Project04

컴퓨터 소프트웨어학부 2022056262 곽주리

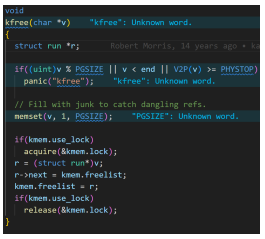
과제목표

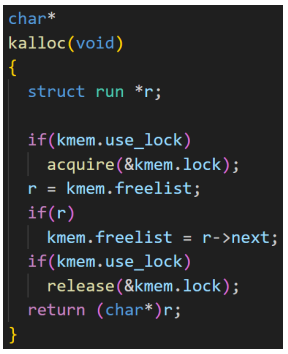
Copy on Write 구현. (물리 주소 및 page table 관리 분석)

1. Initial Sharing
2. Make a copy on write operation

Design

page들의 참조 횟수를 추적하는 데이터 구조를 추가하기 위해 kalloc.c 파일을 살펴보았다.





struct run과 struct kmem을 통해, free frame을 linked list 형태로 관리하는 것을 알 수 있었다. 해당 파일의 모든 함수가 유기적으로 연결되어 있었다.

freerange는 초기에 할당받은 메모리 범위를 page size대로 자른 다음에 kfree 함수를 통해 page들을 freelist에 넣는 함수이다. 따라서 이 때 모든 page의 참조 횟수를 0으로 초기화해주면 된다.

⇒ refc 배열을 사용하기 전에 lock 설정을 위해 초기화 함수를 선언할 것이기 때문에 refc 함수 초기화 함수에서 해당 과정을 진행하였다.

kfree 함수는 해제할 page를 junk로 채우고 freelist의 맨 처음에 해당 page를 넣는다. 따라서 이 함수에서 참조 횟수를 1 감소하면 된다. 또한 참조 횟수가 1 감소 후, 0이 된다면 free 후(memset으로) free list로 반환하면 된다.

kalloc 함수는 freelist의 맨 처음 원소를 빼서 해당 frame을 반환한다. 따라서 이 함수에서 참조 횟수를 1로 설정하면 된다.

page의 시작주소를 알기 위해선 V2P라는 함수를 통해 virtual→physical로 변환하면 된다.

refc 배열 (참조 횟수)

page 참조 횟수를 기록하기 위해 page 갯수 크기의 배열을 만들려고 했는데, kalloc.c에서는 page 갯수를 알 수는 없었다.

따라서 project ppt를 참고하여 mmu.h 파일을 살펴보았고 해당 파일에서 page table 관련 선언을 확인할 수 있었다.

PGSIZE 변수를 사용해서 메모리 크기/PGSIZE로 page 갯수를 구할 수 있을 것 같은데 mmu.h에선 메모리 크기를 알 수 없었다. 구글링을 통해 memlayout.h 파일에 PHYSHOP이라는 이름으로 메모리의 크기가 선언되어 있음을 확인했다. 이 배열은 참조횟수만 저장하고 싶기 때문에 다른 함수에서 이 배열을 수정할 때는 page 번호를 알아야 index로 접근을 할 수 있다.

```
#define PHYSTOP 0xE000000           // Top physical memory
```

page는 같은 크기로, sequential하게 잘리기 때문에 page의 시작주소/page 크기를 통해 page 번호를 알아낼 수 있다.

> 💡 page number= page base addr/pagesize

refc 관련 함수를 구현하기 전에, locking 구현 방식을 생각해보았다. project 3에서 locking 구현을 위해 ptable에서 lock 사용하는 법, spinlock 구조체를 살펴보았던 경험을 토대로 kalloc.c 내에서 spinlock 구조체를 선언할 것이다. 이미 kmem 구조체 안에서 spinlock 구조체를 사용하였기 때문에 구분을 위해 이름만 변경하여 선언하였다.

