**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

조 / 조원 : 20211513 김소현

개발 기간 : 2023.10.31~ 2023.11.10

1. **개발 목표**

* Alarm Clock을 수정하여 스레드가 Ready, Running state만 가지는 것이 아니라, Blocked state를 추가하여 스레드가 sleep하는 동안 Blocked state에 머무르도록 한다.
* 핀토스의 스케줄링 방식을 RR에서 priority 스케줄링으로 변경한다.
* 추가구현으로 BSD scheduler을 구현한다. Priority 스케줄링과 유사하지만 priority에 영향을 주는 요소들이 더 많아지고, 이 요소들도 시간에 따라 값이 변화하므로 더 복잡하다.

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

기존에는 Running 스레드가 sleep하는 동안 계속 Ready state와 Running state를 반복하기 때문에 CPU를 비효울적으로 사용하는 문제가 있었다. 이를 해결하고자 Blocked state를 추가하여 sleep하는 스레드를 Blocked state로 만들고, 나중에 깨워 Ready(?Running?) state로 만들어준다. 이를 추가함으로서 스레드가 sleep하는 동안은 절대 Running state가 되지 않기 때문에 CPU를 효율적으로 사용할 수 있다.

* 1. Priority Scheduling

현재 핀토스는 Round-Robin 스케줄링을 하고 있는데, 이는 convoy effect 등의 문제가 발생할 수 있기 때문에 최선의 스케줄링 알고리즘은 아니다. 따라서 스레드에 우선순위를 부여하고, 이에 따라 스케줄링하는 priority 스케줄링을 도입한다. 이를 추가함으로서 CPU를 더 효율적으로 사용할 수 있다.

* 1. Advanced Scheduler (추가구현을 한 경우)
* Priority를 따지는데는 많은 요소가 영향을 준다. BSD 스케줄러를 구현하면 매 interrupt마다 업데이트되는 nice, recent\_cpu, load\_avg를 고려하여 priority를 새로 업데이트해주기 때문에 더 정확한 priority를 알 수 있어 더욱 효율적인 스케줄링을 할 수 있다.
  1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

스레드가 Block되는 시간이 tick 단위이므로, 타이머 인터럽트가 발생할 때 깨우면 된다. 이를 위해 스레드가 sleep하기 전에 깨어나야할 시간을 저장하고, Blocked 상태의 스레드들을 따로 리스트에 저장해둔다. 그리고 매 타이머 인터럽드 때마다 지금 깨어나야할 스레드가 있는지 찾아서 이를 Blocked 리스트에서 제거하고 Ready state로 바꿔준다.

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

현재 스레드를 Ready state로 바꾸고 새로운 스레드를 실행시킨다. 현재 스레드를 ready list에 삽입할 때는 priority 순서대로 넣어준다.

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

nice, recent cpu, load\_avg를 사용해서 priority를 계산한다.

* Nice는 priority를 낮추는 변수로, 스레드(프로세스)가 처음 생성될 때 부모 프로세스에게 물려받는다.(부모 프로세스가 없으면 0으로 세팅) 그리고 핀토스 실행 중 thread\_set\_nice()가 호출될 때 변경된다. -20~20 사이의 값을 가진다.
* Recent\_cpu는 스레드의 CPU time의 예측값으로, 스레드(프로세스)가 처음 생성될 때 부모 프로세스에게 물려받는다.(부모 프로세스가 없으면 0으로 세팅) 모든 프로세스의 recent\_cpu값은 매 타이머 인터럽트마다 새로 계산되고, 4초마다 1씩 증가한다. Running 스레드는 매초마다 증가한다. Recent\_cpu는 아래와 같은 식으로 계산한다.

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

* load\_avg는 ready state인 스레드 수의 예측값으로, 매 타이머 인터럽트마다 계산된다. 계산식은 아래와 같다.

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

* priority는 위 요소들을 사용하여 4초마다 계산한다. 계산식은 아래와 같다.

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

nice를 제외한 위 값들은 모두 float이지만, 핀토스에서는 floating point 연산을 지원하지 않기 때문에 이들에 2^14을 곱해 하위 14비트를 소수점 아래 수를 나타내는데 사용한다. 이러한 연산을 하기 위한 함수를 따로 정의해야 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10.31~11.1 : Alarm Clock 구현

11.2~11.4: aging을 제외한 priority scheduling 구현

11.5~11.9 : priority aging, BSD 스케줄러 구현

11.10 : 보고서 작성

* 1. **개발 방법**

1) Alarm Clock

우선 thread 구조체를 수정하여 스레드를 깨워야할 시간을 저장할 변수(wake\_tick)을 추가하고, sleep하는 스레드를 저장할 리스트도 추가한다.

timer.c에서는 timer\_sleep()함수를 수정하여 이 함수가 호출될 때 현 스레드를 Blocked state로 변경하기 전, wake\_tick에 일어날 시간을 저장하고 sleeping\_list에 추가하도록 한다. Timer\_interrupt()도 수정하여 여기서 sleeping 스레드를 깨우도록 한다. Sleeping\_list를 순회하며 wake\_tick과 현재 tick이 같은 스레드를 찾아 ready state로 바꿔주면 된다.

