**Pintos Project 3: Threads**

담당 교수 :

조 / 조원 :

개발 기간 :

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

첫번째로 Alarm Clock에 대해 구현한다. 현재 RUNNING 상태인 쓰레드에서 timer\_sleep() 함수를 호출하면 지정한 ticks가 지나갈 때까지 thread\_yield()를 반복적으로 호출한다. 이런 방식은 조건을 지속적으로 확인하여 CPU 자원을 비효율적으로 사용한다. 그리고 여러 쓰레드가 sleep 상태에 들어가 thread\_yield() 함수를 계속 호출하면 스케쥴링 효율성이 떨어진다. 이를 해결하기 위해 sleep 상태의 쓰레드를 한데 모아 관리할 수 있는 자료구조를 구현하고자 한다.

두번째로 Priority Scheduling에 대해 구현한다. 기존의 방식은 우선순위를 이용하지 않는 단순한 Round Robin 방식이다. 이를 변경하여 PRIORITY가 높은 쓰레드가 스케쥴링의 우선권을 받을 수 있도록 ready\_list를 PRIORITY에 대한 내림차순 정렬 상태를 유지하는 것을 구현하고자 한다. 또한 특정 semaphore에 대해 기다리는 waiter 쓰레드들에 대해서도 가장 PRIORITY가 높은 waiter가 우선 스케쥴링 될 수 있도록 구현한다.

세번째로 Advanced Scheduler를 구현한다. PRI\_MIN과 PRI\_MAX에 해당하는 각각의 ready list가 존재하며 가장 높은 PRIORITY를 가진 non-empty list에서 scheduling할 쓰레드를 선택하며, 쓰레드가 여러 개일 경우 Round Robin 방식을 취한다. 그리고 여러 metric들을 통해 일정 시간마다 모든 쓰레드에 대한 PRIORITY를 update하는 것을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

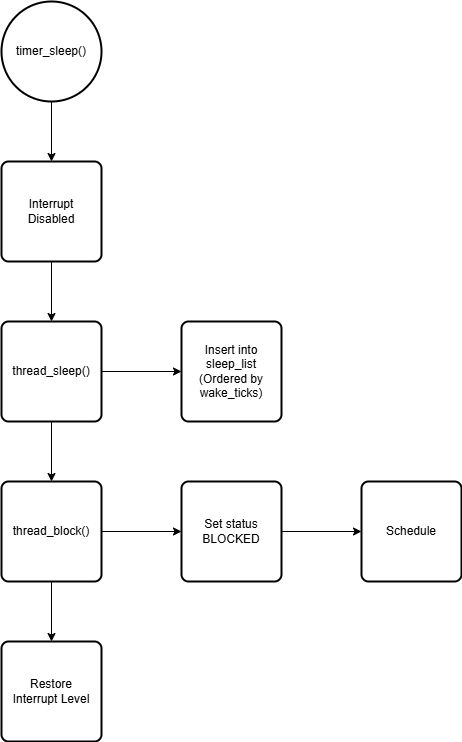
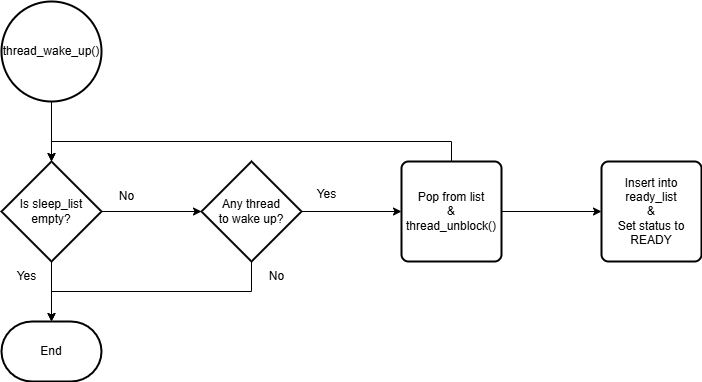
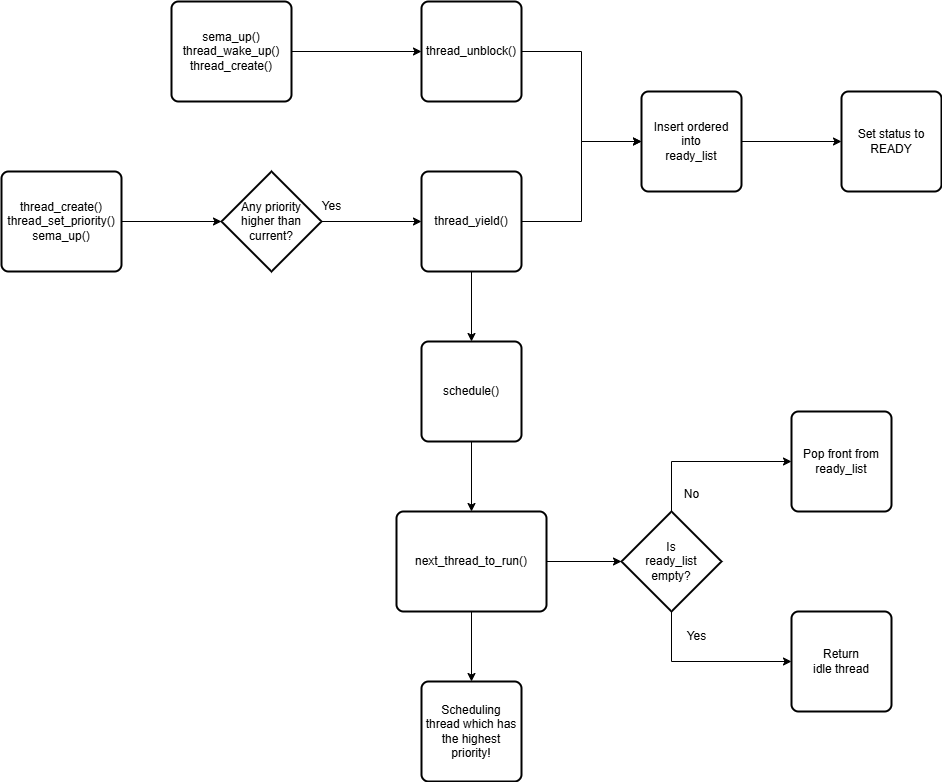
* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock  
     현재 반복문을 통한 thread\_yield() 호출 방식은 BLOCKED 상태의 쓰레드가 READY가 되었다가 다시 BLOCKED으로 돌아오는 과정이 반복적으로 수행되기 때문에 CPU의 자원을 비효율적으로 사용한다. 그래서 sleep 상태의 쓰레드를 관리하는 list를 만들어 별도로 관리한다. 이를 통해 CPU 낭비 시간을 최소화하고 스케쥴러의 효율성을 높일 수 있을 것이다.
  2. Priority Scheduling  
     현재 사용중인 Round Robin Scheduling은 모든 쓰레드에 대해 똑같은 비중을 부여하기 때문에 중요하거나 긴급한 작업이 지연될 수 있다. 그래서 ready list를 priority에 따라 내림차순으로 정렬된 상태를 유지하여 우선순위가 높거나 긴급한 작업이 우선적으로 수행될 수 있도록 하여 스케쥴링의 효율성을 높일 수 있다. 또한 semaphore waiters들에 대해서도 같은 기준을 적용하여 우선권이 높은 쓰레드가 먼저 깨어날 수 있도록 하여 synchronization mechanism이 최적의 쓰레드를 선택할 수 있다.
  3. Advanced Scheduler (추가구현을 한 경우)  
     Priority Scheduler에서 보다 발전한 Advanced Scheduler를 구현하는데, nice, recent\_cpu, load\_avg와 같은 metric을 이용하여 모든 쓰레드에 대한 priority를 주기적으로 update한다. 이러한 dynamic priority update를 통해 starvation을 방지하고 CPU 효율성을 높일 수 있다.
  4. **개발 내용**
* 아래 항목의 내용만 서술

