**Pintos Project 3: Threads**

담당 교수 : 김영재

조 / 조원 : 고동헌

개발 기간 : 10/30 ~ 11/06

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.
* 이번 프로젝트에선, timer\_sleep함수를 호출하여 sleep 상태로 들어간 thread를 깨워주는 alarm\_clock기능을 효율적으로 개선하고, 현재 Round Robin형태로 구현된 pintos의 스케쥴러를 각 thread간의 priority를 고려하도록 변경하는 것이 목적이다. 또한, priority scheduling 형태의 스케쥴러에서 발생할 수 있는 문제를 priority aging으로 해결하고, 추가구현으로 BSD 스케쥴러를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock
  2. Priority Scheduling
  3. Advanced Scheduler (추가구현을 한 경우)

1. **Alarm Clock**

* 기존 pintos에서, alarm clock 기능은 timer\_sleep 함수에 구현돼있다. 반복문을 통해 timer\_sleep함수가 호출된 시간과, 호출된 후 흐른 시간과의 차이를 계속해서 확인하면서 해당 thread가 sleep하고자 하는 시간과 비교한다. 더 작다면 thread\_yield함수를 호출하여 다시 ready\_list에 들어가고, 요청된 시간이 지났다면 while문을 탈출하여 다시 Thread가 실행된다.

이 방식의 문제점은, busy waiting 방식으로 thread가 ready 상태와 running상태를 계속해서 왔다갔다하기 때문에 굉장히 비효율적이라는 것이다. 따라서, sleep을 호출했을 때 요청 시간만큼의 시간이 흐르지 않았다면 해당 thread block 시켜놓고, 요청시간이 끝났을 때 block상태를 해제해 ready list에 다시 넣어줌으로써 이러한 비효율을 해결할 수 있다. 이를 통해, ready list에 쓰레드들이 없을 경우 idle thread가 실행될 수 있도록 하여, cpu utilization을 높일 수 있다.

1. **Priority Scheduling**

* Round Robin 스케쥴링 방식은 공평하다는 장점이 있지만, 각 thread 간의 우선순위를 고려하지 못한다는 단점이 있다. Priority scheduling 을 구현함으로써, 스케쥴링 시 각 thread들 간의 우선순위를 고려할 수 있게 된다. 우선 순위가 높은 thread는 낮은 thread에 비해서 더욱 자주 스케쥴링 돼, cpu를 많이 사용할 수 있다. 하지만, 이렇게 될 경우 starvation 문제가 발생할 수 있기 때문에 일정 주기마다 Ready list에 있는 thread들의 priority를 상승시켜, starvation 문제를 해결해야 한다.

1. **Advanced Scheduler**

* Advanced Scheduler의 경우, priority 스케쥴러와 같이 기본적으로 priority가 높은 thread를 실행시킨다. 하지만 Priority를 계산할 때, 해당 thread가 최근에 cpu를 얼마나 많이 사용했는지를 고려하고, 또한 추가적으로 각 thread의 nice값을 고려한다. 이를 통해 최근에 cpu를 많이 사용한 thread는 우선 순위가 낮아지게 되고, ready list에 오래 들어가 있어 cpu를 사용하지 못한 thread들은 우선순위가 높아지게 된다.

Recent cpu값을 고려하는 Advanced scheduler의 구현을 통해서, 일반적인 priority scheduler보다 더욱 실용적인 스케쥴링이 가능해진다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.
3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

* Thread가 Timer sleep함수를 호출할 때, 얼마만큼의 시간 동안 sleep 할지 parameter로 전달 해준다. 해당 함수가 호출된 시간을 start, 전달된 시간을 ticks 라고 하면 이 thread가 깨어나야 되는 시간은 (start+ticks)이다. thread struct에 waking\_time이라는 변수를 추가하여, timer\_sleep함수를 호출한 thread의 waking\_time에 (start+ticks)를 저장한다. 그리고, while문으로 이 시간을 확인하는 것이 아니라, 이렇게 sleep상태에 들어간 thread들을 blocked list 에 넣어준다.

Timer\_interrupt함수는 매 tick마다 호출된다. 따라서 위의 thread들을 깨우기 위해선, timer\_interrupt부분에서 blocked list를 확인하면서, 현재 시간과 blocked list에 있는 각 thread들이 깨어날 시간을 비교하면서, 만약 깨어날 시간이 지났다면 thread\_unblock함수를 호출하여 해당 thread 를 다시 Ready list로 넣어준다.

이렇게 blocked list와 timer interrupt를 이용하여, busy waiting방식이 아닌, 보다 효율적인 방식으로 block상태의 쓰레드들을 깨울 수 있다.

Running thread보다 높은 priority를 가진 thread가 Ready list에 들어올 경우, running thread는 cpu를 yield 해서, 자기보다 높은 priority를 가진 thread가 실행될 수 있도록 해야한다.

