Operating Systems

Spring, 2022
School of Software, CAU

Project #2
- Page Allocation and Task Scheduling -

[프로젝트 보고서]
Template

학번: 20175371

이름: 강 명 석

1. 서론

본 보고서에서는 페이지 할당자와 스케줄러의 구현 방법과 결과에 대해 기술한다.

1.1 Page Allocation

Pintos에 기본 값으로 구현된 페이지 할당자의 정책은 First Fit Page Allocation 정책이다. First Fit Page Allocation 이란, 페이지 영역의 시작 주소부터 탐색을 시작하여 할당 받고자 하는 크기 만큼의 빈 공간을 찾으면 해당 공간을 반환해 주는 정책이다. 구현이 쉽다는 장점이 있지만, 페이지를 여러 번 할당하고 나면 큰 공간을 쪼개어 사용하게 된다. Buddy System Page Allocation의 경우 페이지를 할당할 때, 2^n 단위로 할당하는 정책이다. 할당 할 당시에는 요청한 크기보다 큰 공간을 할당 받기 때문에 메모리 낭비가 존재하지만, 해제 시에는 기존에 빈 공간과 다시 합치기가 용이하다는 장점이 있다. 자세한 내용은 본론에서 실제 구현된 내용을 가지고 설명하겠다.

1.2 Task Scheduling

Pintos에 기본 값으로 구현된 스케줄러의 정책은 RR 정책이다. RR정책의 경우 큐의 구분 없이 먼저 들어온 쓰레드를 먼저 처리하는 FIFO(First In First Out)정책을 적용한다. 반면 MFQ 정책은 우선순위가 다른 여러 개의 큐를 두어서, 더 빠르게 처리해야 하는 쓰레드를 먼저 처리할 수 있다는 장점이 있다.

2. 본론

본론에서는 Page Allocation과 Task Scheduling에 대한 상세 구현 방법에 대해서 설명한다.

1.1 Page Allocation

1.1.1) 기존 코드 분석

다음 [표 1]에 있는 함수들은 palloc.h에 선언된 함수에 대한 내용이다.

Palloc_init	페이지 할당자를 초기화 하는 함수. 커널 페이지와 사용자 페이지로 나눠서 각자 할당
Palloc_get_page	단일 페이지를 할당하는 함수
Palloc_get_multiple	인자로 주어진 수 만큼의 페이지를 할당. 해당 함수에 RR 정책이 적용된 알고리즘을 호출
Palloc_free_page	단일 페이지 free
Palloc_free_multiple	여러 페이지 free

[표 1] palloc 관련된 함수 들

이 중에서 palloc_get_multiple()함수를 분석하면 Round Robin 정책을 어떻게 구현했는지 확인 가능하다.

```
Void *palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)
{
  // 생략..
  // 아래 bitmap_scan_and_flip()함수에서 bitmap 스캔을 통해서 사용 가능한 first fit 페이지를 찾는다.
  Page_idx = bitmap_scan_and_flip (pool->used_map, 0, page_cnt, false);

// 생략..
```

[코드 1] - palloc_get_multiple 함수

Palloc_get_multiple()함수는 내부적으로 bitmap_scan_and_flip()함수를 호출해서 할당 받을 페이지의 인덱스를 계산한다. Bitmap_scan_and_flip() 함수는 다음과 같다.

```
size_t bitmap_scan_and_flip (struct bitmap *b, size_t start, size_t cnt, bool value)
{
  size_t idx = bitmap_scan (b, start, cnt, value);
  if (idx != BITMAP_ERROR)
    bitmap_set_multiple (b, idx, cnt, !value);
  return idx;
}
```

[코드 2] - bitmap_scan_and_flip 함수

Bitmap_scan_and_flip()함수는 내부적으로 다시 bitmap_scan()함수를 호출한다. Bitmap_scan_and_flip()함수는 bitmap_scan()함수를 통해서 찾은 인덱스에 대해서 페이지들의 할당 공간을 저장하고 있는 비트맵을 flip 하는 기능을 가진 함수이다.

```
size_t bitmap_scan (const struct bitmap *b, size_t start, size_t cnt, bool value)
{
  if (cnt <= b->bit_cnt)
    {
      size_t last = b->bit_cnt - cnt;
      size_t i;
      for (i = start; i <= last; i++)
        if (!bitmap_contains (b, i, cnt, !value))
          return i;
    }
  return BITMAP_ERROR;
}
```

[코드 3] - bitmap_scan 함수

최종적으로 First Fit 알고리즘이 구현된 내용은 bitmap_scan()함수이다. 함수의 로직을 분석하면 비트맵의 시작부터 연속적으로 같은 value가 할당된 영역을 찾자마자 해당 시작 인덱스를 리턴하고 있다.

