**Pintos Project 3: Threads**

담당 교수: 박성용 교수님

학번/이름: 20191286 김나현

개발 기간: 2021/11/20~2021/12/05

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

thread 구조체에 변수들을 추가하여 busy waiting 없이 scheduling을 하도록 구현하고 starvation problem을 방지하기 위해 thread의 priority를 일정 시간마다 aging 하도록 구현합니다. 추가적으로 average response time을 줄여주는 multilevel feedback queue scheduling을 구현합니다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

현재 구현된 timer\_clock 함수는 busy waiting을 하며 ticks의 시간이 흐르지 않으면 반복적으로 thread\_yield 함수를 호출하므로 thread가 RUNNING 상태와 READY 상태를 반복하게 됩니다. 이는 효율적이지 않은 방법이므로 이러한 비효율적인 busy waiting을 피하기 위해 매 tick마다 호출되는 timer\_interrupt 함수를 수정할 것입니다.

* 1. Priority Scheduling

지금까지 사용한 pintos는 round-robin scheduling 방법에 의해 thread를 scheduling하였지만 본 프로젝트에서는 thread들이 자신만의 priority를 가지며 현재 cpu를 차지하고 있는 thread보다 더 높은 priority를 갖는 thread가 들어오면 cpu를 해당 thread에게 양보하도록 구현해야 합니다. Starvation problem을 방지하기 위한 priority aging 방법은 pintos kernel이 ‘-aging’이라는 옵션을 받았을 때만 thread\_prior\_aging 전역 변수 flag가 1(true)로 설정되므로 이 경우에만 사용이 가능합니다.

* 1. Advanced Scheduler (추가구현을 한 경우)

Thread의 average response time을 줄여주는 Multilevel feedback queue scheduling을 구현하기 위해 Advanced Scheduler를 추가적으로 구현해야 한다. 다른 priority scheduling 방법과 마찬가지로이 scheduling 방법 또한 thread의 priority를 토대로 수행할 thread를 고르지만, priority donation은 하지 않는다는 차이가 있습니다. pintos kernel이 ‘-aging’이라는 옵션을 받았을 때만 thread\_prior\_aging 전역 변수 flag가 1(true)로 설정되어 priority aging을 위해 구현된 코드를 사용할 수 있었던 것과 같이, BSD scheduler를 사용하기 위해서도 pintos kernel이 ‘-mlfqs’라는 옵션을 받아야 합니다.

* 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)

THREAD\_BLOCKED 상태가 된 thread들은 timer\_interrupt 함수에 의해 매 초마다 get\_up\_time과 ticks를 비교하여 get\_up\_time이 tick보다 작거나 같을 때, thread\_unblock의 argument로 passing되어 thread\_unblock 함수가 호출되게 합니다. thread\_unblock 함수에서는 blocked된 상태의 thread를 THREAD\_RUNNING 상태로 바꿔주고, list\_insert\_ordered라는 함수에 의해 자신의 priority에 맞는 자리를 찾아 ready\_list에 들어가게 합니다.

현재 ready\_list에 있는 thread의 priority 중 가장 높은 priority 값을 반환하는 max\_priority 함수를 만들어 thread\_set\_priority나 thread\_set\_nice, priority\_aging 함수 등에서 thread\_current()->priority보다 max\_priority()가 더 클 경우에 thread\_yield나 intr\_yield\_on\_return 함수를 호출하여 더 높은 priority를 갖는 thread에게 cpu를 양보하도록 구현할 수 있습니다.

Pintos kernel은 floating-point 연산을 지원하지 않기 때문에 Advanced scheduler에서는 real number인 recent\_cpu와 load\_avg의 연산을 위해 floating-point 연산을 대체하기 위해 사용되는 fixed-point 연산이 사용됩니다. 이때, fixed-point 연산을 위해 pintos/src/threads 디렉토리에 floating\_operation c파일, h파일을 만들어 특정 함수들을 통해 해당 연산을 가능하게 합니다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

float\_operation.c, float\_operation.h 만들기: fixed-point real arithmetic을 위한 함수들 구현

thread.c, thread.h, init.c 수정하기: thread 구조체에 들어가야 할 변수 추가, system wide value 변수들 선언, 파일에 있는 함수 수정 및 추가 구현하기

