**Pintos Project 3: Threads**

담당 교수 : 이혁준

조 / 조원 : 고성빈

개발 기간 : 20/11/14 ~20/12/08

1. **개발 목표**

* Alarm clock을 수정해 cpu power을 덜 소비하도록 한다.
* round-robin 방식으로 관리되던 pintos를 priority scheduling으로 관리되도록 수정한다.
* MLFQ를 통한 Advanced Scheduler를 구현한다. 이 과정에서 스케쥴러가 thread nice value를 확인해 priority에 반영하고, recent\_cpu와 load\_avg 수치를 관리할 수 있도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Alarm Clock

* 기존의 timer\_sleep 함수는 즉시 thread\_yield 함수를 호출해 타이머가 종료되기 전까지 비효율적으로 대기하게 된다. 따라서 새로운 ready queue를 만들고 해당 queue에 blocked된 thread를 추가해주는 식으로 관리하게 되면 더 효율적으로 대기중인 thread를 관리할 수 있게 된다.

1. Priority Scheduling

* 기존의 pintos는 간단한 Round-robin 방식으로 스케쥴링 된다. 하지만 round robin 방식으로는 높은 priority를 가진 thread를 우선적으로 수행할 수 없기 때문에 이를 처리하기 위해서는 priority scheduling을 구현해주어야 한다.

1. Advanced Scheduler (BSD Scheduling)

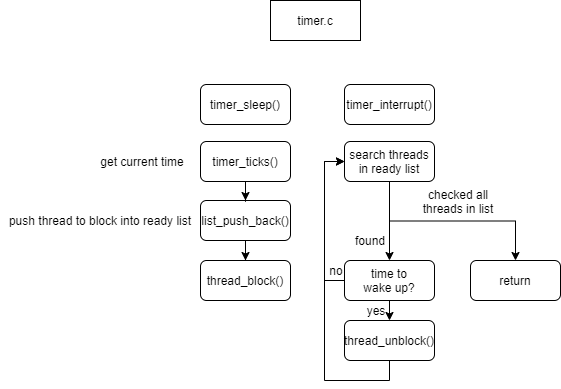
* BSD scheduler는 Multi-Level Feedback Queue를 통해 구현하게 된다. queue는 64개의 별개의 queue를 작성하거나, 하나의 queue 상에 구현할 수 있다.
* 또한 각 thread의 nice value를 확인해 priority에 반영할 수 있도록 하고, recent\_cpu와 load\_avg 수치를 관리할 수 있도록 한다.
* pintos에서는 floating point를 지원하지 않으므로 fixed point 방식으로 실수 데이터 구조를 직접 구현해주어야 한다.
  1. **개발 내용**

1. Blocked 상태의 스레드를 깨우기 위해서는 새로운 queue를 만들어서 block 상태의 thread를 모두 넣어 관리해주어야 한다. queue에는 block된 스레드의 정보와 함께 wake up time을 저장해주어 언제 thread를 깨워야 하는지도 알수 있어야 한다. timer\_interrupt를 통해 해당 시간에 도달하면 blocked queue로부터 스레드를 빼 ready list에 넣어주도록 한다.
2. priority scheduling을 사용할 경우 Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 현재 실행중인 thread를 즉시 종료한 뒤, higher priority를 가진 thread를 실행하도록 프로그램을 수정해야 한다.
3. BSD scheduler에서 priority 계산에 필요한 요소는, recent\_cpu(thread의 CPU time)와 thread의 nice value값이다. 이 값의 측정을 위해서 각 thread의 nice value 정보를 읽어 처리해주어야 하고, recent\_cpu의 값을 계산해주어야 한다. recent\_cpu 값을 계산하기 위해서는 실수값에 대한 연산을 해주어야 하는데 pintos에서는 부동소수점 연산을 지원하지 않으므로 고정소수점 연산으로 직접 값을 계산해주어야 한다.
4. **추진 일정 및 개발 방법**
   1. **추진 일정**

