**Pintos 프로젝트 1. Pintos Thread**

**(설계 프로젝트 수행 결과)**

**과목명 : [CSE4070-02] 임베디드시스템소프트웨어**

**담당교수 : 서강대학교 컴퓨터공학과 김 영 재**

**조원 : 28조 안진우**

**개발기간 : 11월15일 ~ 11월28일**

**최 종 보 고 서**

**프로젝트 제목: Pintos 프로젝트 1. Pintos Thread**

**제출일: 2017.11.28**

**참여조원: 28조 안진우**

**I. 개발 목표**

이번 프로젝트에서는 thread의 scheduling을 직접 구현하는 것이 주요 목표이다. 기본 pintos가 FIFO 방식의 단일 큐 형태로 스케줄링이 구현되어 있는데, 이를 개선하여 piority를 index로 가지는 64 queue 방식의 priority scheduling을 구현하고, 더 나아가 여기에 response time을 보장하기 위한 BSD scheduling과 starvation을 방지하는 Aging을 구현해야 한다. 추가로 busy waiting방식의 sleep기능을 제공하는 alarm clock을 개선하여 BLOCK방식으로 sleep기능을 구현해야 한다. 이를 통해 23개의 test case를 적절히 통과해야 한다. 또 이번에 구현한 thread 프로젝트가, 이전 프로젝트인 User program과 맞물릴 수 있게 통합하여, 기존의 User program의 test case 76개역시 문제없이 통과하는 것도 굉장히 중요한 목표라고 생각한다.

**II. 개발 범위 및 내용**

1. **개발 범위**

alarm clock을 구현하기 위해서는, 타이머 관련 모듈에서 sleep(), timer\_interrupt()함수를 재개발해야 한다. 또, priority queue를 구현하기 위해 thread 모듈의 ready queue를 64배열로 손질하고, 모든 ready queue과 관련된 처리를 하는 모듈을 priority방식으로 수행하도록 바꾸고, priority가 바꼈을시 thread\_yield함수를 통해 context switch가 일어나도록 해야한다. 추가 구현인 BSD scheduling의 경우는, 소수점 연산이 불가능한 핀토스를 위해 fixed point연산을 object로 생성하고, timer interrupt에서 주기적으로 priority를 재계산하도록 코드를 추가해야 한다.

**나. 개발 내용**

1. blocking 방식의 alarm clock 구현

현재 pintos는 sleep이 busy waiting방식으로 이루어진다. 프로세스는 사용되지 않는 sleep도중에도 cpu를 소모하여 오버헤드가 커지는데, 이를 막기 위해 blocking 방식으로 alarm clock을 구현한다.

2. 64 queue 방식의 priority scheduling 구현

현재 pintos는 단일 큐 방식의 FIFO 스케줄링을 수행한다. 여기에 priority기능을 추가하여 priority에 따라 우선순위로 스케줄링되는 priority scheduling을 구현하고, starvation을 방지하기 위해 흐른 시간에 비례하여 priority를 높여주는 aging기능을 추가한다.

3. priority에 따라 unblock되는 semaphore 구현

기존의 스케줄링 방식에 따라 semaphore내에서 lock된 process는 FIFO 순서로 wake된다. 하지만 process에 priority가 생겼기 때문에, priority가 높은 process를 우선적으로 wake시켜주어야 한다.

4. response time이 보장되는 BSD scheduling 구현

2에서 구현한 64 multi-level ready queue에 BSD scheduling을 구현한다. BSD scheduling은 최근 cpu 사용빈도를 통해 우선순위를 주기적으로 재조정한다. 최근 cpu사용이 많으면 priority를 낮추는 방식으로 fair하게 process가 스케줄링되도록 하고, 모든 프로세스에 대한 response time을 줄여준다.

5. Fixed point 구현

BSD scheduling에서 priority를 재계산하는 과정에서 소수점 연산이 사용된다. 하지만 pintos에는 소수점 연산을 제공하지 않기 때문에, fixed point를 구현하여 소수점 연산을 software적으로 수행하도록 해야한다.

6. 기존 User program과의 통합

이번 프로젝트를 구현하면서 전체 코드가 수정되다보니, 기존의 User program에서 통과하던 76개의 테스트케이스중 일부가 통과하지 않는 문제가 발생하였다. 이는 thread scheduling구현중에 예상치 못한 변화로 기존의 Userprog를 손상시킨 것인데, 이 테스트케이스가 잘 돌도록 고쳐야 핀토스 프로그램이 성공적으로 통합되었다고 할 수 있다.

