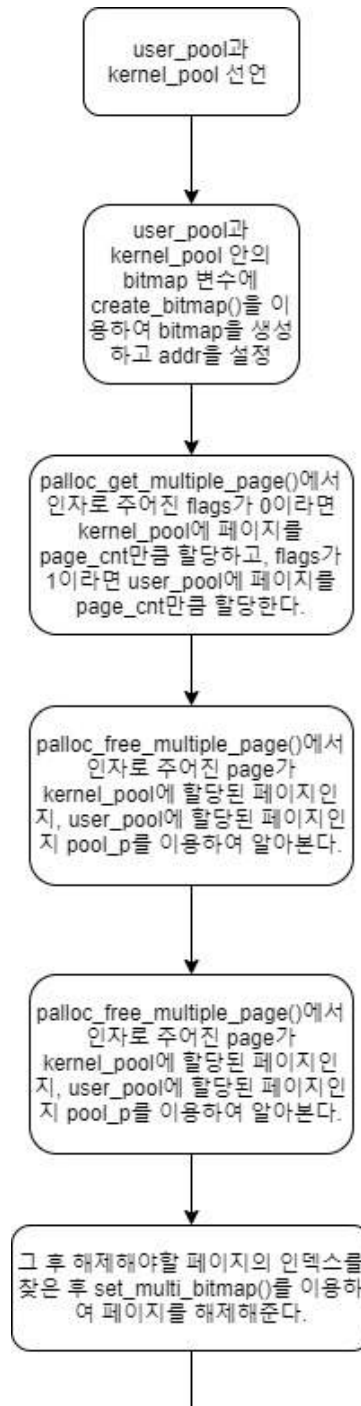
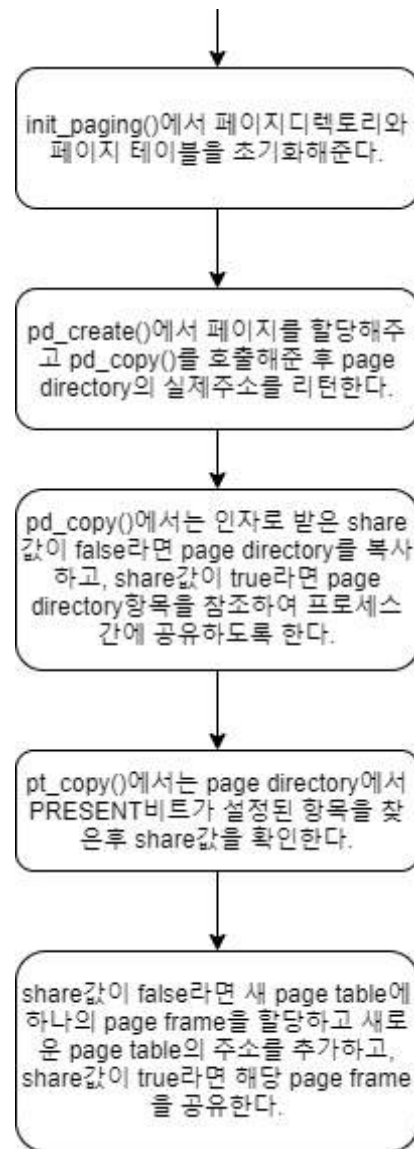


## 1. 개요

- 가상메모리란 컴퓨터에서 주기억 장치 안의 프로그램 양이 많아질 때, 사용하지 않는 프로그램을 보조 기억 장치 안의 특별한 영역으로 옮겨서, 주기억 장치처럼 사용할 수 있는 공간이다. 프로세스에 할당된 가상메모리공간은 물리 메모리 공간에 맵핑이 되며 맵핑된 정보는 테이블 형태로 저장되어 주기억장치에 저장이 된다. 실 메모리를 kernel pool과 user pool로 나누어 구현한 후 각각 bitmap을 사용하여 사용가능한 page frame을 관리하는 Page Allocator 구현해본다.

