**Pintos Project 3: Threads**

담당 교수 :문의현

학번 / 이름 :20181256/김도현

개발 기간 :22.11.23~22.11.29

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

Thread 를 관리하기 위한 시스템 기능인 alarm clock, priority scheduling, 그리고 추가적으로 advanced scheduler (BSD schedular)에 대한 코드를 구상하고 또 구현하는 것이 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

Thread가 일을 하지 않을 때 sleep mode 에 들어가게 되는데 기존의 코드는 반복문을 통해 timer ticks 시간 만큼 thread 가 sleep 이 되도록 지속적으로 확인한다. 이때 thread 는 계속해서 running state, ready state 를 왔다 갔다 하기 때문에 cost 적인 측면이 크다. 즉 busy waiting 을 방지하기 위해 block 하는 방식을 취해야 한다. timer ticks 가 아직 지나지 않았으면 blocked state thread 를 담고 있는 queue 에 넣고 wake\_up 시간이 되면 이 thread 를 다시 ready\_queue 에 넣는다. 이로써 CPU TIME cost 를 줄일 수 있다.

* 1. Priority Scheduling

Thread\_yield 나 thread\_unblock 이 실행될 때 기존의 코드는 우선순위 를 고려하지 않고 무조건 ready\_queue 의 가장 마지막에 들어가는 형태 였다. 하지만 thread 별로 먼저 실행되어야 하는 우선순위가 있을 것이다. 이를 위해 각 thread에 우선순위를 추가해 이 우선순위에 따라 ready\_list 에 priority 로 정렬된 상태로 추가되어야 한다. 이때 우선순위가 낮은 thread 들은 계속해서 실행이 밀리는 현상이 발생한다. 이를 막기 위해 ‘aging’ 기법을 도입하여 시간이 지날 수록 priority(우선순위) 를 단계적으로 높여 일정 시간 안에 해당 thread 를 실행하도록 해준다.

* 1. Advanced Scheduler (추가구현을 한 경우)

기존의 ready-queue가 아닌 Multi-level feedback queue(MLFQ)를 사용하는 scheduler 를 구현해야 하는데 우선순위에 따라 여러개의 ready queue 들이 64개가 있고 각 ready queue 는 round robin 규칙을 따른다. 이 구현을 하게 되면 평균적으로 response time이 줄어든다는 이점이 존재한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Blocked 된 상태의 thread 들은 blocked queue에 들어 있다. 이때 해당 queue 의 각thread wake up 시간을 체크하여 wake up 시간이 되었다면 해당 thread 를 ready queue 로 넣어주면 이 방법이 blocked thread 를 깨우는 것이다. 그리고 timer\_interrupt 함수에서 tick 호출 시 tick++ 되는데 이 함수를 이용해서 wake up thread 를 체크하고 찾는 것이다.

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

Priority 는 0~63 까지이고 초기값은 31이다. 만약 현재의 cpu 내 running thread 보다 더 높은 priority를 가진 thread가 들어오게 된다면 우선순위가 더 높기 때문에 새로운 thread 에게 thread\_yield 를 해주어야 한다.

그리고 ready queue 에 넣을 때 기존의 list\_push\_back 함수가 아니라 list\_inserted\_ordered 함수를 사용해 우선순위에 따른 정렬 상태인 ready list 로 저장하고 있으면 된다.

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

Advanced scheduler(bsd scheduler) 는 64개의 ready queue 가 있다. 여기서 초기값으로 31이 thread\_create 함수에서 결정된다. 여기서는 4 ticks 씩 thread priority 를 다시 계산한다.

변수는 nice, recent\_cpu, load\_avg 가 있다.

Priority=pri\_max –(recent\_cpu/4)-nice\*2 공식으로 priority 를 계산한다.

Nice 값은 -20~20 사이의 값으로 nice 가 0보다 클 때는 해당 thread의 priority 값을 줄인다. 초기값은 0으로 세팅하고 만약 부모 thread 가 존재한다면 부모 thread 의 nice 를 자식 thread 도 받게 된다.

Load\_avg 값은 ready state 로 있는 thread 들의 평균 개수를 나타낸다. System booting 마다 이 값은 0으로 초기화 되어 있다. Load\_avg 는 매초마다 다시 계산을 통해 업데이트 된다.

