**Pintos Project 3: Threads**

담당 교수 : 김영재

조 / 조원 : 20190388 백상욱

개발 기간 :2023/10/30 ~ 2023/11/19

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

기본적으로 핀토스에서는 프로세스들이 여러가지 돌아가고 있다. 물론 핀토스에서는 프로세스가 아니라 스레드가 여러개 돌고있는 것이지만 이는 프로세스와 동일한다. 그렇기 때문에 이번 보고서에서는 프로세스가 아닌 스레드라고 작성하겠다. 하지만 핀토스에서 실행할 수 있는 프로세스는 하나이기 때문에 실제로 돌아가고 있는 프로세스, 즉 cpu를 점유하며 running state에 있는 프로세스는 하나이다. 그렇다면 나머지 프로세스를 어떻게 관리할 것인가에 대해서 이번 프로젝트에서 구체적으로 구현한다. 현재 핀토스는 프로세스의 state를 running, ready로 나눠서 관리한다. 따로 blocked인 상태가 없다. 그래서 특정 스레드를 잠시 재운 뒤에 다시 깨우는 이러한 alarm 시스템콜을 구현함에 있어 busy waiting을 사용하고 있는데, 이는 비효율적이기 때문에 blocked list를 추가하여 alarm 시스템콜의 수행을 효율적으로 만들어준다. 추가로 현재 핀토스에서는 ready list에 있는 스레드들을 스케줄링할 때 가장 단순한 round robin 방식을 채택한다. 하지만 스레드들마다 우선순위가 존재할 수 있고 더 나아가 이러한 우선순위를 바꿔주거나 다양한 큐에서 관리해줌으로서 스케줄링을 더 효율적이고 효과적으로 만들 수 있다. 이번 프로젝트에서는 이를 위해 alarm, priority scheduler, advanced scheduler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock  
     기본적으로 pintos에서는 alarm system call이 구현되어있다. alarm시스템콜은 호출한 프로세스를 특정 시간동안 재운 후에 다시 실행시키는 시스템 콜이다. 이는 스레드를 아예 스케줄링의 대상에서 벗어나도록 하는것과 동일하다고 볼 수 있다. 기존의 pintos는 이를 busy waiting방식을 이용해서 구현한다. busy waiting 방식을 사용하면 알람을 통해 재운 스레드가 여전히 ready list에 존재하고 스케줄링의 대상이 될 수 있다. 그리고 만약 스케줄링의 대상이 되었다면 alarm으로 지정받은 길이와 틱을 비교해서 아직 일어날 때가 되지 않았다면 thread\_yield로 점유권을 포기한다. 결국 스케줄의 대상이 되고 cpu를 낭비하게 되기 때문에 이는 비효율적인 과정이다. 이를 위해서 아예 스레드가 스케줄링의 대상이 되지 않도록 재우고 일정 시간이 지난후에 이를 다시 스케줄링의 대상이 되도록하는 sleep, wake up을 구현해준다. 이렇게 되면 아예 cpu를 점유하지 않을 수 있기 때문에 보다 효율적으로 alarm을 구현할 수 있다.
  2. Priority Scheduling  
     현재 스케줄러는 priority를 따로 지어주지 않고 round robin 방식만으로 작동한다. 이는 fairness 측면에서는 좋을 수 있지만 현실의 프로그램에서는 더 높은 우선순위를 가져서 더 많이 스케줄 되어야하는 프로세스가 존재하고 이를 고려해서 프로세스를 스케줄링하는 것이 전반적인 퍼포먼스의 증가로 이어질 수 있다.이를 위해서 round robin으로 스케줄을 하더라도 Priority를 고려하여 스케줄링을 할 수 있게 구현해준다.
  3. Advanced Scheduler (추가구현을 한 경우)  
     priority scheduler의 경우 priority가 굉장히 단순하게 구현된다. 고정되어있거나 set priority를 이용하여 일부를 수정해주는 경우 혹은 aging을 구현해서 priority가 조금씩 증가하여 starvation을 막아주는 정도가 알고리즘의 전부이다. 하지만 이는 priority를 보다 잘 구현했다고 보기 어렵다. 반면 advanced scheduler는 cpu 점유 정도, 현재 기다리고 있는 또는 돌아가고 있는 스레드들의 개수에 따라 priority가 유동적으로 변화하기 때문에 이에 맞춰서 response time, fairness등을 더 높이면서 스레드들을 스케줄링할 수 있다.
