**Pintos Project 3: Threads**

담당 교수 : 박성용

조 / 조원 : 이우진

개발 기간 : 2023/10/30 – 2023/11/12

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

본 프로젝트에서는 OS에서 thread와 관련된 기능을 구현한다. 본 프로젝트를 수행하기 이전의 pintos에서 매우 비효율적으로 구현되어 있는 thread의 sleep 기능을 효율적으로 변경하고, thread에 우선순위를 부여하여 우선순위에 따라 schedule될 수 있도록 구현한다. 추가적으로, 기존의 round-robin 방식의 scheduler이 아닌 MLFQ 방식의 scheduler을 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

기존의 pintos에서, timer\_sleep( ) 함수를 사용하게 되면 해당 함수를 호출한 시점의 thread가 schedule 되어 다시 실행될 때 sleep 상태에서 벗어나야 하는지 확인한다. 그런데 위와 같이 구현하게 되면 sleep 상태에 있는 thread가 context switch을 통해 schedule되어 실행되고, cpu을 사용하게 된다.. 각각의 thread을 실행하면서 검사하지 않고 context switch가 일어나는 중에 검사하게 되면 sleep 상태에 있는 thread을 실행하지 않아도 되므로 보다 효율적이다.

1. Priority Scheduling

Priority scheduling을 구현하여 thread에 우선순위를 지정하여 우선순위에 따라 실행할 수 있다. 이 때, priority로 인해 발생하는 starvation을 priority aging을 도입하여 해결할 수 있다.

1. Advanced Scheduler

기존에 pintos에서는 round-robin방식을 사용하여 scheduling을 하였는데, round-robin은 response time 측면에서는 좋지만 turnaround time은 좋지 않다는 단점이 존재한다. 따라서 thread scheduling에는 MLFQ가 널리 사용되는데, MLFQ를 사용하게 되면 짧은 cpu time을 가지는 thread의 우선순위를 높게 하여 실행하므로 turnaround time을 향상시킬 수 있다.

* 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)
4. 기존에 구현되어 있던 timer\_sleep( ) 의 경우, block 상태에 놓인 thread가 schedule될 때 마다 block 상태에서 벗어날지 결정하였다. 본 프로젝트에서는 block 상태에 있는 thread을 저장하기 위한 sleep\_list을 사용한다. Thread가 timer\_sleep( )을 호출하면 해당 thread의 PCB에 thread가 block 상태에서 벗어나는 시간을 저장하고, thread을 sleep\_list에 추가한다. 이후 timer interrupt가 발생할 때마다 sleep\_list에 존재하는 모든 thread을 확인하여 저장된 시간이 지난 thread의 경우 sleep\_list 에서 제거하고, ready\_list에 추가함으로서 block 상태의 thread을 깨울 수 있다.
5. Ready list에 running thread보다 높은 priority을 가진 thread가 들어올 경우, preemptive 한 방식을 사용하여 yield( ) 을 통해 새로운 thread을 실행하여야 한다.
6. Advanced Scheduler에서 priority을 계산하기 위해, 새로운 전역 변수인 load\_avg와 thread 별로 가지는 변수인 recent\_cpu와 nice을 추가한다. 각각의 변수는 다음과 같은 의미를 가진다.
   * Load-avg : OS가 시작되었을 때부터, idle thread을 제외하고 실행될 준비가 되어 있는 thread의 개수를 확인한다. 그 값의 평균을 저장하고 있는 값이 load\_avg이다. Load\_avg는 1초마다 다시 다음과 같이 계산된다.

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

* + Recent\_cpu : recent\_cpu 값은 최근에 부여된 CPU time을 나타내는 변수이다. Recent cpu의 값이 크다면 최근에 CPU을 오래 점유하고 있었다는 말 이므로, priority가 계산될 때 낮은 priority가 계산된다. tick마다 현재 실행되고 있는 thread의 recent의 값을 1만큼 증가시키고, 1초마다 모든 thread의 recent\_cpu을 아래의 식에 따라 다시 계산한다.

