**Pintos Project 3: Threads**

담당 교수 : 박서용

학번 / 이름 : 20160641 / 조보현

개발 기간 : 2021/11/27~2021/12/5

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

현재까지 busy-waiting을 사용해 비효율적인 Alarm Clock을 효율적으로 작동할 수 있도록 수정하고, Round-Robin 스케쥴링을 사용하는 스케쥴러를 thread의 priority를 기준으로 priority 스케쥴링을 할 수 있도록 구현한다. 또한 priority가 낮은 thread가 수행되지 않는 starvation문제를 방지하기 위해 aging 기능도 구현한다. 마지막으로 BSD 스케쥴러를 추가적으로 구현할 수 있으면 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock

기존 핀토스는 busy-waiting을 사용하므로 thread가 running 상태와 ready 상태를 반복하게 되어 매우 비효율적이다. 이 부분을 busy-waiting이 아닌 다른 방법을 사용해 효율적으로 만들수 있다.

* 1. Priority Scheduling

우선순위 스케쥴링을 구현하면 각 프로세스마다 우선순위를 부여하여 이에 따라 thread들이 스케쥴링 될 수 있다. 특히 우선순위가 높은 프로세스가 도착하면 해당 프로세스를 수행할 수 있도록 Preemtive하게 작동할 수 있다. 또한 aging을 통해 우선순위가 낮은 thread들도 점점 priority가 높아져 starvation 문제를 해결할 수 있다.

* 1. Advanced Scheduler (추가구현을 한 경우)

-구현하지 않음

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

blocked 상태의 thread들을 저장하는 큐를 만들어 특정 thread가 block될 때 해당 thread가 깨는 시간을 같이 저장하여 큐에 저장해놓는다. 이후 매 tick마다 호출되는 timer\_interrupt()함수에서 큐 전체를 검사하여 깰 시간이 된 thread를 unblock하여 깨운 뒤 ready\_list에 넣으면 된다.

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

우선순위 스케쥴링에서 preemtive 방식과 non-preemtive 방식이 동작이 다르다. preemtive 방식의 경우에는 기존 thread가 실행중이더라도 더 높은 우선순위를 가진 thread가 들어오면 바로 해당 thread에게 CPU점유를 양보해야 한다. 하지만 non-preemive 방식에서는 기존 thread보다 높은 우선순위를 가진 thread가 들어오더라도 해당 thread의 실행이 끝난 뒤 다음 thread를 실행하게 된다. 이번 프로젝트에서는 preemtive 방식을 적용한다.

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

-구현하지 않음

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

1. Alarm Clock 구현 : 2021/11/15~2021/11/20

2. Priority Scheduling 구현 : 2021/11/21~2021/11/29

3. Advanced Scheduler 구현 : 2021/11/29 ~ 2021/12/04

4. Document 작성 : 2021/12/05

* 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
* Alarm Clock

threads/thread.h에서 thread 구조체에 block된 thread들을 저장할 list인 sleepList 자료구조를 추가한다. 또한 block이 시작된 시간인 sleepStart와 block이 끝날 시간인 sleepEnd를 추가한다.

threads/thread.c에서 thread\_init() 함수에 sleepList를 초기화 하는 코드를 추가한다. block될 thread의 sleepStart, sleepEnd를 set하는 함수 thread\_sleep() 함수를 추가한다.

devices/timer.c에서 timer\_sleep()함수의 기존 busy-waiting 방식의 코드를 삭제하고 thread\_sleep()함수를 호출하도록 수정한다. 마지막으로 timer\_interrupt()함수에서 thread\_awake()함수를 호출하여 thread를 깨우는 작업까지 수행할 수 있도록 한다.

* Priority Scheduling

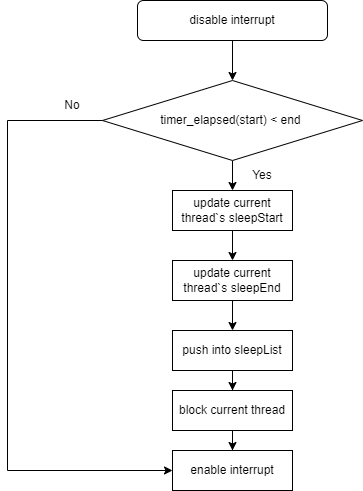
threads/thread.c에 thread들의 우선순위를 비교하는 priority\_greater()함수를 추가한다. list\_insert\_order() 함수에 priority\_greater()함수를 이용하여 thread\_unblock()함수와 thread\_yield()함수에서 thread가 우선순위에 따라 ready\_list에 들어갈 수 있도록 한다. preemtive하게 동작하도록 thread\_set\_priority()함수와 thread\_create()함수에서 현재 thread인 current\_thread()와 ready\_list의 가장 앞 thread(우선순위가 가장 높은 thread)의 우선순위를 비교하여 ready\_list의 가장 앞 thread의 우선순위가 높다면 thread\_yield()함수를 이용해 양보하도록 한다. 또한 thread\_aging() 함수를 추가하여 starvation 문제를 해결할 수 있도록 한다.

