OS project 3

2016310936 우승민

이번 프로젝트의 목표는 xv6 에 **mmap(), munmap()** 두개의 system call 을 **paging** 하도록 구현하여 File 의 **mapping** 과 **anonymous mapping** 을 가능하게 만드는 것입니다.

```
struct map{
  int  addr;
  int valid;
  struct proc *p;
  int length;
  int flags;
  int fd;
  struct file *f;
  int offset;
};

struct map m2[100];
```

가장 먼저 저는 **virtual memory** 의 memory region 을 관리하는 구조체를 만들어 주었습니다.

구조체는 현재 공간이 할당 되어있는지 나타내는 valid 를 포함하여, mapping area 에 대한 정보들이 담겨있습니다.

```
static void
mapmain(void)
{
  int i;
  for(i=0; i<100; i++){
        m2[i].length =0;
        m2[i].valid = 0;
        m2[i].addr = 0;
        m2[i].fd = -555;
        m2[i].p = 0;
        m2[i].flags =0;
        m2[i].f = 0;
        m2[i].offset = 0;
        }
}
"main.c" 132L, 3504C written
```

Main.c 함수에서 구조체안의 내용을 초기화 해주었습니다.

그 후 이전 과제와 마찬가지로 syscall 을 사용할 수 있도록, **defs.h, user.h** 등 함수를 연결시켜주었습니다.

```
int             mmap(int fd, int offset, int length, int flags);
int             munmap(void* addr, int length);

// swtch.S
void            swtch(struct context**, struct context*);

"defs.h" 196L, 5717C written
```

다만 **defs.h** 에서 **mmap** 함수의 return 값을 int 로 해주었기 때문에, **user.h** 에서 void pointer 형으로 변형하도록 해주었습니다.

```
void *mmap(int, int, int, int);
int munmap(void*, int);


// ulib.c
int stat(const char*, struct stat*);
"user.h" 44L, 1044C written
```

```
int
sys_mmap(void)
{
  int fd, offset, length, flags;

  if(argint(0,&fd) <0)
        return -1;
  if(argint(1,&offset) <0)
        return -1;
  if(argint(2,&length) <0)
        return -1;
  if(argint(3,&flags) <0)
        return -1;

  if(fd == -1){
        offset = 0;
        }

  int a;
  a = mmap(fd, offset, length, flags);
  return a;
}
```

```
int
sys_munmap(void)
{
  int length;
  int addr;

  if(argint(0,&addr) <0)
        return -1;
  if(argint(1,&length) <0)
        return -1;

  return munmap((void *)addr, length);
}
```

Sysproc.c 코드에서 **argint** 를 통해 인자를 넘겨주었습니다. 주의할 점은 munmap 함수의 addr 은 void pointer 형으로 넘겨야 한다는 것입니다.

```
#include "param.h"
#include "types.h"
#include "defs.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "elf.h"

#include "fs.h"
#include "spinlock.h"
#include "sleeplock.h"
#include "file.h"
```

mmap, munmap, page_handler 은 **walkpgdir** 함수와 **mappages** 함수를 사용하기 위해 **vm.c** 함수에 구현해 주었습니다. (**static** 함수라서 vm.c 에만 사용 가능)

아래 4 개의 헤더파일은 File structure 를 사용해 주기위해 추가해 주었습니다.

mmap 함수를 살펴보면 우선 mmap 에 실패할 경우 **-1** 을 return 하도록 해주었습니다.

```
int mmap(int fd, int offset, int length, int flags)
{

  if(length % PGSIZE !=0)
        return -1;
  int addr;

  addr = myproc()->sz;

  int i;
  char buf[offset];

  struct file *f;

  f = myproc() -> ofile[fd];

  if(fd != -1){
        if(f->writable){
                if((flags & MAP_PROT_WRITE)!=MAP_PROT_WRITE)
                        return -1;
        }else{
                if(flags & MAP_PROT_WRITE)
                        return -1;
        }
        fileread(f,buf, offset);
  }
```

Length의 단위가 4kb가 아닐 경우

File의 open flag와 mmap의 flag가 다를 경우

그리고 만약 fd가 -1이 아니라면 offset만큼 파일의 위치를 이동시켜 주었습니다.