**III. 추진 일정 및 개발 방법**

**가. 추진 일정**

|  |  |
| --- | --- |
| 11/15 ~ 11/19 | pintos 메뉴얼 분석 및 코드 이해 |
| 11/20 ~ 11/24 | alarm clock, priority scheduling 기본 구현 및 테스팅 |
| 11/25 ~ 11/27 | BSD scheduling 추가 구현 및 테스팅 |
| 11/28 ~ 11/28 | 최종 보고서 작성 |

**나. 개발 방법**

이번 프로젝트에서는 안정된 scheduling 및 alarm을 구현하는 것이 목표이다. 그런데 이 과정에서 기존에 구현한 User program의 테스트케이스를 통과하지 못하는 경우가 다수 발생하였다. 따라서 이 에러 역시 수정하여 thread, userprogram의 테스트 케이스를 모두 통과하는 성공적인 pintos를 구현하는 것 역시 중요하다고 생각했다.

1. blocking방식의 alarm clock

busy waiting으로 돌던 alarm clock을 blocking방식으로 바꾸기 위해 wake time과 blocked thread를 저장할 큐를 새로 생성했다. sleeping때는 therad block후 이 큐에 삽입하고, timer interrupt를 통해 주기적으로 큐를 체크하여 wake time이 지난 thread를 깨워주었다.

2. 64 queue 방식의 priority scheduling

priority scheduling을 구현하기 위해 기존 단일 큐 방식의 ready queue를 64개의 배열을 갖는 multi-level queue로 다시 생성하였다. 이에 따라 ready queue를 생성, 삽입, 삭제 등 관리하는 모든 모듈을 수정하였다.

-thead\_unblock() : blocking된 thread를 wake시켜 ready queue에 삽입하는데, priority에 맞춰 삽입하고, 만일 해당 priority가 running thread보다 높으면 즉시 thread\_yield()를 통해 리스케줄링 하였다.

-next\_thread\_to\_run() : ready queue에서 running할 thread를 결정하는 함수로, 여기선 priority가 가장 높은 ready queue이 thead를 round robin방식으로 결정했다.

-thread\_yield() : 현재 running thread를 ready queue로 집어넣을때 우선순위에 맞는 index로 집어넣었다.

3. priority에 따라 unblock되는 semaphore

semaphore에서는 blocked thread를 깨울때 priority순으로 깨워야한다. 따라서 blocked thread의 priority를 모두 조사하여, 가장 높은 priority를 가진 thread를 깨웠다.

4. thread aging

priority scheduling에서는 starvation문제가 발생할 수 있기 때문에 thread aging을 통해 주기적으로 priority를 올려주어야 한다. thread에 recent\_unused 변수를 추가하여 마지막으로 실행된 시간을 정하였다. timer interrupt handler함수에서는 이 변수를 이용하여 실행되지 않은 시간을 계산하고, 1초가 넘어갈때마다 priority를 증가하도록 구현하였다.

5. response time이 보장되는 BSD scheduling

BSD scheduling 을 위해 thread에 nice, recent\_cpu vaiable을 추가하였다. timer interrupt에서는 사용되는 load average와 recent\_cpu는 1초마다 재계산하였다. 또, recent cpu는 매 tick마다 1씩 증가시켰다. 또 priority를 4tick마다 재계산하였는데, priority가 바뀌면 ready queue에 새로 배치하고, intr\_yield\_on\_return()함수를 통해 인터럽트가 끝날때 yield 가 수행되어 리스케줄링 되도록 구현하였다.

6. Fixed point

BSD scheduling에서는 recent cpu, load average에 소수점연산이 사용되므로, fixed point연산을 object형태로 구현하였다. convert to int, convert to fp, add, subtract, mult, devide함수를 구현하여 해당 함수를 통해 적절한 연산이 수행되도록 하였다.

7. 기존 User program과의 통합

thread scheduling방식을 통째로 바꾸다보니 기존 User program의 기능이 손상되는 경우가 다수 발생하였다. 따라서 User prog도 지속적으로 테스팅을 하여 에러를 수정하고 all pass시켜서 통합을 완성하였다.