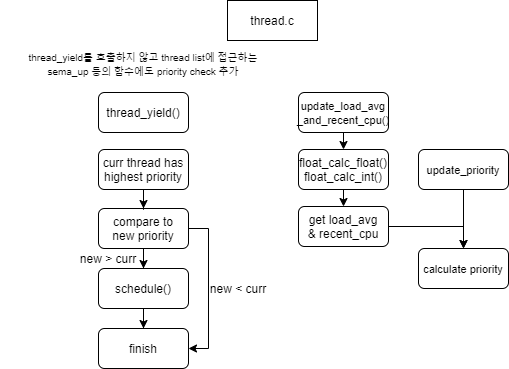
* 프로젝트에 구현할 내용이 많지 않고, 이미 모두 배운 내용에 대한 프로젝트이기 때문에 빠르게 프로젝트를 시작할 수 있었다. 다만 기존 프로젝트에서 사용한 파일과는 다른 파일에 대해 코드를 작성했기 때문에 새로운 구졸ㄹ 이해하기 위해서 11월 19일에는 코드를 읽어보았다.
* 이어서 11월 20~21일에 걸쳐 프로젝트 영상을 보며 구현을 시작했고, priority scheduling까지의 구현을 마쳤다.
* 이후 11월 28~29일에 걸쳐 BSD scheduling의 추가 구현을 마친 뒤 12월 5일에 보고서 작성을 마쳤다.
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정이 필요한 부분
    - 기존의 alarm clock을 수정하기 위해서 timer\_sleep 함수를 수정해주어야 한다. 이를 위해서 timer\_sleep 함수의 busy waiting 코드를 수정해 interrupt에 의해 blocked queue 상의 데이터를 출력할 수 있도록 만들어야 한다.
    - 기존 round-robin 방식으로 ready list의 맨 마지막에 thread가 추가되는데, 이를 priority 기반으로 다시 스케쥴링 해주는 thread\_yield 함수를 수정해주어야 한다. 또한 semaphore를 조정해주는 sema\_up 함수에도 priority를 적용해야 한다.
    - priority에 nice value와 recent\_cpu가 적용되도록 연산을 수정해주어야 한다.
    - 기존 ready list를 multilevel freedback queue의 형태로 구현해주어야 한다.
  + 추가해야 하는 부분
* alarm clock을 수정하는 과정에서 thread 내부에 wakeup time 변수를 추가해주어야 한다.
* priority aging을 구현하기 위해서 실수 연산을 해주어야 하는데 이를 위한 고정소수점 데이터와 실수에 대한 연산을 추가해주어야 한다.