또한 kinit 함수들에서 lock을 초기화해서 사용하는 것을 참고하여 refc의 lock도 같은 방식으로 초기화해주면 될 것이다.

```
initlock(&kmem.lock, "kmem");
```

Initial sharing

fork를 수행할 때 copy 대신 page를 공유시켜야 한다. 이 때, 페이지의 참조 횟수를 증가시키는 incr_refc 함수를 호출해야 한다. copy 대신 page 공유를 시켜야 하기 때문에 copyuvm 함수를 수정해야 한다고 생각하였다. 읽기 전용으로 표시하는 것은 mmu.h에 PTE_W로 선언되어 있었기 때문에 이 변수를 이용하면 될 것이다. 페이지 권한 수정 후에 필요한 과정들은 과제 명세에 나와있듯 lcr3를 호출하면 된다.

vm.c에 있는 copyuvm 함수를 뜯어보았다. 오류 처리하는 과정을 제외하고, memmove같은 page를 copy하는 부분만 해석하였다.

```
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");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);

    // if((mem = kalloc()) == 0)
    //   goto bad;
    // memmove(mem, (char*)P2V(pa), PGSIZE);

    if (mappages(d, (void*)i, PGSIZE, pa, flags) < 0)
      goto bad;

    incr_refc(pa);
  }
  lcr3(V2P(d));
  return d;

bad:
  freevm(d);
  return 0;
}
```

setupkvm 함수로 page directory를 새로 설정한다. 이건 fork된 자식 process에게도 필요한 과정이라 수정하지 않았다. 가상 주소 공간을 page 단위로 순회하고 kalloc 함수를 이용해서 새로운 페이지를 할당한다. 이 과정 대신 페이지를 공유할 수 있도록 바꿔야 한다. memmove 역시 메모리 내용을 복사하는 것이기 때문에 삭제해주어야 한다.

mappages로 페이지 디렉토리에 새로 할당된 페이지를 넣는데, 부모와 페이지를 공유하게 만들고 페이지 디렉토리에 해당 페이지를 넣어주긴 해야 하기 때문에 삭제하면 안 된다. 다만 기존 copyuvm에서 mappages를 호출할 때, mem이라는 변수를 인자로 전달했는데 kacIloc 함수를 제거하면서 mem을 삭제하게 되어 이 인자를 다른 걸로 변환해야 했다.

V2P(mem)은 새로 할당받는 page의 physical address를 의미하므로 이거 대신 부모 page의 physical address를 인자로 넘기면 된다.

PTE_W를 어떻게 설정해야 권한을 수정할 수 있을지 생각해보았다. PTE_W가 set되면 쓰기 권한이 허용되는 것이므로 not 연산으로 unset한 뒤 이걸 &연산으로 적용해주면 될 것이다.

Make a copy

trap.c 파일에 page fault 발생 시 CoW_handler 함수를 호출할 수 있도록 코드를 추가한다. 이 핸들러에선 user memory의 복사본을 만들어야 한다. 제일 먼저 walkpgdir 함수로 page table entry 주소를 받아온다. walkpgdir 함수는 copyuvm 함수에서 보았듯 가상 주소와 page table을 기반으로 page table의 entry를 반환해준다. 따라서 그 전에 rcr2 함수를 통해 walkpgdir에 전달할 page fault가 발생한 가상 주소를 받아와야 한다.

받아온 가상 주소가 잘못된 범위에 있는지 확인해야 한다. pte가 존재하지 않거나 유효하지 않으면 error 문구를 출력하도록 한다.

그렇게 page table entry를 받아와서 현재 이 page에 몇 개의 process가 접근 중인지 확인하고 그에 따라 수행할 일을 분리하면 된다. 만약 부모 process 외 다른 process도 접근 중이라면 copy본을 만들어야 하는데, 이는 기존 copyuvm 함수에 있는 코드를 사용해서 구현하면 될 것이다.

만약 참조하고 있는 process가 1개이하라면 쓰기 권한을 다시 복구해주면 된다. 이는 copyuvm에서 set했던 것과 반대로 읽기 전용 제한을 제거하면 된다.

구현할 함수

kaclloc.c

`void incr_refc(uint)`

페이지 참조 횟수를 1 증가시키는 함수. 어떤 process가 page에 접근하면 1을 증가시킨다.

kalloc에서 호출할 함수이다. lock으로 경쟁 조건을 피한 후, 인자로 받은 page의 시작 주소를 토대로 page number를 알아낸 후 refc 배열에서 해당 index의 값을 1 증가시킨다. lock 방식은 kalloc.c 내부에서 kmem lock을 사용하는 방식을 참고하였다.