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

Ready\_threads 는 ready 나 running state 에 있는 # thread 이다.

Recent\_cpu 값은 최근 어떤 thread 가 cpu time을 사용한 값으로 최근에 사용한 cpu time 이 더 높은 값을 받도록 가중치를 설정한다. Thread 가 create 될 때는 초기값은 0으로 세팅하고 만약 부모 thread 가 존재한다면 부모 thread 의 recent\_cpu 를 자식 thread 도 받게 된다. Time interrupt 함수가 실행 되면 running 상태인 thread 들의 recent\_cpu 값이 1 증가하게 된다. 또한 매 초 마다 어떤 상태이든(running, ready, blocked) 모든 thread 에서 recent\_cpu 값은 해당 식으로 다시 계산되어 업데이트 된다.

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

즉 해당 계산식들을 토대로 요약하면 thread의 cpu time 이 클수록, 해당 thread 의 next scheduling 에서는 더 낮은 우선순위를 가지는 것이다.

**추진 일정 및 개발 방법**

* 1. **추진 일정**
* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

22.11.23~22.11.24

ppt 이해 및 강의 영상 시청, alarm clock 구현

22.11.24~22.11.26

priority scheduling, advanced scheduler 일부 구현

22.11.26~22.11.28

advanced scheduler 구현

22.11.28~22.11.29

코드 간소화 및 수정, 보고서 작성

* 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. alarm clock**

Devices/timer.c

Blocked state 인 thread 들을 저장할 blocked\_list 를 새로 만들어 준다. 이를 timer\_init 함수에서 list\_init() 함수를 통해 초기화를 시켜준다. Timer\_sleep 함수에서 기존의 thread\_yield(busy waiting)를 thread\_block 방식으로 바꿔준다. Timer\_interrupt 함수에서 wake up 할 thread를 체크하는 코드를 추가해야 한다.

threads/thread.h

struct thread 내부에 int wake\_up 변수를 추가하여 thread 가 얼마나 blocked 된 상태로 있어야 하는지 저장할 수 있게 설정한다.

Threads/thread.c

Wake\_up 변수에 대한 초기 설정을 init\_thread 함수 내부에서 0으로 초기화 시켰다.

**2. priority scheduling**

Threads/thread.h

priority scheduling 시 필요한 aging 를 구현하기 위해서 bool 형식의 prior\_aging 변수를 선언하였다.(강의 자료 내용에 사진으로 되어 있는 코드)

threads/init.c

thread.h 에서 작성한 aging 시 prior\_aging 변수에 true 를 줘서 aging 할 수 있도록 해준다.(강의 자료 내용에 사진으로 되어 있는 코드)

threads/thead.c

bool 형식의 prior\_aging 변수를 선언한다.(강의 자료 내용에 사진으로 되어 있는 코드)

thread\_tick 함수에서 prior\_aging 이 true 면 thread\_aging\_fun 함수를 실행하도록 코드를 추가한다. (강의 자료 내용에 사진으로 되어 있는 코드)

thread\_create 함수, thread\_yield 함수, thread\_unblock 함수에서 기존 코드에서 thread 의 priority 에 맞게 ready\_list 가 재배치 될 수 있도록 list\_push\_back 함수에서 list\_insert\_ordered 함수로 수정한다.

List\_insert\_ordered 함수를 사용하기 위해 cmp\_pri 함수를 추가한다. 이 함수는 2 개의 thread를 parameter 로 받아 priority 를 비교해서 왼쪽이 더 큰지 확인해 그 결과를 bool 형식으로 return 해준다.

Thread\_set\_priority() 함수를 추가해 current thread의 Priority 를 계속해서 업데이트 해주는 과정이 필요하다. 이때 받는 parameter 를 thread priority 로 주면 된다.

Thread\_get\_priority() 함수에서 코드를 추가해 현재의 thread 의 Priority 를 가져오도록 한다.

Devices/Timer.c

Timer\_interrupt함수에서 recent\_cpu 값을 계산하고 timer\_tick 함수를 불러와 thread\_aging\_fun 함수를 호출 여부를 결정하여 aging 과정을 해준다.

추가적으로 advanced scheduler 와 겹치는 함수 및 코드는 아래서 설명하도록 하겠다.

