**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

조 / 조원 : 20191138 이현규

개발 기간 : 11/07~11/25

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.  
  현재 PintOS는 Round Robin 스케줄러를 기반으로 작동하고 있다. 이런 단순한 RR 스케줄러를 벗어나, thread의 우선순위를 고려할 수 있는 Priority Scheduler를 도입한다. 필요하다면 lock과 semaphore등의 동기화 기법을 사용하여 이를 효과적으로 구현한다. 이를 위해서 전체적인 scheduling 과정에서 우선순위를 고려할 수 있도록 구현해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock  
     프로세스가 계속 timer\_sleep에서 tick만큼을 시간이 경과되었는지 확인을 위해 oscillate 하는 것이 아니라, sleep 처리된 thread들이 wake 되도록 대기하도록 하는 리스트를 만든다. 해당 리스트는 다른 리스트와 마찬가지로 init되어야 한다. 따라서 while 루프에서 특정 프로세스 동작이 완료되기만을 기다려야 하는 busy waiting을 제거할 수 있다.
  2. Priority Scheduling  
     struce thread에 이미 int pritority가 존재하므로, 이를 사용하도록 코드를 수정하면 된다. next\_thread\_to\_run이나 thread 리스트에 thread를 삽입하는 과정에서 prioroity 순서로 이를 정렬하는 식으로 동작하게 하여야 한다. 프로세스의 우선순위에 따라 CPU 자원이 할당되고, 특히나 우선순위가 높은 프로세스가 새로 만들어질 때 이에 양보할 수 있도록 하여 우선순위에 비례하여 자원이 할당될 수 있도록 한다.
  3. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.  
   timer\_sleep 함수에서 매개변수로 받은 tick과 현재 tick을 더한, 자신이 다시 wake up 해야할 타이밍을 wake\_me\_up\_at로 저장하고, sleeping list로 보내진 blocked thread는 timer\_interrupt() 함수에서 관리된다. 이 함수는 wake\_me\_up\_time 보다 현재 tick이 큰지, 즉 sleep에서 주어진 tick만큼 대기하여 이 스레드가 깨어날 시간이 되었는지 확인한다. 이 대기열은 깨어나는 시간대로 오름차순으로 정리되어 있다. 이 함수는 wake up time에 도달한 스레드를 이 sleeping list에서 제거하고, ready 상태로 전환한 뒤에, 이 스레드를 ready\_list에 추가한다(unblock).
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.  
   새롭게 ready list에 들어온 thread가 running\_thread보다 우선순위가 높을 경우, 즉시 thread\_yield()를 호출해 CPU 자원을 반환한다. thread\_yield()는 schedule() 함수를 호출하고, 새로 들어온 고-우선순위 thread가 포함된 Ready List를 바탕으로 next\_thread\_to\_run() 함수가 작동하여 해당 thread로 전환된다.
3. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
* 11/07 ~ 11/14: 프로젝트 명세서 및 핀토스 매뉴얼 탐독
* 11/15 ~ 11/20: Alarm Clock 구현
* 11/21 ~ 11/23: Priority Scheduling 구현
* 11/24 ~ 11/25: 보고서 작성 및 디버깅
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  1. devices/timer.c
     + sleeping list 선언 - 단순한 while 구문이 아니라, sleep 중인 스레드를 관리할 수 있는 리스트인 sleeping list가 필요하다. 이 리스트는 항상 정렬된 형태를 유지할 수 있도록 다른 함수들에서 기존의 list\_push\_back/front가 아니라, list\_insert\_ordered를 사용한다.
