**Pintos Project 3: Threads**

담당 교수 : 김영재

조 / 조원 : 20161580 김현규

개발 기간 : 2020.11.23 ~ 2020.12.05

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

Alarm Clock, Priority Scheduling을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

src/devices/timer.c는 busy waiting 방식으로 구현되어 있다. 이는 무한 루프를 돌면서 체크하므로 CPU 자원을 낭비하게 된다. 이를 해결함으로써 좀더 효율적으로 구현할 수 있다.

* 1. Priority Scheduling

현재 pintos는 round-robing 방식으로 scheduling해주고 있다. 이를 priority scheduling으로 바꿔준다.

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

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

pintos에 내재되어있는 함수 thread\_unblock(&thread)를 이용하면 unblocked된 thread를 깨울 수 있다.

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

Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 rescheduling을 통해, 높은 priority를 가진 thread를 run하고, running thread는 ready queue로 돌려놓는다.

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

2020.11.23 ~ 2020.11.30 1)Alarm clock 구현

2020.12.01 ~ 2020.12.05 2)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

1) Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

src/threads/thread.h의 struct thread에 해당 thread가 wake할 시간을 저장하는 변수 저장.

src/devices/timer.c에 아직 wakeup하지 않고 sleep하는 thread들을 저장하는 list선언.

void timer\_sleep(int64\_t ticks)에서 busy waiting으로 구현된 sleep 과정을 함수 thread\_block()을 이용하여 수정. 그리고 sleep하는 thread들을 list에 추가하게끔 수정

void timer\_interrupt(~)에서 list를 탐색하여 깨어날 시간이 된 list를 발견하면 깨워주게끔 수정

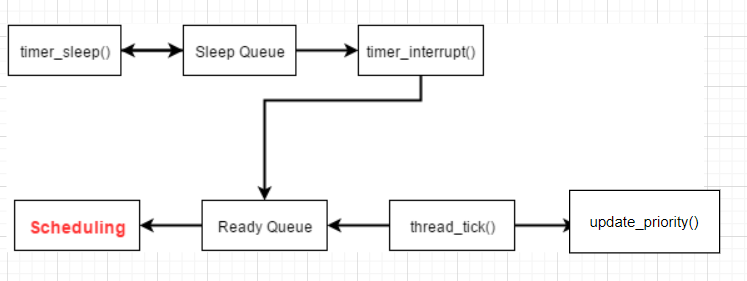
2) Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.

src/threads/thread.c에 두 thread 간의 priority를 비교해주고 그 결과를 return하는 함수 bool priority\_comp 구현. thread\_create 그리고 현재 thread\_yield와 thread\_unblock은 priority scheduling에 관련없이 ready\_list에 thread를 추가해주고 있다. 이를 priority에 고려하게끔 수정.

thread\_create, thread\_set\_priority 두 함수 역시 priority를 고려하지 않았다. 새로 추가된 thread의 priority가 현재 thread들중 가장 높은 priority보다 클 경우 다시 scheduling하게끔 수정

1. **연구 결과**
   1. **Flow Chart**

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



* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 1. Alarm Clock

현재 Pintos의 timer\_sleep()은 Busy Waiting으로 구현되어 있다. 무한 루프를 돌면서

체크를 하기 때문에 CPU를 낭비한다. 이렇게 하지 않고, Thread마다 시간을 체크하여 BLOCK 상태를 만들고(Waiting State), 후에 조건이 되면 UNBLOCK 상태로 전환해줌으로써 이를 해결할 수 있다.

struct Thread에 Thread가 깨어날 시간을 의미하는 변수인 wakeup\_time를 선언한다. Sleeping하는 Thread를 따로 모아주는 list를 선언하고, 해당 list를 탐색할 때 해당 thread의 변수가 Wake up할 시간이 되었으면 해당 Thread를 깨워준다. 즉, Ready Queue에 삽입한다.

깨워주는 작업은 src/devices/timer.c의 static void timer\_interrupt(~)에서 구현한다. 해당 thread의 변수인 wakeup\_time이 현재 시간(ticks)보다 작거나 같으면, 일어나야 하는 시간이므로, 해당 thread를 sleeping llist에서 삭제해준 뒤, thread\_unblock(thread)을 통해 UNBLOCK상태로 전환해준다.

* + 1. Priority Scheduling

src/threads/thread.c void thread\_yield(void): 해당 함수는 현재 thread를 ready\_queue(코드에서는 ready\_list)에 넣고 rescheduling하는 함수다. thread들이 priority 크기 순으로 정렬하게 하기 위해, ready\_queue에 thread를 넣을 때 list\_insert\_ordered(&ready\_list, …, priority\_comp, …)을 실행한다. 같은 이유로, thread\_unblock(~)에서도 마찬가지로 list\_insert\_ordered(~)을 실행한다. 한편, 함수 bool priority\_comp(const struct list\_elem\* left, const struct list\_elem\* right, void\* aux)는, left와 right의 thread(list\_entry(left, struct thread, elem))의 priority를 비교하여, left가 더 크면 true를 return, 다른 경우 false를 return하는 함수이다. 이 함수를 list\_insert\_ordered의 매개 변수로 씀으로써 해당 함수를 사용할 때 list의 thread들의 priority를 고려하여 insert할 수 있다.

src/threads/thread.c tid\_t thread\_create(~): 해당 함수는 thread를 새로 생성하는 함수다. 이 때 priority 역시 고려해줘야 한다. priority shceduling의 특성상 현재 수행되고 있는 thread의 priority(이는 thread\_get\_priority()로 알 수 있다.)가 새로 생성할 thread의 priority보다 큰 경우 rescheduling해줘야한다. 따라서 이 부분을 추가해줘야한다.