페이지 할당의 로직을 정리하자면, palloc_get_multiple()함수를 통해서 인자로 주어진 page_cnt 만큼의 페이지를 할당한다. 이때, 내부적으로 bitmap_scan_and_flip() -> bitmap_scan()함수를 통해서 비트맵에서 가장 최초의 인덱스를 계산한다.

페이지 해제의 경우는 palloc_free_multiple()함수가 수행하며 할당받은 페이지의 시작 주소와 페이지 수인 cnt변수를 넘겨주면 할당을 해제해 준다.

1.1.2) Buddy System 설계 및 구현

기존의 palloc_get_multiple()함수를 이용하여 buddy system을 구현했다. [코드 1]을 보면 Page_idx 를 bitmap_scan_and_flip()함수를 통해서 계산하여 넘겨주기 때문에 알맞은 idx를 넘겨 주면 그 이후는 이전에 구현된 palloc_get_multiple()함수의 내머지 부분을 그대로 이용할 수 있다. 따라서 bitmap_scan_and_flip()함수를 대신하여, buddy system 로직에 맞는 page_idx를 반환하도록 함수를 새로 구현한다.

다음은 bitmap_scan_and_flip()대신에 buddy system을 구현한 함수이다.

```
size_t bitmap_scan_and_flip_buddy (struct buddy* buddy,struct bitmap *b, size_t start, size_t cnt, bool value)
{
  // buddy를 할당하고 해당 인덱스를 받아온다.
  size_t idx = buddy_alloc(buddy,cnt);

  // buddy를 할당하면 2^n 형태가 되므로 이를 계산
  cnt = next_power_of_2(cnt);

  // idx,cnt만큼 기존 bitmap을 flip한다.
  if (idx != BITMAP_ERROR)
    bitmap_set_multiple (b, idx, cnt, !value);

// buddy_alloc()에서 계산된 idx를 리턴하여 페이지를 할당 한다.
return idx;
}
```

[코드 4] bitmap_scan_and_flip_buddy 함수

기존 함수에서 bitmap_scan()함수를 통해서 계산하던 인덱스를 buddy_alloc()함수를 통해 buddy system에

맞는 인덱스를 계산하도록 하였다. Buddy 시스템의 구현은 깃허브[1] 주소를 참고하여 pintos에서 적용 가능하도록 수정하여 사용하였다.

Buddy system을 구현하기 위해서 추가한 관련 함수들은 다음과 같다.

Bitmap_scan_and_flip_buddy	Buddy를 할당한 뒤에 idx만큼 bitmap을 flip
Next_power_of_2	현재 숫자보다 큰 2의 제곱 수 중 가장 작은 2^n을 반환
Buddy_new	Buddy 자료구조를 초기 설정하는 함수. Complete binary tree 구조이며, array형태로 구현하였다.
Choose_better_child	Buddy system을 탐색하면서 최적의 child 노드를 찾는 함수이다. 해당 함수에서 반환되는 idx로 page를 할당한다.
Buddy_alloc	기존 bitmap_scan을 대신하는 함수. Bitmap을 스캔하는 대신, buddy를 할당할 수 있는 idx를 찾아서 해당 idx를 반환. 내부적으로 choose_better_child 함수를 호출
Buddy_free	Palloc_free를 호출할 때 같이 호출해 줌으로써 buddy 를 삭제한다.

