**Pintos Project 3: Threads**

담당 교수 / 분반 : 박성용 교수님

이름 / 학번 : 홍주표 / 20181702

개발 기간 : 2022.10.28 ~ 2022.11.12

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

thread와 관련된 기능을 구현한다. 여기에는 alarm clock, priority scheduling (aging)이 속하고 추가로 advanced scheduler인 BSD 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

기존에 존재하는 thread에서는 timer\_sleep() 함수가 호출되어 thread를 sleep시킨다. 이 때 timer가 만료되지도 않았는데 thread\_yield() 함수가 계속 호출되어 thread의 state를 running과 ready를 반복적으로 변경되게 한다. 이는 효율적이지 못한 sleep 방법이므로, 이보다 효율적인 sleep 방법을 통해 비효율성을 없애도록 한다.

* 1. Priority Scheduling

기존에 존재하는 pintos에서는 round-robin scheduling을 사용했다. 다시 말해서 이느 thread\_yield() 함수나 thread\_unblock() 함수가 호출되면 현재 thread나 unblocked thread가 그 우선순위와 관계없이 ready list의 마지막에 추가된다는 것을 의미한다.

이번 프로젝트에서는 thread의 우선순위를 고려한 scheduler를 적용시키도록 한다. 즉, 현재 thread의 우선 순위보다 높은 우선 순위를 가진 thread가 있다면 해당 thread를 실행시킨다.

여기서 priority scheduling은 낮은 우선순위를 가지는 thread의 경우 starvation을 가진다는 취약점이 있기 때문에 이를 해결하기 위하여 aging도 구현한다. 따라서 시간이 지나면 우선순위를 높여주어 starvation 현상을 없앤다.

* 1. Advanced Scheduler (추가구현을 한 경우)

BSD scheduler를 구현한다. MLFQ (Multi-Level Feedback Queue)나 MLRQ (Multi-Level Ready Queue)를 사용하여 각 우선순위 마다 각자의 ready queue를 가지게 한다. schedule() 함수가 호출되면, thread는 가장 높은 우선순위를 가진 queue를 선택하고, 여기서 각 priority의 ready queue는 round-robin을 따른다. pintos 매뉴얼 (또는 프로젝트 명세서)에 제시된 nice, priority, recent\_cpu, load\_avg 값을 가지고 priority를 계산한다. 이 때, pintos kernel은 부동 소수점 계산을 지원하지 않기 때문에 이를 사용하여 계산하도록 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

blocked 상태의 thread를 관리하는 list를 새로 만들어서 해당 상태의 thread를 모두 저장한다. 그리고 매 tick마다 이 list에 있는 thread를 검사하며 wake\_up\_tick이 된 thread를 list에서 제거하면서 그 상태를 unblocked로 바꾸며 ready queue에 넣어준다. 이로써 blocked 상태의 thread를 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 들어오면 thread의 우선 순위 순서에 따라서 높은 우선 순위를 가진 thread가 실행되어야 한다. 이 때, running thread를 ready 상태로 바꾸고 ready list에 우선 순위 순서에 맞춰 정렬되도록 running thread를 넣는다. 새로 들어온 thread도 마찬가지로 우선 순위에 맞춰 ready list에 넣는다. 그리고 ready list 중 가장 우선 순위가 높은 thread를 실행시킨다.

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

nice, recent\_cpu, load\_avg 값을 계산하여 이를 토대로 priority 값을 계산한다. 처음에는 nice와 recent\_cpu, load\_avg 값은 0으로 초기화된다.

최종적인 priority 계산식은 다음과 같다.

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

여기서 priority 값은 0 (PRI\_MIN)부터 63 (PRI\_MAX) 사이의 값을 가진다.

* nice

-20부터 20 사이의 값을 가지고, nice 값이 클수록 우선 순위는 낮아진다. 또한 자식 thread의 경우에는 부모 thread의 nice 값을 받는다. 이 때, nice 값이 변경된다면 thread의 priority 값을 다시 계산해야 한다.

* recent\_cpu

해당 thread의 CPU time을 추정한다. time interrupt가 일어나면 (idle\_thread가 아닌 경우에), running state의 recent\_cpu 값은 1 씩 증가하고, 모든 thread (running, ready, blocked)의 recent\_cpu 값은 매초 다시 계산된다. 여기서 사용되는 계산식은 다음과 같다.

