Project04 Wiki

2020081958 송재휘

1) Design

명세에서 주어진 대로 Copy-on-Write(CoW)를 구현하기 위해 2가지 하위 파트로 나누어 구현할 계획이다. 첫번째는, initial sharing 부분이다. 여기서 system call과 이를 구현하기 위해선, 특정 물리 페이지의 참조 횟수를 알아야 하기 때문에 명세에 주어진 대로 kalloc.c 파일에서 참조 횟수 기록을 위한 배열을 선언해줄 것이다. 참조 횟수는 최소 0의 값을 가지기 때문에 uint형의 배열을 선언해줄 것이고, 해당 배열은 존재하는 모든 물리 페이지를 다 커버할 수 있어야 하기 때문에 시스템의 총 물리 주소의 크기를 나타내는 PHYSTOP(memlayout.h에 정의됨)을 PGSIZE인 4096 byte(= 2^12 byte)로 나누어줄 것이다. 위 과정을 12비트만큼를 오른쪽으로 이동하는 비트 연산(>>)으로 수행할 것이고, 해당 비트의 수를 mmu.h에서 PG_BIT라는 이름으로 정의해줄 것이다. 따라서, uint ref_count[PHYSTOP >> PG_BIT[;로 참조 횟수 추적을 위한 자료구조를 선언한다. 또한 시스템 콜에서 이후에 빈 페이지 수를 리턴해줘야 하기 때문에 kmem 구조체 내부에 빈 페이지의 수를 나타낼 int len이라는 특성을 추가할 것이다. 이후 main.c에서 freerange함수를 돌면서 빈 페이지들이 초기화되는데, 위 함수를 실행하는 것은 kinit1, kinit2 두 함수 뿐이다. 근데 main.c를 보면 kinit1이 먼저 실행되기 때문에, kinit1에서 kmem.len을 0으로 초기화 시켜주겠다. 이후 freerange 함수에서 페이지를 초기화 시켜주는 루프를 돌 때 매 스텝마다 해당 물리페이지를 0으로 초기화 시켜줄 것이다. 물리페이지의 인덱스를 찾는 것 역시 위의 비트 연산을 활용할 것이다.(ref_count[V2P(p) >> PG_BIT] = 0;) 이후 kmem.len을 1씩 올려준다. 이후, kalloc()에서는 물리 페이지를 할당해주는 것임으로 새로 할당할 때 해당 물리 페이지의 참조횟수를 1로 설정하고, kmem.len--;를 수행할 것이다. CoW이기 때문에 fork 시에 물리 페이지를 복사하는 것이 아니라, 수정이 일어나기 전까지 같은 물리페이지를 가리키도록 만들어야 하는데, 이를 위해 기존 시스템에서 물리 페이지를 복사하는 copyuvm이라는 vm.c 파일 내의 함수를 수정해줄 것이다. 기존의 copyuvm에서는 kalloc()을 통해 새로운 물리페이지를 할당 받아 copy를 진행하는데, 이를 수정해 kalloc()없이 페이지 테이블이 존재하는 부모 프로세스의 물리 페이지를 가리키도록 코드를 짤 것이다. 추가적으로, 이때 해당 물리 주소의 쓰기가 발생하면 트랩을 발생시키기 위해 해당 페이지 테이블 엔트리의 flag에서 PTE_W(쓰기 허용)을 빼줄 것이다. 이후 각 물리페이지를 참조할 때마다 참조횟수를 뒤에서 언급할 incr_refc 함수를 통해 증가시켜 주도록 만들 것이다. 이후 페이지 테이블이 수정되었기 때문에 lcr3(V2P(pgdir));을 통해 TLB를 flush해준다. 기존의 시스템에서는 kfree를 통해 바로 해당 물리 페이지를 놓게 되는데, CoW에선 해당 물리 페이지를 여러 프로세스가 공유 중일 수 있기 때문에 바로 놓아선 안된다. 따라서, kfree함수 호출시 해당 물리 주소의 참조횟

