**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20171659 / 유경연

개발 기간 : 10/27~11/4

1. **개발 목표**.

이번 프로젝트에서는 현재 Round Robin으로 구현되어 있어서 priority를 고려하지 않는다. Thread Scheduling을 priority를 반영하는 scheduler로 변경한다. 또한 aging을 통해 priority를 증가시켜줄 수 있도록 구현한다.

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

현재 timer\_sleep() 함수는 thread\_yield()를 호출해서 ready 상태로 바꾸는 busy-wating 방식으로 구현되어 있다, 따라서 CPU 자원의 낭비가 일어나게 된다. 이 방식을 바꾸어 아직 blocked 상태에서 벗어날 시간이 되지 않은 thread를 계속 block 상태를 유지하게 하여 CPU 자원을 효율적으로 사용할 수 있다.

* 1. Priority Scheduling

현재 pintos의 scheduler는 Round Robin으로 구현되어 있다. 따라서 우선순위가 고려되지 않기 때문에 fairness는 높지만 효율적이지 못하다. 우선순위를 고려하여 scheduling하게 되면 효율적인 scheduling이 가능하여 turnaround time이 개선될 것이다. 또한 starvation 문제를 해결하기 위해서 aging을 도입한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

단순히 우선순위 scheduling을 하는 경우 여러 문제가 발생할 수 있고 대표적으로는 starvation과 game the scheduler 문제들이 있다. 이런 문제들은 scheduler를 개선해서 해결할 수 있는데 이번 프로젝트에서는 BSD scheduler를 구현한다. BSD scheduler는 여러 요소를 이용해 우선순위를 재계산하여 이런 문제를 해결한다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Blocked 상태의 thread를 저장하는 list를 만들어서 thread가 blocked 상태로 바뀔 경우에 이 list에 저장한다. 이후 매 tick마다 list를 확인해서 조건을 만족한 경우에 ready 상태로 바꾼다. List에 저장할 때 오름차순으로 저장한다면 더욱 효율적으로 list를 사용할 수 있다.

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

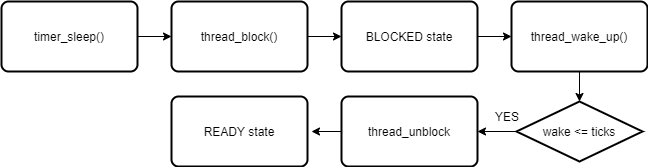
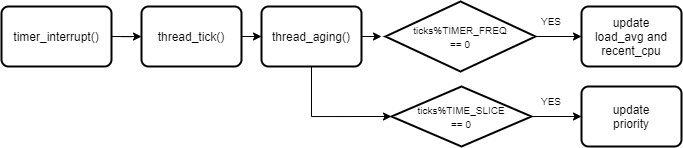
Preemptive하게 scheduling하는 경우에 현재 running thread보다 priority가 더 높은 thread가 들어오면 현재 running thread를 blocked 상태로 변경하고 새로운 thread를 running 상태로 설정한다. 이를 수행하기 위해서 priority가 높은 thread가 들어오는 경우에 thread들을 새로 scheduling한다.

