**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20171669 / 이재영

개발 기간 : 2021.10.27 ~ 2021.11.05

1. **개발 목표**

Pintos의 thread에서 alarm clock을 개선하고 priority scheduling 방식을 구현해서 thread가 효율적으로 수행되도록 한다. 추가로 좀 더 효율적인 BSD Scheduler(MLFQS)를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
   2. **Alarm Clock**

기존 busy\_waiting 방식의 timer\_sleep 함수는 thread를 sleep 시킬 때 CPU를 비효율적으로 사용하는 문제가 있으므로 비효율적인 문제를 해결한다.

* 1. **Priority Scheduling**

기존의 scheduling의 경우 thread가 들어오면 ready list에 넣고 순서를 기다리는 구조인데 이 경우에는 앞의 thread가 cpu를 독식하고 새로 들어온 thread가 실행이 안되는 starvation 문제가 발생한다. 그러므로 thread마다 우선순위를 설정해서 우선순위가 높은 순서대로 실행되도록 구현한다. 이 때, aging 방식을 이용한다.

* 1. **Advanced Scheduler (추가구현을 한 경우)**

기존의 FIFO 방식의 scheduling을 Multi-level feedback Queue Scheduler의 방식으로 구현한다. MLFQS는 우선순위마다 ready queue가 존재하고 가장 높은 우선순위의 queue에 존재하는 thread가 먼저 선택되는 방식으로 진행된다. 이 때, 우선순위가 같은 queue에서는 Round Robin 방식을 채택한다.

* 1. **개발 내용**

1. **Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.**

Blocked 상태의 thread를 깨우기 위해서는 thread가 blocked 상태로 될 때, ready list에 깨우는 변수, wake를 저장해서 timer interrupt가 일어난다면 src/devices/timer.c에 있는 timer\_interrupt 함수에서 깨울 thread가 있는지 확인하고 깨울 thread가 있다면 thread\_unblock() 함수를 호출해서 unblock을 해주고 list에서 thread를 remove하면서 blocked 상태의 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 들어오는 경우 priority가 높은 thread가 먼저 수행되어야 하기 때문에 running thread를 ready list에 push하고 priority가 높은 thread가 running thread의 상태로 수행된다. 이 방식은 thread\_yield 함수를 호출하면서 수행할 수 있다.

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

Advanced Scheduler로는 BSD Scheduler를 구현했다. 즉, Multi-level feedback Queue를 구현했다. 이 때, priority를 계산하기 위한 요소들로는 niceness, recent\_cpu, load\_avg 가 있다.

Niceness : 각각의 thread들은 -20 ~ 20사이의 nice값을 가지는데 nice 값이 양수인 경우는 다른 thread가 CPU를 사용할 수 있도록 priority를 낮춰주고 0인 경우는 priority에 영향을 주지 않는다.

recent\_cpu : thread의 CPU를 사용하는 시간을 나타내는 값으로 가장 최근의 CPU 시간이 weighted되며 time interrupt가 발생할때마다 RUNNING state의 thread의 recent\_cpu값이 1 증가한다. 모든 상태의 thread의 recent\_cpu 값은 매초마다 다음과 같이 계산된다.

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

load\_avg : ready 상태의 thread들의 평균값을 나타내는 변수로 매초마다 update된다. 여기서 ready\_threads변수가 쓰이는데 이는 READY 또는 RUNNING 상태의 thread의 개수를 나타낸다.

load\_avg = \* load\_avg + \* ready\_threads

따라서 위의 priority계산을 위한 요소들을 통해서 priority는 다음과 같이 구할 수 있다.

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

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10.27 ~ 10.28 : pintos manual의 thread부분 분석

10.29 ~ 10.31 : Alarm Clock, Priority scheduling 구현

11.01 ~ 11.02 : Advanced scheduling(BSD Scheduler) 구현

11.03 ~ 11.04 : 보고서 작성

* 1. **개발 방법**

<src/threads/thread.h>

* Thread를 wake하는 것을 정할 변수 int64\_t wake와 BSD scheduler를 구현하기 위해 사용되는 priority를 구현하기 위해 필요한 요소들인 cpu사용하는 시간의 recent\_cpu변수와 nice변수를 thread 구조체에 선언한다.
* Fixed-pointer format을 사용하기 위해 필요한 fraction 즉, 14bit를 사용할 수 있는 것으로 FP 14를 define한다.