**다. 연구원 역할 분담**

1. 핀토스 메뉴얼 분석 및 코드 이해 : 안진우 100%

2. 프로그램 생성 및 구현 : 안진우 100%

3. 테스팅, 에러 처리 및 주석, 보고서 : 안진우 100%

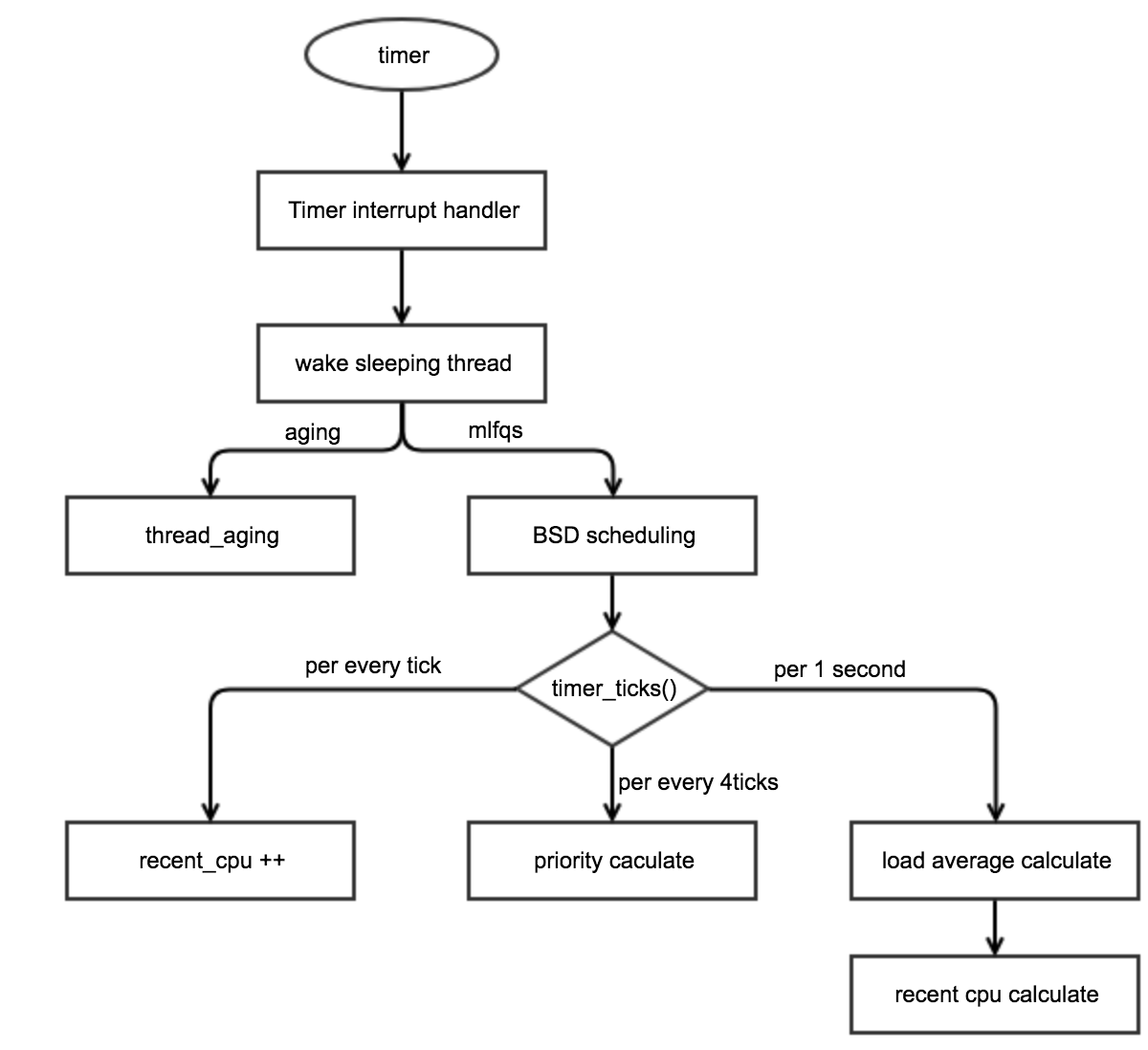
**IV. 연구 결과**

1. 합성 내용:

1.1. 프로그램 실행 구성도

1. timer 발생시 flow

이번 프로젝트에서 대부분의 구현은 timer handler함수 내에서 이루어졌다. interrupt handler함수에서는 sleeping thread를 깨우는 alarm clock과 starvation을 방지하는 priority scheduling의 thread\_aging, response time을 보장하는 BSD scheduling이 발생한다.



2. 제작 내용: 개발 결과

1. blocking 방식의 alarm clock

|  |
| --- |
| 기존의 cpu 소모가 큰 busy waiting방식의 alarm clock을 개선하여 blocking방식으로 수행되도록 하였다. 이를 통해 cpu 소모 오버헤드를 효과적으로 감소시켰다.  1. list\_alarm  sleep된 thread는 나중에 timer interrupt에서 깨워줄때까지 보관해야 한다. 이를 위해 list\_alarm을 선언하고, 각 element로 thread\_alarm을 선언하여, 해당 구조체에 thread와 깨울 시간을 저장할 수 있도록 하였다.  스크린샷%202017-11-28%20오후%203.22.29.png  2. timer\_sleep()  timer\_sleep에서는 기존의 busy waiting방식으로 sleep시키던 것을 blocking방식으로 수정하였다. 이를 위해 list\_alarm에 깨울 시간과 thread를 삽입하고 thread\_block() 을 시켜 thread를 sleep하였다. list\_alarm은 critical section이기 때문에 intr\_disable을 하여 atomic하게 수행토록 하였다.  스크린샷%202017-11-28%20오후%203.23.34.png  3. timer\_interrupt()  timer interrupt 함수는 매 10ms마다 타이머에 의해 호출되는 함수이다. 따라서 sleep에 의해 block된 thread들은 timer interrupt함수에서 깨울 시간이 지났는지 판단하도록 하였다. 매 함수가 호출될 때마다 list\_alarm을 linear하게 조사하여 깨울 시간이 되었으면 thread\_unblock을 통해 깨우도록 구현하였다. 또, 깨운 thread는 list\_alarm으로부터 제거하도록 구현하였다.  여기서 한가지 해결못한 이슈가 있었는데, 이전 프로젝트에서 multi-oom을 통해 free()의 중요성을 깨달은 바라, list에서 제거된 element를 free 시키고자 시도하였으나, 에러가 발생했다. 자세히 분석해보니, free함수는 내부에서 lock을 호출되기 때문에 sleep될 가능성이 있고, interrupt context는 sleep이 될 수 없기 때문에 발생한 문제였다. 따라서 interrupt context에서 어떻게 할당받은 메모리를 해제할것인지는 조금 더 연구가 필요할 것 같다. 다만 이 문제는 현재로선 userprog, thread 테스팅에 악영향을 미치지 않는다.  스크린샷%202017-11-28%20오후%203.25.22.png |

