**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20181693 / 최원빈

개발 기간 : 2022.10.28 ~ 2022.11.12

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

이번 프로젝트에서는 첫번째 목표는 현재 busy waiting 기법을 기반으로 구현된 pintos에서 sleep/wake up 기반의 alarm clock을 구현하는 것이다. 이후 현재 round-robin으로 구현된 pintos scheduler를 priority를 고려하여 scheduling 하도록 수정한다. 마지막으로 각 priority마다 자신만의 ready queue를 갖는 advanced scheduler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

현재 핀토스의 devices/timer.c의 timer\_sleep() 함수를 보면 busy waiting 기법을 기반으로 구현되어 있다. 일정 시간(tick)만큼 loop를 돌면서 thread\_yield() 함수로 cpu를 양보하고, thread를 ready\_list에 삽입한다. 즉 thread의 상태가 running state와 ready state를 반복하고 있고 이는 매우 비효율적이다. 이를 해결하기 위해 thread가 잠이 들면 ready state가 아닌 block state로 두어 깨어나기 전까지 scheduling에 포함되지 않게 하고, 깨어날 시간이 되었을 때 ready state로 바꿔준다. 이러한 기법을 잘 구현한다면 이전에 있던 심각한 오버헤드를 줄일 수 있다.

* 1. Priority Scheduling

현재 핀토스는 우선순위에 대한 고려 없이 thread를 새로 ready\_list에 넣을 때 항상 맨 뒤쪽에 넣고 있다. 또 ready\_list에서 thread를 가져올 때는 항상 앞에서 가져온다. 이는 간단하지만 우선순위는 전혀 고려하지 않아 제대로 된 스케쥴링이라 보기 어렵다. 따라서 이를 고쳐주어 ready\_list에 thread를 push할 때 priority 순서에 맞춰 push하여(즉 ready\_list를 정렬된 상태로 유지), pop을 할 때 자연스럽게 가장 priority가 높은 thread를 가져오도록 하게 한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

이전 단계에서 구현한 priority scheduler는 오직 한가지 요소 priority만을 기준으로 schedule하여 priority가 낮은 thread들은 cpu를 점유하기가 매우 어려운 상태이다. 이는 평균적인 response time을 높일 위험이 있으므로 수정할 필요가 있다. 따라서 조금 더 개선된 scheduler를 구현할 필요가 있는데 여러 방식이 있겠지만 이번 프로젝트에서는 priority를 실시간으로 조절하는 muli-level feedback queue scheduling 방식을 도입한다. 앞선 단계에서 이미 priority에 따라 ready\_list를 정렬해 먼저 실행될 수 있게 구현하였으므로, 적절하게 priority만 조절한다면 이번 문제는 충분히 해결할 수 있다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Blocked 상태의 thread를 관리해줄 sleep\_list를 새로 선언한다. Thread 구조체에는 깨어날 시간(thick)을 저장해줄 wakeup\_time이란 멤버를 추가한다. 이후 timer\_sleep() 함수에서 thread를 재울 때 wakeup\_time에 따라 먼저 깨어날 순서대로 sleep\_list에 thread를 넣어준다. 이후 timer\_interrupt() 함수가 호출될 때마다 thread\_wakeup이라는 함수가 호출되게 하고 이 함수에서 현재 시간(thick) 같거나 작은 wakeup\_time을 가진(즉 깨어날 시간이 지난) thread들을 unblcok 해주어 ready\_lsit에 들어가게 한다.

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

Ready list에 running thread보다 더 높은 priority를 가진 thread가 들어올 경우 thread\_yield()를 호출하여 running thread를 교체해줘야 한다. 마찬가지로 running thread의 priority가 수정되어 더 작아질 때도 thread\_yield()를 호출해 더 높은 priority의 thread가 ready list에 있으면 교체해준다.

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

Nice: 각 thread 별로 가지고 있는 고유값이다. 이 값이 클수록 thread가 멋있어져 다른 thread에게 cpu time을 양보하는 정도가 커진다. 즉 priority가 낮아진다.

Recent\_cpu: 역시 각 thread 별로 가지고 있는 고유값이다. 이 값은 thread가 최근에 얼마나 많은 cpu time을 사용했는지를 나타낸다. 따라서 recent\_cpu가 크면 최근에 덤낳은 cpu를 사용했음으로 priority가 낮아진다.

