**Pintos Project 3: Threads**

담당 교수 : 박성용

조 / 조원 : 김태곤 / 20191583

개발 기간 : 11.13 ~ 11.19

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

이번 프로젝트에서는 thread에 관련된 기능을 구현한다. alarm clock을 구현하고, busy waiting으로 구현된 프로젝트에서 우선순위를 사용하는 priority scheduling을 구현한다. 또한 advanced scheduler인 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

초기 구현되어 있는 timer\_sleep() 함수는 busy waiting 방식을 사용한다. 이는 프로세스가 특정 시간동안 일시 정지해야 할 때, CPU 시간을 낭비하면서 그 시간이 지나길 기다린다. 즉 프로세스가 아무런 유용한 작업을 하지 않으면서 CPU를 점유하고 있어 CPU 자원의 낭비가 된다. 따라서 프로세스가 지정된 시간동안 CPU를 사용하지 않고 대기(Block) 상태로 전환하게 하여, CPU가 다른 작업을 수행할 수 있도록 한다.

* 1. Priority Scheduling

초기 구현되어 있는 scheduling 메커니즘은 Round Robin 방식이다. 즉 프로세스 사이 우선순위를 두지 않고, ready\_list의 마지막에 추가된다. 이를 우선순위를 고려한 scheduler를 적용시켜 보다 높은 우선순위를 가진 thread를 먼저 수행시킨다.

* 1. Advanced Scheduler (추가구현을 한 경우)

Ppt에 제시된 조건에 맞게 BSD Scheduler를 구현한다. BSD Scheduler에서는 각 MLFQ를 구현하여 각 priority마다 ready queue를 가지게한다. 또한, 제시된 nice, priority, recent\_cpu, load\_avg를 이용하여 priority를 적절히 계산한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 thread를 어떻게 깨울 수 있는지 서술.

먼저 timer\_sleep함수에서 thread를 block상태로 만드는 과정이 일어난다. 해당 과정에서 timer\_ticks()함수를 호출하여 start(시간)을 가져온다. 현재 thraed의 end\_time을 설정하고, ticks 수를 더해 계산한다. 이후 end\_time 기준으로 정렬하여 alarm\_list에 삽입한다. 이로 인해 가장 먼저 깨워져야 할 thread가 list의 앞에 오게 된다. Unblock을 하게 될 상황이 오면, list의 맨 앞의 thread를 wake시키면서 Block 상태의 thread를 우선순위에 맞게 깨울 수 있게 된다.

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 들어온 경우, preemptive하게 switch가 발생한다. Priority를 비교해서 thread\_yield를 수행시키고, thread\_yield는 우선순위에 맞게 list에 push한다.

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

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

각 thread의 우선 순위는 기본 우선 순위 값에서 thread의 recent\_cpu와 nice를 고려하여 감소시킨다.

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

recent\_cpu는 thread가 얼마나 많은 CPU 시간을 사용했는지를 추적한다. 이 값은 시간이 지남에 따라 load\_avg에 비례하여 증가하고, thread의 nice 값에 의해 조정된다.

load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_threads

load\_avg는 시스템의 평균 부하를 나타내며, 실행 준비 상태에 있는 thread의 평균 수를 반영한다. 이 값은 시스템이 얼마나 바쁜지를 나타내는 지표로 사용되며, recent\_cpu의 계산에도 영향을 미친다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

11.13~14 : 프로젝트 이해 및 공부

11.15 : Alarm Clock 구현

11.16~17 : Priority Scheduler 구현

11.18~19 : BSD Scheduler 구현

11.19 : 보고서 작성

* 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

**-Alarm Clock**

Thread 구조체 내 end\_time 변수를 추가하여 각 thread의 만료 시간을 나타냈다.

struct list alarm\_list : 슬립 중인 thread들을 관리하기 위한 전역 리스트가 추가되었다. 이 리스트는 각 thread의 end\_time에 따라 정렬된다.

timer\_sleep() : 주어진 틱 수만큼 현재 thread를 슬립 상태로 만드는 함수이다. 이 함수는 현재 thread를 alarm\_list에 삽입하고 thread를 Blocked 상태로 만들어 스케줄러가 실행하지 않도록 한다.