2. 64 multi level queue 방식의 priority scheduling

|  |
| --- |
| 이번 프로젝트의 주요 이슈는 thread scheduling이다. priority scheduling을 구현하기 위해 기존의 단일 ready queue 를 64개의 배열을 갖는 multi-level ready queue로 수정하고, 인덱스가 priority를 의미하도록 설정하였다. 따라서 관련된 컨트롤을 하는 모든 모듈을 수정하였으며, priority가 바뀔 경우에는 ready queue 내의 위치를 재조정하고, 필요에 따라 thread\_yield를 호출하여 rescheduling되도록 하였다.  1. 64 level ready queue  가장 먼저 64개의 priority를 각기 다른 인덱스로 관리하기 위하여 ready\_list를 64개의 배열을 갖는 multi level queue로 수정하였다.  스크린샷%202017-11-28%20오후%203.33.57.png  2. thread\_unblock  ready queue 구조자체를 통째로 수정함에 따라 ready queue에 삽입 또는 삭제하는 모든 모듈의 구조를 바꿔야 했다. thread\_unblock함수는 I/O queue에 있는 thread를 ready queue에 삽입하는데, priority에 맞게 적절한 삽입이 필요하다. 뿐만 아니라, 현재 삽입된 ready queue의 priority가 running thread보다 높을 경우 thread\_yield()를 통해 리스케줄링 하여 즉시 running되도록 해야한다.  스크린샷%202017-11-28%20오후%203.37.47.png  여기서 thread\_yield()함수는 interrupt disable일 경우에는 호출되서는 안된다. 이 이유는 interrupb disable일때 preemption을 하면 원래 disable한 목적인 atomic한 수행이 불가능하기 때문이라 추정한다. 까라서 intr\_get\_level()함수를 통해 enable일 상태일 경우에만 yield하도록 구현하였다.  사실 이는 굉장히 번거로올 수 밖에 없는데, 왜냐하면 이 경우 interrupt disable을 하고 thread\_unblock함수를 호출한 함수에 대해서는, 해당 함수 내에서 atomic한 수행이 끝나고 다시 thread\_yield()함수를 호출해야 하기 때문이다. 따라서 thread\_unblock()함수를 호출하는 모듈을 일일히 조사하였고, 실제로 semaphore의 sema\_up함수에서 atomic한 수행이 끝난 뒤에 thread\_yield()를 호출하도록 아래 그림과 같이 구현하였다.  스크린샷%202017-11-28%20오후%203.37.47.png  3. next\_thread\_to\_run  next\_thread\_to\_run함수는 thread\_yield()함수에서 리스케줄링될때, 다음 run할 thread를 ready queue로부터 가져온다. 여기서 가장 높은 top priority를 가진 thread를 가져오도록 구현하였다.  스크린샷%202017-11-28%20오후%203.50.03.png  4. thread\_get\_priority, thread\_set\_priority  thread\_get\_priority에서는 현재 thread의 priority를 받고, therad\_set\_priority에서는 현재 thread의 priority를 새로 setting하도록 구현하였다. 여기서 thread\_set\_priority는 priority를 새로 set후에, 만일 이 thread가 ready list의 top priority보다 우선순위가 낮으면 thread\_yield를 통해 리스케줄링 해주도록 구현하였다.  스크린샷%202017-11-28%20오후%203.47.46.png |

3. semaphore에서의 priority에 따른 unblock

|  |
| --- |
| priority scheduling을 구현하였기 때문에 semaphore에서 잠든 thread들을 unblock할때 역시 priority에 맞춰 깨워줘야 한다. 이를 위해 semaphore의 sema\_up함수에서 unblock을 할때 linear search를 통해 가장 우선순위가 높은 thread를 우선적으로 깨우도록 구현하였다.  스크린샷%202017-11-28%20오후%203.52.19.png |