<src/threads/thread.c>

* 새로 구현한 함수들

bool comp\_priority : List element 두 개의 priority를 비교해서 앞의 thread의 우선순위가 높으면 true, 낮으면 false를 return 하는 함수이다.

void calculate\_all

: 함수로 넘어온 check값을 확인해서 check값이 0이면 다음과 같은 식으로 priority를 계산한다. priority가 범위를 벗어나면 조정해주는 과정을 수행한다.



check값이 1이면 아래의 식을 통해 load\_avg와 recent\_cpu를 계산해준다. 이 때 float끼리의 곱과 나눗셈은 새로 만든 함수로 계산하는데 아래에서 설명한다.



int multi\_float : Pintos kernel에서는 floating-point 연산을 지원하지 않는데 recent\_cpu, load\_avg를 구할 때는 floating-point 연산을 필요로 하기 때문에 fixed-point format을 대신 사용하는데 이는 F\_P를 통해 사용되며 float 곱을 구현하는 함수이다.

int divide\_float : float사이의 나눗셈을 구현하는 함수이다.

* 수정한 함수

thread\_create : 새로 들어온 thread의 우선순위가 현재 running thread의 우선순위보다 높으면 thread\_yield 함수를 호출해 running thread를 ready상태로 만드는 코드를 추가한다.

thread\_yield : 현재 running thread의 우선순위보다 새로 들어온 thread의 우선순위가 높은 경우 ready\_list에 running thread를 넣어주는 함수로 기존에 list\_sort로 삽입했던 것을 우선순위에 맞게 삽입하고 sort하는 List\_insert\_ordered로 수정한다.

thread\_unblock : blocked 상태의 thread를 unblock해주는 함수로 위와 같이 기존에 list\_sort로 삽입했던 것을 우선순위에 맞게 삽입하고 sort하는 List\_insert\_ordered로 수정한다.

thread\_set\_priority : thread\_mlfqs가 0이 아니라면 현재 thread의 우선순위에 새로운 우선순위를 넣고 현재 우선순위보다 새로운 우선순위가 작다면 thread\_yield()함수를 호출하도록 한다.

thread\_set\_nice : 새로운 nice값을 현재 thread의 nice값으로 설정하는 함수로 priority를 계산하고 ready\_list의 list의 앞쪽의 있는 것의 우선순위가 현재 우선순위보다 크면 thread\_yield()를 호출하도록 구현한다.

thread\_get\_load\_avg : 현재 system의 load average \* 100을 return 해준다.

thread\_get\_recent\_cpu : 현재 thread의 recent\_cpu \* 100을 return 해준다.

init\_thread : thread의 recent\_cpu, nice 값을 현재 running\_thread의 recent\_cpu, nice로 넣는다.

<src/devices/timer.c>

* sleep하는 thread들을 저장할 list , static struct list sleeping\_list를 추가하고 timer\_init함수에서 list\_init(&sleeping\_list)로 초기화한다.
* 수정한 함수

timer\_sleep : 현재 thread의 wake값을 start + ticks로 설정하고 sleeping list에 현재 thread의 element를 push한 후 thread\_block을 호출한다.

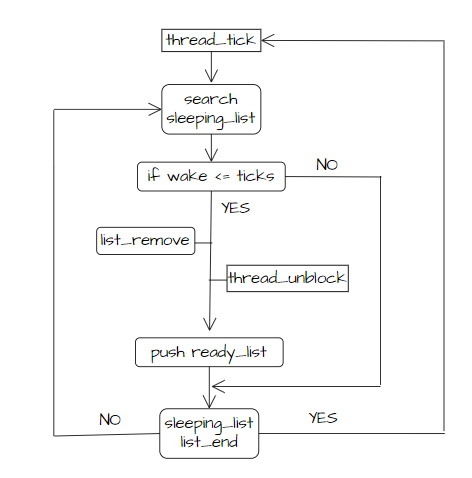
timer\_interrupt : time interrupt가 걸리면 sleeping\_list의 맨 앞의 것 entry의 wake 값을 찾아가면서 wake할 시간이 지난 경우에는 sleeping list에서 지우고 thread\_unblock, 즉 깨우게 하고 wake할 시간이 지나지 않았다면 다음 것을 확인한다. 그 후에 thread\_prior\_aging 또는 thread\_mlfqs가 참이라면 현재 recent\_cpu의 값을 구하고 timer\_ticks() % TIMER\_FREQ가 0이라면 calculate\_all함수로 check값 1을 보내서 load\_avg와 recent\_cpu를 계산한다. 또 timer\_ticks() % 4가 0이라면 calculate\_all함수로 check값 0을 보내 priority를 계산한다.