## 2. 상세설계

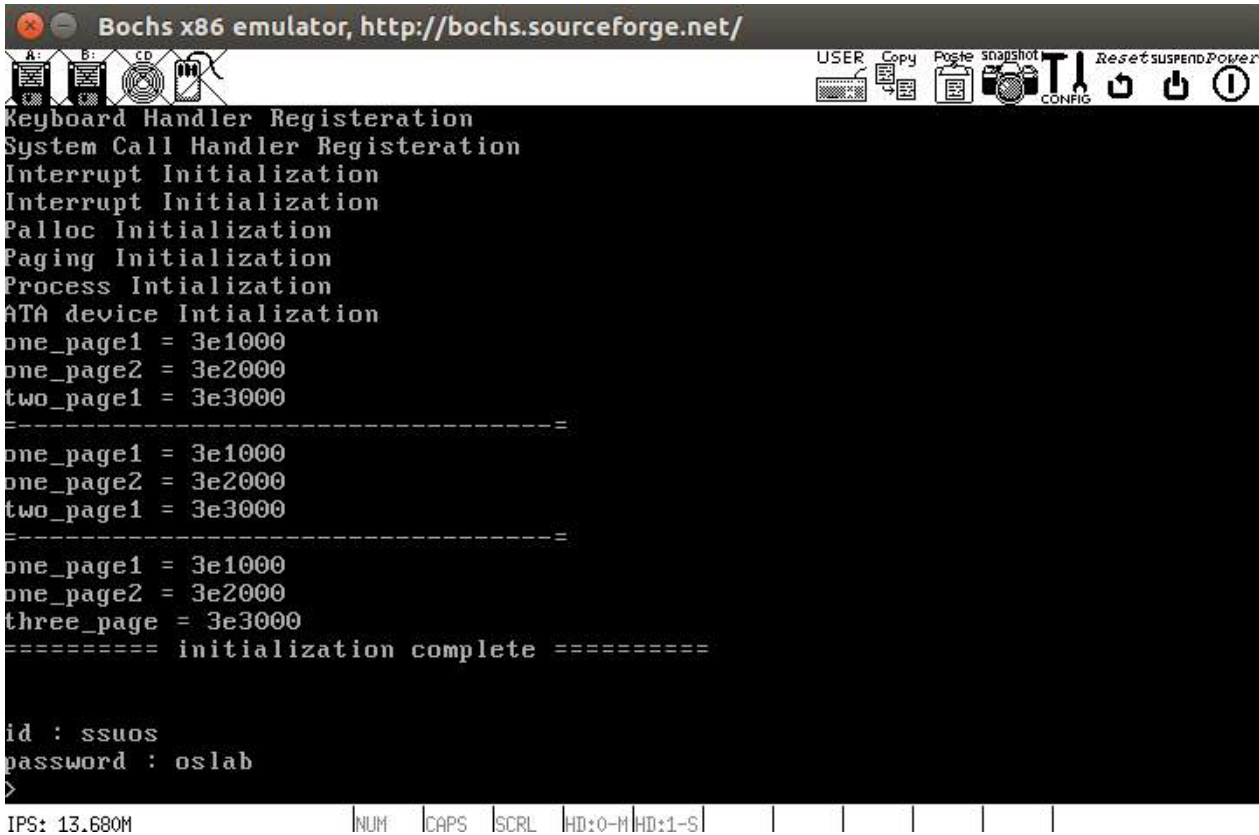




-bool pool\_p ( struct memory\_pool \*pool, void \*page); //페이지가 어느 memory\_pool에 속해있는지 알려주는 함수  
-void init\_palloc (void); //memory\_pool 초기화 함수  
-void\* palloc\_get\_multiple\_page(enum palloc\_flags flags, size\_t page\_cnt); //인자로 주어진 cnt만큼의 페이지를 할당해주는 함수  
-void\* palloc\_get\_one\_page (enum palloc\_flags flags); //페이지 하나 할당해주는 함수  
-uint32\_t \*palloc\_free\_multiple\_page(void \*pages, size\_t page\_cnt){ //인자로 주어진 cnt만큼의 페이지를 해제해주는 함수  
-uint32\_t \*palloc\_free\_one\_page (void \*pages); //한 페이지 해제해주는 함수