  4. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.  
   이를 구현하기 위해서는 우선 blocked 상태의 스레드를 따로 관리할 수 있는 blocked list를 만든다. 스레드 구조체에 wakeup이라는 추가 변수를 선언한다. 이는 alarm이 호출되었을 때, 몇틱이 지난 후에 일어나야하는지에 대해서 저장해주는 변수이다. 그래서 alarm이걸린다면 thread를 재우기 위해서 alarm에서 명시된 tick을 thread의 wakeup에 저장하고 이 스레드를 blocked list에 집어넣어준다. 이렇게 함으로서 block된 스레드들을 하나의 리스트에서 관리한다. 이후에 timer interrupt가 걸릴 때마다 tick 증가하게 되는데, 이 tick을 가지고 blocked list를 순환한다. 순환하며 tick과 스레드에게 걸려있는 wake up을 비교하여 wakeup이 더 낮다면 스레드를 깨워도되는 상태가 되었기 때문에 blocked list에서 제거하고 thread\_yield를 호출해줌으로서 스레드를 깨워줄 수 있다.
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.  
   현재 running thread보다 높은 priority를 가진 thread가 온다면 해당 스레드가 다시 cpu를 가질 수 있게 만들어줘야한다. 그것이 pintos 메뉴얼에 적혀있는 priority schedule의 내용이다.
3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)  
   priority의 계산에서 필요한 요소는 recent cpu와 nice정도이다. 이 둘은 각 스레드가 가지고 있는 값이다. recent cpu는 해당 스레드가 최근에 얼마나 많은 cpu를 사용했냐에 대한 값이고 이 값이 낮을 수록 높은 priority를 가진다. 그리고 nice 값은 -20부터 20의 값으로 이또한 낮은 값을 가질수록 높은 priority를 가진다. 이러한 recent cpu와 nice는 모두 0으로 우선 초기화가 된다. 그리고 만약 부모로부터 만들어진 스레드라면 부모의 값을 가져오게 된다. nice는 따로 계산식이 존재하지않는다. 반면 recent cpu는 계산을 해줘야한다. 그리고 이 계산을 위해서는 load\_avg값과 nice를 이용해서 계산해야한다. load\_avg값은 초기 값은 0이고 평균적으로 어느정도의 스레드가 Ready되어있는지, 스케줄 될 것인지에 대한 값으로 특정 스레드에게 있는 값이 아닌 모든 스레드들이 공유하는 값이다. 이 값은 load\_avg값 그자체와 ready\_threads라는 값에 의해서 계산된다. read\_threads는 ready상태에 있거나 현태 돌아가고 있는 스레드의 개수를 나타내는 값이다. 이러한 값들을 계산하면 결과적으로 priority를 계산할 수 있다. 이때 주의할 점은 priority, nice는 int 값을 가지게 되며 load\_avg, recent cpu는 소수 값을 가진다는 것이다. 하지만 핀토스에서는 소수점 연산을 지원하지 않기 때문에 비트 연산을 하는 것 처럼 정수형으로 저장하고 이를 이용해서 고정 소수점 연산을 수행해야한다. 그리고 이를 위해서 각 연산식을 구현해야한다. 이번 프로젝트 중에서는 정수를 고정 소수로 바꾸는 연산, 고정 소수를 정수로 바꾸는 연산, 소수 더하기 소수 연산, 소수 빼기 소수 연산, 소수 곱하기 소수 연산, 소수 나누기 소수 연산, 소수 곱하기 정수 연산을 구현한다.
4. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
* 10/30 ~ 11/5 : alarm 구현  
  11/5 ~ 11/10 : priority scheduler 구현  
  11/10 ~ 11/17 : advanced scheduler 구현  
  11/17 ~ 11/19 : 보고서 작성
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
