**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

조 / 조원 : 20190345 김동현

개발 기간 : 2023.11.1 ~ 2023.11.17

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

OS Scheduler를 구현한다. 기존 busy waiting으로 인한 자원의 낭비를 막기 위해 Block 상태를 도입한 Ålarm CLock을 구현한다. 기존 round robin으로 구현된 scheduling 정책을 priority scheduling 정책을 구현하여 우선순위를 고려하도록 한다. 마지막으로, Advanced Scheduler로서 Multi-Level Feedback Queue 방식의 BSD 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  
     기존 스케줄러는 running list가 IO와 같은 CPU를 사용하지 않음에도 CPU를 점유하고 있는 상황이 발생한다. 이는 ready와 running 상태만을 가지는 busy waiting 방식으로 인한 것이다. 이로 인해 CPU 자원이 낭비되고 있었으며 성능 저하로 이어져 왔다. 이를 해결하기 위해 alarm clock을 도입하여 CPU 자원의 낭비를 막고자 한다. alarm clock은 프로세스가 sleep 상태에 들어가더라도 CPU를 점유하고 있는 것이 아닌, block list에서 대기를 하며 다른 스레드에 CPU를 양도한다. 이후 block list에 있는 스레드의 대기 시간이 끝난 경우 다시 Ready list로 이동하여 CPU 자원의 낭비를 막고, 성능을 향상시키고자 한다.
  2. Priority Scheduling  
     기존의 스케줄러는 round robin 방식으로 ready list에 들어온 순서대로 running 상태로 CPU를 점유하였다. 이러한 방식은 모든 스레드가 공정하게 CPU 점유 시간을 보장받지만, 스레드의 특성에 따라 다음과 같은 성능 이슈가 발생한다. CPU를 짧게 점유하고 block 상태에 들어가는 경우 잠깐의 CPU 점유를 위해 오랜시간 ready list에 대기하는 경우가 발생할 수 있다. 이를 해결하기 위해 각 스레드마다 할당된 우선 순위에 따라 CPU를 점유할 수 있도록 하는 priority scheduling 기법을 적용한다. 또한 우선 순위가 낮은 스레드가 계속해서 CPU를 점유하지 못하는 starvation 상태를 해결하기 위해 aging 기법을 적용한다.
  3. Advanced Scheduler (추가구현을 한 경우)  
     기존 priority scheduling은 우선순위가 고정되어 있고, aging 기법을 통해 우선순위가 변하기는 하지만, 1씩 증가만 할 뿐 감소하지 않는다. 그 결과 시간이 흐름에 따라 모든 스레드의 우선순위는 PRI\_MAX에 도달할 것이다. 또한 이는 시간의 흐름에 따른 스레드의 특성을 반영하지 못한다. 이를 해결하기 위해 BSD 스케줄러를 통해 priority, nice, recent\_cpu, load\_avg의 값을 활용하여, 특정 주기 마다 값을 업데이트하여 어떤 스레드가 CPU를 점유할지를 정한다. 그 결과 스레드의 특성에 따른 우선순위를 고려하는 정책을 유지함과 동시에 시간이 흐름에 따라 변화하는 스레드의 특성 역시 고려할 수 있게 된다.
  4. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.  
   timer\_sleep 함수가 호출될 때, 해당 스레드는 thread\_block을 호출하여 block상태로 전환한다. 이 전에 얼마동안 block 상태를 있을지를 스레드 내부에 저장을 하고, block list을 만들어 저장함으로써 block 상태의 스레드들을 관리한다. 이후 timer\_interrupt 함수가 호출될 때, block 스레드를 저장하고 있는 리스트를 순회하면서 block 상태로 있기로 한 시간이 지난 스레드는 해당 리스트에서 제거한 뒤, thread\_unblock 함수를 호출하여 스레드를 깨워 ready 상태로 돌아가도록 한다.
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.  
   현재 running 상태의 스레드보다 높은 우선순위를 가진 스레드가 ready list에 있다면 두 스레드간 context swtich를 해주어야 한다. 이러한 상황이 발생하는 경우는 thread\_create 함수를 통해 새로운 스레드가 만들어지는 경우와 thread\_set\_priority 함수를 통해 현재 실행중인 스레드의 우선순위를 변경하는 경우이다. 이러한 경우에는 현재 running 상태의 스레드의 우선순위와 ready list의 스레드 중 가장 높은 우선순위를 비교하여 ready 상태의 스레드의 우선순위가 더 높다면 thread\_yield 함수를 호출하여 현재 running 상태의 스레드가 CPU 점유를 포기하고, scheduling을 통해 다음 스레드의 실행을 결정한다.
3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)  
   priority 계산을 위해 load\_avg, recent\_cpu, niceness, priority가 필요하다. 각 요소에 대해 설명하면 다음과 같다.  
   load\_avg는 1분 간 Ready list의 평균 개수를 나타내는 실수값이다. 전역변수로 선언하여 초기 할당값은 0이고, 시스템 전체에서 하나의 값으로만 접근한다.   
   recent\_cpu는 해당 스레드의 최근 cpu 점유 정도를 나타내는 실수값이다. 각 스레드마다 해당 값을 가지고 있으며, initial\_thread는 0으로, 그 이외의 스레드는 본인을 생성한 부모 스레드의 recent\_cpu 값을 상속받는다.  
   priority는 스케줄러가 어느 스레드를 running 시킬지 결정하기 위해 비교하는 정수값이다. 높을수록 우선순위가 높고, 낮을수록 우선순위가 낮다. PRI\_MIN(0)에서 PRI\_MAX(63)의 범위를 가지며 하나의 큐를 사용하더라도 MLFQ의 원리를 적용할 수 있는 BSD 스케줄러를 구현할 수 있도록 하는 값이다. 같은 우선순위는 같은 큐에 위치한다고 생각할 수 있다.  
   niceness는 해당 스레드가 다른 스레드들에게 얼마나 cpu를 잘 yield하는지를 나타내는 정수값이다. -20에서 20의 범위를 가진다.
4. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

