|  |
| --- |
| Operating Systems  Spring, 2022  School of Software, CAU  **Project #2**  **- Page Allocation and Task Scheduling –**  **[프로젝트 보고서]**  **Template**  학번: 20181594  이름: 김 형 기 |

**1. 서론**

본 보고서에서는 페이지 할당자와 스케줄러의 구현 방법과 결과에 대해 기술한다.

**1.1 Page Allocation**

Pintos는 가상 주소 공간에서 페이징을 지원한다. 페이지 할당 정책은 메모리 공간 중에서 어느 위치에 페이지를 할당할 것인지를 결정하는 정책이다. Pintos에서 제공하는 정책은 First-Fit이다. 이는 메모리의 시작 지점부터 순차적으로 검사해서 처음 발견된 할당 가능한 공간으로 결정한다. First-Fit은 구현이 간단하고 빠르지만, 시작 지점 근처에서 자주 단편화가 발생하게 되며, 큰 공간을 쪼개어 사용하게 된다.

Buddy system은 고정 크기 Partition과 가변 크기 Partition의 절충안으로, 요청 크기보다 크거나 같으면서 가장 가까운 2^N 크기의 공간을 찾아서 할당하는 방식이다. 메모리 영역이 재귀적으로 1/2 크기로 나눠질 때 같은 크기의 두 공간이 생기게 되는데 이 공간을 Buddy라고 한다. 요청된 크기보다 큰 공간을 할당하기 때문에 낭비가 생기지만, 메모리 해제 시에 Buddy와 함께 병합하기 때문에 Compaction[[1]](#footnote-1)을 하지 않고 외부 단편화에 의한 문제를 해결할 수 있어서 비용을 절약할 수 있다.

**1.2 Task Scheduling**

Pintos는 멀티스레딩 환경으로 작업의 단위는 스레드이다. 따라서 threads/thread.c에 정의된 스레드 스케줄링 정책을 변경해야 한다. Pintos가 기본으로 제공하는 교체 정책은 Round Robin(RR) 방식이다. RR 은 먼저 요청된 스레드를 먼저 처리하는 비선점 방식인 FCFS에서 실행 시간이 짧은 프로세스가 불리한 단점을 극복하고자 선점 방식을 도입한 것이다.

구현하고자 하는 Multi-level Feedback Queue(MFQ) 방식은 각 스레드는 우선순위에 따라 서로 다른 Time slice를 배정받는다. Ready 큐 역시 우선순위에 따라 나눠지며, 긴 실행 시간을 가지는 프로세스는 더 많은 Time slice를 배정받을 수 있는 하위 우선순위 큐로 이동한다. 또한 상위 우선순위 큐부터 차례대로 스케줄링이 진행되기 때문에 기아상태를 막기 위해서 Aging을 지원해야 한다. 하위 우선순위 큐에서 스케줄링 되지 못한 Tick만큼 Age는 증가한다. Age가 최댓값인 20에 도달하면 상위 우선순위 큐로 이동하여 무기한 연장을 방지할 수 있다.

구현하고자 하는 MFQ는 4단계의 우선순위 (0~3)을 가지고자 하며, 숫자가 낮을 수록 높은 우선순위를 갖는다는 뜻이다.

**2. 본론**

**1.1 Page Allocation**

**1.1.1 Pintos의 페이지 할당 정책**

Pintos에서는 페이지를 kernel\_pool, user\_pool의 두 메모리 풀에 나눠서 저장한다.

|  |
| --- |
| /\* A memory pool. \*/  struct pool  {  struct lock lock; /\* Mutual exclusion. \*/  struct bitmap \*used\_map; /\* Bitmap of free pages. \*/  uint8\_t \*base; /\* Base of pool. \*/  }; |

각 풀은 lib/kernel/bitmap.c에 정의된 bitmap 구조체로 데이터를 저장한다. 이외 구성요소는 상호배제를 위한 lock, 풀의 시작 주소를 가리키는 base가 있다.

palloc\_init에서는 Pintos의 전체 메모리 중 시작점으로부터 1MB까지의 범위는 접근하지 않으며 이를 제외한 나머지 영역을 두 풀에 1/2로 나눠서 저장한다. 그 후 init\_pool 함수를 이용해서 페이지 개수만큼의 비트맵을 생성한다. 이제 페이지 할당 관련 정보는 풀의 비트맵에서 관리할 수 있다.

1. **페이지 할당**

Pintos에서 페이지 할당을 요청하는 함수는 palloc\_get\_multiple(enum palloc\_flags, size\_t) 이다. 새로 할당할 페이지 위치를 구하기 위해서 bitmap.c에 정의된 bitmap\_scan\_and\_flip를 사용한다.

|  |
| --- |
| size\_t  bitmap\_scan (const struct bitmap \*b, size\_t start, size\_t cnt, bool value)  {  ASSERT (b != NULL);  ASSERT (start <= b->bit\_cnt);  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;  }  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;  } |

이 함수는 bitmap\_contains를 이용하여 비트맵의 start 위치 이후에 cnt 개의 페이지가 연속적으로 모두 value로 이루어져 있는지 확인한다. 만약 그런 위치를 찾으면 반환하고, 없다면 BITMAP\_ERROR를 반환한다.

핵심은 기존 Pintos는 이 부분을 이용해서 전체 비트맵 중 처음으로 찾은 할당 가능한 공간을 이용하는 First-Fit을 지원하는 것이다. 이후 다룰 구현에서는 비트맵에서 할당 가능한 공간을 Buddy system으로 찾는 방법을 제시할 것이며 이 때 bitmap\_contains를 사용할 예정이다.