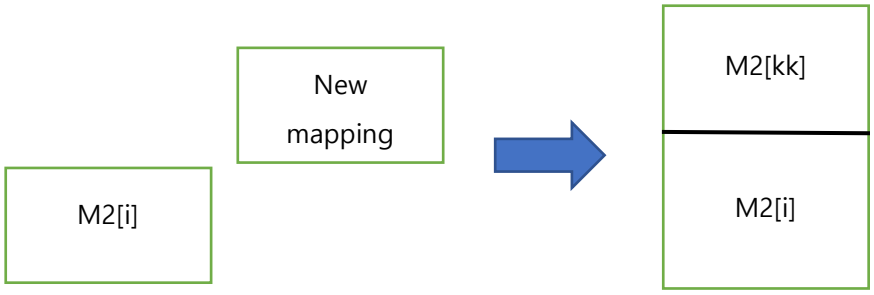
그 후 while문과 for문을 사용하여 구조체를 탐색하여 mapping해주었습니다.

```
int t=0;
while(1){
	if(t==1)
		break;
	for(i=0; i<100; i++){
		if(m2[i].p == myproc()){
			if(addr < m2[i].addr + m2[i].length){
				if(m2[i].fd == fd){
					m2[i].length = m2[i].length + length;
				}
```

만약 기존에 같은 프로세스이면서, mapping 범위가 겹치지 않고, 파일이 같으면, 기존 구조체에 length를 더해주어 합쳐주었습니다.


```
		else{
			for(int kk=i+1; kk<100; kk++){
				if(m2[kk].valid==0){
					addr = m2[i].addr + m2[i].length;
					m2[kk].addr = addr;
					m2[kk].p = myproc();
					m2[kk].length = length;
					m2[kk].f = f;
					m2[kk].valid = 1;
					m2[kk].fd = fd;
					m2[kk].flags = flags;
					m2[kk].offset = offset;
					break;
				}
			}
		}
		t = 1;
		break;
	}
}
```

파일이 다를 경우에는 i이후에 빈 공간을 찾아 새로 추가해주었습니다.



```
if(m2[i].valid == 0){
        m2[i].addr = addr;
        m2[i].p = myproc();
        m2[i].valid = 1;
        m2[i].length = length;
        m2[i].f = f;
        m2[i].fd = fd;
        m2[i].flags = flags;
        m2[i].offset = offset;
        t =i;
        break;
}
```

위 두경우 이외에는 새 프로세스에서 mapping을 한 것이므로 새 구조체를 바로 사용하게 해주었습니다.

이후 map protection flag 에 populate 를 입력해주었으면, **kalloc** 과 **mappages** 함수를 이용하여 physical memory 를 할당하고, 새 PTE 를 만들어 주었습니다.

```
if(flags & MAP_POPULATE){
     for(i=0; i<length; i+= PGSIZE){
             char *mem;
             mem = kalloc();
             memset(mem, 0, PGSIZE);
             if(fd !=-1)
                     fileread(f, mem, PGSIZE);
             if(flags & MAP_PROT_WRITE)
                     mappages(myproc()->pgdir, (void *)(addr + i), PGSIZE, V2P(mem), PTE_W|PTE_U);
             else
                     mappages(myproc()->pgdir, (void *)(addr + i), PGSIZE, V2P(mem), PTE_U);
     }
}

return addr;
```