- 2023.11.01 - 2023.11.03 : pintos manual 및 ppt 확인

- 2023.11.04 - 2023.11.07 : alarm 구현

- 2023.11.08 - 2023.11.11 : priority scheduling 구현

- 2023.11.12 - 2023.11.15 : advanced scheduler 구현

- 2023.11.16 - 2023.11.18 : 디버깅 및 보고서 작성

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드  
    1. Alarm Clock  
    - devices/timer.c timer\_init에서 list\_init 함수를 호출하여 block 상태의 스레드를 저장할 연결리스트를 초기화한다.  
    - devices/timer.c timer\_sleep에서 기존 busy waiting 구현 대신 인자로 들어온 tick만큼 현재 스레드의 block될 tick을 저장해 깨어날 시간을 저장해준 뒤, block list에 넣어준다. 이후 해당 thread를 block해준다.  
    - devices/timer.c timer\_interrupt에서 block 리스트를 순회하면서 스레드에 저장된 깨어날 시간을 확인한다. 만약 깨어날 시간이 지났다면 해당 스레드를 block\_list에서 제거하고 unblock해준다.  
    2. Priority Scheduling  
    - threads/thread.c thread\_init에서 initial\_thread의 niceness 및 recent\_cpu값과 전역변수 load\_avg을 default값으로 초기화해준다.  
    - threads/thread.c thread\_tick에서 thread\_prior\_aging이 참일 때 thread\_aging함수를 호출하여 우선순위를 aging할 수 있도록 한다.  
    - threads/thread.c thread\_create에서 스레드가 만들어지고 난 뒤, 현재 실행되는 스레드와 우선순위를 비교하여 더 높은 우선순위라면 thread\_yield를 호출하여 현재 실행되는 스레드가 CPU를 양도하도록 한다.  
    - threads/thread.c thread\_unblock에서 unblock된 스레드를 Ready list에 정렬된 상태로 삽입되도록 한다.  
    - threads/thread.c thread\_yield에서 CPU를 양도하는 스레드가 idle 스레드가 아니라면 다시 ready list로 정렬된 상태로 삽입되도록 한다.  
    - threads/thread.c thread\_set\_priority에서 현재 스레드의 우선순위를 인자로 들어온 우선순위로 업데이트 하고, ready\_list를 순회하면서 더 높은 우선순위를 가진 스레드가 있을 경우 thread\_yield를 호출하도록 한다.  
    - threads/synch.c sema\_up에서 세마포어를 기다리는 리스트를 정렬해준다. 이후 리스트의 가장 앞에 있는 스레드가 우선순위가 가장 높은 스레드이므로 해당 스레드에게 세마포어를 획득할 수 있도록 한다.  
    - threads/synch.c sema\_down에서 세마포어를 기다리는 스레드를 저장하는 리스트에 넣는 방식을 우선순위를 기준으로 내림차순으로 삽입하도록 한다.  
      
