**Pintos Project 3: Threads**

담당 교수 : 박성용

조 / 조원 : - / 강민석

개발 기간 : 2020.11.23 ~ 2020.11.29

1. **개발 목표**

cpu 자원을 효율적으로 사용하기 위하여 thread가 busy waiting 대신 sleep 할 수 있도록 변경하고, priority를 고려한 scheduling이 가능하도록 만든다. 추가구현으로는 priority에 더해 nice, recent\_cpu, load\_avg와 같은 변수를 고려하여 보다 합리적인 scheduling(BSD)을 구현한다.

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

기존 pintos에서는 timer\_sleep 함수를 이용하여 thread가 sleep 상태에 진입하도록 만든다. 그러나 sleep에서 깨어나기 이전까지 계속 시간을 체크하는 busy waiting 방식으로 구현되어 있기 때문에 보다 프로젝트 요구사항을 구현함으로써 cpu를 효율적으로 사용할 수 있다.

* 1. Priority Scheduling

기존 pintos에서는 round-robin scheduling을 이용한다. Priority를 고려한 scheduling을 구현함으로써 중요한 스레드를 먼저 실행시킬 수 있게끔 한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

Priority Scheduling에서는 priority가 낮은 thread는 실행이 안되는 starvation 현상이 발생할 수 있다. 이를 해결하기 위하여 여러 요소를 고려하여 시간이 지남에 따라 priority가 변하는 scheduling을 구현하므로써 보다 합리적으로 scheduling이 이뤄질 수 있도록 한다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Blocked 상태의 스레드는 blocked\_list(새로 선언한 리스트)에 있다가 tick이 변화했을 때, 리스트의 내부를 순회하면서 깨어나야할 thread가 있는지 검토한다. 해당 thread가 발견되면 thread\_unblock을 호출함으로써 깨워준다.

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

이번 프로젝트에서 구현한 priority scheduling은 preemption을 허용한다 따라서 running thread보다 높은 priority를 갖는 thread가 ready\_list에 들어올 경우 running thread는 thread\_yield를 통해 cpu를 양보하고 scheduling을 통해 새로 들어온 thread가 실행된다.

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

nice : thread가 다른 thread에게 얼마나 ‘nice’한지를 나타낸다. 즉 값이 클수록 priority 계산에서 nice 값이 차감되는 형태로 나타나 다른 thread에게 더 많은 cpu 자원을 양도한다.

recent\_cpu : thread가 사용한 cpu시간을 근사한 수치이다. 매초마다 갱신된다.

load\_avg : 전역 변수로 초기에 0으로 시작하며, 매초마다 update된다. load\_avg는 recent\_cpu 업데이트에 사용된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2020.11.23 ~ 2020.11.24 : Alarm Clock 파트 구현

2020.11.25 ~ 2020.11.26 : Priority Scheduling 파트 구현

2020.11.27 ~ 2020.11.28 : BSD Scheduling 파트 구현

2020.11.29 : code 정리 및 수정 작업

* 1. **개발 방법**

(1) blocked 상태의 thread를 깨우는 부분 구현

우선 thread가 block되는 상태를 먼저 구현해야 한다. 기존에는 devices/timer.c 파일의 timer\_sleep함수에서 busy\_waiting(while loop)을 통해 구현되었었는데 이부분을 지우고 새로 thread를 blocked\_list에 집어넣는 함수를 호출한다. 해당 기능을 하는 함수를 thread\_sleep이라 정의하고 thread.c 파일에 정의하였다. thread\_sleep 함수에서는 thread가 깨어나야할 시간을 저장하고, 해당 thread를 list\_push\_back 함수를 이용하여 blocked\_list에 집어넣는다. 그리고 해당 thread에 대해 thread\_block함수를 호출하면 block이 완료된다.

이후에 thread가 깨는 과정은 다음과 같다. devices/timer.c에 정의되어 있는 timer\_interrupt함수는 매번 호출된다. 이 때마다 tick이 하나씩 증가하는데, 증가할 때마다 thread.c에 정의되어 있는 thread\_tick()함수가 호출된다. 이 함수에서 thread\_wake\_up()함수를 호출하여 blocked\_list에 깨어나야 되는 thread가 있는지 탐색한다. 만약 있다면, list에서 제거하고 thread\_unblock함수를 호출함으로써 다시 ready\_list에 집어넣어주는 작업을 진행하였다.

