Operating Systems

Project #2

21조

20130398 구영서

20131961 김민조

20160462 한명지

Ⅰ. 스케줄링 알고리즘

● (1)-a: RR스케줄러 관련 분석

① Pintos의 타이머 인터럽트는 1초에 100회 발생한다. 준비 큐에 있다가 Dispatcher에 의해 실행이 시작된 스레드는 다른 스레드가 실행되기 전까지 몇 초(또는 타이머 틱) 동안 실행되는가?

➔ 프로젝트 디렉토리 threads/thread.c 의 소스코드를 보면 58열에 TIME_SLICE가 4로 정의되어 있는 것을 볼 수 있다. 그리고 122열의 thread_tick 함수를 보면 thread_ticks가 TIME_SLICE 보다 크거나 같으면 다른 스레드가 실행되도록 되어있다. 따라서 실행이 시작된 스레드가 다른 스레드가 실행되기 전까지 실행되는 타이머 틱은 평균 4틱이다.

② RR 스케줄러는 어떤 자료구조를 이용하여 준비 큐에 머무르고 있는 스레드를 관리하는가?

➔ 라운드로빈 스케줄러에서는 list를 이용하는데 threads/thread.c 의 소스코드를 보면 리스트인 ready_list와 all_list 등을 활용하여 FIFO의 방식으로 준비 큐에 머무르고 있는 스레드들을 관리함을 알 수 있다.

③ threads/thread.c 소스 코드에서 어떤 함수들이 질문 ②에서 언급된 자료구조에 접근하는가?

➔ 508열의 init_thread에서는 list_push_back 함수를 이용하여 all_list에 접근하고 232열의 thread_unblock 함수에서는 list_push_back 함수를 이용하여 ready_list에 접근한다.

④ RR 스케줄러는 threads/thread.h에 정의된 5가지의 스레드 우선순위를 사용하는가?

➔ threads/thread.h 24열에서 default priority를 의미하는 PRI_DEFAULT 가 2로 설정되어 있는데 thread_init 함수에서 스레드를 생성할 때 init_thread의 우선순위 매개변수를 PRI_DEFAULT 값으로 보내기 때문에 RR 스케줄러는 우선순위 2만 사용한다.

- **(1)-b: MFQ 스케줄러 구현**

  ① **threads/thread.c와 threads/thread.h에서 RR 스케줄러를 교체하기 위해 수정 및 추가한 부분**

  ➔

```
/* List of processes in THREAD_READY state, that is, processes
   that are ready to run but not actually running. */

static struct list ready_list[5];   // Feadback Queue
```

ready_list를 우선순위에 따라 5개의 배열로 구성하였다.

```
void
thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);

  lock_init (&tid_lock);
  for (int i = 0; i < 5; ++i){
    list_init (&ready_list[i]);
  }
  list_init (&all_list);
  list_init (&sleep_list);
```

thread_init 함수에서 우선순위에 따라 ready_list 배열의 리스트를 루프를 사용하여 초기화 해주었다.

```
585   switch(t->priority){
586     case 0:
587     t->time_slice = 6;
588     break;
589     case 1:
590     t->time_slice = 5;
591     break;
592     case 2:
593     t->time_slice = 4;
594     break;
595     case 3:
596     t->time_slice = 3;
597     break;
598     case 4:
599     t->time_slice = 2;
600     break;
601   }
```

init_thread 함수에서 switch문을 활용하여 스레드의 우선순위에 따라 각각에 해당하는 time_slice 값을 입력해주었다.

```
282  void
283  thread_unblock (struct thread *t)
284  {
285    enum intr_level old_level;
286
287    ASSERT (is_thread (t));
288
289    old_level = intr_disable ();
290    ASSERT (t->status == THREAD_BLOCKED);
291
292    list_push_back (&ready_list[t->priority], &t->elem);
293    t->status = THREAD_READY;
294    intr_set_level (old_level);
295  }
```

thread_unblock 함수에서 list_push_back 함수를 이용하여 스레드를 ready_list로 보낼 때 그 스레드의 우선순위에 해당하는 리스트로 보낸다.

```
625  static struct thread *
626  next_thread_to_run (void)
627  {
628
629    if (list_empty (&ready_list[4])){
630      if (list_empty (&ready_list[3])){
631        if (list_empty (&ready_list[2])){
632          if (list_empty (&ready_list[1])){
633            if (list_empty (&ready_list[0])){
634              return idle_thread;
635            }
636            else{
637              return list_entry (list_pop_front (&ready_list[0]), struct thread, elem);
638            }
639          }
640          else{
641            return list_entry (list_pop_front (&ready_list[1]), struct thread, elem);
642          }
643        }
644        else{
645          return list_entry (list_pop_front (&ready_list[2]), struct thread, elem);
646        }
647      }
648      else{
649        return list_entry (list_pop_front (&ready_list[3]), struct thread, elem);
650      }
651    }
652    else{
653      return list_entry (list_pop_front (&ready_list[4]), struct thread, elem);
654    }
655  }
```