thread/synch.c에서 sema\_down()함수에서 list\_insert\_ordered()함수를 이용하여 thread를 추가하고, sema\_up()함수에서 waiters에서 깨어날 경우 가장 높은 priority의 thread가 깰 수 있도록 한다.

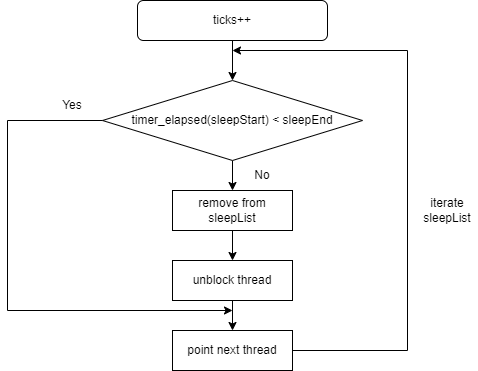
1. **연구 결과**
   1. **Flow Chart**

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

1. timer\_sleep()

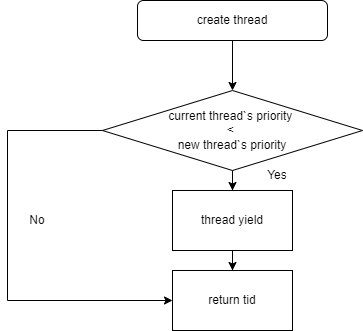


2. timer\_interrupt()

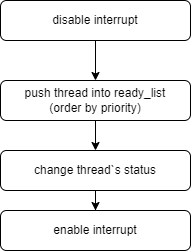


* Priority Scheduling

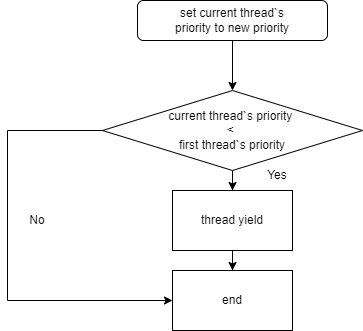
1. thread\_create()



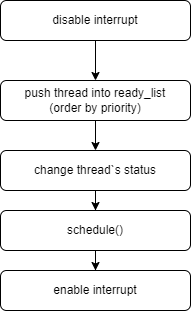
2. thread\_unblock()



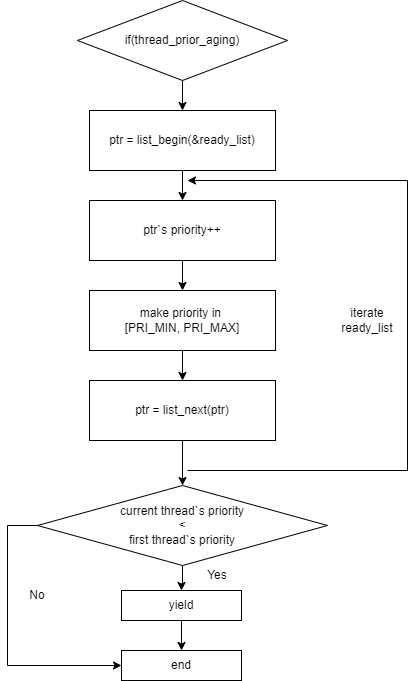
3. thread\_set\_priority()



4. thread\_yield()



5. thread\_aging()



* 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
* Alarm Clock



threads/thread.h에 block된 thread들을 저장할 list인 sleepList를 선언한다.



thread 구조체에 block 된 시작 시간과 끝날 시간을 저장하는 sleepStart와 sleepEnd 변수를 추가해준다.



threads/thread.c에서 thread\_init() 함수에서 sleepList를 초기화하는 코드를 추가한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

thread를 sleep하게 만드는 함수인 thread\_sleep() 함수를 선언한다. start에서부터 걸린 시간을 체크해주는 함수인 timer\_elapsed()함수를 사용하여 end만큼의 시간이 지나지 않았다면 sleepStart와 sleepEnd를 갱신해주고 sleepList에 넣는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

devices/timer.c에서 time\_sleep()함수에서 기존 busy-waitng 방식의 코드를 주석처리하고, thread\_sleep() 함수를 사용하도록 수정한다.

