**Pintos Project 3: Threads**

담당 교수 : 김영재

조 / 조원 : 우석민

개발 기간 : 2023-11-15 ~ 2023-11-18

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

Alarm clock이 현재는 busy waiting 방식으로 구현되어 있다. 이를 더 효율적으로 구현하기 위해 blocked state로 놓는다. 각 tick마다 시간이 충분히 지났는지 확인하고, ready state로 바꾼다.

현재 Scheduling 알고리즘이 단순한 round robin으로 구현되어있다. 이를 Priority scheduling 방식으로 다시 구현을 한다. Priority가 높은 thread가 CPU를 점유해야 하며, priority가 도중에 바뀌면 더 높은 priority를 가진 쓰레드가 preemptive하게 실행되어야 한다. Starvation을 방지하기 위해 priority aging을 구현한다.

Priority scheduler은 수동으로 priority를 지정해야 한다는 문제가 있다. 쓰레드가 얼마나 많이 CPU를 점유하고 있었는지를 바탕으로 적절하게 priority를 조절할 수 있는 MLFQ중 BSD Scheduler을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

현재의 Alarm clock은 busy waiting 하는 방식으로, 다시 scheduling이 되면 yield를 하는 방식으로 구현이 되어있다. Context switch의 overhead를 줄이기 위해 따로 관리하여 tick마다 sleep이 끝났는지 확인한다. Context switch의 overhead를 줄일 수 있기 때문에 성능이 좋아질 것으로 예상된다.

* 1. Priority Scheduling

현재는 단순한 round-robin scheduler가 구현이 되어있다. 따라서 특정 쓰레드가 중요하다고 해서 그 쓰레드를 계속 schedule할 방법이 없다. Priority scheduling을 구현한다면 priority에 따라 쓰레드가 실행되어 중요한 쓰레드를 먼저 실행할 수 있다. 이 때 priority가 낮은 쓰레드가 아예 실행이 안되는 경우를 대비하여 priority aging도 구현해야 한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

위에서 구현한 Priority scheduling에서는 사용자가 직접 priority를 조정해야 한다. 따라서 쓰레드가 compute intensive에서 I/O intensive으로 바뀌는 등의 상황에서 적절히 priority 조절을 하는 것이 어렵다. 쓰레드의 지금까지의 cpu 이용 패턴을 이용해서 priority를 자동으로 조절하여 scheduling을 하는 MLFQ를 사용하면 쓰레드가 중간에 자원 사용 패턴을 바꿔도 유동적으로 scheduling 전략을 바꿀 수 있게 된다. 따라서 이 구현을 하면 자원을 더 효율적으로 쓸 수 있게 될 것이다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Pintos에서 Blocked 상태의 스레드는 thread\_unblock에서 진행된다. Blocked된 쓰레드는 ready queue에 들어있지 않는데, ready queue에 넣어주면 다시 schedule이 가능해진다. 따라서 ready queue에 넣고, process의 상태를 나타내는 변수를 ready로 바꿔주면 스레드를 깨울 수 있다.

Alarm clock을 구현할 때는 timer\_interrupt함수에서 계속 blocked된 상태의 thread가 깨워져야하는지 (시간이 다 지났는지) 확인하고, thread\_unblock을 호출해주면 된다.

1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.

Priority scheduling에 따르면 더 높은 priority를 바로 실행해야 한다. 예를 들어 lock에서 깨어난 thread가 priority가 running thread보다 priority가 높으면 바로 깨어난 thread를 실행해준다. 이를 위해 ready queue에 쓰레드가 들어가는 모든 경우에서 running thread와 priority를 비교해서 yield를 할지 정해야 한다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

Priority를 계산하기 위해서는 크게 4가지가 더 필요하다.

* + - Fixed-point arithmetic

핀토스에는 floating point number을 제공하지 않아서, 아래의 load\_avg와 recent\_cpu를 나타낼 수가 없다. 따라서 비교적 구현이 간단한 fixed-point arithmetic을 구현하여 load\_avg와 recent\_cpu의 계산을 할 수 있게 해야한다. Fixed point arithmetic에서는 고정된 비트 수를 분수를 나타낼 수 있게 할당하여 정수와 분수 비트를 합하여 숫자를 나타내게 된다. 이렇게 얻은 fixed point number을 integer와 함께 계산을 하기 위해서는 추가적인 연산 방법을 써야 한다. 예를 들어서 q의 비트를 분수 비트로 할당하고, f = 1 << q라고 했을 때 fixed point number x와 integer n을 더하기 위해서는 x + n \* f를 해야한다.