수를 읽어와 0보다 큰 경우 참조를 하는 페이지가 하나 이상 있다는 뜻이기 때문에, 먼저 참조횟수를 1 감소시킨다. 이후 다시 한번 참조횟수를 읽고, 해당 값이 0인 경우 (해당 물리페이지를 참조하는 마지막 프로세스인 경우) 기존과 동일하게 물리 페이지를 free 시켜줄 것이다. 이따 당연하게 free page의 수를 의미하는 kmem.len을 1 증가시킨다. 위 과정에서 참조 횟수를 읽거나 올리거나 내리거나 하기 위해서 추가 함수를 구현해야 하는데 명세에서 주어진 이름을 그대로 사용해 똑같이 물리 주소를 PG_BIT를 활용한 비트 연산을 통해 물리 페이지를 계산하고 값을 변경 혹은 return해줄 것이다. 자세한 코드는 implementation 부분에서 설명하겠다. 이후 system call의 경우 구현해야 할 것이 4개 있는데 countfp는 단순히 위에서 만든 kmem.len을 반환하면 되기 때문에 kalloc.c에 나머지 countvp, countpp, countptp의 경우는 vm.c에 구현할 것이다. countvp 시스템콜의 경우 단순히 프로세스의 유저메모리에 할당된 가상 페이지의 수를 반환하면 되기 때문에 myproc()->sz를 PGROUNDUP 시키고 PGSIZE로 나눈 값을 반환할 것이다. 이때 PGROUNDUP을 하는 이유는 프로세스의 sz가 마지막 페이지를 꽉 채우지 못했더라도 해당 페이지를 쓰고 있는 것이기 때문에 해당 페이지까지 카운트 하기 위함이다. 이후, countpp의 경우 유효한 물리 주소가 할당된 page table entry의 수를 반환해야 하는데 xv6의 구조상 demand paging이 없기 때문에, 모든 가상 페이지는 물리 페이지에 매핑된다. 따라서 countvp와 같은 값을 가진다. 하지만, 구현 과정에서는 page table을 순회하며 pte가 존재하는 경우를 세서 반환할 것이다. 마지막으로 countptp의 경우, page table 엔트리의 총 수가 1024이기 때문에 0부터 1023까지 차례로 pgdir의 엔트리를 돌면서 해당 엔트리에 속하는 페이지가 할당되었는지를 파악해 count해주고, 이를 반환해줄 것이다. 이렇게 선언한 system call과 함수들은 defs.h에 등록하고, system call은 syscall.c, syscall.h, user.h, usys.S에 차례로 등록한다. (sysproc.c에 wrapper function도 구현).

이제는 CoW의 2번째 파트인 make a copy이다. 1번 파트에서 물리 페이지를 공유하고, PTE_W를 disable 시켰기 때문에, 쓰기 발생 시 page fault가 발생한다. 이를 trap.c에서 case T_PGFLT:를 추가해 CoW_handler()라는 함수를 활용해 처리해줄 것이다. CoW_handler()는 vm.c에서 선언할 것이다. 위 핸들러 함수에서는 우선 page fault가 발생한 위치를 rcr2()로 읽고, 해당 가상 주소가 가리키는 page table entry를 찾는다 이후, 해당 가상 주소가 페이지 테이블에 매핑 되어 있지 않다면, 에러 문구를 출력하고 종료시킬 것이다. 반대로, 올바르게 매핑 됐다면 해당 pte가 가리키는 물리 주소를 PTE_ADDR(*pte);를 통해 구하고 해당 물리 주소가 해당하는 페이지의 참조 횟수를 읽어온다. 이후 참조 횟수가 1보다 크다면 (여러 프로세스가 공유 중) 해당 물리 페이지를 복사해야 한다. Kalloc()을 통해 물리 페이지를 할당 받고, 이전에 copyuvm에서 하던 것처럼 새롭게 할당 받은 물리 페이지에 참조 중이던 물리 페이지의 컨텐츠를 복사한다. 이후, 기존 페이지 테이블 엔트리가 새롭게 할당 받은 물리 페이지를 가리키도록 설정하고, flag는 다시 PTE_W할 수 있도록 부여한다. 반대로 참조 횟수가 1이라면 (해당 프로세스

