**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

조 / 조원 :

개발 기간 : 2023-11-01 ~ 2023-11-04

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.
* Alarm Clock (쓰레드를 일정 시간 재우고, 다시 깨우는 기능을 구현)
* Priority Scheduling (우선순위가 높은 쓰레드가 먼저 CPU를 할당받을 수 있도록 스케쥴링 알고리즘 구현)
* Priority Aging (starvation 문제를 방지하기 위해 쓰레드의 우선순위를 올려주는 테크닉이 필요)
* Advanced Scheduler (MLFQ) (BSD Scheduler 로서, recent\_cpu 와 nice 값을 기준으로 우선순위를 4 ticks 마다 재계산하고, 이에 따라 스케쥴링하는 MLFQ의 일종이다.)

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock – 원래 구현되어 있는 timer\_sleep()의 방식은 busy-waiting 기반의 방식으로서, 매번 깨어날 시간이 되지 않았다면, thread\_yield()를 통해서 RUNNING 상태와 READY 상태를 왔다갔다하도록 구현되어 있다. 하지만 이러한 방식은 계속 두 상태를 오고 가야 하므로, 이러한 쓰레드들의 상태를 관리해야 하는 OS 입장에서는 비효율적일 수 밖에 없다. 따라서, 한번 sleep()이 호출된 쓰레드의 경우, BLOCKED 상태로 만들어주어 재우다가, 깨어날 시간이 다 된 쓰레드들을 한꺼번에 깨워주는 방식으로 구현하는 것이 보다 효율적일 것이다. BLOCKED 상태가 되면, Ready Queue에도 들어가지 않아, 관리하기 편하다는 장점이 있을 것이다.
  2. Priority Scheduling – 현재 핀토스에서 작동하고 있는 ready queue는 라운드 로빈 방식을 채택하고 있어 우선순위는 고려되지 않는다. 각각의 쓰레드들에 우선순위가 부여되고, 이러한 우선순위를 고려해서 우선순위가 높은 쓰레드들을 우선적으로 RUNNING 상태가 되도록 하는 Priority Scheduling 방식을 구현하게 되면, 각 쓰레드들의 특성을 더 잘 반영하여 스케쥴링할 수 있게 된다.
  3. Advanced Scheduler (추가구현을 한 경우) – 매 4 ticks 마다 쓰레드들의 우선순위를 재계산하여 스케쥴링에 반영함으로써, 스케쥴링에 쓰레드들의 특성을 활용할 수 있다. 우선순위를 계산할 때 사용하는 recent\_cpu 변수는 각 쓰레드의 CPU time을 예상하는 변수이므로, 다소 복잡할 수 있지만 보다 효율적인 스케쥴러를 구현할 수 있다.
  4. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Alarm Clock에서 Blocked 상태의 스레드를 깨우기 위해서 timer\_interrupt가 걸릴 때마다, blocked 상태의 스레드들을 확인해보면서, 자신이 깨어날 시간이 지났으면, ready 상태로 바꾸도록 구현해야 한다. 현재 핀토스에서는 Blocked 상태를 갖는 스레드들만 있는 리스트가 없기 때문에 따로 만들어주어 관리하고, timer\_interrupt가 걸릴 때마다 깨어날 시간을 확인 후 만약 지났다면, thread\_unblock() 함수를 호출해 Blocked 상태에서 Ready 상태로 변경하여 스레드들을 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 들어온 경우 Priority scheduling에서는 즉시 더 높은 priority를 가진 thread에게 CPU 소유권을 양도해야 한다.

이를 위해서는 thread\_create() 함수를 주목해보면 되는데, 새로운 thread가 만들어지면, init\_thread()를 통해서 해당 Priority를 가진 쓰레드를 초기화해주게 된다. 따라서, 새로운 쓰레드가 만들어진 이후에, thread\_unblock()을 통해서 Ready queue로 들어가게 되는데, Ready queue를 높은 우선순위부터 내림차순 정렬이 되도록 만들어주면, 가장 앞에 있는 쓰레드와 우선순위를 비교해서 CPU를 계속 점유할지 혹은 양도할지를 정할 수 있다.

또한, BLOCKED 상태의 스레드가 다시 Ready list에 들어올 때도 동일한 동작을 하도록 구현하면 될 것이다.

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

Advanced Scheduler에서 사용하는 Priority 계산식은 다음과 같다.

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

이 때, PRI\_MAX는 쓰레드가 가질 수 있는 최대 우선순위 값인 63이다. 우리가 주목해야 할 요소는 recent\_cpu 변수와 nice 변수이다.

recent\_cpu는 해당 쓰레드가 최근에 사용한 CPU time의 예측치이다.

recent\_cpu 또한 아래와 같은 계산식이 있다.