    3. Advanced Scheduling  
    - threads/thread.c thread\_tick에서 thread\_mlfqs가 참일 때 load\_avg, priority, recent\_cpu의 값을 업데이트 한다. idle\_thread가 아니라면 현재 스레드의 recent\_cpu 값을 1증가시키고, timer\_tick이 4의 배수라면 load\_avg를 업데이트하고, 모든 스레드의 recent\_cpu를 업데이트 한다. timer\_tick이 100의 배수, 즉 1초가 지났다면 모드 스레드의 우선순위를 업데이트 한다.  
    - threads/thread.c thread\_set\_nice에서 현재 스레드의 niceness값을 인자로 들어온 niceness값으로 변경한다. 이때 niceness의 범위를 확인하고 업데이트 해야한다. 이후 현재 스레드의 우선순위를 업데이트 해주고, ready\_list를 순회하면서 우선순위가 더 높을 스레드가 있을 경우 thread\_yield를 호출한다.  
    - threads/thread.c thread\_get\_nice에서 현재 스레드의 niceness값을 리턴하도록 한다.  
    - threads/thread.c thread\_get\_load\_avg에서 전역변수 load\_avg값에 100을 곱해 리턴하도록 한다.  
    - threads/thread.c thread\_get\_recent\_cpu에서 현재 실행되는 스레드에 저장된 recent\_cpu값에 100을 곱해 리턴한다.  
    - threads/thread.c init\_thread에서 niceness, recent\_cpu가 부모 스레드의 값을 상속받아 할당하도록 한다.
  + 수정하거나 추가해야 하는 자료구조  
    1. Alarm Clock  
    - devices/timer.c에서 block 상태인 스레드를 저장할 수 있는 연결리스트 자료구조를 추가한다.  
    - threads/thread.h에서 struct thread에 해당 스레드가 어느 tick까지 block상태인지를 저장하는 정수를 추가한다.  
    2. Priority Scheduling  
    - threads/thread.c에서 aging 사용 여부를 결정할 boolean형 전역 변수를 추가한다.  
    3. Advanced Scheduling  
    - threads/thread.c에서 load\_avg를 저장할 전역 변수를 정수형으로 추가한다.  
    - threads/thread.h에서 thread 구조체에 각각의 스레드들이 가지고 있어야 할 niceenss, recent\_cpu 값을 추가한다.  
    - threads/thread.h에서 niceness, recent\_cpu, load\_avg의 디폴트 값을 선언한다. 또한 실수 연산이 어떤 연산인지를 나타낼 메크로 변수를 선언한다.
  + 수정하거나 추가해야 하는 함수  
    1. Alarm Clock  
    - 없음  
    2. Priority Scheduling  
    - threads/thread.c에서 ready list를 순회하며 모든 스레드의 우선순위를 1씩 증가시키는 aging을 수행할 thread\_aging 함수를 추가한다. 이때 이미 우선순위가 PRI\_MAX라면 우선순위를 증가시키지 않는다.  
    - threads/thread.c에서 ready 리스트를 우선순위 내림차순으로 저장하기 위해 스레드 간 우선순위를 비교하는 cmp\_priority함수를 추가한다.  
    3. Advanced Scheduling  
    - threads/thread.c에서 공식에 따라 priority, recent\_cpu, load\_avg를 계산할 함수 calculate\_priority, calculate\_recent\_cpu, calculate\_load\_avg를 추가한다.   
    - 실수 연산을 위한 함수들을 추가한다. 두 입력과 계산 모드를 입력 받아 올바른 함수를 호출하여 결과를 반환하는 float\_operation함수와 정수와 실수 혹은 실수간의 덧셈, 뺄셈, 곱셈, 나눗셈을 수행하는 계산 수행 함수들을 추가한다.