Ready list에, running thread보다 더 높은 priority를 가진 Thread가 들어갈 수 있는 함수는 크게 3가지가 있다. 첫 번째는, thread\_create 함수이다. Thread\_create는 새로운 thread를 생성해주는 함수로, 만약 이 함수에서 running thread의 priority보다 큰 Priority를 가진 thread 가 생성될 경우, 이 함수가 return 되기 전 thread\_yield함수를 호출하여 cpu를 내주어야 한다.

두 번째는, sema\_up함수이다. Sema\_up함수에선, 특정 semaphore가 release 될 때까지 기다리는 thread들이 waiter list에 들어가 있다. Semaphore가 release되면 이러한 waiting thread들 중 가장 priority가 높은 thread를 unblock시켜서 다시 ready list에 넣어준다. 이 때, 현재 thread의 priority보다 높은 priority를 가진 thread가 unblock돼서 ready list에 들어갈 수 있으므로, 이를 확인하고 thread\_yield를 호출해야 한다.

마지막으로, thread\_aging함수이다. Aging 스케쥴러를 사용할 경우, 매 timer interrupt마다 Ready list를 순회하면서 각 thread들의 priority를 1 증가시켜준다. 이 때 running thread보다 높은 priority를 가진 thread가 생길 경우, external fragmentation이 return 되기 전 running thread가 cpu를 yield 할 수 있도록 intr\_return\_on\_yield 함수를 호출하여, 다시 스케쥴링이 되도록 해야한다.

Advanced scheduler 에선, 시간이 지남에 따라 각 thread의 priority가 동적으로 변경된다. Priority를 계산하기 위해서, nice/recent\_cpu 를 이용한다. Priority 계산 공식은 다음과 같다.

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

Pintos에서 nice값은, -20~20까지의 값을 가진다. Priority계산 공식에서, nice값은 maximum priority값에서 빼지는 요소이다. 따라서, 매 4 tick마다 thread 들의 priority가 재계산될 때, 양수 nice값을 가지고 있는 thread들의 priority는 계속해서 감소하고, 음수 nice값을 가지고 있는 thread들의 priority는 계속해서 증가한다. Nice값이 0일 경우, priority에 영향을 끼치지 못한다.

Nice의 초기 값은, thread의 종류에 따라 다르다. Initial thread일 경우, 0으로 초기화되고, 다른 모든 thread들은 자신의 부모 thread의 nice값을 물려받게 된다. 또한, 실행중인 thread는 thread\_set\_nice함수를 호출하여 자신의 nice값을 동적으로 변경할 수도 있다.

Recent\_cpu는 특정 thread가 최근에 cpu를 얼마나 많이 차지해서 사용했는지를 나타내는 값이다. 이 값 역시, initial thread는 0으로 초기화 되며, 다른 thread들은 부모의 recent\_cpu값을 물려받게 된다. Recent\_cpu 계산 공식은 다음과 같다.

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

load\_avg는 각 시점의 ready to run상태인 thread들이 얼마나 있는 지를 나타내는, moving average이다. Load\_avg의 계산 공식은 다음과 같다.

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

Ready\_threads는 계산 시점에 ready to run상태인 thread들의 개수이다.

매 1초마다, recent\_cpu 값의 갱신이 일어나야 한다. Recent\_cpu는 load\_avg의 영향을 받으므로, recent\_cpu를 갱신하기 전 load\_avg값을 갱신하고, 이 값을 통해서 recent\_cpu 값을 갱신한다. 갱신된 recent\_cpu는 매 4 tick마다 재계산되는 thread의 priority에 영향을 미치게 되고, 이를 통해서 최근에 cpu를 사용하지 못한 thread들의 우선순위가 높아져 starvation이 방지되며, thread들 간의 cpu사용률을 고려한 스케쥴링이 가능해진다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

1. **Alarm Clock : 10/30 ~ 10/31**
2. **Priority Scheduler : 11/1 ~ 11/3**
3. **Advanced Scheduler : 11/3 ~ 11/5**
4. **Report : 11/5 ~ 11/6**
   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**

* **추가한 변수 및 함수**

1. **thread.h/struct thread/waking\_time**

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

실행중인 thread가 timer\_sleep함수를 호출할 경우, blocked list에서 깨어나야 할 시간을 저장해야 한다. Waking time변수에 이 값이 저장된다.

1. **thread.c/blocked list**

**폰트, 텍스트, 그래픽, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

Timer\_sleep 함수를 호출해 block된 thread 들이 저장되는 list이다.

1. **Thread.c/ thread\_sleep, thread\_wake\_up**

Thread\_sleep함수는 timer\_sleep함수에서 호출하는 함수로, timer\_sleep함수를 호출한 thread를 block list에 넣어주는 함수이다. 코드는 다음과 같다.텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Thread\_wake\_up함수는, thread\_ticks 함수에서 호출되는 함수로, block list에 들어있는 Thread 들 중, 깨어날 시간이 된 thread 들을 깨워주는 함수이다. 매 tick마다 호출되는 timer\_interrupt함수는 thread\_ticks() 함수를 호출하고, thread\_ticks함수에서 이 함수를 호출한다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Blocked list를 순회하면서, os부팅 후 흐른 시간인 ticks와 깨어나야 될 시간인 waking\_time과 비교해 가면서, 깨어날 시간이 지났으면 해당 thread를 깨워준다.