* + - Load\_avg

Load\_avg는 지난 1분동안에 ready state에서 schedule이 되기를 기다리는 쓰레드가 몇 개인지 나타낸다. 이는 ready thread의 개수의 moving average를 구하는 형식으로 계산이 된다.

Load\_avg = (59/60)\*load\_avg + (1/60)\*ready\_thread

이는 1초마다 계산이 되며, 그렇기 때문에 계산할 때마다 59/60의 기존 load\_avg와 1/60의 ready\_thread를 더해줘서 moving average를 구하는 것이다.

* + - niceness

이는 특정 thread가 다른 thread에게 얼마나 nice한지 나타낸다. Nice 하다는 것은 cpu를 더 잘 내어준다는 뜻이다. 따라서 niceness가 양수이면 priority가 작아지고, 음수이면 priority가 커진다.

* + - Recent\_cpu

Recent\_cpu는 각 쓰레드마다 최근에 얼마나 많이 cpu를 썼는지 나타낸다. Interrupt가 일어날 때마다 running thread의 recent\_cpu는 1씩 증가된다. 또한 시간에 따라서 최근에 cpu를 받지 못한 thread의 recent\_cpu값이 decay되어야 한다. 이는 1초마다 하고, exponentially weighted moving average를 활용한다.

recent\_cpu = (2\*load\_avg)/(2\*load\_avg + 1) \* recent\_cpu + nice

이렇게 최근에 cpu를 계속 쓴 thread는 1초마다 recent\_cpu가 증가하지만, 나머지 thread들은 위의 식으로 인해 약간 decay가 된다. 이 때 nice 값을 함께 넣어주어서, nice가 높으면 recent\_cpu가 같이 높아져서 priority가 더 낮아진다.

* + - Priority

Priority의 계산 자체는 아래의 식을 사용하고, 근접한 정수로 PRI\_MIN과 PRI\_MAX 사이의 값으로 조절한다. Priority 계산은 4 tick마다 모든 쓰레드에 대해서 다시 계산이 되어진다.

priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2)

식을 살퍄보면 recent\_cpu가 높으면, 즉 최근에 cpu를 많이 사용했다면 priority가 낮아진다. 또한 nice가 높으면 priority가 낮아진다. 이런 요소들을 이용해서 priority 계산을 하게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2023-11-15 ~ 2023-11-16 Alarm Clock 구현

2023-11-16 ~ 2023-11-17 Priority Scheduling 구현

2023-11-17 ~ 2023-11-18 Advanced Scheduler 구현

* 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
  + **Blocked 상태의 쓰레드를 깨우는 방법**

Timer\_sleep으로 block된 thread는 따로 list에 넣어서 관리한다. 이 때 매 tick마다 이 list에 있는 thread들이 충분히 sleep을 하였는지 확인하여 thread\_unblock을 호출하면 된다. 이를 위해서는 thread\_sleep을 할 때 thread 자신의 wakeup time을 따로 저장을 해두어야 한다.

* + **Ready list에 running thread보다 priority가 더 높은 쓰레드가 들어온 경우 preemption을 바로 하는 방법**

Ready list에 running thread보다 priority가 더 높은 쓰레드가 들어올 수 있는 경우는 여러가지가 있고, 이 때 priority를 비교해서 yield를 해야한다.

먼저 thread.c에 있는 thread\_create를 수정해야한다. Running thread보다 높은 thread를 생성하는 경우, thread\_yield()를 호출하여 preemption을 한다.

Thread.c에 있는 thread\_set\_priority에서도 priority를 set을 하고 thread\_yield를 진행해 현재 thread보다 더 높은 thread가 ready queue에 있으면 preemption을 한다.

Sync.c에 있는 sema\_up에서 깨어난 thread가 running thread보다 priority가 높으면 thread\_yield를 부르게 한다.

* + **Advanced scheduler의 priority를 구현하는 방법**

Fixed-point arithmetic은 계산이 까다로운 경우 함수를 만들어서 계산한다. 먼저 integer을 fixed point number로 바꾸는 itof, 반대의 ftoi가 필요하다. 그 다음 fixed point number이라 integer을 더하거나 뺄 때 쓰는 fiadd, fisub가 필요하다. 마지막으로 fixed point number끼리 곱하거나 나눌 때 쓰는 ffmul, ffdiv가 필요하다. Fixed point number끼리 더하고 빼는 경우는 정말 더하거나 빼기만 하면 되어서 따로 구현을 하지 않고, fixed point number이랑 integer를 곱할 때도 마찬가지이다. 이러한 fixed-point arithmetic은 thread.c 에서만 쓸 것이므로 이 파일에 추가해주면 된다.