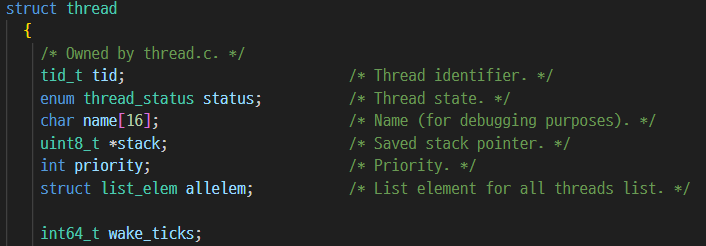
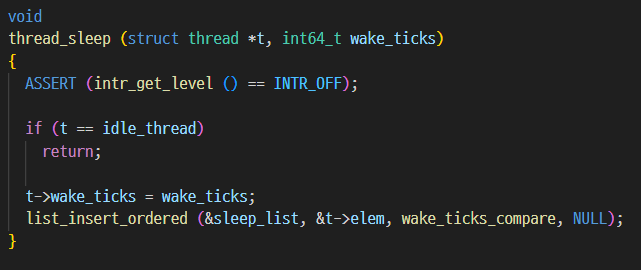
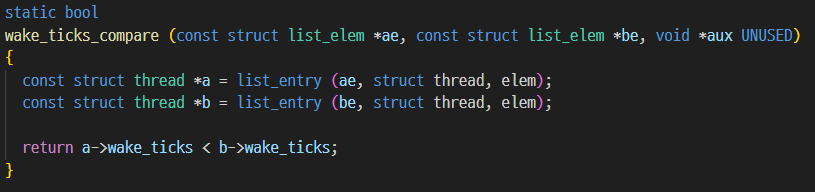
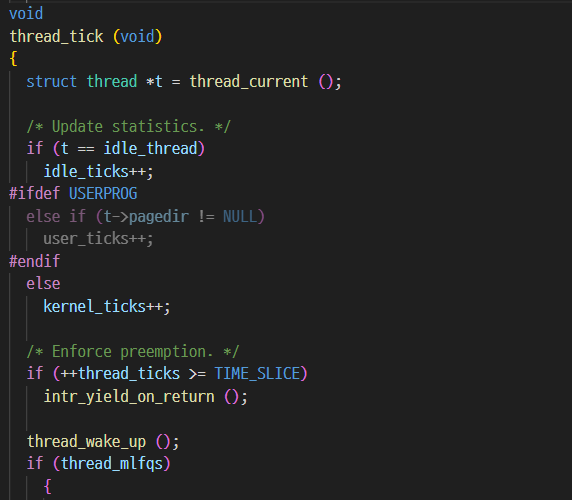
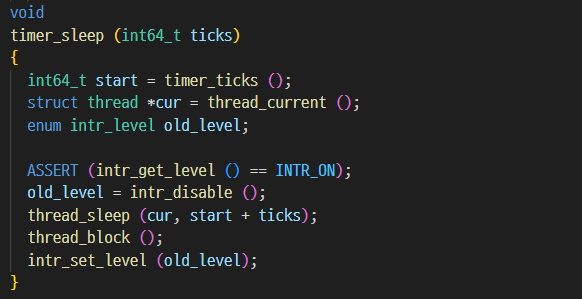
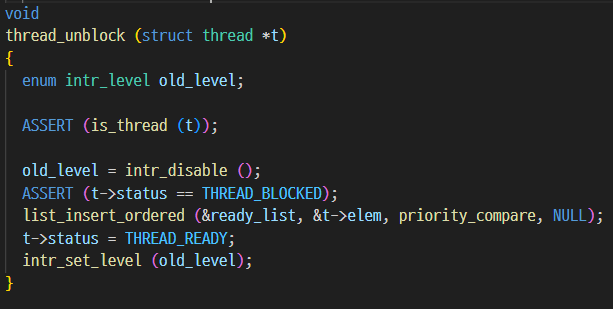
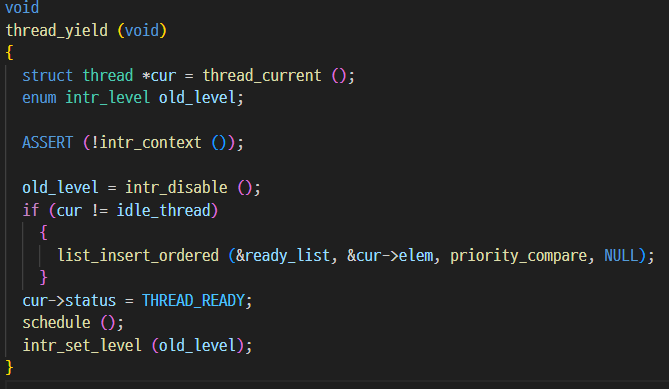
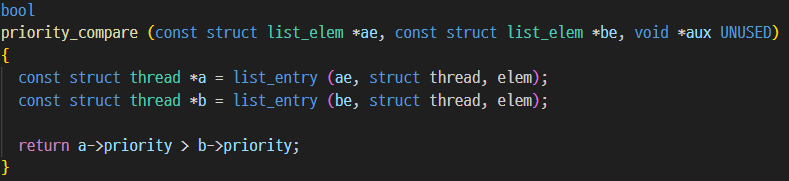
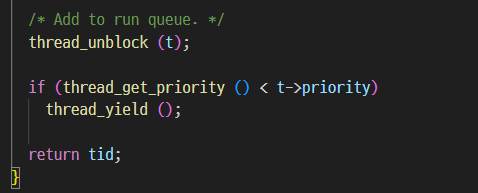
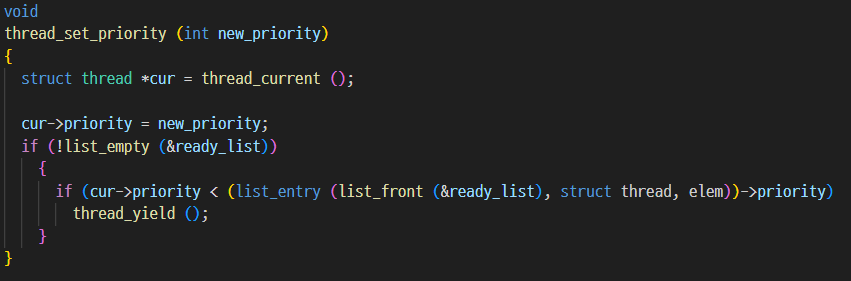
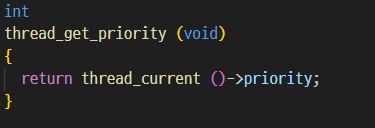
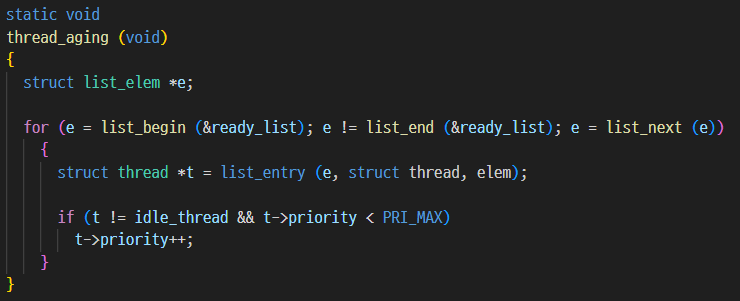
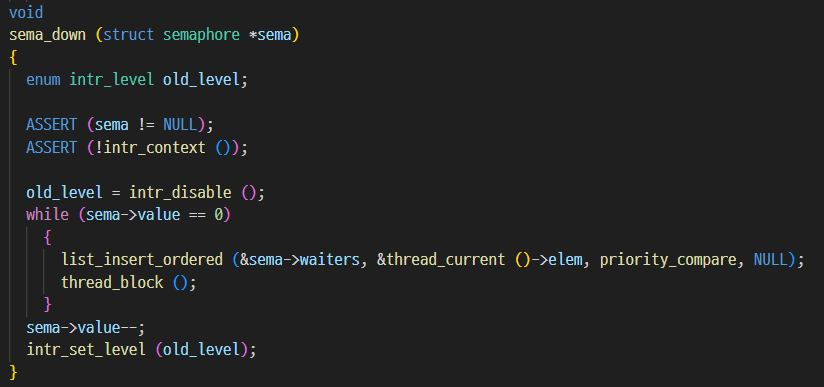
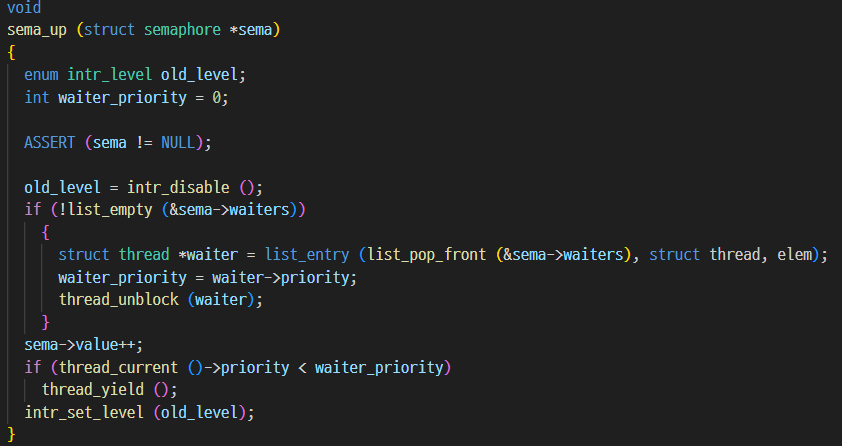