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

이번 프로젝트에서는 추가로 BSD scheduler를 구현한다. 이 scheduler는 MLFQ의 구성을 갖는다. 이때 priority를 새로 계산하기 위해서 몇가지 요소가 필요하다. 첫번째 요소로는 niceness가 있다. 이 요소는 한 thread가 얼마나 다른 thread에게 CPU를 양보하는지에 대한 값이다. 이 값은 -20에서 20사이의 값을 같고 값이 작을수록 다른 thread에게 양보하는 정도가 크고 값이 클수록 다른 thread로부터 CPU를 빼앗아 오는 정도가 크다. Pintos 상에서는 nice로 정의한다. 두번째는 calculating priority다. BSD scheduler는 우선순위를 0(PRI\_MIN)부터 64(PRI\_MAX)까지의 값으로 나눈다. 이 값이 클수록 우선순위가 높다. 이 값은 4 tick마다 새로 계산한다. 세번째로 recent\_cpu가 있다. Recent\_cpu는 thread가 최근에 CPU를 얼마나 사용했는지 나타내는 값이다. 이 값은 매 초마다 새로 계산된다. 네번째는 load\_avg 값이다. 이 값은 평균적으로 몇 개의 thread가 ready 상태인지 추정한다. 이 값 또한 매초마다 새로 계산된다. 마지막으로 이런 요소들을 새로 계산하기 위해서 몇가지 계산식이 사용되는데 recent\_cpu나 load\_avg가 실수 값이기 때문에 실수 값 연산을 처리해야할 필요가 있다. Pintos에서는 부동 소수점 연산을 지원하지 않기 때문에 고정 소수점 연산을 사용해야한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 10/27 : Alarm clock 구현
* 10/29 : Priority 구현
* 11/01 : Priority Semaphore 구현
* 11/02 : Aging 구현
* 11/03 : Alarm 오류 수정
* 11/04 : BSD 추가 구현
  1. **개발 방법**
  2. Alarm Clock
     + 수정하거나 추가해야 하는 자료구조
       - Devices/timer.c : blocked된 thread를 저장하는 리스트를 만든다.
       - Threads/thread.h : thread를 언제 wake up 시켜야 하는지를 저장하기 위해 변수를 추가한다.
     + 수정하거나 추가해야 하는 함수
       - Devices/timer.c : timer\_sleep()를 수정해서 thread를 blocked 상태로 만든다
       - Devices/timer.c : 리스트를 오름차순으로 관리하기 위한 함수를 추가한다.
       - Threads/thread.c : thread를 wake up 시킬 시간이 된 경우 wake up 시키는 함수를 추가한다.
  3. Priority Scheduling
     + 수정하거나 추가해야 하는 자료구조
       - Threads/thread.c, Threads/thread.h : bool thread\_prior\_aging을 추가한다. 이 변수는 priority aging을 위해서 필요하다.
     + 수정하거나 추가해야 하는 함수
       - Threads/init.c : parse\_option()을 수정해서 –“aging”을 입력 받았을 때 적절한 수행을 할 수 있도록 한다.
       - Threads/thread.c : thread\_unblock()을 수정하여 thread가 unblock 되는 경우 ready queue에 priority를 반영하여 넣는다.
       - Threads/thread.c : ready list를 내림차순으로 관리하기 위한 함수를 추가한다.
       - Threads/thread.c : thread\_yied()를 수정해서 thread를 ready 상태로 만들 때 우선순위를 고려할 수 있도록 한다.
       - Threads/thread.c : thread\_create()를 thread가 새로 생성되었을 때 preemption이 가능하도록 수정한다.
       - Threads/thread.c : thread\_set\_priority()를 수정해서 입력 받은 우선순위로 thread의 우선 순위를 변경할 수 있도록 한다.
       - Threads/thread.c : thread를 aging해주는 함수를 추가한다.
       - Threads/thread.c : 현재 가장 높은 우선순위를 찾는 함수를 추가한다.
       - Threads/synch.c : semaphore up, down 함수에 우선순위를 고려할 수 있도록 함수를 수정한다.
  4. Advanced Scheduler (추가구현을 한 경우)
     + 수정하거나 추가해야 하는 자료구조
       - Threads/thread.h : thread 구조체에 recent\_cpu와 nice 변수를 추가한다.
       - Threads/thread.c : load\_avg 변수를 추가한다.
     + 수정하거나 추가해야 하는 함수
       - Threads/thread.c : load\_avg와 recent\_cpu를 업데이트해주는 함수를 작성한다.
       - Threads/thread.c : priority를 새로 계산해주는 함수를 작성한다.
       - Threads/thread.c : thread\_init(), init\_thread()에서 recent\_cpu와 nice 값을 초기값으로 초기화한다.
       - Threads/thread.c : thread\_get\_nice(), thread\_get\_load\_avg(), thread\_get\_recent\_cpu()함수를 수정한다.
       - Threads/thread.c : thread\_set\_nice() 함수를 수정한다.