* 수정된 함수

1. **Devices/timer.c/timer\_sleep**

기존 busy waiting 방식으로 thread 들을 확인하였던 것과 달리, 내부적으로 thread\_sleep함수를 호출함으로써 이 함수를 호출한 Thread를 block list에 넣어준다. 코드는 다음과 같다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

1. **Threads/thread.c/thread\_ticks**

timer\_interrupt 함수에서 매 tick 마다 호출되는 함수이다. 기존 코드에 더불어, 내부적으로 thread\_wake\_up함수를 호출하여 block list에 있는 thread들 중 깨어날 시간이 된 thread들을 깨워준다. 기존의 parameter는 void type 이었지만, thread\_wake\_up 함수를 호출할 때 현재 시간인 ticks를 넘겨줘야 하므로, 이 함수도 parameter로 ticks를 받도록 변경됐다. 수정된 부분은 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. **devices/timer.c/timer\_interrupt**

thread\_ticks 함수의 parameter가 변경됨에 따라서, timer\_interrupt 함수에서 thread\_ticks 함수를 호출할 때 ticks를 함께 넘겨준다. 수정 부분은 다음과 같다.

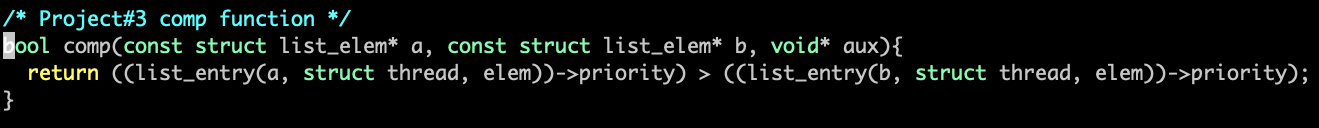


1. **Priority Scheduler**

* 기본적으로, aging을 구현하기 위해 ppt에 나온 부분은 모두 동일하게 추가/수정하였다.
* **추가한 함수**

1. **Threads/thread.c/comp**

기존 Round Robin scheduler의 scheduling함수에서, 다음에 실행할 thread를 가져오기 위해 ready list의 가장 앞에서 대기하고 있는 thread를 가져온다. Priority scheduling에서도 이를 그대로 이용할 수 있도록 ready list에 thread를 삽입할 때 각 thread 들이 priority의 내림차순으로 삽입될 수 있도록 하였다. 이는 구현돼 있는 api 인 list\_insert\_ordered 함수를 이용하면 된다. 이 함수를 사용하기 위해서 비교함수를 정의해서 넘겨 주어야 하는데, 이 비교 함수가 comp함수이다. 두 개의 list\_elem를 받아서, struct thread를 추출한 뒤 priority를 비교해서 return 해준다. 코드는 다음과 같다.



1. **Threads/thread.c/thread\_aging**

Priority scheduler에서 발생할 수 있는 starvation 문제를 해결하기 위한 함수이다. Thread\_ticks 함수에서 호출되며, 매 tick마다 ready list에 있는 thread들을 순회하면서 priority를 1 증가시켜준다. 이때 running thread보다 큰 priority를 갖는 thread가 생길 경우 true값을 return 해주어, thread\_ticks 함수에서 intr\_yield\_on\_return함수를 호출하도록 한다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* **수정된 함수**

1. **threads/thread.c/thread\_yield, thread\_unblock**

* 위의 함수들은, ready list에 thread 들을 push 해주는 함수들이다. Thread들은 priority의 내림차순으로 삽입돼야 하므로, 기존 list\_push\_back 함수에서 list\_insert\_ordered 함수를 호출하도록 변경됐다.

1. **Threads/thread.c/thread\_create**

* 새로운 thread 를 생성해주는 함수이다. 이 때 running thread보다 높은 priority 를 가진 thread가 생성됐을 경우, thread\_yield를 호출하여 새로 생성된 thread가스케쥴링 될 수 있도록 해야한다. 수정된 부분은 다음과 같다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

1. **threads/synch.c/sema\_up**

* semaphore에 의해 block된 thread 중 하나를 unblock해주는 함수이다. Priority scheduler에선 우선 순위가 가장 높은 thread가 가장 먼저 unblock 돼서 ready list에 들어가야 한다. 이 부분이 수정됐다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. **threads/thread.c/thread\_tick**

* 매 tick마다 thread\_tick이 호출되면, 이 함수에서 thread\_aging 함수를 호출하여 우선 순위를 증가시켜 준다. 이 때 return된 flag에 따라서 현재 실행 중인 thread를 yield 할 것인지 결정한다. 수정된 부분은 다음과 같다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

1. **Threads/thread.c/thread\_set\_priority, thread\_get\_priority**

* 현재 실행 중인 thread의 priority를 다른 값으로 set 하거나, 얻어오는 함수이다. thread\_set\_priority 함수로 인해 현재 thread 의 priority가 낮아졌다면, yield 함수를 호출해 다시 한 번 스케쥴링을 해야 한다. 코드는 다음과 같다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

1. **Advanced Scheduler**

* **추가된 변수 및 함수**

1. **Threads/thread.h/ nice, recent\_cpu**

Mlfq에서 priority를 계산하기 위해, 각 thread들의 nice값, Recent\_cpu값이 사용된다. 이를 저장해놓기 위한 변수이다.

**텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명**

1. **Threads/thread.c/load\_avg**

* Load\_avg역시, mlfq를 위한 변수이다. Recent\_cpu를 계산할 때 이용된다.

**텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명**

1. **Threads/thread.c/FRACTION, INT\_TO\_FLOAT, FLOAT\_TO\_INT, MUL\_FLOAT, DIV\_FLOAT**

* Priority를 계산하기 위해 사용되는 변수들 중, load\_avg, recent\_cpu 는 실수값을 가진다. 하지만 pintos는 floating point를 지원하지 않기 때문에, fixed point을 구현하여 실수를 표현해야 한다. 나는 공식 계산을 할 때, 우선 실수가 아닌 것들을 모두 실수 표현으로 바꾸고 계산을 한 뒤, 추후 이 값이 int로 사용되면 FLOAT\_TO\_INT 매크로를 통해 정수형으로 바꾸어주는 식으로 구현을 하였다. MUL\_FLOAT, DIV\_FLOAT는 실수값 끼리의 곱셈, 나눗셈을 위한 매크로이다. 이 매크로들은 pintos manual 을 참조하여 정의하였다.

**텍스트, 스크린샷, 폰트, 그래픽이(가) 표시된 사진

자동 생성된 설명**

1. **Threads/thread.c/**

**update\_priority\_per\_fourth\_tick, update\_recent\_cpu\_per\_seconds**

* Mlfq에서, 매 4 tick마다 모든 thread의 priority 를 재계산하고, 매 1 second 마다 recent\_cpu값을 재계산해주어야 한다. 각각은 이를 위한 함수들이다. 구현이 길어, B-2부분에서 자세히 설명하겠다.
* **수정된 함수**

1. **Devices/timer.c/timer\_interrupt**

* Mlfq 스케쥴러를 사용할 경우, pintos가 결정해놓은 주기마다 각 thread 들의 priority, recent\_cpu, running thread의 recent\_cpu 값을 변경해주어야 한다. 코드는 다음과 같다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

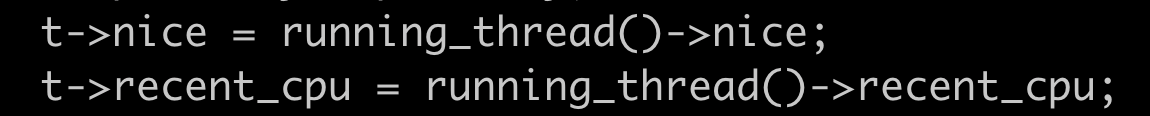
자동 생성된 설명**

1. **Threads/thread.c/thread\_init, thread\_create**

* 각 thread는 nice, recent\_cpu 값을 가지며 initial thread는 0값을 갖고, 그 외의 thread 들은 parent thread의 값을 물려받는다. Thread\_init 함수는 initial thread를 초기화 해주는 함수이고, thread\_create 함수는 새로운 thread 를 만드는 함수이므로, 이 부분을 수정하여 위의 사항을 구현하였다.

텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

****

1. **Threads/thread.c/thread\_set\_priority**

* 이 함수는 priority scheduling 을 구현할 때 이미 수정하였지만, mlfq까지 구현할 경우 추가적인 수정이 필요하다. Mlfq에서는 직접적으로 priority를 수정할 수 없고, 시간에 따라서 interrupt가 걸려 재조정 되거나, nice 값을 통해서 간접적으로만 Priority를 수정할 수 있다. 따라서, mlfq를 사용할 때 이 함수가 호출되면 아무것도 하지 않고 return 해주어야 한다. 수정된 부분은 다음과 같다.

**폰트, 그래픽, 텍스트, 타이포그래피이(가) 표시된 사진

자동 생성된 설명**

1. **Threads/thread.c/thread\_get\_nice, thread\_set\_nice**

* 각각 현재 실행중인 thread의 nice 값을 받아오고, 해당 thread의 nice값을 다른 값으로 변경해주는 함수이다. Thread\_set\_nice 함수에 의해서 현재 실행중인 thread의 Priority가 Ready list에 있는 thread보다 낮아질 수 있으므로, 이를 확인하는 과정이 필요하다. 자세한 구현은 뒤에서 설명하겠다.