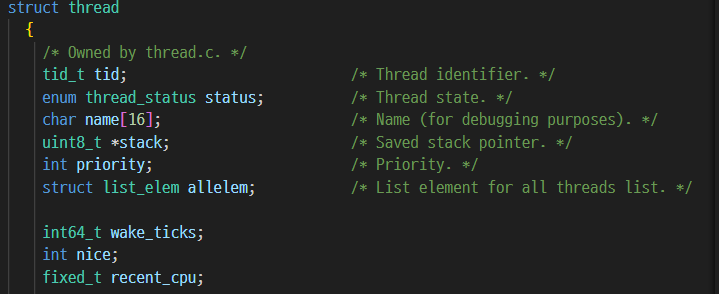
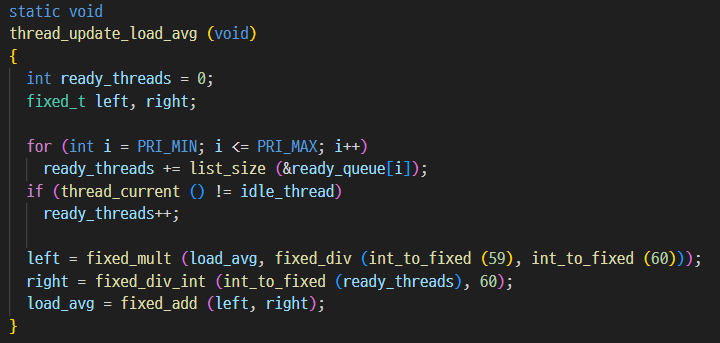
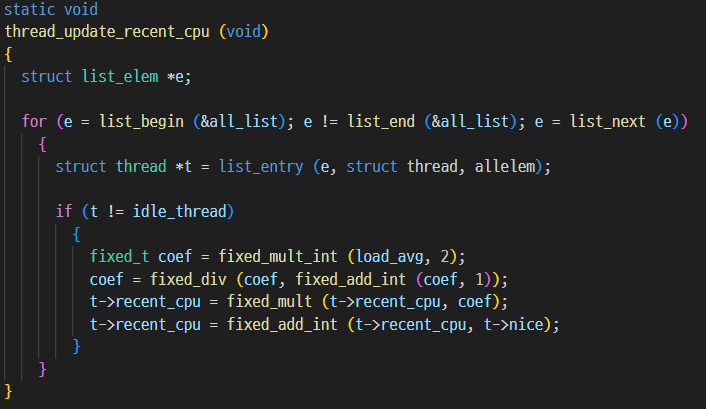
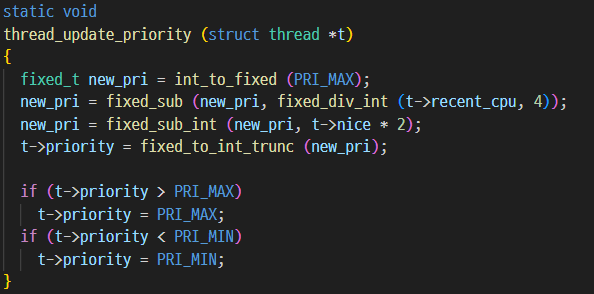
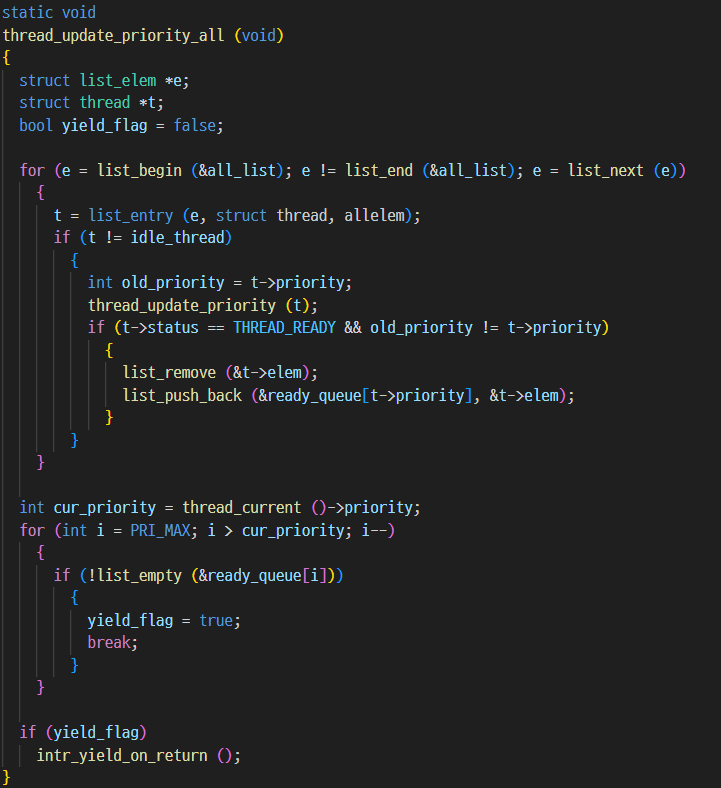
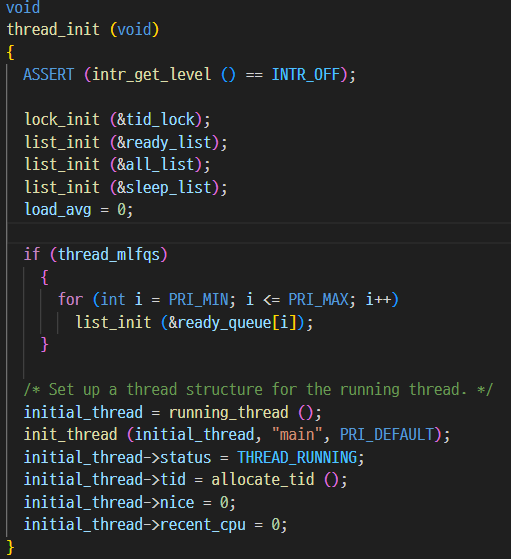
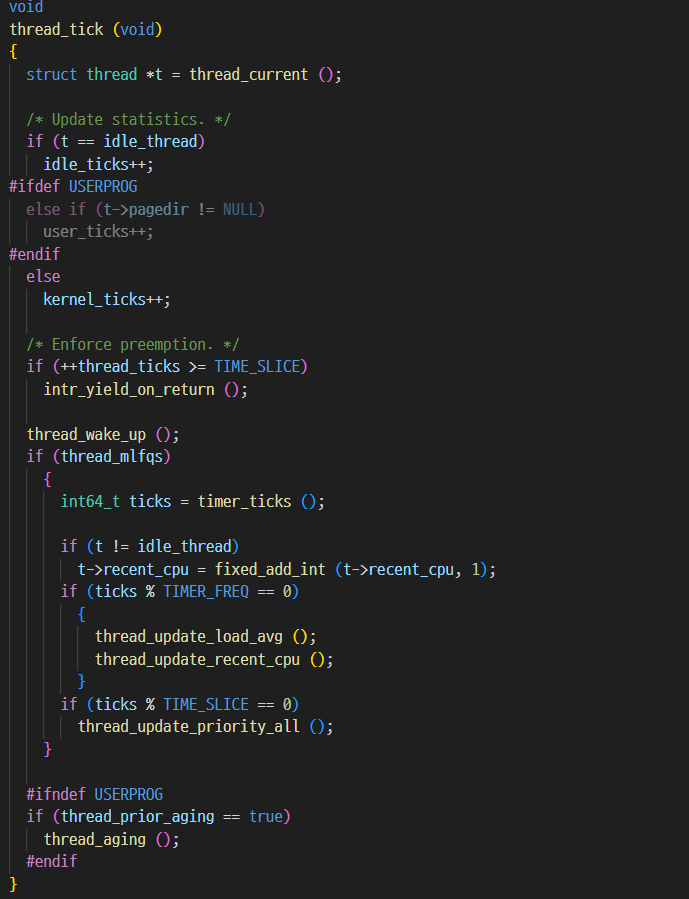
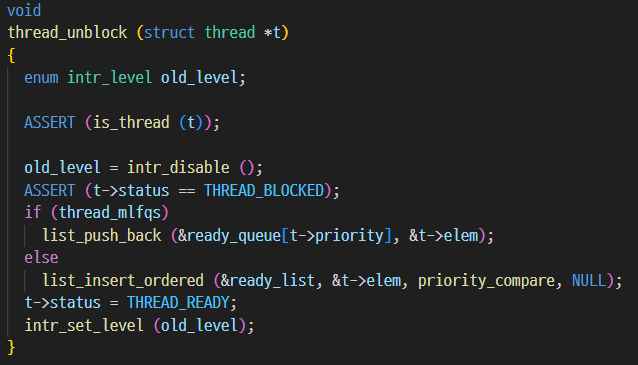
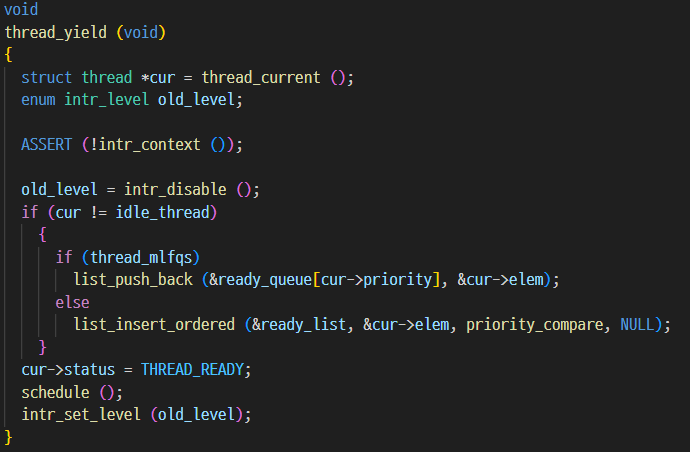
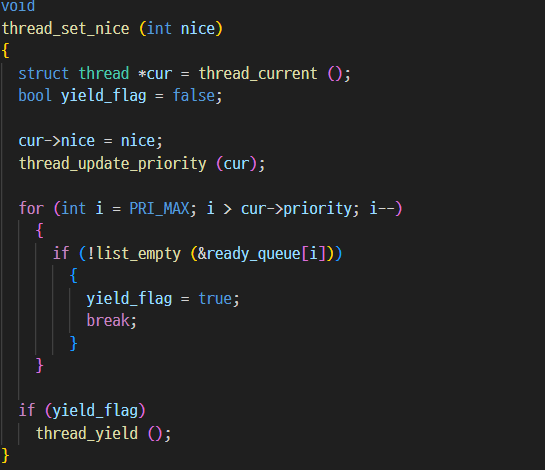
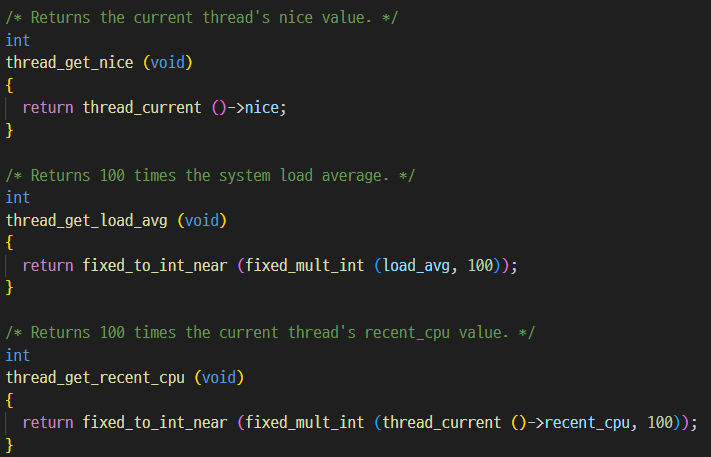
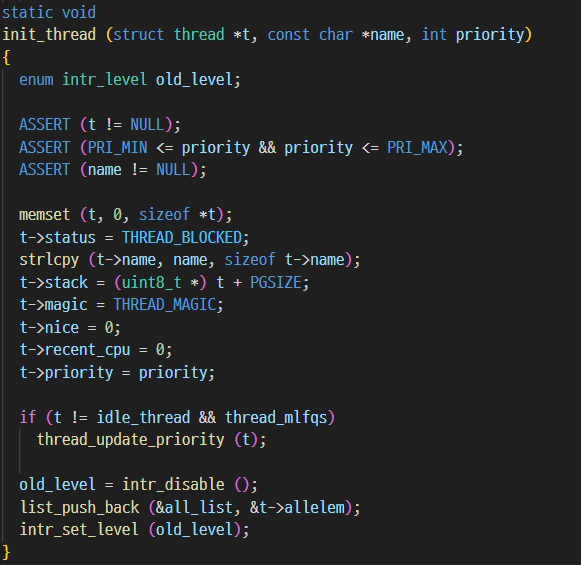
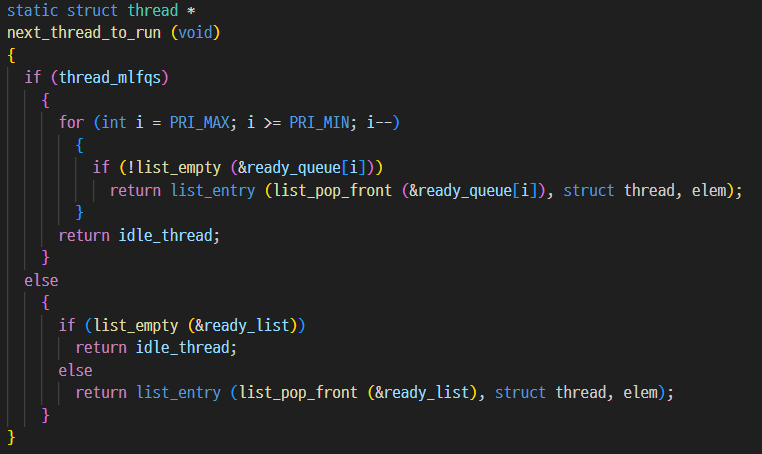
