**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

조 / 조원 : 20181679 장서우

개발 기간 : 2020.11.18 ~ 2020.12.06

1. **개발 목표**

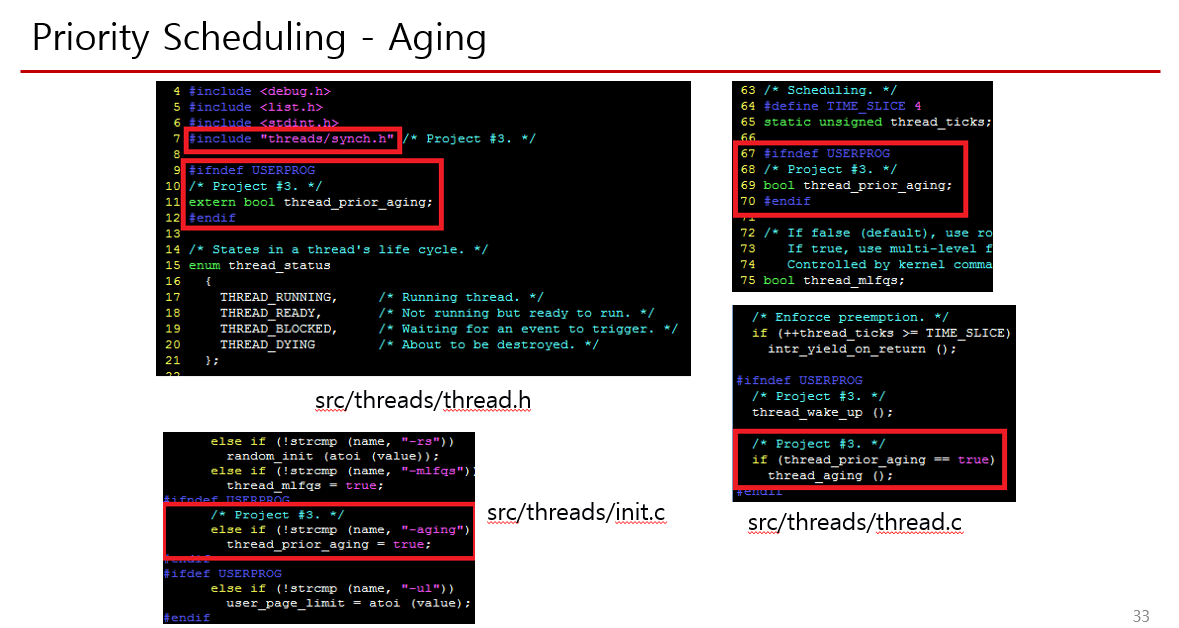
* 해당 프로젝트에서 구현할 내용을 간략히 서술.

기본으로 제공된 thread 시스템의 기능을 확장하여 Alarm Clock, Priority Scheduling, Advanced Scheduler (BSD Scheduler)를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

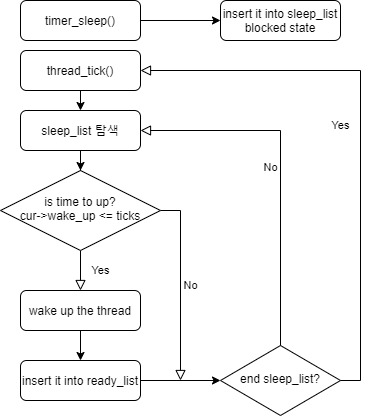
* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock
     + 기존의 thread의 running state와 ready state를 오가는 방식의 비효율적인 busy-waiting으로 구현되어 있는 timer\_sleep()와 timer\_interrupt()를 개선하여 CPU Time의 낭비를 막는다.
  2. Priority Scheduling
     + 각 thread에 priority를 부여하여 이 priority까지 고려하는 complex scheduler가 되도록 구현한다.
     + 낮은 우선순위를 가진 thread의 starvation 현상을 막기 위해, 시간이 지남에 따라 비례해서 priority가 높아지는 aging technique을 구현한다.
  3. Advanced Scheduler (추가구현을 한 경우)
     + BSD Scheduler에서는 각 priority마다 자신의 ready queue를 가지는 Multi-Level Feedback Queue(MLFQ)를 사용한다. 이번 프로젝트에서는 64개의 MLFQs를 사용한다. 각 ready queue는 round-robin 정책을 따른다.
     + schedule()이 호출되면 가장 높은 priority queue에서 round-robin 정책에 따라 다음 thread를 골라서 CPU에서 실행을 하게 된다.
  4. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.
   * 일어나야 될 시간이 되지 않았다면 ready state로 보내는 것이 아니라 blocked state로 가게 한다.
   * 이 blocked state를 관리하기 위해 새로운 queue를 만들어야 한다. 따라서 해당 thread를 이 queue(sleep\_list)에 넣을 때, 일어나야 할 시간(wake\_up)을 같이 저장해준다.
   * 시간이 되면, 해당 thread를 깨우고 다시 ready queue에 넣어준다.
   * timer\_interrupt()가 매 tick 마다 호출되면서 tick 값을 올려주는데, 이 함수에서 일어나야 될 thread를 찾아준다. 찾았으면, 해당 thread를 ready queue에 넣어준다.
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.
   * 현재 CPU에서 우선순위 31인 thread가 수행 중에 있고, 63의 우선순위를 갖는 thread가 새로 들어왔다면, 31보다 더 높은 우선순위를 가지므로, 현재 실행중인 thread와 switching이 일어나게 된다.
   * priority scheduling을 구현하기 위해서는 ready list의 우선순위에 따라 push 해야하므로 list\_push\_back() 대신 list\_inserted\_ordered()를 사용하여 우선순위에 따라 list에 push한다.
   * 여기서 priority의 범위는 0부터 63까지의 우선순위를 가지도록 한다. 기본적인 priority value는 31이다. 더 낮은 숫자가 더 낮은 우선순위를 의미한다.
3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)
   * BSD scheduler에서는 64개의 ready queue를 가지고, 초기값은 thread\_create()에서 default값인 31로 일반적으로 설정된다. 그 이후에 매 4 tick마다 시스템에 있는 모든 thread가 priority를 새로 계산하게 된다. 그 식은 다음과 같다.
   * priority = PRI\_MAX – (recent\_cpu / 4) – (nice \*2)
     + recent\_cpu: 최근에 thread를 돌리는 데 사용된 CPU Time이다. 최근에 사용한 소모한 CPU Time일수록 더 높은 가중치를 받게 되는 특징이 있다. time\_interrupt가 실행될 때마다 running state에 있는 thread의 recent\_cpu는 1씩 증가한다. 또한, 매초마다 시스템에 있는 모든 thread의 recent\_cpu 값들을 새로 계산한다. 이는 recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu + nice의 식을 따른다.
       - load\_avg: ready state에 있는 thread의 개수의 평균을 의미한다. 이는 system이 booting될 때 0으로 초기화 되며 매초마다 새로 계산된다.
       - load\_avg = (59/60)\*load\_avg+(1/60)\*ready threads의 식을 따른다.
       - 여기서 ready threads는 ready state 뿐만 아니라 running state에 있는 thread의 개수를 말한다.
     + nice: 각 thread는 priority와 관련된 nice value를 가지는 데, 범위는 -20 ~ 20이다.
   * 이 식을 사용하면 어떤 thread가 CPU Time을 많이 사용했다면, 그 thread는 다음 scheduling에서 더 낮은 priority를 갖게 된다.
4. **추진 일정 및 개발 방법**
   1. **추진 일정**