1. **페이지 반환**

palloc\_free\_multiple(void \*, size\_t)는 포인터가 가리키는 페이지로부터 page\_cnt만큼 연속된 페이지를 할당 해제한다.

|  |
| --- |
| /\* Frees the PAGE\_CNT pages starting at PAGES. \*/  void  palloc\_free\_multiple (void \*pages, size\_t page\_cnt)  {  struct pool \*pool;  size\_t page\_idx;  ASSERT (pg\_ofs (pages) == 0);  if (pages == NULL || page\_cnt == 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 \* page\_cnt);  #endif  ASSERT (bitmap\_all (pool->used\_map, page\_idx, page\_cnt));  bitmap\_set\_multiple (pool->used\_map, page\_idx, page\_cnt, false);  } |

**1.1.2 Buddy System 구현**

버디 시스템은 다양한 방법으로 구현할 수 있을 것이다. Pintos는 최대 64MB의 메모리를 지원하는 경량 운영체제이므로 자원을 최소한으로 소모하면서 버디를 구현하고자 한다.

버디 시스템의 핵심은 1/2 크기로 재귀적인 공간의 분할 및 병합이다. 하지만 Pintos에서 구현된 페이지 할당 해제인 palloc\_free\_multiple은 해제할 페이지의 포인터 및 개수를 입력 받는다. 따라서 병합 과정을 위한 특별한 구조는 필요 없다고 결론 지었다. 이제 구현 관심사는 버디 시스템을 따르는 페이지 할당 정책으로 한정된다.

기존 Pintos에서 할당 가능한 페이지 위치를 찾기 위해 사용한 bitmap\_scan\_and\_flip을 대체할 함수를 구현하였다.

|  |
| --- |
| size\_t bitmap\_scan\_buddy\_and\_flip (const struct bitmap \*b, size\_t page\_cnt, bool value) {  size\_t msize = bitmap\_size(b);  size\_t find;  ASSERT (page\_cnt <= msize);  find = buddy\_find(b, page\_cnt, 0, msize-1);  if (find == BUDDY\_NOT\_FOUND)  return BITMAP\_ERROR;  bitmap\_set\_multiple(b, find, next\_pow2(page\_cnt), !value);  return find;  } |

bitmap\_scan\_buddy\_and\_flip은 해당 비트맵에서 Buddy 방식으로 할당 가능한 페이지 위치를 찾아서 반환한다. 만약 없다면 BITMAP\_ERROR를 반환하는 것은 동일하다. 분할정복으로 구현된 buddy\_find 함수를 이용해서 위치를 반환한다. buddy\_find를 시작하는 범위는 할당하고자 하는 풀의 페이지 전체이다.

만약 buddy\_find에 의해 적합한 위치를 찾았다면 비트맵에서 해당 위치로부터 2^N개의 페이지가 할당되었음을 기록한다.

|  |
| --- |
| size\_t buddy\_find(const struct bitmap \*b, size\_t page\_cnt, size\_t start, size\_t end) {  size\_t gap = end - start + 1;  size\_t upper;  // If the gap is smaller than the page size, an error is returned.  if (gap < 1)  return BUDDY\_NOT\_FOUND;  upper = next\_pow2(gap);  if ((upper / 2) < page\_cnt && page\_cnt <= upper) {    if (page\_is\_empty\_multiple(b, start, page\_cnt)  return start;  }  else {  if (upper < 2)  return BUDDY\_NOT\_FOUND;  size\_t first = buddy\_find(b, page\_cnt, start, start + upper / 2 - 1);  if (first != BUDDY\_NOT\_FOUND)  return first;  size\_t second = buddy\_find(b, page\_cnt, start + upper / 2, start + upper - 1);  if (second != BUDDY\_NOT\_FOUND)  return second;  }  return BUDDY\_NOT\_FOUND;  } |

buddy\_find 함수는 주어진 범위 내에서 요청 크기만큼 할당 가능한 공간이 있는지 찾는다. 만약 범위가 페이지 1개 크기보다 작거나, 할당 가능한 공간이 없다면 BUDDY\_NOT\_FOUND를 반환한다.

이 때 탐색 범위의 크기는 next\_pow2를 이용해 요청크기보다 크거나 같은 가장 가까운 2^N 페이지 만큼으로 결정된다. 입력된 범위 내에 모든 페이지가 사용 가능한 상태일 때 그 위치를 반환한다.

|  |
| --- |
| size\_t next\_pow2(size\_t num) {  size\_t res = 1;  while (res < num) res <<= 1;  return res;  } |

next\_pow2는 요구되는 페이지 개수 k에 대해 2^L < k <= 2^U를 만족하는 2^U를 반환한다.

|  |
| --- |
| void \*  palloc\_get\_multiple (enum palloc\_flags flags, size\_t page\_cnt)  {  struct pool \*pool = flags & PAL\_USER ? &user\_pool : &kernel\_pool;  … (생략)  lock\_acquire (&pool->lock);  page\_idx = bitmap\_scan\_buddy\_and\_flip (pool->used\_map, page\_cnt, false);  lock\_release (&pool->lock);  … (생략)  } |

palloc\_get\_multiple에서는 페이지를 할당할 위치를 bitmap\_scan\_buddy\_and\_flip으로 얻은 후에 버디 예약을 요청한다. 이 때 예약할 페이지 크기는 할당할 페이지의 크기보다 크거나 같은 가장 가까운 2^N 개이다.

|  |
| --- |
| void  palloc\_free\_multiple (void \*pages, size\_t page\_cnt)  {  … (생략)  bitmap\_set\_multiple (pool->used\_map, page\_idx, next\_pow2(page\_cnt), false);  } |

palloc\_free\_multiple에서는 전달 받은 페이지 포인터가 가리키는 페이지부터 next\_pow2(page\_cnt)의 페이지만큼을 할당 해제한다.