1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.  
   (1) thead\_sleep()을 통해 sleep\_list에 삽입된 경우  
   우선 thread\_sleep() 함수를 통해 스레드를 sleep\_list에 삽입할 때마다 깨어나야 하는 ticks에 대한 오름차순으로 정렬된 상태를 유지한다. 그리고 timer interrupt handling 과정에서 thread\_wake\_up() 함수를 호출하는데, wake ticks에 대한 오름차순으로 정렬이 되어있으므로 sleep\_list의 front 요소만 반복적으로 확인하면서 깨어나야 하는 스레드가 있다면 thread\_unblock()을 호출하여 ready\_list 혹은 ready\_queue에 삽입한다. 마지막으로 스레드의 status를 THREAD\_READY로 변경한다.  
   (2) semaphore의 waiter에 삽입된 경우  
   우선 sema\_down()을 수행하는 과정에서 waiters list에 스레드를 삽입해야 하는 경우 priority에 대한 내림차순 정렬을 유지하면서 list에 삽입한다. 그리고 sema\_up()이 호출되어 waiters에 있는 스레드 중 하나를 깨워야 할 때 list의 front 요소를 꺼내와서 thread\_unblock()을 호출하면 가장 priority가 높은 waiter 스레드를 깨울 수 있다.
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.  
   Priority scheduling에서는 항상 가장 높은 priority를 가진 스레드가 RUNNING 상태를 유지해야 한다. 만약 현재 RUNNING 상태인 스레드의 priority가 변경되거나, 주기적으로 모든 스레드에 대한 priority update가 이루어졌을 때 ready\_list에 현재 스레드보다 높은 priority를 가진 스레드가 있다면 그 즉시 현재 스레드는 CPU를 양보해야 한다. 이렇게 운영체제의 적극적인 개입으로 스케줄링 하는 방식을 preemptive scheduling이라고 한다.
3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)  
   우선 priority 계산식은 아래와 같다.  
     
