**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

조 / 조원 : 20181593 계인혜

개발 기간 : 20.11.23~20.12.04

1. **개발 목표**

Busy waiting 방식으로 구현되어 비효율적인 Alarm clock(ready와 running 상태를 반복)을 non-busy waiting 방식(blocked state로 변경 후 시간이 다 되면 ready queue에 다시 추가)으로 변경하여 효율적인 방식으로 적용한다.

Priority를 고려하지 않는 round-robin 스케줄링 방식에서 priority를 고려하는 스케줄링 방식으로 변경하고, starvation 문제를 해결하기 위해 aging 기법을 추가한다.

Multi-Level Feedback Queue를 구현하고, nice, recent\_cpu, priority값을 알맞게 계산한다. 또한, floating-point arithmethic을 지원하지 않는 pintos를 위해 적절하게 floating-point 연산을 할 수 있도록 한다.

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

타이머가 끝나지 않아도 thread\_yield 함수를 호출해서 thread가 RUNNING, READY 상태를 왔다 갔다 해야 하는 비효율적인 기존의 방식에서, 타이머가 끝나기 전에는 해당 thread를 BLOCKED 상태로 바꿔 깨어나야 할 때 UNBLOCK 하는 효율적인 방식으로 바꾼다.

* 1. Priority Scheduling

Thread의 priority를 고려하지 않는 단순한 Round-Robin 스케줄링에서 priority를 고려하는 스케줄링으로 바꾼다. 이로 인해 새로운 thread가 ready list에 들어올 때, priority에 의해 정렬되어 들어간다. 이때 현재 thread보다 새로 들어온 thread의 priority가 더 높다면 thread의 context switching이 일어난다. 또한, aging 기법을 적용하여 priority가 낮은 thread가 계속 돌지 못하는 starvation 문제를 해결한다.

* 1. Advanced Scheduler

Priority를 결정할 때 recent\_cpu, nice 값을 반영하여 스케줄링한다. Multi-Level Feedback Queue(MLFQ) 방식을 사용하여 priority마다 round-robin 스케줄링 기법을 따르는 각각의 ready queue를 갖는다. 이때, Priority 계산에 필요한 실수 연산을 따로 구현해야 한다. 이로 인해 좀 더 효율적인 스케줄링을 할 수 있다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Blocked 된 스레드를 깨우기 위해서 thread\_awake 라는 함수를 새로 정의하였다. 이 함수는 스레드의 tick이 현재의 tick 보다 작거나 같은 경우 해당 스레드를 Unblock 시키는 함수이다. 현재 자고 있는 스레드의 리스트(sleep\_list)를 하나씩 살펴보며 스레드의 tick과 현재 tick을 비교한다. 이때 스레드의 tick 시간이 지났다면 이를 sleep\_list에서 제거하고, 스레드를 unblock 한다. 이후 인터럽트가 끝나면 이 스레드는 ready queue에 들어가서 정상적으로 CPU의 할당을 받을 수 있다.

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

Running thread 보다 높은 우선순위를 갖는 thread가 들어오면 thread의 context switching이 일어난다. 즉, 우선순위가 높은 thread가 수행되며 돌고 있던 running thread는 자신의 정보들을 PCB에 저장하고 수행을 멈춘다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.
   * Nice : -20~20 사이의 값을 가지며, 우선순위와 밀접한 관계가 있다. Nice 값이 높을수록 우선순위는 낮아지며, 0일때는 우선순위와 관계가 없다.
   * Recent\_cpu : 스레드의 CPU 사용시간을 나타내는 값으로 idle 스레드가 아니면 (RUNNING, READY, BLOCKED) 매 tick 마다 1씩 증가한다. 초기값은 0이고 자식 스레드의 경우는 부모 스레드의 값을 상속받는다. 막연하게 스레드의 CPU 사용시간을 의미하는 것이 아니라 최근의 CPU 사용시간에 조금 더 중점을 두어 계산한다. 계산 시 소수 값이 나오지만, pintos 환경에 맞게 100을 곱해 가장 가까운 정수로 반올림한다.
   * Load\_avg : system-wide 변수로 시스템이 부팅될 때 초기값을 0으로 한다. Recent\_cpu 값을 계산하기 위해 필요하며, READY, RUNNING 상태에 있는 스레드의 개수를 통해 계산된다. 이 값은 매초마다 갱신되며 1초 전의 load\_avg 값과 READY, RUNNING 상태에 있는 스레드의 개수에 각각 가중치를 59, 1로 두어 계산한다. 계산 시 소수 값이 나오지만, pintos 환경에 맞게 100을 곱해 가장 가까운 정수로 반올림한다.
2. **추진 일정 및 개발 방법**
   1. **추진 일정**

