**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

조 / 조원 : 20211608 홍지연

개발 기간 : 2023.11.15 ~ 2023.11.19

1. **개발 목표**

핀토스 운영체제에서 scheduling의 기본 단위인 thread에 대해 round-robin 방식보다 효율적으로 작동하도록 하게 만든다. round-robin은 각 thread의 우선순위를 고려하지 않고 들어온 순서대로 일정한 time slice에 따라 순환하며 수행되는 scheduling algorithm이다. 또한 busy-wait 방식으로 구현된 alarm clock을 효율적으로 수정한다. 더불어 scheduler에 대해 발전된 방식으로 추가 구현한다.

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

현재 핀토스의 alarm clock은 busy-wait 방식으로 구현되어 있다. 이는 현재 시간(tick)을 계속 확인하고 충분한 시간이 지나기 전까지 while문 안에서 thread\_yield() 함수를 호출한다. 이를 개선하기 위해 thread 구조체에 정보를 저장한 뒤 BLOCKED 상태가 되도록 만들고 새로운 queue에서 sleep\_thread를 관리하도록 한다. 이후 timer\_interrupt() 함수를 통해 일정 시간이 지난 후 다시 READY state로 만들어 ready queue에 넣어준다.

* 1. Priority Scheduling

현재 핀토스는 Round-robin 방식이 스케줄링 알고리즘으로 구현되어 있다. Round-robin 방식은 thread의 우선순위를 고려하지 않고 들어온 순서에 따라 time slice를 할당하여 순환하는 방식이다. 이를 개선하기 위해 thread에 우선순위를 적용하여 우선순위에 따라 thread가 수행되는 선점형 priority scheduling을 구현하도록 한다. 또한 priority scheduling은 낮은 우선순위에 대해 starvation 문제가 생길 수 있다. 이를 방지하기 위해 aging을 고려하여 시간이 지날수록 우선순위가 높아지도록 한다.

* 1. Advanced Scheduler

BSD 스케줄러는 여러 multi-level queue가 각각 다른 scheduling을 하는 방식으로 동작한다. 이에 따라 여러 thread의 특성에 맞춰 다양한 스케줄링 알고리즘의 구현이 가능하다. 스케줄러는 각 우선순위마다 자신의 ready queue를 가지게 되는데, priority가 가장 높은 queue에서 thread가 선택되도록 한다. 이때 nice, priority, recent\_cpu, load\_avg 값을 통해 priority를 계산한다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

thread가 timer\_sleep() 함수로 잠들 때 잠들어 있어야 하는 시간을 알기 위해 thread 구조체에 새로운 변수 wakeup\_time을 선언하고 값을 저장한다. 이후 blocked state로 전환하고 sleep된 thread를 관리하는 list를 새로 만들어 해당 thread를 모두 저장한다. 매 tick마다 timer\_interrupt() 함수를 활용하여 list의 thread를 검사하고, wakeup\_time이 된 thread를 list에서 제거한 뒤, 상태를 unblock state로 변환하고 ready\_queue에 넣는다.

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

새로운 thread가 생성되거나, BLOCKED state의 thread가 UNBLOCKED되거나, 어떤 thread의 priority가 달라지면 running thread보다 높은 priority를 가진 thread가 들어온다. current thread를 READY state로 바꾸고 ready list에 priority 순서대로 정렬되도록 running thread에 thread를 넣고 높은 priority 순서대로 실행한다. 이를 위해 thread\_current()->priority보다 새로 추가되는 thread의 priority가 높으면 thread\_yield()를 하도록 구현한다.

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

priority의 계산식은 다음과 같다.

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

이처럼 priority 계산에는 각 thread의 recent\_cpu, nice 값이 필요하다. nice와 recent\_cpu는 thread가 생성될 때 0으로 초기화되고, 자식 thread는 부모 thread의 값을 받는다. 이때 양수인 nice의 값은 priority를 낮춰준다. recent\_cpu는 thread의 CPU time을 추정한다. time interrupt가 발생하였을 때 RUNNING state의 thread의 recent\_cpu는 1씩 증가한다. 그리고 매초마다 모든 thread의 recent\_cpu가 다시 계산된다.

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

여기서 load\_avg는 READY state의 thread의 수 평균을 추정하는 전역변수로, 매초마다 update 된다.

