|  |
| --- |
| Operating Systems  Spring, 2022  School of Software, CAU  **Project #2**  **- Page Allocation and Task Scheduling –**  **[프로젝트 보고서]**  학번: 20180500  이름: 장 훈 석 |

**1. 서론**

본 보고서에서는 페이지 할당자와 스케줄러의 구현 방법과 결과에 대해 기술한다.

**1.1 Page Allocation**

하나 이상의 메모리 블록을 할당하기 위해 필요한 공간을 효율적으로 탐색하는 것이 페이지 할당자의 역할이다. 핀토스에서는 Page Allocation을 위해 First Fit방식으로 구현되어 있다. First Fir알고리즘이란 메모리의 처음부터 검사하여 크기가 충분한 첫번째 메모리 블록을 선택하여 할당하는 방식이다. 더 자세히 설명해보자면 메모리의 처음부터 검사하여 페이지를 할당하고 싶은 크기만큼의 메모리블록이 비어있다면 그 비어있는 곳에 페이지를 할당하는 것이다. 이때 First Fit 알고리즘은 가장 간단하게 구현할 수 있고 빠른 할당이 가능하다는 장점을 가진다. 하지만 First Fit알고리즘은 페이지를 여러번 할당하고 Free를 해주게 되면 중간 중간 빈공간이 생겨 큰 공간을 할당 할 때 빈공간이 큰 공간보다 더 많이 남아있을 수 있지만 중간중간에 뚫린 형식으로 공간이 남아 있을 수 있기 때문에 비효율적인 단점이 있다. 그렇기 때문에 우리는 이번 Project에서 이 First Fit알고리즘을 Buddy System으로 교체하고자 한다.

Buddy System이란 페이지를 할당할 때 2^n 단위로 페이지를 할당하는 방식이다. 2^n 단위로 할당하기 때문에 페이지를 할당할 당시에는 요청한 크기보다 더 크게 할당할 가능성이 있기 때문에 메모리 낭비가 존재하긴 하지만 해제 시에는 기존에 있던 빈공간(기존에 사용하고 있지 않던 Buddy들)과 해제한 Buddy를 하나로 합쳐 First Fit알고리즘의 단점을 보완함을 알 수 있다. 본론에서 더 자세히 설명해 보겠다.

**1.2 Task Scheduling**

기존 Pintos의 경우 Task Scheduler로 Round-Robin 알고리즘을 사용하고 있다. Round-Robin 방식의 경우 일정 시간 tick을 넘어가는 순간 실행을 강제로 중단시키고 다음 Task를 진행하는 방식이다. Round-Robin의 경우 클럭 인터럽트가 발생하면 준비큐에 대기하고 있던 다음번 프로세스를 골라 실행한다. 이때 다음번 프로세스는 가장 오래 기다린 Process이다. 그리고 Round-Robin 방식은 큐의 구분 없이 먼저 들어온 쓰레드를 먼저 처리하는 FIFO(First in First Out)방식으로 스케줄한다.

하지만 우리가 구현할 MFQ는 우선순위가 있는 여러개의 큐를 두어 스케줄하는 방식이다. 쓰레드는 생성될 때 할당된 우선순위에 따라 해당 Feedback 큐로 이동하여 진행한다. 진행 방식을 간단히 살펴보면 우선순위가 높은 큐에 있는 쓰레드들이 모두 실행되게 되면 큐가 비게되는데 큐가 비게되면 다음 우선순위 큐에 있는 쓰레드들을 실행하기 시작한다. 또한 MFQ는 우선순위가 낮은 쓰레드들의 Starvation현상을 막기 위해 Aging기법을 사용한다. 낮은 우선순위를 가진 쓰레드들이 실행준비 상태가 된다면 매 시간 틱마다 Age를 1씩 증가시켜 Age가 특정 값에 도달하면 우선순위를 높이는 형식으로 Starvation현상을 해결할 것이다.

**2. 본론**

본론에서는 구현 방법에 대해 상세히 기술한다.

* 1. **Page Allocation** 
     1. **기존 함수 분석**

먼저 palloc.c에 있는 함수 동작 방식을 설명해 보겠다.