next_thread_to_run함수에서 다음 실행을 넘겨받을 스레드를 선택하기 위해 높은 우선순위의 리스트부터 차례로 탐색한다. ready_list[4] 부터 ready_list[0]까지 탐색하는데 만약 리스트가 비어 있으면 다음 우선순위 리스트로 탐색을 진행한다. 진행 중에 비어 있지 않은 리스트가 있다면 해당 리스트의 첫번째 스레드를 list_pop_front 함수를 사용하여 반환한다.

```
407  /* Yields the CPU.  The current thread is not put to sleep and
408     may be scheduled again immediately at the scheduler's whim. */
409  void
410  thread_yield (void)
411  {
412    struct thread *cur = thread_current ();
413    enum intr_level old_level;
414    ASSERT (!intr_context ());
415
416    old_level = intr_disable ();
417    if (cur != idle_thread){
418      cur->priority--;
419      cur->time_slice++;
420      if(cur->priority < 0){
421        cur->priority = 0;
422        cur->time_slice = 6;
423      }
424      list_remove(&cur->elem);
425      list_push_back (&ready_list[cur->priority], &cur->elem);
426
427    }
428    cur->status = THREAD_READY;
429    schedule ();
430    intr_set_level (old_level);
431  }
```

thread_yield 함수에서 현재 스레드가 다음 스레드로 교체되기 전에 현재 실행 중인 스레드 cur의 priority를 낮춰주고 그에 맞춰 time_slice도 알맞게 올려준다. 이때 만약 cur의 우선순위가 0보다 작아진다면 0보다 작은 우선순위는 존재하지 않기 때문에 우선순위와 시간 할당량 모두 그대로 0으로 유지시켜 준다. 그리고 나서 list_remove 함수를 이용하여 cur을 기존 우선순위 리스트에서 삭제하고 list_push_back 함수를 이용하여 바뀐 우선순위에 해당하는 리스트에 추가한다. 마지막으로 schedule 함수로 현재 스레드와 다음 스레드를 교체해준다.

② 준비 큐에 머무르고 있는 스레드를 관리하기 위해 사용한 자료구조

➔ 기존과 마찬가지로 기본 구조는 list를 쓰되 우선순위에 따라 리스트가 총 5개가 필요하므로 배열 구조를 5개 추가적으로 구성하였다.

③ 스레드 Aging 구현 방법

➔

```
166  static bool thread_aging(struct thread *t){
167    thread_foreach(&thread_age_up, t);
168    return true;
169  }
```

```
433  /* Invoke function 'func' on all threads, passing along 'aux'.
434     This function must be called with interrupts off. */
435  void
436  thread_foreach (thread_action_func *func, void *aux)
437  {
438    struct list_elem *e;
439
440    ASSERT (intr_get_level () == INTR_OFF);
441
442    for (e = list_begin (&all_list); e != list_end (&all_list);
443         e = list_next (e))
444      {
445        struct thread *t = list_entry (e, struct thread, allelem);
446        func (t, aux);
447      }
448  }
```

기본적으로 thread_aging 함수로 구현한다. 리스트를 처음부터 끝까지 탐색하는 tread_foreach 함수에 thread_age_up 함수와 현재 실행 스레드 t를 매개체로 보낸다.

tread_foreach 함수에서 리스트의 처음 스레드부터 마지막 스레드까지를 나타내는 t와 실행 스레드 aux를 매개로 thread_age_up 함수를 실행한다.

```
171  static void thread_age_up(struct thread * t, void * aux){
172      struct thread *cur = aux;
173      struct list_elem *e;
174      if(t->priority < cur->priority){
175        ++t->age;
176        if(t->age >= 20){
177          for (e = list_begin (&ready_list[t->priority]); e != list_end (&ready_list[t->priority]);
178            e = list_next (e)){
179            struct thread * t2 = list_entry (e, struct thread, elem);
180            if(!strcmp(t->name,t2->name)){
181              list_remove(&t2->elem);
182            }
183          }
184          t->time_slice--;
185          t->priority++;
186          t->age = 0;
187          list_push_back (&ready_list[t->priority], &t->elem);
188        }
189      }
190  }
```

현재 실행중인 스레드보다 우선순위가 낮은 스레드들은 모두 age를 증가시킨다. 만약 age 값이 20 이상이 된 스레드가 있으면 그 스레드의 우선순위를 증가시키고 우선순위에 맞게 시간 할당량도 줄여준다. 또한 age를 다시 0으로 바꾼 후 새로운 우선순위의 리스트에 추가해준다.

Age가 20인 스레드를 기존 우선순위 리스트에서 삭제하는 과정은 for loop문으로 나타나 있다. 모든 리스트를 처음부터 끝까지 탐색하는데 age가 20인 스레드를 발견하면 그 스레드의 해당 우선순위 리스트로 가서 같은 이름의 스레드를 찾아 list_remove 함수를 이용해 삭제해준다.