**-Priority Scheduling**

thread\_set\_priority()와 thread\_get\_priority() 함수는 thread의 우선 순위를 업데이트하고 반환하는 기능을 제공한다.

Thread\_create(), thread\_unblock() 및 thread\_yield() 함수에서 Ready 리스트에 thread를 삽입할 때 우선 순위를 비교한다.

Ppt32쪽에 나온 extern bool thread\_prior\_aging을 추가한다. 이를 통해 기아 상태를 해결한다.

Sema\_up()을 수정하여 priority를 고려해준다.

**-BSD Scheduler**

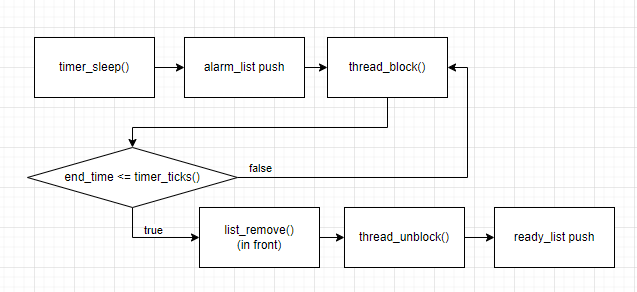
BSD Scheduler에서는 MLFQ를 구현을 위한 priority, recent\_cpu, load\_avg 계산 과정에서 Fixed-point format 계산이 필요하다. 이를 위해 foperation 함수를 선언하여, operation type에 따른 fixed-point format 계산을 수행하였다. 또한, operation type을 위한 enum type을 선언하였다.

Update\_larc() 함수를 선언하여 load\_avg와 recent\_cpu를 update하는 함수를 선언하였다. 또한, priority update를 위한 update\_priority 함수를 선언하였다.

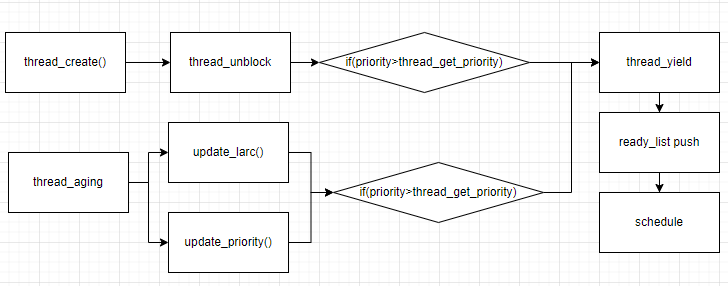
1. **연구 결과**
   1. **Flow Chart**

* 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)
  + 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**

먼저 thread 구조체에 thread의 끝 시간을 나타내는 end\_time을 추가하고, 초기화 해주었다. 이후 timer\_sleep 함수를 수정하여 기존 busy waiting 방식에서 block을 추가하였다. Timer sleep 함수에서는 end\_time을 timer\_ticks()를 통해 얻은 start값과, ticks 값의 합으로 업데이트하고, alarm list를 순회하면서 현재 end\_time이 순회하면서 있는 thread의 end time보다 작을 경우 insert과정을 수행해 오름차순으로 정렬했다. 이를 통해 unblock을 할 때 가장 먼저 오는 원소를 비교함으로 효율을 높일 수 있었다. 이후 thread\_block을 통해 block 상태로 만들어 주었다.

Unblock은 thread\_tick() 함수에서 일어난다. Thread\_tick함수에서 unblock 시키는 thread\_wake함수를 호출한다. Alarm list를 순회하면서 end\_time이 timer\_ticks()보다 작은 경우 해당 요소를 제거하고, thread\_unblock을 시켜준다. Thread\_unblock은 ready\_list를 순회하면서 해당 thread의 priority가 다른 thread의 priority보다 큰 경우 ready\_list에 insert 시키며 관리하였다.