1. **연구 결과**
   1. **Flow Chart**

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

**- Alarm Clock  
텍스트, 스크린샷, 도표, 디자인이(가) 표시된 사진

자동 생성된 설명  
- Priority Scheduling  
텍스트, 스크린샷, 도표, 디자인이(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  
    1. Alarm Clock  
    block 상태의 스레드를 저장하기 위한 blocked\_list를 static struct list의 전역변수 형태로 선언한다. 이후 timer\_init 함수에서 list\_init을 통해 blocked\_list를 초기화 한다. list\_init은 인자로 들어온 연결리스트를 사용할 수 있도록 초기화해주는 함수이다.   
    block 상태의 스레드가 어느 tick까지 block 상태를 유지하는지 알기위해 thread 구조체에 int64\_t 형태로 blocked\_ticks를 선언한다. 사용자 프로그램이 timer\_sleep을 호출하게 되면, 현재 tick과 인자로 들어온 sleep을 할 tick을 더해 언제까지 해당 스레드가 block 상태를 유지할지 계산하여 현재 스레드의 blocked\_ticks에 저장한다. 이후 해당 스레드를 blocked\_list에 push하여 저장하고, thread\_block을 호출하여 스레드를 block 상태로 바꾼다. thread\_current는 현재 실행중인 스레드 구조체를 반환하는 함수이며, list\_push\_back은 인자로 들어온 스레드를 인자로 들어온 리스트의 가장 마지막에 추가해주는 함수이다. thread\_block은 스레드의 상태를 block 상태로 바꾸는 함수이다.   
    timer\_interrupt는 매 tick을 알리는 함수로 tick마다 block\_list를 반복문으로 순회한다. 해당 스레드에 저장된 blocked\_ticks가 전역 변수로 선언된 현재 ticks보다 작을 경우 unblocked될 시간이 지난 경우이므로 blocked\_list에서 해당 스레드를 제거하고 thread\_unblock을 호출하여 스레드를 unblock 시켜준다. list\_begin와 list\_end은 인자로 들어온 스레드의 시작 또는 마지막 인자를 반환하는 함수이다. list\_next는 다음 인자를 반환해주는 함수이다. thread\_entry는 해당 인자에 들어있는 실제 데이터를 반환해주는 함수로 해당 구현에서는 sturct thread를 반환한다. list\_remove는 인자를 리스트에서 제거해주는 함수이다. thread\_unblock은 block 상태인 스레드를 ready\_list에 삽입해주고, ready상태로 바꿔주는 함수이다.  
    2. Priority Scheduling  
    매 tick마다 thread\_tick이 호출된다. 이때 thread\_prior\_aging 값이 true이면 thread\_aging 함수를 호출하도록 한다. thread\_aging에서는 ready\_list를 순회하면서 ready list에 저장된 스레드의 priority를 1씩 증가시켜 준다. 이때 PRI\_MAX를 넘어가서는 안되므로, PRI\_MAX보다 작은 priority에 대해서만 증가를 시켜준다. 이로 인해 CPU를 사용하고 있지 않은 스레드의 우선순위를 증가시켜 주어 우선순위가 낮아서 CPU를 계속해서 점유하지 못하는 Starvation 문제를 일부 해결해준다.   