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

* load\_avg

ready 상태에 있는 thread의 평균 개수를 나타내는 값이다. 이 값은 매초마다 다시 계산된다. 여기서 사용되는 계산식은 다음과 같다.

ready\_threads는 idle thread를 제외하고 ready state나 running state에 있는 thread의 개수를 의미한다.

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

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

22.10.28 ~ 22.10.29: Alarm Clock 구현

22.10.30 ~ 22.11.04: Priority Scheduling 구현

22.11.05 ~ 22.11.10: Advanced Scheduler 구현

22.11.11 ~ 22.11.12: 보고서 작성

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  1. 수정해야하는 소스코드
  2. 수정하거나 추가해야 하는 자료구조
  3. 수정하거나 추가해야 하는 함수

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

blocked 상태의 thread를 관리하는 sleep\_list 자료구조를 추가한다. 또한 thread를 sleep하는 timer\_sleep() 함수에서 thread를 언제 깨울 것인지를 저장하는 wakeup\_tick 변수를 thread 구조체에 추가한다. thread를 깨우는 것은 timer\_interrupt() 함수 내부에서 호출되는 thread\_wake\_up() 함수를 통해 깨우도록 한다. 여기서 timer\_interrupt() 함수는 매 tick마다 호출된다.

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

현재 실행되고 있는 running thread보다 높은 우선 순위를 가지는 thread가 들어오는 경우 ready\_list에 우선 순위에 맞춰 추가를 해야 한다. 기존에는 list\_push\_back() 함수가 호출되어 단순히 추가만 되는 상황이기 때문에, list\_insert\_ordered() 함수로 변경하여 순서를 맞춰 추가하도록 변경한다. 이 때, 우선 순위를 비교하기 위해서 priority\_comp 함수를 추가한다. 이로써 ready\_list가 우선 순위에 맞게 관리된다.

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

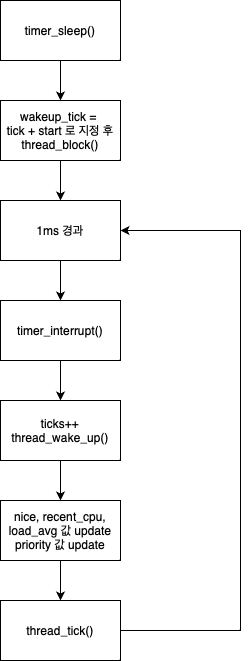
thread 구조체에 recent\_cpu와 nice 변수를 추가한다. 또한 update\_priority() 함수와 update\_recent\_cpu() 함수를 추가한다. nice 값의 update는 thread\_set\_nice() 함수에서, load\_avg 값과 recent\_cpu 값의 update는 update\_recent\_cpu() 함수에서 실행하고, 최종적인 priority 값의 update는 update\_priority() 함수에서 실행한다.

이 때, 단순 사칙 연산을 사용하는 것이 아닌 부동소수점 계산을 실행해야 한다. pintos 매뉴얼에 나와있는 계산 방법에 맞춰 각 값들을 계산할 때 적절히 사용한다.

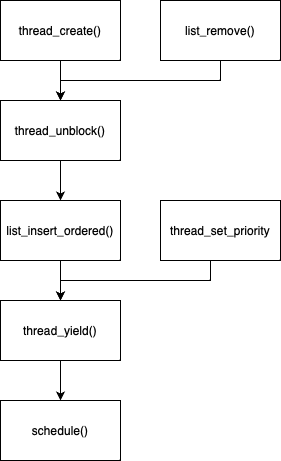
1. **연구 결과**
   1. **Flow Chart**

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

1. Alarm Clock



1. 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

텍스트이(가) 표시된 사진

자동 생성된 설명

우선 thread 구조체에 wakeup\_tick 변수와 추가 구현을 위한 recent\_cpu, nice 변수를 추가한다.