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

* + Nice : nice 값은 사용자가 결정하는 변수로, 사용자는 제한적이지만 기존의 priority 값을 설정하는 것과 같이 어느정도 thread간에 우선순위를 설정할 수 있다. 초기값은 부모 thread가 존재하는 경우 부모 thread의 nice 값을 따르고, 존재하지 않는 경우 0을 가진다. Nice값은 -20에서 20 사이의 값을 가지며 값이 클 수록 thread의 우선순위가 낮다.

위 변수들을 이용하여, 모든 thread의 priority을 4 tick마다 다음과 같은 식에 따라 계산한다. Priority는 0에서 63 사이의 값을 가져야 하므로, 해당 범위를 벗어났을 때는 최댓값 또는 최솟값으로 조정해 주어야 한다.

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

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

**10/30 – 11/5 :** Alarm\_clock에 대한 구현과, priority scheduling을 구현한다.

**11/12 – 11/12 :** Advanced scheduler 을 구현한다.

**11/12 – 11/14 :** 위 개발 과정에서 발생하였으나 완전히 수정되지 않은 버그를 수정한다.

* 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

Alarm Clock을 구현하기 위해, 먼저 threads/thread.h 파일에서 *struct thread*에 thread가 wake up을 하게 될 시간을 저장하는 변수를 추가한다. Thread가 sleep 상태에 들어가게 하는 *thread\_sleep( )*, sleep 상태에 있는 thread을 모두 확인하여 wake up을 할수 있도록 *check\_wakeup( )* 함수를 threads/thread.h 및 threads/thread.c에 추가한다. 이후 기존에 device/timer.c 파일의 *timer\_sleep( )* 함수에 구현되어 있던 코드를 새롭게 정의한 *thread\_sleep( )* 함수로 대체하고, Timer\_interrupt 가 발생할 때마다 *check\_wakeup( )* 함수를 실행하도록 *timer\_interrupt( )* 함수에 *check\_wakeup( )* 함수를 호출한다.

* 1. Priority Scheduling

먼저, threads/thread.c 파일에서 user program 측에서 priority을 변경하고, 현재 실행중인 thread의 priority 값을 얻는 함수인 *thread\_get\_priority( ), thread\_set\_priority( )* 함수를 구현한다. Priority에 따라 scheduling을 하기 위해서는 *ready\_queue*에서 priority가 큰 순서대로 schedule을 할 수 있어야 한다. 따라서, *ready\_queue*가 정렬된 상태로 유지될 수 있도록 새로운 thread을 삽입하는 함수인 *insert\_ready( )*을 추가하고, thread\_unblock( ) 등의 *ready\_queue*에 새로운 thread을 삽입하는 루틴을 가진 모든 함수의 루틴을 *insert\_ready( )*로 대체한다. Semaphore up을 하는 과정에서도 priority에 따라 semaphore up을 해줄 thread을 선택할 수 있도록 threads/synch.c에도 비슷한 함수인 *find\_sema\_up( )* 함수를 추가하고, 기존의 루틴을 대체할 수 있도록 한다.  
 Priority aging을 구현하기 위해 threads/thread.h 및 threas/thread.c 파일에 thread\_aging( )을 추가하고, tick마다 실행되는 thread\_tick( ) 함수에 해당 함수를 추가해준다.

* 1. Advanced Scheduler

Advanced Scheduler을 구현하기 위해서, priority을 계산하기 위한 변수인 nice, recent\_cpu을 struct thread에 추가하고, threads/thread.c 파일에 load\_avg을 추가해준다. 그리고 해당 변수들을 설정하고 계산하기 위한 함수들인 *calc\_load\_avg( ), calc\_recent\_cpu( ), get\_mlfq\_priority( ), thread\_get\_recent\_cpu( ), thread\_get\_load\_avg( ), thread\_get\_nice( ), thread\_set\_nice( )* 함수들을 모두 선언하고 구현한다. 변수들을 update하는 데는 pintos에서 제공하는 thread\_foreach( ) 함수를 이용한다. 이를 위해, thread\_update\_recent\_cpu( ), thread\_update\_priority( )함수를 구현한다. 마지막으로, priority가 계속해서 다시 계산되므로 기존의 ready\_list로는 정렬되었음을 보장할 수 없다. 따라서, pintos에서 제공하는 list\_sort() 함수를 이용하여 thread\_update\_priority( ) 호출 이후에 호출하도록 구현한다. Update을 위한 함수들은 thread\_tick( ) 함수에서 조건에 따라 실행되도록 추가한다.