### 3. 실행결과

#### 1) [Case1 결과] 프로그램 수행시 결과화면 일부

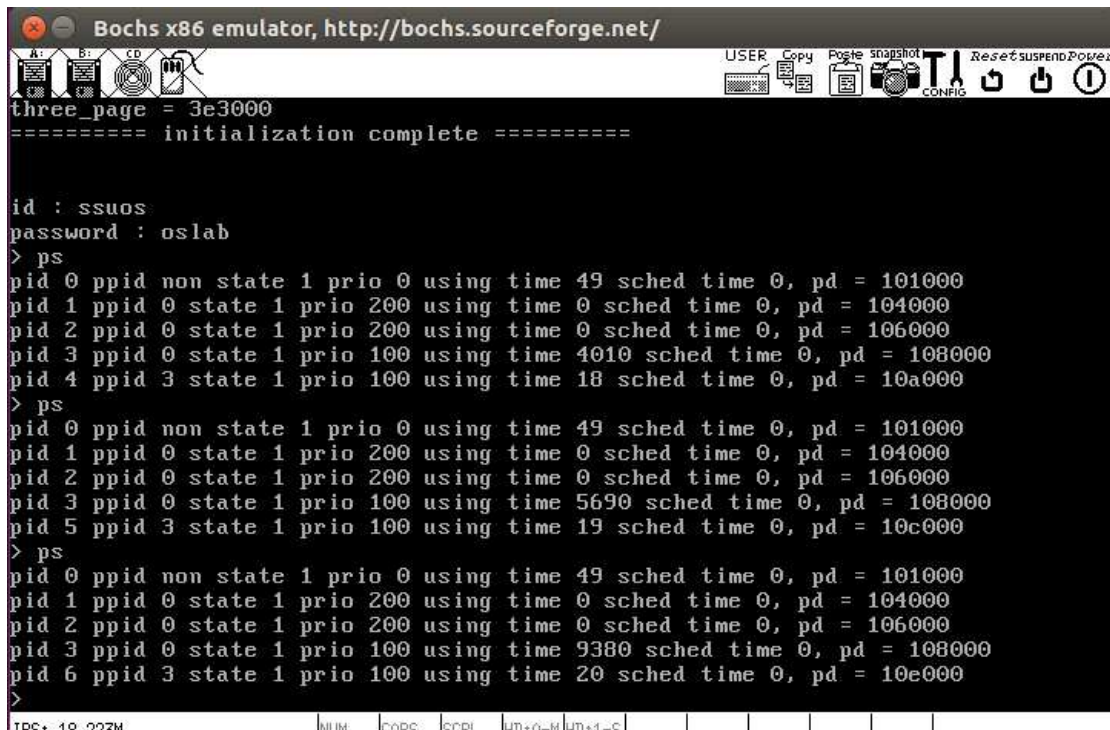


The screenshot shows the Bochs x86 emulator window with the title bar "Bochs x86 emulator, http://bochs.sourceforge.net/". The main window displays a series of initialization messages in a monospaced font. The messages include "Keyboard Handler Registration", "System Call Handler Registration", "Interrupt Initialization", "Pallocc Initialization", "Paging Initialization", "Process Initialization", and "ATA device Initialization". Below these, there are memory address assignments: "one\_page1 = 3e1000", "one\_page2 = 3e2000", and "two\_page1 = 3e3000". This is followed by a separator line "======", then the same assignments are repeated. Another separator line "======" follows, then "three\_page = 3e3000". The final line of the initialization sequence is "=====  
initialization complete =====". Below this, the user is prompted for a "id : ssuos" and a "password : oslab". The bottom status bar shows "IPS: 13.680M" and various hardware status indicators like "NUM", "CAPS", "SCRL", "HD:0-M", and "HD:1-S".

```
Bochs x86 emulator, http://bochs.sourceforge.net/
Keyboard Handler Registration
System Call Handler Registration
Interrupt Initialization
Interrupt Initialization
Pallocc Initialization
Paging Initialization
Process Initialization
ATA device Initialization
one_page1 = 3e1000
one_page2 = 3e2000
two_page1 = 3e3000
=====
one_page1 = 3e1000
one_page2 = 3e2000
two_page1 = 3e3000
=====
one_page1 = 3e1000
one_page2 = 3e2000
three_page = 3e3000
=====
initialization complete =====

id : ssuos
password : oslab
>
IPS: 13.680M  NUM  CAPS  SCRL  HD:0-M  HD:1-S
```