④ MFQ 스케줄러의 동작을 확인하기 위한 테스트 방법

➔

```
130  void
131  thread_tick (void)
132  {
133    struct thread *t = thread_current ();
134    struct list_elem *e;
135
136
137    /* Update statistics. */
138    if (t == idle_thread)
139      idle_ticks++;
140  #ifdef USERPROG
141    else if (t->pagedir != NULL)
142      user_ticks++;
143  #endif
144    else
145      kernel_ticks++;
146
147    /* Enforce preemption. */
148    if (++thread_ticks >= t->time_slice && thread_aging(t)){
149      intr_yield_on_return ();
150
151      for (int i = 0; i < 5; ++i)
152      {
153        printf("queue : %d list size : %d\n", i, list_size(&ready_list[i]));
154        for (e = list_begin (&ready_list[i]); e != list_end (&ready_list[i]);
155             e = list_next (e)){
156          struct thread * t2 = list_entry (e, struct thread, elem);
157          printf("queue : %d name : %s age : %d\n", i, t2->name, t2->age);
158        }
159      }
160
161    }
162  }
```

tread_tick 함수 안에 진행상황을 출력하는 코드를 추가하였다. 기본적인 테스트 코드는 샘플 코드와 같다.

```
4_Thread 0 got tick.
4_Thread 0 got tick.
queue : 0 list size : 5
queue : 0 name : queue 0의 0번 age : 1
queue : 0 name : queue 0의 1번 age : 1
queue : 0 name : queue 0의 2번 age : 1
queue : 0 name : queue 0의 3번 age : 1
queue : 0 name : queue 0의 4번 age : 1
queue : 1 list size : 5
queue : 1 name : queue 1의 0번 age : 1
queue : 1 name : queue 1의 1번 age : 1
queue : 1 name : queue 1의 2번 age : 1
queue : 1 name : queue 1의 3번 age : 1
queue : 1 name : queue 1의 4번 age : 1
queue : 2 list size : 5
queue : 2 name : queue 2의 0번 age : 1
queue : 2 name : queue 2의 1번 age : 1
queue : 2 name : queue 2의 2번 age : 1
queue : 2 name : queue 2의 3번 age : 1
queue : 2 name : queue 2의 4번 age : 1
```

실행하면 이런 식으로 출력되며 MFQ가 제대로 동작함을 알 수 있다.

Ⅱ. 메모리 할당

- **(2)-a: 메모리/페이지 할당 관련 분석**

  ① **Pintos 메모리 시스템의 기본 페이지 크기는 몇 바이트인가?**

➔ Pintos 메모리 시스템의 기본 페이지 크기는 PGSIZE에 정의되어 있는 값으로 4KB이다. 이는 vaddr.h의 소스코드 20열에서 확인할 수 있다.

② **threads/malloc.c에서 기본 페이지보다 작은 크기의 메모리 영역을 할당하기 위해 어떤 방법이 사용되고 있는가?**

➔ malloc 함수에서 free_list가 비어 있는지 검사하고 비어 있으면 새로운 arena를 만들고 초기화한 후 arena에 있는 블록을 free_list에 추가한다. 만약 free_list가 비어 있지 않다면 free_list에서 쓸 수 있는 블록을 반환하여 사용한다.

③ **threads/palloc.c와 lib/kernel/bitmap.c 내의 어떤 함수에 페이지 할당 알고리즘 (First Fit)이 구현되어 있는가?**

➔ palloc_get_multiple 함수에서 bitmap_scan_and_flip 함수를 이용하여 페이지 인덱스를 반환 받는데 bitmap_scan_and_flip 함수에서는 우선 bitmap_scan 함수를 이용하여 비트맵의 시작부터 탐색하며 페이지의 개수만큼 인덱스를 반환한다. 시작부터 탐색했기 때문에 First Fit 알고리즘을 구현한 것이 되며 비트맵을 탐색하는 과정에서 조건에 적합한 페이지가 있는지 확인할 때에는 bitmap_contain 함수를 활용한다.

● (2)-b: 메모리/페이지 할당 알고리즘 구현

① **threads/palloc.c와 lib/kernel/bitmap.c 에서 페이지 할당 알고리즘 구현을 위해 수정 및 추가한 부분**

1. Next Fit

```
328  size_t
329  bitmap_scan (const struct bitmap *b, size_t start, size_t cnt, bool value)
330  {
331
332    ASSERT (b != NULL);
333    ASSERT (start <= b->bit_cnt);
334
335    static size_t recent = 0;      //가장 최근에 할당된 위치 (next fit 알고리즘을 위한 변수)
336    static size_t check_cnt = 0;    //해당 함수가 실행되는 횟수
337    static size_t check_size = 0;   //할당되는 공간의 크기
338
```