Niceness는 매번 계산하는 것이 아니라, 따로 설정을 하거나, thread가 만들어질 때 부모의 값을 inherit하고, 기본 값은 0이다. Thread.c에 있는 init\_thread에 niceness를 0으로 초기화 한다. 그 다음 thread\_create에서 부모의 nice값을 그대로 child의 nice에 넣어준다. Nice값이 바뀌면 priority 계산을 다시 해줘야 하기 때문에 thread\_set\_nice에서 priority를 계산하고, thread\_yield를 할지 말지 결정을 한다.

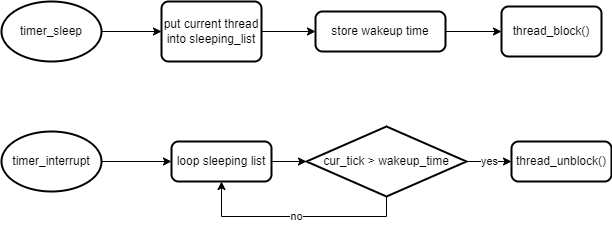
Load\_avg는 모든 thread가 공통으로 사용하는 변수이다. Thread.c에 전역변수로 선언하고, thread\_init에서 0으로 초기화 해준다. External interrupt 환경에서 thread\_tick이 불리는데, 여기에서만 load\_avg값이 변경되어서, 중간에 interrupt를 걱정할 필요는 없다. 매 초마다 load\_avg값이 업데이트 되기 때문에 timer\_ticks()를 TIMER\_FREQ로 나눈 나머지를 구해서 0이 될 때 새롭게 계산식을 이용해서 계산한다. 이때 ready\_state에 thread가 얼마나 있는지 계산할 때 idle\_thread는 제외해야한다.

Recent\_cpu는 tick마다 running thread에 1씩 더해줘야 한다. 이 또한 thread.c에 있는 thread\_tick에서 계산하면 된다. 또한 1초마다 모든 thread에 대해서 다시 계산을 해야 한다. 이 때에도 timer\_ticks()%TIMER\_FREQ가 0인지 확인하면 된다. 그 다음 thread\_foreach 함수를 이용해서 모든 쓰레드에 대해서 calculate\_recent\_cpu와 같은 함수를 부르게 하고, 이 함수에서 계산을 진행하면 된다.

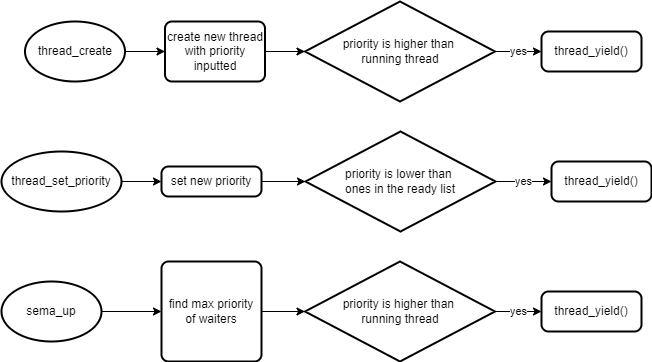
마지막으로 priority는 4 tick마다 한번씩 모든 thread에 대해서 계산을 해야 한다. Thread.c에서 thread\_tick에서 계산하면 된다. 이 때에도 thread\_foreach 함수를 이용해서 calculate\_priority같은 함수를 줘 모든 thread의 priority를 재계산한다. 이때 priority가 재계산 되어 ready queue에 있는 thread의 순서가 priority 순서가 아니게 될 수 있는데, sorting을 진행한다. 그 다음 running thread보다 priority가 높은 thread가 있으면 yield를 하게 하면 된다.

1. **연구 결과**
   1. **Flow Chart**

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

****

Alarm clock을 구현할 때 Blocked 상태에서 깨우기 위해서는 크게 두 함수를 수정해야 한다. Timer\_sleep에서는 sleeping\_list에 sleep하는 thread를 넣어주고 wakeup time을 저장한다. Timer\_interrupt에서는 sleeping list를 순회하며 wakeup time까지 도달했으면 thread\_unblock()을 불러 blocked 상태에서 깨우게 된다.



