**Pintos Project 3 : Threads**

**(설계 프로젝트 수행 결과)**

과목 명 : [CSE4070] 운영체제 (1반)

담당 교수 : 박성용 교수님

조 / 조원 : 10조, 김예찬(20151529), 이수연(20161620)

개발 기간 : 2019/11/16 – 2019/12/8

**프로젝트 제목 : Pintos Project 3 Threads**

**제출일 : 2019년 12월 8일**

**참여 조원 : 김예찬(20151529), 이수연(20161620)**

1. **개발 목표**

* **기존 pintos 상태:** 기존 코드는 timer\_sleep()이 호출된 thread에 대해 while loop로 busy waiting을 수행하였다. Scheduling 정책에 관해서는 ready queue가 round robin 방식을 사용하였다.
* **추가할 기능:**

1. **Alarm clock:** Thread busy waiting으로 인해 발생하는 CPU waste를 막고자, thread block을 이용한 alarm clock을 구현해야 한다.
2. **Priority scheduling:** Priority scheduling 방식에 aging 기능을 추가해야한다. 또한 추가로 MLFQ(Multilevel Feedback Queue Scheduling)을 구현할 것이다.
3. **개발 범위 및 내용**
   1. **개발 범위**

* **Alarm clock:** 기존의 busy waiting 방식과 달리, thread를 block하면 해당 thread가 ready queue에 들어가지 않는다. 이를 구현하기 위하여 Pintos에서 제공하는 thread\_block(), thread\_unblock() 함수를 이용한다.
  + **수정되어야 할 내용:** Timer\_sleep()이 호출되면, 해당 thread를 thread\_block() 함수를 이용해 block 시켜놓는다. 설정해 둔 알람 시간이 다 된 경우, thread\_unblock()을 이용해 해당 thread를 unblock 해준다.
* **Priority Scheduling:** 각 thread 별로 우선 순위를 부여하여 priority scheduling을 구현한다. Priority를 계산하는 방법은 명세서에 제시된 식을 사용한다. 이때 시간에 따라 변화하는 변수(recent\_cpu)를 이용하여 aging이 가능하도록 한다. 이렇게 구현한 priority scheduler를 기반으로, 추가로 MLFQ scheduling을 구현한다.
  + **수정되어야 할 내용:** src/threads/thread.c의 thread\_yield() 함수는 thread rescheduling 시 호출하는 함수이다. 이 함수의 코드 내에, pintos에서 제공하는 list\_insert\_order()를 이용하여, ready queue 내의 thread들이 항상 우선 순위가 높은 순으로 정렬된 상태를 유지하게끔 한다. Thread 별 priority는 명세서에 제시된 식을 이용한다. (Priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2))
  1. **개발 내용**
* **Busy waiting vs Thread block:** Busy waiting은 thread가 스케줄링 될 조건을 계속해서 확인하기 때문에 CPU waste가 생긴다. 그러나 thread를 block하면 해당 thread는 ready queue에 들어가지 않고 밖에서 기다리게 된다(sleeping). 따라서 CPU를 점유하지 않게 되어 CPU waste가 발생하지 않는다.
* **Priority scheduling with aging:** 각 thread마다 우선 순위가 부여되고 그에 따라 scheduling 되는 것을 priority scheduling이라고 한다. 이러한 스케줄링 정책을 이용할 경우, prioirty가 낮은 thread는 무기한으로 기다려야하는(wait) 상황이 발생할 수 있다. 이를 방지하기 위하여 해당 thread의 우선 순위를 조금씩 높여주는 방법을 사용하는데, 이를 aging이라고 한다.
* **MLFQ(Multilevel Feedback Queue) Scheduling:** OS는 특정 process가 CPU bound process인지, I/O bound process인지에 따라 time quantum을 다르게 줄 필요가 있다. 따라서 초기 time quantum을 주고 빠르게 결과가 나오면 I/O bound process, 그렇지 않으면 CPU bound process로 판단하면 된다.

