**Pintos Project 3: Threads**

담당 교수 : 박성용

학번 / 이름 : 20181632 / 박성현

개발 기간 : 2022- 11-05 ~ 2022-11-12

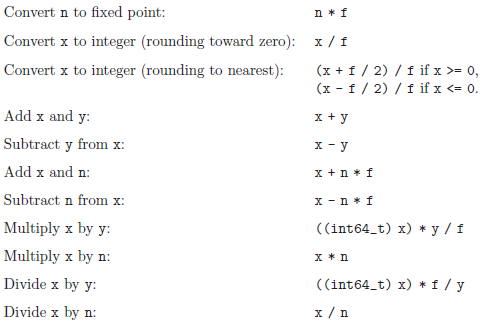
1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.
* Busy-waiting 방식으로 구현되어 있는 스케줄링 방식을 blocked\_list를 사용해 더욱 효율적이게 변경
* Priority에 의해 thread의 우선순위가 결정될 수 있게 구현
* BSD scheduler 구현

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock
     + 현재 pintos는 Busy-waiting으로 스케줄링이 되고 있다. 이는 blocked-list없이 계속 ready 와 running state를 번갈아 돌아가며 깨어날 시간을 확인한다. 이는 매우 비효율적이기에 blocked-list를 만들어 깨어날 시간이 되지 않은 thread들을 넣어둔다. 깨어날 시간이 된 thread만 ready-list로 옮겨 비효율성을 개선시킨다.
  2. Priority Scheduling
     + 각 thread들은 각자 priority를 가지고 있다. 이 priority에 맞춰 우선순위가 높은 thread가 먼저 실행될 수 있도록 한다.
  3. Advanced Scheduler (추가구현을 한 경우)
     + 주어진 시간마다 priority, load-avg, recent-cpu를 재계산한다. 1개의 queue를 사용한다. 4 ticks마다 priority를, 1초마다 recent-cpu와 load-avg를 계산해 실시간으로 스케줄링을 한다.
  4. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.
   * Timer-interrupt가 일어날 때 마다 현재 시간 ticks를 인자로 thread\_wake\_up함수를 호출한다. 이 함수에서 blocked-list의 thread의 wake-up 변수와 현재 시간을 비교해서 일어날 시간이 되었으면 blocked-list에서 제거하고 ready-list에 넣는다.
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.
   * Thread\_create함수에서 thread를 생성 후, running thread와 ready-list의 가장 앞 thread의 priority를 비교한다. Ready-list의 가장 앞 thread는 가장 큰 priority를 가지므로 running thread와 가장 앞 thread를 비교해 running thread가 더 작은 priority를 가지면 thread\_yield함수를 통해 cpu의 제어권을 넘겨준다. 또한 thread\_set\_priority함수에서 새로 지정할 priority가 running thread의 priority보다 더 작을 경우, running thread의 priority가 가장 크다고 보장할 수 없기에 thread\_yield를 호출한다.
3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)
   * Nice: 각 thread가 가지고 있으며 -20 ~ 20의 값을 가진다. 값이 클수록 우선순위를 낮춘다. 기본적으로 0 값이며 부모 thread에서 값을 상속받는다.
   * Recent\_cpu: 각 thread가 가지고 있으며 실수 값을 가진다. 최근에 얼마나 cpu를 많이 사용했는지 나타낸다. 값이 클수록 더 많은 cpu time을 사용했다고 할 수 있다. 1초마다 재계산된다.
   * Load\_avg: 전역 값이며 실수 값이다. 최근 1분동안 수행가능한 평균 thread의 개수를 나타낸다. 값이 클수록 priority가 천천히 증가하고 작으면 빨리 증가한다. 수행가능한 thread의 개수가 많으면 모든 thread가 cpu를 골고루 사용해야하기에 priority를 천천히 증가시켜 모든 thread가 cpu time을 배분 받을 수 있게 해야 하고 수행가능한 thread의 개수가 적으면 priority가 빨리 증가해 하나의 thread가 조금 많은 cpu time을 가져도 모든 thread들이 골고루 받을 수 있기 때문이다. 1초마다 계산한다.
   * 값들은 다음과 같이 계산된다.
     + Priority = PRI\_MAX(63) – (recent\_cpu / 4) – (nice \*2)
     + Recent\_cpu = (2 \* load\_avg)/(2 \* load\_avg + 1) \* recent\_cpu +nice
     + Load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_threads
   * 또한 실수 계산을 지원하지 않으므로 다음과 같이 fixed\_points연산을 구현한다.