는 특정 물리 페이지를 참조하는 유일한 프로세스) 단순히 *pte의 flag에 PTE_W를 추가하고 사용하면 된다. 이후, 페이지 테이블이 수정되었기 때문에 lcr3(V2P(myproc()->pgdir));을 활용해 TLB flush를 진행한다.

2) Implementation

1. kalloc.c

Free page의 수를 세기 위해 kmem 구조체에 int len을 추가하였다.

```
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
  int len;  // for counting # of free pages
} kmem;
```

물리 페이지의 참조횟수를 추적하기 위해 전역 변수로 선언하였다. 참조 횟수는 최소 0임으로 uint로 선언했다. PHYSTOP은 물리 주소의 끝을 나타내는 definition으로 memlayout.h에 선언되어 있다. PG_BIT의 경우 mmu.h에 직접 선언해주었다. Xv6의 페이지 사이즈인 4096 바이트 만큼 비트연산을 통해 나누어 주기 위해 (4096 == 2^12) PG_BIT를 12로 선언해주었다. (mmu.h)

```
// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory entries per page directory
#define NPTENTRIES      1024    // # PTEs per page table
#define PGSIZE          4096    // bytes mapped by a page
#define PG_BIT          12      // 4096 byte == 2^12 byte
```

Kalloc.c에 전역변수로 선언

```
// array for reference count of physical addr (PHYSTOP is defined in memlayout.h, PG_BIT in mmu.h)
uint ref_count[PHYSTOP >> PG_BIT];
```

Kmem.len을 kinit1에서 0으로 초기화한다. (main.c에서 kinit1이 kinit2보다 먼저 실행되기 때문)

```
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  // initialize # of free pages as 0
  kmem.len = 0;
  kmem.use_lock = 0;
  freerange(vstart, vend);
}
```

Freerange 함수 내에서 반복문을 돌며 page를 초기화할 때, 해당 물리페이지의 참조 횟수를 0으로 초기화해준다. 추가적으로, free page의 수를 의미하는 kmem.len++; 수행

```
void
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
  {
    // initialize ref_count to 0
    ref_count[V2P(p) >> PG_BIT] = 0;
    kfree(p);
    // calculate # of free pages
    kmem.len++;
  }
}
```

물리 주소가 인자로 전달 되면 비트 연산을 통해 12 비트만 큼 shift right해서 해당 물리 페이지의 참조 횟수를 1 증가 시킨다.

```
// increment ref_count of a certain physical page
void
incr_refc(uint pa)
{
  ref_count[pa >> PG_BIT]++;
}
```

위와 동일한 로직으로 참조 횟수를 1 감소 시킨다.

```
// decrement ref_count of a certain physical page
void
decr_refc(uint pa)
{
  ref_count[pa >> PG_BIT]--;
}
```

위와 동일하게 물리 페이지를 구하고, 해당 물리 페이지의 참조횟수를 나타내는 배열인 ref_count는 uint형이기 때문에 명세에서 요구한 대로 int형으로 캐스팅해 반환한다.

```
// return current ref_count of a certain physical page
int
get_refc(uint pa)
{
  uint count = ref_count[pa >> PG_BIT];
  return (int)count;
}
```