   priority = PRI\_MAX – (recent\_cpu / 4) – (nice \* 2)  
     
   여기서 PRI\_MAX는 pintos 상에서 가장 높은 priority인 63을 의미하고, recent\_cpu는 스레드가 최근에 사용한 CPU 시간을 나타낸다. 마지막으로 nice는 스레드의 nice 값을 나타낸다.  
   recent\_cpu의 계산식은 아래와 같으며, 매초마다 모든 스레드에 대해 갱신된다.  
     
   recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu + nice  
     
   여기서 load\_avg는 READY 상태인 스레드의 개수에 대한 평균값이며 계산식은 아래와 같다.  
     
   load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_threads  
     
   여기서 ready\_threads는 READY와 RUNNING 상태인 스레드의 수를 의미하여, idle thread는 제외한다.  
     
   priority 계산 과정은 ready\_threads부터 recent\_cpu까지 차례대로 갱신한 다음 priority를 계산하면 된다.

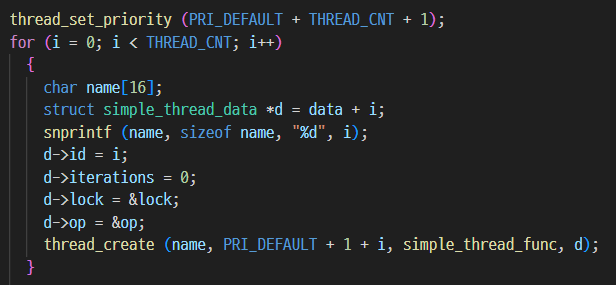
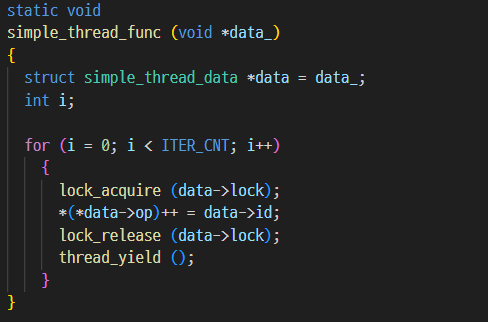
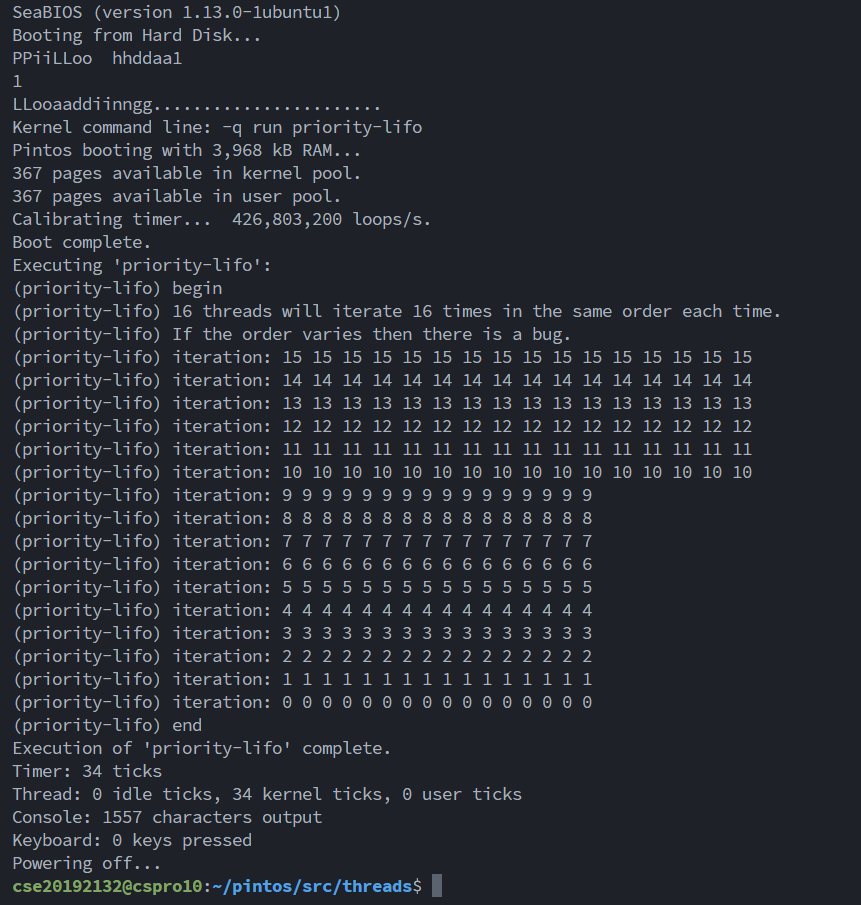
1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성  
  11/9 : Alarm Clock, Priority Scheduling 구현  
  11/10 ~ 11/11 : Advanced Scheduler 구현
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드  
    threads/thread.c  
    threads/thread.h  
    threads/synch.c  
    devices/timer.c  
    threads/fixed.h => fixed-point operation을 위해 새롭게 추가한 파일
  + 수정하거나 추가해야 하는 자료구조  
    1. threads/thread.c  
    - struct list sleep\_list : sleep 상태의 스레드를 관리하는 list 자료구조  
    - struct list ready\_queue[] : BSD Scheduler 구현을 위한 multi-level queue 자료구조  
      