Load\_avg: 전역으로 선언한 변수이다. 1분 동안 수행 가능한 thread의 평균 개수를 나타낸다. 이 값이 크면 recent\_cpu는 느리게 감소하고, 작으면 빠르게 감소한다. 이유는 이 값이 크면 수행 가능한 평균 thread 개수가 많다는 것이고 이 말은 모든 thread가 공평하게 cpu time을 배정받으려면 이미 사용한 thread의 priority가 천천히 증가해야 한다는 것이다. 반대로 평균 개수가 적으면 더 빨리 증가해도 비교적 공평하게 cpu time을 받을 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

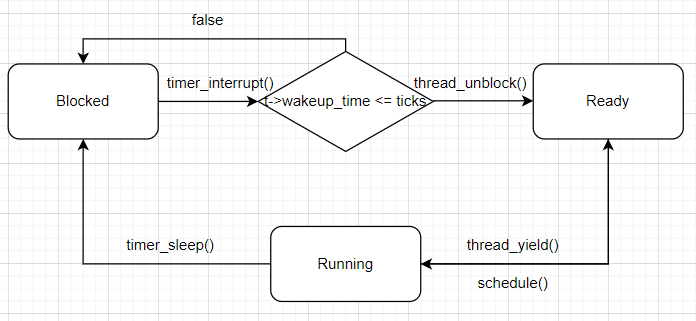
2022.10.28 ~ 2022.10.30: alarm clock 구현

2022.10.31 ~ 2022.11.12: priority scheduling 및 advanced scheduler 구현

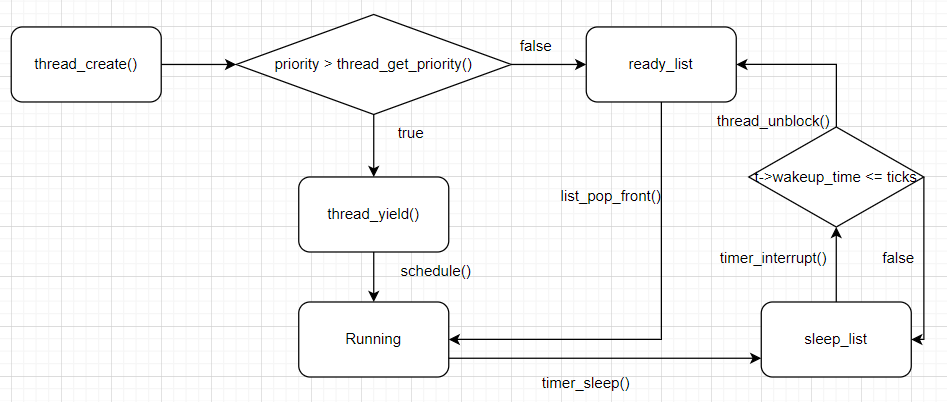
* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수
* Alarm clock: threads/thread.h의 thread 구조체에 wakeup\_time이란 깨어날 시간을 나타내는 멤버를 추가한다. 또 전역을 sleep\_list를 선언하는데 이때 extern 키워드를 붙여 device/timer.c에서 같은 이름을 가진 변수를 공유하게 했다. devices/timer.c의 timer\_sleep() 함수를 수정해 thread를 재울 때 wakeup\_time에 따라 정렬하면서 sleep\_list에 넣는다. 이후 timer\_interrupt()에서 thread\_tick()을 호출하는데 여기서 thread\_wakeup이라는 함수를 호출하게 해 깨어날 시간이 된 thread들을 ready\_list에 넣어준다.
* Priority scheduling: thread를 ready\_list에 넣을 때 priority를 고려할 것이므로 threads/thread.c의 thread\_yield()와 thread\_unblock()을 수정해 thread를 priority에 따라 정렬하면서 ready\_list에 넣는다. 또 thread\_create()를 수정해 thread가 생성될 때 priority도 초기화하게 하고, thread\_set\_priority(), thread\_get\_priority()를 구현한다. 또 synch.c의 sema\_up() 함수를 수정해 sema\_up을 해도 priority 정렬에 어긋나지 않게 해준다. 이후 aging을 위해 init.c에 aging의 경우를 추가해주고, thread.c에 thread\_prior\_aging 변수를 추가해준다. thread\_tick()에서 thread\_aging()을 호출해주게 한다.
* Advanced scheduler: thread 구조체에 nice, recent\_cpu 변수를 추가해준다. Thread.c에 전역으로 load\_avg 변수를 선언해준다. 또 fixed-point read arithmetic을 위한 여러 함수들을 선언하고 구현해준다. 이후 thread\_aging에서 위 변수들을 업데이트 해주기 위한 함수(update\_load\_avg\_and\_recent\_cpu(), update\_priority())들을 호출해주고, 여기서 위 고정 소수점 연산 함수들을 이용하여 값들을 업데이트 해준다. 또 thread\_get\_nice(), thread\_set\_nice() 함수를 구현해주고, 고정 소수점 연산 함수들을 이용해 thread\_get\_load\_avg(), thread\_get\_recent\_cpu() 함수를 구현해준다.