**<palloc\_get\_multiple>**

palloc\_get\_multiple 함수이다. 파라미터는 flags 와 page\_cnt 를 받는다. flags 를 통하여 user pool, kernel pool 중에 선택하게 된다. 그리고 페이지를 할당하는 동안 critical section을 보호하기 위하여 Lock을 걸고 작업이 끝나면 Lock 을 해제한다. Lock이 걸린 동안은 bitmap\_scan\_and\_flip 함수를 통해 해당 pool에서 사용 가능한 페이지 인덱스를 page\_idx 에 저장하게 된다. page\_idx 에 PGSIZE만큼 곱하여 page의 메모리를 설정한다. 만약 page\_idx 값이 올바르지 않다면 NULL값을 할당한다. 마지막으로 page를 반환하게 된다. 결국 이 함수는 인자로 주어진 수 만큼의 페이지를 할당하는 함수이다.

이때 bitmap\_scan\_and\_flip함수는 bitmap\_scan 함수를 내부적으로 호출하게 되는데 이 함수에 First-Fit 알고리즘이 구현되어있다. bitmap\_scan함수는 탐색하고 싶은 cnt만큼의 페이지들을 start부터 탐색하기 시작하여 value가 연속적으로 같은 공간이 있다면 그 시작 지점을 return한다. 이로써 할당할 수 있는 공간을 찾게 되는 것이다.

**<palloc\_free\_multiple>**

palloc\_free\_multiple 함수이다. 이 함수는 파라미터로 page 와 page\_cnt 를 받는다. 먼저 page 가 NULL 이거나 page\_cnt 가 0이라면 함수를 종료한다. 그리고 해당 페이지가 어느 pool인지를 확인하는 과정을 거치는데 만약 kernel pool, user pool 둘 중 해당되지 않는다면 NOT\_REACHED() 함수를 호출한다. 이어서 page\_idx 에 pg\_no 함수를 통하여 virtual page number를 저장한다. 그리고 해당 page\_idx 부터 할당하고자 하는 page\_cnt 만큼 bitmap\_set\_multiple 함수를 통하여 선택된 pool의 map의 값을 false로 설정한다. 결국 이 함수는 여러 페이지들을 Free하는 함수이다.

* + 1. **Buddy System 구현**

먼저 기존의 palloc\_get\_multiple 함수를 그대로 이용하는 방식으로 Buddy System을 구현하였다. 기존의 palloc\_get\_multiple함수는 page\_idx를 bitmap\_scan\_and\_flip함수를 이용하여 page의 시작점을 알려주는 방식으로 페이지들을 할당하였다. 고로 나는 bitmap\_scan\_and\_flip함수를 대체할 수 있는 함수를 구현하였다.

**< buddy\_allocation >**

이 함수는 버디시스템 방식에 따라 할당 가능한 공간의 시작 idx를 반환하여 palloc\_get\_multiple함수에서 쓸 수 있게끔 하였다. 구체적인 구현사항을 알아보면

|  |
| --- |
| size\_t buddy\_alloc(sturct bitmap \*b, size\_t start, size\_t cnt, bool value){  size\_t size\_val = 1;  size\_t idx = 0;  while(cnt > size\_val){  size\_val = size\_val \* 2;  }  while(idx <= bitmap\_size(b)){  if(!bitmap\_contains(b,idx,size\_val,!value)){  bitmap\_set\_multiple(b,idx,size\_val,!value);  return idx;  }  else{  idx += size\_val;  }  }  return BITMAP\_ERROR;  } |

먼저 첫번째 변수인 size\_val은 cnt가 2의 진수보다 클때까지 2를 곱함으로써 cnt가 2^n-1 < cnt <= 2^n 범위를 만족하는 2의 진수를 구하는 것이다. 버디시스템에서는 63개를 할당하고 싶다고 요청해도 위의 식을 만족하는 2의 진수인 64를 할당해 주어야 한다. 고로 첫번째 while문을 통해 cnt에 맞는 2의진수를 size\_val에 저장한다. 이후 다음 while 문에서 idx는 0부터 시작하여 할당가능 여부를 판단한다. 여기서 bitmap\_contains 함수를 통해 idx에서 size\_val만큼 연속적으로 메모리 할당이 가능하다면 bitmap\_set\_multiple함수를 이용해 사용하고 있다는 표시를 그 시작지점(idx)부터 size\_val만큼 한 뒤 해당 idx를 반환한다. 만약 검사를 진행하였을 때 할당 할 Buddy가 없다면 BITMAP\_ERROR를 발생시킴으로 error를 Detect하였다.