ready\_list에 list\_push\_back 말고 list\_ordered\_insert를 이용하여 들어갈 수 있게 변경하기

priority-aging, priority-sema 테스트 케이스 통과하게 코드 수정하기

BCD scheduler 구현하기

* 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

pintos/src/threads 디렉토리에 있는 thread.h 파일의 thread 구조체 선언부에 int get\_up\_time; int64\_t recent\_cpu; int nice;를 추가하여 thread의 get\_up\_time, recent\_cpu, nice 정보를 저장할 수 있게 하였고, struct list sleep\_list;를 전역 변수로 사용하여 src/devices/timer.c 파일에서 blocked 상태의 thread에 관한 정보를 저장할 수 있도록 하였습니다. 또한, extern bool thread\_mlfqs; extern bool thread\_prior\_aging; 선언하여 본 프로젝트에서 pintos system의 전반적으로 사용되는 system wide value인 thread\_mlfqs와 thread\_prior\_aging 변수를 thread.c 파일에서도 사용 가능하게 하였고, max\_priority나 priority\_aging, update\_recentcpu\_and\_loadavg 등과 같은 thread.c 파일에서 추가적으로 구현한 함수들의 prototype을 선언하였습니다.

thread.c 파일에서는 bool thread\_mlfqs; bool thread\_prior\_aging; static int load\_avg;를 통해 전역 변수들을 선언해주었고, bool형 데이터를 반환하는 less\_priority 함수를 선언 및 정의해주었습니다. 또한 ready\_list에 있는 thread의 priority 중 가장 큰 값을 반환할 수 있는 max\_priority 함수를 정의하였고, 매 초마다 timer\_interrupt 함수에서 호출되어 모든 thread의 recent\_cpu 값과 load\_avg 값을 update 해주는 update\_recentcpu\_and\_loadavg 함수를 정의하였으며 thread\_get\_priority, thread\_set\_priority, thread\_set\_nice, thread\_get\_nice, thread\_get\_load\_avg, thread\_get\_recent\_cpu 함수를 수정 또는 추가 구현하였습니다. thread\_get\_recent\_cpu와 thread\_get\_priority 함수는 각각 현재 돌고 있는 current thread의 recent\_cpu 값의 100배, priority를 반환하고, thread\_get\_load\_avg 함수는 system load average 값의 100배를 반환합니다. thread\_set\_priority 함수는 현재 돌고 있는 current thread의 priority를 argument로 전달 받은 값으로 바꿔주고 max\_priority 함수를 이용해 구한, 현재 ready\_list에 있는 thread의 priority 중 가장 큰 값보다 바뀐 current thread의 priority 값이 작으면 thread\_yield 함수를 통해 cpu를 양보하는 역할을 합니다. thread\_set\_nice 함수는 현재 돌고 있는 current thread의 nice를 argument로 전달 받은 값으로 바꿔주는데, 이때 thread의 nice 값이 변하면 priority 값 또한 변하기 때문에 thread\_set\_nice 함수에서는 현재 돌고 있는 current thread의 nice 값을 바꾸고, priority를 다시 calculate하여 priority가 PRI\_MAX보다 크면 PRI\_MAX로, PRI\_MIN보다 작으면 PRI\_MIN으로 설정합니다. 그리고 max\_priority 함수를 호출하여 바뀐 current thread의 priority보다 큰 priority를 갖는 thread가 있다면 thread\_yield 함수를 이용해 cpu를 양보하는 역할을 합니다. priority\_aging 함수는 4 ticks마다 호출되는 함수로, 모든 thread의 priority를 증가시켜줍니다. 이때, thread의 priority를 변경하면 list\_sort 함수를 호출해 ready\_list에 있는 thread들이 priority가 큰 거부터 나열될 수 있도록 정렬해줍니다. While 문이나 for 문을 이용한 priority aging이 끝나면 현재 돌고 있는 thread의 priority가 max\_priority 함수를 통해 구한 priority보다 작으면 intr\_yield\_on\_return 함수를 호출합니다. thread\_init 함수에서는 list\_init 함수를 이용하여 ready\_list를 초기화해주고, initial\_thread의 nice와 recent\_cpu를 0으로 초기화해줍니다. 이때, 부모 thread로부터 생성된 자식 thread는 부모의 nice와 recent\_cpu를 전달 받는다고 하였으므로 init\_thread 함수에서는 t->recent\_cpu와 t->nice를 각각 running\_thread()로 구한 부모 thread의 recent\_cpu와 nice로 초기화해줍니다.