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

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

ready\_threads는 READY 상태에 있는 스레드들의 개수와 RUNNING 상태에 있는 스레드 1개의 개수를 더한 값이다. (RUNNING 상태에 있는 스레드가 idle\_thread면 1을 더해주면 안 된다.)

load\_avg 값은 시스템이 부팅될 때, 0으로 초기화된다.

recent\_cpu는 부모 스레드로부터 생성된 스레드의 경우 부모 스레드의 recent\_cpu 값을 상속받고, 아닌 경우 0 으로 시작한다. 또한, timer interrupt가 발생할 때마다 현재 RUNNING 상태에 있는 스레드의 recent\_cpu 값을 1 올려준다. (idle\_thread가 아닌 경우)

recent\_cpu와 load\_avg는 매 초 갱신된다.

nice는 -20부터 20까지의 값을 가지며, 높을 수록 priority를 낮추고, 낮을 수록 priority를 높인다.

Priority를 계산하기 위해 필요한 요소들 간에 사칙연산을 해주어야 하는데, 핀토스에서는 부동 소수점 연산을 지원하지 않기 때문에 fixed-point real arithmetic 방식으로 이를 구현해야 한다.

int 형 4바이트 변수에서 Most Significant Bit는 부호를 나타내고, 그 밑에 17 비트는 소수의 정수부를 나타내고, 마지막 14 비트는 소수부를 나타낸다. 이를 활용해서 실수 간 사칙연산을 구현하면 priority 계산을 할 수 있다. 자세한 구현 설명은 아래에서 하도록 하겠다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2023-11-01 – Alarm Clock 구현 / semaphore 및 lock 개념 복습

2023-11-02 – Priority Scheduling 구현

2023-11-03 – Priority Aging 구현 / fixed-point real arithmetic 이해 및 구현

2023-11-04 – Advanced Scheduler 구현 및 트러블 슈팅

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

BLOCKED 상태의 스레드들을 깨우기 위해서는 해당 스레드가 일어나야 하는 시간에 대한 정보를 가지고 있어야 한다. 또한, 일어날 시간이 지난 스레드들을 순회하면서 깨우기 위해 BLOCKED 상태의 스레드들만 있는 리스트를 만들어주어야 한다.

thread.h 의 struct thread 구조체에 int wakeuptime 이라는 변수를 추가한다.

BLOCKED 상태의 스레드, 즉 수면 상태의 스레드만을 담는 blocked\_list를 thread.c에 추가하고, thread\_init() 함수에서 초기화해준다.

또한, BLOCKED 상태의 스레드를 순회하면서 일어날 시간이 지난 스레드들을 thread\_unblock() 해주는 to\_wake\_up()이라는 함수를 thread.c에 정의했다.

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

우선, ready list에 running thread보다 높은 Priority를 가진 thread가 들어오는 경우는 새로운 스레드가 생길 때, 즉 thread\_create() 함수가 호출되었을 때와 BLOCKED 상태의 스레드가 다시 ready list에 들어오는 경우로 나누어볼 수 있다.

따라서, thread\_create()에서 우선순위를 가지는 스레드가 생성되고 나서, 해당 스레드의 priority 와 현재 RUNNING 상태의 스레드의 우선순위를 비교해서, 만약 해당 스레드의 priority가 높은 경우에 preemption이 일어나도록 한다. (thread\_yield()) 이러한 기능을 하는 함수 preemption을 thread.c 에 새롭게 정의하였다.

BLOCKED 상태의 스레드들을 깨우면, 일어난 스레드들은 Ready list에 들어가게 된다. Priority scheduling의 경우 Ready list를 우선순위를 기준으로 내림차순으로 정렬해주면, timer interrupt가 발생할 때, 맨 앞의 스레드의 우선순위와 현재 스레드의 우선순위를 비교해서 thread\_yield()를 시켜줄 수 있다.

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

우선, Advanced Scheduler 구현을 위한 핵심은 priority 계산에 있다. Priority 계산을 위해서는 앞서 설명했듯이 fixed-point real arithmetic을 사용해야 하는데, 핀토스 매뉴얼에서는 fixed-point real arithmetic 구현에 대해 자세하게 설명하고 있다.

