**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20171680 이호성

개발 기간 :10.30~11.1

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

이번 프로젝트는 busy waiting을 하는 기법에서 block을 사용하는 기법으로 바꿔주는 것과Round Robin을 사용하는 pintos의 scheduling 기법을 Priority Scheduling으로 바꿔주는 것이다. 이에 추가로 MFLQ의 구성을 갖는 BSD Scheduler를 구현할 예정이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock : 현재 timer\_sleep() 함수는 busy waiting 기법을 사용하고 있다. busy waiting 기법이 유용한 상황도 있지만 대부분의 경우 CPU 자원의 낭비를 초래하므로 block 하는 방식을 채택하도록 한다.
  2. Priority Scheduling : 현재 pintos 는 Round Robin 스케줄링 기법을 사용한다. 이를 우선순위를 갖는 Priority 스케줄링 기법으로 바꿔 줄 예정이다. 이를 통해 우선순위가 높은 중요한 thread 들을 먼저 처리할 수 있게 될 것이다.
  3. Advanced Scheduler (추가구현을 한 경우) : niceness를 기반으로 recent\_cpu 와 load\_average를 수정해나가며 Priority 역시 수정해 나간다. 이를 통해 조금 더 공평한 스케줄링 기법을 활용하여 pintos를 구동할 수 있게 될 것이다.
  4. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

blocked thread 들을 list에 만료시간이 적은 순으로 정렬되게 한다. 그 다음 unblock을 할 상황이 오면 list의 맨 앞 thread를 wake 하면 된다.

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

Ready list에 running thread 보다 높은 priority를 가진 thread 가 들어온다면 preemptive 하게 thread\_yeild() 를 하여 running thread를 교체해 준다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

nice값을 기반으로 load\_avg 를 최신화 하고 이를 통해 recent\_cpu를 구해야 하므로 각 식을 계산하는 함수와 지정 시간마다 두 값과 priority를 최신화 하는 함수들을 작성하여 MLFQ를 구현할 수 있도록 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10.30 Alarm Clock 구현

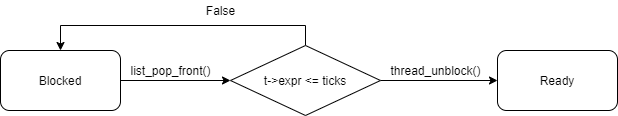
10.31 Priority Scheduling 구현

11.1 BSD Scheduler 구현

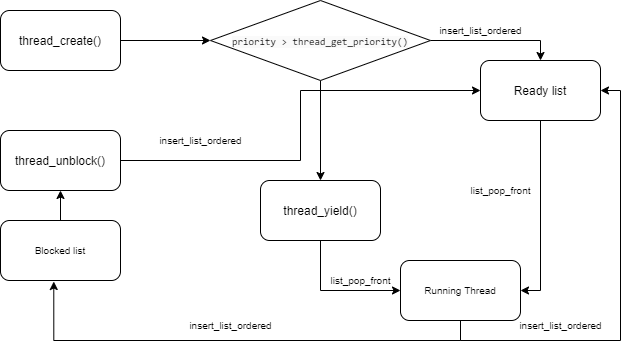
* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
* Alarm Clock : timer\_sleep() 과 timer\_interrupt() 안에 있는 thread\_tick()함수를 변형할 예정이다. 새로운 변수 expr 을 만들고 block 기법을 활용한다.
* Priority Scheduling : Thread 를 만들 때 priority라는 변수를 이제 활용할 예정이므로 thread\_yield() thread\_unblock() thread\_create() thread\_set\_priority() sema\_up() thread\_tick()을 수정/추가 한다. 이때 pintos는 floating point arithmetic 을 채택하지 않고 fixed point arithmetic을 기반 하므로 고정 소수점 연산 함수를 구현하여 계산을 용이하게 한다.
* BSD Scheduler : nice load\_avg recent\_cpu 등의 변수를 생성하여 MLFQ를 구현하도록 한다. 현재 ready list에서 최대 priority를 반환하는 max\_priority 함수, nice , load\_avg recent\_cpu를 계산해주는 함수 이것을 최신화해주는 함수를 만들어 주도록 한다.