**3.advanced Scheduler**

threads/thread.h

Define FRACTION(1<<14) 와 static int load\_avg 를 정의하여 priority 계산 식에 사용 할 수 있도록 하였다.

Thread 구조체 내부에 정수형인 nice, recent\_cpu 를 설정하여 이를 priority 계산 식에 사용 할 수 있도록 하였다.

Device/timer.c

Timer\_interrupt 함수에서 thread\_mlfqs 에 대한 조건을 추가하여 thread\_aging\_fun 을 작동하도록 코드를 수정한다.

Threads/thread.c

Nice, recent\_cpu 변수에 대한 초기 설정을 init\_thread 함수 내부에서 0으로 초기화 시켰다.

Thread\_get\_nice 함수에서 current\_thread 의 nice 값을 return 하는 함수를 추가하고 thread\_set\_nice 함수에서는 current\_thread 에 parameter로 받은 값을 넣어주고 변경된 nice 값에 따라 다시 thread priority 를 계산하면 된다.

Thread\_get\_recent\_cpu 함수에서 current\_thread 의 recent\_cpu 값에 100을 곱하고 return 하는 함수를 추가하고 calculation\_recent\_cpu 에서는 앞서 적은 recent\_cpu를 계산하는 식을 추가한다. 또한 load\_avg 를 위해 thread\_get\_load\_avg 함수 내부에 load\_avg에 100을 곱해 return 하는 코드를 추가적으로 구현하고 calculation\_load\_avg 함수에서 load\_avg 구하는 식을 추가해서 계산하도록 한다.

이때 모든 계산 식들은 fixed point real arithmetic 을 써야 한다. 이를 위해 calculation\_add, calculation\_sub, calculation\_mul, calculation\_div 함수를 구현하여 소수와 정수 계산을 가능하도록 만들어주었다.

Thread\_aging\_fun 함수 내에서 aging 여부에 따라 list\_elem 을 하나씩 돌면서 priority 를 다시 계산해주고 priority 를 업데이트 하는 코드를 추가 및 수정한다.

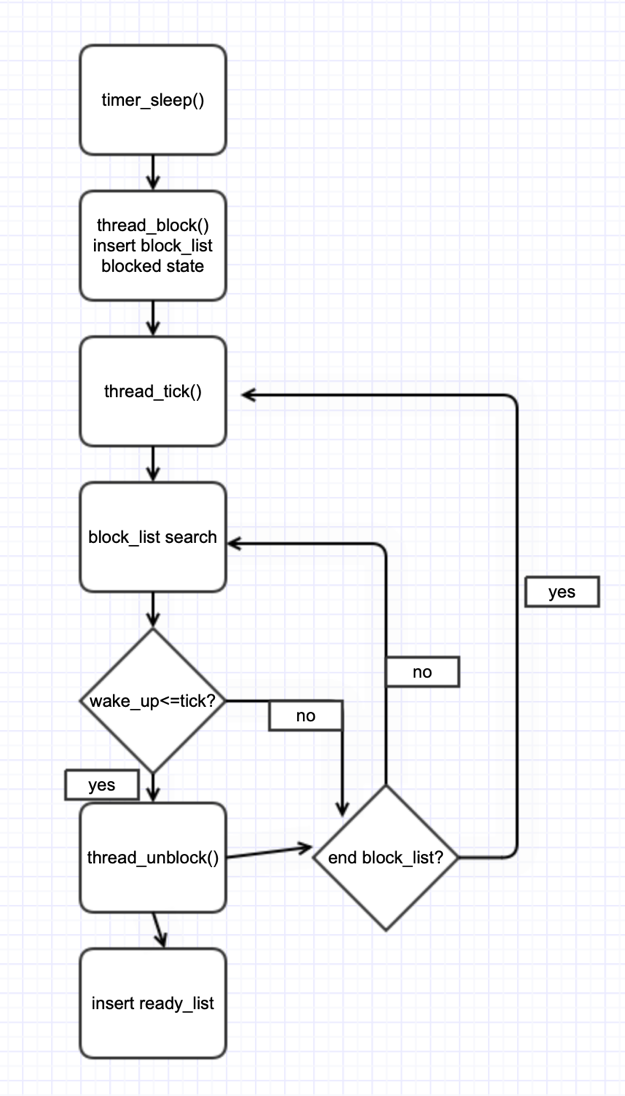
Threads/synch.c

여기서는 sema\_up 함수에서 priority에 따라 sema up 를 할 수 있도록 수정한다. Sema->waiters 에 대한 list를 정렬하고 가장 첫(우선순위가 가장 높은) thread를 뽑아 sema up 을 해주고 이를 다시 scheduling 에 반영하게 thread\_yield 함수를 쓰는 형태의 코드로 수정하면 된다.