     + sleep\_list\_wake\_up\_time\_compare 함수 작성  
       lib/kernel/list.c를 참고하면, list\_insert\_ordered는 세 번째 매개변수로 정렬 방법을 의미하는 함수를 받는다. 깨어나야 하는 시간을 비교해주는 함수를 작성하여, timer\_sleep 함수에서 list\_insert\_ordered를 사용할 때, 해당 함수를 이용할 수 있도록 한다.
     + timer\_sleep 함수 수정  
       기존의 while 루프 + thread\_yield()가 아니라, 자신이 새롭게 깨어나야 할 시간을 계산한 후, sleeping\_list에 스레드를 정렬된 순서로 삽입하고, block(sleep)되도록 수정한다.
     + timer\_interrupt 수정  
       sleeping\_list가 정렬되어 있으므로, sleeping list를 traverse 하면서 현재 시간보다 wake\_me\_up\_at time이 작거나 같은, 일어나야 하는 프로세스가 있으면 이들을 list\_remove하고, thread\_unblock 하여 ready list에 넣어주면 된다. 정렬되어 있으므로, 지금보다 나중에 일어날 것을 하나라도 발견하면 정지해도 된다.
  2. threads/init.c에 주어진 것 처럼 aging 관련 조건을 추가해야 한다.
  3. threads/synch.c
     + sema\_down 수정  
       semaphore의 값이 0일 경우, 현재 threa를 sema->wait 리스트에 삽입하고, 대기 상태로 전환한다. list\_insert\_ordered를 사용하여 높은 우선순위를 가진 thread가 먼저 실행되도록 할것이다. 그 후, thread\_block()을 호출하여 sleep 상태로 전환된다.
     + sema\_up 수정  
       대기중인 thread가 존재한다면, list\_sort를 호출해 우선순위로 정렬한 후, 가장 높은 우선순위 thread를 pop한 뒤, unblock 한다. 만약 이 thread의 우선순위가 running thread보다 높다면, 현재 thread가 thread\_yield 하여 이 thread가 수정될 수 있도록 한다.
  4. threads/thread.c
     + aging 조건 반영 - aging 관련 코드가 실행될 수 있도록 과제 명세서에 나온 것 같이 코드를 추가하였다.
     + thread\_create - 생성된 스레드가 지금 스레드보다 우선순위가 높다면 이를 양보받을 수 있도록 해야 한다. 이를 비교할 수 있도록 하기 위해 thread\_priority\_compare 함수를 작성해야 한다.
     + thread\_unblock - ready\_list에 추가할 때, 우선순위를 반영할 수 있도록 thread\_priority\_compare를 사용하는 list\_insert\_ordered로 수정해야 한다.
     + thread\_yield - ready\_list에 추가할 때, 우선순위를 반영할 수 있도록 thread\_priority\_compare를 사용하는 list\_insert\_ordered로 수정해야 한다.
     + thread\_set\_priority - 우선순위를 고려하기 시작했으므로, 기존의 우선순위보다 낮은 우선순위로 설정되었다면 우선 thread\_yield를 하여 새 우선순위보다 높은 thread가 있다면 이에 양보할 수 있도록 해야 한다.
  5. threads/thread.h
     + aging 조건 추가 - aging 관련 코드가 실행될 수 있도록 과제 명세서에 나온 것 같이 코드를 추가하였다.
     + struct thread 수정 - int wake\_me\_up\_at을 추가하여, sleep된 프로세스가 wake up 될 수 있도록 여야 한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)
* Blocked 상태의 스레드를 어떻게 깨울 수 있는가