[표 2] buddy system 구현 함수

1.2 Page Allocation Test

Pa.c 파일에 호출되는 함수는 다음과 같다. 커널 영역과, 사용자 영역을 구분하여 테스트를 수행 하였다. 각각의 영역에 대해서 페이지 할당 및 해제 직후의 페이지 상태를 출력하는 함수(palloc_get_status())를 호출하여 그 결과를 표시한다.

```
void run_patest(char **argv)
{
    while (1) {
        /* Test for kernel page */
        palloc_get_status(0);
        void * kerenl_palloc1 = palloc_get_multiple(0,13);
        palloc_get_status(0);

        void * kerenl_palloc2 = palloc_get_multiple(0,17);
        palloc_get_status(0);

        void * kerenl_palloc3 = palloc_get_multiple(0,3);
        palloc_get_status(0);

        void * kernel_palloc4 = palloc_get_multiple(0,16);
        palloc_get_status(0);

        ..생략


        /* Test for user page */
        palloc_get_status(PAL_USER);
        void * user_palloc = palloc_get_multiple(PAL_USER,15);
        palloc_get_status(PAL_USER);

        void * user_palloc2 = palloc_get_multiple(PAL_USER,1);
        palloc_get_status(PAL_USER);

        void * user_palloc3 = palloc_get_multiple(PAL_USER,4);
        palloc_get_status(PAL_USER);

        // .. 생략
        timer_msleep(1000);
    }
}
```

[코드 5] run_patest 함수

먼저 커널영역에 대한 페이지 할당이다. 할당 및 해제를 진행한 순서는 다음과 같다. 총 5개의 페이지를 할당 및 해제하였다.

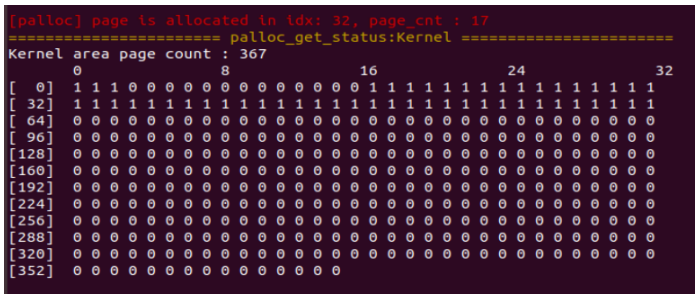
Palloc1	13
Palloc2	17
Palloc3	3
Palloc4	16
Pfree2	17
Pfree3	3
Palloc5	40
Pfree4	16
Pfree5	40
Pfree1	13

커널영역은 쓰레드에게 page를 할당한다. 따라서 맨 처음 시작 시에 3개의 페이지가 이미 할당 된 상태이다.

[1] https://github.com/lotabout/buddy-system

1. 첫번 째 할당인 13을 할당 했을 때의 결과이다. Idx 16에 페이지를 할당하였다.

2. 17만큼을 할당 하였을 때, idx 32번째 부터 32개의 페이지 공간을 할당 받았다.



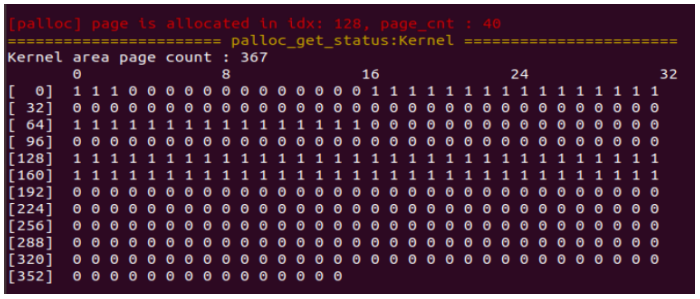
3. 3만큼 할당 하였을 때, idx 4부터 4개의 페이지 공간을 할당 받았다.

4. 16만큼 할당 하였을 때, 16개의 페이지 공간을 할당 받았다.

5. 두 번째에서 17만큼 할당 받은 공간을 free 한 결과이다. Idx 32위치에 32개의 페이지 공간을 free 한다.

6. 세 번째에서 3만큼 할당 받은 공간에 대해서 4개의 페이지를 free 하였다.

7. 40 개의 페이지 공간 할당을 요청 받으면 다음과 같이 할당 된다. 총 64개의 페이지가 필요한데, 0-63,64-127에 할당된 일부 페이지 들이 있기 때문에 Idx 128에 할당 된다.