(20.11.23~20.11.25) Pintos 매뉴얼 및 ppt 분석

(20.11.25~20.11.26) Alarm clock 구현

(20.11.27~20.11.29) Priority scheduling 구현

(20.11.29~20.12.02) Advanced Scheduler(BSD scheduler) 구현

(20.12.02~20.12.04) 보고서 작성 및 FAIL이 뜨는 테스트 케이스 위주의 디버깅

* 1. **개발 방법**
  + 수정해야하는 소스코드

Thread\_init() 함수에서 nice, recent\_cpu 값을 0으로 초기화하여 BSD shceduler의 구현을 정상적으로 작동하게 한다. 또한, thread\_tick() 함수를 수정하여 aging, mlfqs를 구현하였다. Priority scheduling 구현을 위해 thread\_create()에서 running thread의 priority가 새로 들어온 thread의 priority 보다 높다면 context switch가 일어날 수 있도록 코드를 작성하였다. 현재 thread가 mlfqs인지 확인하고 만약 그렇다면 주어진 식에 맞게 priority를 계산할 수 있도록 Thread\_set\_Priority() 함수를 수정하였다.

Thread\_set\_nice, thread\_get\_nice, thread\_get\_load\_avg, thread\_get\_recent\_cpu 함수를 ppt에 적힌 공식대로 계산하여 적절한 값을 읽고 쓸 수 있도록 함수를 수정하였다. 이 때, thread\_set\_load\_avg, thread\_set\_recent\_cpu 함수를 새로 추가하여 load\_avg, recent\_cpu 값을 계산할 수 있도록 했다. Load\_avg를 계산하기 위해서는 현재 ready\_list의 element 개수가 필요하다. 이를 구하기 위해 thread\_get\_ready라는 함수를 새로 정의하였다.

Thread\_sleep() 함수를 추가하여 깨어날 시간을 현재 스레드의 wakeup\_time에 저장하고 이를 sleep\_list에 넣는다. 이후 해당 스레드를 tick이 끝나기 전까지 block하여 CPU의 비효율적인 사용을 막는다.

Thread\_awake() 함수를 추가하여 자고 있는 스레드를 깨울 수 있도록 한다. Thread aging() 함수를 추가하여 Idle, Running 상태의 스레드를 제외한 스레드들의 priority를 1씩 증가시킨다. 또한 Ready\_list\_less\_func 함수를 추가하여 왼쪽, 오른쪽 list\_elem 원소를 비교 후 내림 차순으로 정렬되어 있는지 확인한다. 만약 왼쪽의 priority가 오른쪽의 것보다 더 크다면 true를 리턴하고 그렇지 않다면 false를 리턴한다.

마지막으로 실수형 연산을 수행하기 위한 함수들을 새로 추가하였다. OperationType1Type2 꼴로 예를 들어 sumIntFloat 함수는 int에 float를 더하는 것이고, subFloatFloat 함수는 float에서 float를 빼는 것이다. 같은 방식으로 모든 사칙연산(+, -, \*, /) 과 가능한 타입의 경우의 수({int, int}, {int, float}, {float, int}, {float, float}) 를 모두 조합하여 실수형 연산을 수행할 수 있도록 했다.