스크린샷, 텍스트, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

* Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling의 동작과정.

스크린샷, 포스트잇 노트, 사각형, 다채로움이(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. devices/timer.c
     + sleeping list 선언  
       sleeping 상태의 thread를 관리하는 list를 코드 맨 처음에 선언하고, 이를 timer\_init 함수에서 list\_init 해주었다.  
       텍스트, 스크린샷, 폰트이(가) 표시된 사진

       자동 생성된 설명
     + sleep\_list\_wake\_up\_time\_compare 함수 작성  
       lib/kernel/list.c를 살펴보면, list\_insert\_ordered는 (elem을 삽입할 list, list에 삽입될 list\_elem elem, 정렬 방법인 list\_less\_func, 기타 data aux)를 매개변수로 받는다. if(less(elem, e, aux)) break;의 형태로 less 함수가 사용되므로, 비교되는 thread(두 번째 매개변수 lb)가 나(la)보다 먼저 깨어나야 하는 경우에는 false, 내가 먼저 혹은 동시에 깨어나야 하는 경우에는 true를 반환하는 함수를 작성하였다.  
       텍스트, 스크린샷, 폰트이(가) 표시된 사진

       자동 생성된 설명
     + timer\_sleep 함수 수정  
       핀토스 매뉴얼과 프로젝트 명세서에 while (timer\_elapsed(start) < ticks) thread\_yield();로 구현되어 있는 busy waiting을 제거하라고 되어있다. 해당 부분을 제거하고, 아래의 코드로 대체하였다. sleeping\_list에 대한 race condition을 막기 위해 인터럽트를 선언하고, 내가 새롭게 깨어나야 할 시간을 저장한 후, sleeping list에 선술한 비교 함수 기반으로 정렬된 순서로 나를 삽입한 뒤에, threads/thread.c에 있는 thread\_block()을 사용하여 나를 block/sleep 한다. 그 후, 다시 inter\_set\_level(old\_level)을 호출하여 인터럽트를 해제한다.  
       텍스트, 스크린샷, 폰트이(가) 표시된 사진

       자동 생성된 설명
     + timer\_interrupt 수정  
       사실상 tick만 증가했던 함수를 수정하여 깨어날 시간이 된 thread를 깨우는 기능을 구현하였다. while 반복문은 대기중인 스레드의 리스트인 sleeping list의 list\_begin에서 시작해, 리스트의 마지막을 의미하는 list\_end에 도달하기 전까지 반복된다. if (t->wake\_me\_up\_at <= timer\_ticks()) 조건에 따라 깨어날 시간이 된 thread를 sleeping\_list에서 제거하고, 그 thread를 thread\_unblock을 호출하여 ready list에 삽입한다. sleeping list는 인터럽트와 list\_insert\_ordered를 통해 항상 정렬된 상태를 유지하므로, else일 경우 break 하면 된다. 따라서, if문이 성립하면 e는 list\_remove의 반환값인 다음 element이기 때문에, t는 항상 그 다음 요소를 참조할 수 있게 된다.  
       
  2. threads/init.c - 과제 명세서에 있는 대로, aging 관련 조건을 추가하였다.
  3. threads/synch.c
     + 정렬을 위한 함수 - #include "threads/thread.h"를 통해 thread에 작성해둔 함수를 참고할 수 있으므로, thread\_priority\_compare 함수를 사용하여 thread의 우선순위를 비교할 수 있도록 하였다. 이는 선술한 wake\_me\_up\_at을 비교하는 함수와 비슷하게, list\_insert\_ordered에 필요하다.
     + sema\_down 수정  
       기존에는 그저 thread를 sema->waiters의 마지막에 삽입하는 list\_push\_back 함수가 사용되었지만, 해당 부분을 thread\_priority\_compare 함수를 포함한 list\_insert\_ordered를 사용하여 우선순위가 고려될 수 있도록 하였다.
     + sema\_up 수정  
       기존의 함수는 sema->waiters에 가장 먼저 삽입된 thread를 제거하고, 이를 unblock하는 방식으로 작동하였다. 하지만 이는 스레드 우선순위를 고려하지 않은 형태였다. 이를 개선하기 위해, sema\_down에서 우선순위를 고려하도록 하였다. 하지만 그 시점 사이에 우선순위가 변동할 수 있으므로, list.c를 참고하여 thread\_priority\_compare를 사용하는 list\_sort를 우선 호출한 후, 가장 먼저 있는, 즉 우선순위가 가장 높은 스레드를 list\_pop\_front를 사용하여 제거한 뒤, 이를 thread\_unblock 하도록 수정하였다. 이 스레드가 지금 동작하는 스레드보다 우선순위가 높을 수 있으므로, 이를 확인할 수 있도록 thread\_yield를 호출하도록 하였다.  
       텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