동일한 시간에 깨어나는 스레드가 여러 개 있을 때는 이 중 priority가 높은 것부터 먼저 실행되어야 한다. 따라서 thread\_unblock()을 수정하여 스레드를 priority 순으로 ready\_list에 삽입하도록 한다. Priority 순으로 삽입하는 함수 list\_in\_p\_order()을 새로 만들어서 사용한다.

2) Priority Scheduling

- 스레드 생성될 때

스레드가 처음 생성될 때 priority를 기본값으로 설정해주기 위해 thread\_start()를 수정하여 thread\_create()를 호출하면서 PRI\_DEFAULT를 넣어주도록 한다. 새로 생성된 스레드가 현재 실행 중인 스레드보다 높은 priority를 가지면 새것이 실행되어야 한다. 따라서 thread\_create()의 마지막 부분에 priority를 비교하고 thread\_yiled()를 호출하는 부분을 추가한다.

- 스레드 실행 도중

여러 개의 스레드가 동일한 lock/세마포어를 기다린다면 이 중 priority가 높은 것이 먼저 lock/세마포어를 얻어야 한다. 따라서 sema\_up()을 수정하여 세마포어의 waiters 리스트를 탐색하고 priority가 가장 높은 스레드에게 이를 주도록 한다.

스레드의 priority는 nice, load\_avg, recent\_cpu를 사용하여 계산된다. Thread.h에서 struct thread 선언 부분을 수정하여 recent\_cpu, nice 변수를 추가해준다. load\_avg는 전역변수로 선언한다. 스레드의 priority는 4초마다 새로 계산된다. Timer\_interrupt()를 수정하여 ‘-aging’ 옵션이 들어왔을 때 이러한 업데이트를 수행하는 함수를 호출하도록 한다. Init.c의 parse\_option()을 수정하여 저러한 옵션을 인식하도록 한다.

update\_priority()를 선언하여 모든 스레드의 priority를 새로 계산하게 한다. Thread\_set\_priority()는 현재 스레드의 priority를 새 값으로 바꾸는 함수이므로 그대로 구현해준다. 이 때 새 priority가 현재 priority보다 더 작으면 thread\_yield()를 호출하여 다시 스케줄링 해준다.

3) BSD scheduler

여기서는 스레드의 nice, recent\_cpu, load\_avg가 매초 업데이트 되어 priority에 영향을 준다. Timer\_interrupt()에서 매번 thread\_tick()을 호출하므로 running state 스레드의 recent\_cpu를 업데이트하는 thread\_aging()을 새로 선언하여 thread\_tick()에서 호출한다. 이와 별개로 매초 모든 스레드의 recent\_cpu가 새로 계산되므로 update\_rc\_la()를 선언하여 load\_avg와 모든 스레드의 recent\_cpu를 업데이트하도록 한다. 이 함수는 timer\_interrupt()에서 호출된다.

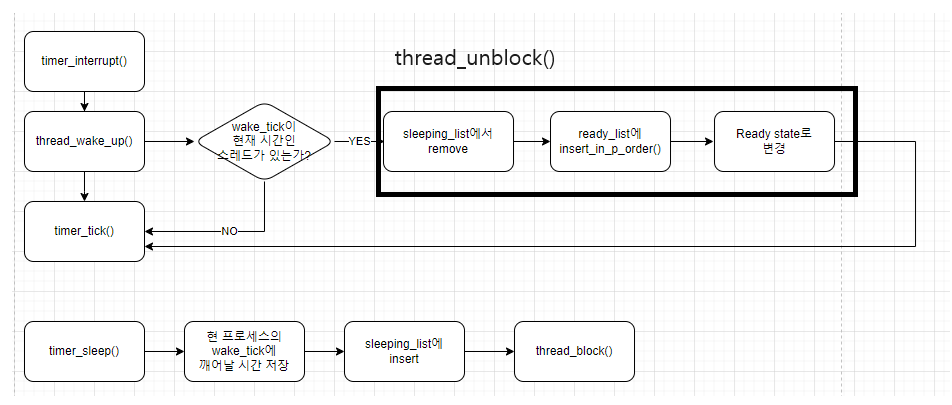
thread\_set\_nice()는 현재 스레드의 nice를 새 값으로 바꾸는 함수이므로 그대로 구현해준다. Nice가 바뀌면 priority도 바뀌므로 새로 계산해줘야 하고, 새 priority가 현재 priority보다 더 작으면 thread\_yield()를 호출하여 다시 스케줄링 해준다. Thread\_get\_ nice/load\_avg/recent\_cpu()는 현재 스레드의 저러한 값들을 리턴해주는 함수이므로 그대로 구현해준다.