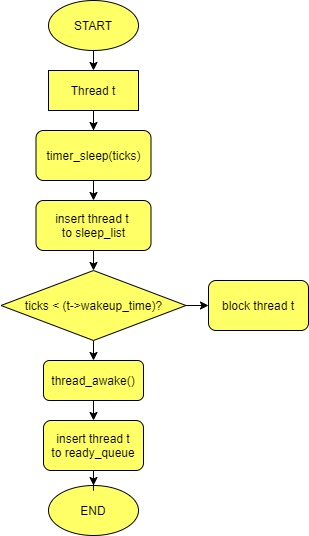
* + 수정하거나 추가해야 하는 자료구조

Struct thread에 int64\_t wakeup\_time, int nice, int recent\_cpu 변수를 추가하였다.

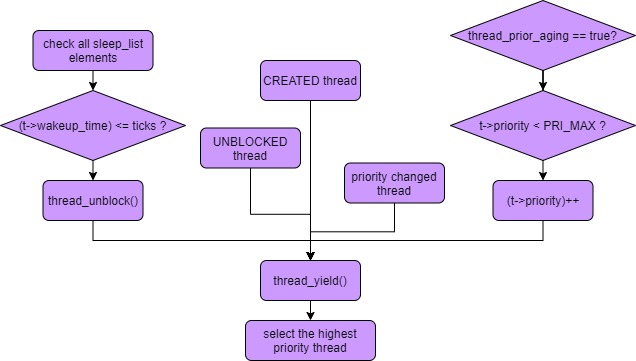
1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성

**(alarm clock)**

****

**(Priority Scheduling)**

****

* 1. **제작 내용**

1. **Alarm clock**

Void thread\_sleep 함수를 추가하여 block 될 시간만큼 해당 스레드를 sleep\_list 에 넣어둔다. 스레드가 sleep\_list에 있는 동안에는 BLOCKED 상태가 된다. 이로 인해 타이머가 다 되지 않았는데 스레드가 계속 RUNNING, READY 상태를 왔다 갔다해서 CPU를 차지하게 하는 비효율적인 알고리즘을 수정하였다.

Void thread\_awake 함수를 추가하여 sleep\_list에 있는 스레드들 중 타이머가 완료된 것을 찾아 UNBLOCK 상태로 바꿔준다. 이는 sleep\_list에 있는 스레드를 하나씩 살펴보면서 해당 스레드의 wakeup\_time이 현재 tick 이하라면 스레드를 UNBLOCK 해주는 방식으로 구현하였다. 이때, wakeup\_time은 타이머가 끝나는 시간 즉, 스레드가 깨어날 시간을 의미한다.

1. **Priority scheduling**

tid\_t thread\_create 함수에서 우선순위를 고려하는 스케줄링을 구현하기 위하여 running thread의 priority와 새로운 스레드의 priority를 비교하는 부분을 추가하였다. 이로 인해 현재 돌고 있는 스레드의 우선 순위보다 새로운 스레드의 우선순위가 더 높은 경우 스레드의 context switching이 일어난다.

Void thread\_unblock 함수에서 스레드를 unblock 시키게 되면 해당 스레드는 다시 ready\_list로 들어가게 된다. 이때 우선순위를 기준으로 하여 오름차순으로 ready\_list 가 정렬될 수 있도록 코드를 수정하였다.

Void thread\_set\_priority 함수를 수정하여 현재 스레드의 priority를 new\_priority로 설정하고, 만약 설정한 priority가 이전 스레드의 priority보다 낮은 경우 thread\_yield 함수를 통하여 switch 한다.

Bool ready\_list\_less\_func 함수를 추가하여 ready\_list가 우선순위를 기준으로 내림차순으로 정렬될 수 있도록 하였다. 왼쪽의 우선순위가 오른쪽의 우선순위보다 작다면 잘못된 정렬이므로 false를 리턴하고 그렇지 않다면 true를 리턴한다.

Void thread\_aging 함수를 추가하여 ready\_list에 있는 스레드들이 낮은 우선순위로 인해 starvation 문제가 일어나는 것을 방지한다. Ready\_list에 있는 스레드들을 하나씩 살펴보며 스레드의 우선순위가 최댓값을 넘지 않으면 우선순위를 1씩 올려준다. Aging은 IDLE, RUNNING 상태에 있는 스레드에는 적용되지 않는다.

1. **Advanced scheduler**

void thread\_init () 함수에서 스레드를 초기화 할 때 nice와 recent\_cpu 값을 0으로 맞췄다.