1. **연구 결과**
   1. **Flow Chart**
      1. Alarm Clock
      2. Priority Scheduling
   2. **제작 내용**
2. Alarm Clock
   * + 수정하거나 추가해야 하는 자료구조
       - Devices/timer.c : list 구조체를 사용해 alarm\_list를 만들어 blocked된 thread를 관리할 수 있도록 했다. 이때 wake 값에 대해 오름차순으로 thread를 저장했다. 이 변수는 thread\_init() 함수에서 초기화했다.
       - Threads/thread.h : thread 구조체에 언제 thread\_wake\_up()을 호출할지 저장 하는 wake 변수를 추가했다. 이 값은 int\_thread()함수에서 0으로 초기화한다.
     + 수정하거나 추가해야 하는 함수
       - Devices/timer.c : timer\_sleep()함수를 수정했다. 이 함수는 thread의 wake 변수에 값에 언제 이 thread를 wake up할지 에 대한 값을 할당하고 alarm\_list에 오름차순으로 저장한다. 이후 thread\_block()함수를 호출해서 thread를 blocked 상태로 만든다.
       - Devices/timer.c : list\_insert\_ordered()함수의 인자로 호출될 compare\_wake()함수를 추가한다. 이 함수는 thread가 alarm\_list에 삽입될 때 wake 값을 기준으로 오름차순으로 저장될 수 있도록 한다.
       - Threads/thread.c : Thread를 unblock하는 thread\_wake\_up() 함수를 새로 추가한다. 이 함수는 alarm\_list에 저장되어 있는 thread들을 검사하여 wake 값이 현재 ticks보다 작은 경우 thread\_unblock()을 호출한다.
3. Priority Scheduling
   * + 수정하거나 추가해야 하는 자료구조
       - Threads/thread.c, Threads/thread.h : PDF와 동일하게 bool thread\_prior\_aging을 추가했다.
     + 수정하거나 추가해야 하는 함수
       - Threads/init.c : PDF와 동일하게 parse\_option()함수를 수정했다.,
       - Threads/thread.c : PDF와 동일하게 thread\_tick()함수를 수정했다.
       - Threads/thread.c : thread\_unblock()함수가 호출되면 thread를 ready\_list에 priority를 기준으로 내림차순으로 삽입하고 ready 상태로 변경한다..
       - Threads/thread.c : list\_insert\_ordered()함수의 인자로 호출되는 compare\_priority()함수를 추가했다. 이 함수는 thread의 priority를 비교해서 ready\_list에 내림차순으로 저장해서 priority scheduler를 가능하도록 한다.
       - Threads/thread.c : thread\_yied()를 수정해서 현재 thread가 idle하지 않는다면 ready\_list에 삽입하여 ready 상태로 만들고 schedule()함수를 호출한다. Schedule()함수가 호출되면 현재 실행되고 있는 thread와 ready\_list의 가장 앞에 있는 thread가 교체된다.
       - Threads/thread.c : thread\_create()를 thread가 새로 생성되었을 때 새로 들어온 thread의 priority가 현재 실행 중인 thread보다 높다면 thread\_yield()함수를 호출해서 다시 scheduling한다..
       - Threads/thread.c : thread\_set\_priority()를 수정해서 입력 받은 우선순위로 thread의 우선 순위를 변경한다. 이때 새로운 priority가 현재 priority보다 작다면 thread\_yield()함수를 호출해서 다시 scheduling한다. 이때 thread\_mlfq가 true인 경우엔 실행하지 않는다.
       - Threads/thread.c : Thread를 aging하는 thread\_aging()함수를 추가했다. 현재 ticks가 TIMER\_FREQ로 나누어 떨어지는 경우 load\_avg와 recent\_cpu를 업데이트하기 위해 update\_load\_avg\_and\_recent\_cpu()함수를 호출하고 TIME\_SLICE로 나누어 떨어지는 경우엔 update\_priority()함수를 호출한다. 이 함수들은 load\_avg, recent\_cpu, priority를 새로 계산해주는 함수로 뒤에서 자세하게 설명한다.
       - Threads/thread.c : 현재 가장 높은 priority를 찾는 find\_max\_priority()함수를 추가했다. Ready\_list에 내림차순으로 저장했으므로 ready\_list가 비어있지 않다면 가장 앞에 저장되어 있는 priorty가 가장 클 것이므로 가장 앞의 priority를 return한다.
       - Threads/synch.c : Sema\_up()를 수정해서 priority를 고려할 수 있도록 한다. 현재 semaphore가 up되기를 기다리는 thread가 저장되어 있는 sema->waiters 리스트를 탐색해서 리스트 안에서 가장 priority가 큰 thread를 찾는다. 이 thread를 unblock시키고 thread\_yield()함수를 호출해서 다시 scheduling한다.
4. Advanced Scheduler (추가구현을 한 경우)
   * + 수정하거나 추가해야 하는 자료구조
       - Threads/thread.h : 고정소수점 연산을 위해서 FRACTION을 새로 define하였다. 고정 소수점 연산에서 문제가 되는 것이 정수와 실수의 연산인데 정수에 2^14를 곱해주어서 문제를 해결했다. 예를 들어서 정수와 실수를 더하는 경우엔 정수에 2^14를 곱해 실수로 바꾸어 실수와 덧셈을 수행한다.
       - Threads/thread.h : thread 구조체에 recent\_cpu와 nice 변수를 추가했다.
       - Threads/thread.c : load\_avg 변수를 전역변수로 추가했다..
     + 수정하거나 추가해야 하는 함수
       - Threads/thread.c : thread\_tick()함수에서 thread가 idle 상태가 아닐 때 recent\_cpu값을 1 증가시켰다.
       - Threads/thread.c : load\_avg와 recenct\_cpu를 새로 계산해주는 update\_load\_avg\_and\_recent\_cpu()함수를 추가했다. 이 함수는 우선 load\_avg 값을 새로 계산하는데 현재 thread가 idle 상태가 아닌 경우 ready\_threads 값을 1 증가시키고 load\_avg 값을 새로 계산한다. 사용하는 식은 다음과 같다. ready\_threads의 값이 정수이므로 FRACTION을 곱해서 계산한다.