1. **Threads/thread.c/thread\_get\_load\_avg, thread\_get\_recent\_cpu**

* 각각 현재 system의 load\_avg값, 현재 실행중인 thread의 recent\_cpu값을 계산해주는 함수이다. 100을 곱해서 return해 주어야 하고, 위의 값들은 실수값이므로 이를 정수로 변환해서 반환해주어야 한다. 코드는 다음과 같다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

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**

**텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명**

1. **Priority Scheduling**

**도표, 라인, 평면도, 기술 도면이(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. **Alarm Clock**

* Thread가 timer\_sleep함수를 호출하게 되면, timer\_sleep 함수에선 내부적으로 현재 시간인 start, sleep하고자 하는 시간인 ticks를 더한 값을 thread\_sleep 함수에 전달한다. Thread\_sleep함수는 다음과 같다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

Interrupt를 off시킨 뒤, 현재 thread의 깨어날 시간을 저장해야 하는 변수인 waking\_time에 parameter로 넘어온 값을 넣어준다. 그 후, sleep 상태에 들어간 Thread 들을 저장하는 list인 blocked\_list에 해당 thread를 넣어주고, thread\_block함수를 호출하여 다시 스케쥴링을 하게 된다.

Block list에 들어간 thread는, 깨어날 시간이 되면 다시 Ready list로 들어가야 한다. 이는 timer\_interrupt 함수를 통해서 구현된다. 매 Tick 마다 내부적으로 Thread\_tick함수를 호출하고, thread\_tick 함수에선 Thread\_wake\_up 함수를 호출한다. Thread\_wake\_up 함수에서 block list를 살펴보면서, 현재 시간과 sleep thread가 깨어날 시간과 비교를 하면서 깨어날 시간이 지난 thread 들을 깨워준다. Thread\_wake\_up 함수의 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Parameter로 들어온 Ticks 변수에는 현재 os 부팅 후 흐른 시간이 저장돼 있다. 또한, thread의 waking\_time에는 os 부팅 후 흐른 시간을 기준으로 언제 깨어나야 하는지 저장돼 있다. 따라서, waking\_time 이 ticks보다 작거나 같으면 깨어날 시간이 지난 것이므로, list\_remove함수를 호출하여 block list에서 해당 thread 를 제거해주고 Thread\_unblock 함수를 호출하여 해당 Thread 를 다시 ready list에 넣어줌으로써, sleep thread를 깨우는 과정이 완료된다.

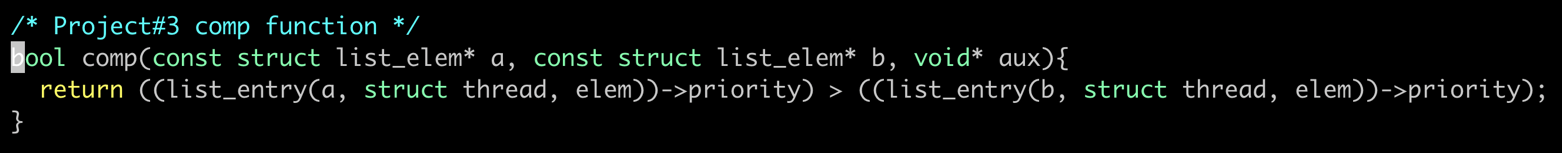
1. **Priority Scheduling**

각 thread의 우선 순위를 고려하여, 가장 높은 우선 순위를 가진 thread가 먼저 실행되게 하기 위해, 우선 next\_thread\_to\_run함수를 먼저 살펴보자.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

위 함수에선, ready list의 가장 앞에 있는 thread를 가져와서 scheduler함수에게 return 해준다. 이 함수를 그대로 이용하기 위해서, 각 thread 들이 ready list에 삽입될 때 priority 가 큰 Thread가 가장 앞에 삽입될 수 있도록 구현하였다. List에 원소를 삽입할 때, 특정 기준에 따라 원소를 삽입할 수 있도록 list\_insert\_ordered api가 이미 구현 돼있다. 이 함수를 사용하기 위해선 비교 함수를 하나 전달 해주어야 하고, 그 비교함수는 다음과 같다.

****

이 함수는 List elem인 a가 b보다 앞에 나올 때 true를 return 하도록 구현해야 한다. Priority 스케쥴링에선 priority가 높은 thread가 ready list의 앞에 나와야 하므로, a의 우선순위가 b의 우선순위보다 클 때 true를 return 하도록 구현한 것을 확인할 수 있다.

**// thread.c/thread\_unblock**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

**// thread.c/thread\_yield**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

위 함수들은, thread를 Ready list에 삽입해주는 함수들이다. 코드를 보면, 단순히 ready list의 가장 뒤에 원소를 삽입하는 것이 아닌, list\_insert\_ordered 함수를 호출해서 비교 함수인 comp를 기준으로 원소를 삽입하도록 하는 것을 확인할 수 있다. 이렇게 삽입을 함으로써, next\_thread\_to\_run 함수에서 항상 우선 순위가 가장 높은 Thread를 Return 할 수 있도록 한다.