1. **연구 결과**
   1. **Flow Chart**

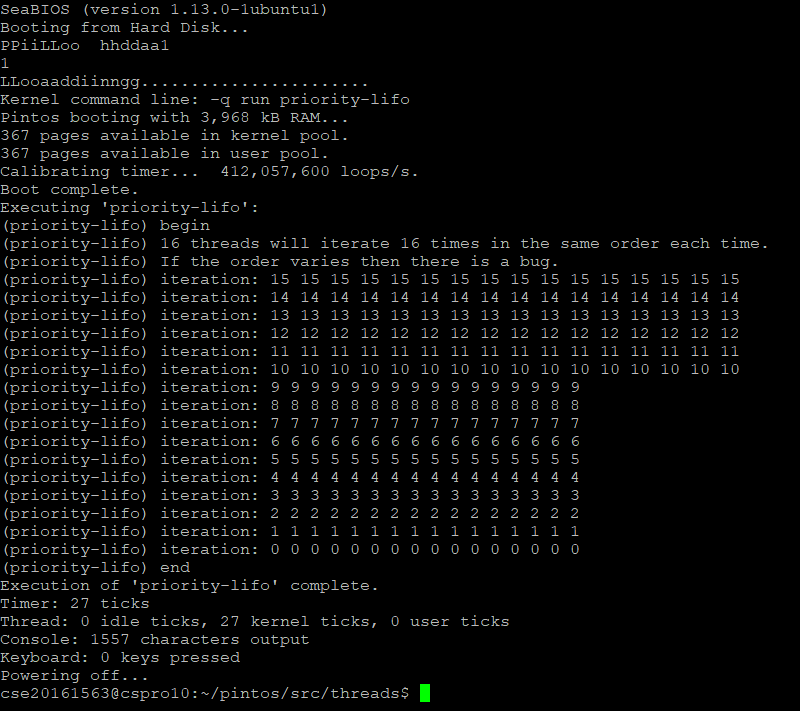
* Alarm Clock



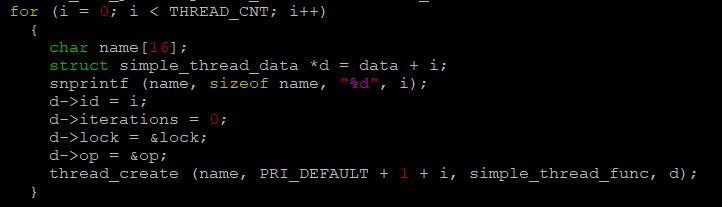
* Priority Scheduling & aging



* 1. **제작 내용**
* alarm clock의 수정을 위해서는 device 파일의 timer.h와 timer.c 파일의 함수를 수정해주었다. timer\_sleep() 함수는 기존 while 반복문을 사용해 busy waiting을 하고 있어 cpu power를 소모하고 있으므로, 이를 수정해주었다. threads.c 파일의 struct thread에 wakeup\_time이라는 변수를 추가해주었고, timer\_sleep 함수에서는 이 wakup\_time 변수의 값을 지정해준 뒤 blocked list에 저장해준다. 이렇게 저장된 thread는 종료되고 ready queue에 들어가지 않고 blocked list에서 꺼내주기 전까지 대기한다.
* timer\_interrupt는 매 time clock마다 호출되는 함수로, 이 함수 내에서는 blocked list의 모든 thread를 확인해가며 block time이 지난 list를 unblock해 준다. 이 과정에서 unblock된 thread는 다시 scheduler에 의해 스케줄되어 CPU에서 실행될 수 있게 된다.
* thready\_yield 함수는 기존 ready\_queue의 맨 마지막에 thread를 추가해주는 방식으로 작동하고 있었지만, priority scheduling을 통해 list에 priority를 비교해가며 추가해주는 방식으로 작동해야 한다. thread\_yield에 접근하지 않고 ready queue에 접근하는 sema\_up 등의 함수에도 priority scheduling 방식에 맞추어 접근하도록 해 주어야 한다.
* aging을 구현하기 위해서는 실수 연산을 필요로 하는데, pintos에서는 실수 자료형과 부동소수점 연산을 지원해주지 않으므로, 고정소수점 연산을 수행해야 한다. 이를 위해 4byte로 저장된 integer 타입을 float처럼 사용하도록 하고 float간의 연산과 int-float 연산을 구현해주어야 한다.
* load\_avg와 recent\_cpu를 계산 공식에 따라 계산해주도록 한 뒤, 이를 통해 priority aging이 적용되는 scheduling을 수행할 수 있도록 해준다.
  1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

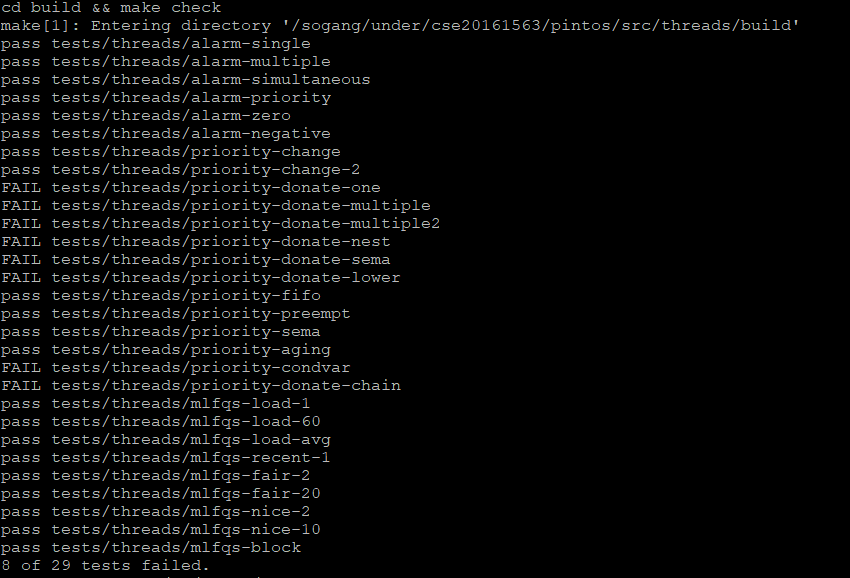


tests/thread/priority-lifo.c



priority-lifo.c 코드를 보면 차례대로 priority가 1씩 증가하는 thread를 생성한다. priority가 점차 증가하므로 last in first out 방식으로 작동해야 하고, 실행코드를 확인해 보면 그러한 방식으로 작동하는 것을 확인할 수 있다.

* make check 수행 결과



22개의 테스트 케이스 중 먼저 확인해 준 priority-lifo를 제외하고 21개의 나머지 테스트를 전부 통과한 것을 확인할 수 있다.