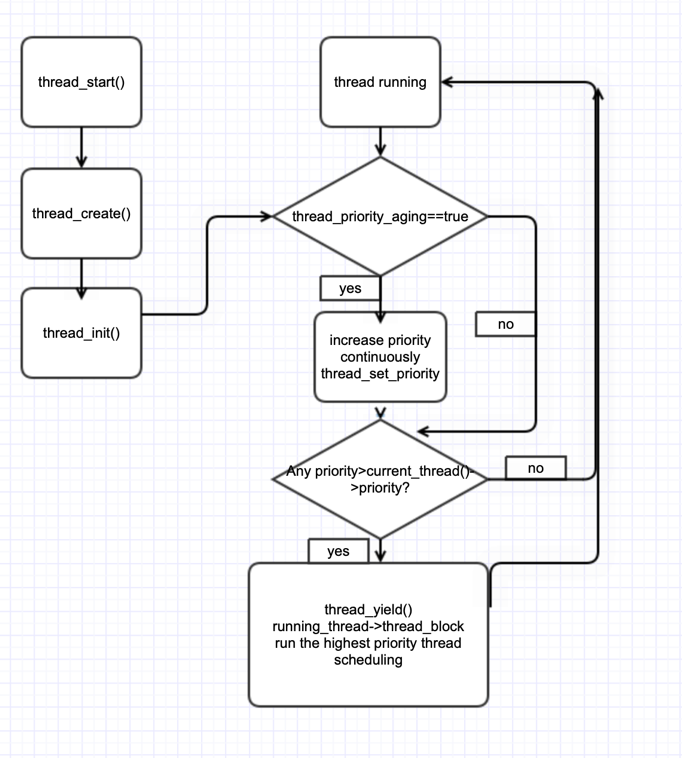
1. **연구 결과**
   1. **Flow Chart**

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

**1. alarm clock**

****

**2. 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

**설명하기에 앞서 자세한 코드의 내용은 사진 캡쳐로 대체하겠다.**

**1. Alarm clock**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Thread/thread.h

blocked 된 thread를 깨울지 말지 결정하는 변수인 wake\_up 변수를 thread 구조체 안에 기입한다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Thread/thread.c

해당 thread에 따른 wake\_up 변수를 0으로 초기화 시켜준다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

devices/timer.c

blocked\_list 라는 blocked 된 thread 를 담을 list 를 추가한다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Devices/timer.c

Blocked\_list 를 timer.init 함수에서 list\_init 함수를 통해 초기화를 시켜준다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Devices/timer.c

Timer\_sleep 함수에서 start와 Ticks를 current wake\_up 변수에 넣어준다. 그리고 thread\_current()의 elem 을 blocked\_list 에 집어 넣고 thread\_block으로 thread를 block 시킨다. (block state 로 변경) wake\_up 변수는 처음 cpu를 할당하고 메모리의 존재한 시간을 더한 변수이다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

Devices/timer.c

Timer\_interrupt 함수에서는 Blocked\_list를 순차적으로 돌면서 해당 thread의 wake\_up 변수가 ticks 보다 작거나 같으면 해당 thread를 blocked\_list 에서 지우고 이 thread를 thread unblock 해준다.

끝나고 thread\_tick() 함수를 불러 tick에 대한 계산을 다시 진행해서 tick update 를 한다.

