**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20161663 허재성

개발 기간 : 2021.10.15 ~ 2021.11.06

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

Project 1, 2까지는 Pintos에서 기본적으로 round-robin 정책을 이용한 scheduler를 이용하여 thread의 우선순위를 고려한 scheduling이 되지 못했다. 다시 말하면 우선 순위가 높은 thread가 먼저 처리되지 못하였다. 이 점을 개선하기 위해 우선 순위가 높은 thread가 먼저 처리될 수 있도록 priority-scheduling을 구현한다. Priority-scheduling을 구현하기 위해 기존에 busy-waiting으로 구현된 alarm clock을 더 효율적으로 수정하고 priority-scheduling의 ready-queue에 삽입되는 정책을 개선한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

Thread를 scheduling하기 위해선 일정 시간 동안 CPU를 양보하고 thread를 sleep 상태로 바꾸는 timer\_sleep 함수와 sleep 상태의 thread 중에서 충분한 시간만큼 sleep 상태에 있던 thread를 깨우는 기능이 필요하다. 기존의 timer\_sleep 함수는 무한 루프를 돌면서 thread가 해당 시간이 지났는지 확인하는 Busy waiting 방식으로 구현되어 있어 CPU 자원을 낭비하게 되어 비효율적이었다. 이를 개선하여 그런 비효율을 최소화한다.

Sleep 상태의 thread가 다시 깨어날 수 있도록 깨워주는 기능도 구현해야 한다. 이를 구현하면 timer\_sleep에 의해 blocked 상태가 된 thread가 다시 ready 상태로 돌아갈 수 있다.

* 1. Priority Scheduling

이전에는 Round-Robin 방법으로 thread가 우선순위에 상관없이 ready-queue에 들어온 순서대로 실행되었다면 우선 순위를 고려하여 ready-queue에 thread가 들어갈 때 가장 우선순위가 높은 thread가 ready-queue의 가장 앞에 위치하도록, 따라서 ready-queue에서 가장 우선 순위가 높은 thread가 먼저 실행 될 수 있도록 한다. 결과적으로 우선 순위가 높은 thread가 먼저 실행 될 수 있도록 한다.

한편 위의 방법대로 우선 순위가 높은 thread가 ready-queue의 앞에 위치할 경우 우선 순위가 낮은 thread는 계속 queue의 뒤에 위치하여 실행되는데 너무 오랜 시간이 걸리거나 더 심각할 경우 실행되지 못하는 startvation 문제가 발생할 수 있다. 이를 방지하기 위해 특정 시간이 지날 경우 ready-queue에 있는 thread들의 우선 순위를 증가시켜서 계속 우선 순위가 낮아 실행 대상에서 배제되는 starvation 문제를 방지한다. 이를 aging이라고 한다.

* 1. Advanced Scheduler (추가구현을 한 경우)
  2. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

timer\_sleep 함수에 의해 sleep 상태로 blocked된 thread들 중 sleep할 시간이 지난 thread들을 깨워야 한다. 우선 timer\_sleep 함수에 의해 sleep된 blocked 상태의 thread들을 sleep\_queue라는 선형 리스트에 저장한다. Blocked 상태의 thread가 깨울 시간이 되었는지 주기적으로 하기 위해 timer interrupt때마다 확인한다. 매번 timer interrupt때마다 sleep\_queue에 저장된 thread들을 확인하여 깨울 시간이 된(지정된 시간만큼 sleep 상태였던) thread들을 깨운다. 깨우는 것을 thread의 상태를 blocked에서 ready 상태로 변환하고 ready\_queue에 삽입하는 것으로 구현한다.

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

기존의 round-robin policy에 의하면 thread가 ready list에 들어갈 때 thread의 priority에 관계 없이 ready list의 뒤에 삽입되어 thread가 scheduling될 때에는 ready list의 가장 앞의 thread가 먼저 scheduling되므로 최근에 ready list에 삽입된 thread는 thread의 priority에 관계없이 가장 나중에 실행되었다. 이를 개선하기 위해선 thread가 ready list에 삽입될 때 priority를 고려하여 priority가 높을 경우 ready list의 가장 앞에 삽입될 수 있도록 해야 한다. List에 삽입될 때 단순 앞, 뒤 삽입이 아닌 정렬 기준에 의한 삽입이 이루어져야 하므로 정렬 기준을 마련해야 한다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)
2. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2021.10.15 ~ 2021.11.06