**1.2 Page Allocation Test**

테스트하기 위해서 페이지 풀의 상태를 출력하는 함수 palloc\_get\_status는 다음과 같이 구현하였다.

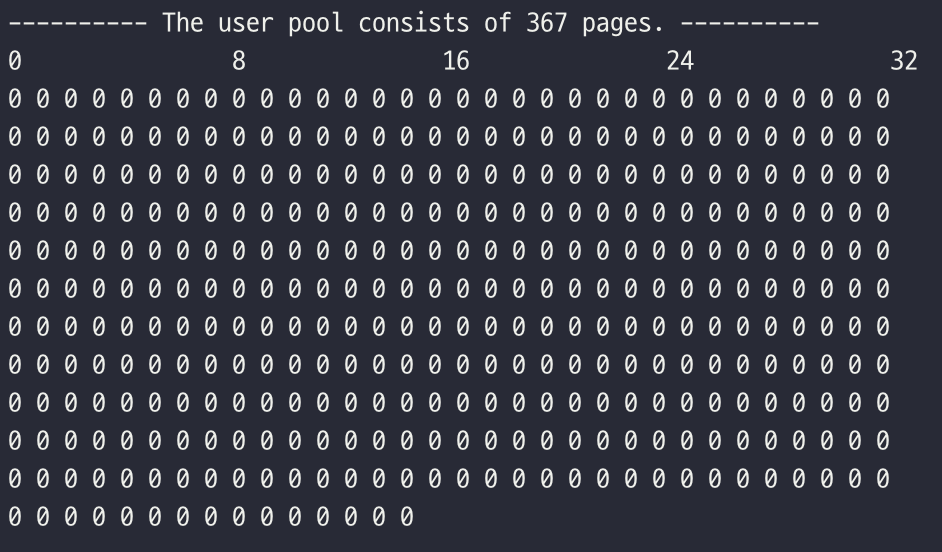
|  |
| --- |
| /\* Obtains a status of the page pool \*/  void  palloc\_get\_status (enum palloc\_flags flags)  {  struct pool \*pool = flags & PAL\_USER ? &user\_pool : &kernel\_pool;  size\_t size = bitmap\_size(pool->used\_map);  size\_t i;  char \*s = (pool == &user\_pool) ? "user" : "kernel";  printf("---------- The %s pool consists of %d pages. ----------\n", s, size);  printf("%d%16d%16d%16d%16d\n",0,8,16,24,32);    for (i = 0; i < size; i++) {  printf("%d ", !page\_is\_empty(pool->used\_map, i));  // 32 page status per line  if (i % 32 == 31)  printf("\n");  }  printf("\n\n");  } |

전달받은 flag에 따라 kernel\_pool 또는 user\_pool에 속해있는 페이지의 정보를 한 줄에 32개씩 나눠서 출력하게 된다.

|  |
| --- |
| void print\_alloc\_pages(int pages) {  printf("allocate %d pages.\n", pages);  }  void print\_free\_pages(int pages) {  printf("free %d pages.\n", pages);  }  void run\_patest(char \*\*argv)  {  // user pool test  palloc\_get\_status(PAL\_USER);  void \* p1 = palloc\_get\_multiple(PAL\_USER, 15);  print\_alloc\_pages(15);  palloc\_get\_status(PAL\_USER);  palloc\_get\_multiple(PAL\_USER, 1);  print\_alloc\_pages(1);  palloc\_get\_status(PAL\_USER);  palloc\_free\_multiple(p1, 15);  print\_free\_pages(15);  palloc\_get\_status(PAL\_USER);  palloc\_get\_multiple(PAL\_USER, 4);  print\_alloc\_pages(4);  palloc\_get\_status(PAL\_USER);  palloc\_get\_multiple(PAL\_USER, 8);  print\_alloc\_pages(8);  palloc\_get\_status(PAL\_USER);  palloc\_get\_multiple(PAL\_USER, 5);  print\_alloc\_pages(5);  palloc\_get\_status(PAL\_USER);  palloc\_get\_multiple(PAL\_USER, 40);  print\_alloc\_pages(40);  palloc\_get\_status(PAL\_USER);  void \*p2 = palloc\_get\_multiple(PAL\_USER, 100);  print\_alloc\_pages(100);  palloc\_get\_status(PAL\_USER);  palloc\_free\_multiple(p2, 100);  print\_free\_pages(100);  palloc\_get\_status(PAL\_USER);  // kernel pool test  palloc\_get\_status(0);  palloc\_get\_multiple(0, 55);  print\_alloc\_pages(55);  palloc\_get\_status(0);  palloc\_get\_multiple(0, 21);  print\_alloc\_pages(21);  palloc\_get\_status(0);  palloc\_get\_page(0);  print\_alloc\_pages(1);  palloc\_get\_status(0);  palloc\_get\_multiple(0, 100);  print\_alloc\_pages(100);  palloc\_get\_status(0);  } |

projects/pa/pa.c에서의 테스트는 위와 같이 구현하였다.

1. 초기 user pool 상태



user pool에는 할당된 페이지가 없는 상태이다.