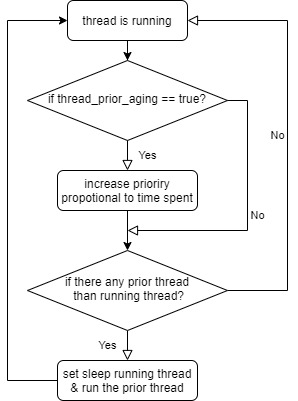
* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
* 11. 18 ~ 11. 24: 매뉴얼 분석 및 pintos 코드 이해
* 11. 25 ~ 11. 29: Alarm Clock, Priority Scheduling
* 11. 30 ~ 12. 05: Advanced Scheduler
* 12. 06: 마무리 작업, 보고서 작성
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수
  1. Alarm Clock
     + devices/timer.c에서 timer\_sleep()에서 기존의 while문으로 구현되어 있던 busy waiting 방식을 수정한다. blocked state를 관리하기 위한 새로운 queue인 sleep\_list를 새로 추가하고, 이를 timer\_init()에서 list\_init()을 시켜준다.
     + devices/timer.c에서 timer\_interrupt()에서 wake up할 thread가 있는지 확인하도록 수정한다.
     + threads/thread.h에서 struct thread 구조체에 int64\_t wake\_up을 추가하였다.
  2. Priority Scheduling
     + threads/synch.c에서 sema\_up()에서 일부 내용을 추가하였다.
     + threads/thread.c에서 thread\_create(), thread\_yield(), thread\_unblock()에서 thread가 우선 순위에 따라 scheduling 되도록 일부 수정해준다.
     + threads/thread.c에서 list\_insert\_ordered를 호출할 때 우선순위에 따라 list를 정렬할 수 있도록 comp\_priority()를 추가한다.
     + threads/thread.c에서 thread\_set\_priority()를 구현하여 current thread의 priority를 새로운 priority로 설정한다.
     + threads/thread.c에서 thread\_get\_priority()를 통해 current thread의 priority를 return한다.
     + aging technique을 위해 ppt 33쪽과 같은 자료구조들을 미리 추가하였다. 또한, threads/thread.c에서 thread\_aging()을 구현하였다.  
       
     + devices/timer.c에서 timer\_interrupt()의 내용을 thread\_prior\_aging에 따라서 수정하였다.
  3. Advanced Scheduler
     + devices/timer.c에서 timer\_interrupt()의 내용을 thread\_mlfqs에 따라서 수정하였다.
     + threads/thread.c에서 niceness를 위해 thread\_get\_nice(), thread\_set\_nice()를 구현한다. recent\_cpu를 위해 thread\_get\_recent\_cpu(), calc\_recent\_cpu()를 구현한다. load\_avg를 위해 thread\_get\_load\_avg(), calc\_load\_avg()를 구현한다.
     + threads/thread.c에서 위의 nice와 recent\_cpu를 사용해서 priority를 계산하는 calc\_priority()를 새로 구현하였다.
     + 또한, 위의 계산들에 사용되는 fixed-point format의 계산을 위해 float\_computation\_add(), float\_computation\_sub(), float\_computation\_mul(), float\_computation\_div()를 threads/thread.c에 구현하였다.
     + threads/thread.c에서 thread\_aging()을 업데이트해준다.