2021.10.15 ~ 2021.10.25 중간고사 기간

2021.10.25 ~ 2021.11.01 프로젝트 요구사항, Pintos Manual 분석, 코드 분석

2021.11.02 : Alarm Clock 구현

2021.11.03~ 2021.11.04 Priority Scheduling 구현

* 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

**추가 자료 구조**

각 thread에 대해서 thread를 sleep 상태에서 깨울 시간을 기록해 놔야 한다. 따라서 threads/thread.h의 struct thread에 wake\_up 변수를 추가한다. 또한 기존의 timer\_sleep에서 busy waiting으로 비효율적으로 thread를 sleep 시키는 방법을 개선하기 위해 devices/timer.c에 선형 리스트 sleep\_queue를 추가하여 thread가 sleep 상태가 될 경우 sleep\_queue에 저장될 수 있도록 한다.

**수정해야 하는 소스코드, 추가해야 하는 함수**

devices/timer.c에서 sleep\_queue를 정상적으로 사용할 수 있도록 timer\_init 함수에 sleep\_queue를 초기화시키는 코드를 추가해야 한다. devices/timer.c의 timer\_sleep 함수와 timer\_interrupt 함수를 수정해야 한다. timer\_sleep에서는 현재 thread의 깨울 시간 wakeup을 설정 후 sleep\_\_queue에 삽입하고, timer\_interrupt에서는 wake\_up 함수를 추가해 wake\_up 함수에서 sleep\_queue에 저장된 thread를 모두 확인해 wakeup이 깨울 시간이 되었을 경우 깨우도록 한다.

Aging 기법을 사용하기 위해 명세서에서 요구하는 대로 threads/thread.h, threads/init.c, threads/thread.c에 코드를 추가했다.

Priority-scheduling을 구현하기 위해 thread\_unblock 함수와, thread\_yield 함수에서 thread가 ready\_list로 들어갈 때 우선순위에 따라 ready\_list에 삽입되는 위치가 결정되도록 하였다. 이를 위해 ready\_list에 삽입될 때 thread의 priority를 비교하는 비교 함수 priority\_comp\_dec 함수를 추가해 ready\_list가 priority에 대해 내림차순으로 정렬된 상태가 되도록 한다. priority\_comp\_dec 함수를 이용해 list\_insert\_ordered 함수를 호출해 thread가 priority로 내림차순 정렬된 채 저장될 수 있다.

Thread가 새로 생성될 경우 새로 생성된 thread가 현재 실행중인 thread보다 우선 순위가 높으면 새로 생성된 thread가 실행되는 것이 옳다. 따라서 thread\_create 함수에 현재 실행중인 thread와 새로 생성된 thread의 우선순위를 비교 후 필요 시 thread\_yield를 호출하는 코드를 추가한다.

비슷한 이유로 현재 thread의 우선순위가 변경될 경우 변경된 우선순위가 ready\_list의 가장 앞에 저장된 ready\_list에 저장된 thread 중 우선 순위가 가장 높은 thread와 비교하여 더 낮아질 수도 있다. 이 경우에도 rescheduling이 필요하다. 따라서 thread\_set\_priority 함수에 현재 thread와 ready\_list의 가장 앞의 thread의 우선 순위를 비교하여 필요할 경우 thread\_yield 함수를 호출하는 코드를 추가한다.