Next fit 구현을 위해 필요한 가장 최근에 할당 받은 위치를 저장하는 recent 변수를 추가하였다.

```
355     else if(pallocator == 1)    //next fit 메모리 할당 알고리즘 호출(-ma=1 입력시 nextfit 실행)
356     {
357       for(i = recent; i <= last; i++) { //가장 최근에 배치된 메모리 위치에서부터 마지막 위치까지 검색
358         if (!bitmap_contains (b, i, cnt, !value))
359         {
360           recent = i;
361           return i;
362         }
363       }
364         for (i = start; i<= recent; i++)  { //메모리의 시작점인 위치부터 가장 최근에 배치된 메모리 위치까지 검색.
365                         //맨 마지막까지 할당받으면 nextfit은 직전에 할당했던곳 부터 할당하는데 마지막까지 할당하는 부분이 최근으로 참조했던 부분
366         if (!bitmap_contains (b, i, cnt, !value))
367
368           {
369             recent = i;
370             return i;
371           }
372       }
373     }
```

1번을 호출한 경우 next fit이 실행되도록 하는 코드를 추가했다. 처음부터 마지막까지 메모리를 탐색하는 first fit과 달리 가장 최근에 할당 받은 위치인 recent부터 마지막까지 적합한 위치를 탐색하고 만약 마지막까지 적합한 위치가 없으면 start부터 recent까지 다시 탐색하며 적합한 위치를 찾는다. 적합한 위치를 찾으면 그 위치를 반환한다.

2. Best Fit

```
374     else if(pallocator == 2)      //best fit 메모리 할당 알고리즘 호출(-ma=2 입력시 bestfit 실행)
375     {
376       size_t idx = 0;       //fit될 수 있는 인덱스
377       size_t tempIdx = 0;     //그 다음의 인덱스
378       bool space = 0;       //이 변수에 할당가능한 인덱스들이 저장.
379       size_t size = b->bit_cnt;   //공간 최대크기
380
381       for(i= start; i <= last; i++){        //메모리 시작점 처음부터 끝까지 검색
382         if(bitmap_test (b, start + i) == false) //만약 i bit가 0이면
383         {
384           if(space == 0)    //할당받을수 있는 첫번째 영역에 인덱스를 만남
385           {
386             tempIdx = i;
387             if(idx == 0) idx = tempIdx;   // 뒷 부분에 할당된 공간이 없을 수도 있기 때문에 미리 인덱스를 저장.
388           }
389           space ++;       //공간에 할당가능한 그 다음의 인덱스가 증가
390         }
391         else  //만약 i bit가 1이면
392         {
393           if((space >= cnt)&&(space < size))
394           {
395             size = space;
396             idx = tempIdx;        //인덱스는 현 인덱스로 저장
397           }
398           space = 0;      //1을 만나면 space를 0으로 초기화
399         }
400       }
401       return idx;   //인덱스 반환
402     }
```

2번을 호출한 경우 best fit이 실행되도록 하는 코드를 추가한다. Fit 될 수 있는 인덱스를 저장하는 변수 idx와 그 다음 인덱스를 저장하는 변수 tempIdx, 요구되는 변수에 할당 가능한 인덱스들의 수를 저장하는 변수 space와 가능 공간의 최대 크기를 의미하는 size 변수를 추가적으로 선언하였다.

탐색은 메모리의 시작점부터 마지막 부분 까지 탐색하며 빈 공간을 찾고 공간에 할당 가능한 그 다음 인덱스를 계속하여 space에 저장한다. 만약 뒷 부분에서 할당된 공간이 없는 경우 반환할 때를 위하여 미리 tempIdx에 i값을 저장한다. 만약 i가 1이 되면 space 값을 요구되는 값과 비교

하여 크거나 같고, 동시에 공간 최대 크기인 size 보다 작으면 반환할 인덱스에 현 인덱스를 저장하고 space를 다시 0으로 초기화 해준다. 과정을 반복하다가 마지막 인덱스를 반환한다.

3. Buddy System

```
8  /* Bitmap abstract data type. */
9  static size_t tree[512];          // buddy 시스템에서 2의 제곱수로 사용중인 영역을 색칠하기 위하여 선언.
```

버디 시스템에서 2의 제곱수로 사용중인 영역을 표기하기 위해 선언하였다.

```
293 void tree_print(size_t * tree){       // 사용 중인 영역을 확인하기 위하여 선언한 테스트 코드.
294   printf("tree-------------------------\n");
295   for(int i = 0 ;i<16 ; i++){
296     for(int j =0 ; j<32; j++){
297       printf("%d",tree[(i*32) + j]);
298     }
299     printf("\n");
300   }
301 }
302 bool
303 buddy_remove (size_t start, size_t cnt) // free를 할때 cnt와 할당 받았던 index가 넘어온다.
304 {
305   size_t binary_size = 1;             // 할당을 2의 제곱수 만큼 했기 때문에 마찬가지로 cnt보다 큰 2의 제곱수를 구한다.
306     while(1){
307       binary_size*=2;
308       if(cnt <= binary_size)
309       break;
310     }
311
312     if(cnt == 1){
313       binary_size = 1;
314     }
315
316   for(int j = start; j< start + binary_size; j++){  // 다시 그 영역을 사용할 수 있으므로 색칠한 영역을 다시 초기화한다.
317       tree[j] = 0;
318   }
319
320 }
```

