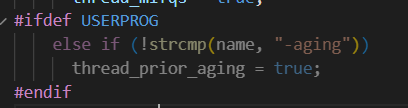
(1) threads/thread.h



(2) threads/thread.c

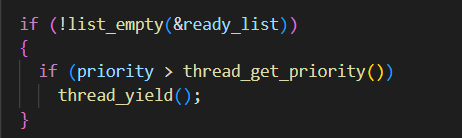


(3) threads/init.c



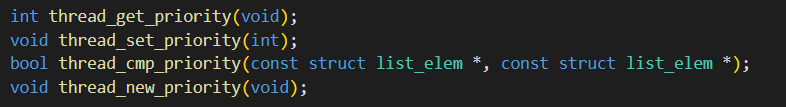
강의자료에 나와있는 대로 aging 기법을 위한 thread\_prior\_aging 변수를 선언하고 aging 기법이 이용될 수 있도록, init.c의 parse\_option 함수에서 thread\_prior\_aging을 true로 바꿔주는 부분을 추가한다.

(4) threads/thread.c

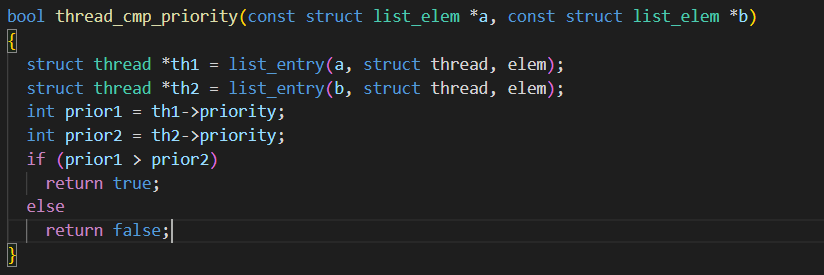


thread\_create 함수 내에 새로운 우선순위 값이 현재 thread의 우선순위 값보다 크다면 thread\_yield 함수를 이용하여 running 상태로 바꿔준다.

