**Pintos Project 3: Threads**

담당 교수 : 김영재

조 / 조원 : 윤다영

개발 기간 : 2020.11.14 – 2020.12.08

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

이번 프로젝트에서는 threads와 관련된 기능들을 개발한다. 프로젝트의 개괄적인 개발 범위는 threads에서 alarm clock이 기존의 pintos 시스템에서 busy waiting 방식으로 작동했던 것과 달리 더 효율적으로 작동할 수 있도록 하는 것과 기존의 pintos 시스템이 round-robin 형식으로만 스케줄링했던 것과 달리 우선순위를 사용하여 preemption이 가능한 스케줄링을 수행하도록 변경하여 priority scheduling을 구현하는 것, 시간이 흐름에 따라 우선순위가 조정될 수 있도록 aging을 구현하는 것, 마지막으로 추가적인 advanced scheduler를 구현하는 것이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

기존의 pintos 시스템에서는 alarm clock 기능이 busy waiting의 형식으로 구현되어 있다. 이런 busy waiting 방식은 CPU 자원을 낭비하게 만듦으로, 성능 개선을 위해서는 다른 방식으로 alarm clock을 구현해야 한다. 따라서 이번 프로젝트에서 sleep된 thread들을 block된 상태로 관리하는 sleep과 wake-up 방식을 사용하여 다시 구현한다. 이와 같이 구현했을 때 alarm clock 기능의 성능이 더 높아질 것이라고 예상할 수 있다.

* 1. Priority Scheduling

기존의 pintos 시스템에서 scheduler는 우선 순위를 고려하지 않는 round-robin 형식으로만 구현되어 있다. 이번 프로젝트에서는 단순히 round-robin 형식으로만 스케줄링이 수행되는 것이 아니라 각 thread에서 계산되는 우선 순위를 고려하여 ready list안에 현재 CPU를 점유하는 thread보다 우선순위가 더 높은 thread가 있다면 이 thread가 preemption이 가능한 스케줄링 기법을 구현하는 것을 목표로 한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

정해진 시간에 따라 priority를 계속 다시 계산하여 우선순위를 관리하고, 이에 따라 스케줄링이 이루어지게 하는 multi-level feedback queue 형식의 advanced scheduler를 구현하는 것을 목표로 한다. 이 때에 priority를 다시 계산할 때에는 새로운 요소로 각 thread의 nice 값을 계산한다. nice값은 그 값이 높을수록 우선순위를 낮추고, nice값이 낮을수록 우선순위를 높이는 결과를 초래한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

blocked된 상태의 thread를 깨우기 위해서는 먼저 thread 구조체에 깨어나야 하는 시간을 저장하는 필드를 추가한다. 그리고 thread가 sleep될 때마다 이 필드에 깨어나야 하는 시간을 저장한 후, 매 tick마다 sleep list에 있는 thread들을 검사하여 깨어나야 하는 시간이 되었다면 깨워주는 함수를 실행시켜 다시 깨어난 thread들을 ready list에 넣어 준다. 이와 같은 방식을 blocked된 thread들을 제 시간에 깨워줄 수 있다.

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

ready list에 새 thread가 들어올 경우를 모두 고려해서 그 때에 우선 순위가 높은 thread가 ready list에 있다면 preemption이 가능하게 구현해준다. 이를 위해서 먼저, thread가 생성될 때마다 thread를 생성하는 함수에서 새로 생성되는 thread의 우선순위와 현재 running 상태인 thread의 우선순위를 비교한다. 만약 새로 생성되는 thread의 우선순위가 더 높다면 바로 preemption이 될 수 있도록 해준다. 또, 기존에 block 되었던 thread가 unblock될 때마다 ready list에 이 unblock되는 thread가 저장되므로, 이때에 ready list에 우선순위 순으로 정렬되어 thread들이 삽입될 수 있도록 해준다. 또, 마지막으로 현재 thread가 더 높은 우선순위가 높은 thread를 위해 CPU를 양보할 때 실행되는 thread\_yield() 함수에서 역시 이 양보하는 thread가 ready list에 우선순위 순으로 정렬되어 삽입되도록 해준다. 이와 같이 ready list에 있는 thread들을 우선순위 순으로 정렬하여 관리할 경우, 항상 ready list의 가장 첫번째 thread가 ready list에서 가장 우선순위가 큰 thread임을 보장할 수 있으므로 더 효율적인 thread 관리가 가능하다.