**2. priority scheduling**

텍스트이(가) 표시된 사진

자동 생성된 설명

Threads/thread.c

Thread\_create 함수 마지막 부분에 현재 current\_thread 의 priority 를 가져와서 새로 만든 thread의 Priority 가 더 크면 thread\_yield 함수를 통해 cpu를 양보한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Threads/thread.c

Unblock 할때 ready list(queue) 를 정렬된 상태로 바꿔야 하기 때문에 list\_insert\_ordered 함수를 호출해서 thread가 priority 에 따라 적절 위치로 가도록 해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Threads/thread.c

Cmp\_pri 함수는 thread의 priority 를 서로 비교하여 왼쪽 thread의 priority가 오른쪽 thread의 Priority 보다 더 큰지 여부를 return 해주는 함수이다. 이는 list\_insert\_ordered 함수를 사용할 때 비교 함수로 쓰인다.

텍스트, 화면, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Threads/thread.c

Thread\_yield 할 때 thread를 queue에 넣을 때 ready list(queue) 를 정렬된 상태로 바꿔야 하기 때문에 list\_insert\_ordered 함수를 호출해서 해당 current thread가 priority 에 따라 적절 위치로 가도록 해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Threads/thread.c

만약 thread\_mlfqs 방식이 아니라면 현재 current\_thread 의 priority 를 가져와서 새로운 priority 로 넣어준다. 만약 ready list(queue) 가 비었다면 아무 일도 하지 않고 아니라면 만약 기존의 thread\_priority와 새로운 priority 를 비교해 기존의 thread\_priority 가 더 크면 thread\_yield 함수를 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Devices/timer.c

Timer\_interrupt 함수에서 추가적으로 prior\_aging 기법을 적용해야 한다. 현재 current thread의 recent\_cpu 에 1를 더해 이를 다시 현재 current thread 의 recent\_cpu 로 둔다. 그리고 매 시간 초를 확인해 timer\_FREQ 배수일때나 4의 배수일 때 thread\_aging\_fun 을 호출하여 다시 thread\_aging 에 대해 계산할 수 있도록 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Threads/thread.h

위에서 언급한 함수들에 대해 header 파일에 선언을 해주었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Threads/thread.h

여기서부터는 강의자료 ppt 31 페이지에 있는 aging 기법에 대한 선언들이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Threads/thread.c

강의자료 ppt 31 페이지에 있는 aging 기법에 대한 선언이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Threads/thread.c

Prior\_aging이 true 라면 thread\_aging\_fun 함수를 실행시켜 thread에 aging 기법을 줄 수 있도록 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Threads/init.c

만약 name이 -aging 이라면 prior\_aging을 true로 하여 thread.c 파일에 있는 thread\_tick 함수에서 thread\_aging\_fun이 실행되도록 해준다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Threads/synch.c

Sema\_up을 할 때 &sema->waiters 의 list가 비어있지 않다면 해당 sema->watiers 를 정렬해주고 가장 앞(우선순위가 가장 높은)의 element를 pop 한다. 그리고 해당 thread 를 thread\_unblock 해주었다. 그리고 thread\_yield 함수를 호출해 현재 current\_thread의 Priority 와 다시 priority 에 대한 검사를 진행한다.

**3. advanced scheduler(BSD scheduler)**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Threads/thread.h

새롭게 작성하는 함수들에 대해 선언을 헤더파일에서 해준다. 그리고 소수와 정수간의 계산을 하기 위해 필요한 FRACTION 도 define 한다. 14 bit은 fraction을 위한 공간이다.

그리고 static int load\_avg 를 선언하여 load\_avg 를 저장하는 int 형 변수를 둔다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Threads/thread.h

Thread struct 구조체 내부에 nice 변수와 recent\_cpu 변수를 두어 해당 변수들을 각 thread에서 사용할 수 있도록 선언한다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Threads/thread.c