#### 2) [Case2 결과] 프로그램 수행 시 결과화면의 일부



The screenshot shows the Bochs x86 emulator window with the title bar "Bochs x86 emulator, http://bochs.sourceforge.net/". The main window displays the continuation of the initialization sequence from the previous screenshot, including the "three\_page = 3e3000" line and the "initialization complete" message. Below this, the user is prompted for a "id : ssuos" and a "password : oslab". After entering the password, the user enters the command "> ps". The output of the "ps" command is a list of processes, each with fields for pid, ppid, state, prio, using time, sched time, and pd. The processes listed are: pid 0 (non state, prio 0, using time 49, sched time 0, pd = 101000), pid 1 (state 1, prio 200, using time 0, sched time 0, pd = 104000), pid 2 (state 1, prio 200, using time 0, sched time 0, pd = 106000), pid 3 (state 1, prio 100, using time 4010, sched time 0, pd = 108000), and pid 4 (state 1, prio 100, using time 18, sched time 0, pd = 10a000). The user enters the command "> ps" again, and the output is repeated. The user enters the command "> ps" a third time, and the output is repeated. The bottom status bar shows "IPS: 13.680M" and various hardware status indicators like "NUM", "CAPS", "SCRL", "HD:0-M", and "HD:1-S".

```
Bochs x86 emulator, http://bochs.sourceforge.net/
three_page = 3e3000
=====
initialization complete =====

id : ssuos
password : oslab
> ps
pid 0 ppid non state 1 prio 0 using time 49 sched time 0, pd = 101000
pid 1 ppid 0 state 1 prio 200 using time 0 sched time 0, pd = 104000
pid 2 ppid 0 state 1 prio 200 using time 0 sched time 0, pd = 106000
pid 3 ppid 0 state 1 prio 100 using time 4010 sched time 0, pd = 108000
pid 4 ppid 3 state 1 prio 100 using time 18 sched time 0, pd = 10a000
> ps
pid 0 ppid non state 1 prio 0 using time 49 sched time 0, pd = 101000
pid 1 ppid 0 state 1 prio 200 using time 0 sched time 0, pd = 104000
pid 2 ppid 0 state 1 prio 200 using time 0 sched time 0, pd = 106000
pid 3 ppid 0 state 1 prio 100 using time 5690 sched time 0, pd = 108000
pid 5 ppid 3 state 1 prio 100 using time 19 sched time 0, pd = 10c000
> ps
pid 0 ppid non state 1 prio 0 using time 49 sched time 0, pd = 101000
pid 1 ppid 0 state 1 prio 200 using time 0 sched time 0, pd = 104000
pid 2 ppid 0 state 1 prio 200 using time 0 sched time 0, pd = 106000
pid 3 ppid 0 state 1 prio 100 using time 9380 sched time 0, pd = 108000
pid 6 ppid 3 state 1 prio 100 using time 20 sched time 0, pd = 10e000
>
IPS: 13.680M  NUM  CAPS  SCRL  HD:0-M  HD:1-S
```

#### 4. 소스코드

-palloc.c

struct memory\_pool user\_pool; //user\_pool 선언

struct memory\_pool kernel\_pool; //kernel\_pool선언

bool pool\_p ( struct memory\_pool \*pool, void \*page); //페이지가 어느 memory\_pool에 속해있는지 알려주는 함수

/\* Initializes the page allocator. \*/

//

void init\_palloc (void)

{

size\_t bm\_pages = DIV\_ROUND\_UP (bitmap\_struct\_size (1), PGSIZE);

lock\_init (&user\_pool.lock);

lock\_init (&kernel\_pool.lock);

user\_pool.bitmap

=

create\_bitmap((RKERNEL\_HEAP\_START-USER\_POOL\_START)/PAGE\_SIZE,USER\_POOL\_START,bm\_pages

\*

PGSIZE); //user\_pool에 bitmap생성

kernel\_pool.bitmap = create\_bitmap((USER\_POOL\_START-KERNEL\_ADDR)/PAGE\_SIZE,

KERNEL\_ADDR,bm\_pages \* PGSIZE); //kernel\_pool에 bitmap생성

user\_pool.addr = USER\_POOL\_START + bm\_pages \* PGSIZE; //user\_pool의 addr 설정

kernel\_pool.addr = KERNEL\_ADDR + bm\_pages \* PGSIZE; //kernel\_pool의 addr 설정

}

/\* Obtains and returns a group of PAGE\_CNT contiguous free pages.

\*/

void\* palloc\_get\_multiple\_page(enum palloc\_flags flags, size\_t page\_cnt){

struct memory\_pool\* tmp\_pool;

void \*pages;

size\_t page\_idx;

size\_t tm;

if (page\_cnt == 0)

return NULL;

//flags에 따라 kernel mode인지 user\_mode인지 판단

if(flags==0){//kernel mode라면

tmp\_pool = &kernel\_pool;

}

else if(flags == 1){//user mode라면

tmp\_pool = &user\_pool;

}

당

```
lock_acquire (&tmp_pool->lock);
page_idx = find_set_bitmap(tmp_pool->bitmap, 0, page_cnt, FALSE); //bitmap에서 비어있는 인덱스 찾아 할
lock_release (&tmp_pool->lock);

if (page_idx != BITMAP_ERROR){
    pages = (size_t *)tmp_pool->addr + (page_idx * PAGE_SIZE/sizeof(page_idx)); //페이지 주소 찾기
}
else
    pages = NULL;

if (pages != NULL)
{
    memset (pages, 0, PGSIZE * page_cnt);
}

return pages;
}
```

/\* Obtains a single free page and returns its address.