* 우선 blocked 상태의 스레드를 깨우기 위해서는 blocked list를 만들어야한다. 이를 위해서 thread.c에서 static 변수로 list형 blocked\_list를 선언한다. 이를 글로벌로 선언하는 이유는 blocked list를 관리하는 것은 전체 스레드들에 대해서 관리를 해야하기 위함이다. 그리고 thread\_init에서 blocked list를 초기화해준다. 그리고 thread\_sleep이라는 함수를 구현한다. 이 함수는 timer\_sleep이 걸렸을 때, 호출되는 함수로 sleep이 걸렸을 때 돌아가고 있는 스레드를 주어진 tick만큼 재워주는 역할을 하게 된다. 그리고 timer\_sleep에서 busy waiting으로 돌아가고 있던 것을 이 thread\_sleep 함수를 호출하는 것으로 대체한다. 그리고 tick만큼 재워주기 위해서는 이를 저장할 수 있는 변수가 필요하다 그렇기 때문에 thread 구조체에 wakeup이라는 변수를 저장한다. 그리고 만약 tick과 wakeup을 비교해서 깨워야하는 상황이라면 이를 깨워줘야하는데 이를 위해서 thread\_wakeup이라는 함수를 추가로 구현한다. 이는 thread를 깨워주는 역할을 한다.이 함수는 매 틱마다 timer\_interrupt가 걸리는데 이 안에서 호출한다. 구체적인 기능은 blocked\_list를 순회하며 만약 tick보다 값이 작아져 깨워야하는 thread가 있다면 이를 Unblock하고 list에서 제거함으로써 리스트를 다시 깨워주는 역할을 한다.
* 다음으로 priority scheduler를 구현하기 위해서는 우선 priority compare함수를 구현해야한다. 이는 비교하고자 하는 스레드를 두개 입력으로 받아 두개의 priority를 비교하여 더 어느쪽이 더 높은지를 반환하는 함수인데, 이 함수를 구현해야하는 이유는 이 함수를 사용하여 이 함수를 기준으로 ready\_list를 정렬해야하기 때문이다. ready\_list를 정렬하는 순간은 thead\_unblock, thread\_yield를 하는 순간이다. 이때마다 새로운 리스트가 ready\_list로 들어가게 되는데 이때마다 list\_insert\_ordered함수를 사용하고 함수의 파라미터로 위에서 선언한 prority\_compare함수를 인자로 넣어서 이를 기반으로 list를 정렬해주면 ready\_list가 늘 priority를 기반으로 정렬되어있게된다. set\_priority라는 함수가 있는데 이는 priority를 새로 지정하게 된다. 근데 만약 지정된 priority가 ready list에 있는 priority값보다 낮은 경우 더 높은 priority에게 cpu를 줘야하기 때문에 set\_priority안에서 Ready list의 priority와 비교해서 더 높은게 있다면 thread\_yield를 해준다. 추가로 thread\_create를 해서 새롤운. thread를 만들었을 때 thread를 생성 한 후 마지막에 만들때 사용한 priority와 현재 스레드의 priority를 비교하여 만약 새로 만든 thread의 priority가 높다면 thread\_yield를 해서 현재 스레드를 레디 리스트로 넣어서 더 높은 priority가 preemtive하게 작동할 수 있도록 구현한다.
* advanced scheduler를 구현하려면 우선 소수 연산을 구현해줘야한다. 이를 위해서 앞서 얘기했던 함수들 convert\_fixed\_point, conver\_to\_integer\_zero, convert\_to\_integer\_nearset, add\_float\_float, sub\_floag\_float, mul\_float\_float, mul\_float\_int, devide\_float\_float, device\_float\_int 를 구현한다.그리고 이 연산을 함에 있어서 정수부와 소수부를 나눠주는 FRACTION을 전역 변수로 선언해줘야한다. 그리고 load\_avg값은 전역변수로 선언하고 nice, recent\_cpu는 thread 구조체의 변수로 선언한다. 그리고 thread\_tick안에서 매 틱마다 연산해줘야하는 값들을 수식에 맞춰서 연산하는 내용을 추가해준다. 구체적인 연산 내용은 이후에 기록한다. 그리고 추가로 recent\_cpu, nice등에 대한 값을 꺼내오는 또는 수정하는 thread\_get\_nice, thread\_get\_recent\_cpu, thread\_get\_load\_avg등을 구현해줘야한다.
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)  
  텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