하지만 위 함수 외에도, running thread보다 우선 순위가 높은 Thread가 들어올 수 있는 경우가 있다. 바로 Thread\_create 함수와, sema\_up함수이다. Thread\_create 함수에서는 새로운 Thread를 생성하기 때문에, 생성되는 thread 가 현재 실행 중인 Thread보다 priority가 클 경우 thread\_yield를 하여 새로운 thread가 실행될 수 있도록 해야 한다. 코드로 구현한 부분은 다음과 같다.

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

현재 실행중인 thread 의 Priority 를 받아오는 thread\_get\_priority 함수를 호출하여 비교를 하고, 새로 생성된 thread의 priority가 더 큰 경우 thread\_yield를 호출하는 것을 확인할 수 있다.

다음은 sema\_up 함수이다. 코드는 아래와 같다.

**텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

특정 semaphore를 waiting하고 있는 thread 들 중 하나를 unblock 시킬 때, priority 스케쥴링 방식에 의하면 우선 순위가 가장 높은 thread를 제일 먼저 unblock 시켜주어야 한다. Max\_pri\_elem, max\_pri\_thread는 이를 구현하기 위한 변수들이다. While문을 통해 waiter 중 가장 높은 우선 순위를 가지고 있는 thread를 찾고, 이를 max\_pri\_thread 변수에 저장한다. 또한 해당 List\_elem는 max\_pri\_elem 변수에 저장하여, 해당 Thread를 sema waiter list에서 제거해준다. 또한, max\_pri\_thread는 unblock돼 ready list에 들어갔으므로, 만약 해당 thread의 priority가 현재 실행 중인 thread의 priority보다 크다면 thread\_yield함수를 호출하여 다시 스케쥴링을 진행한다. 이 부분은 마지막 if 문에 구현돼있다.

위를 모두 구현하면, priority 스케쥴링이 가능해진다. 하지만 starvation이 생길 수 있다는 문제점이 생긴다. 이는 priority aging을 통해 해결할 수 있다. 매 tick마다 timer\_interrupt에 의해서 thread\_ticks함수가 호출돼고, aging 스케쥴러를 사용할 경우 thread\_ticks 함수에선 thread\_aging 함수를 호출한다. Thread\_aging 함수에선, ready Thread 들의 priority 를 1 증가시켜준다. thread\_aging 함수의 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Ready list를 순회하면서, 각 thread의 priority를 1 증가시켜주는 것을 확인할 수 있다. 이 때 PRI\_MAX(63) 보다 priority가 커질 수 있으므로, if문을 통해 해당 상황을 방지하였다. 또한, Priority aging에 의해 현재 실행 중인 Thread보다 더 높은 Priority를 가진 Thread가 생길 수도 있다. yield\_flag 변수는 이를 위한 변수이다. 만약 그러한 Thread가 생기면, yield\_flag를 true로 바꿔주고, 이 Flag를 return 해주어 다시 스케쥴링이 일어날 수 있도록 한다. Thread\_aging함수를 호출하는 Thread\_ticks함수의 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Thread\_aging()함수를 호출한 뒤, return 값이 True 이면 intr\_yield\_on\_return 함수를 호출해 external interrupt가 종료되기 전 다시 한 번 스케쥴링이 일어날 수 있도록 한다. Return 값이 True란 것은, 위에서 설명한 것처럼 현재 thread 보다 높은 priority를 가진 thread가 생겼다는 것을 말한다.

1. **Advanced Scheduling**

* Advanced Scheduling의 설명에 앞서, 매크로 먼저 살펴보자. 매크로는 다음과 같다.

**텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명**

Pintos에선, msb를 sign bit로 사용하고, 나머지 31bit를 이용하여 17-14 floating point로 수를 표현한다. 따라서 FRACTION은 (1<<14)가 돼야한다. INT\_TO\_FLOAT, FLOAT\_TO\_INT 는 매크로 이름에서 알 수 있듯, 각각 정수를 실수로, 실수를 정수로 바꿔주는 매크로이다. MUL\_FLOAT, DIV\_FLOAT는 각각 실수끼리의 곱, 나눗셈을 계산해주는 매크로이다. 계산 과정에서 곱셈을 하면서 오버플로우가 날 수 있기 때문에, 64bit로 캐스팅해서 계산해준다. 나는 recent\_cpu, load\_avg를 계산할 때, 필요한 모든 값들을 실수로 바꾼 뒤 계산하였기 때문에 다른 매크로는 정의 하지 않았다. 이 매크로들은 모두 pintos manual에 따라 작성하였다.