Void thread\_set\_priority 함수에서 Thread\_mlfqs가 켜지면 추가 구현을 위한 것으로 간주하여 4 tick 마다 priority를 갱신한다. Priority의 갱신은 다음과 같은 식을 따른다.

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

void Thread\_tick() 함수에서 Thread\_mlfqs가 켜지면 추가 구현을 위한 것으로 간주하여 상황에 맞게 recent\_cpu, load\_avg 값을 설정하였다.

Void thread\_set\_load\_avg() 함수를 추가하여 현재 스레드의 load\_avg 값을 다음과 같은 공식에 맞게 설정한다. 공식을 살펴보면, 이전의 load\_avg 값과 ready\_thread에 들어있는 스레드 개수에 각각 59, 1의 가중치를 두어 계산한다는 것을 알 수 있다.

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

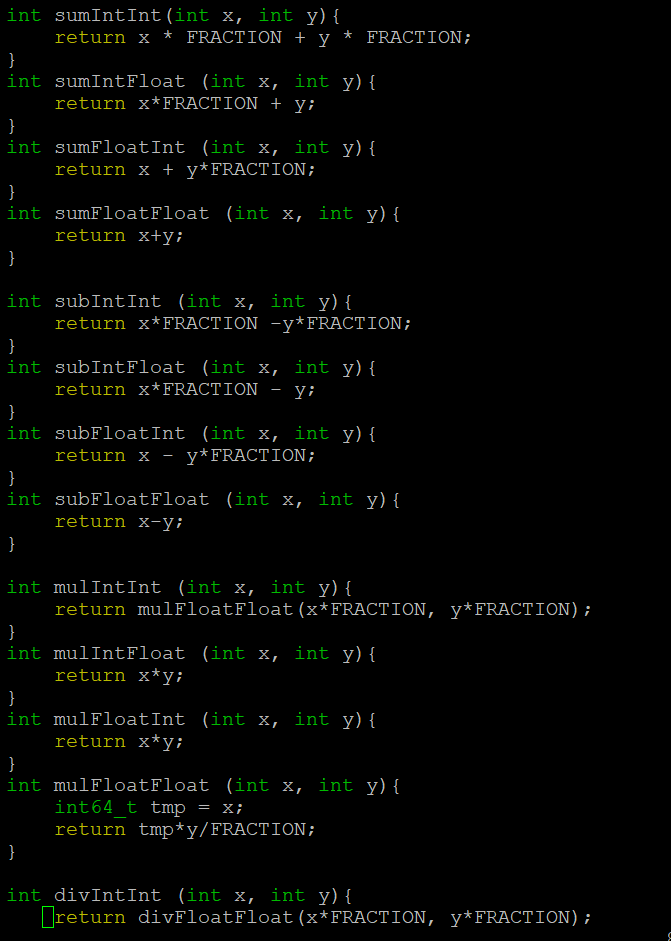
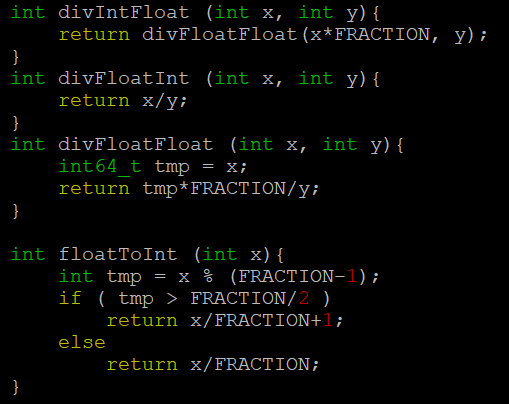
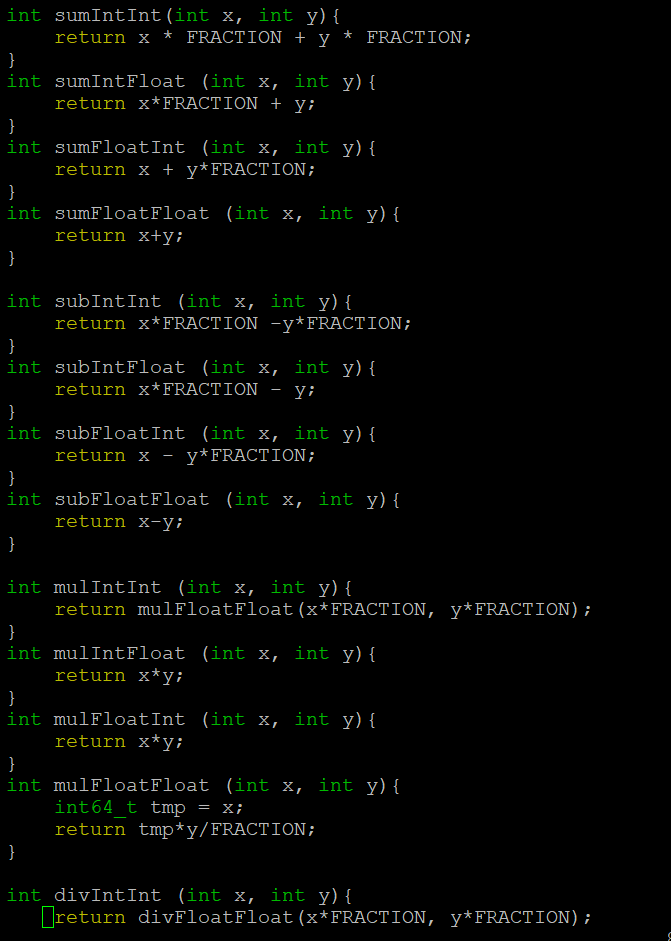
이때 Pintos 커널은 실수 연산을 할 수 없기 때문에 계산 결과에 100을 곱한 수를 반올림한 정수 값을 리턴해야 한다. (int thread\_get\_load\_avg())