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)
* **Alarm Clock**

각 thread의 만료시각을 나타내는 변수 expr을 struct thread안에 추가하며 thread\_init()에 초기화 하는 부분을 추가를 하였다. 그리고 원래 busy waiting으로 구현되어 있던 timer\_sleep()을 호출한 thread를 현재 시각을 기준으로 ticks 시간이 경과할 때까지 block하는 기법으로 교체하였다. thread 의 expr 변수를 strat+ticks로 설정하고 list\_insert\_ordered()함수를 이용하여 expr을 오름차순으로 정렬해놓으면 나중에 unblock할 때 맨 앞 원소만 확인하면 효율적인 구동이 가능해진다. timer\_interrupt() 안에서 thread\_tick()을 확인하게 되는데 이때 피피티에 추가하라고 나와있는 thread\_wake\_up()에서 구현하였다. 여기서는 alarm\_list의 맨 앞원소를 확인하여 만료 시간이 ticks보다 적어졌다면 스레드를 unblock 해주게 한다.

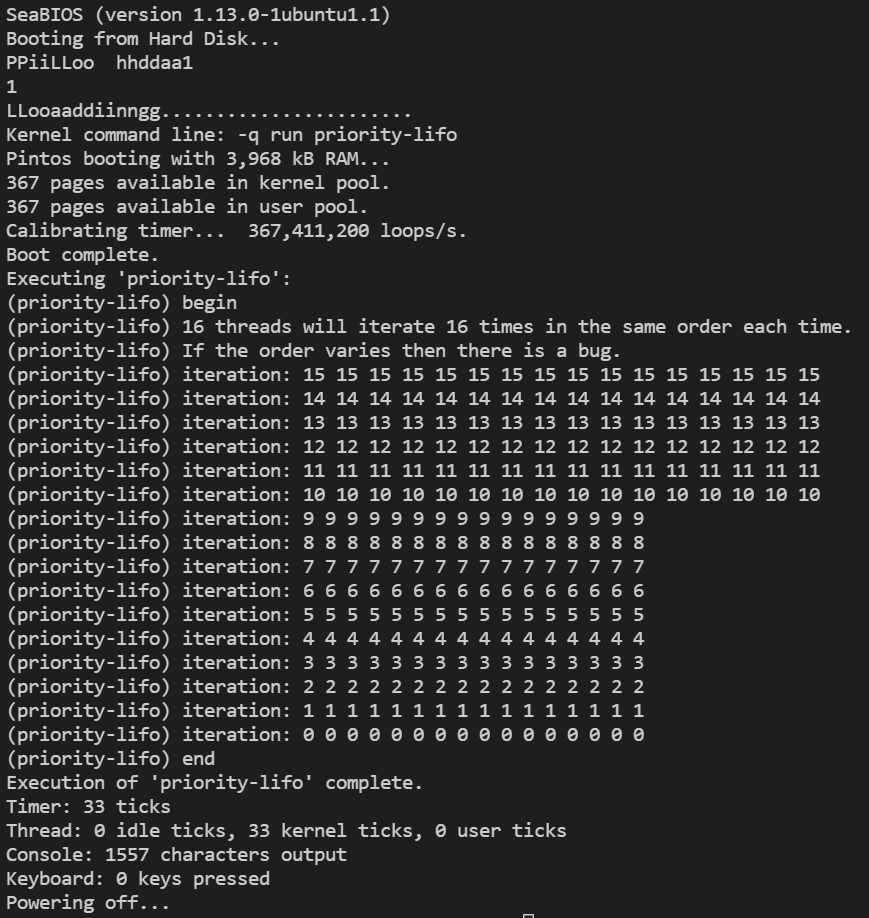
* **Priority Scheduling :**

thread\_yield() 함수에서 ready\_list에 thread를 넣을 때 맨 뒤에 넣는 것이 아닌 priority 순으로 정렬하기 위해 list\_insert\_ordered() 함수를 활용한다. 또한 thread\_unblock() 에서도 ready\_list로 thread를 보내주기 때문에 이 함수에서도 이전과 같이 list\_insert\_ordered()를 추가하여 priority 순으로 정렬되게 만든다. 그 다음 새로 생성되는 thread의 priority가 running thread 보다 크면 다시 스케쥴링을 해야하므로 thread\_create() 에서 생성을 완료 한 후 return 하기 전에 비교를 하여 크다면 thread\_yield() 를 통해 running\_thread를 ready\_list 에 추가하고 priority가 현재 스레드보다 큰 새로 생성된 thread는 ready\_list의 맨 앞에 있을 것이므로 새로 생성된 thread가 돌아가도록 한다. 여기까지 했을 때는 priority 테스트케이스 중 sema 와 aging이 안된다. sema testcase의 경우 sema\_up() 함수에 thread\_unblock()이 있다. 이 부분 역시 원래는 priority 가 없었기 때문에 priority 가 가장 높은 thread를 unblock해줘야 하기 때문에 비교하는 부분을 추가하였다. aging의 부분은 우선 pintos 가 floating-point-arithmetic을 지원하지 않기 때문에 14비트를 옮겨 fixed-point arithmetic을 이용하여 계산해야 한다. 이 계산식들을 사용하기 용이하게 함수로 만들어 두었다. 그 이후 BSD Scheduler 와 함께 구현하였다.

* **BSD Scheduler**