Mlfq를 구현하기 위해선, 매 주기마다 모든 thread들의 priority, recent\_cpu값들을 갱신해주어야 한다. 이를 위해서, timer\_interrupt함수에서 ticks값을 통해서 주기를 체크하고, 갱신하는 주기일 경우 함수를 호출하여 값을 갱신해준다. Timer\_interrupt코드는 다음과 같다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

메뉴얼에 의해, running thread의 recent\_cpu는 매 tick마다 1씩 증가해야 한다. 첫 번째 줄은 이를 위한 코드이다. 이 때 (1<<14)를 더하는 것은, recent\_cpu가 실수값이기 때문이다. 1을 실수값으로 변환하면 (1<<14)이기 이므로, 이 값을 recent\_cpu에 더해준다. 또한, 각각의 if 문을 통해서 메뉴얼이 제시한 주기에 따라 Priority, recent\_cpu를 계산하는 함수를 호출하는 것을 확인할 수 있다.

Update\_priority\_per\_fourth\_tick 의 코드는 다음과 같다.

**텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

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

Priority 의 계산 공식은 위와 같다. 코드에서, all\_list에 있는 모든 thread들을 돌며 priority를 계산하는 것을 확인할 수 있다. 위 식에서 PRI\_MAX, nice값은 정수이다. 모든 정수값들을 실수로 통일해서 계산을 하였기 때문에, 코드를 보면 pri\_max, nice값 계산시 INT\_TO\_FLOAT 매크로를 사용하였다. 또한, 최종 계산 결과인 priority는 정수값이기 때문에, 위 공식을 적용한 결과값에 FLOAT\_TO\_INT 매크로를 적용해 갱신되는 priority를 계산해주었다. 이때, thread의 priority가 제한된 범위가 넘어가지 않도록 if문을 통해 값을 조정해주었다. 또한, priority가 갱신되면서 running thread보다 높은 priority를 가진 thread가 나올 경우, yield\_flag를 true로 설정해주고, 반복문이 끝났을 때 이 flag가 true라면 intr\_yield\_on\_return함수를 호출해 external interrupt가 끝나기 전 thread\_yield를 호출하여, 다시 스케쥴링이 일어날 수 있도록 하였다.

Update\_recent\_cpu\_per\_seconds함수의 코드는 다음과 같다.

**텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

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

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

Recent\_cpu의 갱신을 위해선, load\_avg를 먼저 갱신해주어야 한다. 코드를 보면, Ready\_thread의 수를 구하고, 이 값을 INT\_TO\_FLOAT를 통해 float값으로 변경한 후, 위의 공식에 따라 load\_avg값을 갱신한 것을 확인할 수 있다.

For문을 통해, 각 thread의 recent\_cpu값을 갱신해주었다. 이 때 idle thread인 경우는 갱신하지 않았다. 나는 recent\_cpu값을 계산하기 위해 first\_part, second\_part 변수를 선언하여 계산해주었다. 코드에서, first\_part 부분의 첫 번째 라인은 ***(2 \* load\_avg) / (2 \* load\_avg + 1)*** 를 계산한 부분이다. DIV\_FLOAT 매크로를 이용하여 실수의 나눗셈을 해주었다. 두 번째 라인은 앞에서 계산한 값에, recent\_cpu를 곱해준 것이다. 따라서 First\_part에는 ***(2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu*** 값이 들어가게 된다. Second\_part에는 nice를 실수값으로 변환한 값이 들어간다. 코드를 보면 확인할 수 있다. 최종적으로, 두 개의 값을 더해서 해당 Thread의 recent\_cpu값을 갱신해주게 된다.

Mlfq 에서 각각의 thread 의 priority는 직접적으로 조정하지 못하고, 시간에 따라 조정되거나 nice값으로 간접 조정만 가능하다. 따라서 thread\_set\_priority 함수에서 mlfqs 스케쥴러를 사용할 경우 바로 리턴하도록 하였다. 코드는 다음과 같다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

실행중인 thread는 nice 값을 조정해서 간접적으로 priority를 변경할 수 있다. Thread\_set\_nice함수는 다음과 같다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

코드를 보면, 위의 update\_priority\_per\_fourth\_tick 함수처럼, priority 공식에 들어가는 값을 실수로 바꿔주고, 사칙연산 후 다시 정수로 바꿔주어 갱신하는 것을 확인할 수 있다. 이때도, priority가 범위를 벗어날 수 있으므로 값을 조정해주어야 한다. 또한, 현재 실행중인 thread의 priority값이 ready list의 가장 높은 priority를 갖는 thread보다 낮아질 수 있으므로, 마지막 if문에서 이를 확인하여 thread\_yield함수를 호출해주었다.

Thread\_get\_nice함수는 현재 실행중인 thread의 nice값을 리턴해주는 함수이다. 코드는 다음과 같다.

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

단순히 thread\_current 함수를 호출해 현재 실행중인 thread를 return받고, 해당 thread의 nice값을 리턴해준다.