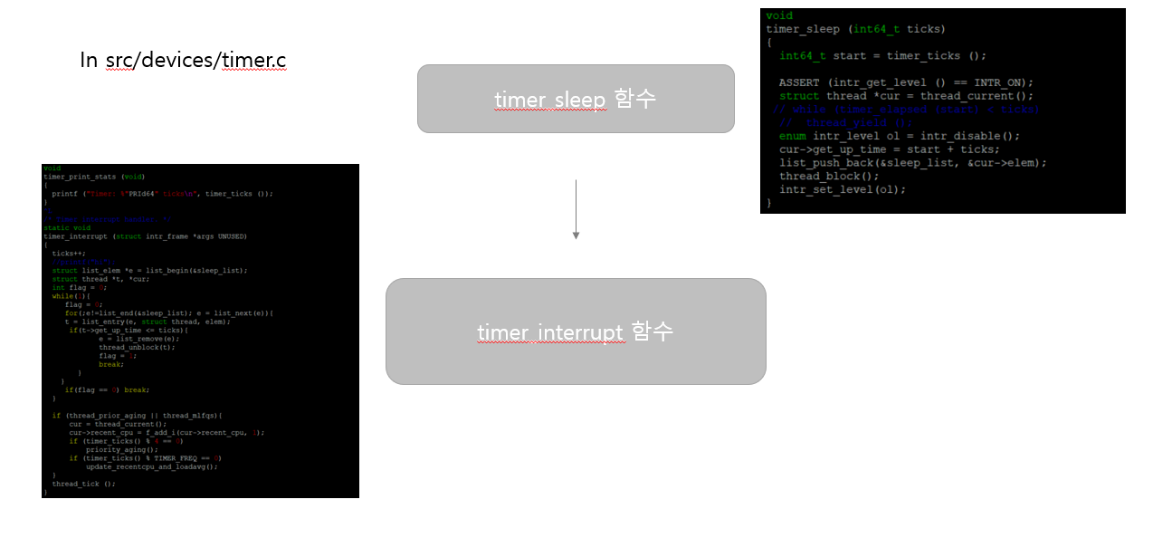
init.c 파일에서는 parse\_options 함수를 수정하여 name과 “-aging”이 같으면 thread\_prior\_aging을 1로 설정할 수 있도록 합니다.

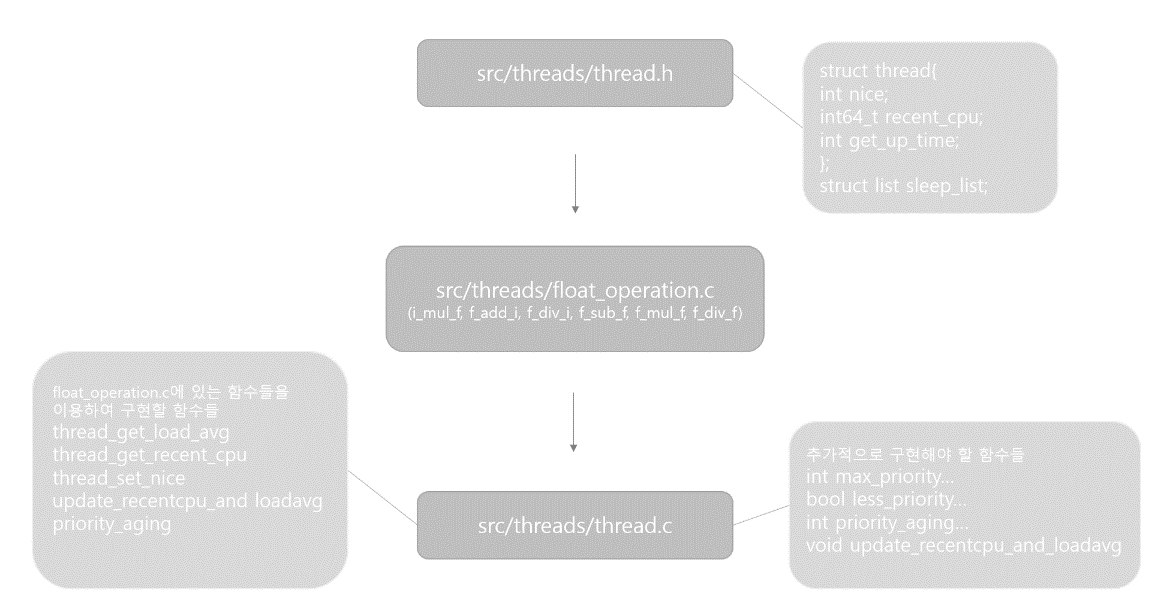
Fixed-point 연산을 위해 pintos/src/threads 디렉토리에 float\_operation.c, float\_operation.h 파일을 추가로 구현하여 thread.c 파일에서 fixed-point 연산을 위해 사용할 함수들을 구현해줍니다. 본 프로젝트에서 fixed-point 연산을 할 변수들은 priority와 recent\_cpu, load\_avg인데 이 연산에는 i를 int, f를 float라고 하고, 덧셈을 add, 뺄셈을 sub, 곱셈을 mul, 나눗셈을 div라고 하면 i\_mul\_f, f\_add\_i, f\_div\_i, f\_sub\_f, f\_mul\_f, f\_div\_f, 이렇게 6개의 연산이 필요하므로 각기 다른 여섯 개의 함수를 구현하면 됩니다.