       자동 생성된 설명
  4. threads/thread.c
     + 비교 함수 추가  
       wake\_me\_up\_at과 비슷하게, priority를 비교하는 함수를 작성하고, 이 선언을 thread.h에 추가하였다.  
       텍스트, 스크린샷, 폰트이(가) 표시된 사진

       자동 생성된 설명
     + aging 조건 반영 - 과제 명세서에 있는 aging 관련 코드를 추가하였다.
     + yield 조건 추가 - thread\_unblock과 thread\_create는 새 thread를 ready list에 추가하는 동작을 한다. 이 새 thread가 현재 thread보다 우선순위가 높은 경우, 양보할 수 있도록 하는 코드를 두 함수 마지막 부분에 추가하였다.  
       텍스트, 폰트, 스크린샷, 친필이(가) 표시된 사진

       자동 생성된 설명
     + thread\_yield 수정  
       thread\_yield 함수가 현재 thread가 양보를 할 때, list\_push\_back을 통해 ready list의 맨 뒤에 삽입을 하는 형태로 구현되어 있다. 우선순위를 고려할 수 있도록, 해당 부분을 list\_insert\_ordered로 수정하였다.  
       텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

       자동 생성된 설명
  5. threads/thread.h
     + aging 조건 추가- 과제 명세서에 있는 aging 관련 코드를 추가하였다.
     + struct thread 수정  
       timer에 사용되는 내가 깨어날 시간을 저장하는 int wake\_me\_up\_at을 struct thread에 추가하였다.
  6. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석
  1. priority-lifo.c 코드 분석  
     이 테스트 코드는 THREAD\_CNT를 16, ITER\_CNT를 16으로 설정하여, 서로 다른 우선순위를 가진 16개의 스레드가 각 16번씩 순서를 지키면서 실행되는지 확인한다. thread\_create의 두 번째 매개변수인 priority가 PRI\_DEFAULT + 1 + i로 생성되므로, 가장 늦게 생성된 i=15인 스레드가 가장 높은 우선순위(기본값+1+15)를 가지게 된다. 가장 늦게 생성된 스레드가 가장 높은 우선순위를 받아 가장 먼저 실행되어야 하므로, LIFO 방식으로 실행되게 된다. if (++cnt % THREAD\_CNT == 0) printf ("\n");를 통해 진행 순서 및 횟수를 확인하기 쉽게 출력되며, 16개의 스레드가 각각 16번 반복되어 실행되어야 한다.
  2. 기대 출력  
     16개의 스레드가 우선순위 순서대로 실행되어야 하므로, 15 15 15 15 .... 15 (16번)\n, 14 14 14 14 .... 14 (16번)\n, ... 0 0 0 0 ... 0 (16번)\n 형태로 출력되고, 부팅 및 종료 과정이 정상적으로 이루어질 것이다.
  3. 실행 결과 및 분석  
     우선 핀토스 부팅, 실행, 종료 과정이 정상적으로 이루어졌다. 또한, 코드가 예상대로 LIFO 방식으로 작동했음을 볼 수 있다. ID가 15(우선순위가 기본값+1+15)인 스레드가 16번, 14가 16번, ... , 0(우선순위 기본값+1+0)번이 16번 작동하는 방식으로 작동하였다. 이는 우선순위 기반 프로세스 스케쥴링이 작동했음을 보여준다.  
     텍스트, 스크린샷이(가) 표시된 사진

     자동 생성된 설명
* make check 수행 결과를 캡처하여 첨부
* 텍스트, 스크린샷이(가) 표시된 사진

  자동 생성된 설명
* priority-aging을 제외하고 모두 통과하였다. 옵션을 parsing하는 과정과 userprog에 대한 부분에 문제가 있는 것으로 보인다. 다음 프로젝트에서 이를 해결하고자 한다.