    2. threads/thread.h – struct thread 멤버 변수 추가  
    - int64\_t wake\_ticks : sleep 상태의 스레드가 깨어나야 하는 timer tick  
    - int nice : nice value  
    - fixed\_t recent\_cpu : recent\_cpu value  
      
    3. threads/fixed.h  
    - 새로운 자료형 fixed\_t (typedef int)
  + 수정하거나 추가해야 하는 함수  
    1. threads/thread.c  
    - void thread\_sleep() : 스레드의 wake\_ticks 저장 후 sleep\_list에 정렬 삽입.  
    - void thread\_wake\_up() : sleep\_list에서 깨워야 하는 스레드 찾아 thread\_unblock().  
    - void thread\_aging() : 매 틱마다 호출되며 ready\_list의 모든 스레드의 priority 증가.  
    - void thread\_update\_load\_avg() : load\_avg 갱신 함수.  
    - void thread\_update\_recent\_cpu () : 모든 스레드에 대한 recent\_cpu 갱신 함수  
    - void thread\_update\_priority() : parameter 스레드에 대한 priority 갱신 함수.  
    - void thread\_update\_priority\_all() : 모든 스레드에 대한 priority 갱신 함수.  
    - bool wake\_ticks\_compare() : sleep\_list에 정렬 삽입시에 사용되는 compare 함수.  
    - bool priority\_compare() : ready\_list, ready\_queue, waiters list에 정렬 삽입시에 사용되는 compare 함수.  
    - void thread\_init() : BSD Scheduler를 위한 초기화 추가.  
    - void thread\_tick() : BSD Scheduler를 위한 priority update 내용 추가. Aging flag에 대한 내용 추가.  
    - tid\_t thread\_create() : 새로운 스레드와 현재 스레드의 priority 비교 후 선점하는 기능 추가.  
    - void thread\_unblock() : Priority Scheduler의 경우 ready\_list 삽입 정책 변경, BSD Scheduler의 경우 priority에 따라 나누어진 list에 삽입.  
    - void thread\_yield() : thread\_unblock()과 동일한 논리 적용.  
    - void thread\_set\_priority() : 현재 스레드의 priority를 new\_priority로 변경하고 ready\_list와 비교하여 선점 여부 결정.  
    - int thread\_get\_priority() : 현재 스레드의 priority 반환.  
    - void thread\_set\_nice() : 현재 스레드의 nice값 변경, priority 갱신 후 ready\_queue와 비교하여 선점 여부 결정.  
    - int thread\_get\_nice() : 현재 스레드의 nice값 반환.  
    - int thread\_get\_load\_avg() : load\_avg \* 100 반환.  
    - int thread\_get\_recent\_cpu() : 현재 스레드의 recent\_cpu \* 100 반환.  
    - void init\_thread() : Priority Scheduler의 경우 priority argument 그대로 적용. BSD Scheduler의 경우 이를 무시하고 nice와 recent\_cpu를 기반으로 priority 계산.  
    - struct thread \*next\_thread\_to\_run() : Priority Scheduler의 경우 ready\_list의 front 스레드 반환. BSD Scheduler의 경우 비어있지 않은 가장 높은 list의 front 스레드 반환. 공통적으로 반환할 스레드가 없다면 idle\_thread 반환.  
      
    2. threads/synch.c  
    - void sema\_down() : semaphore의 waiters list에 priority에 대한 내림차순으로 정렬 삽입.  
    - void sema\_up() : semaphore의 waiters list에서 가장 priority가 높은 스레드를 unblock 한 후 현재 스레드의 priority보다 waiter의 priority가 더 높으면 yield  
      
    3. devices/timer.c  
    - void timer\_sleep() : 현재 스레드에 대해서 thread\_sleep() 호출하고 thread\_block() 호출하여 스레드 상태을 BLOCKED로 변경.  
      
    4. threads/fixed.h  
    - fixed\_t int\_to\_fixed() : 정수형을 (고정소수점) 실수형으로 변환.  
    - int fixed\_to\_int\_trunc() : 실수형을 정수형(소수점 버림)으로 변환.  
    - int fixed\_to\_int\_near() : 실수형을 정수형(반올림)으로 변환.  
    - fixed\_t fixed\_add() : 두 실수형에 대한 덧셈 연산  
    - fixed\_t fixed\_sub() : 두 실수형에 대한 뺄셈 연산  
    - fixed\_t fixed\_add\_int() : 실수형과 정수형에 대한 덧셈 연산  
    - fixed\_t fixed\_sub\_int() : 실수형과 정수형에 대한 뺄셈 연산  
    - fixed\_t fixed\_mult() : 두 실수형에 대한 곱셈 연산  
    - fixed\_t fixed\_mult\_int() : 실수형과 정수형에 대한 나눗셈 연산  
    - fixed\_t fixed\_div() : 두 실수형에 대한 나눗셈 연산  
    - fixed\_t fixed\_div\_int() : 실수형과 정수형에 대한 나눗셈 연산

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

1. Alarm Clock  
   [threads/thread.h]  
     
   새로운 요소 추가. (wake\_ticks)  
     
   [threads/thread.c]  
   struct list sleep\_list : Pintos에 내장된 자료구조인 list 이용.  
     