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

결과적으로 priority를 계산하기 위해 필요한 요소는 nice, recent\_cpu, load\_avg이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

20231115 - 20231116: Alarm clock 구현

20231117 - 20231118: priority scheduling 구현

20231118 - 20231119: advanced scheduling 구현 및 보고서 작성

* 1. **개발 방법**
* Alarm Clock

먼저 thread.h에서 thread 구조체에 int wakeup\_time을 선언하여 thread가 깨어나야 하는 시간을 저장한다. 그리고 thread\_init()에서 이 변수를 0으로 초기화한다.

다음으로 timer.c에서 BLOCKED state의 list를 저장하기 위해 struct list sleep\_list를 선언하고 timer\_init()에서 list\_init()을 통해 sleep\_list를 초기화하였다. 다음으로 timer\_sleep() 함수에서 ticks를 통해 현재 thread의 wakeup\_time을 계산하고 sleep\_list에 추가한 다음 thread를 thread\_block()으로 BLOCKED state로 변경한다.

그리고 잠든 thread가 일정 시간이 지난 이후 깨어날 수 있도록 매 ticks마다 작동하는 timer\_interrupt()를 수정하는데, thread\_wake\_up()을 호출하여 ticks가 변할 때마다 wakeup\_time이 된 thread를 깨운다. 이때 thread\_wake\_up() 함수는 새로 선언하여 사용하였다. 이 함수는 sleep\_list를 탐색하여 thread의 wakeup\_time < ticks일 때 sleep\_list에서 삭제한 뒤 thread\_unblock()을 호출하여 thread를 UNBLOCKED state로 변경한다.

* Priority Scheduling

thread.c에 thread scheduling과 관련한 코드가 구현되어 있으므로 해당 코드들을 수정하였다.

먼저 thread.h에 priority aging을 위한 extern bool thread\_prior\_aging를 선언한다. 다음으로 init.c에서 parse\_options()에 thread\_prior\_aging을 true로 값을 설정한다.

그리고 thread.c를 수정한다. 먼저 priority aging을 위해 thread\_prior\_aging을 bool type의 전역변수로 선언한다.

thread\_create()에서는 생성된 thread의 priority가 실행 중인 thread의 priority보다 클 경우 다시 scheduling을 수행한다.

thread\_yield()는 scheduler에 CPU를 제공하는 함수이다. priority 순서대로 thread를 ready\_list에 넣는다. 현재 thread를 READY state로 변경한 후 schedule()을 호출한다. 이때 두 개의 thread의 priority를 전달받아 boolean 값을 반환하는 thread\_priority\_comp() 함수를 구현하여 사용하였다.

thead\_unblock()은 BLOCKED state의 thread를 READY state로 바꾼다. 따라서 thread가 priority 순서대로 ready\_list에 들어가도록 한다. 이를 위해 list\_insert\_ordered 함수를 사용한다. 이후 status를 thread\_ready 상태로 바꾼다.

thread\_set\_priority() 함수는 current thread의 priority를 수정하기 위해 current thread의 priority에 전달받은 priority를 저장하고, 새 priority가 더 작으면 다시 scheduling을 수행한다.

* Advanced Scheduling

먼저 전역변수 load\_avg를 선언하고 소수점 연산(fixed point)을 위해 FRACTION을 define한다. 또한 priority 계산을 위해 필요한 요소인 nice와 recent\_cpu를 thread 구조체에 추가한다. 그리고 thread.c에서 init\_thread()에서 현재 실행 중인 thread의 것을 새로운 변수가 그대로 받는 코드를 추가한다. 그리고 thread\_init()에서 첫 번째로 생성된 thread와 load\_avg에 대해 0으로 초기화하는 코드를 추가한다.

다음으로 nice, recent\_cpu, load\_avg를 사용해 priority를 계산하는 함수로 update\_nice\_recent(), update\_priority()를 새로 구현하였다. 다만 핀토스는 소수점 계산에서 floating point를 지원하지 않으므로 fixed point 연산을 위해 FRACTION을 계산에 함께 사용한다. thread의 nice의 값을 새로 설정하는 함수인 thread\_set\_nice()와 현재 thread의 priority를 반환하는 thread\_get\_priority(), 현재 thread의 nice 값을 반환하는 thread\_get\_nice(), 현재 시스템의 load\_avg의 값에 100을 곱하여 반환하는 thread\_get\_load\_avg(), 현재 thread의 recent\_cpu에 100을 곱하여 반환하는 thread\_get\_recent\_cpu()를 모두 수정한다. update\_nice\_recent\_cpu()에서 모든 프로세스의 nice, recent\_cpu를 업데이트하고 load\_avg를 다시 계산한 뒤, all\_list에서 idle\_thread가 아닌 모든 thread에 대해 recent\_cpu를 다시 계산한다. 그리고 update\_priority()에서 모든 프로세스의 priority를 업데이트하고 all\_list의 모든 thread의 priority를 다시 계산한다.