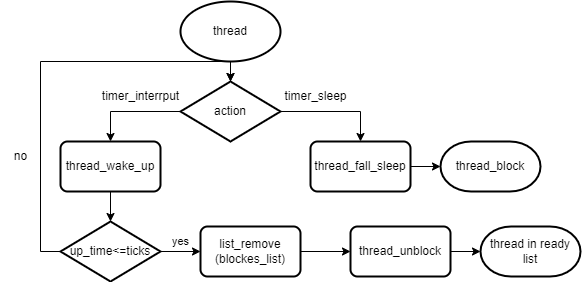
여기서 x, y는 실수이며 n은 정수이다. F는 소수점을 나타내며 32bit 기준 1bit는 sign, 17bits는 integer, 14bit는 소수부분이다. 그러므로 f는 1<<14로 정의한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

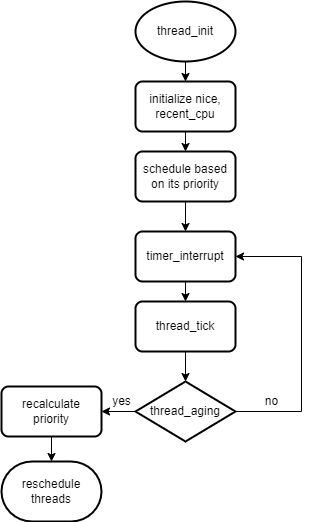
* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
  1. 11-05: 프로젝트 시작 및 alarm-clock 구현
  2. 11-09: prioirty구현 (aging제외)
  3. 11-10: aging 및 BSD구현
  4. 11-11: 오류 fix
  5. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수
    - Alarm clock
      * Device/timer.c에서 timer\_sleep함수에 while문 대신 thread\_fall\_sleep를 추가해 thread를 잠들게 한다.
      * Device/timer.c에서 timer\_interrupt함수에서 thread\_wake\_up 함수를 통해 blocked-list에서 일어날 시간이 된 thread들을 깨운다.
      * Thread/thread.h에서 thread구조체에 up\_time이라는 변수를 추가했다. 이 변수는 각 thread가 일어날 시간을 의미한다.
      * Thread/thread.c에서 잠든 thread의 list blocked-list 선언
      * Thread/thread.c에서 blocked\_list를 돌면서 각 thread의 up\_time을 현재 tick과 비교해 깨울 thread를 blocked\_list에서 제거하고 unblock시키는 thread\_wake\_up함수 추가
      * Thread/thread.c에서 현재 thread에 up\_time을 설정하고 blocked-list에 추가 후 block시키는 thread\_fall\_sleep함수 추가
    - Priority scheduling & BSD scheduler
      * Thread/synch.c에서 sema\_up에 sema\_waiter를 정렬하고 cpu를 넘겨주는 코드를 추가했다.
      * Thread/synch.c에서 sema\_down에 priority순으로 정렬하여 sema\_waiter에 값을 넣어준다.
      * Thread/thread.c에서 thread\_yield함수에 ready\_list에 thread를 넣을 때, 정렬하여 넣어준다.
      * Thread/thread.c에서 list\_insert\_ordered함수 사용을 위한 priority 비교함수 compare\_priority함수 추가
      * Thread/thread.c에서 thread\_create함수에 running thread와 ready\_list의 젤 앞을 비교해 우선순위로 재정렬하는 부분을 추가했다.
      * Thread/thread.c에서 thread\_unblock함수에서 priority를 정렬해서 ready-list에 넣어주는 기능 추가
      * Thread/thread.c에서 Priority를 계산하는 cal\_priority, recent\_cpu를 계산하는 cal\_recent\_cpu, load\_avg를 계산하는 cal\_load\_avg함수 추가
      * Thread/thread.c에서 running thread의 priority를 변경하는 thread\_set\_priority, running thread의 priority를 받아오는 thread\_get\_priority, 현재 thread의 nice값을 변경하는 thread\_set\_nice, nice값을 받아오는 thread\_get\_nice, load\_avg를 받아오는 thread\_get\_load\_avg, recent\_cpu의 값을 받아오는 thread\_get\_recent\_cpu 함수 구현
      * Thread/thread.c에서 system tick계산을 위해 전역변수로 int64\_t ticks를 선언 후, thread\_tick 함수에서 ticks 1씩 증가, thread\_aging과 thread\_mlfqs시 thread\_aging함수 호출하는 부분 추가
      * Thread/thread.c에서 4ticks마다 priority계산, 1초마다 load\_avg와 recent\_cpu 재계산하는 부분 추가
      * Thread/thread.c에서 init\_thread함수에서 현재 thread가 부모의 nice와 recent\_cpu를 상속받을 수 있게 하는 부분 추가
      * Thread/thread.c에서 fixed point연산을 위한 함수들 구현