kfree의 경우 인자로 주어진 v에 해당하는 페이지를 free 시키는 함수로, 기존 코드에서 추가한 부분은 get_refc(V2P(v))를 통해 v를 물리 주소로 변환한 후 해당 물리 주소가 속한 페이지의 참조 횟수를 읽어온다. 참조 횟수가 0보다 크다면 해당 주소를 참조 중임을 의미하기 때문에 decr_refc(V2P(v))를 통해 참조 횟수를 감소 시켜준다. 이후 한번 더 참조 횟수를 읽어오고, 해당 값이 0일 경우, 현재 프로세스가 해당 물리 페이지를 참조하는 유일한 프로세스이기 때문에 해당 페이지를 free 시키고, kmem.freelist에 매달아둔다. 이때, free page의 수가 증가했기 때문에 kmem.len을 1 증가 시켜준다.

```
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) ⩾ PHYSTOP)
    panic("kfree");

  if(kmem.use_lock)
    acquire(&kmem.lock);
  // decrement ref count of V2P(v)
  if (get_refc(V2P(v)) > 0) {
    decr_refc(V2P(v));
  }
  // if this is the last reference for a certain physical page
  if(get_refc(V2P(v)) == 0) {
    // Fill with junk to catch dangling refs.
    memset(v, 1, PGSIZE);
    r = (struct run*)v;
    r->next = kmem.freelist;
    kmem.freelist = r;
    // increment free page num by 1
    kmem.len++;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

Kalloc()의 경우 물리 페이지를 할당 받는 함수로 추가한 코드는 if문 안쪽의 코드이다. Freelist가 r->next를 가리키게 하고, r에 해당하는 페이지를 할당해주는데 이때 free page의 수를 의미하는 kmem.len을 1 감소시킨다. 추가적으로, V2P를 통해 r을 물리 주소로 변환해 해당 주소에 똑같이 PG_BIT를 이용한 비트 연산을 적용해여 해당 물리 페이지의 참조 횟수를 1로 설정해준다. (처음, kalloc을 통해 물리 페이지를 할당 받는 경우는 해당 프로세스가 할당 받은 물리 페이지를 가리키는 유일한 프로세스이기 때문)

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r) {
    kmem.freelist = r->next;
    // decrement free page num by 1
    kmem.len--;
    // set ref_count to 1 when first allocated
    ref_count[V2P((char*)r)  >> PG_BIT] = 1;
  }
  if(kmem.use_lock)
    release(&kmem.lock);

  return (char*)r;
}
```

Free page의 수를 반환하는 시스템 콜로 앞서 언급한 kmem.len을 반환한다.

```
// system call for int countfp(void)
int
countfp(void)
{
  return kmem.len;
}
```

2. vm.c

fork() 실행 시 부모 프로세스의 물리 페이지를 복사하는 함수로, 기존엔 해당 함수에서 새로운 물리 페이지를 kalloc()을 통해 할당 받아서 부모의 물리 페이지를 복사했다. 하지만 CoW 구현을 위해서 kalloc()을 통해 물리 페이지를 할당 받는 것이 아닌, 자식 프로세스의 페이지 테이블을 할당 받고, 부모 프로세스의 물리 페이지를 동일하게 가리키기만 하도록 바꿔주었다. (실제 물리 페이지 새로 할당 X 이후 page fault 발생 시 처리 예정) 추가적으로, *pte &= ~PTE_W;를 통해 해당 엔트리의 writable을 disable 시킨다. 또한 mappages를 통해 부모 프로세스와 같은 물리 주소를 가리키고 나면, 해당 물리 주소가 속한 페이지의 참조 횟수를 incr_refc(pa);를 통해 1 증가 시킨다. 이후 마지막으로, 페이지 테이블이 수정되었기 때문에 lcr3(V2P(pgdir));을 통해 TLB를 flush해준다.

```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");

    // change PTE to read-only (disable writeable flag)
    *pte &= ~PTE_W;
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    // just mapping the pa to child process's table without kalloc()
    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) {
      goto bad;
    }
    // increment ref count by 1
    incr_refc(pa);
  }
  // TLB flush
  lcr3(V2P(pgdir));
  return d;

bad:
  freevm(d);
  lcr3(V2P(pgdir));
  return 0;
}
```