1. **연구 결과**
   1. **Flow Chart**

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

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

스크린샷, 텍스트, 도표, 폰트이(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

Device/timer.c 등의 파일을 살펴보고, pintos에서는 시간을 *int64\_t*로 저장한 것을 알 수 있었다. Threads/thread.h 파일의 *struct thread* 구조체에 *int64\_t wakeup\_time* 변수를 추가하였다. 그리고 sleep 상태인 thread 들을 관리하기 위해, 해당 thread을 담는 list인 sleep\_list을 threads/thread.c 파일에 전역변수로 추가하고, thread\_init( ) 에 sleep\_list을 초기화하였다.

기존에 thread가 sleep을 하게 되는 과정을 대체하기 위해, thread\_sleep( ) 함수를 threads/thread.h 및 threads/thread.c 파일에 추가하였다. 해당 함수는 thread가 깨어날 시간을 가지는 변수인 wakeup\_time을 인자로 받는 함수이다. 함수가 실행되면 sleep 루틴을 호출한 현재 thread의 정보를 얻어 해당 thread struct에 wakeup\_time 변수을 저장하고, sleep\_list에 추가한다. 이후 thread\_block( ) 함수를 호출하여 thread을 block 상태로 전환하고 다른 thread에 yield을 할 수 있도록 한다. 해당함수는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

구현한 함수는 기존에 devices/timer.c 파일의 timer\_sleep( ) 함수에서 기존의 루틴을 제거하고 대신 호출한다. 이 때, 인자로 전달할 wakeup\_time의 값은 기존에 Pintos에서 현재 시간을 제공하는 함수인 timer\_ticks( )을 이용하여 계산한다. 변경한 함수는 다음과 같다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Thread을 깨우는 과정은 threads/thread.c 파일의 check\_wakeup( ) 함수에서 구현하였다. Check\_wakeup( ) 함수는 현재 시간을 가지는 변수 *now*을 인자로 입력받고, 반복문을 이용하여 sleep\_list을 순회하며 wakeup\_time을 확인하여 wakeup\_time보다 now가 큰 값을 가지는 경우 sleep\_list에서 해당 thread을 제거하고, *thread\_unblock( )* 함수를 호출하여 해당 thread가 block 상태에서 벗어나 ready\_list에 추가될 수 있도록 구현하였다. 구현한 함수는 다음과 같다.

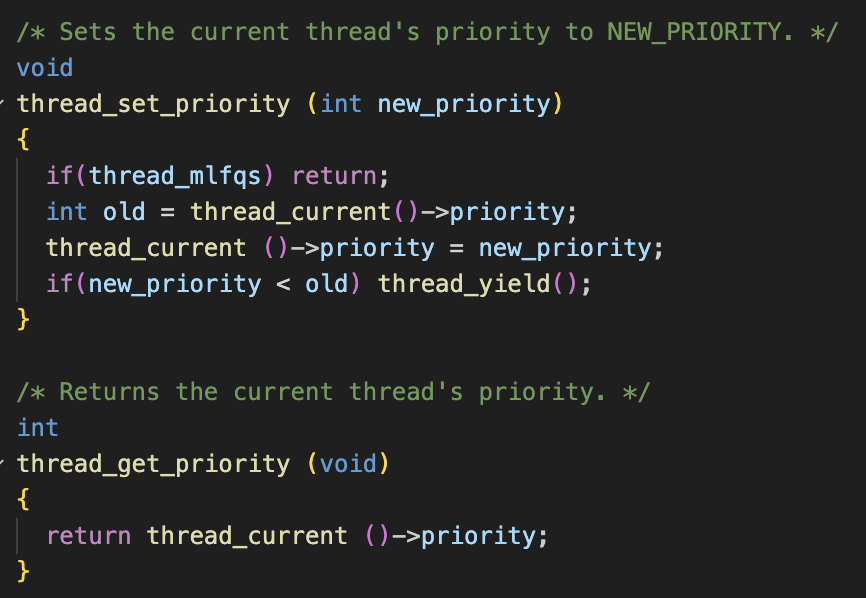
텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

구현한 함수는 timer interrup가 발생할 수 있도록 devices/timer.c 파일의 *timer\_interrupt( )* 함수에서 호출하였다.