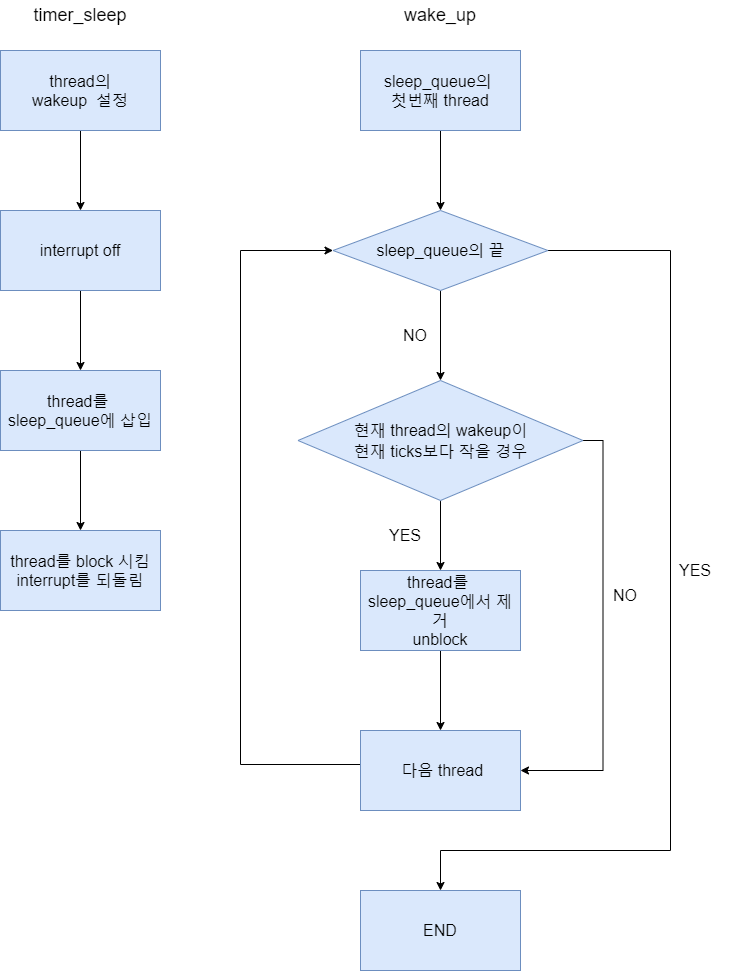
Starvation 문제를 해결하기 위해 thread\_aging 함수를 추가한다. 매번 tick마다 호출되어 ready\_list를 순회해 우선순위를 1씩 증가시킨다. 우선순위 최대값 PRI\_MAX 이하의 값만 1 증가시키며 이로 인해 우선순위가 낮은 thread가 계속 실행되지 못하는 starvation을 방지한다.

기존 semaphore에서 sema\_up 함수의 경우 semaphore의 waiters 리스트에 저장된 thread들이 우선순위에 관계없이 가장 앞에서부터 unblock 되었다. 이를 보완해 우선순위를 고려해 waiters 리스트의 thread 중 가장 우선순위가 높은 thread가 먼저 waiters 리스트에서 제거되고 unblock 될 수 있도록 코드를 수정한다.