1. **연구 결과**
   1. **Flow Chart**

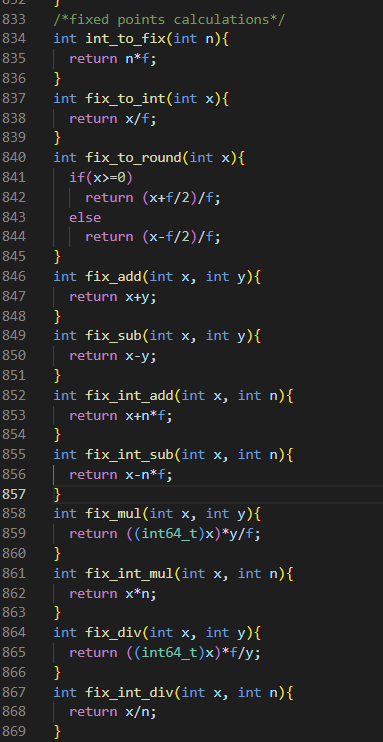
* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)
* Alarm clock



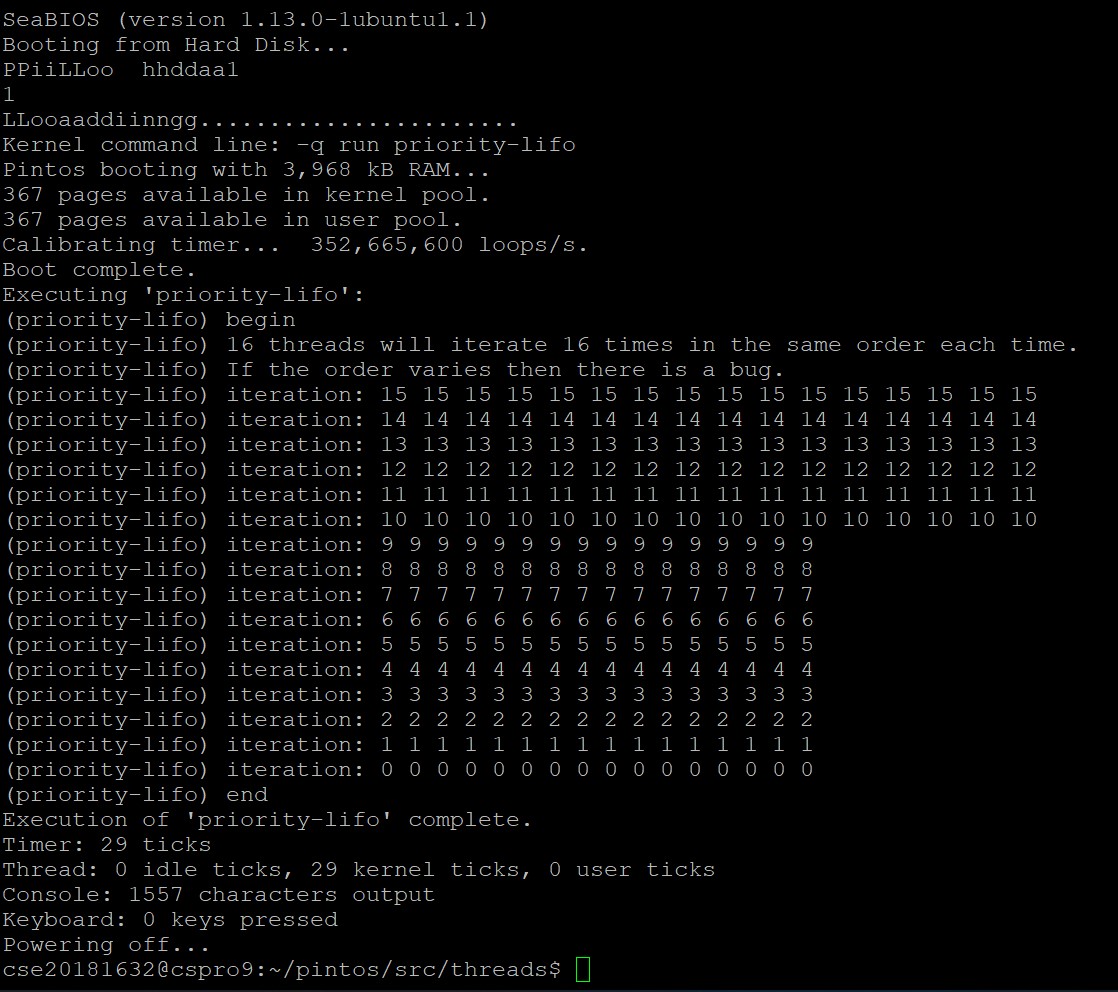
* Priority scheduling



* 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
* **Alarm clock**
  1. 가장 먼저 구현한 부분은 timer.c의 timer\_sleep함수이다. 현재 busy-waiting을 사용하므로 thread들은 계속 running state와 ready state를 왔다갔다 한다. 이런 비효율성을 제거하기위해 잠들어 있는 thread의 list인 blocked list를 만들었다. Timer\_sleep함수에서 thread가 잠이 들 때, thread\_fall\_sleep함수를 호출한다. 호출 시, 인자로는 현재 시간과 잠들어 있을 시간, 즉 start와 ticks를 더해서 넘겨준다. 또한, timer\_interrupt함수에서 매 tick마다 일어날 thread들이 있는지 확인해야 하므로 현재 tick과 함께 thread\_wake\_up을 호출한다.
  2. Thread\_fall\_sleep함수는 잠들 thread들을 blocked-list에 넣는다. thread구조체에 새로 선언한 int64\_t up-time에 인자로 받은 tick을 저장하고 blocked-list에 thread를 넣는다. 그리고 thread\_block함수를 통해 thread를 block state로 만들어 준다.