8. 네 번째에서 16만큼 할당 받은 공간을 free 한다.

```
[pfree] deallocate page in idx: 64, page_cnt : 16
====================== palloc_get_status:Kernel ======================
Kernel area page count : 367
      0               8              16              24              32
[  0]  1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[ 32]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 64]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 96]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[128]  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[160]  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[192]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[224]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[256]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[288]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[320]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[352]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

9. 일곱 번째에서 할당받은 64개의 페이지 공간을 할당 해제 한다.

```
[pfree] deallocate page in idx: 128, page_cnt : 64
====================== palloc_get_status:Kernel ======================
Kernel area page count : 367
      0               8              16              24              32
[  0]  1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[ 32]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 64]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 96]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[128]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[160]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[192]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[224]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[256]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[288]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[320]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[352]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

10. 마지막으로 16개의 페이지 공간을 free 한다.

```
[pfree] deallocate page in idx: 16, page_cnt : 16
====================== palloc_get_status:Kernel ======================
Kernel area page count : 367
      0               8              16              24              32
[  0]  1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 32]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 64]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 96]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[128]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[160]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[192]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[224]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[256]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[288]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[320]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[352]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

사용자 영역에 대한 테스트는 커널 영역과 동일하기 때문에 처음 3개의 할당에 대한 테스트만 보고서에 작성하였다.

커널 영역은 쓰레드들이 사용할 공간이 미리 할당 되어 있었지만, 사용자 영역은 아직 사용되지 않아서 초기엔 페이지가 비어 있다.

```
====================== palloc_get_status:user ======================
User area page count : 367
      0               8              16              24              32
[  0]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 32]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 64]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 96]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[128]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[160]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[192]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[224]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[256]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[288]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[320]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[352]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

1. 처음에 15개의 페이지 공간을 요청하여 16개의 페이지를 할당 받았다.

```
[palloc] page is allocated in idx: 0, page_cnt : 15
====================== palloc_get_status:user ======================
User area page count : 367
      0               8              16              24              32
[  0]  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 32]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 64]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 96]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[128]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[160]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[192]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[224]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[256]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[288]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[320]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[352]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

2. 17개의 페이지를 요청하여 Idx 32에 32개의 페이지를 할당 받았다.

```
[palloc] page is allocated in idx: 32, page_cnt : 17
====================== palloc_get_status:user ======================
User area page count : 367
      0               8              16              24              32
[  0]  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 32]  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[ 64]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 96]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[128]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[160]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[192]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[224]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[256]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[288]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[320]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[352]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

3. 4개의 페이지를 요청하여 Idx 16에 4개의 페이지를 할당 받았다.

```
[palloc] page is allocated in idx: 16, page_cnt : 4
====================== palloc_get_status:user ======================
User area page count : 367
      0               8              16              24              32
[  0]  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 32]  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[ 64]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[ 96]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[128]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[160]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[192]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[224]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[256]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[288]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[320]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[352]  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

2.1 Task Scheduling

2.1.1 기존 코드 분석

pintos에서 쓰레드들이 어떻게 동작 하는 로직은 다음과 같다. 매 단위 시간(tick)을 구현하기 위해서 주기적으로 timer_interrupt()를 발생시킨다. 다음은 timer_interrupt 코드이다.

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();

  if (get_next_tick_to_wakeup() <= ticks) {
    thread_wakeup(ticks);
```

```
  }
}
```

주기적으로 timer_interrupt()가 발생하고, 이때 thread_tick()함수를 호출한다.

Thread_tick()함수는 다음과 같다.

```
void
thread_tick (void)
{
  struct thread *t = thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;

  /* Enforce preemption. */
  if (++thread_ticks >= TIME_SLICE)
    intr_yield_on_return ();
}
```

내용을 보면, 인터럽트를 받은 시점에 실행중인 쓰레드가 thread_tick()을 수행하고, 만약 해당 쓰레드의 thread_ticks 변수가 TIME_SLICE보다 크다면 intr_yield_on_return()함수를 실행한다. 기본 pintos에서 정의된 TIME_SLICE 상수는 4로 4개의 tick동안 쓰레드를 실행하고 intr_yield_on_return() 함수를 실행함으로써 스케줄링이 되는 구조이다. Intr_yield_on_return()함수가 실행되면 몇 개의 함수를 거쳐서 thread_yield()함수가 실행되게 된다. 해당 함수는 현재 실행중인 쓰레드를 ready 상태로 바꾸고 스케줄링을 수행하는 함수이다.