위와 같은 계산을 할 때 실수가 등장하는데 핀토스에서는 floating point 연산을 지원하지 않기 때문에 정수값을 left\_shift하여 하위 14비트에 소수점 아래 값을 저장하여 int 타입으로 fixed point float를 나타낸다. 이러한 연산의 번거로움을 방지하고자 fixed\_point.h, fixed\_point.c를 만들어 fixed point 연산을 해주는 함수를 정의하여 사용한다.

BSD스케줄러에서는 priority가 aging하는 것 외에 중간에 priority를 수정할 수 없기 때문에 thread\_set\_priority()를 수정하여 ‘-mlfqs’ 옵션에 해당하는 flag가 켜져있다면 바로 리턴하도록 한다.

1. **연구 결과**
   1. **Flow Chart**

1) Alarm Clock



2) Priority Scheduling

Timer\_interrupt() 내부 로직이 바뀌었다.

도표, 텍스트, 평면도, 라인이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**
  2. Alarm Clock
* Thread.h의 struct thread

Wake\_tick 변수를 추가 선언하여 깨어나야할 틱을 여기에 저장하도록 한다.



* Timer.c

Blocked state 스레드를 저장하기 위한 리스트 sleeping\_list를 선언하고, 이를 timer\_init()에서 초기화해준다.

폰트, 텍스트, 스크린샷, 타이포그래피이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* Timer.c의 timer\_sleep()

이 스레드가 idle 스레드가 아니면 sleep하게 한다. Start가 핀토스가 부팅된 이후 지금까지 흐른 tick이므로 이 스레드가 깨어날 시간은 start+ticks이다. 스레드의 wake\_tick에 이를 저장하고 sleeping list에 삽입한 후 Blocked state로 만든다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* Timer.c의 timer\_interrupt()

Sleeping 스레드는 매 타이머 인터럽트마다 체크해서 깨워주면 된다. Sleeping list를 전부 순회하며 wake\_tick이 현재 tick과 같은 스레드를 찾으면 깨워준다. 이들을 sleeping\_list에서 제거하고, unblock시킨다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* Thread\_unblock()

해당 스레드를 Ready state로 만드는데, 이 때 priority 순위에 맞춰서 ready\_list에 넣어준다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* Thread\_yield()

CPU를 양보하는 스레드가 ready state가 되므로 ready\_list에 priority 순으로 삽입한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* Insert\_in\_p\_order()

각종 리스트에 priority 순서대로 원소를 삽입하기 위해 새로 정의한 함수이다. while문으로 입력받은 리스트를 순회하며 자신보다 priority가 크거나 같고, 다음 원소는 자신보다 priority가 낮은 원소의 위치를 찾아 거기에 삽입한다.

텍스트, 스크린샷, 소프트웨어, 운영 체제이(가) 표시된 사진

자동 생성된 설명

2) Priority Scheduling, 3) BSD scheduler

\* 한 번에 BSD 스케줄러까지 다 구현하기도 했고, 겹치는 부분이 많아 합쳐서 작성했음.

* Thread\_create()

새로 생성된 스레드가 현재 실행 중인 스레드보다 priority가 더 높다면 새 스레드에게 CPU를 양보하도록 한다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

* Synch.c의 sema\_up()

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* Thread.h

BSD scheduler에서 priority가 계속 변하는 것을 구현하기 위해 priority 계산에 필요한 변수 recent\_cpu와 nice를 struct thread의 멤버 변수로 추가한다.

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

* Thread.c의 thread\_tick과 thread\_aging()

‘-aging’이나 ‘-mlfqs’옵션이 들어왔을 때 priority가 변하도록 하기 위해 thread\_tick에서 해당 flag변수가 true일 때 thread\_aging()을 호출하도록 했다. Thread\_aging()은 새로 구현한 함수로, 현재 running 스레드의 recent\_cpu를 1증가시킴으로서 실행 중인 스레드가 매초마다 cpu 점유 시간을 업데이트하도록 했다.

텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

자동 생성된 설명

* Init.c의 parse\_options()

‘-aging’ 옵션이 들어왔을 때 해당하는 flag 변수를 true로 변경한다. 해당 flag 변수는 thread.h에서 extern bool로 선언하여 다른 파일에서도 사용할 수 있도록 한다.

