**Pintos Project 3: Threads**

담당 교수 : 문의현

학번 / 이름 : 20181134 진건우

개발 기간 : 11/15 ~ 12/05

1. **개발 목표**

이번 프로젝트에서는 Round Robin 형식으로 작동하던 thread의 ready list를 priority에 따라 작동하도록 변경한다. 또한, 이를 더욱 효율적으로 작동시키기 위해 alarm clock과 aging을 추가적으로 구현한다.

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

기존의 thread sleeping은 정해진 시간이 지날 때까지 지속적으로 thread\_yield()를 실행하면서 ready와 running 상태를 반복해야 했으므로 비효율적이었다. 그러나 이를 대신해 thread\_sleep()에서 thread의 상태를 blocked로 바꾸고 시간이 지날 때까지 두면 CPU가 legal한, 올바른 thread만 선택하여 작동할 수 있게 된다.

* 1. ***Priority Scheduling***

기존의 ready\_list는 Round Robin 형식을 채택하여 작동했다. 그러나 thread 구조체에는 분명히 priority가 멤버로써 존재하므로 priority는 계속해서 무시된 채 FIFO 형태로 ready\_list에 추가된 셈이다. 따라서 priority scheduling을 구현하여 priority에 맞게 thread를 처리할 수 있게될 것이다.

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

구현하지 못했습니다.

* 1. **개발 내용**

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

thread의 상태를 blocked로 변경하기 이전에 slepping\_list라는 queue를 따로 생성하여 sleep 상태로 변경될 thread를 저장했다. 또한 thread에 wakeUpTime 멤버를 추가하여 해당 thread에게 할당된 sleeping time을 저장해두었다. 따라서 매 틱마다 호출되는 timer\_interrupt()에서 sleeping\_list queue를 확인하여 해당될 경우 queue에서는 pop하고 thread를 unblock하는 방식으로 쓰레드를 깨울 수 있다.

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

우선 ready list에서의 정렬이 먼저다. thread\_create()에서 thread를 생성하고 마지막에 unblock한다. 기존의 방식에서는 thread\_unblock()에서 thread를 FIFO 형식으로 push back했다. 그러나 priority를 기준으로 ordered insert하면서 새로 들어온 thread가 priority가 가장 높을 경우 ready\_list의 맨 앞에 자리잡게 된다. 이후 current thread와 ready\_list 맨 앞의 thread를 비교하여 current thread가 더 낮은 priority를 가졌을 경우 thread\_yield()를 호출하여 preemption을 발생시킨다.

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

구현하지 못했습니다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

1st Week : 추진 계획 작성 및 Alarm Clock 구현

2nd Week : Priority Scheduling 구현

3rd Week : Priority Aging 및 BSD scheduling 구현

* 1. **개발 방법**

***1) Alarm Clock***

**>>> int64\_t wakeUpTime : threads/thread.h**

thread마다 sleep 상태로 접어들 때 할당될 sleeping time을 저장하기 위해 threads/thread.h의 thread 구조체에 추가

**>>> struct list sleeping\_list : threads/thread.c**

thread를 sleep으로 바꿀 때 ready가 아닌 blocked로 변경하기 위해 새롭게 sleeping\_list를 생성했다.

**>>> timer\_sleep() : devices/timer.c**

기존에 thread\_yield()를 지속적으로 호출하는 방식을 제거하고 thread.c에 새로 구현한 thread\_sleep(start + ticks)을 호출하도록 변경했다.

**>>> void thread\_sleep(int64\_t ticks) : threads/thread.c**

전체적으로 thread\_yield()의 형태를 비슷하게 띄고 있다. 다만 함수의 parameter로 받은 ticks를 thread의 wakeUpTime에 저장하는 것과 sleeping\_list queue에 thread를 저장한 후 상태를 ready가 아닌 blocked로 변경하는 것에 차이가 있다.

**>>> timer\_interrupt() : devices/timer.c**

해당 함수가 매 틱마다 호출된다는 점을 활용하여 threads/thread.c에 새로 구현한 thread\_wakeup(ticks)를 호출하도록 추가했다.