Thread_yield()함수는 다음과 같다.

```
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

Thread_yield()함수는 Ready_list에 현재 실행 중인 쓰레드를 넣고, 상태를 Running -> Ready로 바꾼 뒤에, 스케줄링 함수를 호출하고 있다.

Schudule()함수는 다음과 같다.

```
static void
schedule (void)
{
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

스케줄링 함수에서는 next_thread_to_run()함수를 통해서 다음에 실행할 쓰레드를 선택하고, switch_threads()함수를 호출하여 선택된 쓰레드와 현재 쓰레드의 문맥 교환을 수행한다. 따라서 다음 선택될 쓰레드를 선택하는 부분이 스케줄링에서 중요한 역할을 한다.

다음은 next_thread_to_run()함수이다.

```
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
    return list_entry (list_pop_front (&ready_list),
struct thread, elem);
}
```

이 부분이 Round Robin 스케줄링 정책을 구현한 내용이다. 단순히 ready_list에서 가장 앞에 있는 쓰레드를 꺼낸 뒤에 그 쓰레드를 리턴 하고 있다. 그 이외에 block -> unblock되는 경우, 쓰레드 생성 시점 등 다양한 영역에서 고려해야 하는 경우가 많기 때문에 모두 다 작성하진 않고 주요 스케줄링에 대한 구현은 위와 같이 되어 있다.

2.1.2 Multi Level Queue 설계 및 구현

MLQ를 구현하는 주요 아이디어는 2가지 이다.

1. Ready_list -> Multi level queue

2. Aging 기능

쓰레드 구현상에 여러 곳에 구현된 ready_list 관련된 부분을 모두 multi level queue에 들어가도록 바꿔주는 내용과, 매 틱마다 현재 실행 중인 큐보다 낮은 우선순위를 가지고 있는 쓰레드들의 age 값을 높여주는 내용이다.

1. Multi level queue

먼저 기존에 하나로 사용되던 큐를 다음과 같이 4개의 큐에 들어가도록 구현을 바꿔 준다.

```
static struct thread *
next_thread_to_run (void)
{
  // if (list_empty (&ready_list))
  //   return idle_thread;
  // else
  //   return list_entry (list_pop_front (&ready_list),
struct thread, elem);
  int next_queue = next_queue_to_search();
  switch (next_queue)
  {
  case -1:
    return idle_thread;
  case 0:
    return          list_entry          (list_pop_front
(&feedback_queue_0), struct thread, elem);
  case 1:
    return          list_entry          (list_pop_front
(&feedback_queue_1), struct thread, elem);
  case 2:
    return          list_entry          (list_pop_front
(&feedback_queue_2), struct thread, elem);
  case 3:
    return          list_entry          (list_pop_front
(&feedback_queue_3), struct thread, elem);
  }
```

그리고 매 thread_tick 마다 TIME_SLICE 가 고정되어 있었는데, 이 부분도 다음과 같이 현재 큐에 따라서 다르게 적용되도록 바꿔준다.

```
void
thread_tick (void)
{
  struct thread *t = thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;

  /* Enforce preemption. */
  // if (++thread_ticks >= TIME_SLICE)
  // intr_yield_on_return ();

  switch(current_queue){
    case 0:
      if(++thread_ticks >= TIME_SLICE_0)
        intr_yield_on_return();
      break;
    case 1:
      if(++thread_ticks >= TIME_SLICE_1)
        intr_yield_on_return();
      break;
    case 2:
      if(++thread_ticks >= TIME_SLICE_2)
        intr_yield_on_return();
      break;
    case 3:
      if(++thread_ticks >= TIME_SLICE_3)
        intr_yield_on_return();
      break;
  }

  /* increase age of low priority queue */
  aging();
}
```

이 외에도 Thread_create를 할 때, Thread_unblock 을 할 때 등, 현재 쓰레드의 우선순위와 queue를 고려하여 알맞은 queue에 쓰레드를 넣음으로써 구현 가능하다.