Ready list에 priority가 더 높은 thread가 들어올 수 있는 경우는 크게 3가지 이다. Thread\_create, thread\_set\_priority, sema\_up 함수에서 이런 일이 이러날 수 있는데, 이 때마다 priority가 더 높은게 들어갔는지 확인하고 thread\_yield를 하면 된다.

* 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

**Blocked 상태의 쓰레드를 깨우는 방법**

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

Device/timer.c에 있는Timer\_sleep에서 sleeping\_list라는 list를 따로 만들었다. Timer\_sleep 함수가 불리면 이 list에 넣었고, wakeup\_time도 thread에 미리 저장해둬 나중에 비교를 용이하게 하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

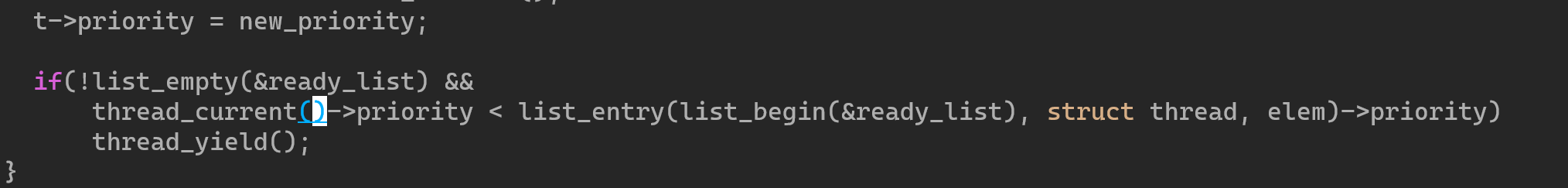
Timer\_interrupt 함수에서는 sleeping list를 순회하면서 각 thread의 wakeup time보다 현재 tick이 큰지 확인하여 unblock을 해줘야하는지 확인한다.

**Ready list에 더 높은 priority를 가진 thread가 들어온 경우 preemption을 진행하는 법**

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

Thread\_create 함수에서 마지막에 running thread와 새롭게 생성된 priority를 비교해서 thread\_yield()를 불러준다.



Thread\_set\_priority에서는 running thread의 priority를 바꾸는 것이기 때문에 바뀐 priority와 ready queue에 있는 thread 중 가장 높은 priority를 가진 thread와 비교하면 된다. Ready queue에 priority로 정렬되게 insertion을 진행했기 때문에 list의 맨 처음 부분에 있는 thread의 priority와 비교하면 된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Sema\_up에서 wait\_thread에 있는 thread 중 가장 높은 priority를 가진 thread를 깨워야 한다. 그러기 위해 get\_max\_priority\_thread라는 함수를 새롭게 만들고, 그 thread를 unblock 해주었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Get\_max\_priority\_thread 함수는 단순히 waiters list를 순회하면서 가장 높은 thread를 찾는 함수이다. 이렇게 unblock한 thread와 running thread의 priority를 비교해서 unblock된 thread의 priority가 더 높으면 yield를 진행한다. 바로 yield를 진행하면 interrupt level이 disabled된 상태에서 yield가 되기 때문에 old\_level으로 복구 한 뒤에 thread\_yield를 진행한다.

**Advanced priority scheduler 구현**

Advanced priority scheduler의 구현을 위해서는 fixed-point arithmetic을 수행할 수 있어야 한다. Recent\_cpu와 load\_avg가 real number이기 때문이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Thread.c 상단에 위와 같은 함수들을 작성하여 integer와 fixed-point number 사이의 계산을 용이하게 하였다.

Tick 단위로 업데이트를 해야하는 recent\_cpu, load\_avg, priority는 thread\_tick에서 업데이트를 진행하였다. 이때 mlfqs와 관련된 부분들은 mlfqs를 사용한다는 flag가 set되어 있다면 계산해야 한다. Thread\_mlfqs 변수가 이를 알려주는데, advanced priority scheduler와 관련된 내용을 계산하기 전에 이 변수를 확인해야 한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Load\_avg를 1초마다 업데이트 하기 위해 cur\_tick%TIMER\_FREQ가 0인지 확인하고, ready list를 순회하면서 idle thread가 아닌 thread의 개수를 샌다. 그리고 load\_avg의 식을 따라 계산을 진행하였다.

텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

Recent\_cpu 같은 경우에도 1초마다 계산을 해야 해서 cur\_tick%TIMER\_FREQ가 0인지 확인한 if block 안에 calculate\_recent\_cpu를 thread\_foreach 함수를 이용해 모든 thread에 대해서 불러주었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Recent\_cpu의 식을 그대로 fixed-point arithmetic을 이용하여 계산하였다. 이 때 2\*load\_avg/(2\*load\_avg+1)을 먼저 계산하는 이유는 recent\_cpu와 계산을 먼저 하면 overflow가 발생할 수 있기 때문이다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

또한 매 tick마다 recent\_cpu를 1씩 증가시켜야 하기 때문에 thread\_tick에 위와 같이 running thread의 recent\_cpu를 1씩 증가시켰다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

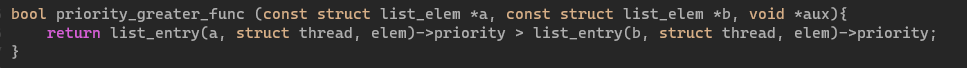
자동 생성된 설명

Priority 같은 경우에는 매 4 tick마다 모든 priority에 대해 계산을 해주었다. 이 때에도 thread\_foreach 함수를 활용해 모든 thread에 대해 calculate\_priority를 호출하였다. 이렇게 priority를 업데이트 하면 ready\_list에 있는 priority가 정렬이 어긋날 수 있기 때문에 다시 sorting을 진행하였다. 그리고 마지막에 ready\_list에 running thread보다 더 높은 priority가 있는지 확인하기 위해 ready\_list의 첫 thread와 priority를 비교하였다. 그리고 필요하면 intr\_yield\_on\_return()을 불러주었다. Thread\_tick 함수는 external interrupt 환경이기 때문에, 바로 thread\_yield()를 부르면 안된다. Intr\_yield\_on\_return은 interrupt가 끝나고 return 하고 나서 바로 thread\_yield()를 부르게 해주는 함수이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

위에서 thread\_foreach 함수로 calculate\_priority를 불러주었는데, 여기에서는 각 thread마다 priority 식을 이용하여 priority를 업데이트 해준다. 이 때 priority가 PRI\_MIN과 PRI\_MAX 사이를 벗어날 수 있으니 이런 경우를 처리해 주어야 한다.



또한 sorting을 진행할 때 priority\_greater\_func로 priority가 큰 경우에 true를 return하게 하였다. 이런 함수를 sorting 함수에 넣으면 priority가 가장 큰 원소가 맨 앞으로 오게 된다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Nice 값 같은 경우에도 nice값에 따라 priority가 달라질 수 있기 때문에 calculate\_priority를 호출해 준다. 그 다음에 priority가 바뀌었으면 ready\_list에 있는 thread보다 낮아졌을 수 있기 때문에 이를 확인하고 thread\_yield를 진행한다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

마지막으로 thread\_create에서 recent\_cpu와 nice같은 경우에는 부모 thread의 값을 inherit하는데, 이와 관련된 내용을 넣어준다.

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

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

먼저 현재 thread의 priority를 가장 높은 priority로 설정한다. (PRI\_DEFAULT+THREAD\_CNT+1) 그 다음 for문을 돌면서 priority가 다른 thread들을 생성한다. 이 때 id가 높은 thread가 더 높은 priority를 가지는 형식으로 실행이 된다. 마지막으로 현재 thread의 priority를 낮게 설정하여 새롭게 만든 thread들이 schedule 될 수 있도록 한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

새롭게 만들어진 thread들은 simple\_thread\_func가 불리게 된다. 여기에서 op의 특정 부분에 자신의 id를 넣게 된다. Op는 output buffer position을 뜻하고, 여기에 나중에 output할 내용들이 들어가게 된다. 이는 내 처음에 동적할당을 해주었다.



각 쓰레드에서 for문 안에서 thread\_yield를 부르는데, 이 안에서 priority가 변하지는 않기 때문에 계속 높은 priority를 가진 자기 자신이 schedule 된다. 그렇기 때문에 output buffer에 id가 연속으로 들어가게 된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

마지막에 부모가 생성한 thread들의 실행이 다 끝나면 다시 부모가 schedule이 된다. 이 때 cnt%THREAD\_CNT 마다 (priority-lifo) iteration: 을 출력하고, op에 있는 내용을 순서대로 출력한다. 위에서 thread 마다 id를 연속으로 op에 넣었기 때문에 (priority-lifo) iteration 뒤에는 하나의 id만 연속으로 계속 출력이 되어야 한다.

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

실제 출력은 이렇게 나왔고, 위에서 설명한대로 iteration: 뒤에 하나의 id만 연속해서 출력이 되었다.

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명