(2) Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 대한 구현

위와 같은 경우 preemption이 일어나야 한다. 해당 부분에 대한 경우는 thread\_unblock을 통하여 ready\_list에 삽입된 후에 추가 조건문을 이용하여 구현할 수 있다. 즉 만약에 삽입된 thread의 priority와 thread\_current()(현재 실행중인 thread)의 priority를 비교하여 만약 삽입된 thread의 priority가 더 크다면 therad\_yield()함수를 호출하여 preemption이 일어나도록 할 수 있다. 또한 ready\_list의 thread들을 priority 내림차순으로 정렬하기 위하여 insert할 때마다 list\_push\_back 대신 list\_insert\_ordered함수를 이용하면 보다 빠르게 현재 실행 중인 thread와 ready\_list에 있는 thread 중 가장 큰 priority 값을 가지는 thread와의 priority를 비교할 수 있다.

(3) Advanced Scheduler에서 priority 계산에 필요한 각 요소 서술

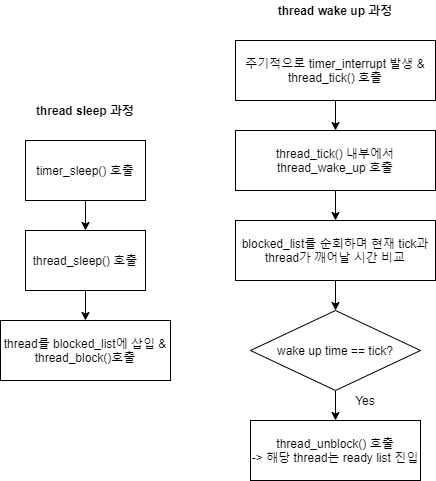
우선 BSD scheduler에서는 실수 연산이 필요하지만, pintos는 실수 연산을 지원하지 않는다. 이를 위해 간단히 fixed point 실수 연산을 지원하는데 해당 부분을 thread 디렉토리의 fixed\_point.h 파일에 정의하였다. 이후에 thread.h의 thread 구조체에 nice 변수, recent\_cpu 변수를 선언하고 thread.c 파일에 load\_avg를 전역변수로 선언한다. 추가적으로 thread.c 파일의 슬라이드에서 제시하는 priority 계산 등의 항목을 함수로 정의하고, 마지막으로 devices/timer.c 파일에서 timer\_interrupt가 발생할 때 다음과 같은 항목을 추가한다.

- 1초(100 tick)마다 모든 thread의 recent\_cpu 갱신, load\_avg 변수 갱신

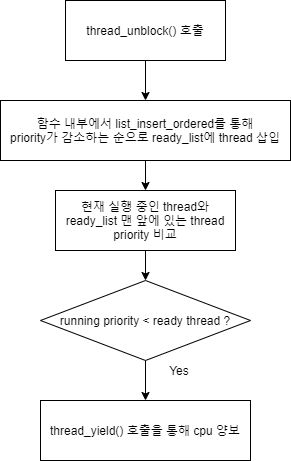
- 4tick 마다 모든 priority 갱신

1. **연구 결과**
   1. **Flow Chart**

- Alarm Clock

****

- Priority Scheduling

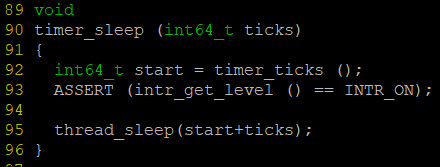
****

* 1. **제작 내용**

(1) Blocked 상태의 thread를 깨우는 방법

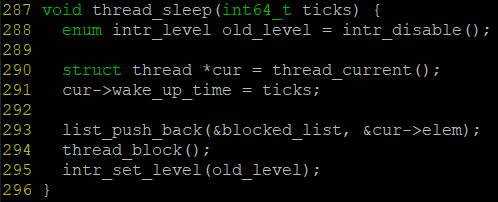
- thread의 sleep 과정

우선 devices/timer.c 파일의 busy-waiting 부분을 수정하여야 한다.



(devices/timer.c)

위와 같이 기존의 while문을 제거하고 thread\_sleep이라는 함수를 추가하였다. thread\_sleep은 thread.c에 추가로 구현한 함수로, blocking 상태의 thread를 list에 집어넣는 기능을 한다. 이를 위해 다음과 같이 thread.h 파일에 blocked\_list라는 struct list를 새로 선언하였다. thread\_sleep 함수는 다음과 같다.