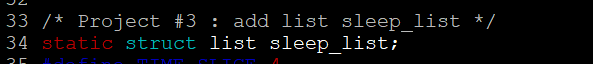
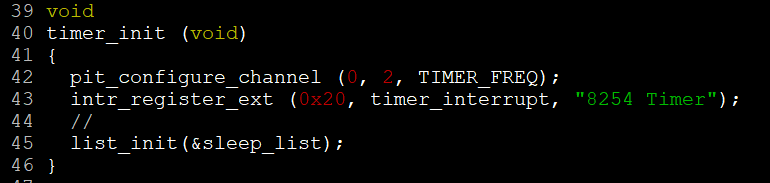
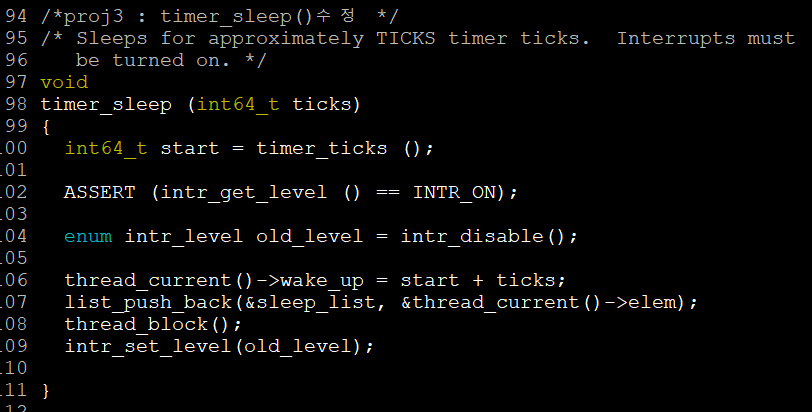
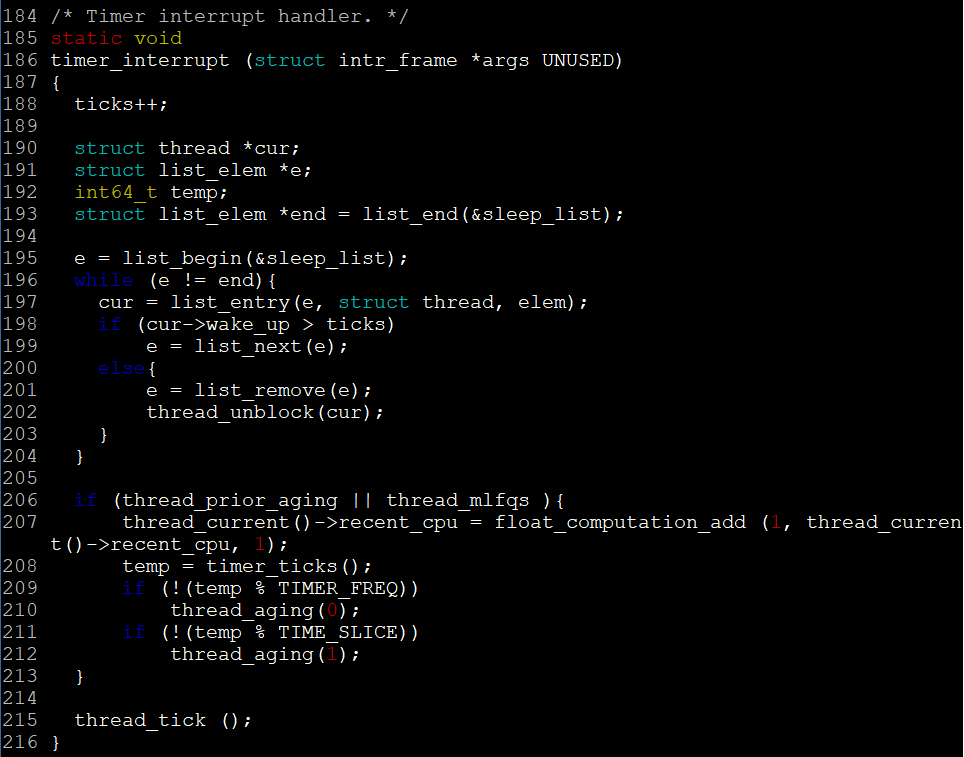
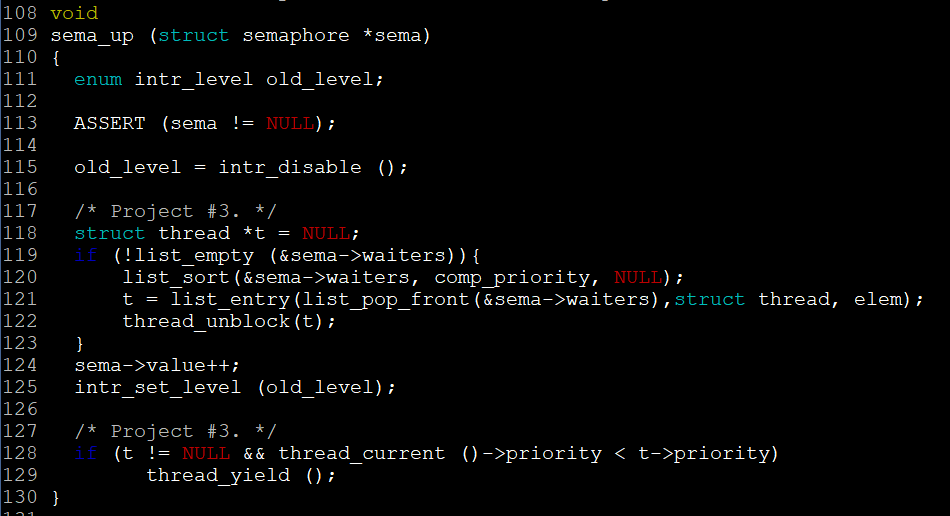
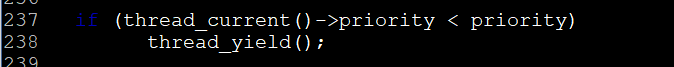
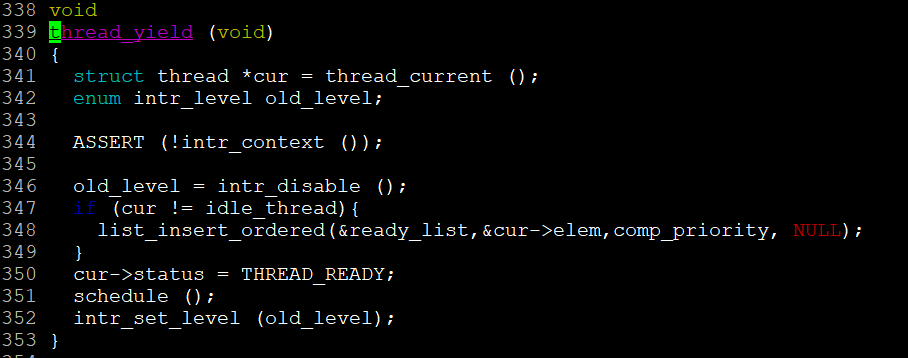
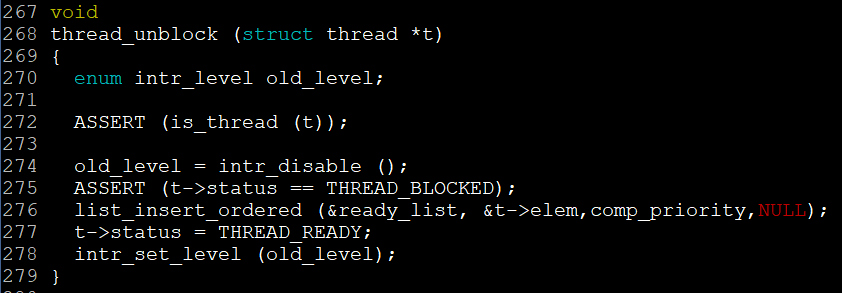
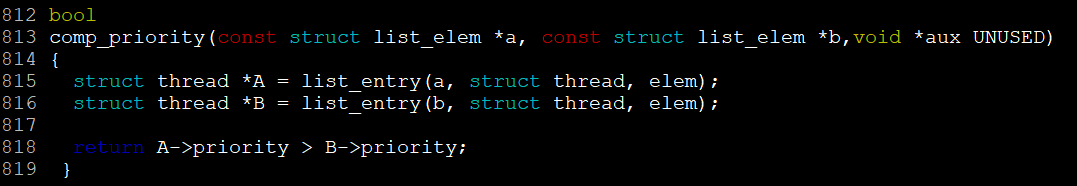
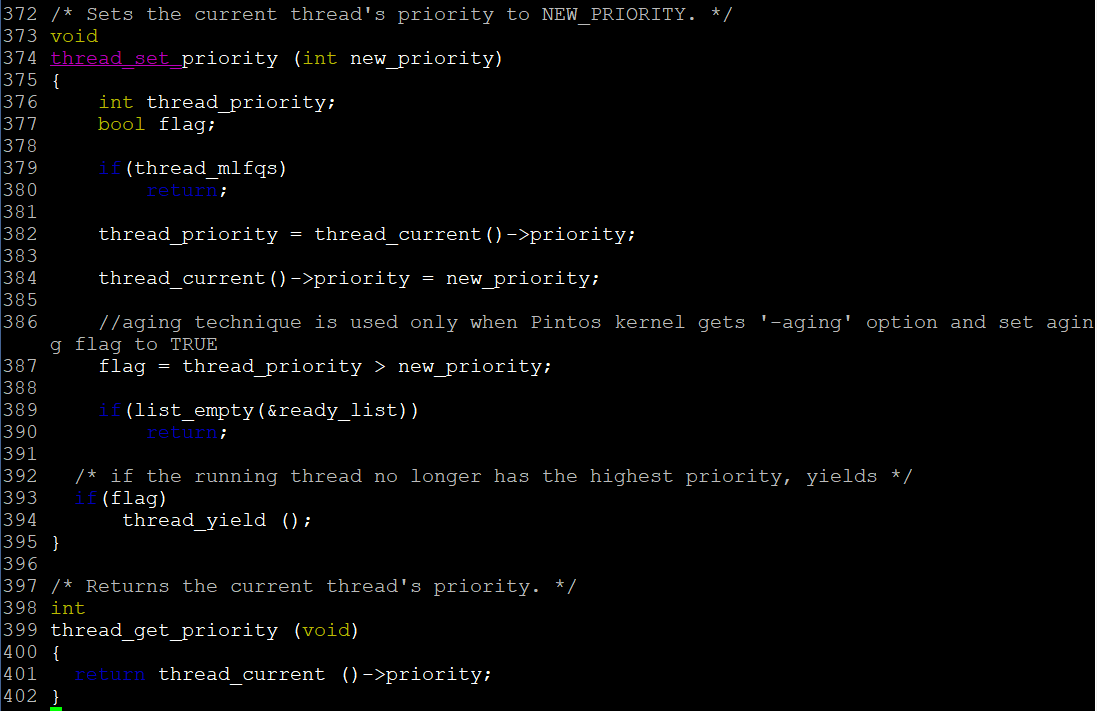
1. **연구 결과**
   1. **Flow Chart**