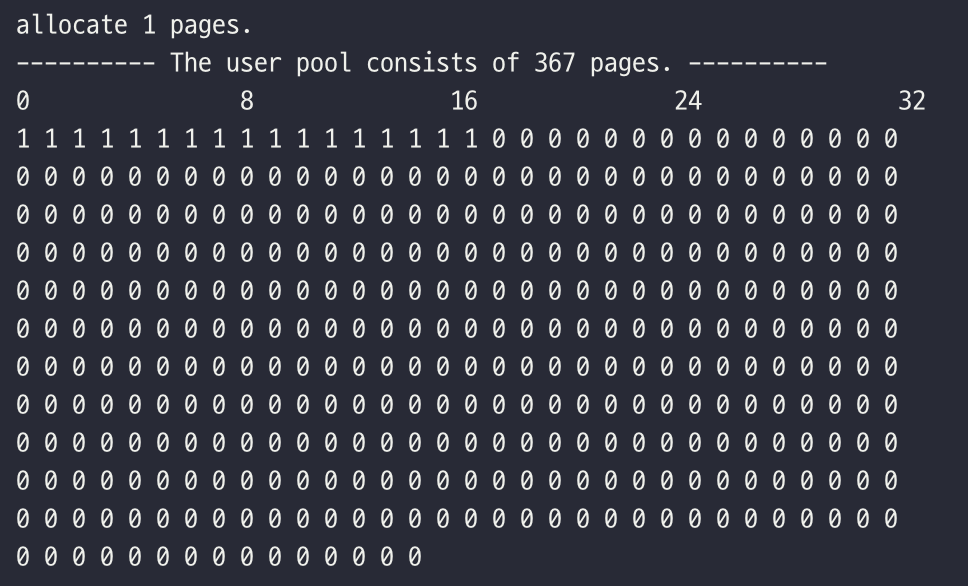
1. 15 페이지 할당

주방용품이(가) 표시된 사진

자동 생성된 설명

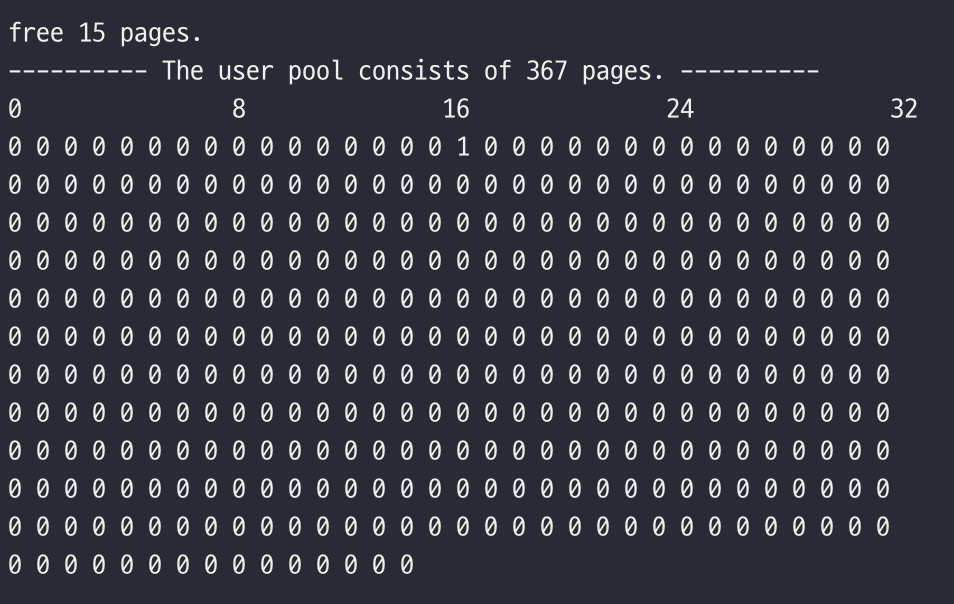
모든 공간이 비어 있으므로 페이지 인덱스 0부터 시작한다. 2^4 = 16 페이지 크기를 할당 받는다.

1. 1 페이지 할당

****

페이지 1개가 할당 가능한 공간인 16번 위치에 할당된다.

1. 15 페이지 해제



15 페이지의 해제를 요청했으며, 실제로 할당되었던 16페이지가 해제된 것을 확인할 수 있다.

