**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20161614 원성현

개발 기간 : 10.27 ~ 11.5.

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.  
  이번 프로젝트에서는 alarm clock의 기본을 구현하고, 이 기능을 이용해 실제로 priority scheduling을 구현하고 적용해본다. 추가로 Advanced Scheduling역시 구현, 적용해본다.

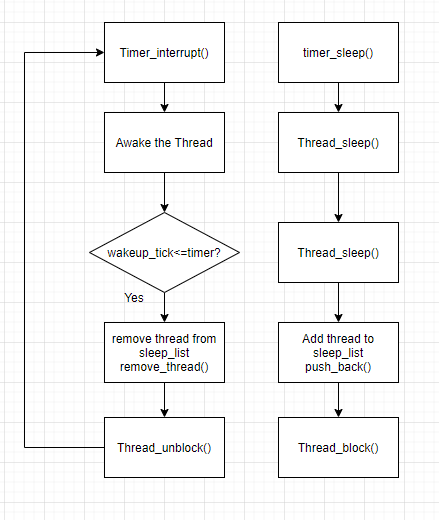
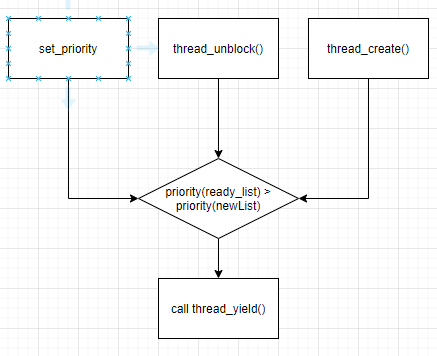
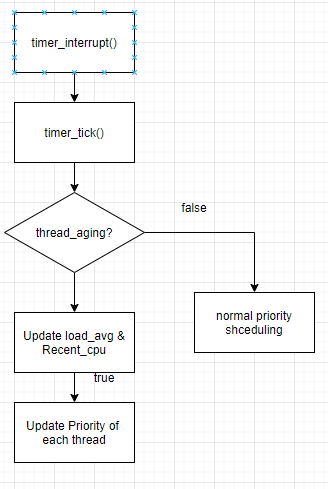
1. **개발 범위 및 내용**
   1. **개발 범위**

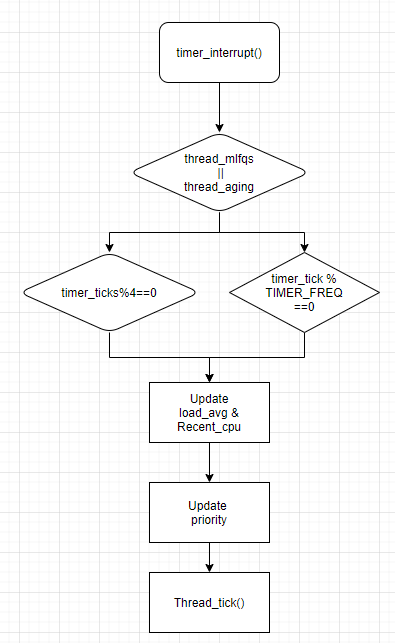
* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock  
     Alarm Clock이란, 핀토스에서 실행중 스레드를 잠시 sleep시켰다가 일정 시간이 지나면 다시 깨우도록 하는 기능이다. 현재 alarm-colck은 busy-waiting으로 구현되어 있는데, 이것을 수정해서 잠든 스레드를 ready가 아닌 blocked state로 두고 깨어날 시간에 ready state로 변경한다.
  2. Priority Scheduling  
     기존의 pintOS에서는 round-robin scheduling을 이용하여 프로그램들을 실행하였는데, priority scheduling을 사용해서 priority에 따라 스레드를 실행하게 만드는 스케쥴링을 구현한다.
  3. Advanced Scheduler (추가구현을 한 경우)  
     Advanced Schedular에서 우리는 BSD스케쥴러를 사용하게 된다. BSD는 multi-level feedback queue schedular로 ready queue가 여러개 존재하고, 각 queue는 서로 다른 priority가 존재한다. 스레드가 실행이 끝나면 가장 높은 priority를 가진 queue로부터 job을 가져와 실행한다. 강의에서 다루었던 개념이다.
  4. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.  
   blocked state에서는 스레드가 일어날(스레드를 깨울) 시간이 되었는지 확인할 방법이 없으므로, 각 스레드마다 일어나야 하는 시간에 대한 정보를 저장해아 하므로 wakeup\_time이라는 int형 변수를 thread 구조체에 추가한다.  
   또한, block된 상태가 아니라 sleep된 상태의 스레드들의 정보를 저장하기 위해서 sleep\_list를 추가하고 스레드를 처음 만들때(초기화)때 초기화 시킨다.  
   스레드들은 timer\_sleep()의 호출을 통해 blocked상태가 되고, 매 타이머를 interrupt 할 때 sleep\_list 구조체를 계속 iteration하면서 Sleep\_list에서 제거한 뒤에, 스레드를 unblock하면 된다.
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.  
   priority scheduling은 higher priority를 가진 Thread가 들어오면, ready\_list에 push할 때 higher priority를 먼저 push한다. 다른 말로는, ready\_list에 insert를 할 때 high-priority-ordered 하게 작업을 스케쥴링한다. 이 로직을 구현화한 코드가 list.c에 제공된다.
3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)  
   우리가 사용중인 mlfqs에서는 priority를 계산하는데 여러가지 변수가 들어간다.  
   priority=pri\_max – (recent\_cpu/4) – (nice\*2) 의 식으로 나타내어지므로, 총 세가지의 변수에 대해 알아야한다. Pri\_max는 pintOS에서 세팅된 정적값이고, nice는 이 스레드가 다른 스레드에 대해 어떠한 태도를 취하는가 에 대한 값이다. 값이 클수록 다른 스레드의 Priority를 상대적으로 낮추어 cpu time을 탈취한다. 마지막으로 recent\_cpu 는 말 그대로 최근에 얼마나 cpu에 의해 사용되었는지에 대한 값이다. 이 값을 계산하기 위해서는 load\_avg와 recent\_cpu가 필요하다. 주의 할 점은 PINTOS는 소수점 연산을 지원하지 않으므로 manual에 나온 방법대로 계산해야 한다.
4. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성  
  10.27. ~ 10.30. Alarm Clock 구현.  
  10.30 ~ 10.31. Priority Scheduling 구현.  
  11.1 ~ 11.5. Priority Aging , BSD Scheduling 구현.
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드  
    1) Alarm Clock  
      