Countvp 시스템콜로 해당 프로세스의 유저 메모리에 할당된 가상 페이지의 수를 반환한다. 가상 주소 0에서 struct proc에 저장된 가상 주소 공간까지의 페이지 수를 세서 반환하면 되기 때문에 myproc()->sz를 PGROUNDUP 시킨 뒤 PGSIZE로 나눠 반환한다. 이때 PGROUNDUP을 하는 이유는 현재 프로세스의 가상 주소 공간이 마지막 페이지 전체를 채우지 못했더라도, 해당 페이지의 일부를 사용 중이기 때문에 해당 페이지를 포함하기 위해 PGROUNDUP을 해준다.

```
// function for system call int countvp(void)
// return # of allocated virtual pages for the process's user memory
int
countvp(void)
{
  return PGROUNDUP(myproc()->sz) / PGSIZE;
}
```

Countpp 시스템콜로 페이지 테이블 탐색 후, 유효한 물리 주소가 할당된 page table entry의 수를 반환한다. 이를 위해 가상 주소 0부터 struct proc에 저장된 가상 주소 공간까지 루프를 돌며, 각 엔트리가 유효한 물리 주소에 매핑 되었는지 확인한다. Page directory table부터 page table까지 타고 가 해당 if(*pte & PTE_P)를 통해 해당 엔트리가 유효하다면 count++을 해준다. 이후, count 값을 반환한다. Xv6는 demand paging을 지원하지 않기 때문에 가상 주소는 반드시 물리 주소에 매핑 되어야 해서 countvp와 같은 값을 반환하게 된다.

```
// function for system call int countpp(void)
// return # of page table entries allocated to physical memory
int
countpp(void)
{
  struct proc *curproc = myproc();
  pde_t *pgdir = curproc->pgdir;
  pde_t *pde;
  pte_t *pgtab, *pte;
  uint i;
  int count = 0;
  // count # of page table entries allocated to physical memory
  for(i = 0; i < curproc->sz; i += PGSIZE) {
    pde = &pgdir[PDX(i)];
    if(*pde & PTE_P) { // if pde exists
      pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
      pte = &pgtab[PTX(i)];
      if(*pte & PTE_P) { // if pte exists
        count++;
      }
    }
  }
  return count;
}
```

Countptp 시스템콜은 프로세스의 페이지 테이블에 의해 할당된 페이지의 수를 반환한다. 이때 명세에서 프로세스 내부 페이지 테이블을 저장하는데 사용된 모든 페이지와 페이지 디렉토리에 사용된 페이지도 포함해야 한다고 언급하고 있기 때문에 처음엔 주석처리한 부분을 포함해 코드를 구현했다. 각 페이지 테이블은 총 1024개의 entry를 갖기 때문에 0부터 1023까지 page table을 돌며 해당 엔트리가 할당 되었는지를 확인해 수를 세고 이를 반환한다. 처음에는 page directory table과 page table을 차례로 뒤져서 모든 엔트리를 count를 하도록 구현했지만, test0에서 ptp 값이 66이 나오는 것을 보고, 내부 for loop는 실행하지 않고 page directory entry만을 count하도록 수정했더니 같은 값이 나오는 것을 확인할 수 있었다. (명세에서는 page directory table과 page table 전체를 돌면서 보도록 구현하라는 것처럼 처음엔 이해 했는데, 잘못 이해한 듯하다)

```
// function for system call int countptp(void)
// return every allocated page table entries both in page directory and page table
int
countptp(void)
{
  struct proc *curproc = myproc();
  pde_t *pgdir = curproc->pgdir;
  pde_t *pde;
  //pte_t *pgtab, *pte;
  //int i, j;
  int i;
  int count = 0;
  // total # of entries per table
  int num_entry = 1024;
  // search every allocated page table entries in page directory table and page table
  for (i = 0; i < num_entry; i++) {
    pde = &pgdir[i];
    if(*pde & PTE_P) {
      count++;
      //pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
      //for (j = 0; j < num_entry; j++) {
        //pte = &pgtab[j];
        //if(*pte & PTE_P) {
          //count++;
        //}
      //}
    }
  }
  count++;
  return count;
}
```