**>>> void thread\_wakeup(int64\_t ticks) : threads/thread.c**

thread\_sleep()에서 sleep 상태에 들어갈 thread를 sleeping\_list에 추가하였다. 그리고 이러한 thread들은 자신의 wake up time이 지났을 경우 sleeping\_list에서 제거한 후 unblock 해야 한다. 이와 같은 목적을 위해 해당 함수를 구현했으며, sleeping\_list의 맨 앞 요소만 복사하여 ticks보다 적은지 (이미 wake up time이 지났는지) 확인한다. (이는 timer\_sleep()에서 thread를 sleeping\_list에 추가할 때 push back이 아닌 wake up time 기준으로 ordered\_list에 추가함으로써 가능하다) 만약 그럴 경우, 해당 thread를 unblock해준다.

***2) Priority Scheduling***

**>>> thread\_create() : threads/thread.c**

thread\_unblock() 이후 ready\_list에서 current thread보다 priority가 더 높은 thread가 CPU를 빼앗게 하기 위해 preemption\_test() 함수를 새로 구현하여 호출했다.

**>>> thread\_unblock() : threads/thread.c**

ready\_list()에 해당 thread를 추가할 때 push back이 아닌, priority를 기준으로 ordered insert하기 위해 list\_insert\_ordered(&ready\_list, &t->elem, &priority\_cmp, NULL)을 호출했다.

**>>> bool priority\_cmp(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED) : threads/thread.c**

thread의 priority를 기준으로 두 개의 thread를 비교하여 정렬하는 데에 활용하는 함수이다.

**>>> void preemption\_test(void) : threads/thread.c**

priority를 기준으로 내림차순 정렬되어 있을 ready\_list에 대해 첫 thread와 current thread 간 priority를 비교하여 current thread가 더 낮을 경우 thread\_yield()를 통해 CPU를 양보한다.

**>>> void thread\_yield() : threads/thread.c**

이 함수는 current thread가 ready\_list의 thread에게 CPU를 양보하게 하는 함수이다. 따라서 current thread가 ready\_list에 다시 삽입되는 과정을 포함하는데, 이 때 push back 과정을 통해 ready\_list에 추가되던 것을 ordered insert 방식으로 변경하였다.

**>>> thread\_set\_priority() : threads/thread.c**

이 함수는 current thread의 priority를 파라미터로 받은 new\_priority로 변경한다. 즉 priority의 변경이 있으므로 ready\_list 내 thread와의 비교를 통해 preemption을 결정하기 위해 preemption\_test()를 호출하도록 추가했다.

***3) Priority-sema***

**>>> sema\_down() : threads/synch.c**

기존에 semaphore를 thread가 가져가는 순서는 FIFO 방식으로 진행되었다. 그러나 이는 2)에서 구현한 priority와 어긋나므로 priority-sema 테스트를 통과하지 못했다. 따라서 sema\_down()에서 push back이 아닌, priority를 기준으로 비교하여 ordered insert 방식으로 변경하여 waiters 리스트에서 thread들이 priority 순으로 정렬되도록 하였다. (list\_insert\_ordered(&sema->waiters, &thread\_current()->elem, &priority\_cmp, NULL))

**>>> sema\_up() : threads/synch.c**

sema\_down()에서 waiters 리스트에 priority 순으로 thread들을 정렬시켜두었기 때문에 up을 통해 다른 thread에게 semaphore를 넘길 때 waiters의 맨 앞 thread들을 넘기도록 pop front 하였다.

***4) Priority-aging***

**>>> fixed-point.c : threads**

aging을 구현하기 위해 priority를 일정 시간마다 변경해줘야 했고, 이를 위해 필연적으로 고정소수점을 사용하는 핀토스의 방식에서 부동소수점을 사용하는 방식을 구현해야 했다. 다행히 매뉴얼에 이에 대한 연산이 상세하고 간단하게 나와 있어서 이에 맞춰서 위 소스파일에 함수들을 구현했다.

**>>> int conv\_int\_to\_fixed(int n) : threads/fixed-point.c**

정수를 고정소수점 실수로 전환하여 표현한다.

**>>> int conv\_fixed\_to\_int(int x) : threads/fixed-point.c**

rounding toward zero 방식으로 고정소수점 실수를 정수로 전환하여 표현한다.

**>>> int conv\_fixed\_to\_int\_nearest(int x) : threads/fixed-point.c**

rounding to nearest 방식으로 고정소수점 실수를 정수로 전환하여 표현한다.