Tree_print 함수를 추가하여 사용 중인 영역을 출력할 수 있도록 하였다. 그리고 buddy_remove 함수를 추가하여 할당 받았던 영역이 반환되는 경우 시행하도록 한다. 매개 변수로 free를 시행할 때 cnt와 할당 받았던 인덱스를 받는데 while 문에서 계속해서 2를 곱하며 요구되는 크기인 cnt와 가장 근접하면서 큰 조건을 만족하는 binary_size를 찾는다. 그리고 영역이 반환되며 다시 그 영역을 할당할 수 있도록 색칠되어 있던 부분을 다시 0으로 바꿔주며 초기화한다.

② 각 페이지 할당 알고리즘의 구현 방법

1. Next Fit

➔ 1번을 호출하면 가장 최근에 배치된 메모리의 위치를 의미하는 recent함수를 사용하여 recent 함수부터 마지막까지 탐색하며 적합한 공간을 찾는다. 적합한 위치를 찾으면 그 인덱스를 반환한다. 만약 마지막까지 적합한 공간을 찾지 못하면 다시 처음인 start로 돌아가 recent까지 탐색한다. 마찬가지로 적합한 공간을 발견하면 그 해당 인덱스를 반환한다. 이때 적합한 공간을 찾는 함수는 bitmap_contains 함수를 활용한다.

2. Best Fit

➔ 2을 호출 받게 되면 best fit을 실행한다. 우선 메모리의 시작점부터 마지막까지 탐색하며 bitmap_test 함수를 사용하여 빈공간을 찾는다 가용 공간인 빈공간을 찾으면 할당가능한 인덱스를 space에 저장한다. 이때 반환되는 인덱스를 저장하는 변수는 idx 인데 할당 가능한 공간이 없을 경우를 대비해 미리 tempIdx를 저장한다. 다시 돌아와서, 루프를 돌면서 구해진 space 값이 요구되는 크기인 cnt보다 크거나 같고 공간의 최대 크기인 size 보다 작으면 idx 값에 현재 인덱스를 저장한다. 그리고 이때의 space 값으로 size 값을 업데이트 시켜주며 이후에 구해질 space와 다시 비교할 수 있도록 한다. 이 과정을 반복하며 cnt를 수용할 수 있으며 동시에 가장 작은 공간의 인덱스를 idx에 저장하고 이 값을 반환한다.

3. Buddy System

➔ bitmap에 할당을 할때 cnt(사용자가 필요로 하는 page 수)만큼 start index부터 1이 된다. 하지만 buddy system은 2의 지수만큼 1이 되야한다. 즉 cnt + 알파 만큼 할당을 해주는 시스템이다. 따라서 tree라는 bitmap 사이즈와 같은 size_t 배열을 만들었다. 만약 cnt가 5이라면 8만큼 start index 부터 8만큼 tree에 1로 색칠한다. bitmap_contains의 함수는 for문 0~512 만큼 돌면서 start index부터 원하는 cnt만큼 할당할 수 있는 영역의 첫 index를 반환하는 함수이다. 이 함수와 2가지 조건을 걸어서 index를 반환하게 해주었다. 첫 번째 조건은 bitmap_contains의 함수에서 받아온 index가 tree에서 1이 아닌 부분에 해당하면 첫 번째 조건을 만족시킨다. 두 번째 조건은 index%(cnt+ 알파) == 0 을 만족하면 두 번째 조건을 만족시킨다. 두 번째 조건에 대해서 자세히 말하자면 cnt 보다 큰 근접한 2의 지수값을 i와 %연산자를 했을 때 0이면 이는 반대쪽의 영역을 침범하지 않는 index라고 할 수 있다. 예를 들어서, cnt+ 알파가 8이라고 한다면 index%8 == 0 인 조건이다. 이는 0, 8 , 16, 24 ... 8의 배수의 index만을 허용하게 되는 것이다. 그렇게 되면 원하는 buddy 크기의 start index 알 수 있게 된다.

모든 조건을 예를 들어서 설명하자면, 사용자가 원하는 cnt가 31이다. 이 값보다 크고 근사한 2의 지수 값은 32이다. 즉, 32만큼의 크기를 할당하면 될 것이다. 0부터 512까지 for문을 돌면서 32만큼의 크기를 할당받을 수 있는 index를 bitmap_contains로 찾는다. 이때 index의 조건은 32,64,96,128 ... 의 index만 가능하다. 또한, 이 32의 배수 index들이 tree[index] == 0 이여야 한다. 이때, 이 index를 반환해준다.