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

텍스트이(가) 표시된 사진

자동 생성된 설명

기존의 busy waiting 코드를 지우고 현재 thread의 wakeup\_tick에 start와 ticks를 합한 값으로 하고, 다음 sleep\_list에 list\_push\_back() 함수를 사용하여 추가한 후, thread\_block() 함수를 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

timer\_interrupt() 함수는 ticks를 증가시키고 thread\_wake\_up() 함수를 호출한다. 따라서 ticks가 변할 때마다 깨워야 할 thread를 찾아서 깨우는 역할을 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

timer\_interrupt() 함수에서 호출되는 thread\_wake\_up() 함수는 위와 같다. sleep\_list를 전부 탐색하여 wakeup\_tick (깨워야할 시간)이 ticks보다 작은 경우 sleep\_list에서 thread를 삭제하고 thread\_unblock() 함수를 호출하여 해당 thread를 unblocked 상태로 변경함으로써 그 thread를 깨운다.

1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.
2. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

텍스트이(가) 표시된 사진

자동 생성된 설명

priority aging을 위한 thread\_prior\_aging 변수와, load\_avg 변수, 그리고 fixed point 연산을 위해서 FR 값을 지정한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그리고 thread\_prior\_aging을 true로 설정한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

또한 thread\_init() 함수에서 wakeup\_tick, nice, recent\_cpu 값을 모두 0으로 초기화한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

sema\_up() 함수에서도 우선 순위와 관련된 처리를 한다. waiters에는 sema되기를 기다리는 thread가 저장되어 있기 때문에, 그 중 가장 큰 우선 순위를 가지고 있는 thread를 찾아서 그 thread를 list\_remove() 함수로 제거한 후, thread\_unblock() 함수를 통해 unblocked 상태로 변경한다. 그리고 나서 thread\_yield() 함수를 통해 다시 scheduling한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

thread\_unblock() 함수와 thread\_yield() 함수에서 기존에 list\_push\_back() 함수를 통해 ready\_list에 cur->elem을 넣는 과정을 list\_insert\_ordered() 함수를 통해 넣는 것으로 변경한다. 이 때 우선 순위 순서에 맞춰 삽입하는 것이 가능하도록 priority\_comp() 함수를 구현한다. 내용은 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

전달받은 두 thread의 우선 순위를 비교하여 true 혹은 false로, 즉 bool type으로 값을 return한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 사진은 thread\_create() 함수의 마지막 부분이다. 여기서 새로 생성되는 thread의 우선 순위가 현재 실행 중인 thread의 우선 순위보다 큰 경우에 thread\_yield() 함수를 통해 다시 scheduling을 하게 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

thread\_set\_priority() 함수는 현재 thread의 우선 순위를 변경하는 작업을 실행한다. 따라서 파라미터로 들어온 new\_priority 값을 현재 thread의 우선 순위로 저장한다. 이 때, 만약 현재 thread의 우선 순위가 새로 들어온 thread의 우선 순위보다 여전히 크다면 thread\_yield() 함수를 호출하고 다시 scheduling 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

thread\_get\_priority() 함수는 현재 thread의 우선 순위를 return하는 역할을 한다.

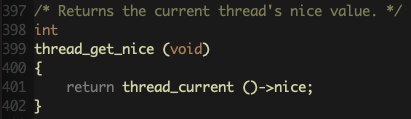
텍스트이(가) 표시된 사진

자동 생성된 설명

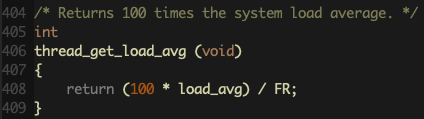
thread\_set\_nice() 함수는 현재 thread의 nice 값을 파라미터로 받은 nice 값으로 설정하는 함수이다. 따라서, 이 함수에서 nice 값을 변경하고 현재 thread의 priority 값도 재설정한다. 그 계산식은 아래와 같다. 아래부터 나오는 계산식은 단순 사칙연산이 아닌 fixed point 연산을 실행해야 하기 때문에 pintos 매뉴얼에 나온 연산방법을 사용한다.