4. thread\_aging

|  |
| --- |
| priority scheduling방식에서는 우선순위가 낮은 thread에게 starvation이 야기될 수 있는 위험이 있다. 이를 방지하기 위해 thead\_aging 함수를 통해 aging을 구현하여 running된지 오래된 thread는 초단위로 priority가 1씩 증가하도록 구현하였다. 이를 위해 thread\_aging을 timer interrupt함수내에서 second단위로 호출하고, 내부에선 ready queue를 전수조사하여, thread가 1초이상 ready 상태일때마다 priority가 1씩 증가하도록 하였다. 여기서 중요한 것은 thread priority에 변화가 생겼으므로, 이 priority가 running thread보다 높으면 리스케줄해야 한다는 것이다. 그런데 thready\_yield함수는 interrupt context에서는 호출될 수 없다. 따라서 interrupt가 종료될때 thread\_yield()함수를 호출하는 intr\_yield\_return함수를 호출하였다.  스크린샷%202017-11-28%20오후%203.58.58.png |

5. BSD scheduling

|  |
| --- |
| BSD scheduling은 이번 프로젝트의 추가구현부로, response time을 보장하기위해 최근에 cpu를 점유한 thread의 priority를 떨어뜨리는 방식이 스케쥴링 기법이다. 해당 함수에서는 복잡한 연산이 발생하는데 우선순위에 user가 영향을 줄 수 있는 nice, 최근 cpu 점유시간을 의미하는 recent\_cpu, ready queue의 평균갯수를 의미하는 load\_avg를 moving average방식으로 timer 인터럽트가 호출될때 주기적으로 다시 계산하게 된다. 이를 위해 timer interrupt 핸들러 내에서 BSDschedule()을 호출하여 내부에서 priority를 위한 연산을 수행하도록 구현하였다.  1. BSD\_schedule()  BSD schedule에서는 주기적으로 priority연산을 위한 연산을 수행한다. 우선 running thread의 recent\_cpu는 매tick마다 증가하고, 1초마다 load\_average 와 recent\_cpu를 다시 계산한다. 또, 4tick마다 priority를 재계산한다.  스크린샷%202017-11-28%20오후%204.11.01.png  2. BSD\_calc\_load\_average  해당 함수에서는 load average를 1초마다 재계산한다. 여기서 소수점 연산이 수행되므로 fixed\_point를 직접 오브젝트로 구현하여 이를통해 다음 연산을 수행하였다.  ***load\_avg* = (59/60) \* *load\_avg* + (1/60) \* *ready\_threads***  스크린샷%202017-11-28%20오후%204.14.30.png  3. BSD\_calc\_recent\_cpu  해당 함수에서는 1초마다 모든 thread의 recent cpu를 재계산한다. 이 역시 소수점 연산이므로 fixed point 연산을 통해 처리하였다.  ***recent\_cpu* = (2 \* load\_avg) / (2 \* load\_avg + 1 ) \* *recent\_cpu* + nice**  스크린샷%202017-11-28%20오후%204.16.14.png  4. BSD\_calc\_priority  4tick마다 호출되는 해당 함수는 priority를 다음과 같은 공식으로 재계산한다. 여기서 priority범위를 벗어나지 못하도록 제한하였다.  ***priority* = PRI\_MAX – (*recent\_cpu* / 4) - (*nice* \* 2)**  스크린샷%202017-11-28%20오후%204.18.25.png  또한 해당 thread가 ready 상태이면, ready queue내에서 재배치를 해줘야한다. 여기서 잘못된 설정으로 2일동안 mlfqs-nice-10 테스트케이스를 통과하지 못하여 애를 먹었다. 테스트케이스는 다음과 같이 미세한 오차가 발생하였다.  KakaoTalk_Photo_2017-11-27-21-23-52.png  이틀만에 찾아낸 에러의 원인은, 이 함수 내에서 모든 리스트를 전수조사하면서 다음과 같이 구현했기 때문이다.  스크린샷%202017-11-28%20오후%204.16.14.png  이 구현의 경우 ready queue내의 모든 thead를 다시 push back하기 때문에, priority가 변하지 않았더라도 새로 삽입되게 되고, 이는 같은 priority내에서의 round robin스케줄링 정책에 위배된다. 따라서 다음과 같이 priority queue가 변한 경우에만 새로 삽입하도록 구현하니 해결되었다.  스크린샷%202017-11-28%20오후%204.31.50.png  이를 통해 오차범위 10 이내로 거의 모범답안과 유사하게 테스트를 통과할 수 있었다.  마지막으로 priority를 계산후엔 필요에 따라 yield함수를 호출하여 리스케줄링이 가능하도록 구현하였다.  5. thread\_get\_nice, thread\_set\_nice  nice는 priority를 조정하기 위한 변수로, thread\_get\_nice, thread\_set\_nice를 통해 해당 value를 조정한다. 특히 thead\_set\_nice는 nice를 할당하고, priority를 재계산해야 한다. 또, priority 를 재계산하였으므로, 필요에 따라 리스케줄링을 위해 thread\_yield() 함수를 호출하도록 하엿다.  스크린샷%202017-11-28%20오후%204.40.09.png |