만약 자식 프로세스같은 추가 process가 이미 존재하는 page를 가리킬 때 이 함수를 호출해주어야 한다. 이건 fork 함수나 자식 process가 부모 process의 메모리를 copy할 때 copy 대신 공간 공유를 시킬 때 호출하는 방식으로 구현하면 될 것이다.

`void decr_refc(uint)`

페이지 참조 횟수를 1 감소시키는 함수. locking을 사용하여 refc 배열의 원소를 1 감소하였다.

incr_refc와 동일한 구조이다.

`int get_refc(uint)`

incr, decr과 구조는 유사한데, 이 함수는 참조 횟수를 return해야 하므로, refc 배열의 원소 값을 받아와서 그 값을 반환한다.

vm.c

`void CoW_handler(void)`

```
void CoW_handler(void){
  //user memory 복사본 만들어야 됨. 누군가 공유 page에 write할라 했으니까
  //copy본 줘서 거따 write하라 해야 됨
  //rcr2로 page fault 발생한 주소 받기.
  //받은 주소 유효한 값인지 확인. ptable에 mapping안 된 범윈지 어케 알지
  //그거 아니면 ptable entry 가서 physical address 알아오기. 근데 이거 V2P로 안 되낭
  //만약 page 공유 중이라면 새로운 page 할당~ 그리고 참조 횟수 줄여
  //줄였는데 참조 프로세스 하나면 writable 수정해줘
  //권한 수정했으니까 lcr로 flush
  uint va=rcr2();
  struct proc *proc=myproc();
  pte_t *pte=walkpgdir(myproc()->pgdir,(void*)va,0);
  if(!pte | (!(*pte&PTE_P))){
    panic("error: pte is weird\n");
    return;
  }
  uint pa=PTE_ADDR(*pte);
  uint refc=get_refc(pa);
  char *mem;

  if(refc>1){
    if((mem = kalloc()) == 0){
      panic("failed: kalloc\n");
      return;
    }
```

```
    memmove(mem, (char*)P2V(pa), PGSIZE);
    *pte = V2P(mem) | PTE_FLAGS(*pte) | PTE_W;
    decr_refc(pa);
    lcr3(V2P(proc->pgdir));
  }
  else{
    *pte |= PTE_W;
    lcr3(V2P(proc->pgdir));
  }
  lcr3(V2P(proc->pgdir));
}
```

syscall

kalloc.c

`int countfp(void)`

시스템에 존재하는 free page의 개수를 반환하는 syscall이다. kaclloc.c 안에서 kalloc, kfree를 통해 frame을 할당하고 반환할 때 개수를 계산해서 저장해두면 될 것이다. 원래 이 syscall에서 freelist linked list를 돌면서 개수를 계산하려고 했는데 overhead가 너무 클 것 같아서 방식을 변경하였다.

아무튼 전역변수로 설정한 free page의 개수를 kfree, kalloc 함수에서 수정하고 countfp에선 그 변수를 반환하도록 구현할 것이다.

vm.c

`int countvp(void)`

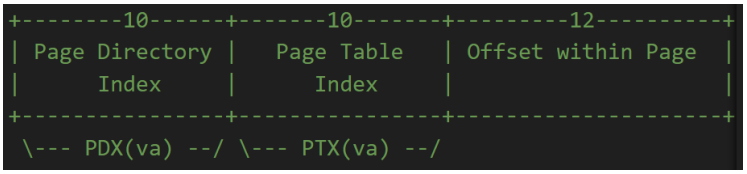
page table을 순회하면서 entry의 valid bit이 set되어 있는 지를 확인하면 된다. 아까 mmu.h을 살펴보았을 때 page table에 관련된 매크로들이 존재한 것을 확인했기 때문에 다시 mmu.h로 가서 ptable의 entry가 valid한지 확인할 수 있는 요소가 있는지 알아보았다.

Page table entry flags로 PTE_P라는 매크로가 있다. 이게 Present를 알려주는 flag이므로 valid bit라고 판단하였다.

또한 NPDENTRIES라는 매크로가 page directory에 들어 있는 entry 수, NPTENTRIES는 page table의 entry 수를 의미하는 것을 보아 xv6는 2단계 계층 구조로 이루어져 있다고 추측할 수 있다.