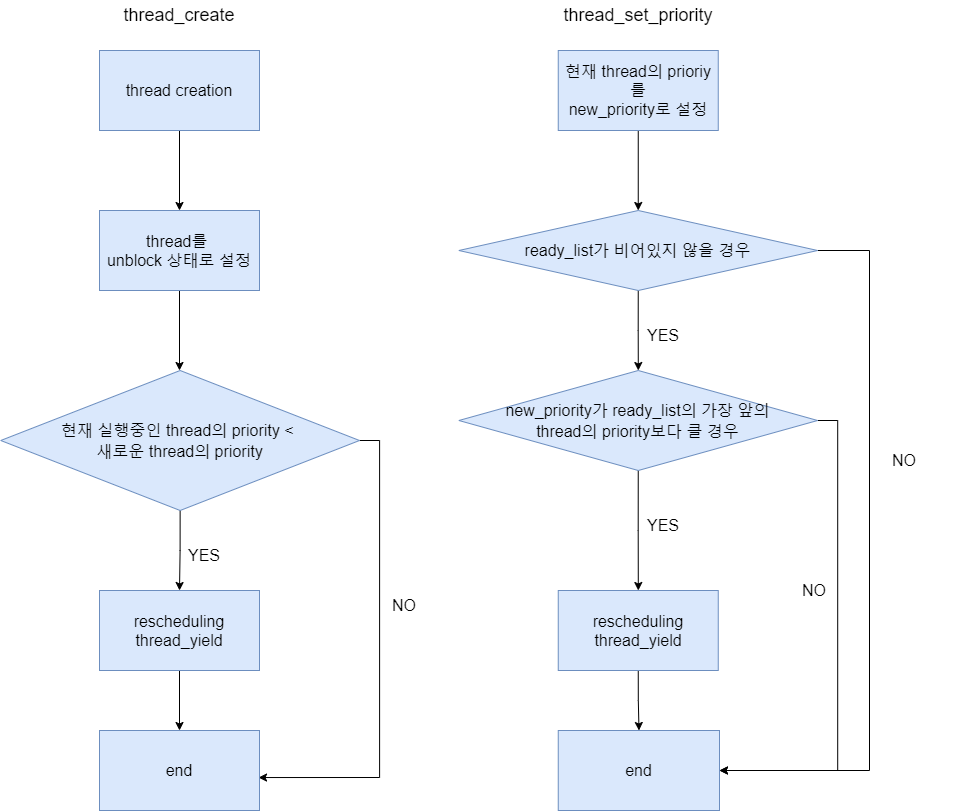
1. **연구 결과**
   1. **Flow Chart**

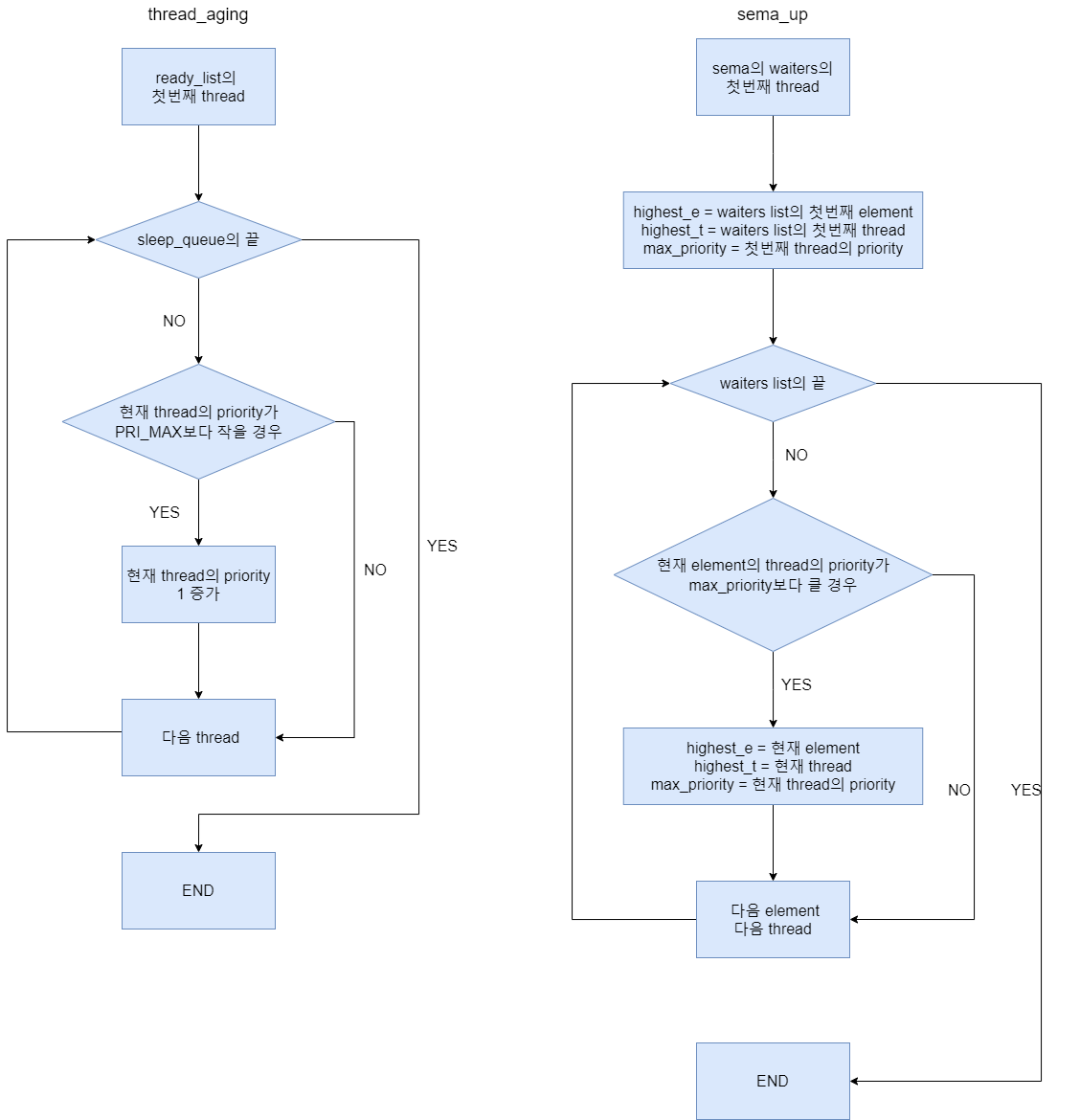
* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)