2. Aging 기능

우선 해당 함수는 매 Thread_tick()함수가 호출 될 때, 호출되게 함으로써 매 tick()마다 쓰레드들의 age를 바꿀 수 있다. 자세한 구현은 다음과 같다.

```
void aging(void){
  struct list_elem *e;
  struct thread *t;
  switch(current_queue){
```

```
    case 0:
      for (e = list_begin(&feedback_queue_1); e != list_end(&feedback_queue_1);)
      {
        t = list_entry (e, struct thread, elem);
        t->age = t->age + 1;
        if (t->age >= 20){
          t->age = 0;
          t->priority = 0;
          list_pop_front(&feedback_queue_1);
          list_push_back(&feedback_queue_0,&t->elem);
          e = list_begin(&feedback_queue_1);
          if(list_empty(&feedback_queue_1)){
            break;
          }
        }else{
          e = list_next(e);
        }
      }
    case 1:
        // 생략 case 0 과 동일

    case 2:
        //생략 case 0 과 동일
  }
}
```

현재 queue에 따라서 다르게 적용되도록 switch 문을 이용 했으며, 각각의 큐를 모두 보면서 age++을 해주고, 만약 age>=20 이 된다면, 현재 큐보다 한 단계 위의 큐에 삽입을 한다음 현재 큐에서 삭제 해주는 과정을 구현하였다.

추가적으로 스케줄링의 결과를 확인하기 위해서 debug_queue()함수를 구현하여 콘솔로 출력을 하였다.

2.2 Task Scheduling Test

생성한 프로세스들이 끝나지 않도록 단순 연산을 반복하는 코드를 test_loop()에 작성해 주었다.

```
void test_loop(void *aux)
{
    tid_t tid = thread_tid();
    while(1){
        int i =2;
        printf("[%s]                    test_loop in₩n",thread_name());
        for(;;){
            if(2147483647 % i == 0){
                printf("find the prime");
                break;
            }
            i++;
            if(i %100000000==0){
                timer_msleep(300);
            }
        }
        timer_msleep(1000);
    }
}
```

프로그램 실행 input
: mfq [p1.0]:[p2.1]:[p3.2]:[p4.0]:[p5.3]

디버깅 정보는 쓰레드가 교체될 때, 그 당시의 큐의 상태와 쓰레드들의 age, 매 틱 마다 실행 중인 Thread를 출력하였다.
먼저 가장 처음 쓰레드 5개가 생성된 직후 첫 번째로 실행되는 쓰레드는 p1이다. 큐를 보면 fq0:p4, fq1:p2, fq2:p3, fq3:p5,main이 들어있는 상태이다.

```
======================== Debug Info [MLQ] ========================
Current Working Thread: [p1] pri:0
current ticks: 35

=========== feedback_queue_0 =============
[p4] pri:0, age : 0
=========== feedback_queue_1 =============
[p2] pri:1, age : 0
=========== feedback_queue_2 =============
[p3] pri:2, age : 0
=========== feedback_queue_3 =============
[p5] pri:3, age : 0
[main] pri:3, age : 0

======================== Debug End [MLQ] ========================

[thread_tick] current_thread : p1, ticks:35
[thread_tick] current_thread : p1, ticks:36
[p1] test_loop in
[thread_tick] current_thread : p1, ticks:37
[thread_tick] current_thread : p1, ticks:38
```

P1은 fq0에서 실행되었기 때문에 4개의 tick을 마친 다음 fq0에 있는 p4가 실행된다.
P4가 실행되는 순간의 큐이다. P1의 경우 fq0에서 4개의 tick을 모두 수행하여 인터럽트 당했기 때문에, 우선순위가 하나 낮은 fq1로 들어갔다. 그 다음 나머지 fq0보다 낮은 큐에 들어있는 쓰레드들은 age가 실행된 tick인 4만큼 증가된 것을 볼 수 있다.

```
======================== Debug Info [MLQ] ========================
Current Working Thread: [p4] pri:0
current ticks: 39

=========== feedback_queue_0 =============
feedback_queue_0 is empty!
=========== feedback_queue_1 =============
[p2] pri:1, age : 4
[p1] pri:1, age : 0
=========== feedback_queue_2 =============
[p3] pri:2, age : 4
=========== feedback_queue_3 =============
[p5] pri:3, age : 4
[main] pri:3, age : 4

======================== Debug End [MLQ] ========================

[thread_tick] current_thread : p4, ticks:39
[p4] test_loop in
[thread_tick] current_thread : p4, ticks:40
[thread_tick] current_thread : p4, ticks:41
[thread_tick] current_thread : p4, ticks:42
```