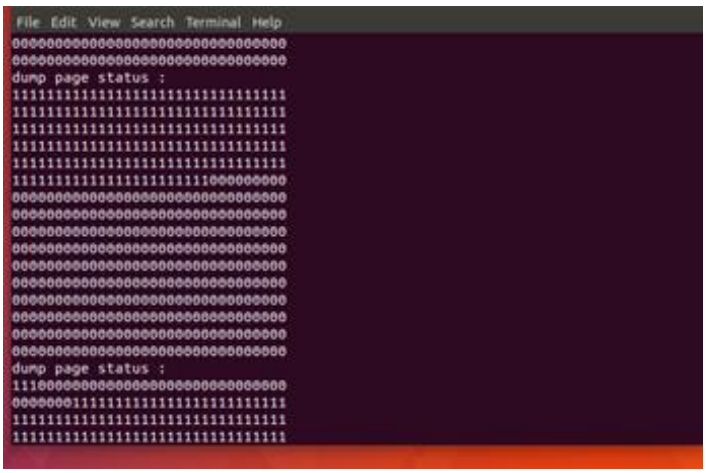
free도 마찬가지이다. free를 할때 start index와 cnt를 받아온다. 이때 cnt를 마찬가지로 2의 지수 값으로 만들고 tree[index]부터 tree[index + 2의 지수 값] 만큼 0으로 만들어서 그 영역을 사용할 수 있게 한다.

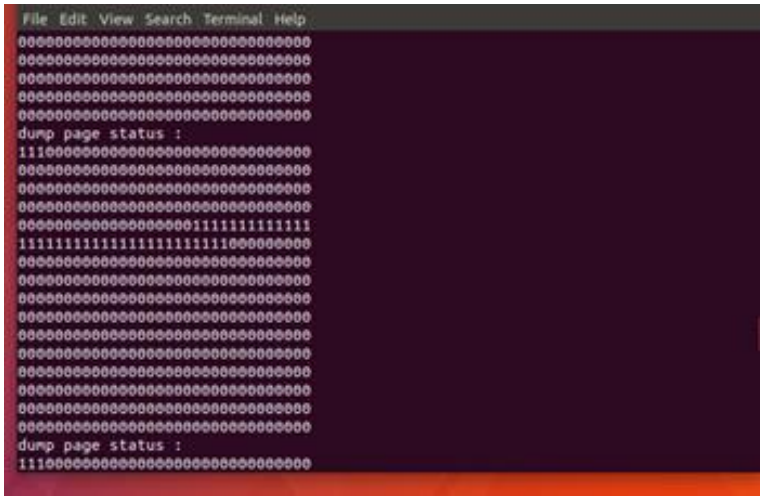
③ **페이지 할당 알고리즘의 동작을 확인하기 위한 테스트 방법**

각 메모리 할당 알고리즘을 테스트하기 위해서 memalloctest.c 파일에서 테스트 샘플 코드를 임의로 만들고 테스트를 했다. ../../utils/pintos -ma=(0,1,2,3) memalloc 명령어를 입력하고 각 알고리즘에 맞는 숫자를 넣고 실행한다. 이미 구현되어 있던 FirstFit은 0, NextFit은 1, BestFit은 2, BuddySystem은 3으로 설정햇다.

NextFit은 아래 테스트코드로 5번째까지 순차적으로 할당해준 후 순차적으로 free 해준다. 직전까지 할당한 곳부터 나머지를 할당해주고 free해주면서 NextFit이 실행된다.

```
if(pallocator == 1)
{
    size_t i;
    char* dynamicmem[11];

    for (i=0; i<5; i++){
        dynamicmem[i] = (char *) malloc (145000);
        memset (dynamicmem[i], 0x00, 145000);
        printf ("dump page status : \n");
        palloc_get_status(0);
    }

    thread_sleep (100);

    for (i=0; i<5; i++){
        free(dynamicmem[i]);
        printf ("dump page status : \n");
        palloc_get_status(0);
    }

    for (i=7; i<10; i++){
        dynamicmem[i] = (char *) malloc (145000);
        memset (dynamicmem[i], 0x00, 145000);
        printf ("dump page status : \n");
        palloc_get_status(0);
    }

    for (i=7; i<10; i++){
        free(dynamicmem[i]);
        printf ("dump page status : \n");
        palloc_get_status(0);
    }

}
```