따라서 countpp 안에서, page directory를 순회하면서 각 entry가 유효하다면, page table을 순회하면서 page table의 entry도 유효한 지 확인하면 된다. 둘 다 통과하면 유효한 물리 주소가 할당된 page table entry의 수라는 뜻이므로 이 때 pp를 증가시켜주면 될 것이다.

page entry를 돌면서 가상 주소 공간 크기를 벗어나는지 확인해야 한다. 가상 주소 공간의 크기는 project 3에서 사용했듯 sz라는 member에 저장되어 있으므로 sz와 크기 비교를 하며 순회하면 된다. sz와 크기를 비교할 가상 주소를 계산하기 위해 mmu.h에 있던 주석을 참고하였다.



pdx는 페이지 디렉토리 entry 번호를 shift하는 거라 i를 shift, ptx도 같은 이유로 j를 shift하면 될 것 같은데 offset은 어떻게 구해야 할 지 감이 안 잡혔다.

그런데 이 함수에서 구하고자 하는 것은 가상 페이지의 수기 때문에 offset을 구하는 것은 불필요한 과정이라는 생각이 들어, pdx ptx로만 가상 주소를 계산하여 sz와 비교해주었다. sz를 넘어가는 순간 지금까지 센 페이지 수를 return하면 되고 sz를 넘어가지 않으면 다 계산한 뒤에 vp를 return해주면 될 것이다.

`int countpp(void)`

가상 주소 공간을 돌면서 page table의 entry를 받아, 해당 entry가 유효하다면 pp를 증가시켜주면 된다. 가상 주소 공간은 memlayout.h에 정의되어 있듯 KERNBASE가 커널 공간이므로 그 전까지만 돌면 된다. page table의 entry는 walkpgdir 함수

로 받아오면 된다.

`int countptp(void)`

page table 디렉토리를 순회하면서 유효한 page가 할당되어 있을 때 ptp를 1씩 증가해주면 된다. page table 디렉토리를 저장한 페이지도 추가해야 하므로 초기 ptp를 1로 설정해야 한다.

Implement

초기 세팅

`int refc[PHYSTOP/PGSIZE]`, `struct spinlock refc_lock`, `int fp`

```
int refc[PHYSTOP/PGSIZE];
struct spinlock refc_lock;
int fp;
```

메모리 크기/page size로 page 갯수를 구해서 해당 크기의 배열을 선언해주었다.

refc에 접근할 때 locking을 사용해야 하므로 refc_lock이라는 spinlock 구조체도 선언해주었다.

free frame의 개수를 저장하기 위한 전역변수도 선언해주었다.

`void refc_init(void)`

```
void refc_init(void){
  initlock(&refc_lock,"refc");
  for(int i=0; i<(PHYSTOP/PGSIZE); i++) re
}
```

kinit에서 lock을 초기화하는 것을 참고하여 refc 배열 초기화를 위한 함수도 선언해주었다.

초기엔 참조 횟수가 다 0이므로 이 함수에서 원소를 다 0으로 설정해주었다.

구현할 함수

kalloc.c

`void incr_refc(uint)`

```
void incr_refc(uint){
  acquire(&refc_lock);
  refc[uint/PGSIZE]++;
  release(&refc_lock);
}
```

`void decr_refc(uint)`

```
void decr_refc(uint){
  acquire(&refc_lock);
  refc[uint/PGSIZE]--;
  release(&refc_lock);
}
```

`int get_refc(uint)`

```
int get_refc(uint){
  acquire(&refc_lock);
  int counter=refc[uint/PGS
  release(&refc_lock);
  return counter;
}
```

이 함수들은 design part에서 고안하였던 방식을 그대로 적용하였기 때문에 자세한 설명은 생략하였다.

kacllloc.c에서 해당 함수들을 사용하기 위해, kfree, kalloc, kinit1, kinit2 함수를 수정해주었다.

`void kinit1(void *vstart, void *vend)`

```
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  fp=0;
  freerange(vstart, vend);
}
```

`void kinit2(void *vstart, void *vend)`

```
void
kinit2(void *vstart, void *vend)
{
  freerange(vstart, vend);
  kmem.use_lock = 1;
  refc_init();
}
```

맨 처음, free list에 page들을 넣기 전에 free page 개수를 0으로 초기화하였다. page table 완성 후에 page들을 free list에 추가한 뒤 refc 배열 초기화 함수를 호출했다.