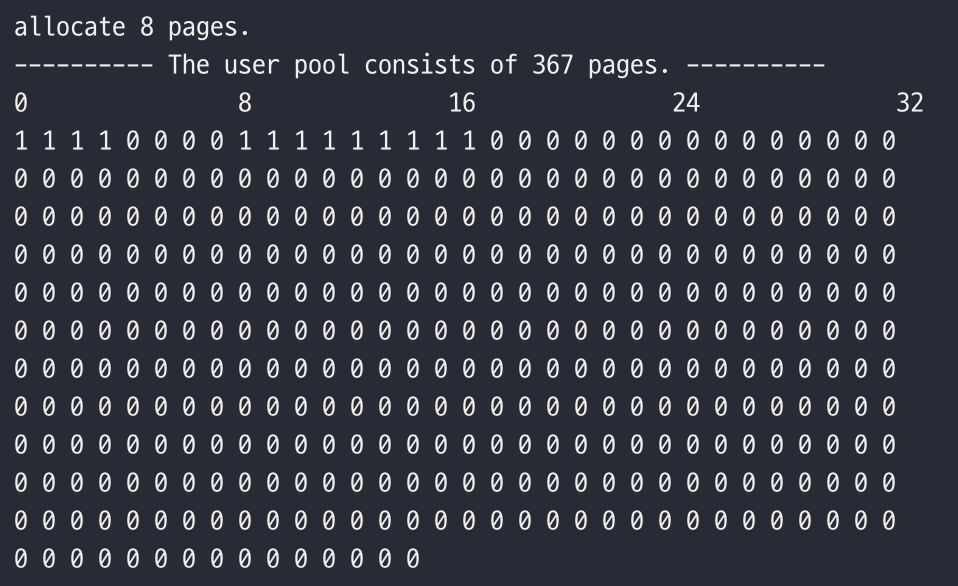
1. 4 페이지 할당

주방용품이(가) 표시된 사진

자동 생성된 설명

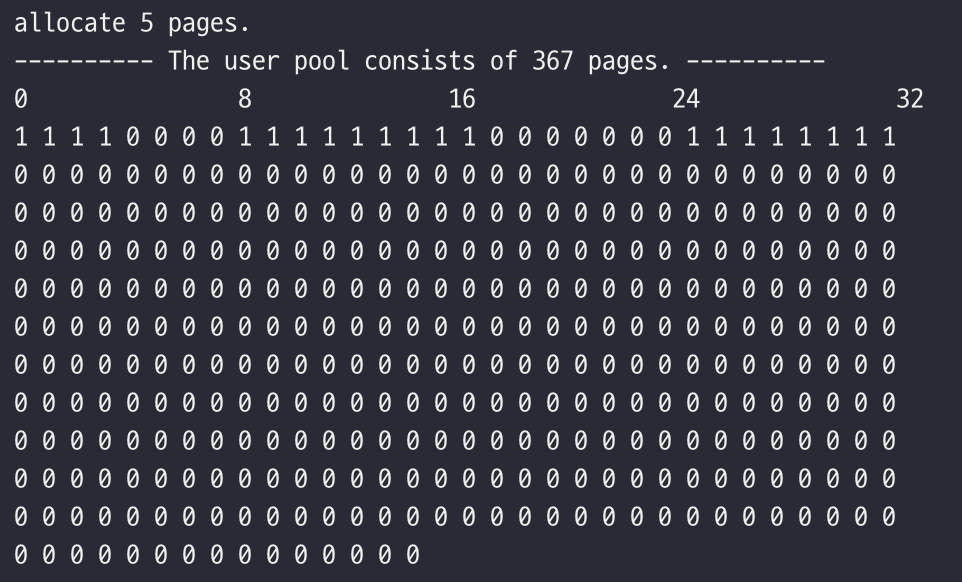
4 페이지를 할당하였다.

1. 8 페이지 할당

****

버디 시스템에 의해 4 ~ 7 번은 사용하지 않고 8 ~ 15번에 8개의 페이지가 할당된 것을 확인할 수 있다.

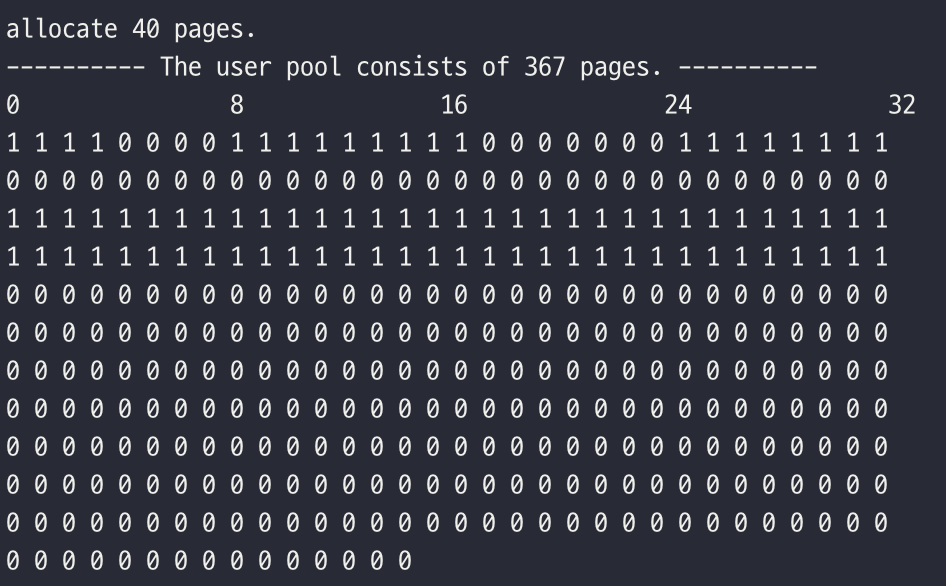
1. 5 페이지 할당



5 페이지는 2^3 = 8 페이지 크기만큼 할당되어야 한다.

16 ~ 23 번 중 16번이 할당되어 있기 때문에 24 ~ 31번 범위에서 할당된 것을 확인할 수 있다.