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

* Thread.c의 thread\_get\_nice(), thread\_get\_recent\_cpu(), thread\_get\_load\_avg(),

thread\_get\_priority()

현 스레드의 nice, recent\_cpu, load\_avg, priority 를 리턴한다. Nice와 priority 값은 정수이지만 recent\_cpu와 load\_acg는 실수인데, 핀토스 상에서 실수를 left shift 하여 int타입에 저장해서 사용하기 때문에 다시 right shift하여 리턴해야 한다. 그리고 이들은 100배 곱하여 리턴해야 하기 때문에 아래와 같이 구현하였다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

* Thread.c의 thread\_set\_nice(), thread\_set\_priority()

현재 스레드의 nice 값을 입력받은 값으로 변경한다. nice값이 바뀌면 priority도 바뀌므로 이를 새로 계산해주고, 새 priority가 이전의 priority보다 더 낮다면 thread\_yield()를 호출하여 CPU를 양보한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Thread\_set\_priority()는 현재 스레드의 priority를 입력받은 값으로 변경한다. 여기서도 새 priority와 기존의 priority를 비교하여 새것이 더 낮다면 thread\_yield()를 호출하여 CPU를 양보한다. 그런데 MLFQ 알고리즘에서는 중간에 임의로 priority를 변경할 수 없으므로(aging으로 인해 변경되는 것 외) ‘-mlfqs’ 옵션에 해당하는 flag가 true라면 바로 리턴한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* Thread.c의 update\_rc\_la()

Load\_average를 업데이트 하고, idle 스레드를 제외한 모든 스레드의 recent\_cpu를 업데이트 해준다. 대부분 Float.h에서 정의한 함수들로 계산했지만, recent\_cpu를 계산할 때는 형변환으로 버림하는 숫자 때문에 오차가 너무 커져서 int64\_t 타입 mul\_nice\_add\_rc를 따로 계산해두고 연산에 사용했다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

* Thread.c의 update\_priority()

Idle 스레드를 제외한 모든 스레드의 priority를 업데이트 해준다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

* Timer.c의 timer\_interrupt()

‘-aging’이나 ‘-mlfqs’ 옵션이 들어왔을 때는 매초마다 load\_average와 recent\_cpu를 계산해주고, 4초마다 priority를 업데이트 해준다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

* Fixed\_point.h와 fixed\_point.c

아래와 같이 만들어 thread.c에서 priority, recent\_cpu, load\_avg를 계산할 때 활용했다. float끼리의 곱셈이나 나눗셈은 중간에 오버플로우가 일어날 경우를 방지하여 int64\_t로 형변환하여 계산한 후 다시 int 타입으로 캐스팅하여 리턴하였다.

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* 1. **시험 및 평가 내용**
* 아래는 priority-lifo.c 코드이다.

이 부분을 보면 총 16개의 스레드를 만드는데, priority가 낮은 것부터 시작하여 오름차순으로 16개를 생성한다. 그러면 priority가 가장 높은 스레드부터 실행될 것이다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

새로 생성된 스레드는 아래 함수를 실행하는데, 모든 스레드의 data->op는 같은 주소를 가리킨다. 따라서 simple\_thread\_func()를 실행하고 lock을 얻은 후에

\*(\*data->op)++=data->id를 수행하면 op라는 int 배열에 현재 스레드 data의 id를 표시할 수 있다. 즉, op에는 스레드가 실행된 순서가 기록되는 것이다.

16개의 스레드를 모두 생성하기 전까지는 Initial 스레드가 가장 높은 우선순위를 갖기 때문에 가장 먼저 lock을 얻는 것은 새 스레드 중 가장 높은 priority를 가진 스레드이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

모든 스레드가 op에 자신의 data->id를 표시한 후 반복문을 통해 op에 있는 data->id를 하나씩 가져온다. 그리고 이를 인덱스로 하여 각 스레드가 저장된 배열 data에서 해당하는 스레드를 찾아와 이의 id를 출력한다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

아래는 priority-lifo.c를 수행한 결과이다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

Priority가 높은 것부터 연속적으로 자신의 id를 출력하는 모습을 볼 수 있다. 따라서 priority 순서대로 스케줄링이 잘 됨을 알 수 있다. 그리고 priority가 낮은 것부터 먼저 생성되었기 때문에, 테스트 이름대로 나중에 생성된 스레드들이 먼저 실행되는 lifo 양상도 확인할 수 있었다.

* make check 수행 결과를 캡처하여 첨부
* 주어진 테스트를 모두 통과했다.
* 텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

  자동 생성된 설명