6. Fixed point

|  |
| --- |
| pintos 에서는 소수점연산을 지원하지 않는다. 따라서 fixed point연산을 software적으로 구현하여 소수점 연산을 수행하였다.  1. fixed point구조  우선 데이터타입을 차별화하기 위하여 fp\_t 타입을 선언하였다.  또, linux에서 사용하는 object 형태로 fixed point structure을 구성해보고자 하였다. 따라서 함수 포인트를 통해 메소드를 묶어 encapsulation하게 연산을 수행하도록 하였다.  스크린샷%202017-11-28%20오후%204.46.03.png스크린샷%202017-11-28%20오후%204.44.11.png  2. fixed point 연산  fixed point연산은 핀토스에 설명된 듯이 convert, add, subtrate, multiply, devide에 대해 구현하고, 추가적으로 디버깅을 위해 fp\_t 타입을 print하는 함수를 추가하였다.  스크린샷%202017-11-28%20오후%204.48.00.png |

3. 시험 및 평가 내용:

-평가방법

1. 23개의 테스트 케이스 수행하기 (make clean 상태일때)

1. $cd pintos/src/thread
2. $make
3. $cd build
4. $make check
5. 22개의 테스트케이스가 통과하였는지 확인한다.
6. run -v -- -q run priority-lifo를 통해 적절한 출력인지 확인한다. (뒤에 내용 참조)

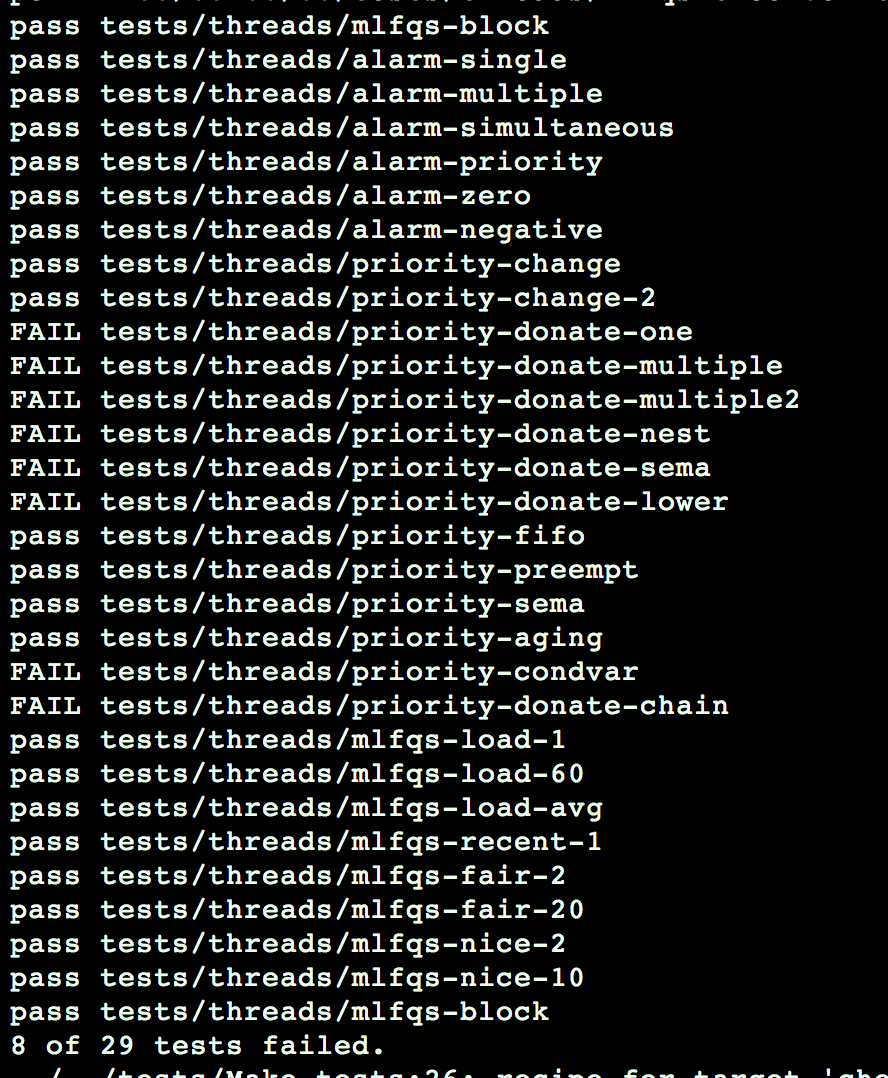
2. Userprog와 통합적 수행이 가능한지 확인을 위한 76개의 테스트케이스 수행

1. $cd pintos/src/userprog
2. $make
3. $cd build
4. $make check
5. 76개의 테스트케이스가 통과하였는지 확인한다.