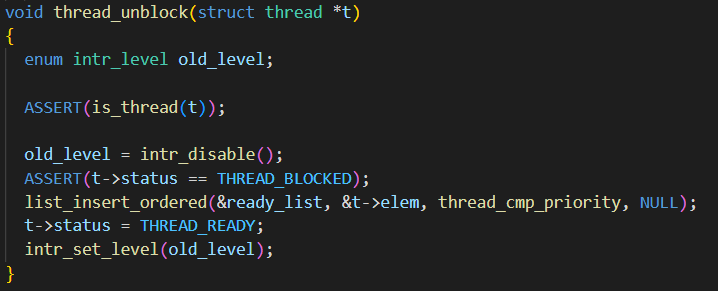
(5) threads/thread.h

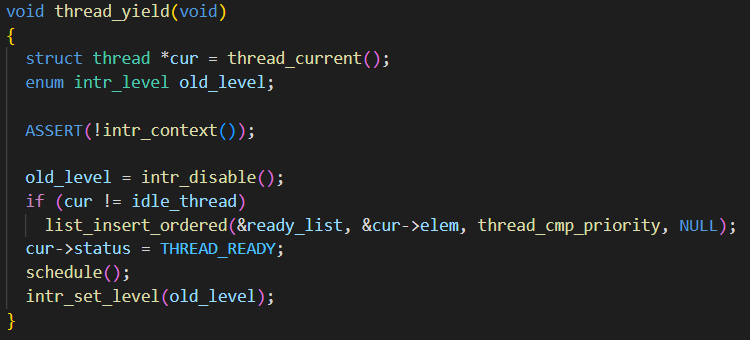


(6) threads/thread.c



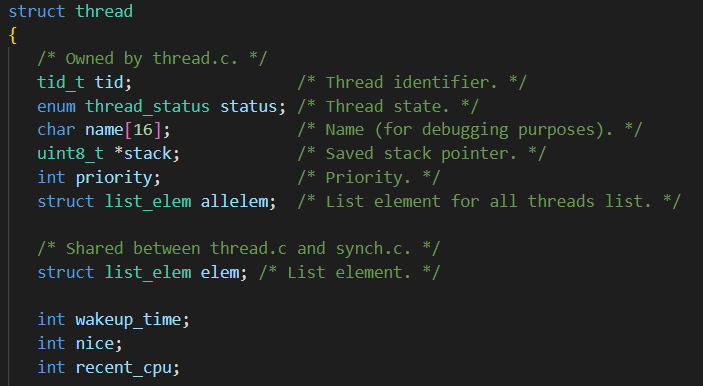
우선순위 비교를 위한 함수 thread\_cmp\_priority를 구현한다. 정의하기 전에 헤더파일에 먼저 선언을 한다. 인자로 받은 elem 변수 a와 b를 먼저 list\_entry 함수를 이용하여 thread에 접근하고, 2개의 thread의 우선순위를 비교하여 높은 우선순위 값을 가진 thread일수록 ready\_queue의 앞 부분에 정렬되도록 한다. 이 우선순위 비교 함수는 아래에 설명할 thread\_unblock 함수와 thread\_yield 함수에 이용된다.

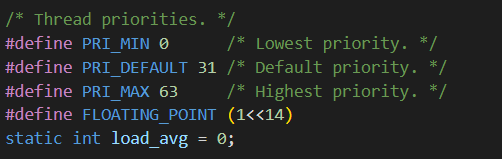


list\_insert\_ordered 함수와 위에서 구현한 thread\_cmp\_priority 함수를 이용하여 우선순위가 높은 thread를 ready queue의 앞부분에 위치하도록 한다.

3) Advanced Scheduler (추가구현을 한 경우)

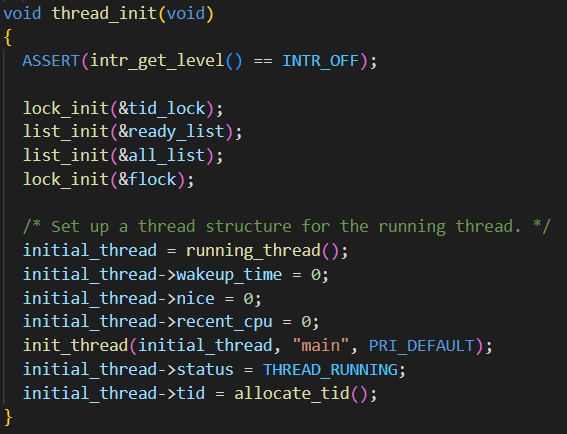
(1) threads/thread.h





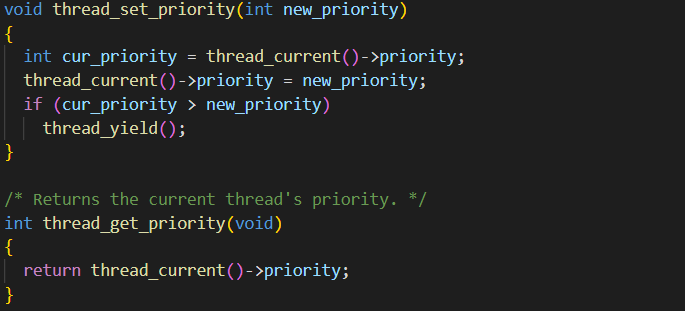
먼저 thread 구조체 내에 nice 변수와 recent\_cpu 변수를 선언한다. 그리고 핀토스는 floating-point arithmetic을 지원하지 않으므로 32-bit 중 하위 14-bit 소수부분인 것을 이용하여 FLOATING\_POINT 값을 (1<<14)로 정의하고, 이를 이용하여 nice 변수와 recent\_cpu 변수 그리고 우선순위 값 연산을 float자료형처럼 연산할 수 있게 된다. 또한, thread의 평균 개수를 나타내는 load\_avg를 전역 변수로 선언한다.