텍스트이(가) 표시된 사진

자동 생성된 설명thread/thread.c에서 thread를 깨우는 함수인 thread\_awake()함수를 선언한다. sleepList를 탐색하며 block된 thread들이 깰 시간인지 체크하여 해당 thread들을 sleepList에서 제거해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

devices/timer.c의 매 tick마다 호출되는 timer\_interrupt() 함수에서 thread\_awake()함수를 부르도록 하여 매 틱마다 체크하도록 설정한다.

* Priority Scheduling

텍스트이(가) 표시된 사진

자동 생성된 설명

threads/thread.c에서 두 thread의 우선순위를 비교하는 함수인 priority\_greater()함수를 선언한다.

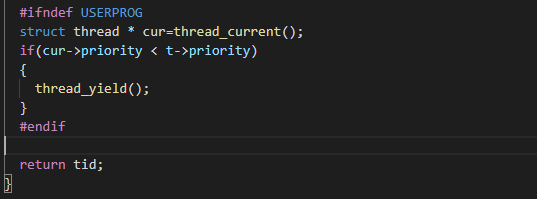
텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

thread\_unblock() 함수와 thread\_yield()함수에서 list\_insert\_ordered()함수를 이용하여 thread가 우선순위에 따라 정렬되어 ready\_list에 들어갈 수 있도록 한다.



thread\_create()함수에서 새로 만들어진 thread의 우선순위와 현재 thread의 우선순위를 비교하는 코드를 작성한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

현재 thread의 우선순위를 새로운 우선순위로 바꿔주는 함수인 thread\_set\_priority() 함수를 작성한다. ready\_list의 가장 앞 원소는 가장 우선순위가 높은 thread이므로 해당 thread와 새로 만들어진 thread의 우선순위를 비교하는 코드를 작성한다.



threads/synch.c의 sema\_down()함수에서 semaphore의 waiters에 thread가 추가될 경우 waiters가 우선순위에 따라 정렬되어 있을 수 있도록 list\_insert\_oredered()함수를 사용해 thread를 추가하도록 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

threads/thread.c에서 ready\_list안의 모든 thread들의 우선순위를 1씩 증가시키는 thread\_aging()함수를 선언한다. 이때 우선순위의 범위는 PRI\_MAX를 넘지 않고, PRI\_MIN보다 작지 않도록 설정한다. 또한 ready\_list의 가장 앞 thread와 현재 thread의 우선순위를 비교하여 yield 할 수 있도록 해야 preemptive하게 작동한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

또한 매 tick마다 thread\_aging()함수를 수행하도록 thread\_tick()함수에 코드를 작성한다.

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

priority-lifo.c 코드를 살펴보면 초기 thread의 우선순위를 PRI\_DEFAULT\_THREAD\_CNT+1로 설정한다. 이를 통해 초기 thread의 우선순위가 뒤에서 만들 thread들 보다 높도록 설정했다. 또한 초기 THREAD\_CNT개수만큼의 thread들을 만들어 ready\_queue에 넣게 된다. 여기서 만들어진 thread들은 PRI\_DEFAULT+1+i의 우선순위를 갖게 된다. 따라서 늦게 만들어진 thread의 우선순위가 높게 된다. 또한 각 thread들의 data->id는 0~15의 값을 가지게 된다. 이렇게 16개의 thread를 만든 후, 초기 thread의 우선순위를 PRI\_DEFAULT로 초기화함으로써 우선순위가 가장 낮아지게 된다.

결국 가장 마지막에 생성된 thread부터 처음 만들어진 thread까지 역순으로 thread가 수행되게 된다. 이때 각 thread는 simple\_thread\_func()함수를 수행하게 되는데, 이는 각 thread의 data->id를 output buffer에 저장하는 역할을 한다. 따라서 역순인 15부터 0까지의 결과가 16개씩 output buffer에 저장된다.

생성된 모든 thread의 수행이 끝나면 초기 thread가 ouput buffer의 내용을 출력하게 된다. 따라서 결과는 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

data->id가 역순으로 16개씩 잘 출력되는 것을 확인 할 수 있다.

* make check 수행 결과를 캡처하여 첨부

텍스트이(가) 표시된 사진

자동 생성된 설명