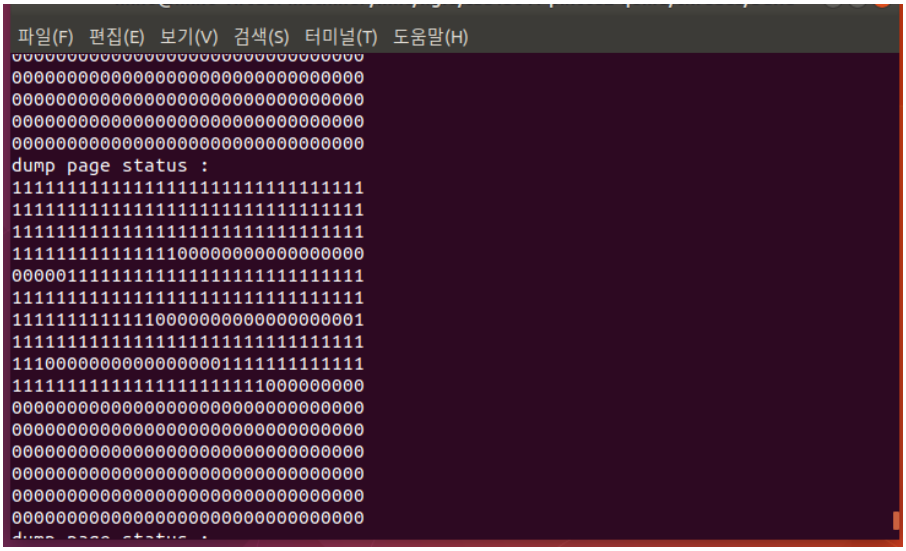


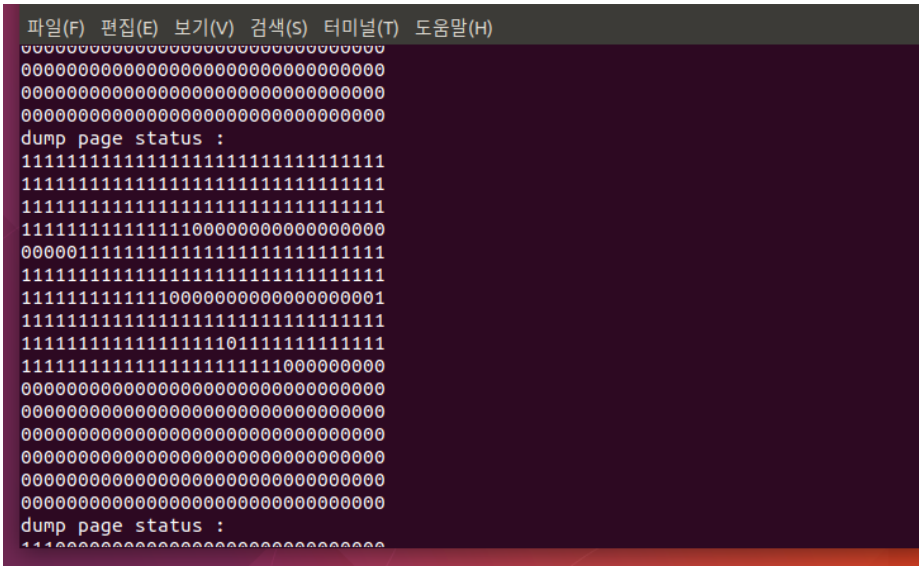


BestFit은 10개로 나뉘어 할당되는데 3번째, 6번째, 8번째 부분을 크기를 다르게 하여 먼저 할당한 후에 10개 모두 할당이 끝이 나고 다시 3,6,8번째의 메모리를 해제하여서 테스트하고자하는 메모리가 어디 영역에 bestFit으로 할당이 되는지 테스트 해보고자 했다.

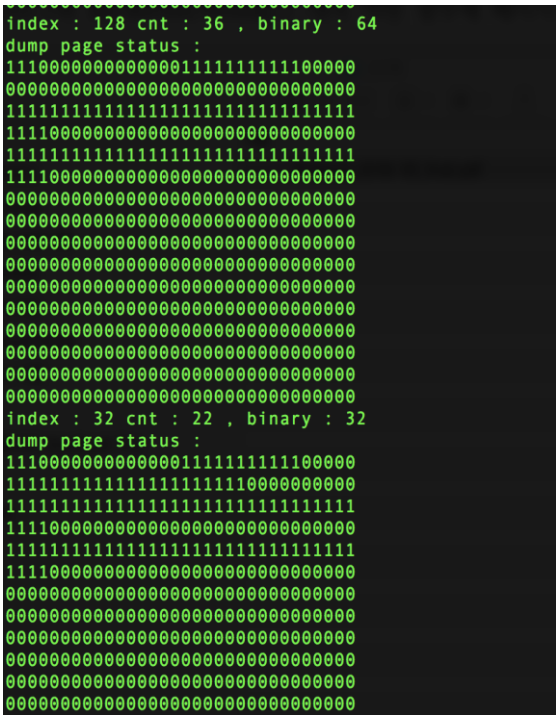
```
dynamicmem[7] = (char *) malloc (145000);
memset (dynamicmem[7], 0x00, 145000);
printf ("dump page status : \n");
palloc_get_status (0);

dynamicmem[8] = (char *) malloc (65000);
memset (dynamicmem[8], 0x00, 65000);
printf ("dump page status : \n");
palloc_get_status (0);

dynamicmem[9] = (char *) malloc (145000);
memset (dynamicmem[9], 0x00, 145000);
printf ("dump page status : \n");
palloc_get_status (0);

free(dynamicmem[3]);
printf ("dump page status : \n");
palloc_get_status (0);


free(dynamicmem[6]);
printf ("dump page status : \n");
palloc_get_status (0);

free(dynamicmem[8]);
printf ("dump page status : \n");
palloc_get_status (0);


dynamicmem[10] = (char *) malloc (60000);
memset (dynamicmem[10], 0x00, 60000);

printf ("dump page status : \n");
palloc_get_status (0);

thread_sleep (100);

for (i=0; i<11; i++) {
    if(i != 3 && i != 6 && i != 8)
        free(dynamicmem[i]);
    printf ("dump page status : \n");
    palloc_get_status (0);
}
```