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

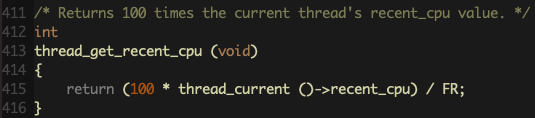
현재 thread의 priority 값을 재설정 한 후, 그 값이 PRI\_MIN과 PRI\_MAX의 범위 안에 있는 값인지 확인해야 한다. 만약 그 값을 벗어나는 값이라면 상황에 맞게 PRI\_MAX, PRI\_MIN으로 재설정한다.



thread\_get\_nice() 함수는 현재 thread의 nice 값을 return하는 역할을 한다.



thread\_get\_load\_avg() 함수는 system load average 값에 100을 곱한 값을 return하는 역할을 한다.



thread\_get\_recent\_cpu() 함수는 현재 thread의 recent\_cpu 값에 100을 곱한 값을 return하는 역할을 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

update\_recent\_cpu() 함수는 load\_avg 값을 다시 계산하고, 이를 토대로 recent\_cpu 값을 다시 계산하는 역할을 한다. 여기서 사용한 계산식은 다음과 같다.

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

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

텍스트이(가) 표시된 사진

자동 생성된 설명

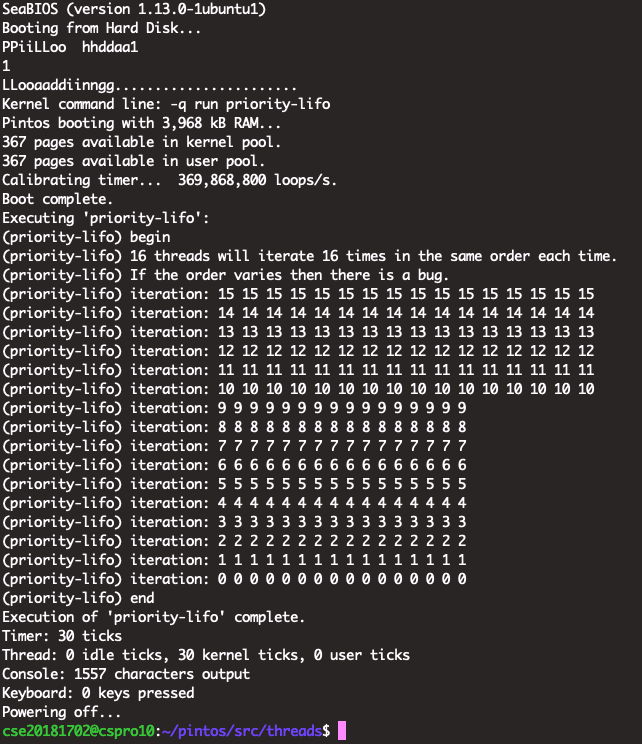
update\_priority() 함수는 위에서 계산한 값들을 기반으로 우선 순위를 update하는 역할을 한다. all\_list의 모든 thread에 대해서 우선 순위를 다시 계산하고 값을 변경한다. 계산식과 최소 최대값에 대한 조건은 thread\_set\_nice() 함수에서 사용한 것과 동일하다.

텍스트이(가) 표시된 사진

자동 생성된 설명

timer\_interrupt() 함수에서 aging과 mlfqs에 대한 처리를 한다. tick이 증가함에 따라 현재 thread의 recent\_cpu 값도 증가시킨다. 그리고 timer\_ticks() 가 TIMER\_FREQ만큼 지났으면 update\_recent\_cpu() 함수를, timer\_ticks() 가 4만큼 지났으면 update\_priority() 함수를 호출한다.

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



priority-lifo.c 코드를 분석해보자. 이 코드에서는 16개 (0부터 15)의 thread를 만든다. 이 때 각각의 우선 순위는 PRI\_DEFAULT + i + 1이다. 그 후, main thread의 우선 순위를 PRI\_DEFAULT로 설정하여 우선 순위가 15인 thread부터 순서대로 실행된다. 이를 계속 반복하더라도 같은 결과가 나오게 된다.

이를 통해, 우선 순위가 높은 thread가 나중에 들어가는 경우에 대해 알 수 있다. 즉, 우선 순위가 높은 thread가 나중에 들어오더라도 먼저 수행을 해야 정상 작동한다고 판단한다는 뜻이다.

위 결과 화면처럼 출력이 된다. 0부터 15까지 차례대로 ready\_list에 들어갔음에도 불구하고, 15의 우선 순위를 가진 thread가 가장 높은 우선 순위를 지녔기 때문에, 가장 먼저 실행되고 종료된다는 것을 볼 수 있다.

* make check 수행 결과를 캡처하여 첨부
* 