* 1. Priority Scheduling

Priority는 이미 struct thread에 선언되어 있으므로 해당 변수를 사용하여 priority scheduling을 구현한다. Threads/thread.c 파일에 선언된 thread\_get\_priority() 함수에서는 thread\_current() 함수를 이용해 현재 실행중인 thread의 priority 값을 반환하도록 구현하였다. thread\_set\_priority() 함수에서는 현재 실행중인 thread의 priority 값을 바꾸어 주고, 바꾸기 이전의 priority가 바꾼 이후의 priority보다 더 높을 경우, 즉 priority 값이 낮아졌을 경우에는 현재 thread의 priority가 가장 높은지 알 수 없으므로 thread\_yield()을 호출하여 가장 높은 우선순위를 가진 thread을 찾아 실행하도록 구현하였다.



현재 실행 중인 thread의 priority보다 새롭게 생성된 thread의 priority가 높을 때에도 새로운 thread을 실행해야 하므로, thread\_create() 함수의 마지막 부분에 해당 조건을 만족할 때 thread\_yield()을 호출하도록 추가하였다.

Priority scheduling을 구현하기 위해서는 *ready\_list* 내의 thread 중 가장 높은 priority을 가진 thread을 선택하여 실행하는 것이 필요한데, *ready\_list*에 thread을 추가할 때 정렬된 상태를 유지하도록 추가하는 방법과, *ready\_list*에서 새로운 thread을 선택할 때 가장 높은 priority을 가지는 thread을 선택하는 방법이 있다. 이 중 첫 번째 방법을 선택하여 구현하였다. Threads/thread.c 파일에 구현한 insert\_ready() 함수가 그 역할을 하는 함수이다. Ready\_list을 순회하면서 *new\_thread*가 삽입될 가장 적절한 위치를 찾고, 해당 위치에 thread을 추가하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

위 함수를 thread을 *ready\_list*에 추가하는 모든 함수에서 기존의 list\_push\_back() 함수를 대체하여 호출하여야 했다.

Semaphore로 인해 block 상태에 있는 thread의 경우에도 priority을 적용하여, sema\_up을 하였을 때 대기하고 있는 임의의 thread을 선택하지 않고, priority가 높은 thread가 선택될 수 있도록 구현하였다. Priority가 가장 높은 thread을 선택하는 find\_sema\_up() 함수를 threads/synch.c 파일에 구현하였다. semaphore의 *waiters*을 인자로 입력 받고, 위의 *insert\_ready()*와 유사하게 waiters을 순회하면서 가장 높은 priority을 가지는 thread을 *waiters*에서 제거하고, 반환하였다.

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

자동 생성된 설명

해당 함수를 같은 파일의 sema\_up() 함수에서 기존의 thread 선택 루틴을 대체하여 호출하였는데, 이 때에도 sema\_up을 한 thread가 기존에 실행되고 있던 thread의 priority보다 높은 경우 해당 thread을 실행하도록 thread\_yield을 실행하였다. 이 때, 만약 interrupt 중 이었다면 thread\_yield()을 호출 하였을 때 에러를 출력하는 것을 확인하였고 때문에 interrupt context였다면 interrupt가 종료된 이후에 yield을 하도록 해주는 intr\_yield\_on\_return ()을 호출하도록 추가하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Thread\_aging은 ready\_list에 존재하는 모든 thread을 순회하며 priority을 1씩 증가시켜주는 thread\_aging() 함수를 추가하고, 해당 함수를 thread\_tick()에서 실행해주었다. 이 때, priority가 PRI\_MAX을 넘어갔을 때는 priority을 PRI\_MAX로 조정해주는 코드를 추가하였다.

* 1. Advanced Scheduler

Advanced Scheduler을 구현하기 위해서는 소수점 연산을 지원하지 않는 pintos 상에서 소수점 연산을 구현하여야 했다. 소숫점 연산은 threads/thread.h에 매크로 형식으로 구현하였다.