예를 들어 cnt가 63이라고 가정하면 size\_val은 64가 된다. 이후 idx를 0, 64, 64+64, 64+64+64 … 이런식으로 더해나가면서 할당가능한 공간을 찾게 되는 것이다.

**Free 해주기.**

먼저 Buddy System에서는 2의 진수가 아닌 페이지를 할당 요청을 해도 무조건 2의 진수 만큼의 Page를 할당해준다. 따라서 Free할때도 2의 진수가 아닌 페이지개수를 free해주라는 요청을 해도 2의 진수만큼 Free를 해줌으로써 Buddy를 합칠 수 있게끔 해야한다. 따라서 cnt를 2의 진수를 만드는 함수를 구현하였다.

**< size\_t get\_power\_of\_two(size\_t cnt) >**

|  |
| --- |
| size\_t get\_power\_of\_two(size\_t cnt){  size\_t size\_val = 1;  while(cnt > size\_val){  size\_val = size\_val \* 2;  }  return size\_val;  } |

위의 buddy\_allocation함수의 초반부와 비슷하게 구성하였다. 이때 cnt를 넣어주면 2^n-1 < cnt <= 2^n 를 만족하는 2^n 값을 반환한다.

이후 palloc\_free\_multiple함수를 수정해 주었다.

|  |
| --- |
| void  palloc\_free\_multiple (void \*pages, size\_t page\_cnt)  {  struct pool \*pool;  size\_t page\_idx;  size\_t  power\_of\_two = get\_power\_of\_two(page\_cnt);  ASSERT (pg\_ofs (pages) == 0);  if (pages == NULL || power\_of\_two == 0)  return;  if (page\_from\_pool (&kernel\_pool, pages))  pool = &kernel\_pool;  else if (page\_from\_pool (&user\_pool, pages))  pool = &user\_pool;  else  NOT\_REACHED ();  page\_idx = pg\_no (pages) - pg\_no (pool->base);  #ifndef NDEBUG  memset (pages, 0xcc, PGSIZE \* power\_of\_two);  #endif  ASSERT (bitmap\_all (pool->used\_map, page\_idx, power\_of\_two));  bitmap\_set\_multiple (pool->used\_map, page\_idx, power\_of\_two, false);  } |

위의 함수에서 입력받은 page\_cnt를 2의 진수로 만든 뒤 page\_cnt가 쓰이는 모든 곳에 power\_of\_two 값을 넣어줌으로 Buddy 블록만큼 Free 할 수 있게 하였다.

* 1. **Page Allocation Test**
     1. **Status출력하기**

먼저 status를 출력하기 위해 프로젝트 요구 조건인 palloc\_get\_status함수를 구현하였다.

이 함수는 32개의 Page를 한줄에 출력함으로 32\*11줄에 15개가 더 출력되어 367개의 페이지의 상태를 출력한다.

패브릭이(가) 표시된 사진

자동 생성된 설명초기 Kernel Pool의 Page 상태이다. 초기 Kernel에는 3개의 페이지가 이미 할당된 상태이다.

* + 1. **Test**

Test는 pa.c 파일에 있는 run\_patest()함수를 이용하여 Test를 진행하였다.

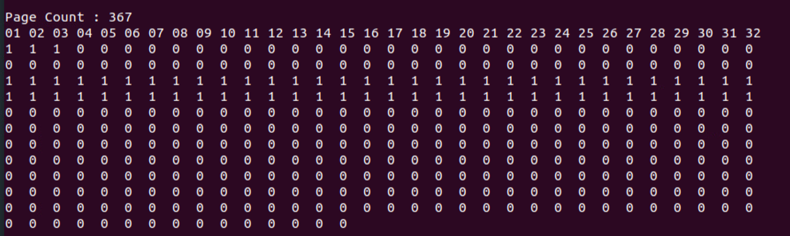
< Test 1 >

Test1 순서

1. page 63개 할당
2. page 63개 할당
3. page 63개 프리(첫번째 63개)
4. page 8개 할당
5. page 14개 할당
6. page 8개 프리
7. page 63개 프리(두번째 63개)
8. page 14개 프리

이 순서로 Test를 진행하였다.