\*/

void\* pallocc\_get\_one\_page (enum pallocc\_flags flags)

{

return pallocc\_get\_multiple\_page (flags, 1); //1페이지 할당할때

}

/\* Frees the PAGE\_CNT pages starting at PAGES. \*/

uint32\_t \*pallocc\_free\_multiple\_page(void \*pages, size\_t page\_cnt){ //여러 페이지 해제해주는 함수

struct memory\_pool\* tmp\_pool;

size\_t page\_idx;

if (pages == NULL || page\_cnt == 0)

return NULL;

if(pool\_p(&kernel\_pool, pages)){ //kernel\_pool이라면

tmp\_pool = &kernel\_pool;

}

else if(pool\_p(&user\_pool, pages)){ //user\_pool이라면

```

        tmp_pool = &user_pool;
    }

    page_idx = pg_no(pages) - pg_no(tmp_pool->addr); //해제해야할 인덱스 찾기

    if(bitmap_all (tmp_pool->bitmap, page_idx, page_cnt))
        set_multi_bitmap (tmp_pool->bitmap, page_idx, page_cnt, FALSE); //비트맵에서 해당하는 인덱스
        찾아 페이지 해제해주기

    return tmp_pool->addr;
}

/* Frees the page at PAGE. */
uint32_t *palloc_free_one_page (void *pages){
    palloc_free_multiple_page (pages, 1); //한페이지 해제할때
}

bool pool_p( struct memory_pool *pool, void *page) //해제해야할 페이지가 user_pool에 있는지 kernel_pool에 있는
지 페이지 주소로 판단하는 함수
{
    return (pg_no (page) >= pg_no (pool->addr) && pg_no (page) < (pg_no (pool->addr) + bitmap_size
(pool->bitmap)));
}

```

-paging.c

```

/*
    page table 복사
*/
void pt_copy(uint32_t *pd, uint32_t * dest_pd, uint32_t idx, uint32_t* start, uint32_t *end, bool share){

    uint32_t *pt = pt_pde(pd[idx]);
    uint32_t *new_pt;
    uint32_t s, e, i;
    new_pt = palloc_get_one_page(kernel_area);
    dest_pd[idx] = (uint32_t)new_pt | (pd[idx] & ~PAGE_MASK_BASE & ~PAGE_FLAG_ACCESS);
    s = pte_idx_addr(start);
    e = pte_idx_addr(end);

    for(i = s; i<e; i++)
    {
        if(pt[i] & PAGE_FLAG_PRESENT) //source page directory에서 전체 항목들을 검사해 PRESENT 비트
가 설정 된 항목들을 찾고

```

```

        {
            //share가 F(0)로 설정된 경우 새로운 page table에 하나의 page frame을 할당하고
destination directory의 인자로 받은 idx 번째 항목에 새로운 page table의 주소를 추가.
            if(share == FALSE){
                uint32_t* pg = palloc_get_one_page(kernel_area);
                new_pt[i] = (uint32_t)pg | (pd[i] & ~PAGE_MASK_BASE & ~
PAGE_FLAG_ACCESS);

                memcpy_hard((void*)(pt[i] & PAGE_MASK_BASE), (void*)pg, PAGE_SIZE);
            }
            //share가 T(1) 인 경우 해당 페이지 page frame을 공유함
            else{ //share == true
                new_pt[i] = pt[i];
            }
        }
    }
}

/*
page directory 복사.
커널 영역 복사나 fork에서 사용
*/
void pd_copy(uint32_t *pd, uint32_t * dest_pd, uint32_t idx, uint32_t* start, uint32_t *end, bool share){

    uint32_t i;

    for(i = 0; i < 1024; i++){
        if(pd[i] & PAGE_FLAG_PRESENT){
            if(share== FALSE){//share = FALSE 일때, page directory를 복사
                pt_copy(pd, dest_pd, i, start, end, share);
            }
            else{ //share = TRUE 일때, page directory 항목을 프로세스간에 공유
                dest_pd[i] = pd[i];
            }
        }
    }
}

uint32_t* pd_create (pid_t pid){
    uint32_t *pd = palloc_get_one_page(kernel_area);
    //pd_copy()호출
    pd_copy((uint32_t*)read_cr3(), pd ,0 ,(uint32_t*)0, (uint32_t*)USER_POOL_START,TRUE);
    //성공시 page directory의 실 주소를 리턴
    return pd;
}

```