   (1) void thread\_sleep()  
     
   wake\_ticks를 저장하고 이에 대한 오름차순으로 sleep\_list에 삽입.  
     
   (2) void thread\_wake\_up()  
     
   sleep\_list에서 깨워야 하는 스레드가 있다면 리스트에서 제거하고 thread\_unblock() 호출.  
     
   (3) bool wake\_ticks\_compare()  
     
   sleep\_list를 정렬하는 데에 사용하는 compare 함수.  
     
   (4) void thread\_tick()  
     
   thread\_wake\_up() 함수 호출하는 코드 추가.  
     
   [devices/timer.c]  
   (1) void timer\_sleep()  
     
   thread\_sleep() 호출하고 thread\_block() 호출하도록 코드 구현.
2. Priority Scheduling  
   [threads/thread.c]  
   (1) void thread\_unblock()   
     
   ready\_list에 priority에 대한 내림차순 정렬 상태로 삽입하도록 구현 변경.  
     
   (2) void thread\_yield()  
     
   thread\_unblock과 마찬가지로 ready\_list에 priority에 대한 내림차순 정렬 상태로 삽입하도록 구현 변경.  
     
   (3) bool priority\_compare()  
     
   ready\_list 및 semaphore의 waiters 리스트 삽입 시에 사용되는 compare 함수.  
     
   (4) tid\_t thread\_create()  
     
   생성한 스레드의 priority가 현재 스레드의 priority보다 높다면 thread\_yield() 호출하여 preemption 구현.  
     
   (5) void thread\_set\_priority()  
     
   현재 스레드의 priority 변경한 후 preemption 구현.  
     
   (6) int thread\_get\_priority()  
     
   현재 스레드의 priority 반환.  
     
   (7) void thread\_aging()  
     
   ready\_list를 순회하며 READY 상태인 스레드에 대해 priority 1씩 증가.  
     
   [threads/synch.c]  
   (1) void sema\_down()  
     
   semaphore value가 0이어서 기다려야 하는 경우 waiters 리스트에 priority에 대한 내림차순 정렬로 현재 스레드 삽입. thread\_block() 호출.  
     
   (2) void sema\_up()  
     
   waiters 리스트에서 가장 priority 높은 스레드 unblock 후 현재 스레드와 비교하여 preemption 구현.
3. Advanced Scheduler  
   [threads/thread.h]  
     
   새로운 요소 추가. (nice, recent\_cpu)  
     
   [threads/thread.c]  
   (1) Ready queues  
     
   64개 (PRI\_MAX + 1) 의 queue 선언. Index = Priority  
     
   (2) void thread\_update\_load\_avg()  
     
   ready\_threads 계산 후 load\_avg 계산식에 따라 fixed-point 연산 후 저장.  
     
   (3) thread\_update\_recent\_cpu()  
     
   idle thread 제외한 모든 all\_list에 있는 스레드에 대해 recent\_cpu 계산하여 갱신.  
     
   (4) void thread\_update\_priority()  
     
   스레드 t에 대해 priority 계산하여 갱신. 반드시 PRI\_MIN ~ PRI\_MAX 구간 내의 값이어야 함.  
     
   (5) void thread\_update\_priority\_all()  
     
   idle thread를 제외한 모든 스레드에 대해 priority 갱신. 만약 READY 상태인 스레드의 priority가 바뀌었다면 새로운 priority에 맞는 큐에 삽입. (정렬 X, 개별 큐에 대해선 RR이므로)  
   마지막으로 현재 스레드의 priority도 변경되므로 더 큰 priority의 스레드가 있다면 cpu 양보하여 preemption 구현.  
     
   (6) void thread\_init()  
     
   ready\_queue 초기화. Load\_avg 초기화. Initial\_thread에 대한 nice와 recent\_cpu 초기화.  
     
   (7) void thread\_tick()  
     
   mlfqs에 대한 동작 구현.  
   현재 스레드의 recent\_cpu에 1 더함.  
   1초마다 모든 스레드의 recent\_cpu 갱신.  
   40ms마다 모든 스레드의 priority 갱신.  
     
   (8) void thread\_unblock()  
     
   mlfqs에 대한 구현 추가. Priority에 맞는 list에 삽입 (정렬 X).  
     
   (9) void thread\_yield()  
     
   mlfqs에 대한 구현 추가. Priority에 맞는 list에 삽입 (정렬 X).  
     
   (10) void thread\_set\_nice()  
     
   현재 스레드의 nice값 변경. 이후 priority 갱신한 후 preemption 구현 추가.  
     
   (11) int thread\_get\_nice(), int thread\_get\_load\_avg(), int thread\_get\_recent\_cpu()  
     
   nice, load\_avg, recent\_cpu에 대한 getter 함수 구현.  
     
   (12) void init\_thread()  
     
   mlfqs에 대한 구현 추가. 인자로 받아온 priority 무시하고 nice와 recent\_cpu 기반으로 초기 priority 설정.  
     
   (13) struct thread \*next\_thread\_to\_run()  
     
   mlfqs에 대한 구현 추가. 비어있지 않은 가장 높은 priority에 대한 list에서 스레드 가져옴.
   1. **시험 및 평가 내용**

* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석  
    
  테스트를 호출하는 스레드의 priority를 PRI\_DEFAULT + THREAD\_CNT + 1로 설정하여 가장 높은 우선순위를 받을 수 있도록 한다. 그리고 THREAD\_CNT 만큼 새로운 스레드를 생성하고 각 스레드에 대해 PRI\_DEFAULT + 1 + i의 priority를 설정한다. 반복문을 보면 priority가 낮은 스레드부터 생성되는 것을 확인할 수 있는데, 이 순서로 ready\_list에 삽입되더라도 priority scheduling이 정상적으로 구현되었다면 priority가 높은 스레드부터 실행될 것이다. 즉, 생성 순서의 역순으로 스케줄링을 받게 될 것이다.  
    
    
  각 스레드는 위 함수의 동작을 수행한다. Output buffer에 자신의 id를 기록하고 CPU를 양보한다.   
    
  실행 결과는 다음과 같다.  
    
  priority가 높은 스레드부터 먼저 스케줄링 받는 것을 확인할 수 있다.
* make check 수행 결과를 캡처하여 첨부  
  