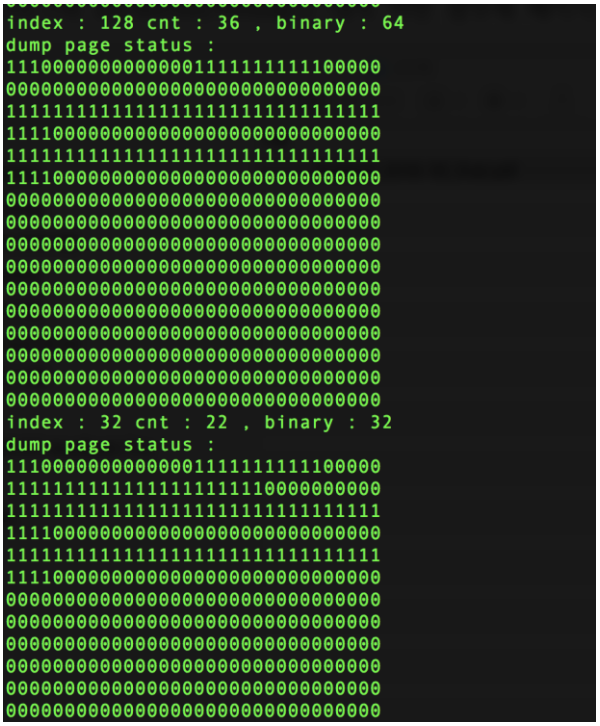
```
if(pallocator == 2)
{
    size_t i;
    char* dynamicmem[11];


    dynamicmem[0] = (char *) malloc (145000);
    memset (dynamicmem[0], 0x00, 145000);

    printf ("dump page status : \n");
    palloc_get_status (0);

    dynamicmem[1] = (char *) malloc (145000);
    memset (dynamicmem[1], 0x00, 145000);
    printf ("dump page status : \n");
    palloc_get_status (0);

    dynamicmem[2] = (char *) malloc (145000);
    memset (dynamicmem[2], 0x00, 145000);
    printf ("dump page status : \n");
    palloc_get_status (0);

    dynamicmem[3] = (char *) malloc (90000);
    memset (dynamicmem[3], 0x00, 90000);
    printf ("dump page status : \n");
    palloc_get_status (0);


    dynamicmem[4] = (char *) malloc (145000);
    memset (dynamicmem[4], 0x00, 145000);
    printf ("dump page status : \n");
    palloc_get_status (0);

    dynamicmem[5] = (char *) malloc (145000);
    memset (dynamicmem[5], 0x00, 145000);
    printf ("dump page status : \n");
    palloc_get_status (0);

    dynamicmem[6] = (char *) malloc (70000);
    memset (dynamicmem[6], 0x00, 70000);
    printf ("dump page status : \n");
    palloc_get_status (0);
```


파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
dump page status :
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111100000000000000000
00000111111111111111111111111111
11111111111111111111111111111111
11111111111110000000000000000001
11111111111111111111111111111111
11100000000000000001111111111111
11111111111111111111111000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000


파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
dump page status :
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111100000000000000000
00000111111111111111111111111111
11111111111111111111111111111111
11111111111110000000000000000001
11111111111111111111111111111111
11111111111111111101111111111111
11111111111111111111111000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
dump page status :

3번인 BuddySystem은 여러 크기의 메모리를 4번 선언하고 프리하면서 테스트를 했다. 각 메모리가 할당 받으려는 cnt보다 큰 2의 지수 값을 할당 받고 그 영역에 침범하지 않는 상황을 테스트 했다.






```c
if(pallocator == 3)
{
    size_t i;
    char* dynamicmem[4];



    dynamicmem[0] = (char *) malloc (145000);
    memset (dynamicmem[0], 0x00, 145000);

    printf ("dump page status : \n");
    palloc_get_status (0);

    dynamicmem[1] = (char *) malloc (45000);
    memset (dynamicmem[1], 0x00, 45000);
    printf ("dump page status : \n");
    palloc_get_status (0);

    dynamicmem[2] = (char *) malloc (145000);
    memset (dynamicmem[2], 0x00, 145000);
    printf ("dump page status : \n");
    palloc_get_status (0);

    dynamicmem[3] = (char *) malloc (90000);
    memset (dynamicmem[3], 0x00, 90000);
    printf ("dump page status : \n");
    palloc_get_status (0);



    for (i=0; i<4; i++) {
            free(dynamicmem[i]);
        //printf ("dump page status : \n");
        //palloc_get_status (0);
    }

}
```