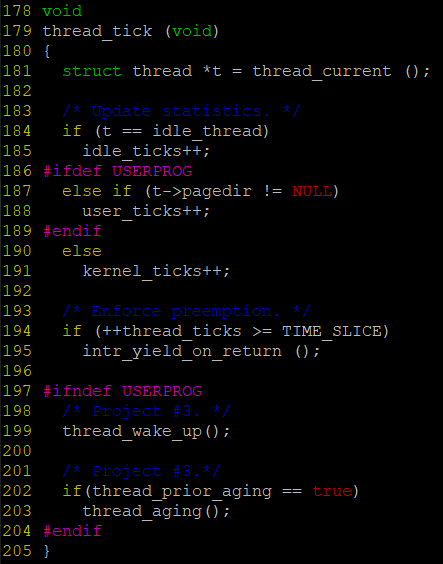


(thread/thread.c)

우선 중간에 interrupt가 발생하면 안되기 때문에 interrupt를 off 하고, 현재 thread에 새로 선언한 int64\_t 변수 wake\_up\_time에 깨어나야 할 시간 ticks 파라미터를 저장한다. 그 후, blocked\_list에 list\_push\_back 함수를 이용하여 집어넣고, thread\_block을 호출한다.

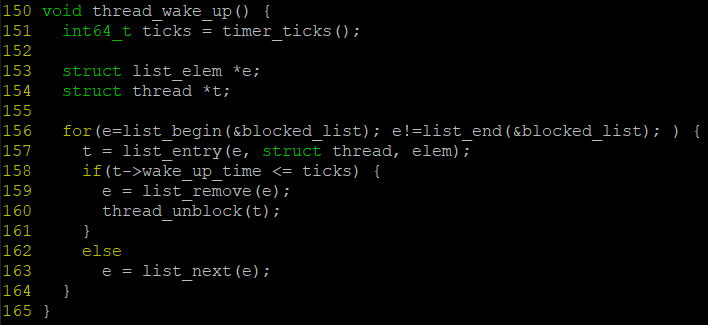
- 깨어나는 과정

pintos에서는 주기적으로 timer\_interrupt가 발생한다. 해당 interrupt가 발생할 때 마다 tick을 하나 증가시키고, thread\_tick()이라는 함수를 호출한다. thread\_tick() 함수는 thread.c에 기본 정의되어 있는 함수로 해당 부분에 새로운 함수를 추가하여 wake up을 구현하였다.



(threads/thread.c)

해당 함수에서 thread\_wake\_up() 함수를 호출하는데 thread\_wake\_up() 함수는 다음과 같다.



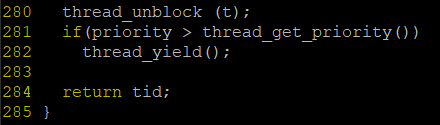
(threads/thread.c)

위 함수는 호출될 때 blocked\_list에 있는 thread들을 순회하면서 현재 깨어나야 할 thread가 있는지 확인한다. (158번 조건문) 만약에 조건을 만족하는 thread가 있으면 blocked\_list로 부터 thread를 제거하고 thread\_unblock을 통해 다시 ready\_list에 thread가 들어가게 된다.

(2) Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우의 priority scheduling

프로젝트를 진행하면서 해당 경우는 다음과 같이 두 가지 경우에 발생하였다.

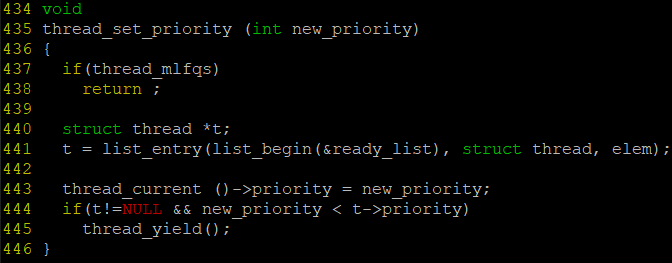
- thread\_create함수를 호출하여 thread를 생성하고 thread\_unblock을 하는 경우



(threads/thread.c)

위와 같이 생성된 thread의 priority와 thread\_get\_priority() (현재 thread의 priority)의 반환값을 비교하여 만약에 현재 생성된 thread의 priority가 높다면 thread\_yield()를 통하여 preemption이 이뤄지도록 하였다.