`void kfree(char *v)`

```
void
kfree(char *v)
{
  ...

 if(kmem.use_lock)
    acquire(&kmem.lock);

  //1 줄였는데 0됐으면 free하고 freelist로~~
  if(refc[V2P(v) / PGSIZE]==0){
    //여기서부터 frame junk로 초기화하고 free로 넣는 코드
    //이 코드는 counter가 0일 때만 진행하면 된다.
    // Fill with junk to catch dangling refs.
    memset(v, 1, PGSIZE);
    //반환할 page 받아서 free list의 원소로 casting하고
    r = (struct run*)v;
    //반환할 page를 kmem의 freelist의 첫 번째에 넣을 준비~~ 반환될 page가 젤 앞
    r->next = kmem.freelist;
    //반환된 page가 젤 앞이니까 freelist의 시작부분 수정~~
    kmem.freelist = r;
    fp++;
  }

  if(kmem.use_lock)
    release(&kmem.lock);
  }
```

기존엔 그냥 free, freelist에 추가하였는데 지금은 참조 횟수가 0인 경우에만 해당 과정을 진행할 수 있기 때문에 조건문을 추가하였다. page의 실제 주소를 refc 관련 함수에 넘겨야 하는데, 이는 V2P 함수를 사용하였다. 기존 kfree 함수에서 char *v 형태 그대로 함수에 보내는 것을 참고하였다.

`char* kalloc(void)`

```
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
 if(r){
        fp--;
    kmem.freelist = r->next;
    refc[V2P((char*)r)/PGSIZE]=1;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
```

```
  return (char*)r;
}
```

freelist에서 할당받을 frame이 있다면 freelist를 변경한 후에 refc 배열에서 참조횟수도 1로 set해준다. 이 때도 page의 실제 주소를 보내야 하는데, 여기선 r이 구조체이기 때문에 char*형태로 casting하여 보냈다.

vm.c

`void CoW_handler(void)`

```
void CoW_handler(void){
  //user memory 복사본 만들어야 됨. 누군가 공유 page에 write할라 했으니까
  //copy본 줘서 거따 write하라 해야 됨
  //rcr2로 page fault 발생한 주소 받기.
  //받은 주소 유효한 값인지 확인. ptable에 mapping안 된 범윈지 어케 알지
  //그거 아니면 ptable entry 가서 physical address 알아오기. 근데 이거 V2P로 안 되낭
  //만약 page 공유 중이라면 새로운 page 할당~ 그리고 참조 횟수 줄여
  //줄였는데 참조 프로세스 하나면 writable 수정해줘
  //권한 수정했으니까 lcr로 flush
  uint va=rcr2();
  struct proc *proc=myproc();
  pte_t *pte=walkpgdir(myproc()->pgdir,(void*)va,0);
  if(!pte || (!(*pte&PTE_P))){
    panic("error: pte is weird\n");
    return;
  }
  uint pa=PTE_ADDR(*pte);
  uint refc=get_refc(pa);
  char *mem;

  if(refc>1){
    if((mem = kalloc()) == 0){
      panic("failed: kalloc\n");
      return;
    }
    memmove(mem, (char*)P2V(pa), PGSIZE);
    *pte = V2P(mem) | PTE_FLAGS(*pte) | PTE_W;
    decr_refc(pa);
  }
  else{
    *pte |= PTE_W;
  }
  lcr3(V2P(proc->pgdir));
}
```

이 함수는 trap 발생 시 호출되어야 하는 함수이다. 따라서 trap.c 파일에서 page fault 발생시 이 함수를 호출하도록 수정해야 한다. 다만 CoW_handler는 trap.c 대신 vm.c 파일에 구현하였다.

design part에서 설명했던 방식 그대로 구현하였다. rcr2로 page fault가 발생한 가상 주소를 받은 바음 그 주소를 이용해서 walkpgdir 함수로 page table entry를 받아온다. pte가 올바른 범위에 있는지 확인하고 올바르다면 page table entry의 주소와 해당 page를 참조하고 있는 프로세스의 개수를 확인한다. 1개 이상이라면 copy본 만들어서 할당하고 기존 page를 참조하는 process의 수를 1 감소시킨다. 그렇지 않다면 그냥 쓰기 권한만 수정해주면 된다. page table 정보를 수정했기 때문에 lcr3 함수를 호출해준다.