이를 구현하기 위하여 각각의 priority를 갖는 여러 개의 queue를 생성한다. Prioirty가 높은 queue일 수록 time quantum이 짧다. 새로운 process가 생성되면 우선 순위가 제일 높은(time quantum이 제일 짧은) queue에서 수행 시켜 해당 process의 time quantum이 얼마나 필요한지 확인한 뒤, 이에 따라 위치하는 queue를 바꿔간다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **11/18 ~ 24:** 코드 설계 및 기능 구현
* **12/3 ~ 8:** 프로그램 테스트 및 보고서 작성
  1. **개발 방법**
* **Alarm clock:** Timer\_sleep()이 호출되면, 해당 thread를 thread\_block() 함수를 이용해 block 시켜놓고, 글로벌 리스트 변수에 해당 thread를 저장한다. 현재 tick(Pintos의 시간 개념)이 임의의 thread의 alarmTime보다 크거나 같을 경우(알람 시간이 다 된 경우), thread\_unbloc() 함수를 이용해 이 thread를 unblock하고 sleepingThreadList에서도 정보를 삭제해준다.
* **Priority Scheduling:** Pintos에서 제공하는 list\_insert\_order()를 이용하여, ready queue 내의 thread들이 항상 우선 순위가 높은 순으로 정렬된 상태를 유지하게끔 한다. 이렇게 하면 ready queue의 첫번째 element는 항상 최고 priority의 thread가 된다. 그 후 Pintos에서 제공하는 schedule() 함수를 호출해 주면 prioirty scheduling이 구현된다.

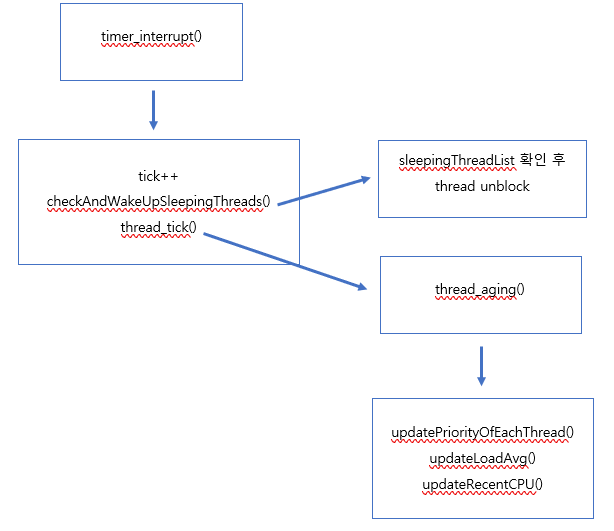
Thread 별 priority는 명세서에 제시된 식을 이용한다. (Priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2)) 이 때 recent\_cpu 변수는 시간에 따라 변화하는 변수로서, 이를 통해 priority aging을 구현할 수 있다. Recent\_cpu는 최근에 cpu를 사용한 정도를 나타내는 변수이다. 따라서 1 tick(Pintos의 시간 단위)이 지날 때마다 1씩 증가시켜준다. 또한 이는 1 second(=100 tick)마다 ready 상태의 평균 thread 개수를 반영하여 전체적으로 재계산한다.

Priority를 계산할 때 문제점은 pintos는 floating point를 지원하지 않는 다는 점이다. 계산 식에서 priority, nice는 정수 값이지만, recent\_cpu와 이를 계산하기 위해 사용하는 변수는 실수 값이다. 따라서 소수점 계산을 가능하게 하기 위한 별도 처리를 해준다.

MLFQ의 경우 이론대로라면 여러개의 queue를 가지고 구현해야 하지만, prioirty와 aging 기능을 사용하면 하나의 queue로도 기능을 구현할 수 있다. 따라서 mlfq scheduling을 사용하는 경우 기존 aging과 같은 작업을 수행하면 된다.

* 1. **연구원 역할 분담**
* **김예찬:** 코드 설계 및 기능 구현
* **이수연:** 보고서 작성

1. **연구 결과**
   1. **합성 내용**

****

* 1. **제작 내용**
* **Alarm clock:** Timer\_sleep() 함수에 parameter로 몇 tick 만큼의 시간동안 thread가 sleep 상태에 들어갈지 전달된다. 이 시간을 thread의 alarmTime 값으로 설정하고, 글로벌 리스트인 sleepingThreadList에 추가 한 뒤, 해당 thread를 block한다(thread\_block()).

Pintos 내에서 시간 개념은 timer\_interrupt() 함수가 호출되며 만들어진다. 이 함수 내에서 글로벌 변수의 tick의 값이 1씩 증가한다. 따라서 그 뒤에 sleep 상태인 thread들의 alarmTime을 확인하고 깨워주는 함수인 checkAndWakeUpSleepingThreads() 함수를 작성한다.

이 함수 안에서는 sleepingThreadList 내에 존재하는 thread 들의 정보를 조회한 뒤, 만약 현재 tick이 임의의 block된 thread의 alarmTime보다 크거나 같을 경우(알람 시간이 다 된 경우), 해당 thread를 unblock 해준다(thread\_unblock()).