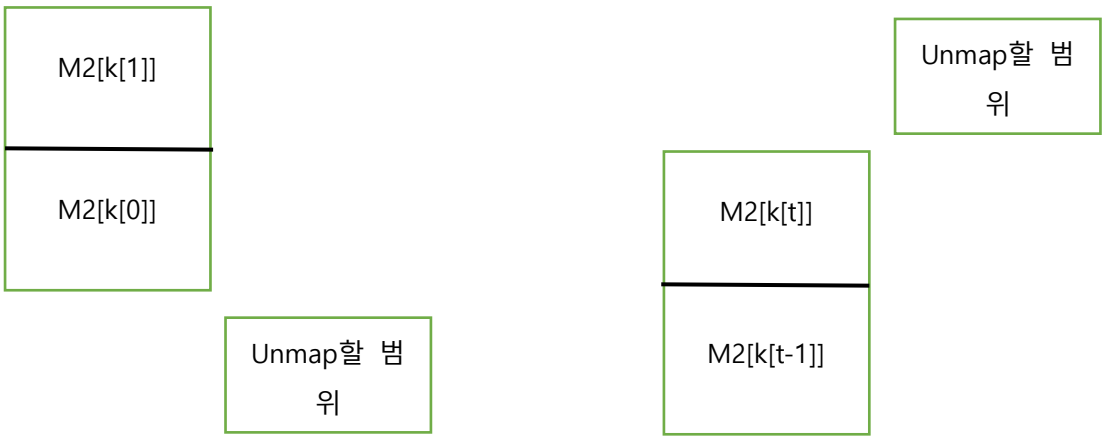
다음으로 munmap 함수를 보면 우선 void pointer형으로 넘어온 addr을 int형으로 변형하여 비교에 용이하게 해주었고, length와 addr가 4kb의 배수인지 확인해 주었습니다.

```
int munmap(void *addr, int length)
{
  int add = (int)addr;
  if((length % PGSIZE !=0) || (add % PGSIZE !=0))
        return -1;
  int i;
  int k[10];
  int t=0;
  for(i=0; i<100; i++){
        if((m2[i].valid == 1) && (m2[i].p == myproc())){
                k[t++] = i;
        }
  }

  t = t-1;

  if(t ==-1)
        return 0;
```

여기서 k array는 구조체를 탐색하여 현재 프로세스와 같은 프로세스였던 구조체들을 구분해 주기 위해 사용하였습니다.

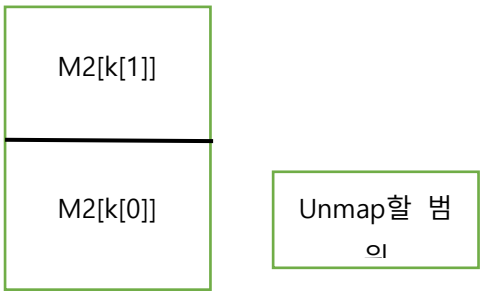
만약 t가 -1이면 프로세스가 구조체에 없었다는 뜻이므로 mapping 되어있는 영역이 없는 것입니다.

```
int cur=0;

while(m2[k[cur]].addr < add)
      cur++;
if(add + length < m2[k[0]].addr)
      return 0;
if(add > m2[k[t]].addr + m2[k[t]].length)
      return 0;
```



위 두 경우는 unmap할 공간이 비어 있다는 뜻이므로 0을 return해 주었습니다.

cur 변수는 현재 unmap할 범위가 어디에 속한지를 확인하기 위해서 넣어주었습니다.



이 경우에는 cur값이 0이 됩니다.

이후 unmap을 4개의 범위에 따라 아래와 같이 나누었습니다.

각각에 unmaping 과정은 똑같으므로 1번째 부분만 설명하겠습니다.

```
int j;

pte_t *pte;
if(add <= m2[k[0]].addr){
      if(add + length < m2[k[t]].addr + m2[k[t]].length){
            for(j=m2[k[0]].addr; j< add + length; j+= PGSIZE){
                  pte = walkpgdir(myproc()->pgdir, (void *)j, 0);
                  if((*pte & PTE_P) !=0){
                        char *v = P2V(PTE_ADDR(*pte));
                        if(m2[k[cur]].fd !=-1){
                              m2[k[cur]].f ->off = m2[k[cur]].offset;
                              if((*pte & 0x40) == 0x40){
                                    filewrite(m2[k[cur]].f, v , PGSIZE);
                              }

                        }

                        kfree(v);
                        *pte = 0;
                        lcr3(V2P(myproc()->pgdir));
                  }
```

우선 for문을 사용하여 unmap이 가능한 범위를 반복하게 해주었고 (1번째 경우는 구조체 0번째부터 넘어온 범위 끝까지 -> m2[k[0]].addr ~ add + length)

만약 현재 범위의 fd가 -1이 아니면 file이 존재하는 것이므로 **pte**의 **dirty bit**가 set되었으면, **write back** 해주었습니다.

free하는 과정은 **deallocuvm**함수를 참고하였습니다.

```
int
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  pte_t *pte;
  uint a, pa;

  if(newsz >= oldsz)
    return oldsz;

  a = PGROUNDUP(newsz);
  for(; a  < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
      a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
    else if((*pte & PTE_P) != 0){
      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      char *v = P2V(pa);
      kfree(v);
      *pte = 0;
    }
  }
  return newsz;
}
```