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

advanced scheduler에서 priority를 계산하는 공식은 다음과 같다.

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

여기서 priority 계산에 추가적으로 사용되는 변수는 recent\_cpu와 nice이다. 위에서 기술한 것과 같이 nice는 thread에 부여되는 값으로, 따로 추가적인 계산이 이루어지지 않는다. 한편, recent\_cpu는 최근에 해당 thread가 얼마나 CPU time을 소모했는지 나타나는 변수이다. recent\_cpu 역시 별도의 계산하는 공식을 가지고 있다. recent\_cpu의 계산식은 다음과 같다.

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

한편, recent\_cpu를 계산하기 위해 사용되는 load\_avg 역시 recent\_cpu 계산을 위해 추가적으로 사용되는 변수로, recent\_cpu, priority, nice가 thread가 귀속되는 변수인 것과 다르게 load\_avg는 독립적인 전역 변수로 정의되어 사용된다. load\_avg는 최근 1분 동안 수행 가능한 평균적인 프로세스의 개수를 의미하며 load\_avg에 대한 계산 식은 다음과 같다. (여기에서 ready\_threads는 ready list안의 thread 개수를 의미한다.)

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

또, priority를 계산할 때는 소수점 연산을 고려해주어야 하는데, 이는 recent\_cpu, load\_avg는 소수인데 pintos에서는 floating-point arithmetic을 지원하지 않으므로, 이 변수들과 관련된 연산에서는 fixed-point arithmetic의 방식으로 실수 연산을 수행될 수 있도록 fixed-point arithmetic을 구현해주어 priority가 제대로 계산될 수 있도록 구현해주어야 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

|  |  |
| --- | --- |
| 날짜 | 구현 내용 |
| 2020-11-20 | 전체적인 개발 계획 수립 |
| 2020-11-22 | alarm clock 구현 |
| 2020-11-23 | priority scheduling 구현 – 1 |
| 2020-11-25 | priority scheduling 구현 – 2 |
| 2020-11-26 | priority scheduling 구현 – semaphore 관련, aging 구현 |
| 2020-11-27 | 중간 디버깅 |
| 2020-11-28 | advanced scheduling 구현 – fixed-point arithmetic, priority 계산구현 |
| 2020-11-29 | advanced scheduling 구현 – nice 관련 구현 |
| 2020-12-01 | advanced scheduling 구현 확인 및 디버깅 |
| 2020-12-02 | 최종 디버깅 |
| 2020-12-03 | 보고서 작성 |
| 2020-12-04 | 보고서 작성 |
| 2020-12-08 | 마지막 점검 및 과제 제출 |

* 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 깨우는 방법 구현**

blocked된 스레드들이 sleep될 때에 다시 깨어나야 할 시간을 스레드를 다시 깨울 때에 알려줄 수 있도록 src/threads/thread.h의 thread 구조체에 각 스레드들이 깨어나야 하는 시간을 기억하는 int64\_t 형 변수 wakeup\_tick을 추가해주었다. 다음으로 sleep된 스레드들을 일괄적으로 관리하기 위해 src/threads/thread.c에 global list 형태의 자료구조를 갖는 sleep\_list를 추가하였다. 새로 sleep\_list를 추가해주었으므로, threading system이 시작될 때 sleep\_list를 초기화 해주었다. 다음으로 src/threads/thread.c에 새롭게 thread가 sleep될 때의 작업을 처리하는 함수 thread\_sleep을 추가해주었다. 다음으로 src/threads/thread.c에서 thread\_awake 함수를 정의하여 sleep 상태였던 스레드가 깨어날 때의 작업을 처리하도록 해주었다.

1. **Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling 구현**

src/threads/thread.c에서 선언된 ready\_list를 우선순위 순으로 좀 더 효율적으로 관리하기 위해 thread\_unblock 함수에서 주어진 스레드가 unblock되거나 thread\_yield 함수에서 preemption이 일어나 원래 실행 중이던 스레드가 ready\_list에 추가될 때 우선순위 순으로 삽입이 되도록 구현해주었다. 다음으로 앞에서 설명한 것과 같이 ready\_list에 새 thread가 들어오는 경우들을 고려해주어야 하는데, 먼저 thread가 생성되는 thread\_create 함수에서 새로 생성되는 스레드가 ready\_list에 삽입될 때에 우선 순위에 의한 스케줄링이 이루어지도록 하였다. 다음으로, thread\_set\_priority 함수를 수정하여 이때에도 마찬가지로 우선 순위에 의한 스케줄링이 이루어질 수 있도록 수정하였다. 이를 위해 src/threads/thread.c에서 스레드들의 우선 순위 순으로 선점이 허용되는 스케줄링을 해주는 priority\_preempt() 함수를 새로 작성하였다.