1. 63개 페이지 할당 요청



먼저 63개를 할당 요청하게 되면 64개 크기의 Buddy를 할당하게 된다. 따라서 위의 결과화면에서 3번째줄부터 4번째 줄까지 64개 크기의 Buddy가 할당 됨을 알 수 있다.

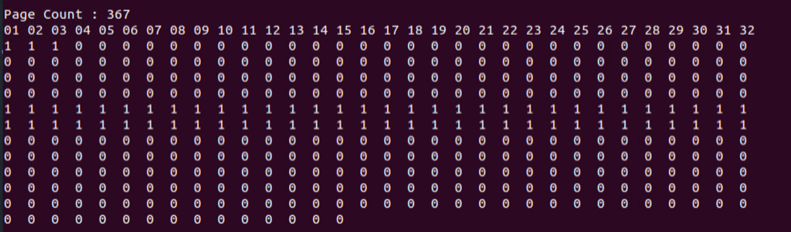
1. 63개 페이지 할당 요청

주방용품이(가) 표시된 사진

자동 생성된 설명

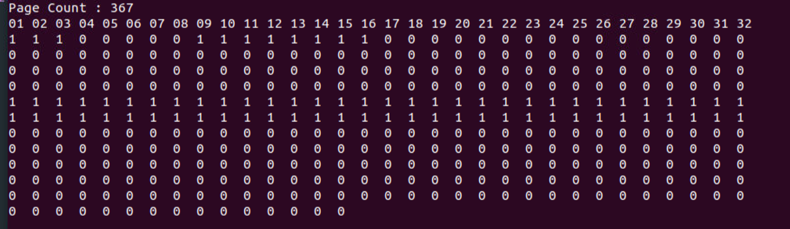
이후 또 63개를 할당 요청하게 되면 64개 크기의 Buddy를 할당하게 되므로 5,6번째 줄 64개에 대해 페이지가 할당된다.

3. 첫번째 63개 페이지 프리 요청



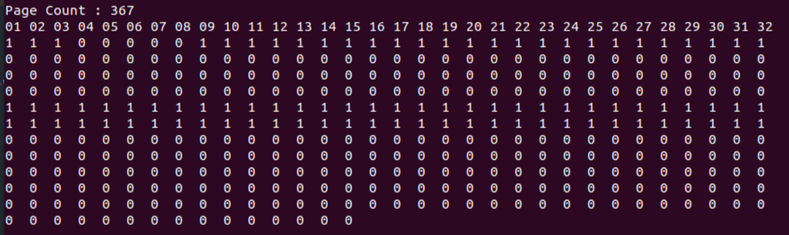
첫번째 할당했던 63개의 페이지를 프리요청한 결과이다. 3, 4번째 줄에 할당되어 있던 Buddy를 모두 프리한 상태이다.

4. page 8개 할당 요청



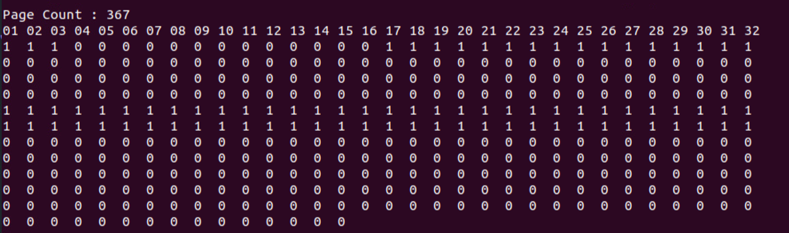
Page 8개를 할당 요청한 것이다. 이때 01~08까지는 3개의 페이지가 사용되고 있기 때문에 09 ~ 16까지 할당 됨을 확인 할 수 있다.