  자동 생성된 설명  
    
  스크린샷, 텍스트, 영수증, 라인이(가) 표시된 사진

  자동 생성된 설명
  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
* alarm을 구현하기 위해서는 우선 blocked list를 선언해줘야한다. 이 리스트는 thread마다 가지고 있는 것이 아니기 때문에 전역 변수로 선언하고 thread\_init에서 미리 list\_init으로 초기화해준다. 그리고 각 thread에게 wakeup이라는 값을 줘야한다. 이값을 기반으로 언제 일어날지를 저장할 수 있기 때문이다. 이후에 timer\_sleep에서 busy waiting으로 구현되고 있던 함수를 제거하고 thread\_sleep을 호출해준다. 여기서 중요한 것은 우선 다른 인터럽트가 발생하지 않도록 Intr\_disable을 걸어준다. 그리고 이전 상태를 old\_level로 저장한다. 그리고 현재 스레드가 idle\_thread가 아닌 경우에 대해서 현재 element를 blocked\_list에 추가해준다. 그리고 wakeup변수를 인자로 받은 tick만큼으로 할당하고 block상태로 바꾼다. 이후에 다시 intr\_set\_level로 인터럽트를 받을 수 있게 해준다. 이제 재워줄 수 있게 되었다. 이후에는 깨워줘야하는데, 깨워주는 것은 매 timer interrupt마다 불러줘야한다. thead\_wakeup이라는 함수가 매 tick마다 호출된다. 이 함수에서는 모든 blocked list를 순회하며 인자로 들어온 tick보다 wakeup이 작거나 같으면 list에서 remove를 해주고 unblock을 호출해야한다. 이때 리스트를 순회하면서 오류가 계속 발생했는데, 이는 list를 remove하면 자연스럽게 원소가 없어지기 때문에 list\_next로 다음 원소를 옮겨줄 필요가 없고 깨워주지 않은 경우에 대해서만 list\_next로 넘어가야하는데 이를 몰랐기 때문이다. 그래서 이를 반영하여 반복문을 짜서 모든 리스트에 대해서 체크하여 깨워준다. 하지만 위와 같이 구현했을 때 한가지 문제가 존재했는데, tick이 언제부터 시작했는지 언제부터 시작해서 그뒤로 얼마나의 tick이 지났는지를명시해줘야했다. 이를 위해서 timer\_sleep 안에서 그당시 tick을 start로 지정하여 thread\_Sleep에 start와 tick을 모두 전해준다. 그리고 각각을 thread 구조체에서 sleep\_start와 wakeup으로 저장한다. 그리고 wakeup에서 start+tick을 wakeup과 비교함으로서 alarm을 구현했다.
* priority scheduler를 구현함에 있어 가장 중요한 부분은 Priority\_compare함수를 만드는 부분이다. 두개의 Thread를 인자로 받고 그 스레드들의 priority를 비교하여 앞 thread가 더 높은 우선순위를 가진다면 true를 반환하고 아니라면. false를 반환하도록 구현한다.  
  그리고 thread\_unblock 에서 list를 unblock하여 ready\_list로 넣을 때 그냥 삽입하는 것이 아닌 list\_insert\_ordered함수를 사용한다. 이함수의 인자로 priority\_compare를 넣어주면 priority를 비교하고 이를 기반으로 ready\_list 자체를 정렬해주는 함수이다. 이를 활용하여 ready\_list를 priority를 기반으로 정렬하여 사용할 수 있다. 또한 thread\_yield에서 thread를 ready list로 다시 돌려줄 때도 마찬가지로 그냥 넣는 것이 아닌 list \_insert\_ordered를 사용하여 넣어준다. 이때 현재 스레드가 idle\_thread가 아닌지 확인해야한다. 그리고 idle\_thread가 아닌 경우에 대해서만 이를 ready\_list에 집어넣는다. 