다음으로 synch.c에서 sema\_up()에서 priority 관련 처리를 한다. sema를 기다리는 thread들 중 가장 큰 priority를 가진 thread를 찾아 UNBLOCKED state로 변경하고 다시 스케줄링을 하도록 만든다. 그리고 마지막으로 timer.c에서 timer\_interrupt에서 priority\_aging과 mlfqs에 대한 처리를 통해 tick이 증가함에 따라 모든 프로세스의 nice와 recent\_cpu 값을 업데이트하고 결과적으로 모든 프로세스의 priority를 업데이트하도록 한다.

1. **연구 결과**
   1. **Flow Chart**

* Alarm Clock

텍스트, 스크린샷, 도표, 직사각형이(가) 표시된 사진

자동 생성된 설명

* Priority Scheduling

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**
* **Alarm Clock**

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

thead/thread.h에 thread 구조체에 thread를 깨우는 flag 역할을 하는 wakeup\_time을 선언한다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

thread/thead.c의 thread\_init() 함수에서 file\_lock을 lock\_init()을 호출하여 초기화하고, wakeup\_time을 0으로 초기화한다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

device/timer.c에서 BLOCKED state의 list를 저장하는 전역변수 sleep\_list를 선언한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

timer\_init()에서 list\_init()을 호출하여 sleep\_list를 초기화한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

주석 처리된 while문은 busy-wait 방식으로 구현되어 있던 이전 코드이다. ticks과 현재 시간을 통해 current thread의 wakeup\_time을 구해서 sleep\_list에 추가하고, thread\_block()을 통해 BLOCKED state로 바꾼다.

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

매 tick마다 작동하는 timer\_interrupt()에서 thread\_wake\_up()을 호출하여 매 ticks마다 wakeup\_time이 된 thread를 찾아서 깨운다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

thread\_wake\_up()은 위와 같이 sleep\_list를 처음부터 끝까지 탐색한다. thread의 wakeup\_time < ticks일 때 sleep\_list에서 해당 e를 삭제하고 thread\_unblock()으로 thread를 UNBLOCKED state로 바꾼다.

* **Priority Scheduling**

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

threads/thread.h에 starvation 문제를 피하기 위한 priority aging을 위하여 thread\_prior\_aging을 선언한다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

threads/init.c의 parse\_options()에서 위와 같이 thread\_prior\_aging = true로 설정한다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

threads/thread.c에서 priority aging을 위해 bool type으로 전역 변수를 선언한다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

위와 같이 thread\_yield()를 수정한다. 이 함수는 새로운 thread를 고르는 scheduler에 CPU를 준다. 따라서 현재 thread가 idle\_thread가 아닐 경우 list\_push\_back 대신 list\_insert\_ordered()를 호출하여 ready\_list에 current thread의 elem를 추가한다. 이때 priority 순서대로 추가하기 위하여 thread\_priority\_comp()를 사용한다. 이후 thread를 READY state로 바꾸고 schedule()을 호출한다.

스크린샷, 텍스트이(가) 표시된 사진

자동 생성된 설명

여기서 사용한 thread\_priority\_comp()는 위와 같다. 전달받은 두 개의 priority를 비교하고, Boolean 값의 결과를 반환한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

thread\_create()에서 생성된 thread의 priority가 실행 중인 thread의 priority보다 클 경우 다시 scheduling을 수행한다. 즉, ready\_list가 비어있지 않으면 비교를 통해 thread\_yield()를 호출하고 scheduling을 수행한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

마지막으로 current thread의 priority를 수정하는 thread\_set\_priority()를 위와 같이 수정한다. 위 함수를 통해 current thread의 priority에 인자인 new\_priority를 새로 저장하고, 만약 new\_priority가 이전 priority보다 작다면 다시 scheduling을 하기 위해 thread\_yield()를 호출한다.

* **Advanced Scheduling**

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

threads/thread.h에 priority 계산에 필요한 요소들을 위와 같이 선언한다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

thread\_init()에서 nice와 recent\_cpu를 0으로 초기화하는 부분을 추가하였다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

init\_thread()에서 RUNNING state의 thread의 값을 초기화하여 nice와 recent\_cpu가 부모의 값을 받도록 한다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

thread\_get\_priority()가 current thread의 priority를 반환하도록 한다.