    thread\_create를 통해 새로운 스레드가 만들어지면, 해당 스레드의 우선순위와 현재 스레드의 우선 순위를 비교한다. 이때 새롭게 만들어진 스레드의 우선순위가 더 높다면 CPU를 점유해야하므로 thread\_yield를 호출한다. 이때 현재 스레드의 우선순위는 thread\_get\_priority를 통해 얻어오며, 새롭게 만들어진 스레드의 우선순위는 thread\_create의 인자로 들어온 priority를 사용하여 비교한다. 또는 thread\_set\_priority를 통해 현재 실행중인 스레드의 우선순위를 인자로 들어온 우선순위로 업데이트 한다. 이때, 새롭게 업데이트한 우선순위가 더 작아지고, ready\_list에 비교할 스레드가 존재한다면 ready\_list를 순회하며 스레드의 우선순위를 비교한다. 이때 ready\_list의 우선순위가 더 커진다면 CPU를 선점하는 스레드가 변경되어야 한다. 이처럼 두 경우에 대해서 thread\_yield를 호출하여 CPU를 포기하고 가장 높은 우선순위를 가지고 있는 스레드가 CPU를 선점하도록 한다. thread\_yield에서는 현재 스레드를 ready\_list에 삽입한 뒤, 스케줄 함수를 호출하게 된다. 이때 ready\_list에 내림차순으로 삽입하기 위해 list\_insert\_ordered 함수를 호출한다. 이 함수는 함수 포인터로 전달된 함수의 기준에 따라 리스트에 인자를 삽입하는 함수이다. 전달된 함수 포인터는 cmp\_priority로 인자로 들어온 두 스레드의 priority를 비교하여 첫번째 인자의 우선순위가 두번째 인자의 우선순위보다 크다면 true를 그렇지 않다면 false를 반환하도록 한다.   
    세마포어를 사용하는 경우에도 스레드를 저장하는 리스트를 사용한다. 여러개의 리스트가 세마포어를 점유하기 위해 대기하고 있다면, 가장 높은 우선순위를 가진 스레드가 세마포어를 점유할 수 있도록 해야한다. sema\_down에서는 list\_insert\_ordered함수를 사용하여 함수 포인터로 전달된 함수의 정렬 순서대로 리스트에 삽입하도록 한다. 이를 위해 compare\_priority를 선언하였고, 이 함수의 기능은 이전의 cmp\_priority와 동일한 기능을 수행한다. 이를 통해 sema->waiter에 저장된 스레드들이 우선순위를 기준으로 내림차순으로 정렬되어 삽입된다. sema\_up에서도 세마포어를 점유하기 위한 스레드를 결정할 때 가장 높은 우선순위를 가지는 스레드를 선택해야 한다. 하지만 sema\_down을 할 때의 우선순위가 달라질 수 있다는 점을 확인하고 양쪽에서 우선순위를 확인해주도록 구현했다. list\_sort함수를 사용하여 sema->waiters 리스트를 우선순위 내림차순으로 정렬하도록 한다. 이후 가장 높은 우선순위를 가진 스레드가 세마포어를 점유했을 경우 현재 실행중인 스레드와의 우선순위 비교를 통해 thread\_yield를 호출할지도 결정해야 한다. 이를 위해 세마포어를 점유한 스레드가 있는지 나타내는 empty\_flag와 해당 스레드의 우선순위를 저장하는 max\_priority를 저장한다. 이후 해당 스레드를 waiters 리스트에서 제거한 뒤 unblock 시켜주어 세마포어를 점유하도록 한다. 또한 세마포어의 점유가 발생했고, 해당 스레드의 우선순위가 현재 실행중인 스레드의 우선순위보다 크다면 thread\_yield를 호출하여 현재 프로세스가 CPU 점유를 포기하도록 한다.  
    thread\_unblock에서 스레드가 ready\_list에 삽입될 때, 우선순위를 기준으로 내림차순으로 정렬되도록 삽입하기 위해서 list\_insert\_ordered를 호출한다. 이때의 정렬 기준을 나타내는 함수 포인터는 기존에 설명했던, cmp\_priority를 인자로 전달한다.  
    3. Advanced Scheduling  
    threads/thread.h에 다음과 같은 매크로를 선언하였다. 우선 niceness, recent\_cpu, load\_avg의 디폴트 값을 0으로 선언하였다. 또한 부동 소수점 연산을 위한 매크로를 각각 실수+실수(ADD\_F\_F), 실수+정수(ADD\_F\_I), 실수-실수(SUB\_F\_F), 실수-정수(SUB\_F\_I), 실수\*실수(MUL\_F\_F), 실수\*정수(MUL\_F\_I), 실수/실수(DIV\_F\_F), 실수/정수(DIV\_F\_I)와 같이 선언한다. thread 구조체 내부에 각 스레드가 가지는 niceness와 recent\_cpu를 int64\_t로 선언한다.   