    Alarm Clock을 구현하기 위해서는 timer.c thread.c thread.h를 수정해야 한다.  
    Thread.h에 스레드가 일어나는 시각을 알려주기 위해 int형 변수를 추가한다.  
    timer.c 에서는 현재 스레드의 wakeUpTime을 set해줘야 한다.  
    그리고 2번에서 얘기한 것처럼 잠든 스레드들을 관리하기 위한 리스트를 만들어준다.  
    timer\_sleep()에는 스레드를 sleep시키기 위해 sleepList에 넣어주는 과정을 구현한다.  
    timer\_interrupt에서는 2번에서 설명한 잠든 리스트가 깨어나는 과정을 구현한다.  
      
    2) Priority Scheduling  
      
    Priority Scheduling을 구현하기 위해 timer.c thread.c cal.c를 수정한다.  
    Priority scheduling의 핵심은 priority에 다라 작업의 순서를 바꾸는 것이다.  
    따라서 우리는 두 스레드의 우선순위를 비교하는 함수를 만들어주고,  
    2번에서 이야기한 것 처럼 pintOS에서 제공하는 함수를 이용해서 thread가 yield를 할 때 priority의 우선순위를 비교하고 정렬해서 ready\_list에 넣어주도록 한다.  
    thread를 unblock할 때에도 마찬가지이다. 그리고 스레드를 만들 때 새로 생성한 스레드가 current thread의 우선순위보다 높다면 thread를 yield할 수 있도록 한다.  
      
    이제 semaphore문제를 해결해야 한다. Sema\_up을 할 때 waiters에서 semaphore의 변동을 기다리는 스레드 중에 가장 높은 priority의 스레드를 찾아서 sema\_up을 해주는 과정이 필요하다. 왜냐면 스레드에 sema와 priority가 동시에 적용되는 케이스가 존재하기 때문이다.  
      
    그리고 ready\_list를 정렬할 때 ready\_list 맨 끝 thread의 priority를 증가시켜줘야한다.  
      