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**

****

Thread\_yield()에서 schedule()을 호출하고 next\_thread\_to\_run()으로 ready\_list에서 첫 번째 thread를 꺼내와 running thread와 switch\_thread()한다.

* 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**

Thread 구조체의 선언부에 각 thread가 깨어날 시간을 나타내는 wakeup\_time이라는 변수를 추가해주었다. 이후 원래 busy waiting 기법으로 구현되어 있던 timer\_sleep() 함수를 호출한 thread의 wakeup\_time에 현재 시간 start에 ticks를 더해준 뒤 저장하고 list\_insert\_ordered() 함수로 sleep\_list에 push해주었다. 이때 먼저 깨어나는 순서대로 정렬해주기 위해 인자로 두 thread 중 첫 번째 thread의 wakeup\_time이 더 작으면 true를 반환해주는 sleep\_list\_ordered() 함수를 작성해 같이 넘겨주었다. 이후 thread\_block()을 호출해 현재 thread의 상태를 THREAD\_BLOCKED로 바꿔준다. 이후 timer\_interrupt()가 호출될 때마다 호출되는 thread\_ticks() 함수 밑에 thread\_wake\_up()을 추가해준다. thread\_wakeup() 함수는 sleep\_list의 thread들을 조사해 만약 해당 thread의 wakeup\_time보다 timer\_ticks()가 크거나 같으면 sleep\_list에서 제거하고 thread\_unblock()을 해준다. thread\_unblcok()에서는 받은 thread를 ready\_list의 맨 뒤에 추가하고 상태를 THREAD\_READY로 바꾼다.

처음에는 그냥 list\_push\_back()을 사용하고 랜덤한 순서로 list를 유지하다가 필요할 때 search하여 가져오는 방식으로 구현하다가, 이 편이 더 비효율적임을 깨닫고 수정하였다. 또 sleep\_list를 thread.c 쪽에 선언하고 그냥 include 하면 될 것이라 생각했지만 오류로 인해 thread.c의 sleep\_list에 extern 키워드를 사용하여 선언하여 해결했다.

* **Priority Scheduling**

우선 ppt에 나와있는 것처럼 init.c에 aging에 해당하는 부분을 추가하고 thread.h와 thread.c에 thread\_prior\_aging 변수를 추가한다. 그리고 이 flag가 true일 경우 thread\_aging()을 실행하도록 thread\_ticks()의 밑쪽에 추가해준다. 이후 thread 구조체의 멤버인 priority를 이용하여 scheduling해야 하는데, 우선 이 값은 thread\_create()의 init\_thread()에서 받은 priority로 초기화해준다. Process\_excute()에서 PRI\_DEFAULT로 넘겨주므로 31로 초기화된다. 이후 thread\_get\_priroty()와 thread\_set\_priority()를 작성해준다. thread\_get\_priroty()는 단순히 thread\_current()의 priority를 반환해주는 함수이고, thread\_set\_priority()는 thread\_current()의 priority를 new\_priority로 바꾸고 이 값이 더 작으면 thread\_yield()를 호출하는 함수다. Thread\_create()의 마지막 부분에서도 새로 생성한 thread의 priority가 현재 trhead의 priority보다 크면 thread\_yield()를 호출하도록 수정해준다. 그후 thread\_yield()와 thread\_unblock()에서 thread를 ready\_list에 추가할 때 priority에 따라 정렬되도록 list\_insert\_ordered()를 사용하도록 바꿔주고, 이때 넘길 인자를 위해 priority\_compare() 함수를 작성해준다. 이 함수는 앞쪽 thread가 뒤쪽 thread보다 priority가 크면 true를 반환하는 함수이다. 마지막으로 synch.c의 sema\_up() 함수에서도 priority를 고려해 락을 획득하기 위해 기다리는 thread 중에서 priority가 큰 thread가 먼저 unblock되도록 수정해준다. 이후 thread\_aging()에서 반복문을 돌면서 모든 리스트의 thread들의 priority를 시간에 따라 증가되도록 작성해준다.