    threads/thread.c에 전역변수로 load\_avg를 static int64\_t로 선언하고 0으로 초기화한다.  
    thread\_init에서 앞서 선언한 매크로로 initial\_thread의 niceness와 recent\_cpu를 초기화한다. 또한 전역 변수 load\_avg를 디폴트 값으로 초기화 한다.  
    init\_thread에서는 현재의 스레드의 niceness와 recent\_cpu를 부모 스레드의 값을 상속받아 저장한다. 이를 위해 아직 부모 스레드를 가리키는 running\_thread를 호출하여 부모 스레드에 접근하여 해당 값을 가지고 와 저장한다.  
    thread\_tick에서 thread\_mlfqs가 true일 때, 정해진 tick 주기에 맞춰 recent\_cpu, load\_avg, priority를 업데이트 해준다. 우선 현재 스레드가 idle이 아닌 경우 recent\_cpu를 1씩 증가시킨다. recent\_cpu는 실수이고, 1은 정수이므로 ADD\_F\_I 옵션의 실수 연산 함수를 호출하여 계산한다. timer\_ticks()를 통해 현재 ticks를 조사하여 1초 = 100tick를 의미하는 TIMER\_FREQ로 나눈다. 해당 연산이 0으로 나누어 떨어진다면 Load\_avg와 recent\_cpu를 업데이트 해야할 시기이므로 각각 calculate\_load\_avg와 calculate\_recent\_cpu를 호출한다. 이때 load\_avg는 전역 변수이지만 recent\_cpu는 모든 스레드에 대해 업데이트를 해주어야 하므로 반복문을 순위하여 all\_list에 저장된 스레드마다 calculate\_recent\_cpu를 호출하여 업데이트 해준다. 이때 해당 스레드가 idle이 아니어야한다는 점에 유의한다. 또한 ticks가 4tick을 의미하는 TIME\_SLICE로 나누어 떨어진다면, priority를 업데이트한다. 우선순위 역시 모든 스레드에 저장되어 있는 값이므로 반복문을 순회하여 모든 우선순위 값을 업데이트 한다.  
    thread\_set\_nice에서는 인자로 들어온 nice값으로 현재 스레드의 nice값을 업데이트 해준다. 이를 위해 인자로 들어온 값이 [-20,20]에 있는지 확인하고 그렇지 않을 경우 가장 가까운 범위 내의 값으로 바꿔준다. 이후 현재 스레드를 반환하는 thread\_current 함수를 통해 스레드에 접근하여 niceness 값을 업데이트 한다. 이후 niceness 값의 변경으로 인해 현재 스레드의 priority의 값도 업데이트를 해준다. 업데이트 결과 해당 스레드의 우선순위에 변동이 생겼으므로, ready\_list를 순회하며 우선순위를 비교한다. 만약 ready list에 있는 우선순위가 더 클 경우 thread\_yield를 호출하여 현재 스레드가 CPU를 반환하도록 한다. thread\_get\_nice는 현재 스레드의 niceness 값을 반환한다. thread\_get\_load\_avg는 전역 변수 load\_avg의 값에 100을 곱한 뒤, 정수값으로 반올림하여 반환하도록 한다. 이때 load\_avg는 실수이고 100은 정수이기 때문에 float\_operation함수에 MUL\_F\_I 옵션을 주어 계산하도록 하고, 정수로의 반올림은 convert\_float\_nearest\_in을 통해 계산한다. thread\_get\_recent\_cpu는 현재 스레드의 recent\_cpu 를 반환한다. 현재 스레드에 저장되어 있는 recent\_cpu의 값에 100을 곱한 뒤, 정수값으로 반올림하여 반환하도록 한다. 이때 recent\_cpu는 실수이고 100은 정수이기 때문에 float\_operation함수에 MUL\_F\_I 옵션을 주어 계산하도록 하고, 정수로의 반올림은 convert\_float\_nearest\_int로 반올림된 정수값을 받아온다.  