  3. Thread\_wake\_up함수는 매 tick마다 불리며 일어날 시간이 된 thread를 blocked-list에서 꺼내 ready-list에 넣는다. Blocked-list를 돌면서 up-time이 지난 thread들을 blocked-list에서 제거하고 thread\_unblock함수를 호출한다. Thread\_unblock함수에서 thread를 ready\_list에 넣어준다.
* **Priority scheduling & BSD scheduler**
* **위 두개의 기능에 대해서는 동시에 구현했다. Aging을 구현하기 위해 시간이 지남에 따라 priority를 계산해야 하는데, 이는 BSD에서도 같은 역할을 하기에 같이 구현하였다.**
  1. 가장 먼저, 내장 함수인 list\_insert\_ordered를 사용하기 위해 compare\_priority라는 함수를 구현했다. List\_insert\_ordered함수를 사용해 priority를 내림차순으로 정렬하여 해당 list에 넣을 수 있다.
  2. Semaphore 사용시에도 priority를 고려해야 한다. Sema\_down함수에 그냥 list\_puch\_back이 아닌 list\_insert\_ordered함수를 사용해 priority순으로 sema\_waiter list에 넣어주었다. 또한, sema\_up 시에도 list\_sort함수를 사용해 한번 더 priority를 정렬해준 후, thread\_yield를 통해 cpu의 제어권을 넘겨준다.
  3. 명세서에 나와있는 대로 다음과 같은 fixed point연산을 위한 함수들을 구현했다. F는 소수 bit인 14자리를 뜻하므로 #define f (1<<14)이다.