Initial sharing을 통해 물리 페이지를 공유하는 프로세스에서 write를 시도하면 page fault가 발생하게 되는데 이를 처리하기 위한 함수이다. 해당 함수에서는 rcr2()를 통해 page fault가 발생한 가상 주소를 읽어온다. 이후 해당 주소가 유효한 값인지 확인한다. Walkpgdir(myproc()->pgdir, (void*)va, 0);을 통해 해당 가상 주소가 가리키는 페이지 테이블 엔트리를 읽어온다. 이후 만약 해당 엔트리가 유효하지 않다면 (페이지 테이블에 매핑 되지 않은 잘못된 범위에 속함을 의미) 에러 메시지를 띄우고 종료 시킨다. 그게 아니라면 PTE_ADDR(*pte);를 통해 해당 엔트리가 가리키는 물리 주소를 읽고 해당 물리 주소가 속한 페이지의 참조 횟수를 읽어온다. 만약 참조횟수가 1보다 크다면, 여러 프로세스가 공유 중임을 의미하기 때문에 kalloc()을 통해 새로운 물리 페이지를 할당 받는다. 이후 해당 페이지에 참조하고 있던 물리 페이지의 컨텐츠를 memmove를 통해 복사하고, 기존 page table 엔트리가 새로 할당 받아서 복사한 물리 페이지를 가리키도록 설정한다. 이때 writable하도록 해주어야 하기 때문에 PTE_P, PTE_U, PTE_W의 모든 flag를 허용해주었다. 이후, 기존에 참조하던 물리 페이지의 참조 횟수를 1 감소시킨다. 반대로 참조 횟수가 1인 경우, 현재 프로세스가 해당 물리 페이지를 참조하는 유일한 프로세스이기 때문에 *pte |= PTE_W를 통해 writable flag만을 추가해주고 사용하면 된다. 마지막으로, 페이지 테이블의 수정했기 때문에 lcr3(V2P(myproc()->pgdir));을 통해 TLB를 flush해준다.

```
// function to handle Cow (Copy on Write) (execute in trap.c)
void
CoW_handler(void)
{
  // read page fault address
  uint va = rcr2();
  if(va < 0){
    panic("Cow_handler va error\n");
    return;
  }
  // find page table entry
  pte_t *pte = walkpgdir(myproc()->pgdir, (void*)va, 0);

  // if va has mapped to page table
  if(*pte & PTE_P) {
    uint pa = PTE_ADDR(*pte);
    int count = get_refc(pa);

    // if there exists multiple processes, which share same physical page.
    if(count > 1) {
      char *mem = kalloc();
      if(mem == 0) {
        cprintf("CoW handler error! Out of memory!\n");
        return;
      }
      // copy contents to mem
      memmove(mem, (char*)P2V(pa), PGSIZE);
      // update pte with newly allocated physical address
      *pte = V2P(mem) | PTE_P | PTE_U | PTE_W;
      // decrement ref_count for pa
      decr_refc(pa);
    }
    // if this is the only process
    else if(count == 1){
      // change page table entry with writeable
      *pte |= PTE_W;
    }
    // TLB flush
    lcr3(V2P(myproc()->pgdir));
  }
  // if va belongs to wrong range
  else {
    cprintf("Error! Virtual adress belongs to wrong range!\n");
    return;
  }
}
```

3. trap.c

이 코드는 trap.c에서 initial sharing 중이던 프로세스가 write를 시도해 page fault 발생 시 이를 handling 하기 위한 코드로 앞서 구현한 CoW_handler()를 호출한다.

```
case T_PGFLT:
  //lazy_allocation();
  // Make a Copy and write
  CoW_handler();
  break;
```