P4 역시 fq0에 있었기 때문에 4개의 tick을 마치고 인터럽트 되며, fq0은 비어있기 때문에 그 다음 순위의 큐인 fq1에서 p2를 실행한다.

```
======================== Debug Info [MLQ] ========================
Current Working Thread: [p2] pri:1
current ticks: 43

========== feedback_queue_0 =============
feedback_queue_0 is empty!
========== feedback_queue_1 =============
[p1] pri:1, age : 4
[p4] pri:1, age : 0
========== feedback_queue_2 =============
[p3] pri:2, age : 8
========== feedback_queue_3 =============
[p5] pri:3, age : 8
[main] pri:3, age : 8

======================== Debug End [MLQ] ========================

[thread_tick] current_thread : p2, ticks:43
[p2] test_loop in
[thread_tick] current_thread : p2, ticks:44
[thread_tick] current_thread : p2, ticks:45
[thread_tick] current_thread : p2, ticks:46
[thread_tick] current_thread : p2, ticks:47
```

P2의 경우 fq1에서 수행되기 때문에 총 5개의 tick을 수행하였다. P3,p5,main의 경우 이전 2개의 쓰레드가 총 8개의 tick을 수행하여 age가 8인 것도 볼 수 있다.

```
======================== Debug Info [MLQ] ========================
Current Working Thread: [p1] pri:1
current ticks: 48

========== feedback_queue_0 =============
feedback_queue_0 is empty!
========== feedback_queue_1 =============
[p4] pri:1, age : 0
========== feedback_queue_2 =============
[p3] pri:2, age : 13
[p2] pri:2, age : 0
========== feedback_queue_3 =============
[p5] pri:3, age : 13
[main] pri:3, age : 13

======================== Debug End [MLQ] ========================

[thread_tick] current_thread : p1, ticks:48
[thread_tick] current_thread : p1, ticks:49
[thread_tick] current_thread : p1, ticks:50
[thread_tick] current_thread : p1, ticks:51
[thread_tick] current_thread : p1, ticks:52
```

P2가 끝나면 fq1에 가장 앞에 있던 p1이 다시 실행된다. P1도 fq0에서는 4tick이 실행되었지만 지금은 fq1에 있기 때문에 5tick을 수행하는 것을 볼 수 있다.

```
======================== Debug Info [MLQ] ========================
Current Working Thread: [p4] pri:1
current ticks: 53

========== feedback_queue_0 =============
feedback_queue_0 is empty!
========== feedback_queue_1 =============
feedback_queue_1 is empty!
========== feedback_queue_2 =============
[p3] pri:2, age : 18
[p2] pri:2, age : 5
[p1] pri:2, age : 0
========== feedback_queue_3 =============
[p5] pri:3, age : 18
[main] pri:3, age : 18

======================== Debug End [MLQ] ========================

[thread_tick] current_thread : p4, ticks:53
[thread_tick] current_thread : p4, ticks:54
[p3] thread age reaches 20. move fq2->fq1
[p5] thread age reaches 20. move fq3->fq2
[main] thread age reaches 20. move fq3->fq2
[thread_tick] current_thread : p4, ticks:55
[thread_tick] current_thread : p4, ticks:56
[thread_tick] current_thread : p4, ticks:57
```

P1이 끝나면, fq1에 있는 p4가 다시 수행되고 이때, p3,p5,main의 age가 각각 20이 되는 것을 볼 수 있다.

P3의 경우에는 fq2에 있었기 때문에 fq1로 상승하고, p5,main의 경우에는 fq2로 올라간다.

```
======================== Debug Info [MLQ] ========================
Current Working Thread: [p3] pri:1
current ticks: 58

========== feedback_queue_0 =============
feedback_queue_0 is empty!
========== feedback_queue_1 =============
feedback_queue_1 is empty!
========== feedback_queue_2 =============
[p2] pri:2, age : 10
[p1] pri:2, age : 5
[p5] pri:2, age : 3
[main] pri:2, age : 3
[p4] pri:2, age : 0
========== feedback_queue_3 =============
feedback_queue_3 is empty!

======================== Debug End [MLQ] ========================

[thread_tick] current_thread : p3, ticks:58
[p3] test_loop in
[thread_tick] current_thread : p3, ticks:59
[thread_tick] current_thread : p3, ticks:60
[thread_tick] current_thread : p3, ticks:61
[thread_tick] current_thread : p3, ticks:62
```

P4가 끝나면 fq1로 올라간 p3가 수행되고, p5,main이 fq2로 올라간 것을 볼 수 있다.