그 후 unmaping한 공간을 다시 활용할 수 있도록 구조체 내용을 초기화 해주었습니다.

```
 for(j=0; add+length >= m2[k[j]].addr + m2[k[j]].length; j++){
         m2[k[j]].addr = 0;
         m2[k[j]].length = 0;
         m2[k[j]].valid = 0;
         m2[k[j]].fd = 0;
         m2[k[j]].flags = 0;
         m2[k[j]].p = 0;
         m2[k[j]].f = 0;
         m2[k[j]].offest = 0;
 }
```

이제 **page_fault**를 다룬 부분을 보면

우선 기존 **trap.c**에서 case문에 **T_PGFLT**인 경우를 추가하여 **page_handler**로 이동하게 해주었습니다.

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_PGFLT:
    if(page_handler(tf, rcr2())!=1)
        myproc()->killed = 1;
    break;
```

여기서 **rcr2**는 fault가 일어난 부분의 **주소**가 담겨 있습니다.

만약 **handler**에서 실패하여 **1**이 나오지 않는 경우는 프로세스를 **kill**해 주었습니다.

기존 mmap함수에서 실패하여 -1이 return된 경우에 **rcr2 주소가 -1**이 되어서 그럴 경우에만 "permission denied"문구를 띄우게 해주었습니다.

```
int page_handler(struct trapframe *tf, uint rcr2)
{
  int i;
  int addr = rcr2;
  if(addr == -1){
        cprintf("permission denied₩n");
        return -1;
  }
```

이후에는 page_fault가 난 경우를 3가지로 구분하였습니다.

1. Read flag가 없는데 read하려고 할 때
2. Write flag가 없는데 write하려고 할 때
3. POPULATE flag를 해주지 않았을 때

```
 for(i=0; i<100; i++){
       char *mem;
       if(m2[i].valid){
              if(m2[i].p == myproc()){
                     if((m2[i].addr <= addr) && (m2[i].addr + m2[i].length > addr)){
                            if(m2[i].flags & MAP_PROT_WRITE){
                                   if(!(m2[i].flags & MAP_PROT_READ)) // you can't read
                                          return -1;

                                   // below : no POPULATE
                                   mem = kalloc();
                                   memset(mem,0,PGSIZE);
                                   if(m2[i].fd !=-1)
                                          fileread(m2[i].f, mem, PGSIZE);
                                   if(m2[i].flags & MAP_PROT_WRITE)
                                          mappages(myproc()->pgdir, (void *)addr, PGSIZE, V2P(mem), PTE_U|PTE_W);
                                   else
                                          mappages(myproc()->pgdir, (void *)addr, PGSIZE, V2P(mem), PTE_U);
                                   return 1;
                            }
                            else  // you can't write
                                   return -1;
                     }
```

1, 2의 경우는 flag를 확인한 후 -1을 return해 주었고, 3의 경우는 mmap함수에서 했던 것처럼 physical memory를 할당해 준 후 PTE를 연결해 주었습니다.

예외적인 상황으로 exit(), close()를 실행할 때 기존의 mapping된 공간이 unmap되어야 합니다.

```
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }
```

그런데 exit함수에 **fileclose**함수가 존재하므로 fileclose함수만 수정해 주었습니다.

```
void
fileclose(struct file *f)
{
  struct file ff;

  for(int i=0; i<100; i++){
        if(m2[i].f == f)
                munmap((void *)(m2[i].addr), m2[i].length);
  }

  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("fileclose");
  if(--f->ref > 0){
    release(&ftable.lock);
    return;
  }
  ff = *f;
  f->ref = 0;
  f->type = FD_NONE;
  release(&ftable.lock);

  if(ff.type == FD_PIPE)
    pipeclose(ff.pipe, ff.writable);
  else if(ff.type == FD_INODE){
    begin_op();
    iput(ff.ip);
    end_op();
  }
}
```

만약 close하려는 파일을 mapping한 범위가 있다면, 그 부분을 unmap하도록 추가해주었습니다.

Mmap_test.c를 아래 상태로 실행하면,