* 1. Ppt에 설명되어 있는 대로 thread\_tick함수에서 thread\_prior\_aging과 thread\_mlfqs시에 thread\_aging을 호출하도록 구현했다. Thread.c에서 timer ticks를 알 수 있도록 ticks이라는 전역 변수를 사용해 tick을 1씩 증가시켰다. Thread\_aging함수는 running thread가 아니면 recent\_cpu의 값을 1 증가시킨다. 그리고 thread\_tick함수에서 설정한 tick을 기준으로 100tick, 즉 TIMER\_FREQ마다 load\_avg를 계산하고 모든 thread의 recent\_cpu를 다시 계산한다. 4tick마다는 모든 thread의 priority를 다시 계산한다.
  2. Thread\_create함수는 thread를 하나 생성한다. 이때, 이 생성된 thread와 ready-list의 젤 앞, 즉 priority가 가장 높은 thread를 비교한다. 만약 생성된 thread의 priority가 더 낮다면 thread\_yield함수를 통해 cpu의 제어권을 넘겨주는 코드를 함수 마지막에 추가했다.
     + 문제점: 현재 thread의 priority를 받아올 때, thread\_get\_priority함수를 사용하면 출력이 밀린다. 그래서 thread\_current()->priority를 사용했다. Thread\_get\_priority함수 내부에서도 똑같이 thread\_current()->priority를 return하는데 출력이 밀리는 이유를 모르겠다.
  3. Thread\_unblock와 thread\_yield함수에서 ready\_list에 thread를 넣을 때, 그냥 list\_push\_back이 아닌 compare\_priority함수를 이용한 list\_insert\_ordered함수를 사용해 priority를 내림차순으로 정렬해 넣는다.
  4. Cal\_priority함수에서는 idle\_thread가 아니면 현재 thread의 priority를 계산한다. 계산식은 위에 설명되어 있다. 만약 priority의 범위 0~63을 벗어나면 재조정해준다. Cal\_recent\_cpu함수도 cal\_priority함수과 같은 기능을 수행한다. 단지 priority가 아닌 recent\_cpu를 계산해 준다. Cal\_load\_avg함수는 load\_avg값을 계산한다. 명세서에 나와있듯이 idle thread가 아니면 ready\_thread의 개수를 하나 늘려준다. Ready\_thread의 개수를 사용해 load\_avg를 계산한다.
  5. Thead\_set\_priority함수에서는 priority를 새로 설정한다. 만약 새로 설정될 priority가 현재 priority보다 낮을 경우, 새로 설정될 priority가 가장 높은 우선순위를 가진다는 보장이 없기에 thread\_yield를 통해 rescheduling해준다. Mlfqs를 사용할 땐, priority가 임의로 변경될 수 없으므로 함수를 비활성화 시켜준다. Thread\_get\_priority는 현 thread의 priority를 반환한다.
  6. Thread\_set\_nice함수는 thread의 nice값을 재설정한다. 이때, interrupt도 비활성화 해준다. Priority는 nice값을 기반으로 계산되므로 priority도 다시 계산한다. Priority가 재설정되었으니 우선순위도 다시 계산한다. Thread\_get\_nice는 현재 thread의 nice값을 반환한다.
  7. Thread\_get\_load\_avg와 thread\_get\_recent\_cpu는 현 thread의 load\_avg와 nice값을 반환한다. 반환하는 모든 함수는 interrupt를 비활성화 해 줘야한다. 반환시 값이 변경되는 것을 방지하기 위함이다.
  8. 마지막으로 init\_thread함수에 현재 thread가 부모 thread의 recent\_cpu와 nice값을 상속받는 기능을 추가한다.
  9. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석



* 가장 먼저 thread\_set\_priority(PRI\_DEFAULT + THREAD\_CNT+1)를 통해 main thread의 우선순위를 1순위로 둔다.
* 그리고 for문을 통해 priority가 1~16인 thread를 생성한다. 0번 thread가 1의 priority를 가지고 가장 먼저 생성된다.
* 하지만 priority scheduling에 의해 가장 priority가 높은 15번째 thread, 즉 16의 priority를 가진 thread가 가장 먼저 수행된다.
* make check 수행 결과를 캡처하여 첨부