**-Priority Scheduling**

Thread scheduling에서 기존 ready\_list의 가장 뒤에 thread를 추가하는 방식이 아닌, priority에 따라 정렬되어 삽입하는 방식으로 수정하였다. thread\_yield() 함수 내에서 thread를 ready\_list에 추가할 때, 기존에는 리스트의 끝에 추가하는 방식이었으나, ready\_list를 순회하며 현재 priority와 비교하고, 현재 priority가 해당 thread의 priority보다 큰 경우 list에 삽입, 순회가 끝날 때까지 삽입이 안된 경우 가장 뒤에 삽입함으로써 우선순위에 따른 정렬을 구현하였다. 이와 같은 방식으로 thread\_unblock() 함수에서도 thread를 ready\_list에 추가할 때 우선순위 순으로 정렬하여 추가하였다.

새로운 thread 생성 시, thread\_create()함수에서 생성이 끝난 후 반환하기 전에 현재 실행중인 thread와의 우선순위를 비교하였다. 새로운 thread의 우선순위가 현재 실행중인 우선순위보다 높다면, thread\_yield()를 호출하여 실행중인 thread를 ready\_list에 추가하였고, 새 thread가 실행될 수 있도록 하였다.

이렇게 우선순위에 따라 thread를 조정하면서 semaphore와 aging관련 케이스에서 문제가 발생하지 않도록 추가 조치하였다. Synch.c의 sema\_up() 함수에서 thread\_unblock() 부분에 우선순위를 비교하는 로직을 추가하여, 가장 높은 우선순위를 가진 thread가 우선적으로 unblock될 수 있도록 구현하였다. 이 과정은 while문을 통해 sema->waiters를 순회하며 현재 priority가 max\_priority\_thread의 priority보다 큰 경우 갱신해주어 구현하였다.

Aging 구현을 위해 먼저 ppt에 있는 코드들을 추가하였다. 이후 thread\_aging() 함수에서recent\_cpu, load\_avg, priority를 조건에 맞게 업데이트 해주었다.

**-BSD Scheduler**

BSD Scheduler를 구현하기 위해 thread 구조체에 recent\_cpu와 nice 값을 추가하였고, 모든 thread가 공유하는 load\_avg는 전역변수로 thread.c에 static int 형태로 선언하였다. 이러한 값들을 활용하여 recent\_cpu, load\_avg를 업데이트 하는 update\_larc() 함수와 priority를 업데이트하는 update\_priority() 함수를 작성하였다. 또한 nice의 값을 설정하는 thread\_set\_nice() 함수와 각각의 요소 값을 얻어올 수 있는 get함수들을 구현하였다. 해당 값들을 계산할 때 Fixed-point format 계산이 필요하여 추가로 OperationType을 나타내는 enum과 foperation 함수를 구현하였다. Foperation 함수는 OperationType과 두 정수를 받아 switch문을 통해 해당 fixed-point format계산을 수행하는 로직을 구현하였다.

Recent\_cpu의 값은 아래와 같은 과정을 통해

(2\*load\_avg)/(2\*load\_avg+1)\*recent\_cpu+nice 해당 식을 구현하였다.

tmp = foperation(INT\_MUL\_FLOAT,2,load\_avg);

tmp1 = foperation(FLOAT\_ADD\_INT,tmp,1);

tmp2 = foperation(FLOAT\_DIV\_FLOAT,tmp,tmp1);

tmp = foperation(FLOAT\_MUL\_FLOAT,tmp2,t->recent\_cpu);

t->recent\_cpu = foperation(FLOAT\_ADD\_INT,tmp,t->nice);

load\_avg의 값은 아래와 같은 과정을 통해 (59/60)\*load\_avg+(1/60)\*ready\_threads 해당 식을 구현하였다.

tmp = foperation(INT\_MUL\_FLOAT,59,load\_avg);