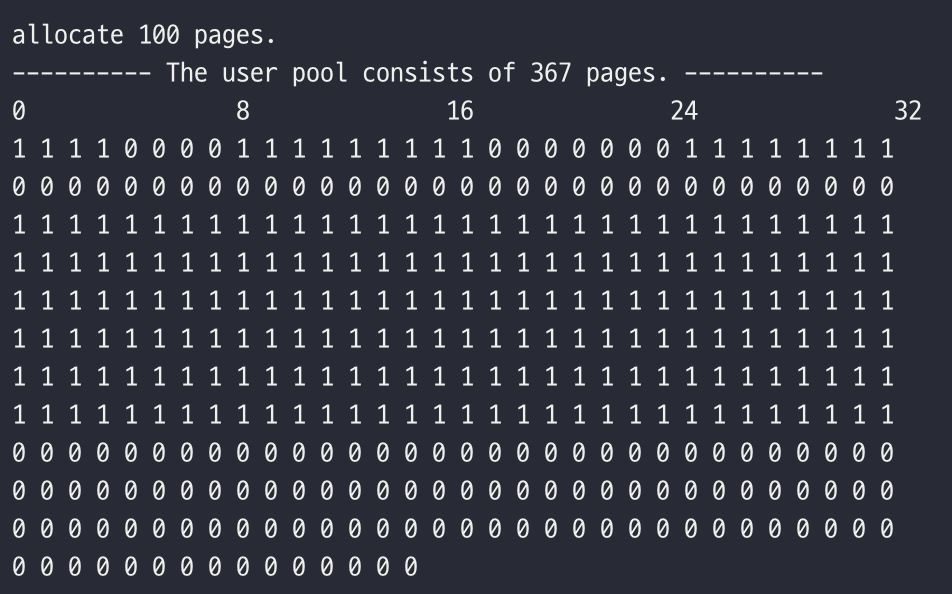
1. 40 페이지 할당



충분히 큰 페이지 할당을 테스트하고자 한다. 40 페이지는 2^6 = 64 페이지 크기만큼 할당되어야 한다.

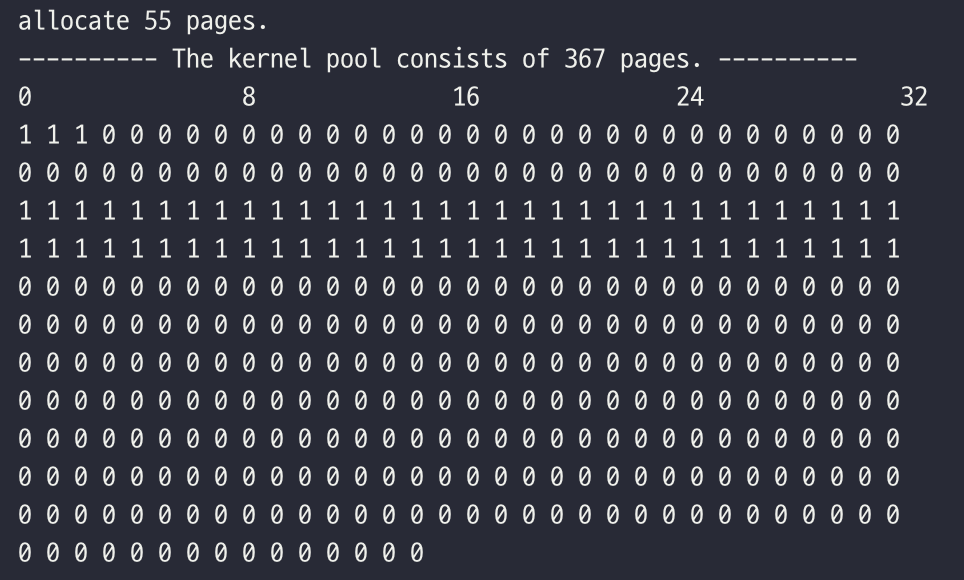
따라서 64 ~ 127번 범위에 할당된 것을 확인할 수 있다.

1. 100 페이지 할당



100 페이지는 2^7 = 128 페이지 크기만큼 할당되어야 한다. 따라서 127 ~ 255번 범위에 할당된 것을 확인할 수 있다.

1. 커널 풀에서의 테스트



커널 풀에는 시스템에 의해 기본적으로 3개의 페이지가 할당된 상태이다. 만약 한 번에 3개 페이지를 요청했다면 4 페이지가 할당되었을 것이다. 따라서 2/1 또는 1/1/1로 할당된 상태라고 유추할 수 있다.

55 페이지 할당 요청은 64 페이지만큼 할당해야 한다. 따라서 64 ~ 127 번 범위에서 할당된 것을 확인할 수 있다.

커널 풀에서도 유저 풀과 동일하게 추가적인 테스트가 존재하지만 동일하기 때문에 생략하도록 한다.

**2.1 Task Scheduling**

**2.1.1 Pintos의 스레드 스케줄링**

Pintos에서는 RR 정책의 선점을 지원하기 위해서 Tick을 사용한다. devices/timer.c에 의해서 Tick이 진행되며, 호출된 thread\_tick()에서 현재 실행 중인 프로세스가 TIME\_SLICE로 지정된 4번의 Tick을 소모하면 인터럽트를 발생시켜 다음 스레드에게 양보한다. 양보한 스레드는 Ready 상태가 되며 Ready 큐로 이동한다.

스케줄링 과정에서 다음 스레드를 결정할 때에는 next\_thread\_to\_run() 함수를 이용하며, Ready 큐에서 다음 스레드를 반환하도록 구현되어 있다. 다음 스레드가 결정되면 context switch를 진행하고 thread\_schedule\_tail(struct thread \*) 함수를 통해 다음 스레드의 상태 변화 및 이전 스레드가 완전 종료되는 상태일 경우 페이지를 반환하는 과정이 이루어진다.

Pintos에서는 모든 스레드가 Block되거나 모든 Ready 큐가 비어서 더 이상 실행할 스레드가 없는 상태를 다루기 위해서 기본적으로 idle 스레드를 운영한다. 따라서 기본으로 제공하는 main 스레드가 Block 되어도 idle 스레드에 의해 OS가 중단되지 않고 실행될 수 있으며, idle 스레드는 yield, sleep 등의 작업을 할 수 없도록 구현되어 있다.