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

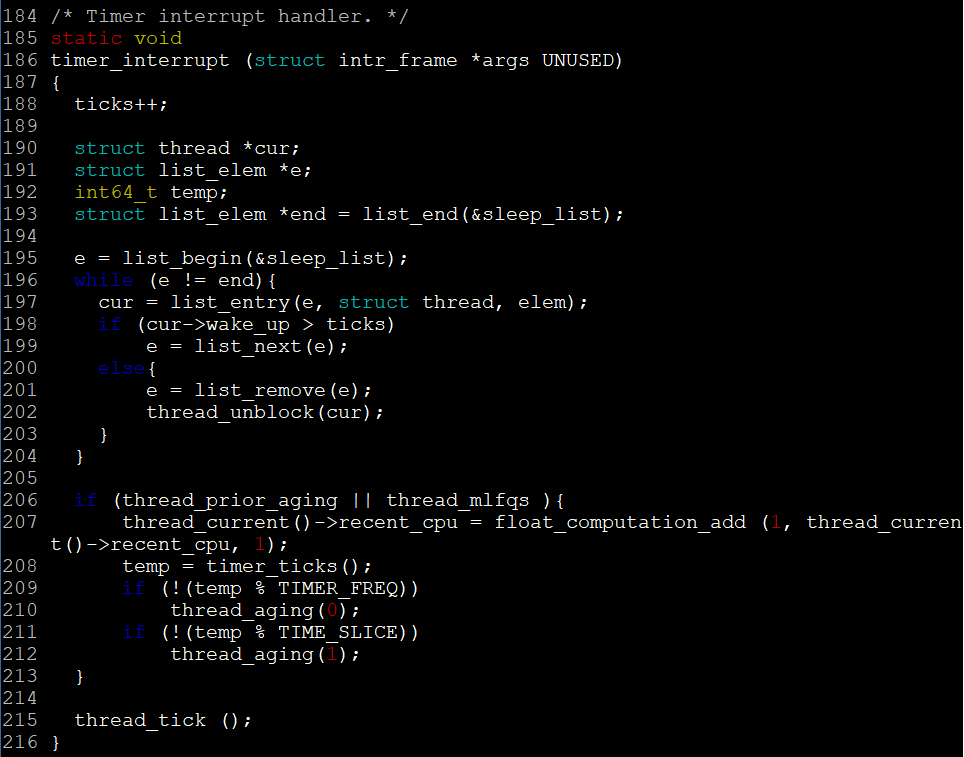
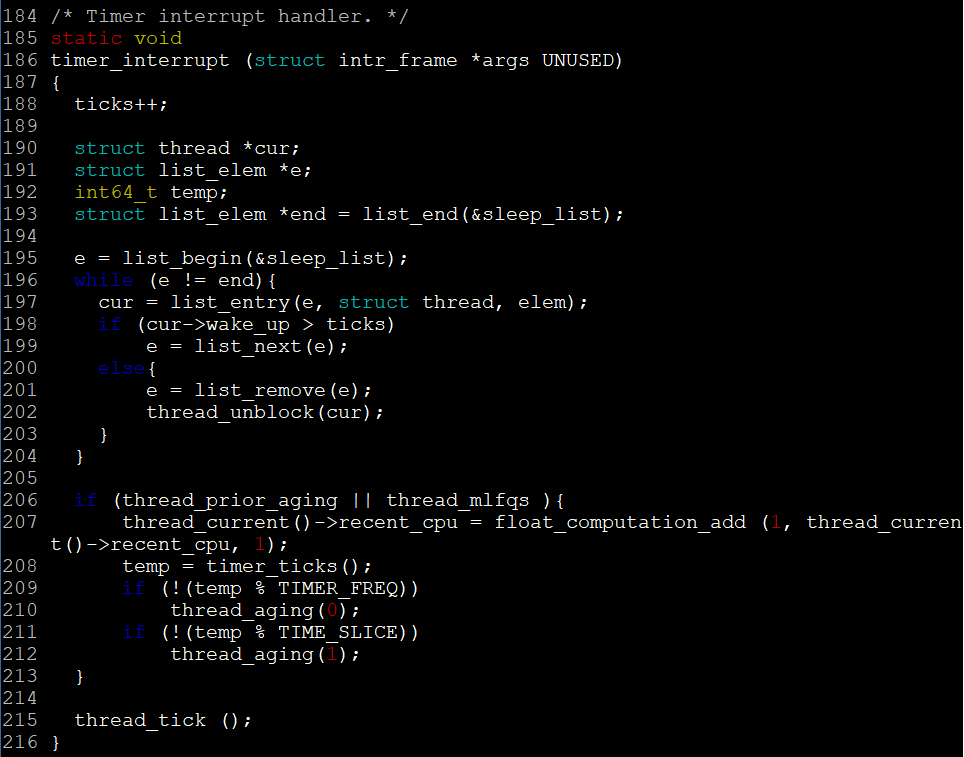
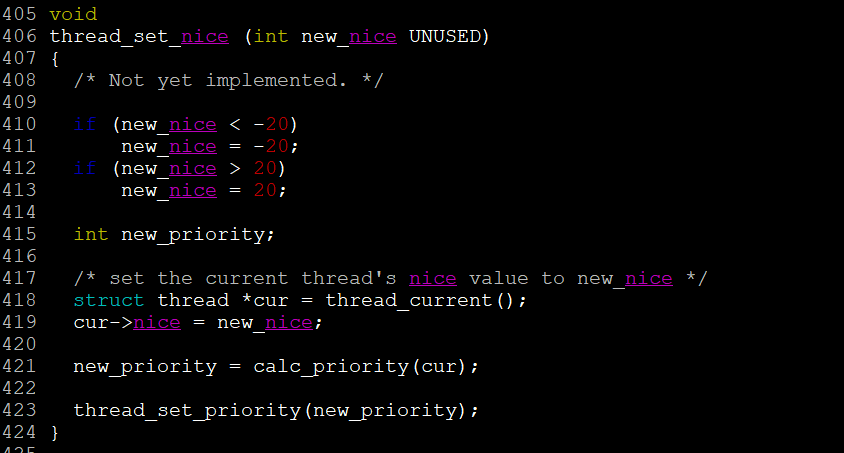
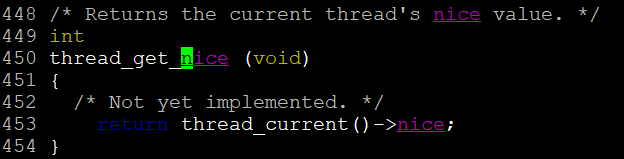