    3) Priority Aging & BSD Scheduling   
    BSD Scheduling을 구현하기 위해 timer.c thread.c init.c를 수정한다.  
    이것을 위해 2번에서 얘기한 manual을 읽고 소수 연산을 위한 함수들을 만들어줘야 하는데, 함수의 개수가 제법 되므로 따로 calc.c파일을 만들어주었다.  
    그리고 load\_avg의 값을 static으로 thread.c에 선언하고 계산에 이용한다. 이때 bit shift 연산을 위해 FRACTION이라는 값을 정의해서 사용한다.  
    Nice의 값 역시 사용해야 하기 때문에 thread.h에 nice를 추가한다. Recent\_cpu 역시 추가해준다. Thead.c에서 이 값들을 초기화해주고, 이 값들을 받아오는 get\_nice와 get\_recentCpu, get\_loadavg 등의 함수들을 구현한다.   
    이러한 준비물을 통해 실제로 스레드들을 iterate하면서 recentCpu와 loadAvg값을 계산하는 함수를 만들어준다. 그 뒤에 이 두 값을 이용해서 최종적으로 우리가 알고 싶은 각 스레드들의 priority를 계산해주는 함수를 만들어준다.  
    이렇게 priority가 전부 계산해주면 timer\_interrupt함수에서 timer가 tick할 때마다 값들을 갱신하게 된다.  
    Priority-aging에서 두 가지 조건을 추가시키면 mlfqs도 통과한다. 앞서 선언한 bool형 변수인 thread\_mlfqs 가 true 일 경우 thread\_set\_priority를 수행하지 않는다. 추가로 thread\_set\_nice를 수정한다.
  + 수정하거나 추가해야 하는 자료구조  
    1) Alarm Clock  
    timer.c - 현재 sleep상태인 스레드들의 정보를 담기 위한 struct list 형 변수인 sleep\_list를 추가한다.   
    thread.h - sleep인 스레드들을 꺠우기 위한 시간의 정보를 담기 위해, int형 변수인 int Wakeup\_time을 선언한다.  
      
    2) Priority Scheduling  
    thread.h에서 bool형 thread\_aging을 선언해야 한다.  
    이렇게 선언한 변수는 init.c에서 command를 parse할 때 사용하게 된다.  
      
    3) BSD Scheduling  
    priority의 계산을 위해 많은 변수를 추가로 선언한다.   
    PPT에 나온대로, int nice, int recent\_cpu, static int load\_avg를 선언해서 priority의 계산에 쓰이도록 한다. bit shift 연산을 위해 FRACTION이라는 값을 정의해서 사용한다.  
    테스트 때 쓸 수 있게 bool thread\_mlfqs변수도 선언한다.
  + 수정하거나 추가해야 하는 함수  
    1) Alarm Clock  
    timer\_sleep() : 현재 thread의 wakeup time을 설정하고, 이 스레드를 sleep\_list에 넣어서 sleep한 것이다라고 알려준다. 마지막으로, 스레드를 block상태로 만들게 한다.  
    timer\_interrupt(): sleep상태의 스레드를 깨워주는 기능을 추가한다. 이것은 sleep\_list를 순회하면서 list에서 remove하고 unblock하는 방식으로 구현한다.  
      
    2) Priority Scheduling  
    bool priority\_cmp : 두 스레드의 priority값을 확인하고 대소를 판별하는 함수이다.  
    tid\_t thread\_create() : 스레드를 만들 때 새로 생성한 스레드가 current thread의 우선순위보다 높다면 thread를 yield할 수 있도록 한다.  
    void sema\_up() : waiters에서 semaphore의 변동을 기다리는 스레드 중에 가장 높은 priority의 스레드를 찾아서 sema\_up을 해준다.  
      
    3) Priority Aging & BSD Scheduling  
    thread폴더에 새로운 c파일인 calc.c를 만들어서 floating point 계산을 가능하게 한다.  
    총 8 종류의 계산식을 만들어서 bitwise연산을 통해 소수 계산을 한다.  
    앞서 nice, recent\_cpu 등의 변수를 선언했기 때문에 init\_thread()와 thread\_init() 함수를 수정해서 변수들을 초기화 한다. Thread\_set\_nice() 함수를 구현해서 현재 스레드의 Nice 값을 가져오는 함수를 만든다.   
    그 외에도 thread\_get\_load\_avg(), thread\_get\_recent\_cpu()등의 getter함수를 만들어서 계산에 이용힌다. 앞서 얘기한 load\_avg와 recent\_cpu를 갱신하는 함수인   
    update\_load\_avg\_and\_recent\_cpu()를 만든다. 최종 계산식에 따라 priority 값을 갱신하는 update\_priority() 함수를 만든다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)  
    
    
  <Alarm Clock>  
  
* <Priority Scheduling>  
    
    
    