**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

devices/timer.c에서 timer\_sleep이 thread를 sleep 상태로 바꾸기 위해 while문으로 loop를 돌며 thread\_yield 함수를 호출하는 busy waiting 방법을 수정하여 같은 파일에 추가한 sleep\_queue에 현재 thread를 깨울 시간 wakeup을 설정 후 삽입하는 것으로 코드를 변경하였다. 이 때 sleep\_queue에 저장될 때에는 단순히 저장만 하면 되므로 lib/kernel/list.c의 list\_push\_back 함수를 이용해 sleep\_queue의 가장 뒤에 삽입되도록 하였다. Sleep\_queue에 저장된 thread 중 시간이 지남에 따라 깨울 thread가 생길 텐데 매번 timer\_interrupt때마다 깨울 thread를 찾을 수 있도록 timer\_interrupt 함수에서 새로 추가한 wake\_up 함수를 호출한다. wake\_up 함수에서는 sleep\_queue를 앞에서부터 끝까지 차례로 탐색하며 wakeup이 지난 thread들을 list\_remove 함수로 sleep\_queue에서 제거하고 thread\_unblock 함수로 ready 상태로 변경한다.

threads/thread.c의 thread\_yield, thread\_unblock 함수에서 현재 thread가 ready\_list로 삽입될 때 round robin 방식이 아닌 priority를 고려한 scheduling이 필요하므로 lib/kernel/list.c의 list\_insert\_ordered 함수를 사용한다. 이 때 list\_insert\_ordered의 인자로 정렬 기준이 될 함수의 포인터가 필요한데 이를 위에서 언급한 priority\_comp\_dec를 전달하여 ready\_list에 thread가 우선순위에 대하여 내림차순으로 저장될 수 있도록 한다.

새로운 thread가 생성되는 thread\_create 함수에서는 현재 실행 중인 thread의 우선순위인 thread\_current()->priority와 새로운 thread의 우선순위 priority를 비교하여 새로운 thread의 우선순위가 높다면 새로 생성한 thread가 바로 실행될 수 있어야 한다. 이를 위해 thread\_yield 함수를 호출해 rescheduling한다.