    calculate\_priority는 공식을 계산하여 인자로 들어온 스레드의 우선순위를 업데이트 해주는 함수이다. 우선 convert\_int\_to\_float 함수를 통해 PRI\_MAX를 실수로 변환한다. 이후 recent\_cpu 실수를 정수 4로 나누는 DIV\_F\_I으로 얻은 값과 정수 niceness와 정수 2를 곱한값을 SUB\_F\_F과 SUB\_F\_I로 각각 빼주어 우선순위를 구한다. 이때 우선순위는 정수값이므로 convert\_float\_down\_int 함수를 통해 정수로 내림해준다. 해당 값이 우선순위의 범위인 [PRI\_MIN, PRI\_MAX]에 있는지 확인한다. 만약 범위 밖에 있다면 가장 가까운 유효한 우선순위로 변경한다. 해당 값을 스레드의 priority에 저장한다.  
    calculate\_recent\_cpu는 공식을 계산하여 인자로 들어온 스레드의 recent\_cpu를 업데이트 해주는 함수이다. 실수 laod\_avg와 2를 MUL\_F\_I 옵션으로 연산하고, 해당 값을 인자로 들어온 스레드의 recent\_cpu 실수값과 MUL\_F\_F 옵션으로 연산한 값을 load\_avg와 2를 MUL\_F\_I로 연산한 값을 구하고 정수 1과 ADD\_F\_I로 구한 값을 DIV\_F\_F로 나누어준다. 이후 정수 niceness를 ADD\_F\_I로 연산한 값을 통해 새로운 recent\_cpu값을 구한다. 해당 값을 스레드의 recent\_cpu에 업데이트 해준다.  
    calculate\_load\_avg는 공식을 계산하여 전역 변수 load\_avg를 업데이트 해주는 함수이다. 실수 load\_avg를 정수 59으로 MUL\_F\_I로 연산한 값을 정수 60과 DIV\_F\_I로 연산해 준다. 또한 ready\_threads 정수값을 convert\_int\_to\_float 값을 통해 실수로 형변환한 뒤, 정수 60으로 DIV\_F\_I로 연산해 준다. 이때 ready\_threads는 ready list의 원소의 개수를 구하는 list\_size에 ready\_list를 인자로 주어 구한다. 또한 Running 스레드가 idle이 아니라면 1을 추가해준다. 이렇게 구한 두 연산결과를 ADD\_F\_F로 더하여 load\_avg에 할당해준다.  
    floating\_operation 함수는 두 수와 옵션 값을 인자로 받아 옵션에 맞는 연산 함수를 호출하고, 결과 값을 반환하는 함수이다. 스위치 문을 활용하여 mode를 확인하여 각각의 함수를 호출하고, 반환 값을 반환한다. 만약 잘못된 입력이 들어온다면 -1을 리턴하도록 한다.  
    convert\_int\_to\_float는 정수를 실수로 변환하는 함수로 인자로 들어온 값에 (1<<14)를 곱해 부동 소수점 형식으로 만든 값을 반환한다.  
    convert\_float\_down\_int는 인자로 들어온 실수 값을 내림한 정수 값으로 반환해주는 함수이다. (1<<14)를 나누어 부동 소수점 형식을 정수 형식으로 만들어 반환한다.  
    convert\_float\_nearest\_int는 인자로 들어온 실수 값을 반올림한 정수 값으로 반환해주는 함수이다. (1<<14)/2 값(=0.5)를 더한 뒤(뺀 뒤), 내림 연산을 수행하여 반올림된 정수 값을 구해 반환한다.  
    add\_float\_float와 sub\_float\_float는 두 실수를 더하거나 빼는 연산을 하는 함수이다. 두 수 모두 실수이기 때문에 단순 덧셈, 뺄셈으로 값을 구하여 반환한다.   
    add\_float\_int와 sub\_float\_int는 실수에 정수를 더하거나 빼는 연산을 하는 함수이다. 실수로의 타입을 통일하기 위해 정수에 (1<<14)곱하여 실수형으로 변환한 뒤, 연산을 수행하여 결과를 반환한다.   
    mul\_float\_int와 div\_float\_int는 실수와 정수를 곱하거나 나누는 연산으로 단순 연산을 한 뒤 반환한다.  
    mul\_float\_float와 div\_flaot\_float는 실수와 실수를 곱하거나 나누는 연산이다. 이때 오버플로우의 가능성이 있기 때문에 더 큰 바이트를 저장하는 int64\_t에서 값을 연산한 뒤, 다시 int로의 형변환을 한 뒤 반환한다.
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