  <Advanced Schedular>



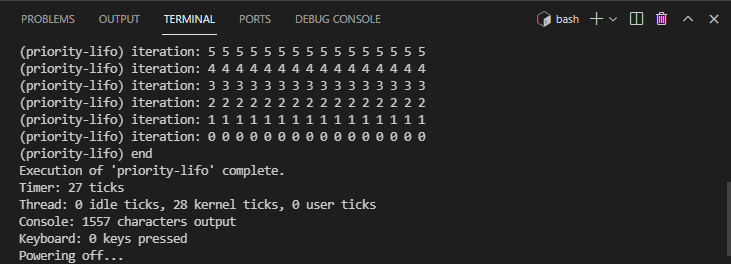
* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명

<ALARM CLOCK>   
Timer\_sleep() : 이 함수의 핵심 기능은 스레드를 sleep시키는 것이다.  
따라서 1번째로 thread\_current()->wakeup\_time에 경과된 시간인 start+ticks를 대입해주고, list\_push\_back(&sleep\_list, …)를 호출해서 스레드를 sleep\_list안에 넣어줘서 sleep 시킨다. 이 일이 끝난 뒤 스레드를 block해서 sleep과정을 마무리한다.  
  
Timer\_interrupt(): timer\_interrupt()에는 스레드를 깨우는(wake\_up) 새 기능을 추가하였다.  
잠든 스레드를 깨우는 과정은 다음과 같다. Struct list\_elem\* 형인 Elem을 선언하고,  
sleep\_list를 순회하면서 현재 스레드인 Thr->wakeup\_time이 ticks 이하인 스레드를 찾는다. 찾았다면 list에서 제거하고 스레드를 unblock해서 ready 상태로 만들어준다.  
만약 aging가 mlfqs가 적용되는 스케쥴이라면, 앞서 장황하게 설명한 두 가지 핵심 기능인 load\_avg/recent\_cpu , priority를 업데이트 하는 과정이 필요하다.  
따라서 update\_load\_avg\_and\_recent\_cpu를 호출하고, update\_priority()를 호출해서 시간이 지남에 따라 계속 priority 계산에 필요한 핵심 값이 변동하게 한다.  
  
  
  
<PRIORITY\_SCHEDULING>  
  
priority\_cmp() : 두 스레드의 priority를 비교하는 함수이다. 그 방법은 매우 간단한데, 인자로 두 list elem\* 를 받아서 list\_entry함수를 호출, list elem\*을 가진 스레드를 얻어온다.  
그 뒤에 두 스레드의 priority 필드에 접근해서 비교하면 된다.  
thread\_unblock(): Unblock()의 핵심은 priority에 따라 unblock하는 대상이 달라진다는 점이다. 따라서 기존에는 평등하게 처리한 부분을 priority에 차별을 두어 처리해야 한다.  
우리는 list.c에서 제공하는 list\_insert\_ordered()를 이용하고, 함수 호출시 인자로 앞에서 만든 priority\_cmp를 넣어서 priority에 따른 unblocking을 가능하게 한다.  
thread\_create(): 새로 생상하는 스레드의 priority가 thread\_current의 priority보다 크면 다시 스케쥴링을 해야한다. 따라서 priority를 고려해서 스케쥴링을 하는 thread\_yield함수를 호출한다.  
sema\_up(): sema->waiters가 비어있지 않다면, 우리는 waiters를 계속 순회하면서 그 waiter을 가지고 있는 thread를 list\_entry함수를 이용해서 불러온다.  
만약 현재 스레드의 priority가 max보다 크다면 max값을 thread와 element 모두 갱신시켜준다. 이렇게 max값을 유지하고, 이 스레드의 정보를 list에서 제거, unblock해준다.

<PRIORITY\_AGING & BSD\_SCHEDULING>  
  
thread\_init(): 새로 추가한 자료구조를 초기화한다. Initial\_thread->Nice,Recent\_cpu에 0을 대입한다.   
init\_thread(): thread\_init()과 마찬가지로 t->Recent\_Cpu에는 running\_thread()의  
Recent\_cpu를, Nice에는 running\_thread()의 nice를 대입한다.  
  
max\_priority(): max\_priority의 구조는 간단하다. Ready\_list의 가장 앞의(front)의 정보를 가져오면 된다.  
  