- set\_priority 함수 호출을 통해 현재 thread의 priority가 변경되는 경우

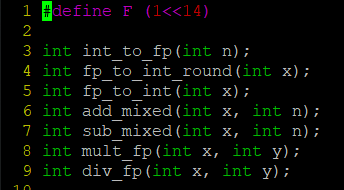


(threads/thread.c)

현재 thread의 priority가 변경되었는데 ready\_list의 가장 큰 priority를 갖는 thread 보다 priority가 작아진다면 preemption이 이뤄져야 한다. 해당 부분은 윗 캡쳐의 443 ~ 445를 통해 구현하였다.

(3) Advanced Scheduler에서 priority 계산에 필요한 각 요소 서술

일단 pintos의 fixed point 실수 연산을 지원하기 위해 새로운 해더파일 fixed\_point.h를 정의하였다.

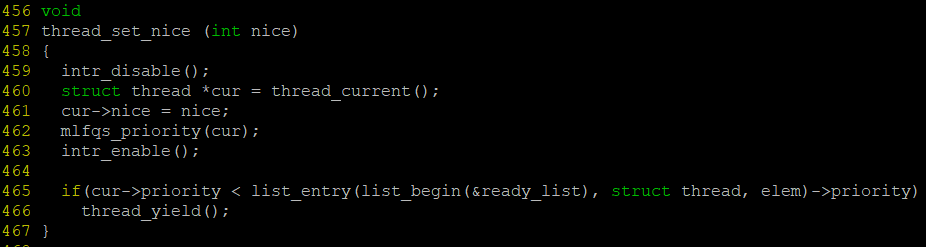


(threads/fixed\_point.h)

위 함수들은 pintos manual에 나와 있는 fixed point 연산에 대한 부분을 그대로 코드로 옮긴 부분이다. 따로 특별한 변환이 필요하지 않은 경우 (ex. 실수와 실수의 덧셈은 그냥 + 기호로 가능)는 생략하여 필요한 함수만 구현하였다.

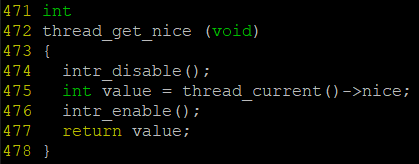
- nice value (정수)

우선 nice 값을 thread.h에 새로 선언하였다. nice 값과 직접적으로 관련된 함수는 set nice 함수가 있는데 해당 부분은 다음과 같다.



(threads/thread.c)

단순히 파라미터 값을 thread의 nice값에 update 하는 함수이다. 단, nice 값이 바뀌면 이에 따라 priority 값도 바뀌어야 하므로 mlfqs\_priority 함수를 호출하여 새로 priority를 update 해주었고, 이에 따라 preemption이 일어날 가능성이 있으므로 465~466 코드를 통해 만약 preemption이 이뤄져야 한다면 thread\_yield()를 호출하였다.

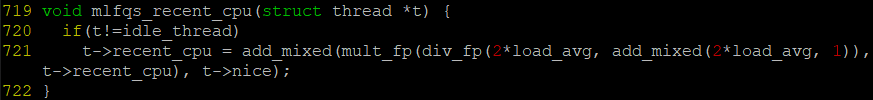


(threads/thread.c)

현재 thread의 nice 값을 반환하는 함수이다.

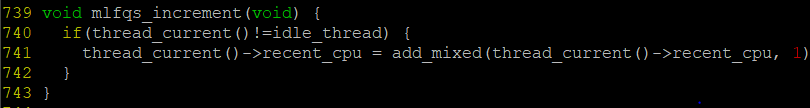
- recent\_cpu (실수)

recent\_cpu 변수를 thread.h에 새로 선언하였다. recent\_cpu와 관련된 연산을 지원하는 함수는 다음과 같다.



(threas/thread.c)

위 함수는 파라미터 thread에 대해 recent\_cpu를 계산하는 것으로 명세서에 나와있는 recent\_cpu = (2\*load avg) / (2\*load avg + 1) \* recent\_cpu + nice 값을 코드로 구현한 부분이다.

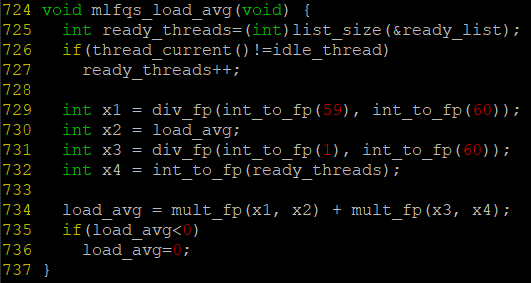


(threads/thread.c)

위 코드는 현재 thread에 대해 recent\_cpu 값을 1 증가시켜주는 함수이다.