**2.1.2 Multi-level Feedback Queue 구현**

Pintos에서 기존 Ready 큐로 사용하던 ready\_list를 feedback\_queue 0 ~ 3으로 대체하였고, 각 우선순위 별로 피드백 큐를 가지게 된다. 기존에 정의된 우선순위 범위를 PRI\_MAX는 3, PRI\_DEFAULT는 1로 변경하였다. 다만 기존 RR 정책에서는 PRI\_MAX에 가까워질 수록 높은 우선순위를 가지지만, MFQ 구현에서는 PRI\_MIN (= 0)에 가까워질 수록 높은 우선순위를 가진다는 것을 혼동하지 않아야 한다. 새로 정의한 thread\_get\_queue(int priority) 함수를 이용하여 우선순위에 해당하는 우선순위 큐를 얻을 수 있다.

기존 RR 정책에서 사용하던 TIME\_SLICE를 TIME\_SLICE\_FQ 0 ~ 3으로 대체하였고, 이는 각 우선순위 별로 다른 실행 시간을 보장한다. 새로 정의한 thread\_time\_slice(int priority) 함수를 이용하여 우선순위에 해당하는 Time slice를 얻을 수 있다.

Tick이 진행될 때마다 호출되는 thread\_tick() 에서는 Time slice를 넘었는지 확인하는데, 다음과 같이 우선순위 별로 다르게 적용되도록 변경하였다.

|  |
| --- |
| 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 >= thread\_time\_slice(t->priority))  intr\_yield\_on\_return ();  /\* aging \*/  thread\_aging(t->priority);  } |

이 때 하위 우선순위 큐에서의 기아상태를 방지하기 위한 Aging을 실행한다. thread\_aging(int) 함수의 구현은 다음과 같다.

|  |
| --- |
| /\* All threads in the feedback queue will be aged.  \*/  static void thread\_aging\_util(struct list\* list) {  ASSERT(list != NULL);  struct list\_elem \*iter;  struct thread \*t;  for (iter = list\_begin(list); iter != list\_end(list);) {  t = list\_entry(iter, struct thread, elem);  t->age += 1;  if (t->age >= AGE\_MAX) {  /\* remove from current queue \*/  iter = list\_remove(&t->elem);  t->priority = (t->priority > PRI\_MIN) ? t->priority - 1 : PRI\_MIN;    list\_push\_back(thread\_get\_queue(t->priority), &t->elem);  t->age = 0;  }  else  iter = list\_next(iter);  }  }  /\* Execute aging on feedback queues lower than the current priority.  \*/  static void thread\_aging(int cur\_priority) {  switch (cur\_priority) {  case 0:  thread\_aging\_util(&feedback\_queue\_1);  case 1:  thread\_aging\_util(&feedback\_queue\_2);  case 2:  thread\_aging\_util(&feedback\_queue\_3);  }  } |

현재 스레드의 우선순위 보다 낮은 피드백 큐에 대해 thread\_aging\_util 함수가 실행된다. 큐의 모든 스레드에 대해 Age를 1 증가시키고, 만약 최댓값인 20에 도달했다면 기존 큐에서 제거하고 우선순위를 증가[[2]](#footnote-2)시킨다. 그리고 알맞는 큐에 삽입시키고 Age를 0으로 초기화하게 된다.

thread\_yield() 함수에서는 현재 스레드가 선점에 의해 양보하여 Ready 상태로 변경되고 다음 스레드의 스케줄링을 진행한다.

|  |
| --- |
| 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) {  // get next priority  cur->priority = (cur->priority < PRI\_MAX) ? cur->priority + 1 : PRI\_MAX;  list\_push\_back (thread\_get\_queue(cur->priority), &cur->elem);  }    cur->status = THREAD\_READY;  schedule ();  intr\_set\_level (old\_level);  } |

자신에게 할당된 Time slice를 모두 소모하여 더 많은 실행 시간을 보장받을 수 있는 하위 우선순위 큐로 이동하고자 하기 때문에 우선순위를 감소하였다.

스레드가 세마포어나 뮤텍스 등의 사용으로 Block 상태에 들어가는 경우 바로 상위 우선순위 큐로 이동하고자 한다. 이를 위해서 thread\_unblock(struct thread \*) 함수에서 Block이 해제될 때 우선순위를 높이도록 구현하였다.

|  |
| --- |
| void  thread\_unblock (struct thread \*t)  {  enum intr\_level old\_level;  ASSERT (is\_thread (t));  old\_level = intr\_disable ();  ASSERT (t->status == THREAD\_BLOCKED || t->status == THREAD\_NEW);  /\* prev thread move to higher priority queue \*/  if (t->status != THREAD\_NEW)  t->priority = (t->priority > PRI\_MIN) ? t->priority - 1 : PRI\_MIN;  list\_push\_back (thread\_get\_queue(t->priority), &t->elem);  t->status = THREAD\_READY;  intr\_set\_level (old\_level);  } |

Pintos에서 처음 생성된 스레드도 Ready가 되기 전에는 Block으로 관리를 하기 때문에 생성되자마자 우선순위가 증가하는 문제가 발생할 수 있다. 따라서 스레드 상태에 THREAD\_NEW를 추가하고, 새로 생성된 스레드는 이 상태를 가지도록 하여 방지하였다.

MFQ 스케줄러가 정상적으로 동작하는지 확인하기 위해서 각 피드백 큐의 상태 및 현재 스레드의 정보를 출력하는 print\_all\_queue() 함수를 구현하였다. 또한 스레드 스케줄링의 마지막 단계에서 호출되는 thread\_schedule\_tail(struct thread \*) 함수에서 이를 호출하여 테스트하였다.