<src/threads/synch.c>

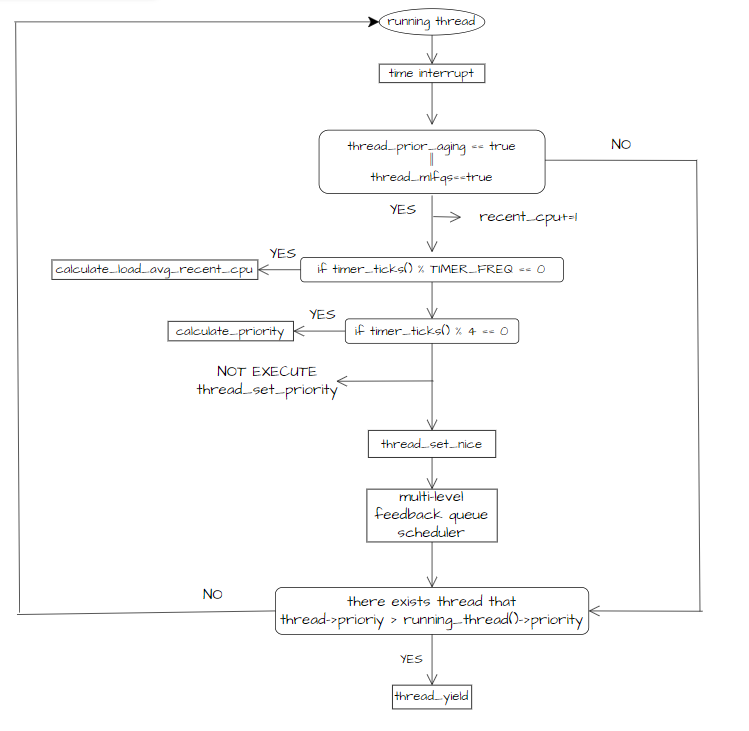
* 수정한 함수

Sema\_up : priority scheduling을 할 때, thread를 돌면서 우선순위를 정렬시키면서 가장 큰 우선순위의 것을 list에서 remove하고 해당 thread를 unblock해준다.

1. **연구 결과**
   1. **Flow Chart**
      1. **Alarm Clock**

****

* + 1. **Priority Scheduling**

****

* 1. **제작 내용**

1. **Alarm Clock**

<src/devices/timer.c>

* timer\_init

텍스트이(가) 표시된 사진

자동 생성된 설명

Static struct list sleeping\_list를 선언하고 timer\_init함수에서 list를 초기화한다.

* timer\_sleep

텍스트이(가) 표시된 사진

자동 생성된 설명

각각의 thread마다 메모리에 존재한 시간을 wake에 저장한 후, sleeping list에 thread를 넣고 thread block을 해주는 함수를 호출해서 blocked상태로 만든다.

* timer\_interrupt

Sleeping list의 끝까지 확인하면서 thread의 wake가 ticks보다 작거나 같은, 즉 wake할 시간이 지난 경우는 sleeping list에서 해당 thread를 지우고 thread unblock을 호출하면서 해당 thread를 깨우고 wake할 시간이 지나지 않은 경우는 다음 thread를 확인하는 방법을 사용한다.

List를 확인한 이후에 낮은 우선순위를 가지는 프로세스의 starvation을 해결하기 위해 thread\_aging방법을 사용한다. Thread\_prior\_aging이 true라면 thread의 recent\_cpu, load\_avg, priority를 다시 계산한다. Recent\_cpu와 load\_avg의 경우는 TIMER\_FREQ로 나누었을 때 나머지가 0일 때, 우선순위는 tick이 4로 나누어 떨어진 경우 다시 계산을 해주면서 aging 기법을 수행한다.

1. **Priority Scheduling**

<src/threads/thread.h>

Aging을 위한 thread\_prior\_aging bool 값과 fixed float format 시 사용되는 float pointer 14bit를 나타내는 F\_P 14를 선언한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Thread 구조체안에서 최근 cpu 시간을 나타내는 recent\_cpu, nice, wake를 선언한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

<src/threads/thread.c>

* thread\_init

텍스트이(가) 표시된 사진

자동 생성된 설명

Initial\_thread의 recent\_cpu와 nice값을 0으로 초기화해주고 load\_avg도 0으로 초기화한다.

* init\_thread

텍스트이(가) 표시된 사진

자동 생성된 설명

Init\_thread함수로 넘어온 thread t의 recent\_cpu와 nice를 각각 running\_thread의 recent\_cpu, nice로 설정한다.

* comp\_priority

텍스트이(가) 표시된 사진

자동 생성된 설명

List\_elem으로 들어온 구조체 포인터를 list\_entry함수를 이용해 thread1과 thread2로 선언하고 각각의 thread의 우선순위를 비교해 처음으로 들어온 thread의 우선순위가 큰 경우 true를, 작은 경우는 false를 반환하는 함수이다. 이 함수는 list\_inserted\_ordered를 사용할 때, 우선순위를 비교해서 insert를 할 때 parameter로 쓰인다.

* thread\_create

텍스트이(가) 표시된 사진

자동 생성된 설명

새로 들어온 thread의 priority가 현재 running thread의 우선순위보다 큰 경우 thread\_yield를 호출해서 running thread를 ready list에 push하고 새로 들어온 우선순위가 높은 thread가 running thread의 상태로 수행된다.