스크린샷, 텍스트이(가) 표시된 사진

자동 생성된 설명

Priority 계산에 필요한 변수인 nice, recent\_cpu을 struct thread에 추가하였고, threads/thread.c에 전역변수 load\_avg을 추가하였다. Nice와 recent\_cpu는 thread\_init()에서 0으로 초기화하고, thread\_create() 에서 부모 thread와 같은 값을 가지도록 변경하였다.

기존에 핀토스에서 선언은 되어있었으나 구현은 되어 있지 않은 thread\_set\_nice(), thread\_get\_nice(), thread\_get\_load\_avg(), thread\_get\_recent\_cpu()을 구현하였다. 이 때, thread\_set\_nice() 에서는 nice가 기존보다 큰 값을 가지는 경우, 우선순위를 알 수 없으므로 thread\_yield()을 호출하도록 하였다.

위에서 구현한 real 연산을 이용하여 load\_avg, recnet\_cpu, priority을 각각 계산해주는 함수인 calc\_load\_avg(), calc\_recent\_cpu(), get\_mlfq\_priority을 구현하였다. 이 때, load\_avg을 계산하기 위해서는 ready\_thread의 개수를 알아내야 하는데, 이를 위한 함수인 get\_ready\_threads()도 추가로 구현하였다. Get\_ready\_threads() 함수는 pintos에서 제공하는 list\_size()을 이용해 ready\_list에 존재하는 thread의 개수를 구하고, 현재 실행중인 thread가 idle thread가 아닌 경우에는 1을 추가하여 반환하였다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Load\_avg는 1초마다 update을 해주어야 하고, recent\_cpu의 값은 실행되고 있는 thread에 한해 1tick마다 1씩 증가시키고 1초마다 모든 thread에 대해 update 해주어야 한다. 그리고, 모든 thread에 대해 priority을 4tick마다 update 해주어야 하므로 모든 thread에 대해 update 하는 로직은 pintos에서 제공하는 thread\_foreach() 함수를 이용하여 구현하였다. 이를 위해서 thread\_foreach()에서 모든 함수를 순회하면서 각각에 대해 실행하는 함수인 thread\_action\_func인 thread\_update\_recent\_cpu() 와 thread\_update\_priority을 구현해야 했다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Thread\_tick() 함수에서 필요한 모든 update을 진행하였다. 이 때, priority가 다시 계산될 때 마다 ready\_list에서의 순서가 알 수 없게 되므로, pintos에서 제공하는 list\_sort()을 사용하여 매번 정렬해주었다. List\_sort()에서 사용하는 비교함수인 reorder\_ready\_list() 함수를 추가로 구현하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

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

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명  
**테스트 결과**

*Priority\_lifo.c* 코드는 다음과 같이 실행된다.

* 1. Main thread의 priority을 PRI\_DEFAULT + THREAD\_CNT + 1 값으로 설정하고, lock을 생성하여 초기화한다.
  2. THREAD\_CNT 만큼의 thread을 생성한다. 이 때, i번째 thread는 1)에서 생성한 lock을 가지며, priority는 PRI\_DEFAULT. + 1 + i 값을 가진다. 이 시점에서 모든 thread의 priority가 main보다 낮은 값을 가지므로, main thread의 priority을 변경하기 전에는 각 thread는 실행되지 않는다.
  3. Main thread의 priority을 PRI\_DEFAULT로 변경한다. 이로써 2)에서 생성한 thread들이 cpu을 점유하여 실행될 수 있다.
  4. 각각의 thread는 ITER\_CNT의 반복문을 돌면서 output을 위한 코드를 수행하고, thread\_yield( )을 수행한다.

4) 에서, 각각의 thread가 yield을 하게 되면 ready\_list에 존재하는 thread중 가장 priority가 높은 therad가 실행되기 된다. 그런데 이 때, 각 thread의 priority가 고정되어 있으므로 가장 높은 priority을 가진 thread부터 모든 iteration을 돌고, thread\_exit을 하게 된다. 따라서, 예상되는 출력값은 15부터 0까지 각각 ITER\_CNT 만큼 출력되는 값이고, 실제 테스트 결과도 동일하였다.

* make check 수행 결과를 캡처하여 첨부

텍스트, 스크린샷, 문서, 패턴이(가) 표시된 사진

자동 생성된 설명