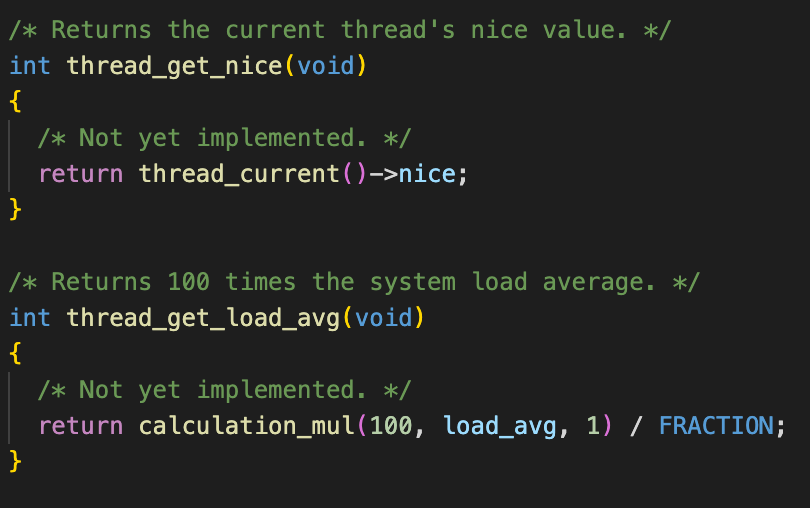
Init\_thread 함수에서 t->wake\_up 을 0으로 초기화 한 것 처럼 thread의 nice, recent\_cpu 값을 모두 0으로 초기화 시킨다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Devices/timer.c

Timer\_interrupt 함수는 이미 alarm, priority scheduling 구현 시 작성을 했지만 추가적으로 thread\_mlfqs 가 들어올때도 해당 aging 기법을 사용하도록 설정해야 한다. Recent\_cpu update, timer\_ticks()에 따라 timer\_freq 배수나 4의 배수라면 thread\_aging\_fun 함수를 실행 시킨다.(이는 위에 언급한 priority scheduling 에서와 동일하다.)

****

Threads/thread.c

Thread\_get\_nice 함수는 현재 thread\_Currnet() 의 nice 값을 반환하고 thread\_get\_load\_avg 함수는 load\_avg 값에 100을 곱한 값을 반환한다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Threads/thread.c

Thread\_get\_priority 함수는 현재 thread\_current의 priority 값을 반환한다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Threads/thread.c

Thread\_get\_recent\_cpu 함수는 현재 thread\_current 의 recent\_cpu 에 100을 곱하여 반환한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Threads/thread.c

Nice의 값은 -20~20 사이의 범위이다. 만약 넘어간다면 이를 재 조정해주고 paramter로 들어온 nice 값을 현재 thread\_current 의 nice 값으로 넣어준다.

그리고 현재 thread\_Current 의 nice값이 변경되었기 때문에 새로운 Priority 를 calculation\_priority 를 통해 계산하고 이를 다시 thread\_set\_priority 함수로 다시 새로운 우선순위로 세팅해준다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Threads/thread.c

Calculation\_recent\_cpu 함수는 Recent\_cpu=2\*load\_avg/(2\*load\_avg+1)\*recent\_cpu +nice 값을 계산하여 parameter로 받아온 thread의 recent\_cpu에 넣어주는 함수이다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Threads/thread.c

Calculation\_load\_avg 함수는 load\_avg=(59\*load\_avg+ready)/60 계산 과정을 통해 load\_avg 값을 다시 계산한다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Threads/thread.c

Thread\_aging\_fun 함수는 aging 을 구현하는 함수이다. Aging 이 1이면 4초 씩 priority 를 새롭게 업데이트를 해준다. All\_list 를 순차적으로 돌면서 새롭게 모든 thread의 priority 를 계산하여 새롭게 thread의 priority를 업데이트 해준다. 그리고 ready\_list 의 가장 앞 thread 를 가져와서 해당 Thread의 priority 와 현재 Thread\_current 의 priority 를 서로 비교하여 만약 thread\_Current 의 priority 가 더 작다면 interrupt 중이기 때문에 intr\_yield\_on\_return 함수를 호출한다.