Unblock 된 thread는 sleepingThreadsList에서도 제거하며, 해당 thread는 다시 ready queue로 들어가게 된다.

* **Priority scheduling:** Thread들이 re-scheduling 될 때 호출되는 함수는 thread\_yield()이다. 이 함수 내에 Pintos에서 제공하는 list\_insert\_order()를 이용하여, ready queue 내의 thread들이 항상 우선 순위가 높은 순으로 정렬된 상태를 유지하게끔 한다. 즉 ready queue의 첫번째 element는 항상 최고 priority의 thread가 된다. 그 후 Pintos에서 제공하는 schedule() 함수를 호출해 주면 prioirty scheduling이 구현된다.

언제나 최고 priority를 가진 thread가 running 되어야 하므로, 다음과 같은 시점에서 running 중인 모든 thread 들에 대한 prioirty 비교가 필요하다.

1. 새로운 thread가 생성될 때 (thread\_create())

2. 임의의 thread의 prioirty가 변경되었을 때 (thread\_set\_priority())

3. Thread aging이 가능하여 모든 thread들의 값이 변경되는 때 (thread\_aging())

위 경우일 때 현재 running중인 thread가, ready queue에 있는 모든 thread들과 비교했을 때 최고 priority를 가지는 지 비교해야 한다. 이를 구현하기 위해 ready queue에 존재하는 thread들 중 최고 prioirty 값을 반환해주는 함수를 구현하였다(getMaxPriorityFromReadyQueue()).

* **Aging:** Thread 별 priority는 명세서에 제시된 식을 이용한다. (Priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2)) 이 때 recent\_cpu 변수는 시간에 따라 변화하는 변수로서, 이를 통해 priority aging을 구현할 수 있다. Aging은 시간에 따라 계속 update 되는 값을 이용해야 하기 때문에, timer\_interrupt() 함수에서 호출되는 thread\_tick() 함수 내에서 thread\_aging()을 호출함으로써 구현하였다. Thread\_aging() 내에서는 아래와 같이 recent\_cpu 값을 update 해준다.

Recent\_cpu는 최근에 cpu를 사용한 정도를 나타내는 변수이다. 따라서 1 tick(Pintos의 시간 단위)이 지날 때마다 1씩 증가시켜준다. 또한 이는 1 second(=100 tick)마다 ready 상태의 평균 thread 개수(load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_threads)를 반영하여 전체적으로 재계산한다. (recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu +nice) 한편 nice 값은 임의의 상수로서 thread\_set\_nice() 내에서 값을 지정할 수 있다.

위와 같은 식을 이용하여 Priority를 계산할 때 문제점은, pintos는 floating point를 지원하지 않는 다는 점이다. 계산 식에서 priority, nice는 정수 값이지만, recent\_cpu, load\_avg는 실수 값이다. 따라서 소수점 계산을 가능하게 하기 위한 별도 처리를 해준다. 4바이트 내에서 가장 작은 14bit들이 소수점을 나타내기 때문에, 필요에 따라 정수 값에 2^14를 곱해주어 실수 값으로 변환하여 사용하거나, 실수 값에 2^14를 나눠주어 정수 값으로 변환한 뒤 사용한다.