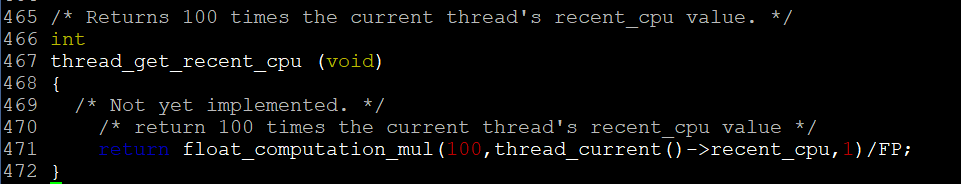
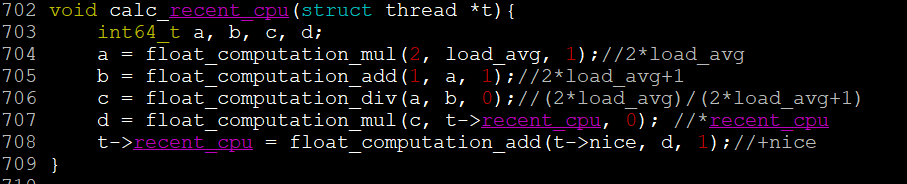
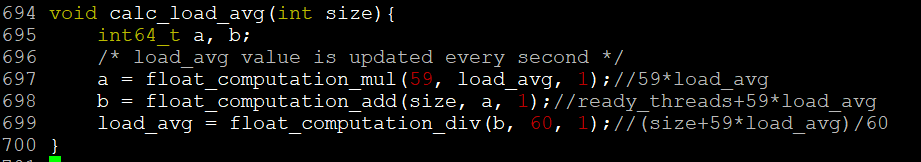
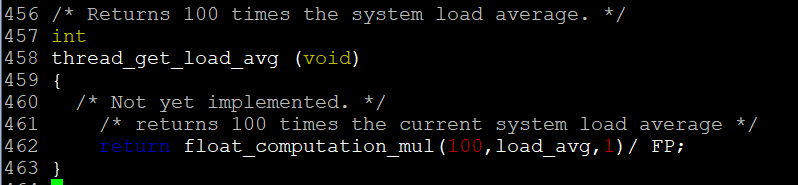
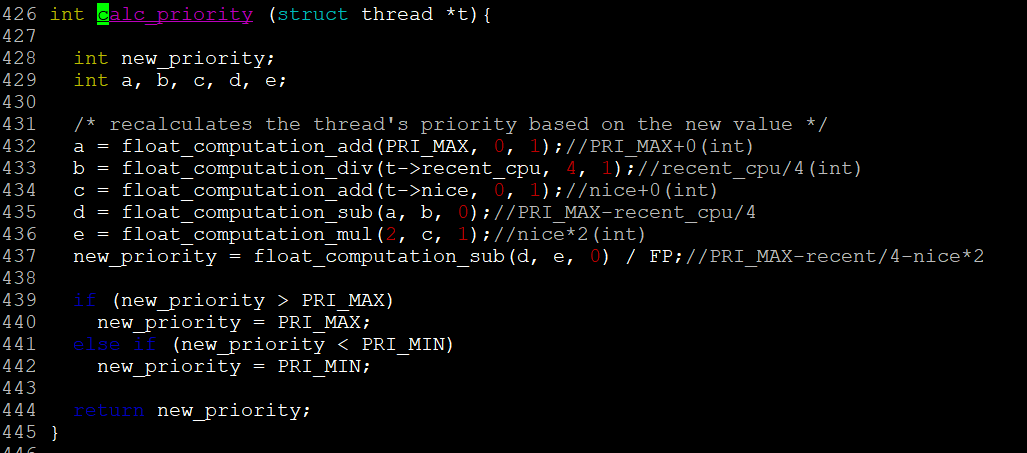
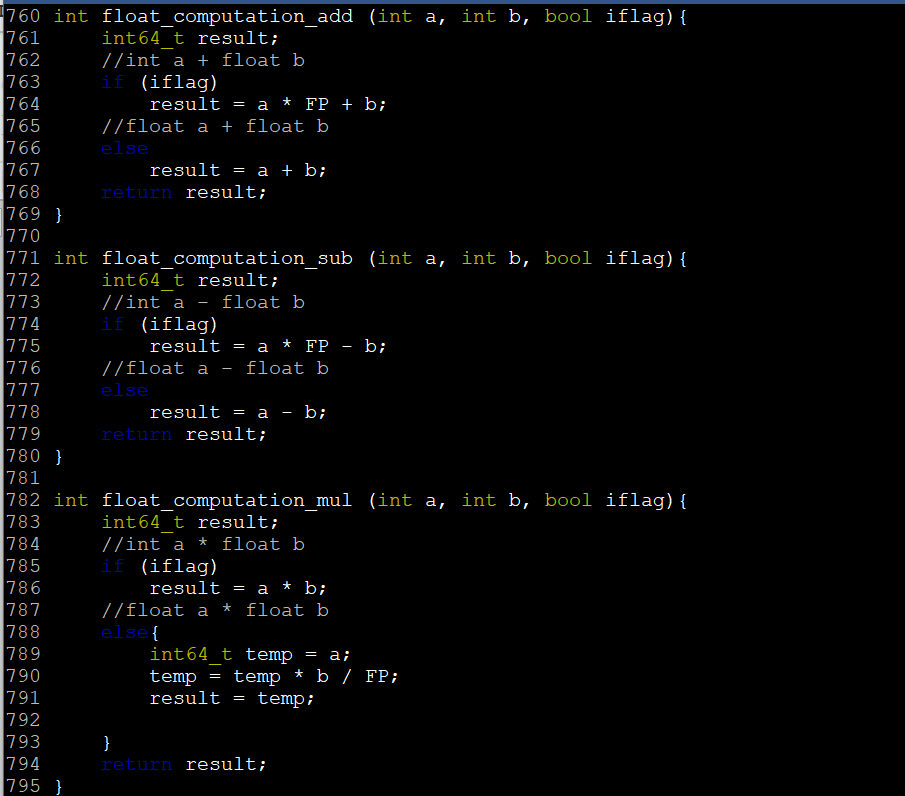