BCD scheduler를 구현하기 위해 thread\_set\_priority 함수에서 thread\_mlfqs가 true이면 더 이상 함수를 수행하지 않고 바로 return하도록 하였고, thread\_mlfqs가 true면 thread\_prior\_aging가 true일 때와 마찬가지로 timer\_interrupt 함수가 호출될 때마다 current thread의 recent\_cpu가 1씩 증가하고 TIMER\_FREQ마다 update\_recentcpu\_and\_loadavg 함수를 호출하며 4 ticks마다 priority\_aging 함수를 호출하도록 구현했습니다. 위에서 언급한 thread.c 파일의 thread\_set\_nice 함수 역시 BCD scheduler를 구현하기 위한 함수입니다.

1. **연구 결과**
   1. **Flow Chart**

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





* 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

현재 구현된 timer\_clock 함수는 busy waiting을 하며 ticks의 시간이 흐르지 않으면 반복적으로 thread\_yield 함수를 호출하므로 thread가 RUNNING 상태와 READY 상태를 반복하게 됩니다. 이는 효율적이지 않은 방법이므로 이러한 비효율적인 busy waiting을 피하기 위해 매 tick마다 호출되는 timer\_interrupt 함수를 수정할 것입니다. 이때, timer\_interrupt 함수에서는 sleep\_list라는 struct list형의 새로운 queue에 저장된 thread 정보를 앞에서부터 하나하나 거치며 해당 thread가 깨어나야 될 시간인 get\_up\_time이 ticks의 시간보다 작거나 같으면, 즉 thread가 깨어날 시간이 되면 sleep\_list에서 해당 thread를 제거하고 thread\_unblock 함수를 통해 thread를 깨워주며 아직 thread가 깨어날 시간이 되지 않았을 때는 queue에서 다음 thread로 넘어가, 같은 방법으로 thread의 기상 여부를 확인합니다. 또한 timer\_interrupt 함수가 호출될 때마다 RUNNING 상태에 있는 thread가 1씩 증가해야 하므로 thread\_prior\_aging이라는 전역 변수가 1일 때, thread\_current의 recent\_cpu를 하나 증가시켜주고, 매 4 ticks마다(tick를 4로 나누었을 때, 나머지가 0일 때마다) 시스템에 있는 모든 thread의 priority를 다시 계산해줘야 하므로 thread의 priority를 증가시켜주는 함수, priority\_aging 함수를 호출합니다. 또한, timer\_interrupt 함수가 호출될 때, 매초마다(tick가 TIMER\_FREQ일 때마다) 모든 thread의 recent\_cpu 값과 전역 변수 load\_avg는 update되어야 하므로 recent\_cpu와 load\_avg를 update하는 함수, update\_recentcpu\_and\_loadavg 함수를 호출합니다.

