**Pintos Project 3: Threads**

담당 교수 / 분반 : 김영재 교수님 / 1

이름 / 학번 : 오승철 / 20161611

개발 기간 : 10/29 ~ 11/3

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

Alarm clock의 개선, priority에 따른 scheduling, starvation 방지, 1queue BSD scheduler

1. **개발 범위 및 내용**
   1. **개발 범위**

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

기존의 timer\_sleep 함수는 비효율적으로 thread\_yield를 반복하는 Busy waiting 방식으로 구현되어 있다. 이를 ready list를 관리하여 wake 되어야 될 때만 thread를 깨우는 방식으로 수정한다.

* 1. Priority Scheduling

Priority에 따라 thread의 우선순위가 정해지고 이에 따라 thread를 실행할 수 있도록 하여 보다 효율적인 scheduling이 가능하게 한다. 추가로 low priority를 가지는 thread들의 Stravation을 막기 위해 aging을 구현한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

각각의 thread들을 I/O와 CPU time을 고려하여 보다 더 효율적으로 scheduling하기 위한 방법으로 multilevel feedback queue방식을 이용한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Timer\_interrupt 발생시 마다 sleep list의 원소들을 순회하며 각각의 원소들이 깨어나야 할 상태이면 list에서 지운 후 unblock해준다.

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

새로 들어온 thread의 priority가 runngin thread의 priority보다 크면 현재 상태에서 가장 큰 priority이므로 running thread가 새 thread로 바뀌어야 한다.

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

Priority 계산을 위해 고려해야할 요소가 수식을 보면 recent\_cpu, nice 두가지가 있는데 recent\_cpu는 load\_avg를 이용해서 계산하게 되므로 recent\_cpu, nice, load\_avg 세가지가 필요하다. Recent\_cpu는 최근 cpu time을 의미하며, nice는 priority에 영향을 주기 위한 추가 변수이고, load\_avg는 ready state인 thread의 수의 평균을 의미한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

Alarm clock :10/29 ~ 10/30

Priority Scheduling : 10/31

Advanced Scheduler : 11/1 ~ 11/3

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드

Devices/timer.c, threads/synch.c, threads/thread.c, threads/thread.h

* + 수정하거나 추가해야 하는 자료구조

Thread의 깨어날 시간(tick)을 저장하기 위한 int

Thread의 recnet\_cpu와 nice를 저장하기 위한 int

Load\_avg를 저장하기 위한 int

Sleep\_list를 위한 list

Aging 적용여부를 확인 하기 위한 bool

고정 소수점 연산을 위한 shift용 상수

* + 수정하거나 추가해야 하는 함수

고정 소수점 곱하기, 나누기 연산을 위한 함수

고정 소수점 수를 int형 priority로 변환하는 함수

List에 삽입할 때 우선순위 계산을 위한 cmp 함수

Priority를 update하는 함수

Recent\_cpu를 update하는 함수

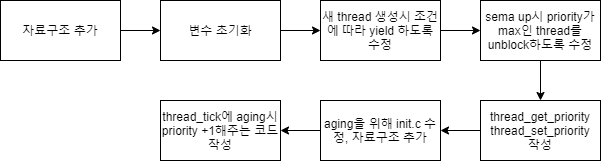
1. **연구 결과**
   1. **Flow Chart**

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

Alarm clock

C:\Users\dd\AppData\Local\Microsoft\Windows\INetCache\Content.Word\제목 없는 다이어그램.png

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

**Alarm clock**

Sleep 중인 thread의 elem을 저장할 sleep\_list를 추가하고 thread의 종료시점을 저장할 tick을 struct thread에 추가한다. 또한 sleep\_list는 thread\_init에서 list\_init 해준다.

timer\_sleep 함수의 busy waiting 부분을 thread에 tick을 저장하고 sleep\_list 뒤에 다시 삽입해주는 코드로 치환한다.

Timer\_interrupt는 interrupt발생시마다 sleep list를 돌면서 thread의 tick값이 현재 ticks보다 작으면, 즉 깨어나야 하는 thread를 찾아서 sleep\_list에서 제거하고 thread\_unblock하고 ready\_list에 뒤에 삽입한다.

**Priority Scheduling**

Alarm clock 에서 sleep list와 ready list에 삽입할 때 list의 맨 뒤에 삽입했다. 이를 list\_elem 2개를 parameter로 받아서 priority를 비교하는 함수 p\_cmp을 구현한 후 list에 삽입시 list\_insert\_ordered를 이용해서 priority 순서대로 삽입될 수 있도록 수정한다.