5. page 14개 할당 요청



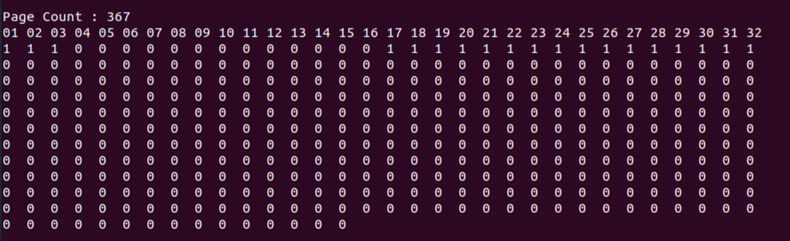
14개를 요청한다면 16개 크기의 Buddy가 할당되어야한다. 고로 17~32까지 16개 크기의 Buddy가 할당 됨을 확인 할 수 있다.

6. page 8개 프리 요청



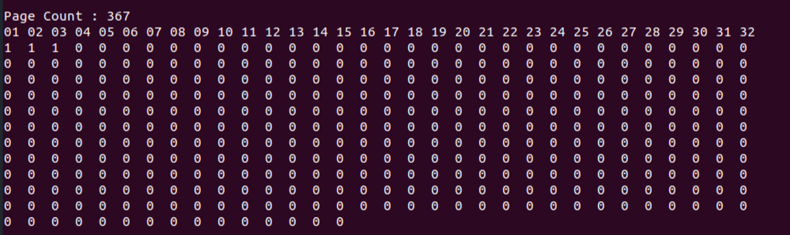
Page 8개를 프리 요청한 상태이다. 09 ~ 16까지 할당 되어있던 Buddy가 프리 됨을 확인 할 수 있다.

7. 두번째 63개의 page 프리 요청



두번째 할당 되었던 63개의 페이지를 프리 요청한 상태이다. 5, 6번째 줄에 할당되어 있던 64개짜리의 Buddy가 프리 됨을 확인 할 수 있었다.

7. page 14개 프리 요청



Page 14개를 프리요청한 상태이다. 17부터 32까지 할당되어있던 16개 크기의 Buddy를 성공적으로 Free한 것을 확인 할 수 있다.

**< Test 2 >**

Test1이 끝나고 바로 Test2를 진행하였다.

Test 2.

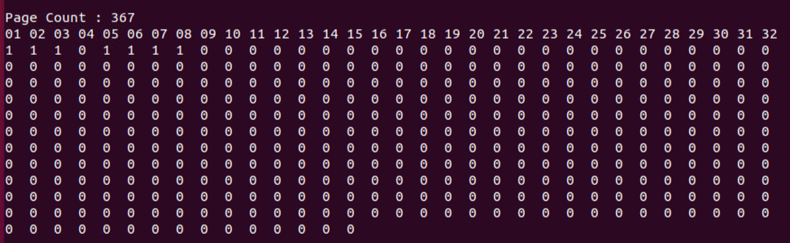
1. 3개 할당 요청

2. 30개 할당 요청

3. 62개 할당 요청

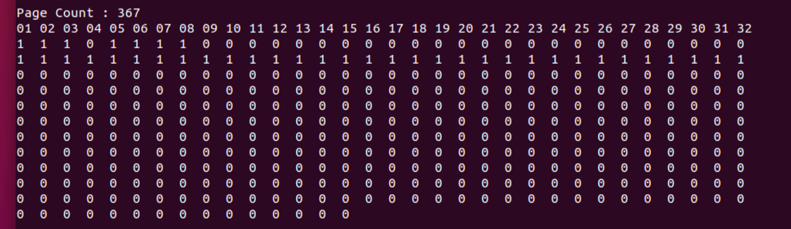
4. 120개 할당 요청

1. 3개 Page 할당 요청



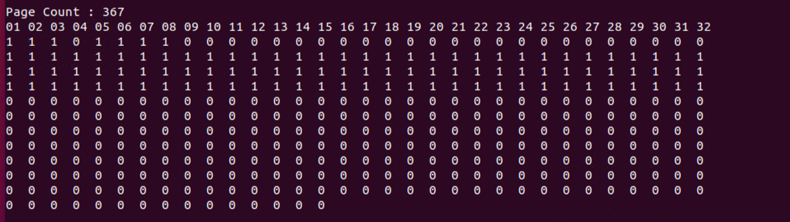
3개의 Page를 할당 요청하게 되면 4개 크기의 Buddy가 할당되어야한다. 고로 05~08까지 할당됨을 알 수 있다.