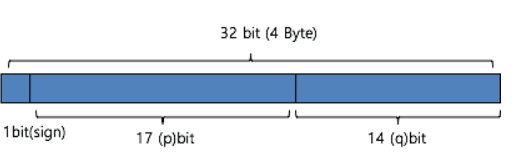
* + 1. Priority aging

priority를 aging해줘야 한다. 그러기 위해서는,

priority = PRI\_MAX – (recent\_cpu / 4) – (nice \* 2)

recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu + nice

위 두 식을 구현해야 한다. 그러기 전에, pintos는 floating point 방식의 연산을 지원하지 않으므로 fixed point 방식의 연산을 구현해줘야 한다.



위 그림을 참고하여 fixed point 방식의 연산을 구현해주자.

threads/thread.c의 void thread\_init(void)에서 load\_avg를 0으로 초기화해준다. 그리고 static void init\_thread(struct thread \*t, const char \*name, int priority)에서 해당 함수의 매개변수인 thread\* t의 recent\_cpu와 nice를 각각 현재 돌아가고 있는 running\_thread의 recent\_cpu와 nice로 초기화해준다.

그리고 load\_avg와 recent\_cpu를 return해주는 함수 thread\_get\_load\_avg와 thread\_get\_recent\_cpu를 구현해주는데, 주석에서 써 있는 바와 같이 100을 곱해준 후, 이를 FRACTION으로 나눔으로써 정수형으로 바꿔준 뒤 return한다.

그리고 load\_avg와 recent\_cpu를 update해주는 함수를 구현한다. 주어진 공식인,





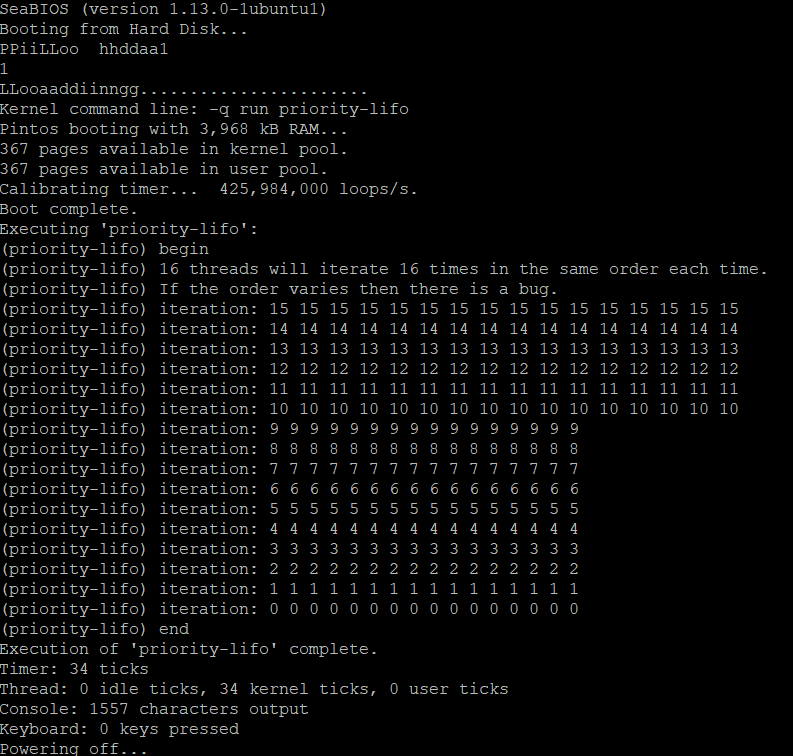
을 fixed point 방식으로 구현해준다.

또한 priority를 update해주는 함수를 구현한다. 주어진 공식인,



을 fixed point 방식으로 구현한다. 이 때, 계산된 priority가 PRI\_MAX 위이거나, PRI\_MIN 아래일 경우 각각 PRI\_MAX, PRI\_MIN으로 선언해준다.

위의 update해주는 함수들은 devices/timer.c에서 실행해준다. timer\_ticks()는 현재 시간을 return해준다. 이를 이용하여, 특정 시간마다 load\_avg와 recent\_cpu, 그리고 priority를 update해주는 함수를 실행한다. 이 경우는 priority-aging할 경우에만 실행이 되어야하므로, 현 상황이 해당 경우일 때 true로 setting되는 flag를 선언한다. (threads/thread.c, threads/thread.h에 선언)

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

우선, 16\*16개의 array에 16개의 threads는 실행될 때 마다, PRI\_DEFAULT+1부터 16개를 넣고, +16까지 들어가게된다. 하지만 Main이 PRI\_DEFAULT+17이므로 수행되지 못하고 대기하다가, Main 이 default값으로 priority를 낮추면 나중에 들어갔지만 제일 우선순위가 높은 thread부터 실행된다. 따라서 위와 같은 출력결과를 얻는다.

* make check 수행 결과를 캡처하여 첨부