* thread\_unblock / thread\_yield

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

Thread\_unblock은 blocked 상태의 thread를 unblock해주는 함수이고 thread\_yield는 현재 running thread의 우선순위보다 높은 새로운 thread가 존재하고 현재 running thread가 idle한 thread가 아니라면 우선순위에 맞게 삽입한다. 이 때 두 함수 모두 기존에는 단순히 list\_push로 사용했는데 priority scheduling을 구현하기 위해 우선순위별로 삽입하고 정렬해주는 list\_insert\_ordered함수로 수정한다. 이 때, 3번째 parameter로 새로 정의한 thread의 우선순위를 비교해주는 comp\_priority함수를 사용한다.

<src/threads/synch.c>

* sema\_up

텍스트이(가) 표시된 사진

자동 생성된 설명

Semaphore를 up할 때 block된 thread를 thread\_unblock해주는 함수로 sema->waiters에 대해서 list가 비어 있지 않은 경우에 list를 탐색하면서 우선순위도 탐색해주면서 가장 큰 우선순위의 list element를 저장해놓고 그것을 list에서 remove하고 해당 thread를 unblock해준다.

1. **Advanced Scheduler – BSD scheduler(MLFQS)**

<src/threads/thread.c>

* thread\_set\_priority

텍스트이(가) 표시된 사진

자동 생성된 설명

Thread의 우선순위를 설정해주는 함수로 BSD scheduler인 mlfqs를 나타내는 thread\_mlfqs가 true가 아닌 경우에만 실행된다. 현재 thread의 우선순위에 parameter로 넘어온 새로운 우선순위로 수정하고 새로운 우선순위가 현재 thread의 우선순위보다 낮으면 thread\_yield를 호출을 수행한다.

* thread\_set\_nice

텍스트이(가) 표시된 사진

자동 생성된 설명

현재 thread의 nice 값을 parameter로 넘어온 nice값으로 바꿔주고 우선순위를 계산해서 현재 thread의 우선순위를 바꿔준다. 이 때, ready\_list가 empty가 아니라면 ready\_list의 앞쪽에 있는 우선순위와 현재 우선순위를 비교해서 현재 thread의 우선순위가 낮다면 thread\_yield함수를 호출한다.

* thread\_get\_load\_avg / thread\_get\_recent\_cpu

텍스트이(가) 표시된 사진

자동 생성된 설명

현재 thread의 recent\_cpu value, load average의 100을 곱한 것을 return해주는 함수이다. 계산하는데는 fixed-point arithmetic을 사용해서 100을 곱한 값에 (1<<F\_P)을 나눠주었다.

* calculate\_all

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

src/devices/timer.c에서 timer\_interrupt함수에서 timer\_ticks()%TIMER\_FREQ가 0이라면 check를 1로 보내고 timer\_ticks()%4가 0이면 check를 0으로 보낸다. calculate\_all함수에서 ready중인 list들인 ready\_list의 size를 ready\_thread\_count에 저장해놓는다.

check변수가 1이라면 현재 thread가 idle\_thread가 아니면 ready\_thread\_count를 1 증가시킨다. Load\_avg는 식을 통해 구하고 recent\_cpu의 경우는 all\_list를 돌면서 recent\_cpu를 계산해준다. 식은 다음과 같이 정해져있다.



check변수가 0인 경우는 all list, 모든 리스트에 있는 thread들을 확인해서 우선순위 계산을 해준다. 계산한 우선순위가 PRI\_MAX보다 크면 그 우선순위를 PRI\_MAX, PRI\_MIN보다 낮으면 그 우선순위를 PRI\_MIN으로 설정한다. 우선순위는 다음과 같은 식을 사용한다.



* 개발 중 발생한 이슈 및 해결 방법

개발 중에서 float연산을 사용하는 load\_avg나 recent\_cpu 계산 시에 기존의 floating point arithmetic을 사용하지 못하기 때문에 fixed-point arithmetic을 수행할 시 기존에 선언한 F\_P 14 즉, 14bit를 사용해서 계산하는데 각각의 경우에 대해서 2^14를 곱하는 것을 적용하지 못해서 결과가 제대로 나오지 못했는데 이를 해결했다.

* 1. **시험 및 평가 내용**

<make check>

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

<priority-lifo>

16개의 thread를 만드는데 lifo이기 때문에 thread가 늦게 만들어질수록 우선순위가 높게 설정된다. 16개의 thread들이 PRI\_DEFAULT(31)로 설정되고 이 뒤로 생성된 thread들은 앞의 16개의 thread보다 높은 우선순위를 가지기 때문에 순차적으로 실행되는 것을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명