trap.c

```
//trap.c에서 trap 함수 발췌
switch(tf->trapno){
  case T_PGFLT:
    CoW_handler();
    break;
```

기존 case들에 T_PGFLT case를 추가하여 CoW_handler 함수를 호출해주었다.

syscall

kalloc.c

`int countfp(void)`

int fp라는 변수를 전역 변수로 설정해서 kinit1에서 fp를 0으로 초기화하고 frame을 할당할 땐 fp를 1 감소, frame을 해제할 땐 1 감소하도록 다른 함수를 수정해주었다.

따라서 countfp syscall에선 전역 변수인 fp 변수를 return만 해주면 된다.

```
int countfp(void){
  return fp;
}
int sys_countfp(void){
  return countfp();
}
```

vm.c

`int countvp(void)`

```
int countvp(void){
  struct proc *proc=myproc();
  pde_t *pgdir=proc->pgdir;
  uint sz=proc->sz;
  int vp=0;

  for(int i=0; i<NPDENTRIES; i++){
    //페이지 디렉토리에서 entry 받아 오기 페이지 테이블 위치
    pde_t pde=pgdir[i];
    if(pde&PTE_P){
      pte_t *ptable=(pte_t*)P2V(PTE_ADDR(pde));
      for(int j=0; j<NPTENTRIES; j++){
        uint va=(i<<PDXSHIFT)|(j<<PTXSHIFT);
        if (va>=sz) return vp;
        pte_t pte=ptable[j];
        if(pte&PTE_P) vp++;
      }
    }
  }
  return vp;
}
int sys_countvp(void){
  return countvp();
}
```

페이지 디렉토리를 돌면서 entry가 유효하다면 entry에 들어 있는 값을(inner page table의 주소) 받아와서 커널이 접근할 수 있도록 가상 주소로 변환한 후 page table의 entry를 순회한다. 순회하면서 현재 가상 주소와 가상 주소 공간 크기를 비교해, 범위 안에 있을 때 entry가 유효한지 확인한다. 유효하다면 user memory에 할당된 가상 페이지이므로 vp를 1 증가한다.

`int countpp(void)`

```
int countpp(void)
{
  struct proc *proc=myproc();
  pde_t *pgdir=proc->pgdir;
  int pp=0;
  pte_t *pte;
  uint va;
  for(va=0; va<KERNBASE; va+=PGSIZE) {
    pte=walkpgdir(pgdir,(const void*)va,0);
    if(pte&&(*pte&PTE_P)) {
      pp++;
    }
  }
  return pp;
}
int sys_countpp(void){
  return countpp();
}
```

design part에서 설명한 방식을 그대로 구현하였다. 큰 틀은 countvp와 유사한데, countpp와 달리 가상 주소 공간을 순회하면서 entry에 접근했다.

`int countptp(void)`

```
int countptp(void){
  struct proc *proc=myproc();
  pde_t *pgdir=proc->pgdir;
  int ptp=1;
  for (int i=0; i<NPDENTRIES; i++) {
    pde_t pde=pgdir[i];
    if (pde&PTE_P) {
      ptp++;
    }
  }
  return ptp;
}
int sys_countptp(void){
  return countptp();
}
```

design part에서 설명했던 방식을 그대로 적용하였다. page table 디렉토리를 저장하기 위해 사용된 page는 1개이므로 ptp를 1로 초기화하고 디렉토리의 entry를 돌면서 유효한지 확인하고, 유효하다면 ptp를 증가해주었다. outer page table을 저장하는 데 필요한 page가 1개라는 사실은 이론 시간 때 배운 사실이다. 만약 page가 2개 이상 필요하다면 이 page table 디렉토리를 가리키기 위한 상위 page table이 필요한데 mmu.h에서 확인했을 때 2단계로만 구성되어 있었기 때문에 1개로 설정해주면 된다.

위와 같은 syscall은 syscall.c, syscall.h 등 필요한 file에 등록하고 makefile에도 추가하는 등의 과정을 거쳤다. 그 외 함수들도 defs.h, makefile, user.h 등에 추가해주었다.