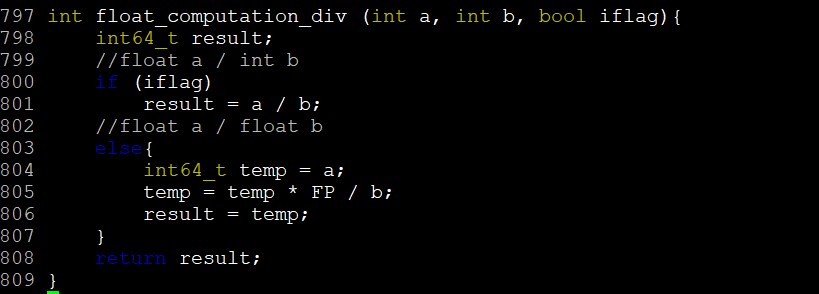
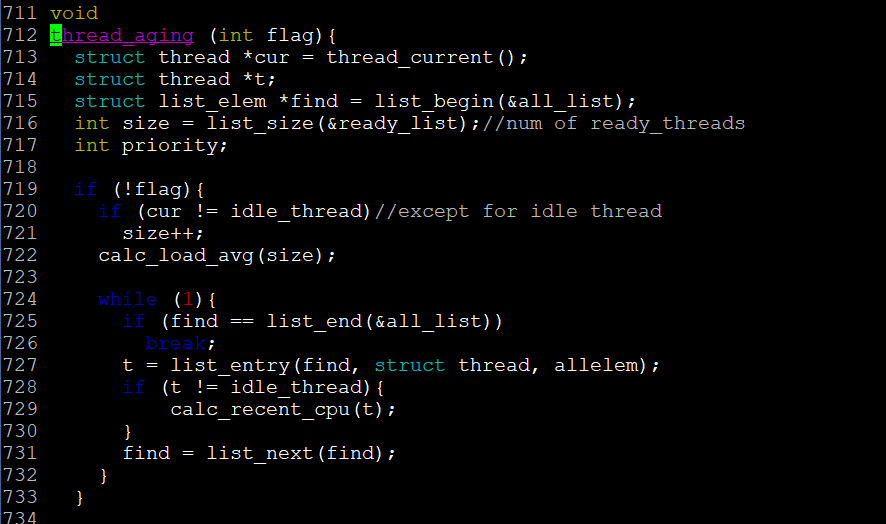
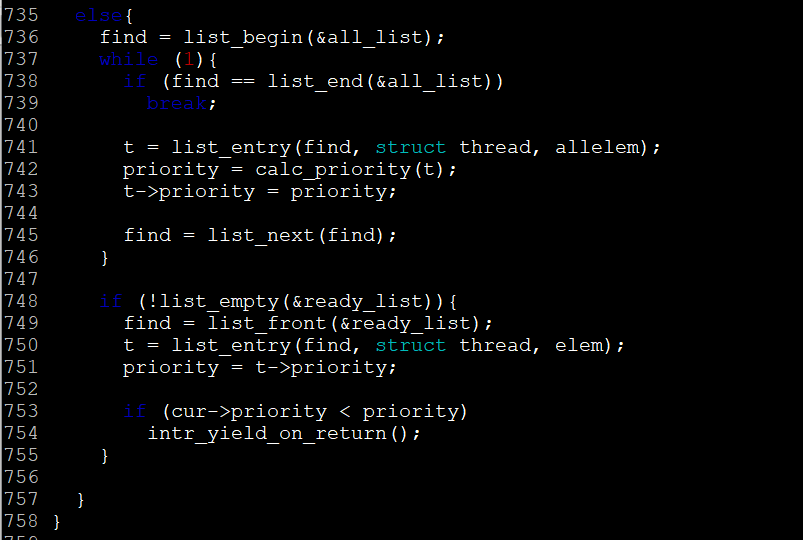
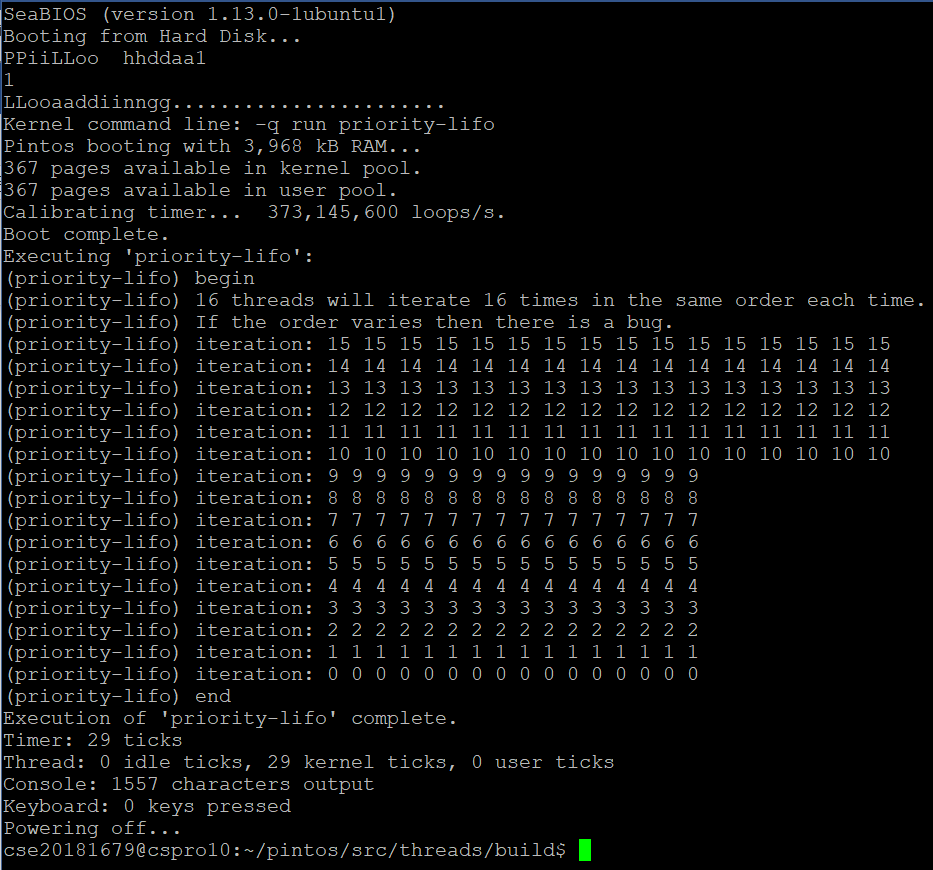
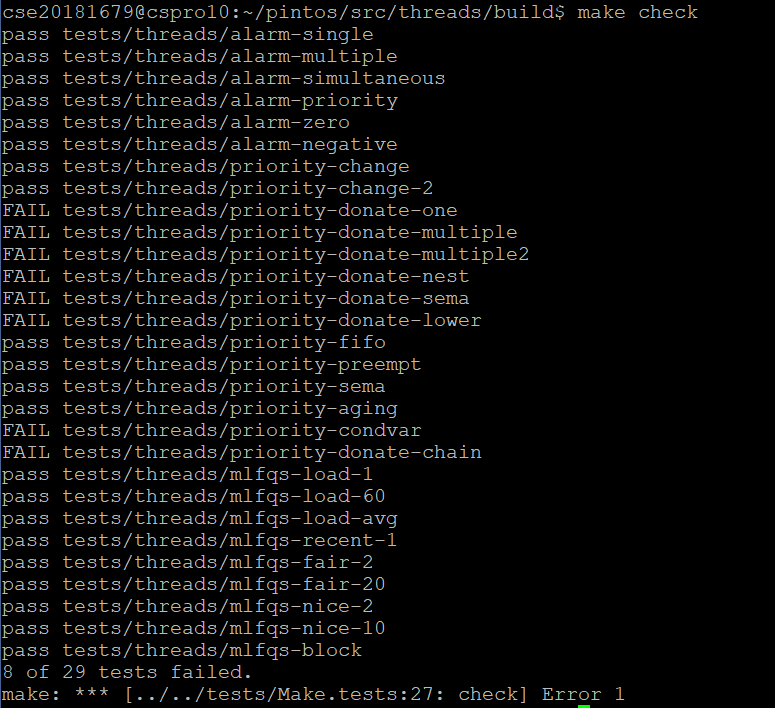