그리고 추가적으로 구현한 내용으로 thread가 block상태로 들어가는 경우 상위의 우선순위의 큐로 이동하도록 구현하였다.

다음 화면을 보면 현재 실행 중인 쓰레드는 p1이고 우선순위(pri)는 2로 fq2에 들어있는 상태이다. 초록색으로 표시한 글자를 보면 p1은 실행중에 thead_sleep으로 block되었고 522tick에서 wakeup 된다.

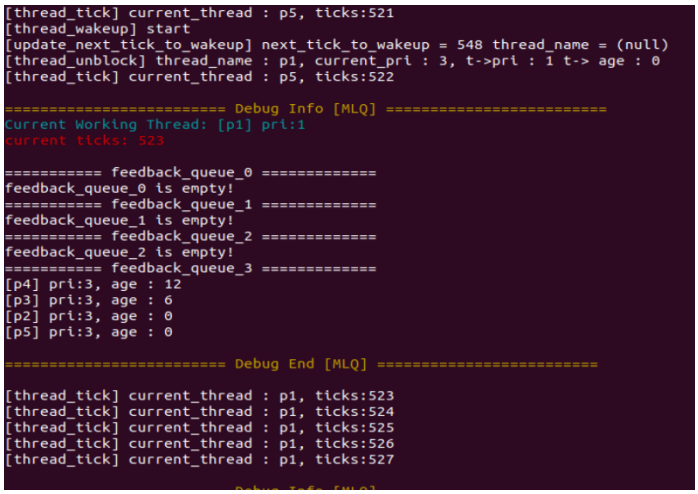
```
======================== Debug Info [MLQ] ========================
Current Working Thread: [p1] pri:2
current ticks: 490

========== feedback_queue_0 =============
feedback_queue_0 is empty!
========== feedback_queue_1 =============
feedback_queue_1 is empty!
========== feedback_queue_2 =============
feedback_queue_2 is empty!
========== feedback_queue_3 =============
[p5] pri:3, age : 18
[p4] pri:3, age : 12
[p3] pri:3, age : 6
[p2] pri:3, age : 0

======================== Debug End [MLQ] ========================

[thread_tick] current_thread : p1, ticks:490
[thread_tick] current_thread : p1, ticks:491
[p5] thread age reaches 20. move fq3->fq2
[thread_sleep] thread_name : p1, ticks : 522
[update_next_tick_to_wakeup] next_tick_to_wakeup = 522 thread_name = (null)
[thred_block] thread_name : p1
```

다음 화면은 522tick이 실행되는 부분이다. 522 tick에서 p1이 thread_unblock 함수에 의해서 다시 큐에 들어가는데, 이때 thread의 priority가 1로 설정된다. 즉, fq2에 있던 p1이 block된 직후에 우선순위가 올라가서 다시 unblock 될때, fq1로 들어가게 된다. 그 결과 아래에 나온 것처럼 523 tick에 p1이 실행될 때는, pri:1로 fq1에서 나온 것을 알 수 있다.



3. 결론

이론적으로만 배우고 외웠던 페이지 할당 정책과, 스케줄링 기법들을 실제로 직접 코드를 보면서 이해를 하고 그 과정을 디버깅 해보면서 확실하게 각각의 정책들의 장단점을 이해할 수 있었다. 개인적으로 MFQ를 구현하면서 가장 시간을 많이 소비한 부분은 스케줄링과 tick에 대한 것을 이해하는 것이었다. 기존의 문제들과 달리 tick이 timer_interrupt()에 의해서 호출된다는 것을 이해하는데 어려움이 많았다. 또한, 쓰레드 단위에서는 디버깅이 정말 어렵다는 것을 몸소 느꼈다.