(2) threads/thread.c

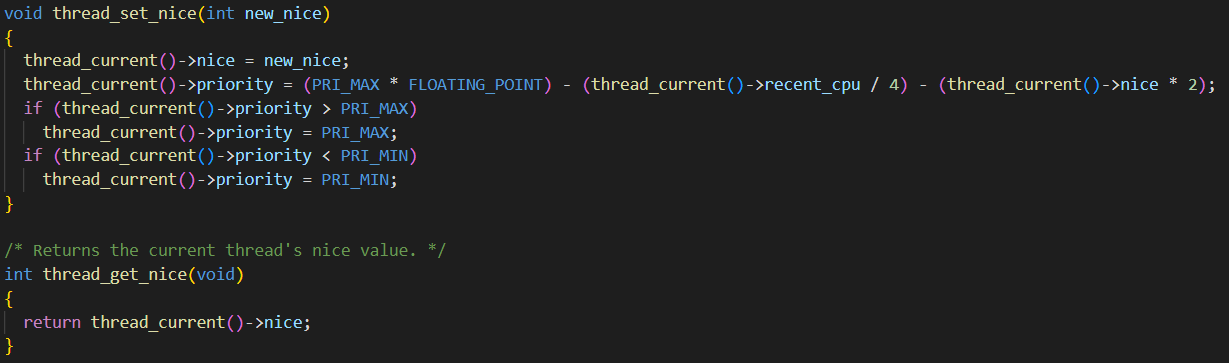




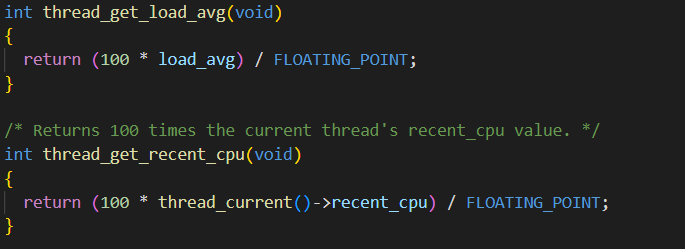
위에서 선언한 nice와 recent\_cpu 변수를 thread\_init 함수에서 0으로 초기화한다. 하지만, 부모 thread가 존재하는 thread라면 0으로 초기화하는 것이 아니라 부모의 nice와 recent\_cpu 값을 물려받아야 하므로 init\_thread 함수 내에서 nice 변수와 recent\_cpu 값을 부모의 값 그대로 할당해준다.



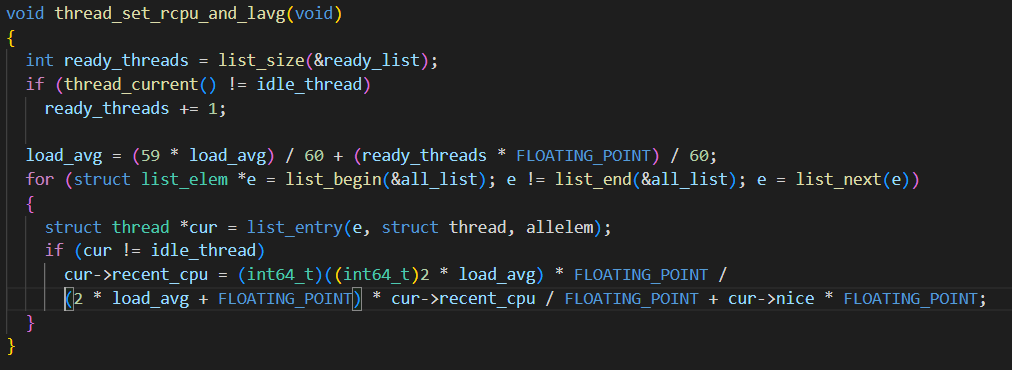
현재 thread의 우선순위 값을 새로운 값으로 바꿀 때 이용하는 함수인 thread\_set\_priority와 현재 thread의 우선순위 값을 반환하는 thread\_get\_priority 함수를 구현한다. thread\_set\_priority 함수에서 인자로 받은 새로운 우선순위 값을 바꾸어 주는데, 만약 기존 우선순위 값보다 새로운 우선순위 값이 작다면 thread\_yield 함수를 호출한다.



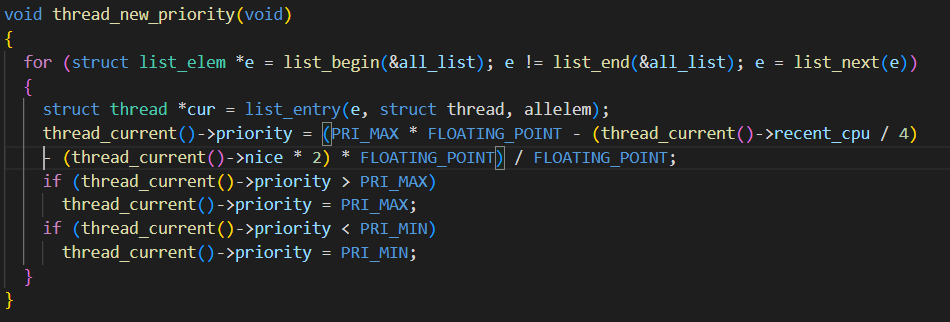
현재 thread의 nice 값을 새로운 값으로 바꿀 때 이용하는 함수인 thread\_set\_nice와 현재 thread의 nice 값을 반환하는 thread\_get\_nice 함수를 구현한다. thread\_set\_nice 함수에서 인자로 받은 새로운 nice 값을 현재 thread의 nice 값으로 바꾸어 준 후, nice 값과 위에서 설명한 우선순위 값을 구하는 계산식을 통해 현재 thread의 우선순위 값을 새로 계산한다. 만약 새로 계산한 우선순위 값이 범위를 벗어나지 않도록 PRI\_MAX와 PRI\_MIN과 비교해주는 부분을 추가한다.