마지막으로 thread\_Create에서 스레드를 새롭게 생성한 후 새로운 priority와 현재 thread의 priority(thread\_current()->priority)와 비교하여 만약 새로운 priority가 높다면 현재 thread\_yield를 호출하여 현재 스레드의 cpu 점유를 포기한다. 추가로 set\_priority 함수에서 파라미터로 들어온 새로운 priority가 있을 때, 새롭게 설정해 준 후 만약 그 priority가 ready\_list에 있는 스레드들의 priority보다 낮다면 cpu를 포기해야하기 때문에 이를 비교한다. 이때 앞서. ready\_list를 priority를 기반으로 정렬했기 때문에 ready\_list의 맨앞에 있는 원소가 가장 높은 priority를 가지게 된다. 따라서 list\_front(&ready\_list)를 하고 그 elem의 list\_entry로 thread를 얻어 해당 priority와 비교하여 더 낮으면 thread\_yield를 선언하여 cpu를 포기한다. 추가로 test를 모두 통과하기 위해서는 priority aging을 구현해야한다. 이는 매 틱마다 priority를 조금씩 증가시키는 것으로 간단하게 구현할 수 있다. 이를 구현하기 위해서는 매틱마다 불리는 thread\_tick을 수정하여 이를 구현한다. priority aging은 thread\_priority\_aging이라는 변수를 기반으로 호출이 될지 안될지가 결정되기 때문에 thread\_priority\_aging이 true인 경우에만 thread\_aging을 호출한다. thread\_aging안에서는 모든 thread들에 대해서 priority를 1씩 증가시켜준다. 이를 위해서 all\_list를 순회한다. 그리고 각 list에서 thread를 꺼낸 후, priority를 1만큼 증가시킨다. 그리고 이 증가한 priority가 PRI\_MAX보다 큰 경우는 PRI\_MAX로 조정해준다. 그리고 PRI\_MIN보다 작은 경우는 PRI\_MIN으로 조정해준다. 이때 list\_entry를 사용하여 thread를 꺼내올 때 마지막 변수로 allelem을 넣어줘야한다. all\_list를 대상으로 순회할 때는 allelem을 넣어줘야하기 때문이다.
* bsd 스케줄러를 구현할 때 앞서 말했듯 소수점 연산을 우선적으로 구현해줘야한다. 그리고 소수점 연산에 필요한 FRACTION(1<<14)을 글로벌 변수로 선언한다.  
  convert\_fixed\_point는 정수를 인자로 받아서 소수형 연산이 가능하도록 소수형으로 전환해주는 함수를 의마한다. 이를 위해서는 인자로 받은 정수를 FRACTION을 곱해준다.   
  conver\_to\_integer\_zero는 소수를 인자로 받아서 이를 내림하여 정수로 반환한다. 이를 위해서 단순하게 FRACTION으로 나눠준다. conver\_to\_integer\_nearest는 소수를 정수로 바꿔줄 때 반올림을 해주는 역할을 한다. 이때 인자로 받은 정수가 양수인지 음수인지에 따라서 나눠서 구현한다. 0보다 크거나 같다면 FRACTION을 더하고 2로 나눠준 뒤, 다시 FRACTION으로 나눠준다. 그리고 만약 0보다 작거나 같다면 FRACTION을 빼고 2로 나눠준 뒤 다시 FRACTION으로 나눠준다. add\_float\_float과 sub\_float\_float은 float들을 더하고 빼주는 연산인데 이는 별다른 조치없이 단순하게 둘을 더하고 빼주는 식으로 구현한다. mul\_float\_int는 float과 int를 곱하는 연산으로 단순하게 둘을 곱하는 식으로 구현한다. devide\_float\_int도 마찬가지이다. 단순하게 둘을 나누는 식으로 구현한다.  