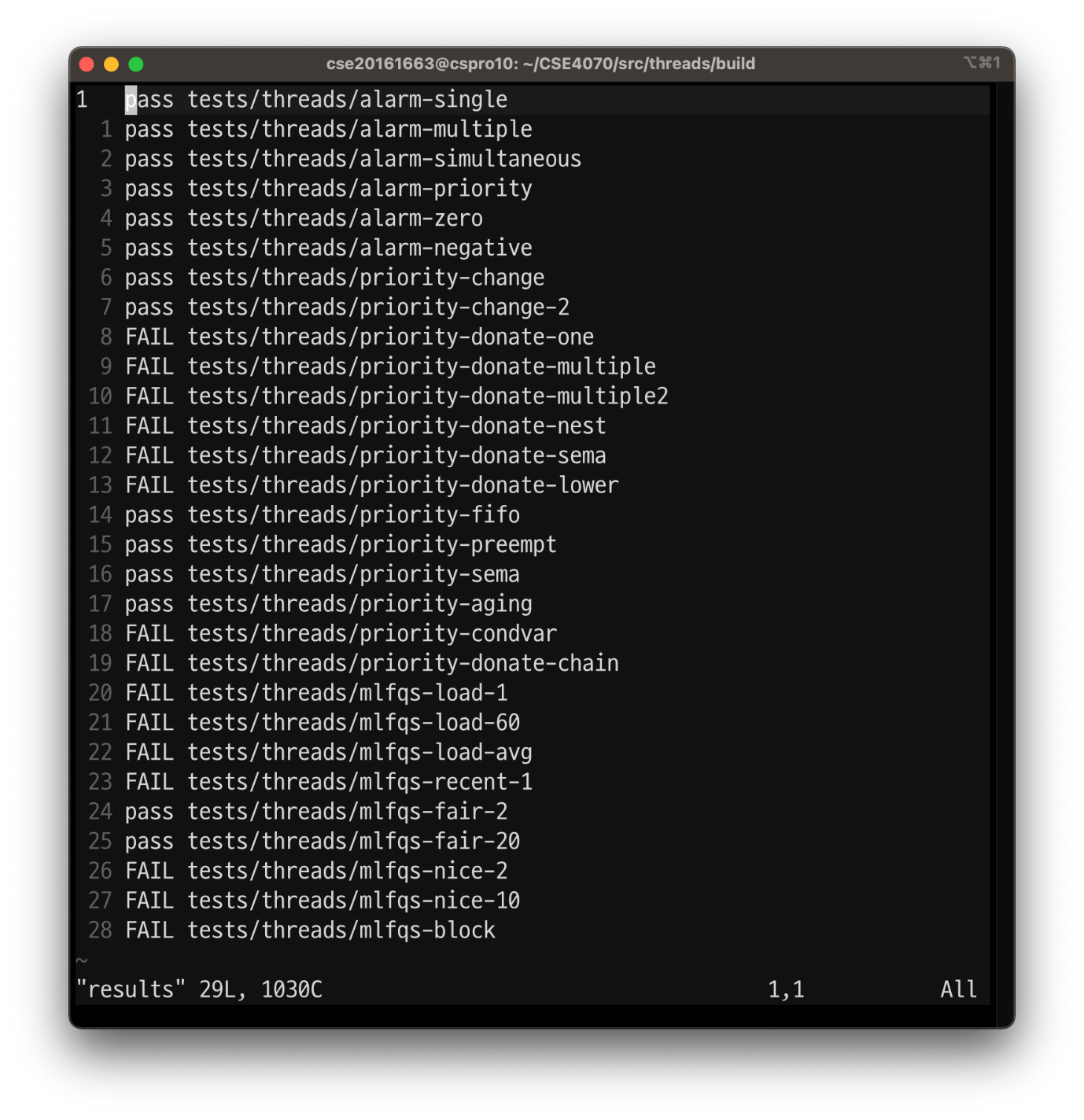
비슷하게 현재 thread가 new\_priority로 우선순위가 바뀔 경우 new\_priority가 이전 우선순위보다 더 높을 경우 상관 없지만 더 낮아질 경우 ready\_list에 존재하는 thread 중 가장 우선 순위가 높은 가장 앞의 thread와 우선순위를 비교해야 한다. 만약 가장 앞의 thread가 new\_priority보다 더 높은 우선순위를 가질 경우 더 먼저 실행되어야 하므로 rescheduling을 위해 thread\_yield 함수를 호출한다.

Starvation을 방지하기 위해 thread\_aging 함수를 추가한다. 매번 thread\_tick마다 호출되어 ready\_list에 존재하는 모든 thread의 우선순위가 PRI\_MAX보다 작을 경우 1씩 증가시킨다.