4. defs.h

앞서 구현한 함수와 시스템 콜을 defs.h에 등록해준다.

```
// kalloc.c
char*           kalloc(void);
void            kfree(char*);
void            kinit1(void*, void*);
void            kinit2(void*, void*);
int             countfp(void);
void            incr_refc(uint);
void            decr_refc(uint);
int             get_refc(uint);
```

```
// vm.c
void            seginit(void);
void            kvmalloc(void);
pde_t*          setupkvm(void);
char*           uva2ka(pde_t*, char*);
int             allocuvm(pde_t*, uint, uint);
int             deallocuvm(pde_t*, uint, uint);
void            freevm(pde_t*);
void            inituvm(pde_t*, char*, uint);
int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
pde_t*          copyuvm(pde_t*, uint);
void            switchuvm(struct proc*);
void            switchkvm(void);
int             copyout(pde_t*, uint, void*, uint);
void            clearpteu(pde_t *pgdir, char *uva);
//int           lazy_allocation(void);
int             countvp(void);
int             countpp(void);
int             countptp(void);
void            CoW_handler(void);
```

이후의 사진들은 system call 등록을 위해 추가한 코드들이다. (sysproc.c, syscall.h, syscall.c, user.h, usys.S)

5. sysproc.c

앞에서 구현한 시스템콜을 위한 wrapper function을 정의해주었다.

```
// wrapper function for int countfp(void) in kalloc.c
int
sys_countfp(void)
{
  return countfp();
}

// wrapper function for int countvp(void) in vm.c
int
sys_countvp(void)
{
  return countvp();
}

// wrapper function for int countpp(void) in vm.c
int
sys_countpp(void)
{
  return countpp();
}

// wrapper function for int countptp(void) in vm.c
int
sys_countptp(void)
{
  return countptp();
}
```

6. syscall.h

```
#define SYS_countfp 24
#define SYS_countvp 25
#define SYS_countpp 26
#define SYS_countptp 27
```

7. syscall.c

```
extern int sys_countfp(void);
extern int sys_countvp(void);
extern int sys_countpp(void);
extern int sys_countptp(void);
```

```
[SYS_countfp]   sys_countfp,
[SYS_countvp]   sys_countvp,
[SYS_countpp]   sys_countpp,
[SYS_countptp]  sys_countptp,
```

8. user.h

```
int countfp(void);
int countvp(void);
int countpp(void);
int countptp(void);
```

9. usys.S

```
SYSCALL(countfp)
SYSCALL(countvp)
SYSCALL(countpp)
SYSCALL(countptp)
```

3) Result

테스트를 진행하기 위해 항상 했던 것처럼 p4_test.zip에 있는 test0.c, test1.c, test2.c, test3.c를 xv6 상에 추가해주었다. 이후 Makefile의 UPROGS와 EXTRA 부분에 해당 파일들을 차례로 등록해주었다. 컴파일 및 실행 과정은 make, make fs.img, ./bootxv6를 통해 수행하였다.

1. Test0

Test0의 경우는 4가지 시스템 콜인 countfp, countvp, countpp, countptp을 사용하여 현재 프로세스의 페이지 정보를 확인한다. 이후 sbrk(4096);을 통해 1페이지를 추가로 할당받기 전과 후의 상태에서 각각 시스템콜을 호출해 페이지가 올바르게 할당되었는지 확인한다. 테스트 결과 sbrk(4096); 이전에 그리고 이후에 호출한 countvp()와 countpp()의 값이 같고, 시스템에 존재하는 free page의 수는 1 감소한 것을 확인할 수 있다. 테스트0을 무리 없이 통과할 수 있었다.

```
$ test0
[Test 0] default
ptp: 66 66
[Test 0] pass
```

2. Test1

Test1의 경우는 부모 프로세스에서 fork()를 사용해 자식 프로세스 생성 시 서로가 같은 물리 페이지를 가리키고 있는지 평가하는 테스트이다. 이를 평가하기 위해 countfp()시스템 콜을 사용하여 존재하는 free page의 개수를 비교한다. 이번 테스트 역시 문제 없이 통과할 수 있었고, 이 과정에서 부모 프로세스에서 자식 프로세스를 fork()시 적절하게 initial sharing을 하고 있는 것을 확인할 수 있었다.

```
$ test1
[Test 1] initial sharing
[Test 1] pass
```