  devide\_float\_float의 경우는 우선 overflow를 방지하기 위해서 int64\_t형 정수 tmp를 선언한다. 그리고 이 값에 중간 값을 저장해서 overflow를 방지한다. 이 tmp 값에 첫 번째 값을 저장하고 FRACTION을 곱한다. 그리고 이 값에 대해서 y를 나눠준다. 그리고 그 값을 다시 int형으로 typecasting을 하고 반환한다. mul\_float\_float의 경우도 오퍼플로우를 방지하기 위해서 int64\_t tmp를 선언한다. 첫 번재 변수를 tmp에 저장하고 tmp와 y를 곱한 후 이를 FRACTION으로 나눠준다. 이 값을 int형으로 형변환하여 반환한다. 이로써 필요한 소수 연산을 모두 구현했다. 그리고 이 연산한 값을 기반으로 계산해야하는 값은 priority, recent cpu, load avg값이다. load avg는 모든 스레드들에 대해서 공통으로 존재하는 값이기 때문에 전역 변수로 선언한다. 그리고 recent\_cpu, priority의 경우는 스레드마다 가지고 있는 값이기 때문에 struct thread에서 선언한다. 추가로 nice의 경우도 struct thread에 선언을 해준다. 그리고 recent cpu, nice의 경우 0으로 초기화가 된다. 그리고 부모가 존재한다면 부모의 값을 가져와야한다. 이를 위해서 thread\_init에서 미리 recent cpu, nice를 0으로 만든다. 결국 모든 스레드는 이 스레드로부터 값을 상속받기 때문에 위에서 0을 선언하고 init\_thread에서 부모의 recent\_cpu, nice값을 가져오게 되면 자연스럽게 0으로 초기화되거나 부모의 값이 있다면 그 값을 상속받게 구현된다.  
  이제 값을 계산해줘야하는데 값을 계산하는 부분은 모두 thread\_tick안에서 일어난다. 앞어 본 priority aging이 thread\_prior\_aging값을 기반으로 이 값이 true일때 호출되었다면 bsd scheduler는 thread\_mlfqs가 true일 때 계산한다. 이때 계산은 어떤 연산은 매틱마다 이루어지고 어떤 연산은 4틱마다, 어떤 연산은 1초마다 이루어지는데 이를 경우를 나눠서 잘 계산해야한다. 우선 모든 틱마다 해줘야하는 것은 현재 돌아가고 있는 thread의 current cpu를 1씩 증가시켜주는 것이다. 그리고 그 다음으로 우선 1초마다 load avg값을 계산하고 모든 스레드의 recent cpu값을 업데이트해줘야한다. 1초는 100틱이기 때문에 1초에 대해서 조건을 거는 것은 timer\_ticks()%100==0인 경우에 대해서 걸어준다. 이때 우선 load\_avg값을 계산한다. Load\_avg의 공식은 (59/60)load\_avg + (1/60)ready\_threads이다. 이때 ready\_threads값은 현재 돌고 있거나 ready list에 있는 스레드들의 개수이다. 이를 계산하기 위해서 ready\_list를 쭉 순회하며 개수를 구한 후 현재 스레드를 포함하여 값을 1 더해줘 ready\_threads값으로 사용한다. 이때 idle\_thread라면 이 값에 포함되어야하지 않기 때문에, 조건문을 걸어서 idle\_threads가 아닌 경우에 1을 더하여 계산한다. 이후 이 값을 가지고 load\_avg를 업데이트 해야하는데 59/60을 계산할 때는 59를 소수연산으로 바꿔주고 60으로 나눌 때 device\_float\_int를 사용하여 계산한다. 그리고 load\_Avg는 float이기 때문에 (59/60)와 load\_avg를 mul\_float\_float으로 구현한다. 이후 1/60은 59/60과 동일한 방식으로 계산한다. ready\_threads는 정수형이기 때문에 소수와 연산을 해주기 위해서 소수로 바꿔준다. 그리고 mul\_float\_float으로 계산한다. 이후에 add\_float\_float으로 두 값을 더해준다. 이 값이 load\_avg가 된다. 코드는 다음과 같다  
  텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