-보건 및 안정, 생산성과 내구성

이번 프로젝트를 수행하면서는 thread를 구현하고, 이를 기존의 Userprog에 효과적으로 통합하는것이 보건성, 안정성, 내구성, 생산성 측면에서 가장 중요하다고 생각하였다. 이를 위해 thread의 모든 test case뿐 아니라 User prog의 기존 76개 테스트케이스도 문제없이 통과할 수 있도록 세심하게 주의를 기울였다. 실제로 thread를 고치면서 Userprog의 테스팅이 전부 fail되는 등 많은 문제가 발생하였으며, 종반부에는 syn-read, multi-recurese에서의 fail문제가 발생하였다. 이는 thread코드를 고치는 도중에, User prog에 대한 예외처리를 제대로 고려하지 못했기 때문에 발생했다. 따라서 User prog의 테스트케이스 에러도 모두 수정함으로써 통합적인 환경의 핀토스를 구축할 수 있었다.

또, thread testing을 할때 테스트를 반복해도 문제가 생기지 않는 점을 증명하고자 하였다. 이를 위해 thread 스케줄링에 영향을 미칠 수 있도록 여러 코어 갯수를 설정하여 실험을 반복적으로 수행하였다. 각 코어 1, 4, 12, 16, 20 개의 실험에 대해 모두 pass하였으며, 이를 통해 thread 프로젝트가 어떤 코어수로 실행해도, 반복적으로 수행된다 하더라도, 문제없이 테스트 케이스를 통과함을 증명할 수 있었다.