1. **Advanced Scheduler에서 priority 계산 구현**

위에서 설명한 것과 같이 priority의 계산식을 구현하기 위해 가장 먼저 fixed-point arithmetic을 구현해주었다. 이를 위해 src/threads 디렉토리에 fixed\_point.h 파일을 새로 생성하여 이 파일에서 fixed-point 형식의 숫자 표현 방식을 포함하는 연산들을 구현해주었다. 다음으로 load\_avg는 기존에 구현되어 있지 않았으므로 src/threads/thread.c에서 int load\_avg를 전역 변수로 선언해주고, thread\_start 함수에 load\_avg를 초기화 시키는 코드를 추가해주었다. 다음으로, priority, recent\_cpu, load\_avg를 계산하는 함수를 새로 thread.c에서 정의해주었다. 이 때에 계산식은 위에서 작성한 계산식을 사용하였고, 이 때에 앞서 구현한 fixed\_point.h의 함수들을 사용하여 fixed-point arithmetic을 구현할 수 있었다. 또, advanced scheduler에서는 일정한 시간마다 load\_avg, recent\_cpu, priority의 값의 계산 값이 갱신되어야 하므로, 이에 대한 함수를 위에서 구현한 세 load\_avg, recent\_cpu, priority를 계산하는 함수를 사용하여 추가적으로 구현해주었다. 또, 기존에는 선언만 되어있던 thread\_set\_nice, thread\_get\_nice, thread\_get\_load\_avg, thread\_get\_recent\_cpu 함수를 구현해주었다. 마지막으로, pintos에서 매 tick마다 호출되는 src/threads/thread.c의 thread\_tick 함수에서 thread\_mlfqs flag가 true일 때, advanced scheduler에 따른 스케줄링이 수행되도록 구현해주었다.

1. **연구 결과**
   1. **Flow Chart**

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

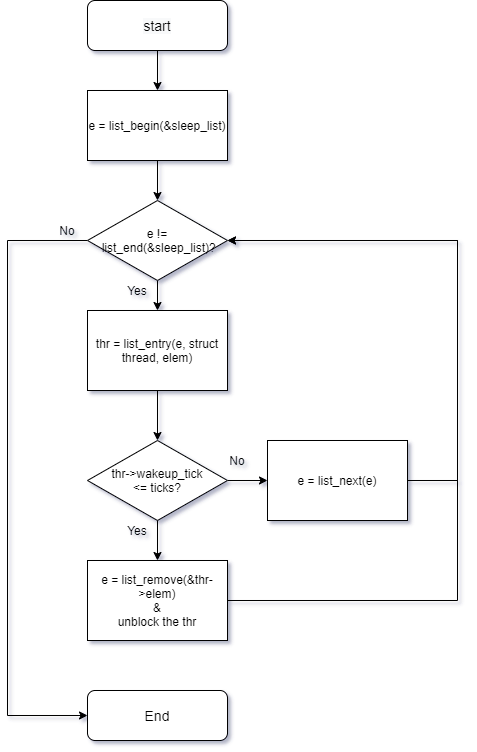
**1. Blocked 상태의 스레드를 깨우는 방법 구현**

**- thread\_sleep() 함수**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**- thread\_awake() 함수**

****

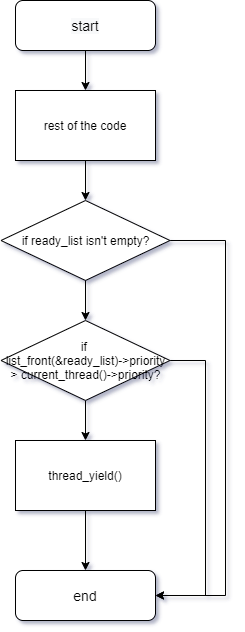
**2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling 구현**

**- thread\_unblock(), thread\_yield() logic**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**- thread\_create(), thread\_set\_priority() logic**