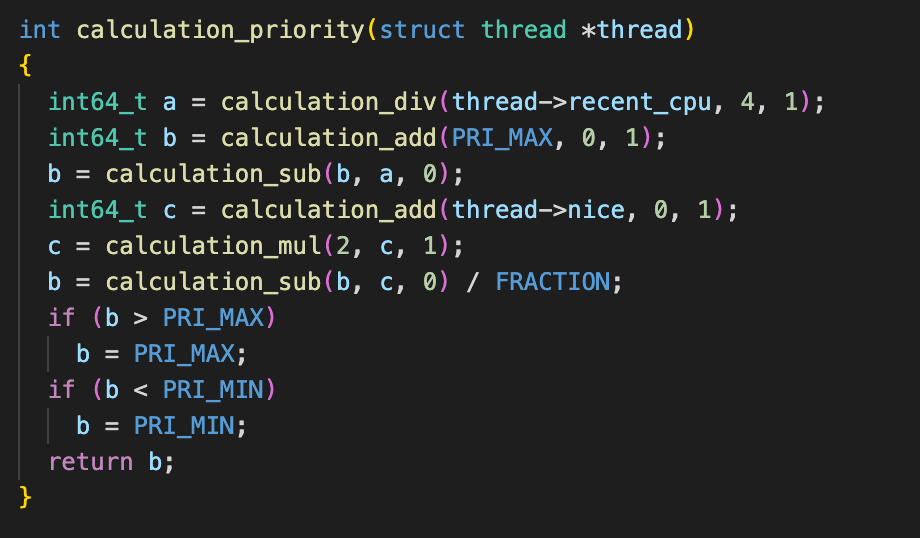
Aging 이 0 이면 매 초단위마다 load\_avg, recent\_cpu 값을 update 해주어야 한다. 만약 현재 thread\_Current 가 idle\_thread 가 아니라면 메모리에 있는 thread 이기 때문에 ready list(queue) 의 list\_size를 1 증가시켜 calculation\_load\_avg 함수에서 load\_avg를 계산해야 한다. Idle\_thread 라면 그대로 load\_avg를 계산한다. 그리고 for 문을 돌면서 순차적으로 모든 thread를 거치는데 thread 가 idle\_thread 라면 아무일도 하지 않고 아니라면 calculation\_recent\_cpu 함수를 통해 recent\_cpu 값들을 계산해야 한다.

**텍스트, 스크린샷, 검은색, 은색이(가) 표시된 사진

자동 생성된 설명**

Threads/thread.c

강의자료에 따르면 정수형과 실수형에 대한 계산을 구현하는 기능이 존재하지 않는다. 그래서 추가적으로 함수를 구현해줘야 한다. Flag 가 1이면 정수와 실수간의 계산을 진행하고 아니면 실수와 실수간의 계산을 하도록 기능을 구현하였다. 각 calculation\_add 는 덧셈 연산, calculation\_sub 는 뺄셈 연산, calculation\_mul 은 곱셈 연산, calculation\_div 는 나눗셈 연산을 하는 함수이다.

****

Threads/thread.c

Calculation\_priority 함수는 계산된 nice, recent\_cpu 값을 사용하여 실제 thread의 priority를 계산하는 과정이다. Priority=PRI\_MAX-(recent\_cpu/4)-nice\*2 계산식을 사용하면 된다. 그리고 해당 priority 가 pri\_max 보다 크거나 pri\_min 보다 작으면 각 각 최대 혹은 최소인 pri\_max, pri\_min 으로 지정한다.

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

priority\_lifo.c 코드

thread\_set\_priorty 함수에서 보면 priority 는 PRI\_DEFAULT 값에 (0~THREAD\_CNT)+1 값을 더한 것이다. 그래서 첫 main thread 에서의 자식은 실행이 안된다.

그리고 thread 의 priority 를 PRI\_DEFAULT 로 설정한다. 즉 priority 가 15 를 가지는 thread 를 시작으로 수행된다. Ready list(queue) 에서 이미 priority 로 정렬이 되어 있기 때문에 이후 14, 13, 12… 이런 식으로 순차적으로 우선순위에 맞게 thread 가 수행된다. 즉 각 16개의 thread 는 16번씩 반복적으로, 그리고 priority 가 높은 순서로 순차적으로 실행된다.

그래서 여러번 돌려도 같은 결과가 나오게 된다.

이 코드는 우선순위에 맞게 제대로 thread가 처리되는지 여부를 확인하는 test 코드이다. Priority 가 높은 thread가 나중에 ready\_list(queue)에 들어와 대기를 하고 있으면 해당 thread를 가장 먼저 실행시켜야 한다. 그리고 priority 가 가장 낮은 0 을 가지는 thread 는 가장 먼저 들어 왔음에도 불구하고 가장 나중에 수행됨을 해당 결과 화면을 통해 이해할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* make check 수행 결과를 캡처하여 첨부
* 텍스트이(가) 표시된 사진

  자동 생성된 설명