3. Test2

Test2의 경우는 부모 프로세스에서 fork()를 통해 자식 프로세스 생성 이후 자식 프로세스에서 공유하고 있는 변수를 수정하려고 할 때 새로운 물리 페이지를 할당 받아서 copy를 진행하는지 확인하는 테스트이다. 공유 중인 변수를 수정하기 전과 후의 free page 수를 countfp()시스템 콜을 통해 반환 받고, 이 둘을 비교해 1개의 물리 페이지가 추가로 할당 됐는지 확인한다. 위 테스트에서 역시 pass를 띄우며 통과할 수 있었고, 이 과정에서 초기 initial sharing 이후 공유 중인 변수를 수정하려고 할 때 적절히 물리 페이지를 copy해 대처하는 것을 확인할 수 있었다.

```
$ test2
[Test 2] Make a Copy
[Test 2] pass
```

4. Test3

Test3의 경우는 test2와 비슷하며 10개의 자식 프로세스를 만든다는 차이가 있다. 이후, 각각의 자식 프로세스에서 공유 중인 변수를 수정할 때 각각 이에 대응해 새로운 물리 페이지를 할당 받아서 copy를 하는지를 평가하는 테스트이다. 추가적으로, 자식 프로세스 종료 후 적절히 free page가 회수되었는지도 평가한다. (부모 프로세스가 자식 프로세스를 만들기 전의 free page 수와 10개의 자식 프로세스가 종료된 후의 free page 수를 비교해 같은 경우 pass) 위 테스트에서 아래의 결과대로 출력되는 것을 확인할 수 있었고, 이 과정에서 앞에서 구현한 대로 10개의 자식 프로세스를 생성 후 수정 발생 시 9개는 새로운 물리 페이지를 할당 받고, 나머지 하나는 PTE_W를 추가해 적절히 대응하는 것을 확인할 수 있었다. 이후 자식 프로세스 종료 시 free page의 회수 역시 적절히 이루어지는 것을 확인했다.

```
$ test3
[Test 3] Make Copies
child [0]'s result: 1
child [1]'s result: 1
child [2]'s result: 1
child [3]'s result: 1
child [4]'s result: 1
child [5]'s result: 1
child [6]'s result: 1
child [7]'s result: 1
child [8]'s result: 1
child [9]'s result: 1
[Test 3] pass
```

4) Trouble Shooting

이번 과제를 하면서 두 번의 고비가 있었다. 처음엔 xv6의 물리 주소의 끝을 나타내는 PHYSTOP이라는 definition이 존재하는지 모르고 물리 페이지의 참조 횟수를 추적하기 위한 배열의 크기를 32bit에서 지원하는 4MB를 각 엔트리에 해당하는 크기인 4Byte로 나눈 1024*1024로 설정했었는데, xv6 부팅 시 계속 재부팅 되는 문제가 발생했다. 처음엔 왜 계속 재부팅 되는지 이유를 알지 못했다. 이후, 코드를 분석해보며 다양한 문제점을 찾아보았고, 재부팅이 발생한 이유가 물리 페이지의 범위 때문이었다는 걸 찾는데 오랜 시간이 들었던 것 같다. 두번째 고비는 아래의 사진과 같이 Booting from Hard Disk.. 에서 멈추는 것이었다. 첫 고비를 넘자마자 위와 같은 현상이 발생해 엄청 고생을 했던 것 같다. 이를 해결하기 위해, 디렉토리를 새로 파 처음부터 구현을 해보았고, 그러던 중 물리페이지의 참조 횟수를 수정하기 위해 선언한 struct spinlock ref_lock;이 문제라는 것을 발견했다. ref_lock을 사용해 참조 횟수를 수정했을 때 위와 같은 현상이 계속 되었고, 이를 없애주었을 때, 올바르게 실행되었다.