```
int main(int argc, char** argv) {


        printf(1, "mmap test \n");
        int i;
        int size = 4096;
        int fd = open("README", O_RDWR);
        char* text = mmap(fd, 2048,   size, MAP_PROT_READ|MAP_PROT_WRITE);
        char* text2 = mmap(-1, 0,  size, MAP_PROT_READ|MAP_PROT_WRITE);  //ANONYMOUS example

        for (i = 0; i < size; i++)
                printf(1, "%c", text[i]);
        printf(1,"\n============file mmap end==========\n\n\n");

        text2[0] = 's';
        text2[4095] = 'Y';
        for (i = 0; i < size; i++)
                printf(1, "%c", text2[i]);
        printf(1,"\n============anonymous mmap end==========\n");

        munmap(text, size);
        munmap(text2, size);
        exit();
}
```

Pdf와 동일한 출력결과가 나오고

```
$ mmap_test
mmap test
-x86 or non-ELF machines (like OS X, even on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".

============file mmap end==========



s
Y
============anonymous mmap end==========
$ QEMU: Terminated
```

Text2 와 text1 의 fd 를 바꾸어서 하면

```
int main(int argc, char** argv) {


        printf(1, "mmap test \n");
        int i;
        int size = 4096;
        int fd = open("README", O_RDWR);
        char* text = mmap(-1, 2048,   size, MAP_PROT_READ|MAP_PROT_WRITE);
        char* text2 = mmap(fd, 2048,  size, MAP_PROT_READ|MAP_PROT_WRITE);

        for (i = 0; i < size; i++)
                printf(1, "%c", text[i]);
        printf(1,"\n============file mmap end==========\n\n\n\n");

        text2[0] = 's';
        text2[4095] = 'Y';
        for (i = 0; i < size; i++)
                printf(1, "%c", text2[i]);
        printf(1,"\n============anonymous mmap end==========\n");

        munmap(text, size);
        munmap(text2, size);
        exit();
}
```

아래와 같이 출력되고 cat README 명령어를 실행하여 write back 이 잘 되었다는 것을 확인할 수 있습니다.

```
$ mmap_test
mmap test

============file mmap end==========



sx86 or non-ELF machines (like OS X, even on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".

Y
============anonymous mmap end==========
$ cat README
xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
but is implemented for a modern x86-based multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also https://pdos.csail.mit.edu/6.828/, which
provides pointers to on-line resources for v6.

xv6 borrows code from the following sources:
    JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
    Plan 9 (entryother.S, mp.h, mp.c, lapic.c)
    FreeBSD (ioapic.c)
    NetBSD (console.c)

The following people have made contributions: Russ Cox (context switching,
locking), Cliff Frey (MP), Xiao Yu (MP), Nickolai Zeldovich, and Austin
```

```
The following people have made contributions: Russ Cox (context switching,
locking), Cliff Frey (MP), Xiao Yu (MP), Nickolai Zeldovich, and Austin
Clements.

We are also grateful for the bug reports and patches contributed by Silas
Boyd-Wickizer, Anton Burtsev, Cody Cutler, Mike CAT, Tej Chajed, eyalz800,
Nelson Elhage, Saar Ettinger, Alice Ferrazzi, Nathaniel Filardo, Peter
Froehlich, Yakir Goaron,Shivam Handa, Bryan Henry, Jim Huang, Alexander
Kapshuk, Anders Kaseorg, kehao95, Wolfgang Keller, Eddie Kohler, Austin
Liew, Imbar Marinescu, Yandong Mao, Matan Shabtay, Hitoshi Mitake, Carmi
Merimovich, Mark Morrissey, mtasm, Joel Nider, Greg Price, Ayan Shafqat,
Eldar Sehayek, Yongming Shen, Cam Tenny, tyfkda, Rafael Ubal, Warren
Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi Watanabe, Nicolas
Wolovick, wxdao, Grant Wu, Jindong Zhang, Icenowy Zheng, and Zou Chang Wei.

The code in the files that constitute xv6 is
Copyright 2006-2018 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

Please send errors and suggestions to Frans Kaashoek and Robert Morris
(kaashoek,rtm@mit.edu). The main purpose of xv6 is as a teaching
operating system for MIT's 6.828, so we are more interested in
simplifications and clarifications than new features.

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
"make". On nonsx86 or non-ELF machines (like OS X, even on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".

Y$
```