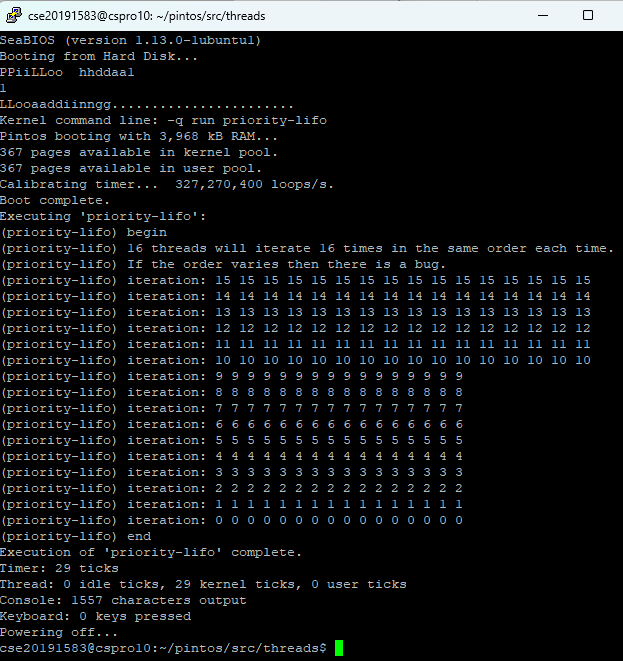
tmp = foperation(FLOAT\_ADD\_INT,tmp,ready\_threads);

load\_avg = foperation(FLOAT\_DIV\_INT,tmp,60);

priority의 값은 아래와 같은 과정을 통해 PRI\_MAX–(recent\_cpu/4)-(nice\*2) 해당 식을 구현하였다.

이처럼 load\_avg와 recent\_cpu를 갱신하는 update\_larc() 함수를 구현하여, 시스템의 부하 평균과 각 thread의 CPU 사용량을 최신 상태로 유지하고, 우선순위를 갱신하는 update\_priority() 함수도 추가하였다. 이를 통해 thread의 우선순위를 지속적으로 업데이트하여, 스케줄링의 효율성을 개선하였다. Thread\_tick() 함수에서도 priority\_aging() 함수를 추가하여 recent\_cpu 값을 증가시키는 기능을 구현하였다. 시스템의 타이머가 일정 주기를 넘을 때마다 recent\_cpu와 load\_avg를 갱신하고, 틱 마다 우선순위를 재계산하여 에이징 처리를 수행하였다. 또한 thread\_mlfqs가 true인 경우 thread\_set\_priority() 함수는 실행되지 않도록 하고, nice 값에 기초하여 우선순위를 결정하는 thread\_set\_nice() 함수를 통해 우선순위를 조정하도록 구현하였다.

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



먼저 c코드를 살펴보면 test\_priority\_lifo 함수가 호출되어 테스트를 시작한다. 이 함수는 MLFQS가 아닌 기본 스케줄링 상태에서 실행됨을 확인한다. 이후 현재 thread의 우선순위가 기본값인지 확인하고, 버퍼 할당 및 lock 초기화를 한다. 현재 thread의 우선순위를 가장 높게 설정하여 모든 thread가 생성되고 실행될 수 있게 하였다.

이후 THREAD\_CNT 수만큼 thread를 생성한다. 각 thread는 simple\_thread\_func 함수를 실행하며, 각자 다른 우선순위를 가진다. 이는 각 thread가 LIFO 순서대로 실행되도록 하기 위함이다. 현재 thread의 우선순위를 다시 기본값으로 설정하고, 다른 모든 thread가 실행을 완료할 때까지 대기한다. thread가 완료되면, output 버퍼에 기록된 ID 순서대로 결과를 출력한다. simple\_thread\_func 함수는 각 thread에 의해 실행된다. 각 thread는 공유된 락을 획득하고, 자신의 ID를 output 버퍼에 기록한 후 락을 해제하고 thread를 양보하는 thread\_yield 과정을 수행한다.

테스트 결과를 보면 테스트는 성공적으로 완료되었다. 각 thread는 자신의 ID를 16번 반복하여 출력하며, 이는 RR 스케줄링이 일관되게 유지되었음을 나타낸다. 결과 로그의 반복 부분에서, 각 반복 시 thread들의 실행 순서가 일관되게 유지되었으며, 모든 thread가 생성 순서대로 실행되었다. 이는 스케줄러가 LIFO 정책에 따라 우선순위가 높은 thread부터 실행하고 있음을 확인시켜준다.

* make check 수행 결과를 캡처하여 첨부