* 1. Priority Scheduling



* 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
     +    
       - devices/timer.c에서 sleep\_list를 추가하고, list\_init()을 해준다.
     +  **-** devices/timer.c에서 thread에 wake\_up을 저장해주고 sleep\_list에 넣어준 후 blocked state로 바꿔준다. wake\_up은 처음 CPU를 할당 받은 순간부터 메모리에 있었던 시간만큼 더한 것을 의미한다.
     +  **-** devices/timer.c에서 sleep\_list에 있는 thread를 탐색하면서 wake\_up이 ticks보다 작거나 같으면 그 thread를 sleep\_list에서 제거하고 unblock 시킨다.   
       - thread\_tick() 함수를 호출하여 tick을 한번 더 계산한다.
  2. Priority Scheduling
     +  **-** threads/synch.c에서 sema를 얻기 위해 기다리는 thread가 존재한다면, 우선 순위가 가장 높은 thread를 가져와서 waiter list에서 빼주고, thread를 unblock하여 다음 cpu를 얻을 후보로 만들어준다.
     +   
       - threads/thread.c에서 thread\_create()에 다음과 같은 코드를 추가하여 현재 실행중인 thread의 우선 순위보다 새로 생성한 thread의 priority가 더 크다면 thread\_yield()를 호출한다.
     +   
       - threads/thread.c에서 thread\_yield(), thread\_unblock()에서 list\_insert\_ordered()를 호출하여 thread가 우선 순위에 따라 ready\_list에 삽입되도록 수정해준다.
     +   
       - threads/thread.c에서 list\_insert\_ordered를 호출할 때 우선순위에 따라 list를 정렬할 수 있도록 comp\_priority()를 추가하여 list\_insert\_ordered()의 인자로 넘겨준다. a의 priority와 b의 priority을 비교하여 bool값을 반환한다.
     +   
       - threads/thread.c에서 thread\_set\_priority()를 구현하여 mlfqs 방식이 아닐 경우, current thread의 priority를 새로운 priority로 설정한다. 현재 thread의 priority보다 new\_priority가 작으면 thread\_yield()를 호출한다.