1. 30개 페이지 할당 요청



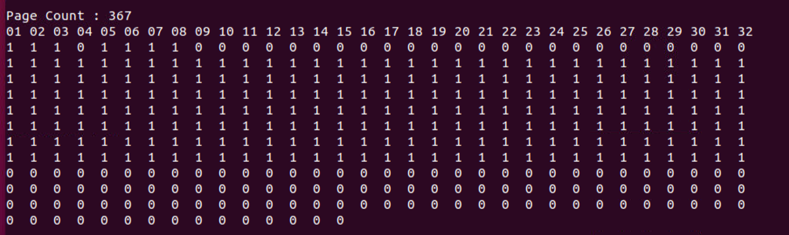
30개의 페이지를 할당 요청한 것이다. 32개의 Buddy가 둘째줄에 잘 할당 된 것을 알 수 있다.

1. 62개 페이지 할당 요청



62개의 페이지를 할당 요청한 것이다. 64개 크기의 Buddy가 3, 4번째 줄에 할당됨을 확인 할 수 있다.

1. 120개의 페이지 할당 요청

120개의 페이지를 할당 요청한 것이다. 128개 크기의 Buddy가 5~8번째 줄에 할당 된 것을 알 수 있다.

**2.1 Task Scheduling**

**2.1.1 Round-Robin스케줄러(기존방식)**

먼저 기존의 pintos에서는 쓰레드들이 라운드로빈 방식으로 구현되어있다. 기존의 코드들을 통해 흐름을 정리해보자면 thread\_tick이라는 함수를 통해 기존에 설정된 TIME\_SLICE보다 실행중인 스레드의 tick이 높다면(기존의 TIME\_SLICE는 4이다) intr\_yield\_on\_return()이라는 함수를 호출한다. 이 함수는 인터럽트를 처리하는 동안 새로운 프로세스를 yield하는 함수이다. 이때 라운드로빈 방식에서의 인터럽트는 주기적인 시간마다 timer interrupt가 발생하게 되어 매 timer tick마다 thread\_tick함수를 확인한다.

이런식으로 진행하다가 intr\_yield\_on\_return()함수를 호출하게 되면 thread\_yield라는 함수가 호출된다. 이 함수에서는 현재 실행중이던 쓰레드가 TIME\_SLICE를 넘어서 READY상태로 넘어가게 한다. 그래서 ready\_list에 그 쓰레드를 넣는다. 그 이후 schedule()이라는 함수를 호출하여 스케줄을 진행한다. 스케줄함수에서는 다음에 올 쓰레드를 next\_thread\_to\_run함수를 통해 선택하고 현재 수행중이던 쓰레드와 다음에 수행될 쓰레드를 switch\_threads함수를 통해 switching한다. next\_thread\_to\_run함수에서는 ready\_list에 제일 앞에 있는 쓰레드를 선택하여 return한다.

**2.2.2 MFQ 구현**

제일 먼저 4개의 레디큐를 만들어 주었다. 기존의 ready\_list는 static struct list ready\_list로 선언되어 있었고 이 list를

**static struct list ready\_list[4];**

로 선언함으로써 4개의 큐를 구현하였다.

**next\_thread\_to\_run (void)**

이후 next\_thread\_to\_run 함수를 수정하였다.

|  |
| --- |
| static struct thread \*  next\_thread\_to\_run (void)  {  struct thread\* next;  for(int i=0;i<4;i++){  // FQ중에서 비어있지않은 큐 체크  if(!list\_empty(&ready\_list[i])){  next=list\_entry(list\_pop\_front(&ready\_list[i]), struct thread, elem);  return next;  }  }  return idle\_thread;  } |

원래의 next\_thread\_to\_run함수에서는 ready\_list에서 제일 앞에 있는 스레드를 next 쓰레드로 설정하였다면 mfq스케줄러 방식에서는 우선순위가 제일 높은 큐에서부터 비어있는지 아닌지를 확인하며 비어 있지 않고 우선순위가 제일 높은 큐에 있는 쓰레드를 다음 쓰레드로 선정하게 된다.