지금까지 사용한 pintos는 round-robin scheduling 방법에 의해 thread를 scheduling하였지만 본 프로젝트에서는 thread들이 자신만의 priority를 가지며 현재 cpu를 차지하고 있는 thread보다 더 높은 priority를 갖는 thread가 들어오면 cpu를 해당 thread에게 양보하도록 구현해야 합니다. 이때, thread의 priority는 높은 값이 높은 priority, 낮은 값이 낮은 priority를 의미하며 PRI\_MIN, 0부터 PRI\_MAX, 63 사이의 값을 가지고 default 값은 PRI\_DEFAULT, 31입니다. 처음에 thread가 만들어질 때, thread\_create 함수에서 priority에 대한 정보를 argument로 전달 받아 새로 생성된 thread의 변수로 저장하게 됩니다. Default priority scheduling, 즉 thread가 생성된 이후에 priority가 바뀌지 않는 scheduling 방법의 경우, priority가 낮은 thread에 대해서는 starvation problem이 발생할 수 있으므로 이러한 문제를 해결하기 위해 본 프로젝트에서는 priority aging 기법을 사용합니다. Priority aging 방법은 thread가 ready list에 들어간 후부터 특정 시간이 지날 때마다 priority가 일정하게 증가하는 방법으로, priority aging을 위해 구현된 코드들은 pintos kernel이 ‘-aging’이라는 옵션을 받았을 때만 thread\_prior\_aging 전역 변수 flag가 1(true)로 설정되므로 이 경우에만 사용이 가능합니다. Initial thread의 nice는 초기에 0으로 설정되며 일부 thread들은 부모로부터 상속받은 nice로 초기화됩니다. Nice를 이용한 priority calculation은 최근에 CPU time을 받았던 thread에게 낮은 priority를 줌으로써 다른 thread들의 starvation을 방지합니다.

BCD scheduler를 구현하기 위해 thread\_set\_priority 함수에서 thread\_mlfqs가 true이면 더 이상 함수를 수행하지 않고 바로 return하도록 하였고, thread\_mlfqs가 true면 thread\_prior\_aging가 true일 때와 마찬가지로 timer\_interrupt 함수가 호출될 때마다 current thread의 recent\_cpu가 1씩 증가하고 TIMER\_FREQ마다 update\_recentcpu\_and\_loadavg 함수를 호출하며 4 ticks마다 priority\_aging 함수를 호출하도록 구현했습니다. 위에서 언급한 thread.c 파일의 thread\_set\_nice 함수 역시 BCD scheduler를 구현하기 위한 함수입니다.

처음에 list\_insert\_ordered 함수를 사용하지 않고 list\_push\_back을 이용하여 thread를 ready\_list에 저장하였는데 그렇게 하였더니 priority 순서대로 ready\_list에 저장되지 않아 alarm-priority 테스트 케이스를 통과할 수 없었습니다. 따라서, Boolean형 less\_priority 함수를 추가적으로 구현하고 list\_insert\_ordered 함수를 이용하여 ready\_list에 priority 순서대로 thread를 저장하였더니 해당 테스트 케이스를 통과할 수 있었습니다.

priority-sema 테스트 케이스가 통과되지 않을 때는 src/threads/synch.c를 수정하여 struct list\_elem 포인터 e를 list\_next(e)로 하나씩 바꿔가면서 e가 list\_end(&sema->waiter)가 되기 전까지 증가시키면서 while 문을 돌면서 priority가 가장 큰 thread를 찾고 thread\_unblock 함수를 통해 해당 thread를 unblock해주도록 구현하여 문제를 해결할 수 있었습니다.

float\_operation.c 파일에 구현한 함수를 이용하지 않고 integer와 real number를 FRACT을 이용하여 일일이 더하고, 빼고, 나누고, 곱하고 하였더니 식이 너무 복잡하고 맞는지 틀린지 알 수가 없었는데 float\_operation.c에 fixed-point real arithmetic에 필요한 함수를 미리 다 구현해놓고 쓰임에 맞게 사용하였더니 priority, recent\_cpu, load\_avg를 구하는 식을 틀린 곳 없이 잘 구현할 수 있었습니다.

* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석
* make check 수행 결과를 캡처하여 첨부

텍스트이(가) 표시된 사진

자동 생성된 설명

priority-lifo.c를 보면 16개의 thread를 같은 순서로 16번 돈다는 것을 알 수 있습니다. 16개의 thread들은 for 문을 돌면서 struct simple\_thread\_data 포인터 d를 이용하여 각 thread의 정보를 저장하게 됩니다. 이때, for 문을 돌면서 i라는 이름의 thread의 priority를 PRI\_DEFAULT+1+i로 설정되기 때문에 thread 0부터 thread 15까지 각각 PRI\_DEFAULT+1부터 PRI\_DEFAULT+16까지의 priority를 갖게 됩니다. priority-lifo.c 파일을 보면 가장 priority가 높은 thread부터 16번의 출력을 끝내면 해당 thread의 priority를 PRI\_DEFAULT로 바꿔주고 두 번째로 priority가 높은 thread 14, thread13, …순서대로 출력을 하게 되는데 pinto -v -- -q run priority-lifo라는 명령어로 확인해본 결과, 예상과 동일한 출력을 얻었다는 것을 알 수 있었습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명