- threads/thread.c에서 thread\_get\_priority()를 통해 current thread의 priority를 return한다.

* + - - threads/thread.c에서 thread\_aging()는 advanced scheduler에서 발전시켰으므로 아래에서 설명한다. recent\_cpu에 비례한 값을 priority로 가지게 하여 낮은 priority도 높아지도록 하였다.
    -  **-** devices/timer.c에서 timer\_interrupt()의 내용을 thread\_prior\_aging에 따라서 수정하였다.  
      - 현재 thread의 recent\_cpu를 float\_computation\_add을 통해 1을 더해서 새로 계산해준다. 또한, TIMER\_FREQ의 배수이거나 TIME\_SLICE의 배수이면 thread\_aging 함수를 통해 우선순위를 다시 계산해준다.   
      - float\_computation\_add는 advanced scheduler에서 설명되어있다.
  1. Advanced Scheduler
     +   
       - devices/timer.c에서 timer\_interrupt()의 내용을 thread\_mlfqs에 따라서 수정하였다.  
       - 현재 thread의 recent\_cpu를 float\_computation\_add을 통해 1을 더해서 새로 계산해준다. 또한, TIMER\_FREQ의 배수이거나 TIME\_SLICE의 배수이면 thread\_aging 함수를 통해 우선순위를 다시 계산해준다.
     +    
       - threads/thread.c에서 niceness를 위해 thread\_get\_nice(), thread\_set\_nice()를 구현한다.   
       - thread\_get\_nice()는 current thread의 nice값을 반환한다.  
       - thread\_set\_nice()는 nice의 값의 범위를 확인해주고, current thread의 nice값을 새로운 nice값으로 설정해준 뒤, 새로운 nice 값으로 재계산된 priority에 대하여 thread\_set\_priority()를 호출한다.   
          
       - recent\_cpu를 위해 thread\_get\_recent\_cpu(), calc\_recent\_cpu()를 구현한다.   
       - calc\_recent\_cpu()는 recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu + nice의 식을 따라 재계산한다.  
       - thread\_get\_recent\_cpu()는 current thread의 recent\_cpu의 100배 값을 return한다.  
         
       - load\_avg를 위해 thread\_get\_load\_avg(), calc\_load\_avg()를 구현한다.  
       - calc\_load\_avg()는 load\_avg = (59/60) \* load\_avg + (1/60) \* ready threads의 식을 따라 재계산한다.  
       - thread\_get\_load\_avg()는 현재 시스템의 load\_avg의 100배 값을 return한다.
     +   
       - threads/thread.c에서 위의 nice와 recent\_cpu를 사용해서 priority를 계산하는 calc\_priority()이다.  
       - 매 4 tick마다 system의 모든 thread의 priority가 재계산되는데 이 식이 사용된다. 재계산 한 후 범위 체크를 해주고 아래 식을 따른 새로운 priority이 return된다.  
       - priority = PRI\_MAX – (recent\_cpu / 4) – (nice \* 2)
     +    
       - 위의 계산들에 사용되는 fixed-point format의 계산을 위해 float\_computation\_add(), float\_computation\_sub(), float\_computation\_mul(), float\_computation\_div()를 threads/thread.c에 구현하였다.  
       - 이는 정수+실수, 실수+실수, 정수–실수, 실수-실수, 정수\*실수, 실수\*실수, 실수/정수, 실수/실수의 기능을 수행한다.
     +   
         
       - threads/thread.c에서 thread\_aging()을 업데이트해준다.  
       - parameter로 받는 flag는 thread\_prior\_aging이다. flag가 0일 때는 매초마다 load\_avg와 recent\_cpu를 업데이트하고, flag가 1일 때는 4초마다 priority를 업데이트 해준다.  
       - flag가 0일 경우는 현재 ready list에 있는 thread의 개수를 size라는 변수에 받아 , 현재 thread가 idle이 아니면 메모리에 있는 thread라 판단하여 size를 1 증가시킨다. 그리고 load\_avg를 update해준다. time\_interrupt가 실행될 때마다 running state에 있는 thread의 recent\_cpu를 1씩 증가시킨다. 또한, 매초마다 시스템에 있는 모든 thread의 recent\_cpu 값들을 새로 계산한다. 이는 calc\_load\_avg()와 calc\_recent\_cpu()를 통해 계산할 수 있다.  
       - flag가 1인 경우에는 4초마다 모든 thread의 우선순위를 calc\_priority()를 호출하여 재계산시킨다.   
       - ready\_list에 있는 thread를 탐색하면서 우선순위가 정해진 범위를 벗어나지 않도록 해준다. 또한 현재 thread가 우선순위가 가장 높지 않다면 interrupt 중이므로 intr\_yield\_on\_return을 호출해준다.
  2. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석
  1. priority-lifo.c
     + priority는 priority는 PRI\_DEFAULT + i(0 <= i <= THREAD\_CNT) + 1이다. 따라서 처음 main thread의 자식들은 실행되지 않는다.
     + 그 후 main thread의 priority를 PRI\_DEFAULT 값으로 설정해 priority가 15인 thread부터 순서대로 실행된다. ready list에서 나오는 thread는 우선순위 기준으로 정렬되어 있기 때문에 그 이후에도 우선순위 기준으로 thread가 실행된다.
     + 결과적으로 아무리 반복하여도 같은 output이 나오게 된다.
     + 이는 결론적으로 priority가 높은 thread가 나중에 들어갔을 경우에 대해 테스트를 해준다. 이 테스트는 priority가 높은 thread가 나중에 들어오더라도 먼저 수행을 해줘야 정상 작동한다고 체크한다.
  2. 아래 수행 화면에서 priority가 높은 15의 thread가 나중에 ready\_list로 들어가더라도 가장 먼저 수행되고 종료되었음을 볼 수 있다. 가장 낮은 priority의 0의 thread는 가장 먼저 ready\_list로 들어가더라도 가장 마지막에 실행되었음을 확인할 수 있었다.
  3. 수행 화면  
     
* make check 수행 결과를 캡처하여 첨부
  1. make check  
     
  2. make grade  
     