그 이후 thread\_tick함수를 수정하였다. 기존의 함수에서는 4틱(TIME\_SLICE)을 넘어가게 되면 무조건 yield를 해주었지만 mfq에서는 각 우선순위 큐마다 TIME\_SLICE가 다르기 때문에 수정해 주었다.

|  |
| --- |
| void  thread\_tick (void)  {  struct thread \*t = thread\_current ();  struct thread\* will\_age;  struct list\_elem\* e;    t->total\_tick++;  …생략  // 각 스레드들의 우선순위에 따라서 실행 시간을 보장해야한다  /\* Enforce preemption. \*/  thread\_ticks++;  if((thread\_ticks >= TIME\_SLICE0) && t->priority == 0)  intr\_yield\_on\_return();  if((thread\_ticks >= TIME\_SLICE1) && t->priority == 1)  intr\_yield\_on\_return();  if((thread\_ticks >= TIME\_SLICE2) && t->priority == 2)  intr\_yield\_on\_return();  if((thread\_ticks >= TIME\_SLICE3) && t->priority == 3)  intr\_yield\_on\_return();  } |

먼저 FQ0의 TIME\_SLICE는 4, FQ1은 5, FQ2는 6, FQ3는 7의 TIME\_SLICE를 가진다. 고로 priority와 TIME\_SLICE 두 조건을 모두 만족하면 intr\_yield\_on\_return()을 호출하게 하여 쓰레드를 yield하게 했다.

그리고 mfq 스케줄링 방식에서는 만약 실행중인 쓰레드가 보장받은 TIME\_SLICE를 모두 소진하면 제일 priority가 낮은 FQ3를 제외한 FQ0 ~ FQ2에 위치한 쓰레드들은 자신이 속했던 큐보다 한단계 Priority가 낮은 FQ로 내려가게된다. 이것은 thread\_yield함수를 수정하였다.

|  |
| --- |
| void  thread\_yield (void)  {  struct thread \*cur = thread\_current ();  enum intr\_level old\_level;    ASSERT (!intr\_context ());  old\_level = intr\_disable ();  if (cur != idle\_thread){  if(cur->priority!=3){// priority를 1씩 올려서 우선순위를 낮춘다  cur->priority += 1;  }  list\_push\_back(&ready\_list[cur->priority], &cur->elem);  }  cur->status = THREAD\_READY;  schedule ();  intr\_set\_level (old\_level);  } |

빨간색으로 표시한 부분을 보면 FQ3에 속한 큐가 아닌 모든 쓰레드들은 thread\_yield함수가 호출되면 상태를 Ready로 바꾸면서 자신이 속했던 Priority보다 한단계 낮은 FQ로 들어가게끔 구현하였다.

마지막으로 Aging에 관한 부분이다. Aging은 thread\_aging 함수를 만들어 구현하였다.

|  |
| --- |
| void  thread\_aging(void){  struct thread \*t = thread\_current ();  struct thread\* not\_current\_FQ;  struct list\_elem\* e;  for(int i = t->priority + 1; i < 4 ;i++){  if(list\_empty(&ready\_list[i])){  continue; //큐가 비어있으면 건너뛴다.  }  for (e = list\_begin(&ready\_list[i]);e != list\_end(&ready\_list[i]); e = e->next){  // 해당 큐의 모든 element 탐색하면서 age추가해주기  not\_current\_FQ = list\_entry(e, struct thread, elem);  // aging해줄 스레드가 스레드가 아닐경우 break  // 만약 스레드의 상태가 BLOCKED일 경우 aging 취소  if(!is\_thread(not\_current\_FQ)) break;  if(not\_current\_FQ->status == THREAD\_BLOCKED) break;  //age를 올리고 20이 넘었으면 높은 큐로 올려  not\_current\_FQ->age = not\_current\_FQ->age + 1;  if(not\_current\_FQ->age >= 20){  if(not\_current\_FQ->priority!=0)  not\_current\_FQ->priority -= 1;  not\_current\_FQ->age = 0;//한단계 위로 올라갔으면 age 0으로 초기화 시켜야함.  }  }  }  } |