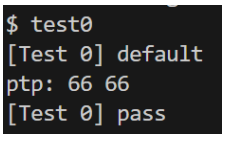
Initial Sharing

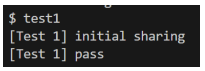
`pde_t* copyuvm(pde_t, *pgdir, uint sz)`

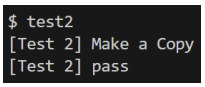
```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;

  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");
    pa = PTE_ADDR(*pte);
    *pte&=~PTE_W;
    flags = PTE_FLAGS(*pte);

    // if((mem = kalloc()) == 0)
    //   goto bad;
    // memmove(mem, (char*)P2V(pa), PGSIZE);


    if (mappages(d, (void*)i, PGSIZE, pa, flags) < 0) {
      goto bad;
    }
    incr_refc(pa);
  }
  lcr3(V2P(pgdir));
  return d;

bad:
  freevm(d);
  return 0;
}
```

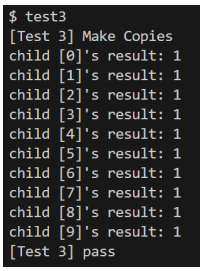
새로운 page를 할당하는 부분을 제거하고 flag에 들어가는 PTE_W를 바꿔줘야 하기 때문에 bit 연산을 통해 flag를 수정하였다. 또한 memory copy 부분은 제거하고 mappages에서 physical address를 부모 page의 physical address인 pa로 변경해주었다.

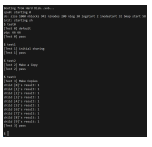
Result











test 0~3 모두 pass를 출력하였다. ppt에서 제시한 예시와 동일하다.

Trouble shooting

처음에 kalloc을 할 때마다 1씩 증가시키는 걸로 이해해서 ++로 구현하려고 하였다.

⇒ kalloc은 말 그대로 free frame을 할당해주는 건데 이미 존재하는 page에 접근하는 경우는 kalloc이 실행되지 않는다는 것을 깨달았다. 따라서 추가 process가 이미 존재하는 page를 가리킬 때 "증가"하고 그 외는 1로 설정하도록 방식을 변경하였다.

xv6 부팅하면, Booting...에서 멈춰 있는 문제 발생

cprintf문을 넣어서 디버깅, 각종 lock 주석처리하면서 디버깅 해본 결과 deadlock kmem lock을 잡고 refc lock을 잡아서 생긴 문제였다. 다른 종류의 lock이라 문제가 없을 거라 생각했으나 순서를 수정하여 double lock 조건을 제거해주니 deadlock이 풀리면서 정상 실행되었다.

count 함수 오류

```
$ test0
[Test 0] default
countfp done
countvp done
countpp done
countptp done
ptp: 65604 65605
[Test 0] fail
```

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

```
$ test0
[Test 0] default
numvp: 3 numpp: 65539
numvpa: 4 numppa: 65540
numfp: 56734 numfpa: 56733
ptp: 66 66
[Test 0] fail
```

디버깅을 통해 어떤 값이 문제인지 확인해보았다. ptp를 셀 때 inner page table의 entry는 셀 필요 없는데 inner page table의 entry까지 더하면서 값이 이상해진 것이었다. countptp의 결과는 올바르게 나오는데 fail이 떠서 확인해보니 countpp의 결과값이 이상했다. countpp는 countvp와 달리 가상 주소 공간 크기를 고려하지 않고 작성하였는데 그게 문제일까 싶어 sz와 비교해서 가상 주소 공간 넘어가면 끝내도록 코드를 수정했더니 올바른 결과가 나왔다.

free frame 수는 sbrk 이후 page 하나를 할당해서 1 감소가 올바르게 출력되었는데, free frame 개수 자체가 올바르게 나온 건지 확인하기 위해서 kinit, freerange 이후 free frame의 개수를 출력해보았다.

```
Booting from Hard Disk..xv6...
freerange: freed 56320 pages from 80400000 to 8e000000
kinit2: total free pages after freerange = 56945
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test0
[Test 0] default
numfp: 56734 numfpa: 56733
ptp: 66 66
[Test 0] pass
```

초기화 후 free frame의 수가 56945개고 test0 실행 후 free frame은 56734로 free frame이 증가하는 등의 이상한 양상을 보이지 않았기 때문에 올바른 출력이라고 생각한다.