가장 오래 걸렸던 부분이다. 사실 맨 처음에 주어진 피피티를 제대로 보지 않고 시작해 sleep\_list를 다루는 부분을 timer.c에 작성했는데 필요한 함수 호출이 제대로 안 이루어지고, 인자를 넘기고 받는 것에 있어서도 어려움을 겪었다. thread\_unblock()에서 thread\_yield()를 호출하여 생기는 오류 등을 고치려다 점점 코드가 복잡해져 결국 리셋하고 처음부터 다시 시작했다. 막상 차례차례 흐름을 따라가면서 코드를 작성하니 큰 문제없이 마무리할 수 있었다. 1차적으로 작성한 뒤 sema와 aging 부분을 제외하고 테스트케이스가 통과하였고 이후 sema\_up()을 수정하고 sema까지 통과하였다. aging부터는 advanced scheduler와 함께 구현하다시피 했는데, 미리 피피티도 읽고 관련 내용을 찾아본 결과 함께하는 것이 그리 어렵지 않아 그렇게 하였다.

* **Advanced Scheduler**

Thread.h의 thread 구조체의 정의 부분에 recent\_cpu와 nice를 추가하고 inti\_thread를 할 때 0으로 초기화하였다. Thread.c에 load\_avg라는 변수도 전역으로 선언해 thread\_init()에서 0으로 초기화하였다. 그 다음엔 고정 소수점 연산을 위해 꼭 필요한 연산들을 미리 함수로 헤더에 선언하고 thread.c 부분에 코드를 작성하였다. 이때 FRACTION이라는 1을 14번 왼쪽을 쉬프트 연산을 해준 매크로를 선언해 사용하였다. 이후 thread\_get\_nice()는 thread\_current()의 nice 값을 반환하게 했고, thread\_get\_load\_avg()와 thread\_get\_recent\_cput()는 load\_avg와 thread\_current()의 recent\_cpu에 100을 곱해 FRACTION으로 나누어 반환했다. Thread\_set\_nice()에서는 thread\_current()의 nice값을 수정하고 priority 값을 주어진 공식으로 연산해 바뀐 priority에 따라 thread\_yield()를 해주었다. Thread\_set\_priority()에서는 thread\_mlfqs가 true일 경우 실행하지 않고 return 하게 해주었는데 nice와 recent\_cpu, load\_avg 값에 따라 수정되어야 하기 때문이다. 이후 thread\_ticks()에서 thread\_mlfqs가 true일 때도 thread\_aging()을 호출하게 하고 thread\_aging()에서는 timer\_interrupt()가 호출될 때마다 recent\_cpu에 1을 더해주고, TIMER\_FREQ 마다 update\_load\_avg\_and\_recent\_cpu() 함수를 호출하여 recent\_cpu와 load\_avg를 업데이트 해주고, 4tick마다 update\_priority()를 호출하여 priority를 업데이트 해준다. 명세서에 따른 공식을 사용하며 이 공식은 위에서 설명한 세 변수 nice, recent\_cpu, load\_avg가 자신의 정의에 따라 작용하도록 설계된 공식이다. 코드를 한번 리셋하고 작성하면서 한 번에 구현하여 오타로 인해 기대와 다른 값이 생성되어 테스트 케이스를 통과하지 못한 경우를 제외하고는 큰 어려움은 없었다.

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

코드: priority-lifo.c의 코드를 보면 thread가 이름 0 ~ 15순으로 생성된다. 이때 각 priority는 점점 커진다. 즉 나중에 생성된 것이 priority가 높다. 각 thread는 simple\_thread\_func를 수행하게 되는데 우선순위가 높은 뒤에 생성된 것부터 수행한다. simple\_thread\_func은 락을 잡고 output buffer의 position을 증가시키면서 현재 id를 저장해둔다. 이후 락을 해제하고 thread\_yield()를 호출해 cpu\_time을 양보한다. 이때 thread들을 생성하기 전후로 thread\_set\_priority() 함수로 running thread의 priority를 조정해 전부 생성한 다음 한 번에 실행되게 해두었다. 따라서 한 번에 실행되며 모든 실행이 끝난 뒤 output buffer의 내용을 출력한다.

테스트 결과: thread가 이름 0 ~ 15순으로 생성된다. 이때 제일 나중에 생성된 thread의 우선순위가 가장 높다. 이후 simple\_thread\_func을 통해 자신의 id를 기록한 뒤 thread\_yield()를 해주지만 여전히 자신의 우선순위가 가장 높아 다시 실행해준다. 즉 name 15의 thread가 먼저 실행되고 끝난 뒤 14, ··· ,0 순으로 실행된다. 따라서 아래 출력 결과는 아래와 같이 15가 16번(ITER\_CNT) 반복되고, 이후 차례대로 16번씩 반복되게 되고 끝나게 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* make check 수행 결과를 캡처하여 첨부

텍스트이(가) 표시된 사진

자동 생성된 설명