**2.2 Task Scheduling Test**

프로그램 입력에 따라 스레드를 생성하는 mfq.c에서는 각 스레드가 실행할 함수를 test\_loop으로 지정하고 있다. 이를 그대로 활용하되, 스레드들이 충분한 Tick 동안 실행되어 스케줄링을 관찰할 수 있도록 2천만번의 루프를 실행하도록 하였다.

|  |
| --- |
| #define RANGE 20000000  /\* One test\_cycle executes 20 million loops.  \*/  void test\_loop(void \*aux)  {  unsigned int a = 1;  while (1) {  a = (a + 1) % RANGE;  if (a == 0)  break;  }  } |

이외의 run\_mfqtest()는 그대로 사용하였으며, main 스레드도 루프를 통해 계속해서 실행되게 된다. 입력은 프로젝트 설명에 제시된 것을 사용하였다.

“mfq [p1.0]:[p2.1]:[p3.2]:[p4.0]:[p5.3]”

1. p1 ~ p5 생성 직전

텍스트이(가) 표시된 사진

자동 생성된 설명

메인스레드는 지속적으로 Tick을 모두 소모하여서 가장 낮은 우선순위인 3으로 실행되고 있다.

1. p1 ~ p5 생성 후

텍스트이(가) 표시된 사진

자동 생성된 설명

메인스레드의 우선순위 3에 해당하는 7 Tick이 경과되었다. Feedback Queue(FQ) 0에서 대기 중인 p1, p4 중 먼저 들어온 p1이 다음 스레드로 선택된다. 메인스레드는 우선순위에 해당하는 FQ3으로 이동한다.

1. 선점 후 p4의 fetch

텍스트이(가) 표시된 사진

자동 생성된 설명

p1의 우선순위 0에 해당하는 4 Tick이 경과되었다. p1은 모든 Time Slice를 소모하였으므로 우선순위가 감소하여 FQ1로 이동하였다. 다음 스레드는 FQ0에서 대기중인 p4가 선택되었다. p1의 우선순위 0보다 낮은 FQ 1 ~ 3의 스레드들은 Age가 경과된 Tick인 4만큼 증가하였다.

1. 선점 후 p2의 fetch

텍스트이(가) 표시된 사진

자동 생성된 설명

p4의 우선순위 0에 해당하는 4 Tick이 경과되었다.

p4는 p1과 마찬가지로 FQ1로 이동하였다. p4보다 우선순위가 낮은 FQ1 ~ 3의 스레드들은 경과된 Tick만큼 Age가 증가하였다. FQ0이 비었으므로, 다음 우선순위인 FQ1에서 p2가 다음 스레드로 선택되었다.

1. 선점 후 p1의 fetch

텍스트이(가) 표시된 사진

자동 생성된 설명

p2의 우선순위 1에 해당하는 5 Tick이 경과되었다.

p2는 Time slice를 소모하였으므로 하위 우선순위를 가지게 되었고 FQ2로 이동하였다. p2보다 우선순위가 낮은 FQ2 ~ 3의 스레드들은 경과된 Tick인 5만큼 Age가 증가되었다. FQ1의 p1이 다음 스레드로 선택되었다.

1. 선점 후 p4의 fetch

텍스트이(가) 표시된 사진

자동 생성된 설명

e)와 마찬가지로 p1의 우선순위에 해당하는 5 Tick이 경과되었고, FQ2 ~ 3의 스레드들은 Age가 5 증가하였다. 다음 스레드는 FQ1의 p4가 선택되었다.

1. Aging에 의한 우선순위 증가

텍스트이(가) 표시된 사진

자동 생성된 설명

p4의 우선순위 1에 해당하는 5 Tick이 경과되었다. FQ 2 ~ 3의 스레드들은 Age가 5만큼 증가하게 된다. p4 스레드가 2 Tick 동안 실행했을 때 FQ3의 p5와 메인스레드는 Age가 최댓값인 20 도달하게 된다. 따라서 FQ3에서 제거되고 FQ2로 이동하게 된다. Age는 0으로 초기화되고 이후 3 Tick이 추가로 경과된 상태이다.

1. p1 ~ p5 실행 완료

텍스트이(가) 표시된 사진

자동 생성된 설명

p1 ~ p5가 주어진 2천만번의 루프를 모두 실행한 뒤에는 스레드가 종료된다. 이후에는 메인스레드만 계속해서 스케줄링 될 것이다.

**3. 결론**

기존 C언어 프로젝트를 진행할 때는 malloc 같은 동적할당을 이용해 상황에 맞는 다양한 구현을 할 수 있었다. 하지만 malloc 보다 더 낮은 레벨인 palloc 내부에서 구현을 하다보니 동적할당이 불가능해서 어려운 점도 있었지만, 그동안 별 생각 없이 사용하던 기능들이 어떤 원리로 동작하고 있는지 알게 되었다. 그리고 공부할 때는 Time slice가 프로세스 클럭에 기반한 매우 낮은 수준일 것으로 이해하였는데 직접 스케줄링을 구현해보면서 OS 레벨에서 어떠한 구조로 선점을 제공하는지 이해하게 되었다.

그리고 페이징에서 생기는 단편화로 인해 1 페이지보다 큰 데이터는 페이지 할당에 실패할 수 있기 때문에 대규모 프로젝트에서는 자주 사용되는 클래스를 설계할 때는 페이지의 크기(예를 들면 4KB)보다 작게 만들도록 할 것 같다는 생각을 하였다.

1. 외부 단편화로 인해 요청한 크기를 할당할 수 없는 문제를 해결하기 위해서 프로세스들을 재배치하는 것 [↑](#footnote-ref-1)
2. 우선순위의 실제 값은 감소하지만, 낮을 수록 높은 우선순위를 갖기 때문에 증가하였다고 표현한다. [↑](#footnote-ref-2)