또한 timer\_interrupt 뿐만 아니라 sema up시에도 thread를 unblock하므로 priority를 고려해서 unblock을 하도록 코드를 수정해야 한다.

Sema->waiters를 순회하며 가장 큰 priority를 가지는 thread를 찾고 list에서 제거 한 후 unblock해주면 된다. 이때 unblock된 thread의 priority가 current thread의 priority보다 크면 먼저 작동해야 하므로 thread\_yield를 호출한다.

Thread\_yield는 현재 thread를 ready\_list 맨 뒤에 삽입하고 ready\_list에서 Thread하나를 꺼내서 running상태로 만들어주는 함수이다. 현재 thread를 ready\_list에 넣을 때 맨 뒤가 아닌 priority를 고려해서 삽입하도록 list\_push\_back을 list\_insert\_ordered로 수정한다.

Priority\_aging은 Pj3 pdf 31p에서 요구하는 기본적인 사항들을 수정해주고 thread\_prior\_aging이 true일 때 ready\_list를 순회하며 모든 priority를 +1해주는 코드를 thread\_tick에 추가하여 구현한다..

thread\_set\_priority는 현재 thread의 priority를 parameter로 들어온 new\_priority로 바꿔주고 new\_priority가 원래 가지고 있던 값보다 작으면 scheduling에 변화가 있을 수 있으므로 thread\_yield를 호출해준다. (ready\_list에 priority가 원래 값보단 작으나 new값보다 큰 thread가 존재 할 수 있다.) thread\_get\_priority는 현재 thread의 priority를 return하면 된다.

**Advanced Scheduler**

Struct thread에 recent\_cpu와 nice를 저장할 변수를 만들고 전역 변수로 load\_avg를 저장할 변수를 만든다. 이 변수들은 thread\_init에서 초기화한다. 고정소수점 shift 연산을 위한 FSHIFT 상수도 작성한다. init\_thread에서 recent\_cpu와 nice는 부모의 값으로 다시 초기화 해준다.

Thread\_tick에 aging과 else로 구분하여 1초(TIMER\_FREQ)마다 load\_avg와recent\_cpu를 갱신하고 4tick(TIME\_SLICE)마다 priority를 update해주는 코드를 작성한다. Load\_avg는 시행당 한번만 계산하면 되지만 recent\_cpu와 priority는 모든 thread에 대해서 갱신해야한다. 따라서 thread\_foreach를 통해서 all\_list를 순회하며 모든 thread에 한번씩 접근하며 값들을 갱신해 준다. 값을 갱신하는 함수는 각각 recpu\_update, pri\_update이며 함수 포인터의 형태로 thread\_foreach의 parameter로 쓰인다.

Priority 와 마찬자지로 thread\_set\_nice, thread\_get\_nice를 작성해야한다.

Thread\_set\_nice에서는 parameter로 현재 thread의 새로운 nice값들어오는데 이를 update해주고 priority가 nice의 영향을 받기 때문에 추가로 update 해 주어야 한다. (recent\_cpu 또한 영향을 받지만 recent\_cpu는 가장 최근 cpu\_time을 의미하므로 nice값이 변해도 변한 값과 무관 하기 때문에 추후에 코드 흐름에 따라 update 되게 하면 된다.)

Priority가 update되면 thread\_set\_priority와 같은 이유로 new\_priority가old\_priority보다 작을 때 thread\_yield를 호출한다.

Thread\_get\_nice또한 thread\_get\_priority와 동일하게 현재 thread의 nice값을 return해 주면 된다.

추가적으로 *#define FSHIFT 1<<14;*와 같은 형태로 코드를 작성해서 한참동안 틀린 부분을 찾지 못해서 고생했다. *#define FSHIFT (1<<14);*와 같이 괄호로 묶어주어야 의도한대로 잘 돌아간다.

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

lifo의 의미대로 나중에 들어온 것이 처음으로 실행되도록 구성된 테스트케이스이다.

코드를 보면 16개의 thread를 priority를 1씩 증가시켜가면서 생성하고 있음을 확인 할 수 있다.

실행결과를 보면 thread\_cnt인 16만큼의 iteration결과가 있으며 나중에 생성된 번호가 큰 thread의 priority가 상대적으로 크기 때문에 큰 것부터 실행되었다. 이는 lifo에 부합하게 살행 된 것이다. 또한 각각의 iter\_cnt가 16이어서 simple\_thread\_func 에서 loop를 16번 돌게 되어있는데 이 때문에 iteration 결과가 16번씩(같은줄에) 출력되는 것 같다.