현재 load\_avg값과 현재 thread의 recent\_cpu 값에 100을 곱한 결과를 반환하는 함수들이다.

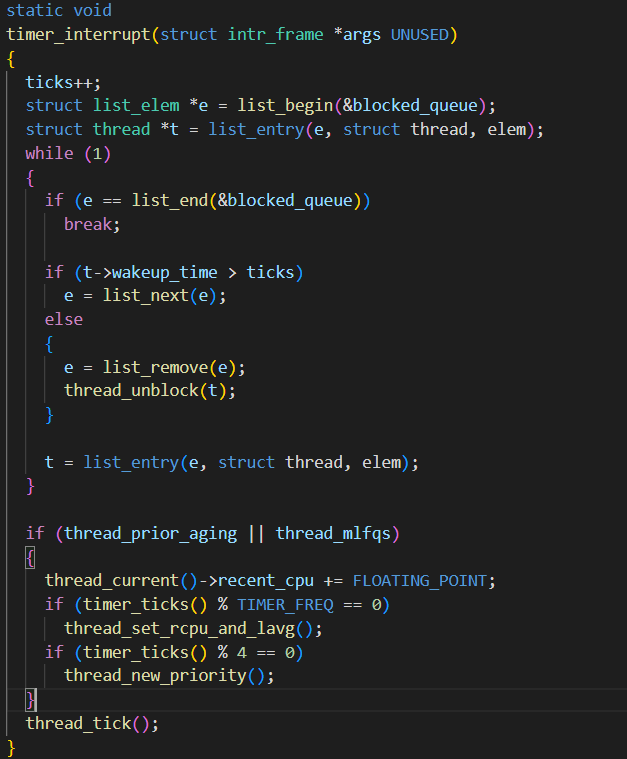


위에서 설명한 계산식을 이용하여 현재 thread의 recent\_cpu 값과 load\_avg 값을 새로 구하는 함수이다.



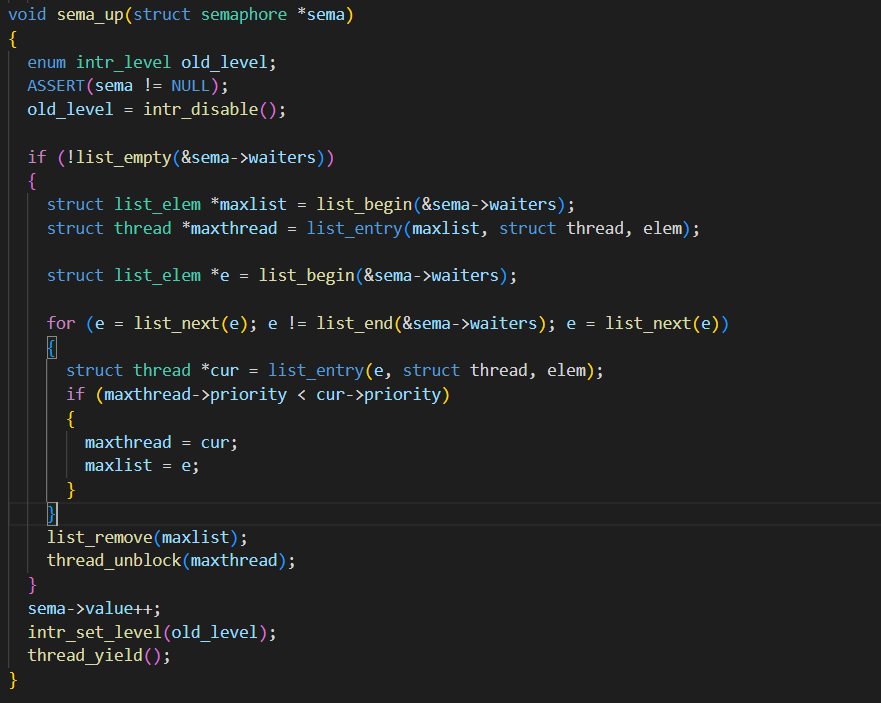
위에서 설명한 계산식을 이용하여 현재 thread의 우선순위 값을 새로 구하는 함수이다.