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

자동 생성된 설명

우선, fixed-point real arithmetic을 사용하기 위해서는 원래 정수값에 (1 << q), 즉 (1 << 14) 만큼 곱해주어야 한다. 그 이유는 아래 14개 비트는 소수점 자리를 나타내는 데에 쓰이기 때문이다. 다시 원래의 정수값으로 변환하려면 (1 << 14)만큼 나누어 주면 된다. 하지만, 이번 과제에서 요구한 것은 round to nearest 방식이므로, (x + (1 << 14) / 2) / (1 << 14)와 같은 방식으로 변환해야 한다. 또한, 실수 끼리의 덧셈, 뺄셈은 다소 직관적이지만, 곱셈과 나눗셈을 할 때에는 정수 오버플로우에 주의해서 (int64\_t)로 변환해준 후에 계산을 해야 한다. 곱셈의 경우에는 두 실수의 곱으로 인해 소수점이 왼쪽으로 14 비트만큼 더 이동하기 때문에 (1 << 14) 으로 나누어주는 모습이다. 나눗셈의 경우 반대이다. 하지만 (1 << 14)을 곱하는 과정에서 오버플로우가 일어날 수 있으므로 (int64\_t)로 변환해준 모습이다.

이번 프로젝트에서 priority 계산을 위해 따로 정의해준 함수들의 리스트이다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

float\_to\_int는 round to nearest 방식으로 실수를 다시 정수값으로 바꾸어주는 함수이다.

int\_to\_float는 정수를 fixed-point 형으로 변환해주는 함수이다.

add\_floats는 두 실수를 더해주는 함수이다.

sub\_floats는 두 실수를 빼주는 함수이다.

mul\_floats는 두 실수를 곱해주는 함수이다.

div\_float는 두 실수를 나누어주는 함수이다.

위 여섯가지 함수를 이용해서 priority 계산을 할 수 있었다.

1. **연구 결과**
   1. **Flow Chart**

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

**텍스트, 스케치, 도표, 그림이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 도표, 친필, 폰트이(가) 표시된 사진

자동 생성된 설명**

* 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

1. **BLOCKED 상태의 스레드 깨우기**

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명**

우선, 앞서 설명했듯이 BLOCKED 상태인 스레드들만을 관리하는 blocked\_list를 관리하는 것이 구현에 있어서 훨씬 편리하다. (sleep 상태에 들어가면 blocked 상태가 되므로, sleep 상태에 있는 스레드들이 있는 리스트라고 생각하면 된다.)

to\_wake\_up 함수가 ticks 값을 인자로 받아서, blocked\_list 전체를 순회하면서 인자로 주어진 ticks가 해당 스레드의 wakeuptime보다 크거나 같으면, blocked\_list에서 해당 스레드를 지워주고, thread\_unblock()을 통해서 ready list에 해당 스레드를 넣는다.

**텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

그렇다면 to\_wake\_up 함수는 언제 호출이 되는지 살펴보자. timer\_interrupt가 호출될 때, ticks를 증가시키는데, 이 때 to\_wake\_up 함수를 사용하여 1 올라간 ticks 값을 넘기게 되면, timer\_interrupt가 걸릴 때마다 깨어날 시간이 지난 스레드들을 ready list에 보낼 수 있게 된다. 이렇게 해서 BLOCKED 상태의 스레드들을 깨울 수 있게 된다.

1. **Ready list에 running thread보다 높은 priority를 가진 스레드가 들어올 경우 priority scheduling의 동작 방식**

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

먼저, ready list에 스레드가 들어오는 경우에 대해서 알아야 한다.

thread\_create() 함수가 실행되어 스레드가 생성될 때, thread\_unblock()을 통해서 ready list에 스레드가 들어가게 된다. 이 때, preemption() 함수를 호출하면 현재 ready list에 들어있는 스레드 중 가장 앞의 스레드의 Priority와 현재 실행 중인 스레드의 priority를 비교 후, 만약 가장 앞의 스레드의 Priority가 더 크면 thread\_yield()를 통해 CPU 를 양도하게 된다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

Priority Scheduler 는 여러 방식으로 구현될 수 있겠지만, 나는 ready list에 해당 스레드를 넣어줄 때, 우선순위를 기준으로 내림차순이 되도록 insert 해주어 항상 ready list가 우선순위 기준 내림차순으로 정렬되어 있는 상태를 만들어주었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

이렇게 thread\_unblock()을 할 때, 해당 스레드를 ready list에 넣어주는데, 원래 수정하기 전에는 list의 맨 뒤로 push\_back 해주었다. 이는 라운드 로빈 방식일 때 그러하다. 하지만, 이번 프로젝트에서 list\_insert\_ordered 함수로 바꾸어주었고, 정렬 기준은 thread\_cmp\_priority 로 구현하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

리스트의 앞의 스레드의 우선순위가 뒤의 스레드의 우선순위보다 크면 원소의 위치를 바꾸지 않게 true를 반환, 뒤바꼈으면 바꿔야 하므로 false를 반환한다. C에서 정렬 함수의 cmp 함수를 줄 때는 바꾸는 상황일 때 false를 반환하도록 해야 한다.

**텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

앞서 timer\_interrupt 함수에서 설명하지 못한 부분은 4 tick 마다 update\_priority를 하여 스레드들 전체의 priority를 재계산하는 부분인데, update\_priority() 함수 내에서도 preemption() 함수를 호출하여 CPU 점유권을 넘겨주도록 하는 구문이 있다.