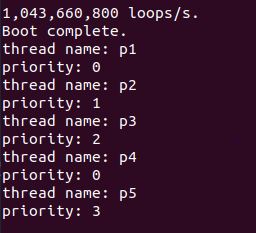
Age는 현재 실행되고 있는 쓰레드의 우선순위보다 낮은 FQ에서 대기하고 있는 쓰레드들이 실행되는 쓰레드의 실행 tick마다 age를 증가시켜 Starvation을 막는 기능을 한다. 고로 현재 실행되는 Priority보다 낮은 Queue들을 하나씩 탐색하면서 그 FQ에서 대기하는 모든 쓰레드들의 age값을 하나씩 늘려준다. 그리고 age값이 20이 넘어가는 쓰레드의 경우 Starvation을 해결하기 위해 Priority를 하나 올려주어 Priority가 높아지도록 구현하였다. 그리고 Priority가 한단계 증가하였으면 age를 0으로 초기화 시켜주었다.

**2.2 Task Scheduling Test**

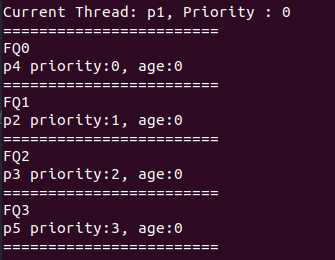
Mfq 스케쥴링의 Test를 위해 크기가 큰 반복문을 사용하였다. 처음엔 작은 반복문으로 실행시키면 눈에 보이는 실행결과가 나올줄 알았지만 작은 반복문을 사용하니 Thread들이 사용을 완료하고 Exit되어 연속적인 FQ활용을 확인하기 어려웠다. 따라서 크기가 매우 큰 반복문을 사용하여 Test를 확인하였다.

하지만 Boot가 완료되면 Print는 제대로 동작함을 확인할 수 있었다.

프로젝트에서 처음 주어진 것처럼   
mfq [p1.0]:[p2.1]:[p3.2]:[p4.0]:[p5.3]를 입력하여 진행하였다. 초기에 p1, p4는 FQ0에, p2는 FQ1에, p3는 FQ2에, p4는 FQ3에 위치해 있다.



이후 p1이 FQ0 제일 먼저 위치하기 때문에 p1부터 실행된다.



다음은 FQ0에서 기다리고 있던 p4가 실행될 때의 모습이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

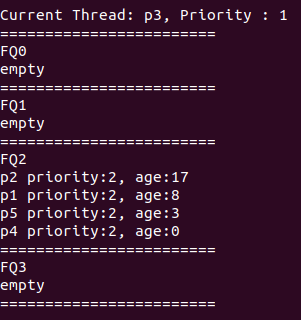
이때 p1은 할당받은 4틱만큼의 수행을 마치고 우선순위가 하나 낮아져 FQ1의 마지막에 들어가게 된다. 그리고 FQ0보다 낮은 우선순위큐에 있던 모든 쓰레드들의 age는 p1이 수행한 4틱만큼 올라감을 확인 할 수 있다.

이후 이런식으로 진행을 하다가 p3의 age가 18이 된 상황이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

FQ1에 위치하고 있던 p4가 실행될 때 p3와 p5는 한번도 실행되지 않았기 때문에 age가 4+4+5+5 가 되어 18임을 확인 할 수 있다. 바로 다음 상황을 살펴보겠다.



이 상황은 FQ3에서 대기하던 p5가 age가 20을 넘게 되어 그 다음 priority 큐인 FQ2로 올라온 모습을 확인 할 수 있다.

이런식으로 종료를 할 시점을 잡지 못한 한계가 있지만 mfq는 제대로 동작한다는 Test를 해보았다.

**3. 결론**

이론으로만 배웠던 Page Allocation, Task Scheuling에 대한 내용을 직접 코드를 통해 분석해나가다 보니 어떤식으로 구동되며, 어떤 알고리즘을 사용하는지 조금 더 명확한 실체로 다가왔던 것 같다. 처음에 코드를 분석할 때는 정말 하나도 이해가 가지 않았지만 하나하나 함수들을 따라가며 이해해보니 생각보다 쉽게 해결된 것 같기도하다. 아주 뜻깊은 프로젝트였다고 생각한다.