(3) device/timer.c



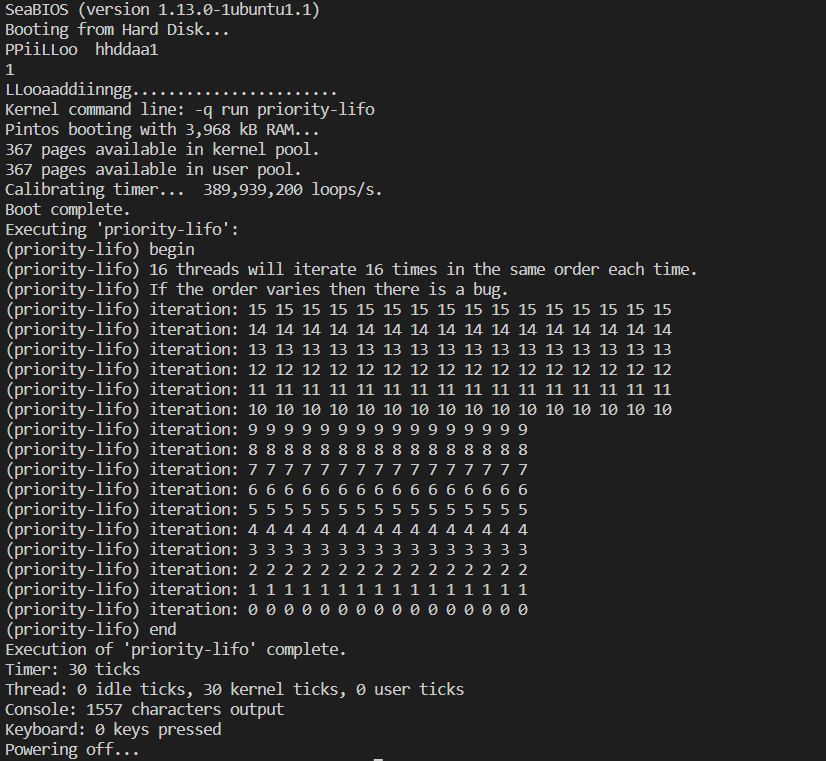
매 초마다 현재 thread의 우선순위 값을 초기화하는 함수인데, ticks에 따라 thread\_new\_priority 또는 thread\_set\_rcpu\_and\_lavg 함수를 호출한다.

(4) threads/synch.c

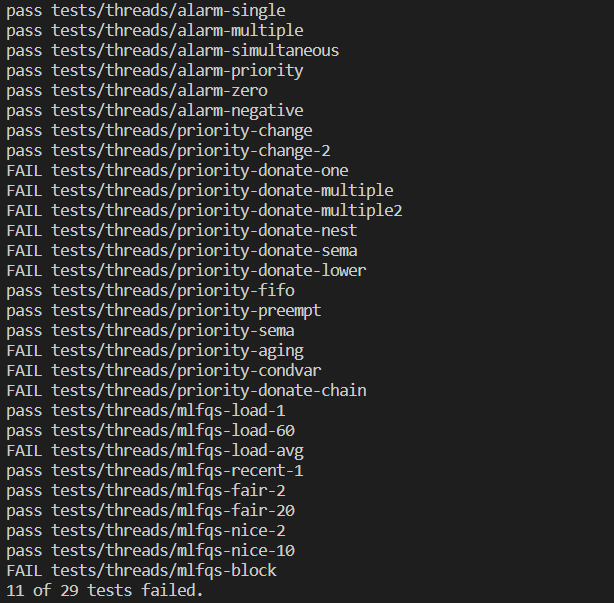


sema의 waiters에 존재하는 thread들 중 우선순위가 가장 큰 thread를 sema\_up를 해주도록 한다.

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



* make check 수행 결과를 캡처하여 첨부



19개의 case를 통과하였지만(mlfqs-block 출력밀림현상 포함), 2개의 case인 priority-aging과 mlfqs-load-avg는 통과하지 못했다.