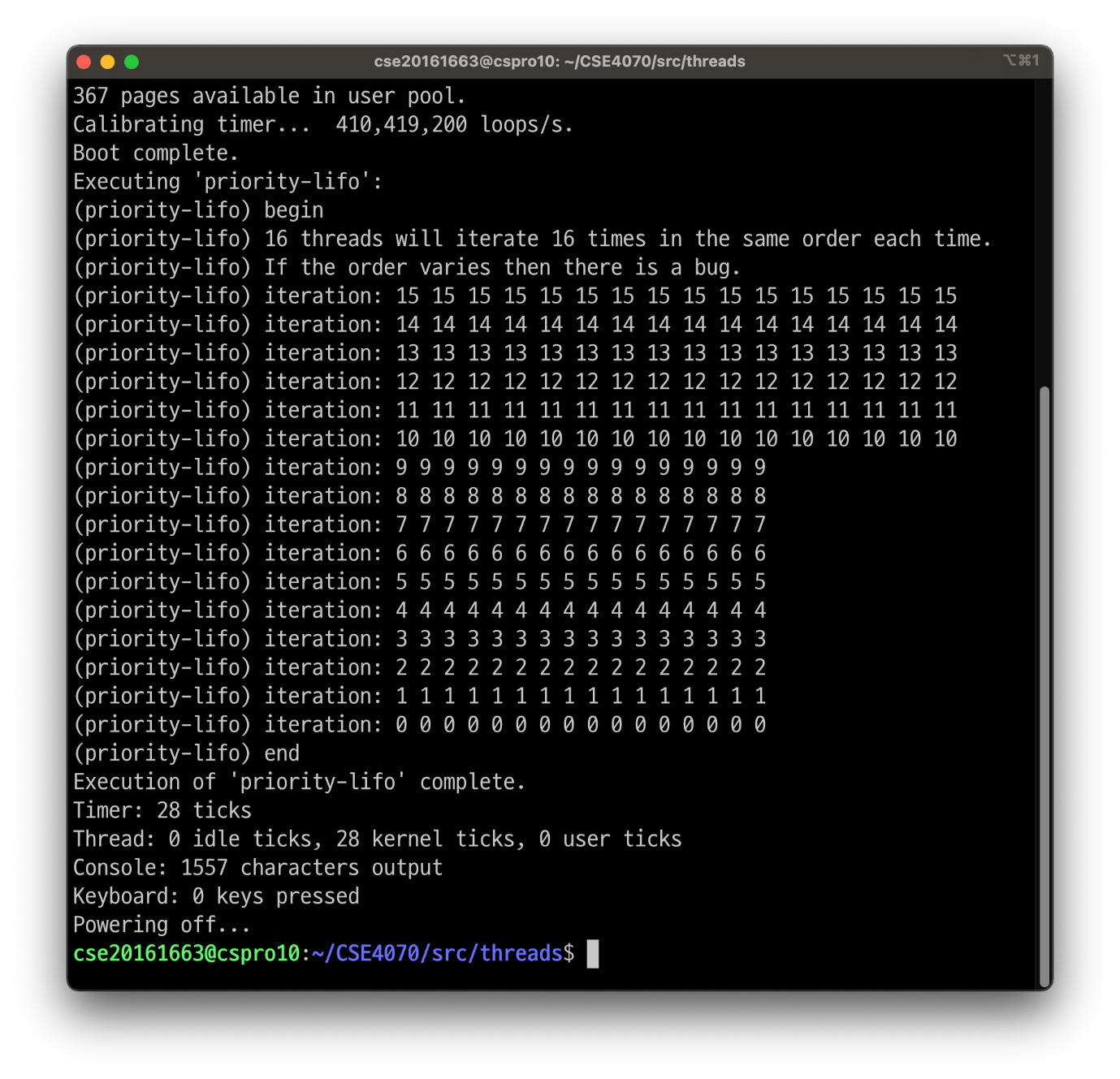
Semaphore와 priority scheduling이 결합된 경우 문제가 없도록 sema\_up 함수를 수정한다. 기존 sema\_up 함수에서는 sema->waiters에서 우선 순위에 관계없이 가장 앞에 존재하는 thread를 찾아 제거 후 unblock 시켰는데 priority를 고려하여 waiters list를 모두 순회하여 우선 순위가 최대인 thread를 찾고 해당 thread를 unblock시킨 뒤 waiters에서 제거하여 가장 우선순위가 높은 thread가 lock을 얻을 수 있도록 구현하였다.

* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석
* make check 수행 결과를 캡처하여 첨부

make check 결과 (추가 구현 미구현)



priority-lifo 테스트 결과



priority-lifo.c

16개의 thread가 16번씩 반복해서 실행된다. 이 때 16개의 thread들은 0번째 thread부터 15번째 thread까지 id는 i, priority가 PRI\_DEFAULT + 1 + i(0 <= i <=15)로 설정된다. 즉 나중에 생성되는 thread가 더 높은 우선순위를 가진다. 그리고 scheduling을 한다. 나중에 생성된 thread가 더 우선순위가 높으므로 더 먼저 실행되고 가장 먼저 생성된 thread가 우선순위가 가장 낮으므로 나중에 실행되므로 위의 결과는 적절하다. Thread가 반복되어 실행될 때마다 thread의 id를 출력한다. 가장 나중에 생성된 thread의 id는 15이고 가장 처음 생성된 thread의 id는 0이다.