* 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 스레드를 깨우는 방법 구현**

thread.c에서 새로운 자료구조 sleep\_list가 추가되었으므로, src/threads/thread.c의 thread\_init()함수에서 sleep\_list가 초기화될 수 있게 list\_init(&sleep\_list) 코드를 추가해주었다. 다음으로 src/threads/thread.c의 thread\_sleep() 함수에서 현재 스레드에 주어진 ticks를 wakeup\_tick에 저장하고, sleep\_list에 해당 스레드를 삽입한 후 해당 스레드를 block해주고, 다시 schedule() 함수를 호출하도록 구현하였다. 그리고 기존의 src/devices/timer.c의 time\_sleep 함수에서는 busy waiting 방식으로 스레드의 sleep을 구현하게 했는데, 이 코드를 삭제하고 앞에서 정의한 thread\_sleep 함수를 호출하였다. 다음으로, sleep하고 있던 스레드들을 깨워주는 함수 thread\_awake를 구현하였는데, thread\_awake가 모든 스레드를 검사하면서 wakeup\_tick 값이 주어진 ticks의 값보다 작거나 같다면 그 스레드를 깨우고 sleep\_list에서 할 수 있도록 구현해주었다. 그 다음으로, threading system에서는 매 tick마다 실행되는 src/devices/timer.c의 함수 timer\_interrupt()에서 src/threads/thread.c의 thread\_tick()함수를 호출하는데, 이 함수가 실행될 때마다, 즉 매 tick마다 thread\_awake함수를 호출하여 깨어나야 할 스레드들은 일어날 수 있도록 구현해주었다.

**2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling 구현**

ready\_list 안에서 우선 순위가 높은 순대로 스레드들을 정렬하여 관리하기 위해, src/threads/thread.c에서 비교 함수 cmp\_priority를 정의해준 후, thread\_ublock 함수에서 주어진 스레드가 unblock될 때에 원래의 THREAD\_BLOCKED 상태에서 THREAD\_READY 상태가 되어 ready\_list에 삽입되는데, 이 때에 ready\_list 안의 스레드들이 우선 순위 순대로 정렬된 상태가 유지될 수 있도록 cmp\_priority를 정렬 기준으로 하는 list\_insert\_ordered 함수를 사용하여 해당 스레드가 ready\_list에 우선 순위 순의 정렬을 유지하며 삽입되도록 구현하였다. 이와 같이 thread\_yield 함수에서도 기존의 실행 중이던 스레드가 ready\_list에 삽입될 때 list\_insert\_ordered 함수를 사용하여 우선 순위 순으로 정렬된 상태가 유지되도록 구현하였다. 또, thread\_create 함수에서 기존의 코드가 실행된 후, 현재 실행되고 있는 스레드와 priority를 비교하여 새로 생성되는 thread의 priority가 더 크다면 thread\_yield 함수를 호출하여 preemption이 일어나도록 추가적인 구현을 하였다. 또, 현재 실행되는 스레드의 우선 순위가 ready\_list의 front보다 적다면, (ready\_list는 항상 우선순위 순으로 정렬되어 있으므로) thread\_yield를 호출하여 CPU를 양보하는 priority\_preempt() 함수를 정의하고, thread\_set\_priority 함수를 수정하여 스레드의 우선 순위가 변경되었을 때, 이 함수를 호출하여 필요한 경우 preemption이 이루어지도록 구현하였다.