Int thread\_get\_ready() 함수를 추가하여 ready\_thread에 들어있는 스레드의 개수를 구해 load\_avg를 구하는데에 사용하였다.

Void thread\_set\_recent\_cpu() 함수를 추가하여 idle 상태에 있는 스레드를 제외한 모든 스레드들의 recent\_cpu 값을 다음과 같은 공식에 맞게 설정한다.

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

이때 Pintos 커널은 실수 연산을 할 수 없기 때문에 계산 결과에 100을 곱한 수를 반올림한 정수 값을 리턴해야 한다. (int thread\_get\_recent\_cpu())

****Advanced scheduler를 구현하기 위해서는 floating point 연산이 필요한데, pintos 체제에서는 이를 지원하지 않는다. 이러한 문제를 해결하기 위하여 다음과 같은 함수들을 추가하였다. 각 함수를 일반화하여 설명하면 operatorType1Type2로, type1에 해당하는 값에 type2에 해당하는 값을 operator 연산하여 실수로 리턴한다. 예를 들어 mulFloatInt 함수는 Float에 int를 곱하는 것이고, divFloatFloat 함수는 float에서 float를 나누는 것이다. 또한, float를 int로 변환하는 함수(floatToInt)도 추가하였다.

* 1. **시험 및 평가 내용**

다음은 make check 수행결과이다. 채점에 포함되는 테스트 케이스에는 노란색으로 밑줄을 쳤으며 모든 테스트 케이스에서 pass를 받았다.

****

Priority-lifo.c의 코드를 분석한 결과는 다음과 같다. 먼저 16개의 스레드를 생성하고 각 스레드에 16개의 데이터를 저장한다. 이때 저장하는 데이터는 생성되는 스레드의 번호(thread #, thread # : 0~15)이며, 생성되는 스레드의 priority는 (PRI\_DEFAULT(31) + 1 + thread #)로, 순서대로 32~47 까지의 값을 갖는다. 이후 priority scheduler를 적절하게 구현했다면 priority가 높은 스레드의 데이터를 한 줄씩 출력한다. 올바른 출력은 priority가 가장 높은 15번째 스레드부터 자신의 thread #(15)를 16개 출력하고, 그 다음은 14번째 스레드의 데이터(14)를 출력하는 방식으로 0번 스레드까지 자신의 데이터(0)를 16개 출력한다. 다음은 priority-lifo 명령 수행 결과이다.