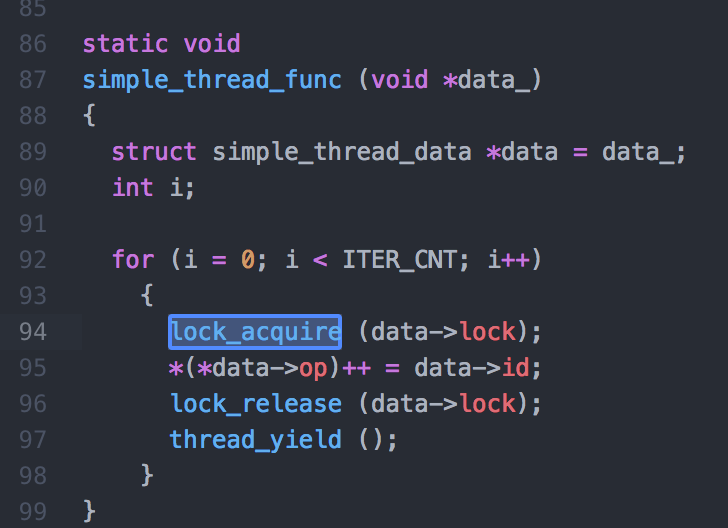




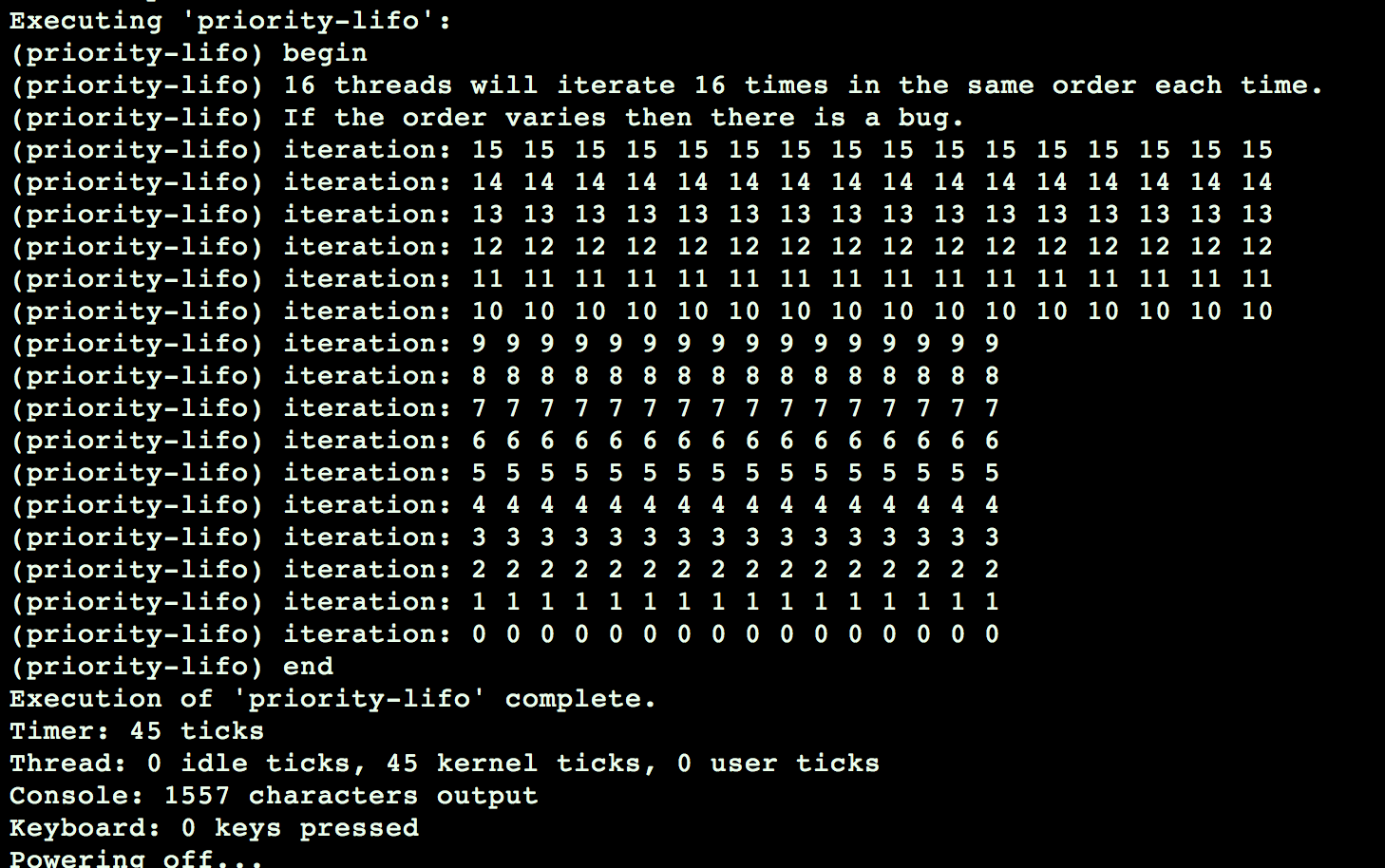
-Additional Requirement

1. Priority-lifo test 결과 분석

priority-lifo는 16개의 thread를 생성하는데 thread는 31~63까지의 priority를 각각 갖는다. 모든 thread는 최초 수행되는 simple\_thread\_func함수에서 공유하는 lock에 걸리게 된다. 따라서 semaphore의 unlock이 제대로 구현되었다면, thread는 priority가 높은 순서대로 수행될 것이다. simple\_thread\_func는 다음과 같은 형태를 가진다.



코드를 보면 최초에 가장 높은 priority를 가진 thread가 lock을 가지고 수행이된다. 이때 나머지 모든 thread는 lock에 걸린다. 그런데 lock을 release하여도 priority때문에 낮은 thread는 해당 스레드가 끝날때까지 running이 불가능하다. 따라서 priority순서대로 수행이 돌아가서 다음과 같은 결과가 나온다.



2. 테스트 분석 결과

이번 thread test는 결과물이 정확히 작동하고, 반복적으로 수행해도 문제가 생기지 않는 지의 여부를 테스트 분석하였다. 다양한 상황에서 테스팅을 하기 위해 코어 갯수를 다르게 하고 테스트를 수행하였다. 각 실험은 코어1 개, 4개, 8개, 12개, 16개, 20개로 총 6회의 테스트를 수행하였으며 결과는 다음과 같이 나왔다.







모든 테스트에 대해 all pass하였으며, 이를 통해 결과물이 정확하게 작동함을 확인할 수 있었다.

**V. 기타**

- 기타 관련 내용을 기술할 것.

1**. 연구 조원 기여도:**

안진우: 100%

2. 기타 본 설계 프로젝트를 수행하면서 느낀 점을 요약하여 기술하라.

이번 프로젝트를 수행하며, 이론적으로 배운 thread scheduling을 실제로 구현하면서 많은 것을 배울 수 있었다. 우선 thread scheduling이 어떻게 수행되어야 하는지, 이론적으로는 알수없을 만큼 깊숙히 이해가 가능했다. 또, User prog랑 통합하는 과정에서, 프로젝트 규모가 커졌을때 이를 통합시키는 과정이 얼마나 험난한지 깨달을 수 있었다. 실제로 User prog 테스트에서 수많은 문제가 발생하였고, 이를 해결하기 위해 많은 노력을 쏟아 부어야만 했기 때문이다. 또, 소수점 연산을 위해 object를 구현하는 연습을 해볼 수 있었으나, 커널 내에서 여러 제약 조건으로 인해 malloc, free를 원하는 데로 사용할 수 없었다는 점도 신기하다. 아마 이를 해결하기 위해서는 interrupt에 맞는 free함수를 생성하거나, 아니면 intr\_yield\_return과 같이 리턴하고 프로세스 컨텍스트에서 수행되도록 하는 방법을 찾아야 할 것 같다는 생각이 들었다. 마지막으로 mlfqs-nice-10 의 테스트케이스의 미세한 오차가 발생하여, 이를 수정하는데 모든 코드를 뒤집어 보는 등 2일의 노력을 쏟아야 했는데, 이 과정 자체가 하나의 성장이 될 것 같다.