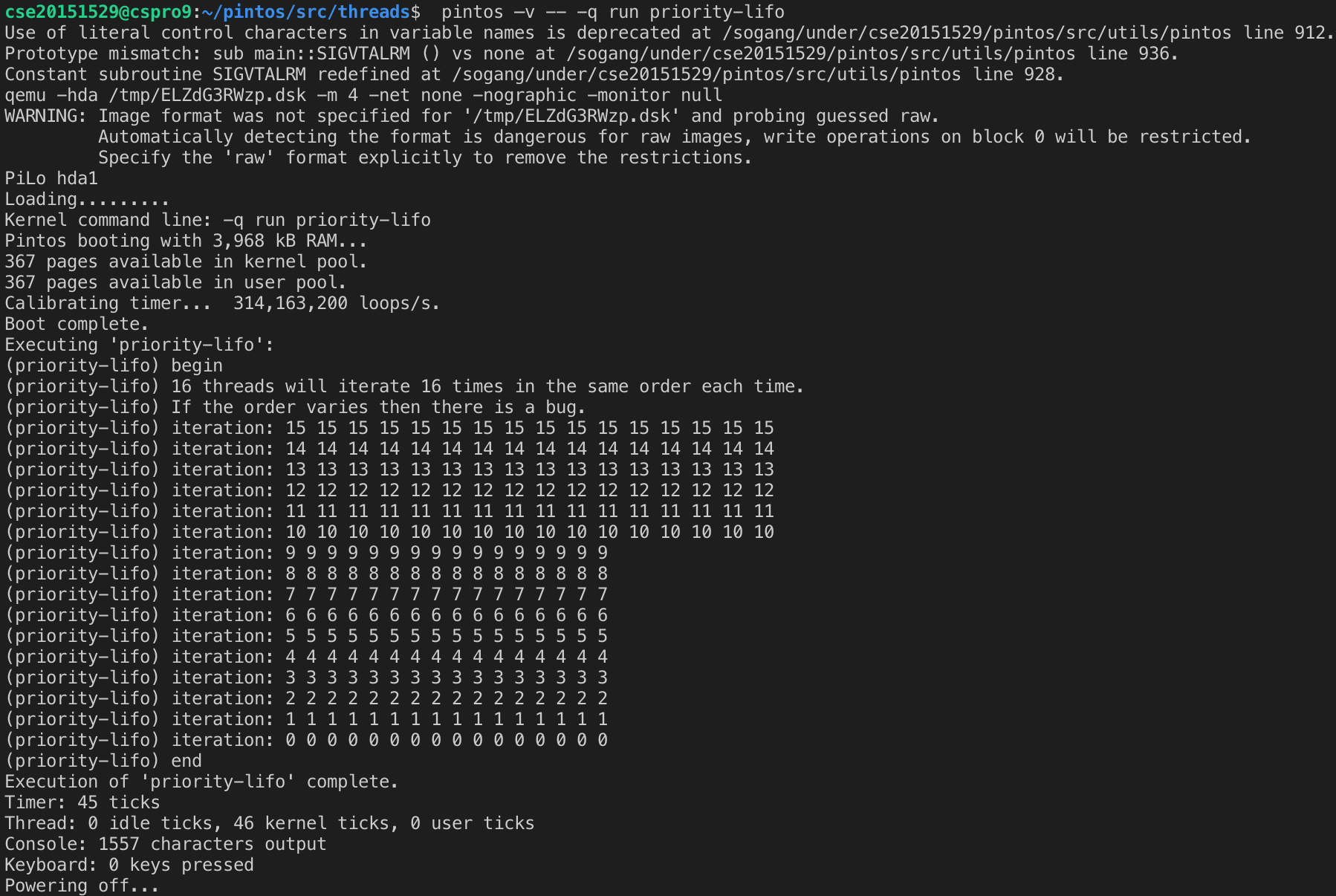
* **MLFQ(Multilevel Feedback Queue) scheduling:** MLFQ의 경우 이론대로라면 여러개의 queue를 가지고 구현해야 한다. 그러나 결국 priority가 높은 애들만 우선적으로 실행하는 것이며, CPU 점유율에 따라 aging을 해주는 것이므로, prioirty와 aging 기능을 사용하면 하나의 queue로도 기능을 구현할 수 있다. 따라서 조건문을 추가하여 priority scheduing with aging인 경우이거나, mlfq scheduling을 사용하는 경우에만 thread\_aging()을 호출하여 기존 aging과 같은 작업을 수행하면 된다. 이를 위하여 thread\_mlfqs를 글로벌하게 선언하여 mlfq schduling을 허용하였는지 확인한다.

또한 mlfq에서는 priority의 값을 임의로 변경할 수 없기 때문에, set\_priority()에서 예외 처리를 해준다.

* 1. **시험 및 평가 내용**

****

**(1) Make Check 결과**

****

**(2) priority-lifo 실행 결과**

1. **기타**
   1. **연구 조원 기여도**

* **김예찬 90%, 이수연 10%**
  1. **소감**
* **김예찬:** 마지막 프로젝트라서 즐거운 마음으로 임할 수 있었다. 설계한대로 구현했는데 잘 돌아가서 역시 즐거웠다.
* **이수연:** Priority scheduling은 명세서에 제공된 식을 이용하면 되었기 때문에 큰 문제가 없었다. MLFQ scheduling의 경우 처음에는 여러개의 큐를 사용해야 한다고 생각했기 때문에, 꽤 까다로운 문제라고 생각하였다. 그러나 그 개념을 정확하게 이해하고 생각해보니, 결국 priority sheduling을 하면서 CPU 점유율에 따라 aging을 시켜주면 되는 것이었다. 그에 따라 기존에 구현한 내용과 하나의 queue로 아주 쉽게 구현할 수 있게 되어 놀라웠다.