thread\_set\_nice(): 인자로 new\_nice를 받아 스레드의 nice값을 변경하고 priority를 갱신한다. 구현은 간단하다. 현재 스레드를 thread\_current()로 가져오고, Nice를 갱신, priority는 calc.c에서 지원하는 함수들을 이용해 floating point 연산을 한다.  
그리고 border check를 해서 t->priority가 bound 이상이면 PRI\_MAX, 이하면 PRI\_MIN을 한다. 만약 max\_priority보다 작다면 우선순위가 밀리므로 thread\_yield로 양보한다.  
  
thread\_get\_load\_avg(): calc.c에서 지원하는 계산식을 이용해 load\_avg값을 구한다.  
  
thread\_get\_recent\_cpu(): calc.c에서 지원하는 계산식을 이용해 Recent\_cpu값을 구한다.  
  
update\_load\_avg\_and\_recent\_cpu(): pintOS manual에 나오는 식 대로 load\_avg와 recent Cpu 값을 계산하는데, 이때 calc.c의 함수들을 이용한다. Recent\_cpu는 각 스레드마다 달라지기 때문에, 모든 리스트(all\_list)를 순회하면서 Thr->Recent\_cpu에 calc.c의 함수들을 이용해 값을 계산한다.  
  
update\_priority(): all\_list를 list\_entry()를 이용해 순회하면서 priority를 계산한다. 계산한 뒤에 boundary를 처리해주면 된다.   
  
timer\_interrupt(): thread\_aging과 thread\_mlfqs에 대한 or flag를 세워서 update\_priority()와 update\_load\_avg\_and\_recent\_cpu를 실행한다.   
pintOS manual에 따라 4 tick마다 update\_priority를, TIMER\_FREQ마다 update\_load\_avg\_ and\_recent\_cpu를 실행하면 된다.

* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석  
    
  코드, 테스트 결과 분석 :  
  i가 작았다 커지면서 스레드의 데이터를 입력. 입력하는 위치인 \* d는 data+i로, data+i가 위치하는 곳에 데이터를 갱신한다. 즉, 여기서 simple\_thread\_data data[]는 스레드들의 데이터들을 저장하고 있게 됨. 순서대로 id가 0~THREAD\_CNT(여기서는 16)인 데이터가 쓰이고 이 데이터를 가진 스레드가 만들어진다. 여기서 스레드가 만들어질때 simpleThreadFunc가 실행되는데, 이것은 op[][](op는 2차원 array지만 1차원으로 ‘늘여’ 구현되었다) 에서 스레드가 위치하는 n-th row에서 ITER\_CNT만큼 데이터를 직선적으로 쓴다. 그렇기 때문에 iteration : n n n n …. 의 꼴이 나중에 출력되게 된다.  
  주의할 점은 처음에 thread를 만들 때 iteration마다 더 높은 priority를 가지고 스레드를 만들게 된다. 또한, 스레드를 만들 때 simple\_thread\_func를 실행하기 때문에 thread\_yield()를 실행하고 따라서 가장 높은 priority가 CPU를 먼저 점유하게 된다.   
  다른 말로는, ready\_list에서 Last-In된, 높은 id값을 가진 스레드들이 Front에 정렬되어 있다. 스레드의 생성과 함수실행(+yield)가 끝나고 나면 main thread는 thread\_set\_priority를 (PRI\_DEFAULT)값으로 설정하고, 이제 ready list에 있는 thread들에게 실행권한이 양보된다. 당연하게도 ready list에서 가장 큰 priority는 id가 15인 thread이므로 이것이 선택되어 실행된다. 따라서 output은 2차원 array와 비슷한 op(구현은 1차원으로 되었다) 을 순회하면서 simpleThreadFunc에 의해 op에 넣어진 값들을 계속 찍어낸다.  
  따라서 15 15 15 … 가 ITER\_CNT만큼 찍히게 되고, ‘1차원적으로 구현된 op’를 2차원처럼 나타내기 위해서 if(++CNT % THREAD\_CNT == 0) printf(“\n”) 이라는 조건문을 집어 넣었다.  
  결과적으로 id15: (15 15 …) , id14: (14, 14, …) , … id: (0, 0, …) 이 쭉 직렬로 op안에 저장되어 있다가, id가 바뀔 때 마다 줄맞춤을 통해 출력되게 된다.   
  따라서 THREAD\_CNT-1부터 0까지 ITER\_CNT만큼 id가 줄맞춤을 구분해서 출력된다.
* 테스트 결과
* 텍스트이(가) 표시된 사진

  자동 생성된 설명  
  텍스트이(가) 표시된 사진

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

  자동 생성된 설명