nice 와 recent\_cpu는 thread 각각의 변수이고 load\_avg는 thread 가 공유하는 변수 이므로 struct thread 부분에 recent\_cpu와 nice를 추가하였다. 그리고 thread.c 에 전역변수로 static int load\_avg를 추가하였다. thread 에 변수를 추가한 만큼 초기화 하는 부분 역시 만들어 주었다. 그 다음 ready list를 순회하며 현재 최대 priority를 구하는 max\_priority() 함수를 만든다. 그리고 현재 nice ,load\_avg, recent\_cpu 값을 반환하도록 thread\_get\_nice() thread\_get\_load\_avg thread\_get\_recent\_cpu 함수를 설정한다. 이때 load\_avg 와 recent\_cpu는 피피티에 나와 있듯 100을 곱해준 값을 반환하도록 한다. 그리고 load\_avg와 recent\_cpu를 최신화 하는 식에 맞춰 load\_avg 를 최신화 한 후 thread 의 리스트를 순회하며 idle thread 가 아닌 경우 recent\_cpu 를 update 하는 update\_load\_avg\_and\_recent\_cpu() 함수와 priority 를 update 하는 update\_priority()를 추가한다. 여기까지 완료 하였으면 thread.c/thread\_tick()에 추가한 priority\_aging()함수를 작성하면 된다. thread\_prior\_aging 변수가 true이면 recent\_cpu 값을 1 증가시키고 recent\_cpu value of all thread is updated in every second (1sec = TIMER\_FREQ) 이라고 피피티에 명시되어있기 때문에 timer\_ticks() 가 TIMER\_FREQ 이 지날 때 마다 update\_load\_avg\_and\_recent\_cpu()를 호출하고 Recalculate priority in every 4 ticks (same as TIME\_SLICE) 이기 때문에 timer\_ticks()가 4씩 증가할 때 마다 update\_priority()를 해준다. 여기까지 한 후 priority-aging을 확인하였더니 통과하였다. 여기서 추가로 thread\_mlfqs 가 true 일 경우 thread\_set\_priority()를 수행하지 않고 thread\_set\_nice()에서 nice 값을 기반을 기반으로 priority를 설정한다. 추가적으로 thread\_aging()을 하는 경우를 thread\_prior\_aging 말고 thread\_mlfqs 가 true일 때도 조건문에 들어가게 한다면 추가구현에 해당하는 testcase 도 통과할 수 있다.

* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석



priority-lifo의 경우 data가 0~15의 순으로 들어오게 된다. 이때 제일 나중에 들어온 15가 실행되며 다시 list에 들어오게 된다. 그 다음 순서 역시 맨 마지막에 들어온 15이기 때문에 이와 같은 방식으로 15가 MAX\_ITER 인 16번을 돌게 되며 그 다음 순차적으로 14~0 이 순서대로 MAX\_ITER만큼 돌게 된다.

* make check 수행 결과를 캡처하여 첨부
* 