Thread\_get\_load\_avg 함수, thread\_get\_recent\_cpu함수는 각각 현재 시스템의 load\_avg값, 현재 실행중인 thread의 recent\_cpu값에 100을 곱하여 리턴해주는 함수이다. 이때 정수로 리턴을 해주어야 하기 때문에, 각각 값에다가 100을 곱하여 FLOAT\_TO\_INT 매크로를 적용한 값을 리턴해주었다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

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

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Priority-lifo 코드를 보면, 우선 순위가 서로 다른 thread들을 THREAD\_CNT 개수(16)만큼 생성하고 thread 루틴으로 simple\_thread\_func함수를 실행한다. 이 때 루프를 돌면서 더 높은 우선 순위를 가지는 thread들이 생성되므로, simple\_thread\_data의 멤버인 id값이 높을 수록 우선 순위 또한 높을 것이다. Simple\_thread\_func 코드는 다음과 같다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

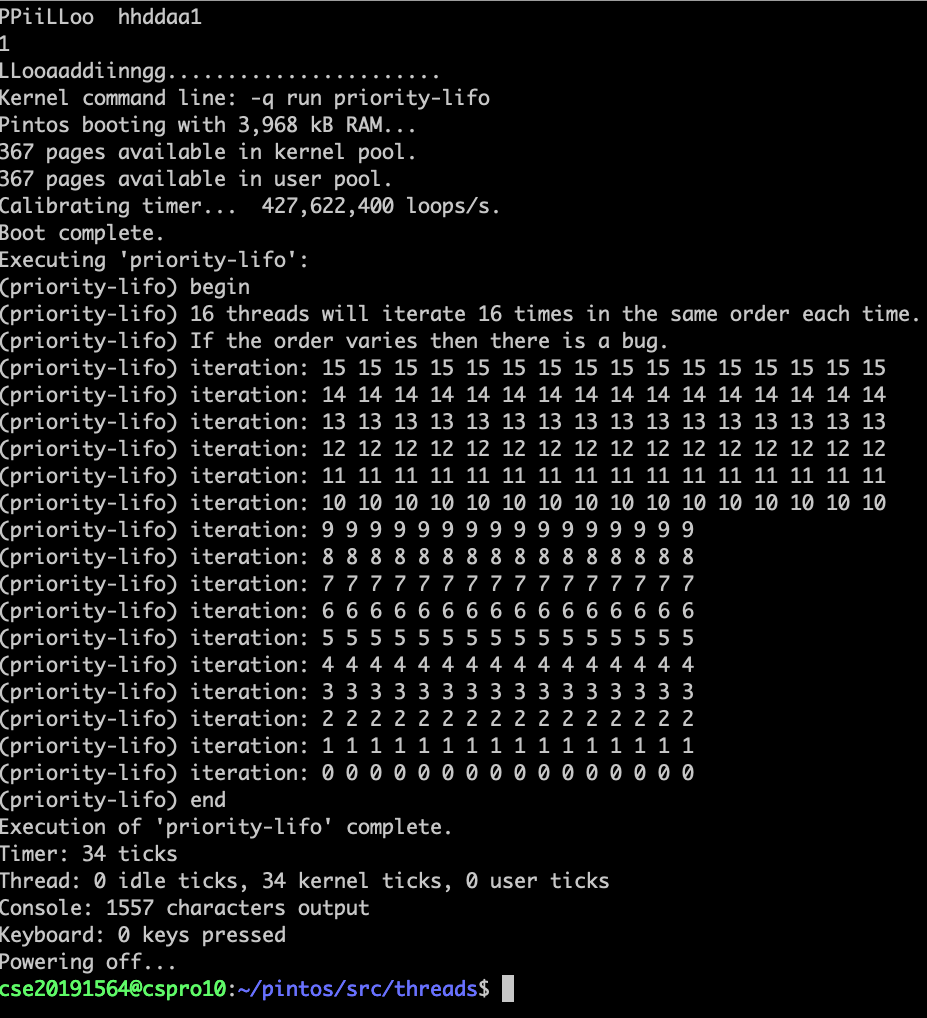
ITER\_CNT인 16만큼 반복하면서, 매 반복문의 끝에 thread\_yield 함수를 호출한다. 이때 thread 루틴의 인자로 넘어온 data가 가리키는 값에, 현재 실행중인 thread 의 id를 저장한다.

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

이 코드는 첫번째 사진 아래의 코드이다. 맨 윗 줄을 보면, 현재 main thread의 priority를 PRI\_DEFAULT값으로 설정하고 있다. 따라서 자신이 생성한 모든 thread들보다 우선순위가 낮아져, 모든 thread들이 종료되고 아래의 코드가 실행된다. 반복문을 돌면서, d에 적혀있는 id를 순차적으로 하나씩 출력해주는데, priority scheduler이 올바르게 구현되면 가장 우선순위가 높은 thread들 부터 16번씩 실행되며 Data field에 자신의 id를 넣을 것이고, 이 반복문에선 그것을 출력하므로 가장 높은 id인 15부터, 가장 낮은 id인 0까지 16번씩 출력될 것이다. 실제 실행 결과는 아래와 같다.

* **실행 결과**



Thread들의 우선 순위대로 잘 실행된 것을 확인할 수 있다.

* make check 수행 결과를 캡처하여 첨부

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

자동 생성된 설명