  자동 생성된 설명  
  이후 모든 thread에 대해서 recent\_cpu값 또한 update해준다. 결국 이 recent\_cpu값이 load\_avg값에 dependent하기 때문에 먼저 위를 계산하고 다음 recent cpu를 계산한다.  
  우선 all\_list에 대해서 순회를 해준다. 그리고 모든 list에 대해서 계산할 recent\_Cpu를 업데이트 하는 계산식은 다음과 같다. (2\*load\_avg)/(2\*load\_avg+1)\*recent\_Cpu +nice이다.  
  load\_avg와 2를 곱하기 위해서 2를 실수형으로 전환한다. 그리고 둘은 곱한다. 그리고 1을 더할 때도 1을 실수형으로 전환하여 더해준다. 그리고 이 둘을 곱해준 뒤 이 값에 대해서 thread의 recent\_cpu를 곱해준다. 두 값을 모두 소수이므로 따로 형변환을 하지 않고 곱해준다. 그리고 마지막에 thread의 nice를 더해야하는데, nice는 정수형이기 때문에 소수로 형을 바꾸고 add\_float\_float으로 더해준다.   
  텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

  자동 생성된 설명  
  코드는 위와 같다. 이후에는 4틱 마다 priority를 계산해줘야한다. 이는 timer\_ticks%4==0을 조건으로 줘서 계산한다. priority를 계산하는 것 또한 모든 리스트에 대해서 해줘야하기 때문에 all\_list를 순회한다. priority를 계산하는 식은 PRIMAX - (recentcpu/4) - (nice\*2)이다. PRIMAX는 int값이기 때문에 float으로 형을 변환한다. 그리고 recent cpu를 4로 나눌 때는 4를 소수로 형변환하여 devide\_float\_float으로 계산한다. 그리고 nice와 2의 계산은 정수와 정수의 계산이므로 일단 2를 곱하여 계산해둔다. 이후에 이 값을 빼줄 때는 소수형으로 다시 변환한 뒤에 빼준다. 그리고 계산한 priority값이 PRI\_MAX보다 크거나 PRI\_MIN보다 작은 경우는 PRI\_MAX, PRI\_MIN으로 조정해준다.  
  텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

  자동 생성된 설명  
  이러면 각 틱에서 계산해줘야하는 부분은 모두 구현되었다. 이후에는 set\_nice, get\_nice, get\_recent\_cpu, get\_load\_avg등을 구현한다. thread\_set\_nice는 그 스레드에 대해서 nice값을 지정해주는 함수이다. 이 함수에서 nice를 조정했다면 이 nice를 기반으로 priority을 다시 계산할 수 있도록 해줘야한다. nice를 새롭게 바꾼 후 priority를 계산하는 식은 위와 동일하다. 이후에 중요한 것은 새롭게 priority가 바뀌었기 때문에 priority가 만약 ready list에 있는 priority보다 낮다면 thread\_yield를 호출하여 우선순위에 맞게 스케줄링을 해야한다. 이를 위해서 set\_priority에서와 동일하게 ready\_list의 맨앞에 있는 element의 thread의 priority와 비교를해준다. 그리고 작다면 thread\_yield를 바로 호출해준다. get\_nice함수는 thread\_current의 nice값을 반환한다. thread\_get\_load\_avg, thread\_get\_recent\_cpu는 각각 load\_avg, recent\_cpu를 반환하면 되는데 이때 두값은 정수기 때문에 정수로 변환해서 반환해야한다. 이를 위해 우선 load\_avg, recent\_cpu에 100을 곱해주고, convert\_to\_integer를 호출해야한다. 이때 pintos 메뉴얼에서는 convert to nearset integer, 즉 반올림을 하여 반환하도록 적혀있지만 내가 구현했을 때는 반올림을 이용하여 반환했을 때는 모두 kernel panic이 걸렸다. 아직 이유를 명확히 발견하지는 않았지만 반환할 때 버림을 하여 convert\_to\_integer\_zero를 통해 정수로 바꿔서 반환해줬다.
  1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석  
  priority-lifo 코드의 경우 몇개의 스레드들이 같은 priority를 가지게 된다. 그리고 이들이 동일한 priority라면 순서대로 round robin으로 나올 수 있는지 확인하는 테스트이다.  
  텍스트, 스크린샷, 문서, 폰트이(가) 표시된 사진

  자동 생성된 설명  
  실행 결과는 위와 같다. 보면 동일한 우선 순위를 가지고 있다면 그들에 대해서 round robin으로 잘출력되고 있는 것을 알 수 있다.
* make check 수행 결과를 캡처하여 첨부  
  텍스트, 스크린샷이(가) 표시된 사진

  자동 생성된 설명