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

자동 생성된 설명

thread\_set\_nice()는 current thread의 nice 값을 인자인 nice로 설정하게 한다. nice 값이 변경되었으므로 priority를 다시 계산하는데, 계산식은 위 2-B-3에서 설명한 계산식을 사용하였다. 이때, 핀토스는 floating point 연산이 불가능하므로 fixed point을 수행하였고 이 때문에 FRACTION을 사용하였다. 또한 priority를 새로 계산하였기에 MAX, MIN을 체크하여 만약 MAX보다 크거나 MIN보다 작을 경우 각각 MAX 또는 MIN으로 재설정한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

thread\_get\_nice()는 현재 thread의 nice 값을 반환한다.

thread\_get\_load\_avg는 현재 load\_avg에 fixed point 연산을 통해 100을 곱하여 반환한다.

thread\_get\_recent\_cpu는 현재 thread의 recent\_cpu에 fixed point 연산을 통해 100을 곱하여 반환한다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

update\_nice\_recent\_cpu는 모든 프로세스의 nice 변수와 recent\_cpu 변수를 업데이트해준다. 2-B-3에서 설명한 수식을 통해 load\_avg를 계산하고, all\_list에 있는 모든 idle\_thread가 아닌 thread에 대해서 2-B-3에서 설명한 수식을 통해 recent\_cpu를 계산한다. 이때 all\_list는 모든 프로세스가 저장되어 있는 list이다.

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

자동 생성된 설명

update\_priority()는 모든 프로세스의 priority를 업데이트한다. 이때 priority가 MAX와 MIN의 범위를 벗어난다면, 각각 MAX 또는 MIN으로 다시 설정한다. 여기서 사용하는 계산식 또한 2-B-3에서 설명한 수식이다.

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

다음으로 threads/synch.c의 sema\_up()에서 priority와 관련된 코드를 추가한다. sema를 기다리는 thread가 저장된 waiters에서 thread를 탐색하여 우선순위가 가장 높은 thread를 찾는다. 그리고 list에서 해당 thread를 waiters에서 제거하고 thread\_unblock()을 호출하여 thread를 UNBLOCK state로 바꾼다. 이후 list에서 제거된 thread가 있으므로 다시 scheduling을 하기 위해 thread\_yield()를 호출한다.

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

마지막으로 device/timer.c의 timer\_interrupt()에서 priority\_aging과 mlfqs에 대한 코드를 추가한다. 매 ticks마다 thread의 recent\_cpu 값이 1씩 증가하는데, timer\_ticks가 TIMER\_FREQ만큼 증가하였다면 모든 프로세스의 nice와 recent\_cpu를 업데이트하기 위해 update\_nice\_recent\_cpu()를 호출한다. timer\_ticks가 4만큼 증가하였다면 모든 프로세스의 priority를 업데이트하기 위해 update\_priority()를 호출한다.

* 1. **시험 및 평가 내용**
* priority-lifo 결과

텍스트, 스크린샷, 패턴, 디자인이(가) 표시된 사진

자동 생성된 설명

priority-lifo.c는 동일한 priority를 가진 thread를 여러 개 만들고 thread가 round-robin에 따라 동일한 순서로 순환하여 실행되도록 한다. 즉, 16개의 thread가 LIFO에 따라 16번 반복된다. priority-lifo 실행 결과를 보았을 때, 0~15까지의 이름을 가진 thread 16개가 위에서 설명한대로 round-robin 순서대로 실행되고 있음을 알 수 있다.

priority는 PRI\_DEFAULT + i + 1이다. 따라서 처음 main thread의 자식들은 실행되지 않고, 그 후 main thread의 priority를 PRI\_DEFAULT로 설정한다. 이로서 priority가 15인 thread붜 순서대로 실행되는데, ready list의 thread는 우선순위 기준으로 정렬되고 따라서 우선순위 기준으로 thread가 실행된다. 결과적으로 가장 높은 priority의 thread는 언제 들어가든 가장 먼저, 가장 낮은 priority의 thread는 언제 들어가도 가장 나중에 실행되는 형태로 결과가 고정된다.

* make check 결과
* 텍스트, 스크린샷, 메뉴, 패턴이(가) 표시된 사진

  자동 생성된 설명