**>>> int add\_fixed\_to\_fixed(int x, int y) : threads/fixed-point.c**

고정소수점 실수끼리 더한다.

**>>> int sub\_fixed\_to\_fixed(int x, int y) : threads/fixed-point.c**

고정소수점 실수끼리 뺀다.

**>>> int add\_fixed\_to\_int(int x, int n) : threads/fixed-point.c**

고정소수점으로 표현한 정수를 고정소수점 실수에 더한다.

**>>> int sub\_fixed\_to\_int(int x, int n) : threads/fixed-point.c**

고정소수점으로 표현한 정수를 고정소수점 실수에서 뺀다.

**>>> int mul\_fixed\_to\_fixed(int x, int y) : threads/fixed-point.c**

고정소수점 실수끼리 더한다.

**>>> int mul\_fixed\_to\_int(int x, int n) : threads/fixed-point.c**

고정소수점 실수에 정수를 곱한다.

**>>> int div\_fixed\_to\_fixed(int x, int y) : threads/fixed-point.c**

고정소수점 실수끼리 나눈다.

**>>> int div\_fixed\_to\_int(int x, int n) : threads/fixed-point.c**

고정소수점 실수에서 정수를 나눈다.

**>>> int recent\_cpu, int nice : threads/thread.h**

priority 연산에 필요한 nice, recent\_cpu 값들을 thread 구조체에 멤버로서 선언했다.

**>>> int load\_avg : threads/thread.c**

Priority 연산에 필요한 load\_av/g 값을 thread.c에서 전역 변수로 선언했다.

**>>> NICE\_DEFAULT, RECENT\_CPU\_DEFAULT, LOAD\_AVG\_DEFAULT : threads/thread.h**

priority 연산에 필요한 nice, recent\_cpu, load\_avg 값들을 초기화하기 위해 thread.h에 명세서에 따른 각 default 값들을 명시해두었다.

**>>> thread\_init() : threads/thread.c**

Nice, recent\_cpu, load\_avg 값들을 thread.h에 명시한 디폴트 값으로 초기화한다.

**>>> init\_thread() : threads/thread.c**

thread\_init()과 달리 thread 전체 시스템이 아닌, thread 하나를 초기화하는 함수이기 때문에, 디폴트 값으로 세팅되는 것이 아니라 부모 thread의 값들을 상속한다.

**>>> int thread\_get\_load\_avg(void) : threads/thread.c**

thread.c 소스파일에 이미 존재하는 load\_avg 값을 return하는 함수이다. 명세서에 나온대로 100을 곱해서 rounding to nearest 방식으로 고정소수점 실수 값을 정수로 변환하여 return한다.

**>>> void thread\_update\_load\_avg(void) : threads/thread.c**

ready thread의 수를 먼저 구해야 한다. ready\_list의 크기에 current thread가 idle이 아닐 경우 1을 더한 값을 ready thread로 설정하고, 명세서에 나온 수식을 그대로 적용하여 load\_avg를 update하는 함수이다.

**>>> int thread\_get\_recent\_cpu(void) : threads/thread.c**

load\_avg를 구하는 방식과 동일하게 recent\_cpu를 구한다. 다만 thread마다 다른 값을 가지므로 current thread의 멤버 recent\_cpu에 추가했다.

**>>> void thread\_update\_recent\_cpu(void) : threads/thread.c**

load\_avg와 다르게 recent\_cpu는 모든 thread마다 고유하게 가지는 값이다. 따라서 ready를 포함한 모든 상태의 thread를 포함하는 all\_list를 for loop로 돌면서 각 thread의 recent\_cpu 값을 명세서에 나온 수식대로 update 해주었다.

**>>> void thread\_update\_priority(void) : threads/thread.c**

priority 역시 모든 thread마다 고유하게 가지는 값이다. 따라서 all\_list를 for loop로 돌면서 각 thread의 priority 값을 명세서에 나온 수식대로 update 해주었다. 다만 최솟값과 최댓값이 상수로 정해져 있기 때문에 범위를 벗어나지 못하도록 if 문으로 묶어주었다.