텍스트, 소프트웨어, 멀티미디어 소프트웨어, 스크린샷이(가) 표시된 사진

자동 생성된 설명

밑에 자세히 보면, premption() 코드와 유사한 부분이 있다. 이 때, intr\_yield\_on\_return()을 한 이유는 interrupt 도 중이기 때문에 thread\_yield() 대신 호출하였다.

1. **Advanced Scheduler 에서의 priority 계산**

**스크린샷, 텍스트이(가) 표시된 사진

자동 생성된 설명**

앞서 설명했던 fixed-point real arithmetic에서 사용할 함수 6가지이다. 사칙연산 함수 4개와 실수에서 정수 변환, 정수에서 실수 변환 함수 2개이다. 이 때, F 변수는 매크로로 정의해둔 상수로서, (1 << 14)의 값을 가진다. 이는 핀토스에서 활용할 소수부의 비트 개수가 14개이기 때문이다.

먼저, priority 계산 함수를 보자.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

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

이 계산식에 의해서 계산이 되고, 위에서 정의한 함수들을 여러 개 겹쳐 사용한 모습을 볼 수 있다. 너무 복잡하지만은 않은 것이 recent\_cpu 값은 이미 실수이므로 변환이 필요 없고, niceness와 같은 정수값은 실수 계산을 위해 int\_to\_float을 불러준 모습이다. 그 외의 연산은 위 식과 동일하다.

그 다음은 recent\_cpu 계산 함수이다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

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

recent\_cpu도 위 계산식을 그대로 대입하여 계산하였다. 이 때, load\_avg 값은 실수이므로 그대로 사용했고, niceness는 int\_to\_float으로 실수로 변환하여 사용하였다.

마지막으로 load\_avg 계산 함수이다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

load\_avg는 시스템 ready list에 있는 스레드들의 개수에 현재 실행 중인 스레드가 idle\_thread가 아니면 1 더한 값을 나타낸다.

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

이 계산식을 사용하여 계산하였고, 59, 60, 1과 같은 정수값은 int\_to\_float으로 실수 형태로 represent 하게 바꾸어 주었고, 나머지 연산은 동일하게 대입하여 계산했다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

이는 핀토스에서 구현한 각각 recent\_cpu, load\_avg, nice 값을 받아오는 함수이다. load\_avg와 recent\_cpu는 100을 곱한 값을 반환하도록 해야 한다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

다음은 thread\_set\_nice 함수이다.

현재 스레드의 niceness 값을 인자로 주어진 nice 값으로 바꾸어주고, 바뀐 값을 바탕으로 priority를 다시 계산해주어야 한다. 이 때, 다시 계산한 priority 값이 PRI\_MAX를 넘으면 PRI\_MAX로 바꾸어 주고, PRI\_MIN보다 작으면 PRI\_MIN으로 바꾸어 주어 예외 처리를 해준다.

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

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

priority-lifo 코드를 보면, 현재 스레드의 우선순위를 PRI\_DEFAULT + 16 + 1로 설정해놓고, 16개의 스레드를 thread\_create()로 생성하게 된다. 이 때, 각 스레드의 우선순위는 PRI\_DEFAULT + i + 1로서, PRI\_DEFAULT + 1 부터 PRI\_DEFAULT + 17 까지 우선순위를 갖게 될 것이다. 이 뜻은 현재 스레드를 생성하는 부모 스레드보다 우선순위가 커지지 않게 하여 CPU 점유가 되지 않도록 하는 것이다. 이렇게 스레드를 다 만든 후에는 밖에 나와서 현재 스레드의 우선순위를 PRI\_DEFAULT로 바꾼다. 이 때, 현재 스레드의 우선순위가 만들어진 스레드의 우선순위보다 낮아져 새로 만들어진 스레드들의 CPU preemption이 일어나게 된다. 이 때, 우선순위가 가장 높은 PRI\_DEFAULT + 17의 우선순위를 가진 스레드가 실행이 될 것이다. 이 스레드는 가장 마지막에 들어온 스레드이고, 이 스레드가 가장 먼저 실행되게 된다. 그래서 lifo 의 의미를 여기서 깨닫게 된다.

그렇게 해서 실행된 결과를 출력하는데, 실행된 스레드의 아이디를 총 16번 출력하는 것을 볼 수 있다. 밑에 출력 결과를 확인해보자.

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

출력 결과를 확인해보면, ID가 15인 스레드, 즉 가장 마지막에 ready list에 들어온 스레드가 가장 먼저 출력되는 것을 볼 수 있다. 잘 출력이 된 것이다.

* make check 수행 결과를 캡처하여 첨부

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명