- load avg (실수)

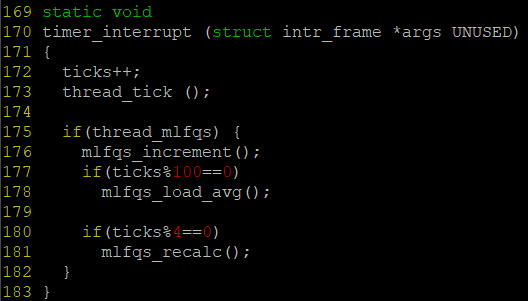
해당 변수는 thread.c 파일에 static 전역 변수로 선언하였으며, 0으로 초기화하였다. 관련된 함수로는 다음과 같다.



(threads/thread.c)

위 함수는 명세서에 나와있는 load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_threads 를 구현한 부분이다. ready threads를 list\_size 함수를 호출하여 계산하고, fixed\_point.h에 정의되어 있는 연산 함수를 호출하여 load\_avg를 갱신하였다. load\_avg는 음수일 수 없으므로 0보다 작은 경우 0으로 보정해주는 코드를 736줄에 추가하였다.

위 세 가지 요소(nice, recent\_cpu, load\_avg)를 바탕으로 priority 갱신이 주기적으로 이뤄지는데, 그 과정은 다음과 같다.



(threads/thread.c)

timer\_interrupt가 발생하는데 매 tick 마다 mlfqs\_increment()함수를 호출하여 현재 running thread에 대한 recent\_cpu값을 1증가시키고, 100 tick(1초)마다 mlfqs\_load\_avg를 호출하여 load\_avg값을 갱신한다. 또한 4 tick 마다 mlfqs\_recalc()를 호출하여 모든 스레드에 대해 priority를 갱신해준다.

* 1. **시험 및 평가 내용**

- priority-lifo.c

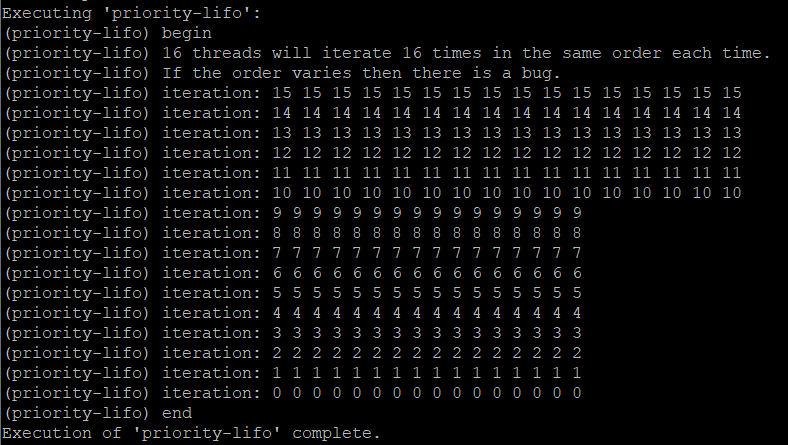
priority-lifo.c에서는 thread를 16개 생성하여 모든 thread가 round-robin order로 돌아가는지 확인하는 코드이다. 과정은 다음과 같다.

(1) main thread의 priority를 PRI\_DEFAULT +17로 설정

(2) child를 16개 생성하는데 PRI\_DEFAULT + 1 / PRI\_DEFAULT +2 / ... 로 1씩 증가하게끔 priority를 설정

(3) main thread의 priority를 PRI\_DEFAULT로 설정 (preemption으로 인해 child가 돌아갈 것임을 예측할 수 있다.)

priority-lifo 테스트 결과



테스트 결과 위와 같았다. 즉 15번째(가장 마지막에 생성된 therad)가 실행되다가 마지막에는 0번째(처음 생성된 thread)가 실행되고 종료되는데, 이는 마지막에 생성된 thread의 priority가 PRI\_DEFUALT + 16으로 가장 높기 때문이고, 처음에 생성된 thread의 priority가 PRI\_DEFAULT + 1로 가장 낮기 때문이다. 따라서 Last In(created) First Out 형태의 결과가 화면에 출력된다.

- make check 결과 화면