**>>> timer\_interrupt() : devices/timer.c**

명세서에서 도움을 준 대로 thread\_prior\_aging이라는 bool 변수를 통해 aging을 요청하는 명령어가 들어왔을 때 priority, recent\_cpu, load\_avg를 update해주어야 한다. 그러나 시간의 단위가 서로 다르다는 점을 감안해야 한다. recent\_cpu는 매틱마다 매 틱마다 값을 늘려줘야 한다. priority는 4틱마다 값을 update 해야 한다. recent\_cpu와 load\_avg는 1초마다 값을 update 해야 한다. timer\_interrupt는 틱을 단위로 호출되기 때문에 초를 단위로 변경해야 할 경우, TIMER\_FREQ라는 100틱(= 1초)을 한 단위로 가지는 상수를 활용했다.

1. **연구 결과**
   1. **Flow Chart**

***1) Alarm Clock***

>>> thread 구조체에 wakeUpTime 멤버 추가 및 sleeping\_list 선언

>>> timer\_sleep()에서 while 문과 thread\_yield 삭제하고 thread\_sleep() 구현해서 호출

>>> sleeping thread를 깨우기 위해 thread\_wakeup() 구현하고 timer\_interrupt()에서 호출

***2) Priority Scheduling***

>>> priority를 기준으로 list\_insert\_ordered()를 호출하기 위해 priority\_cmp() 구현

>>> priority를 기준으로 preemption 가능 여부를 검사하기 위해 preemption\_test() 구현

>>> thread가 새로 생성되거나 ready\_list에 추가되는 경우, priority가 변경되는 경우에 대해 push back에서 ordered insert로 변경

>>> priority-sema 테스트를 통과하기 위해 sema\_down()과 sema\_up()을 priority 기준의 list\_insert\_ordered()로 수정

***3) Priority Aging***

>>> fixed-point.c 소스파일 생성해서 고정소수점 관련 연산 함수들 구현

>>> load\_avg, nice, recent\_cpu 선언

>>> 각 변수들 구하는 함수, update하는 함수 thread.c에 구현

>>> timer\_interrupt()에서 각 변수들 조건에 맞춰 값 변경하는 코드 구현 및 update 함수 호출

***4) BSD Scheduling***

>>> 구현하지 못했습니다.

* 1. **제작 내용**

**\*\*\* 구현 내용과 알고리즘에 대해 “개발 내용” 섹션에서 함수 별로 모두 설명하여 중복되는 것 같아 이 섹션에서는 개발 중 있었던 이슈를 위주로 설명하겠습니다.**

***1) Alarm Clock의 sleeping list***

원래 구현했던 방식은 sleeping\_list에 추가할 때에는 timer\_sleep()에 의해 호출된 순서였다. 따라서 sleep 이후 시간이 지나 깨워줘야 할 때마다 sleeping\_list를 모두 돌아야 했고, 그 과정에서 테스트를 통과하지 못했다. 그런데 이번 프로젝트의 중요 과제 중 하나가 priority를 기준으로 thread를 list에 저장하는 것이었고, 나는 priority가 아닌 wake up time을 기준으로 sleeping\_list를 활용하기로 전략을 바꾸었다. Sleeping\_list에 wake up time이 가장 짧은 thread가 들어있다면 리스트를 전부 돌 필요 없이 맨 앞 요소만 체크하면 함수 호출 여부를 알 수 있었기 때문이다. 이러한 방식 수정으로 테스트를 통과할 수 있었다.

***2) fixed-point.c 소스 코드 추가***

사실 이전의 프로젝트에서는 새로운 소스 코드를 추가할 일이 없어 처음 시도해보는 방식이었다. 역시나 make 과정에서 문제가 있었고, 1차적으로 생각한 것은 소스 코드가 존재하는 디렉토리의 Makefile에 접근하여 수정하는 것이었으나 수정 이후에도 여전히 컴파일을 통과하지 못했다. 그러나 다행히 사이버캠퍼스의 질의응답 란에서 방법을 알게 되어 src 디렉토리의 Makefile.build를 수정하여 컴파일을 성공시킬 수 있었다.

* 1. **시험 및 평가 내용**

**1) priority-lifo.c 및 테스트 결과**

**텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명**

**2) make check 수행 결과**

**\*\*\* mlfqs 테스트들을 통과하지 못해 시간이 지연되는 관계로 priority-fifo, priority-preempt, priority-sema, priority-aging 테스트들에 대해서는 개별 make 명령어를 통해 확인했습니다.**

**텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명**