priority scheduling의 테스트 코드 중 priority-sema에서 계속해서 Fail을 하였다. 그 이유를 살펴보았을 때, 처음에는 sema\_up만 우선순위를 기준으로 내림차순으로 sema->waiters에 스레드를 삽입하였는데, 실행 중 해당 스레드의 우선 순위가 변경되는 등의 이슈로 우선순위가 바뀔 수 있음을 깨달았다. 이를 위해서 sema\_up에서도 list\_sort를 통해 sema->waiters를 정렬시킨 뒤 어떤 스레드에 세마포어를 줄 것인지를 결정하도록 했다. 또한 해당 스레드가 현재 실행중인 스레드 보다 높은 우선순위를 가질 경우 따로 thread\_yield를 호출해야함을 깨달아야 했다.  
advanced scheduler를 구현하는 중 recent\_cpu, load\_avg는 실수이며, priority, niceness는 정수로 저장되어야 한다는 점에서 혼란이 발생하였다. 이와 같은 이유로 정수를 실수로 착각하거나, 실수를 정수로 착각하는 실수를 저질러 이를 디버깅하는 데에 많은 시간을 사용하였다. 또한 오버 플로우를 막기 위한 연산 순서가 오히려 잘못된 연산 결과로 이어졌다. recent\_cpu를 계산하는 과정에서 2\*load\_avg를 2\*load\_avg+1로 나눈 뒤, recent\_cpu를 곱하였더니 테스트 과정에서 계속 같은 값으로 유지되는 현상이 발생하였다. 이를 해결하기 위해 큰수로 만들고 나누는 방법을 선택하여 2\*load\_avg\*recent\_cpu를 2\*load\_avg+1로 나누어 해결하였다.

* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석  
  priority-lifo.c 코드를 살펴보면, test\_priority\_lifo 함수에서 16개의 스레드를 생성한다. 각 스레드는 thread\_create를 통해 생성되며 점점 높은 우선순위를 가지고 새로운 실행흐름에서 simple\_thread\_func를 실행한다. 각 스레드는 CPU를 선점하고 lock을 한 뒤, 데이터에 접근해 메인 스레드에 본인의 아이디를 남긴다. 이후 unlock하고 thread\_yield를 호출하여 스케줄링을 통해 CPU를 양도한다. 그 결과 우선순위가 높은 스레드가 계속해서 CPU를 점유하게 되고, 모든 작업이 완료된 후 스레드가 종료되고 난 이후에 다음으로 높은 스레드가 CPU를 점유할 수 있다.

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

우선순위가 높은 스레드 15부터 우선순위가 낮은 스레드 0까지 순서대로 출력된다.

* make check 수행 결과를 캡처하여 첨부

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

alarm clock test 6개, priority test 6개, mlfq test 9개, priority-lifo 총 22개가 통과한 것을 확인할 수 있다.