이후 모든 idle 상태가 아닌 thread의 recent\_cpu를 새로 계산한다. 사용하는 식은 다음과 같다. 이 식 또한 정수값에는 FRACTION을 곱해서 계산한다. 또 overflow를 방지하기 위해서 int\_64로 형변환을 해주었다. 

* + - * Threads/thread.c : 모든 thread의 priority를 새로 계산해주는 update\_priority()함수를 추가했다. 사용된 식은 다음과 같다. 정수 값엔 FRACTION을 곱하여 계산하고 priority 값은 정수 이므로 shift operation을 사용하여 정수 값으로 변환하였다.



새로 계산된 priority가 현재 가장 큰 priority보다 작아지는 경우엔 intr\_yield\_on\_return()함수를 호출하였다. 이때 thread\_yield()함수를 사용할 경우 intr\_context()함수가 제대로 실행되지 않아서 Kernel panic이 발생한다.

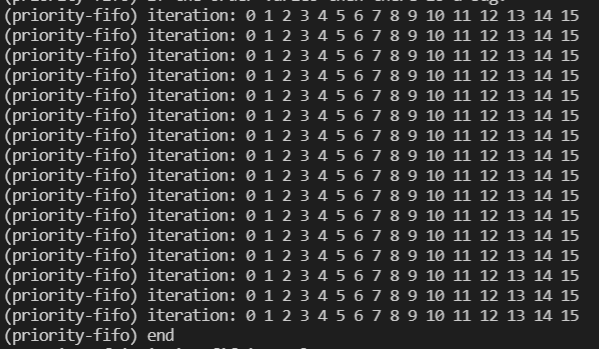
* + - * Threads/thread.c : thread\_init()에서 initial\_thread()의 nice와 recent\_cpu값을 0으로 초기화했다.
      * Threads/thread.c : init\_thread() 함수에서 nice와 recent\_cpu 값을 부모 thread의 값으로 초기화했다. ,
      * Threads/thread.c : thread\_get\_nice(), thread\_get\_load\_avg(), thread\_get\_recent\_cpu()함수를 수정해서 현재 thread의 값을 반환하도록 했다. Load\_avg와 recent\_cpu의 경우 실수이므로 100을 곱해서 FRACTION으로 나눈 값을 반환했다.
      * Threads/thread.c : thread\_set\_nice() 함수를 수정해서 입력 받은 값으로 nice값을 변경하고 nice값이 바뀌었으므로 새로 priority를 계산하였다. 사용한 식은 위에서 사용한 식과 같다. Priority가 작아진 경우엔 thread\_yield()를 호출했다..
  1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

priority\_lifo의 경우 모든 thread의 priority를 똑같이 생성하는 priority\_fifo와 다르게 다음과 같이 thread를 생성할 때 나중에 생성하는 thread의 priority를 이전에 생성한 thread보다 크게 설정한다. 따라서 늦게 생성된 thread의 우선순위가 높기 때문에 먼저 실행되게 된다.



텍스트이(가) 표시된 사진

자동 생성된 설명각 thread는 ITER\_CNT만큼 실행된다. 현재 코드에선 이 값이 16이므로 결과를 보게 되면 15가 먼저 16번 실행되고 14가 실행되는 것을 볼 수 있다.

Priority-fifo의 결과는 모든 thread의 priority가 같으므로 0~15가 순서대로 실행되는 것을 확인할 수 있다.

make check 수행 결과를 캡처하여 첨부