**3. Advanced Scheduler에서 priority 계산 구현**

fixed-point arithmetic 관련 연사들을 구현하기 위해 새로 생성한 src/threads/fixed\_point.h 파일에서 관련 연산들을 수행하는 함수들을 독립적으로 만들어주었다. 가장 먼저, fixed-point 형식으로 표현한 1의 값을 F로 정의하고, fixed-point 형식으로 수를 표현할 때에는 이 값을 곱해주기로 하였다. fixed\_point.h에서 정의된 int\_to\_fp는 정수형 수를 fixed-point 형 수로 바꾸어 주는 함수이고, fp\_to\_int\_round와 fp\_to\_int는 fixed-point 형 수를 반올림, 버림을 사용하여 정수형 수로 바꾸어 주는 함수이며, add\_fp, add\_mixed는 fixed-point 형 두 수, fixed-point 형 수와 정수형 수의 덧셈 연산을, sub\_fp, sub\_mixed는 fixed-point 형 두 수, fixed-point 형 수와 정수형 수의 뺄셈 연산을, mult\_fp, mult\_mixed는 fixed-point 형 두 수, fixed-point 형 수와 정수형 수의 곱셈 연산을, div\_fp, div\_mixed는 fixed-point 형 두 수, fixed-point 형 수와 정수형 수의 나눗셈 연산을 해주는 함수이다. 이렇게 구현된 fixed\_point.h를 include하여 src/threads/thread.c에서 priority, recent\_cpu, load\_avg를 계산하는 mlfqs\_priority, mlfqs\_recent\_cpu, mlfqs\_load\_avg를 구현해주었다. 이 때, nice, priority, ready\_threads는 정수형 값을 담으며, recent\_cpu, load\_avg는 실수형 값을 담음에 유의하여 계산을 해주었다. 다음으로, multi-level queue 방식을 사용한 advanced scheduler에서는 이러한 스케줄러가 사용됨을 나타내는 flag thread\_mlfqs가 true일 때, 우선순위가 임의로 변경될 수 없으므로 thread.c의 thread\_set\_priority 함수에서 이 flag가 true일 때 우선순위를 임의로 변경할 수 없도록 수정하였다. 그리고 thread\_set\_nice 함수에서는 현재 nice의 값을 바꾼 후, 해당 priority를 다시 계산해주었고, 이 바뀐 priority에 따라 다시 스케줄링을 해주었다. thread\_get\_nice에서는 현재 스레드의 nice 값을 반환해주었고, thread\_get\_load\_avg와 thread\_get\_recent\_cpu 함수에서는 각각 load\_avg와 recent\_cpu에 100을 곱한 값이 리턴되도록 구현해주었다. 마지막으로, src/threads/thread.c의 thread\_tick 함수가 호출될 때 만약 thread\_mlfqs flag가 true라면 매 tick마다 현재 스레드의 recent\_cpu를 1 증가시키고, 매 4 tick마다 스레드들의 priority를 다시 계산하고, 매 1초마다 load\_avg, 스레드들의 recent\_cpu, priority가 다시 계산되도록 구현하였다.

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

1. priority-lifo.c 코드 분석

priority-lifo.c는 가장 먼저 순서대로 PRI\_DEFAULT+1, PRI\_DEFAULT+2, … , PRI\_DEFAULT+16을 priority로 갖는 스레드들을 16개 생성한다. 그 후, 스레드들을 생성했던 원래의 스레드의 priority를 낮추어, 가장 priority가 높은 스레드에서부터 simple\_thread\_func가 실행되는데 이 함수에서는 또 ITER\_CNT인 16만큼 op에 thread의 추가적인 data에 포함된 필드 id를 저장한다. 그리고 thread\_yield가 호출되기 때문에, 이 때에 새로 스레드가 스케줄링되어 그 스레드가 진행된다. 모든 스레드의 simple\_thread\_func 함수 실행이 끝나면, 다시 원래의 test\_priority\_lifo 함수로 돌아와 그 다음 for loop에서 op에 저장된 값들이 차례대로 출력되고 priority-lifo.c 코드가 종료된다.

2. priority\_lifo 테스트 결과 분석

priority\_lifo 테스트를 실행한 결과는 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위에서 설명한 것과 같이 16개의 스레드들이 모두 생성되면, 가장 priority가 높은, 즉 가장 마지막에 생성된 스레드에서부터 simple\_thread\_func가 실행되는데 이 함수에서는 또 ITER\_CNT인 16만큼 loop가 실행되는데, 이때 각 loop에서 op에 thread의 추가적인 data에 포함된 필드 id를 저장하고 thread\_yield가 실행되는데, 여기서 위에서 구현한 priority-scheduling에 따라 thread의 우선순위가 높은 순으로 스케줄링되므로, 위에서 가장 마지막으로 생성된 스레드가 16번의 loop를 먼저 수행하고, 그 다음으로 가장 마지막 전에 생성된 스레드, 즉 data->id가 14인 스레드가 16번의 loop를 수행하고, 그 다음으로 13인 스레드가 16번의 loop를 수행하는 식으로 진행된다. 마지막으로 가장 처음에 생성된 스레드까지 simple\_thread\_thread의 수행을 마치면 op에는 차례대로 15가 16번, 14가 16번, 13이 16번, … , 0이 16번 저장되게 되고, 이를 차례대로 출력하면 위의 priority\_lifo 테스트를 실행한 결과의 캡처 이미지와 같은 결과를 출력하게 된다.

* make check 수행 결과를 캡처하여 첨부

텍스트이(가) 표시된 사진

자동 생성된 설명