컴퓨터소프트웨어학부
2022074057
홍바다

Project04 - Wiki

I. Design

A. CoW - Initial Sharing

Copy-on-Write(CoW)는 메모리를 효율적으로 사용하기 위해 fork로 자식 프로세스를 만들 때 먼저 부모 프로세스의 페이지를 공유 받아 접근할 수 있게 한다. 만약 한 프로세스가 수정하려고 할 때에만 페이지가 복사된다. 이렇게 하면 필요시에만 하나씩 분할되어 overhead를 줄이는 동시에 사용자에게 똑같은 효과를 주게 된다. 이를 구현하기 위해서는 원래 fork를 할때 바로 부모 프로세스를 복제하여 새로운 프로세스를 생성했던 것을 일단 처음에는 같은 페이지 테이블을 공유하도록 수정해줘야한다. 그래서 먼저 proc.c에 있는 fork 함수를 보았다. copyuvm을 통해 프로세스를 복제하는 것을 볼 수 있었다. 그리하여 copyuvm의 로직을 바꿔야겠다고 생각하였다. 그리고 자식 프로세스가 부모 프로세스의 페이지를 참조할 수 있도록 물리 페이지의 참조 횟수 관리를 위해 kalloc.c도 수정해야한다. 이를 위해 명세에 나와 있듯이 kmem 데이터 구조에 필드 추가가 필요해보였다. 또한, 명세에 주어진 함수를 이용하여 참조 횟수를 관리하는 것도 필요하다.

kalloc.c와 vm.c를 살펴보며 지금껏 자주 사용하지 않았던 definition이나 macro가 보였다. PTE_P, PTE_W, PTE_UPTE_ADDR, P2V(), V2P() 등의 정의와 쓰임새를 mmu.h와 memlayout.h를 통해 알아두어 구현할때에 도움이 되었다.

본격적으로 구현하기 전에 kalloc.c에서 어디에 어떠한 형식으로 참조 횟수를 추적할 변수를 만들지 고민을 해보았다. 실제 물리 페이지에 대해 참조 횟수를 제어해야하기 때문에 각 페이지마다 관리할 수 있도록 전체 물리 페이지 수만큼의 배열이 필요해보였다. 전체 페이지 사이즈의 배열을 할당하기 위해 memlayout.h에 아래처럼 정의 되어있는 definition을 사용하였다.

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

PHYSTOP은 physical memory의 upper bound이다. 이는 커널이 물리주소를 관리하는 최대치를 뜻한다. 이 최대치를 이용하여 배열을 만들어주었다. 그리고 연속적으로 있는 물리 주소들은 페이지 사이즈를 기준으로 분할된다. 아래와 같이 mmu.h에서 PGSIZE를 확인하니 각 페이지가 4096 bytes로 이루어진걸 볼 수 있다.

```
#define PGSIZE          4096    // bytes mapped by a page
```

PGSIZE = 4096 bytes = 2^{12} 이므로 전체 물리주소를 12로 나누면 실제 물리주소 수만큼의 배열을 생성할 수 있다. mmu.h에 정의된 definition을 보면 아래와 같이 정의된 PTXSHIFT를 볼 수 있다.

```
// 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 PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address
```

해당 설명을 보면 linear address, 즉 virtual address에서 page table index (PTX)의 offset을 뜻한다는 것을 알 수 있다. 이 bitwise shift를 사용하여 나눗셈을 효과적으로 하기 위해 "PHYSTOP >> PTXSHIFT"를 해준다. 여기서 나눗셈(/)을 사용하지 않고 biwise shift를 사용한 이유는 프로세서가 직접적으로 지원하는 명령어는 bitwise shift이므로 더 효율적이고 빠르게 계산할 수 있기 때문이다. 이렇게 정의한 배열을 이용하여 전체 물리 페이지에 대해 각각의 참조 횟수를 관리해준다. 참조 횟수에 대한 변수는 kalloc.c에 있으므로 관련 함수들인 incr_refc(), decr_refc(), 그리고 get_refc()도 kalloc.c에 정의해주는 것이 좋아보였다.

B. CoW - Make a Copy

페이지 공유를 완료한 후에 만약 한 프로세스가 데이터 수정을 하려하면 페이지 복사를 해줘야한다. 이를 위해 trap handling이 필요하다. 실습 시간에 해보았던 lazy allocation을 기반으로 trap.c에서 handling을 해주도록 케이스를 덧붙이고, 새로운 함수(CoW_handler)를 이용하여 페이지 폴트 처리를 해주어야한다. 이 함수에서 복사본을 만들고, 마지막 프로세스인 경우에 대해서도 따로 처리해주면 된다. 여기에서도 참조 횟수를 고려해야한다. 그리고 페이지 테이블 항목을 수정할 때마다 TLB 항목들을 flush, 즉 update를 해주는 것을 주의하여 구현해야 한다.

또한, 명세에 따라 4개의 시스템 콜을 구현하기 위해 이전 프로젝트들 처럼 함수를 등록해주고 유저 프로그램에서도 사용할 수 있도록 해주었다.

II. Implement

1. CoW - Initial Sharing

먼저 kalloc.c부터 수정해주었다. copyuvm에서 fork 할 때 자식 프로세스를 부모 프로세스의 페이지를 참조하게 작동된다는 것을 가정하고 작성하였다. (copyuvm 구현은 후에 설명할 것이다.)

```
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
  uint ref_count[PHYSTOP >> PTXSHIFT]; //array to count reference number for each page
} kmem;
```

→ 위에서 설명하였듯이, 참조횟수를 관리하는 배열을 kmem 구조체에 추가해주었다. kmem에서는 lock을 사용하며 물리 메모리에 대해 관리하므로 여기에 추가해주었다. PTXSHIFT는 위에서 설명하였듯이 PGSIZE 값에 따라 12로 mmu.h에서 정의된 것이다. 각 물리 페이지의 참조 횟수를 관리하는 ref_count는 아래처럼 정의하였다.

```
void
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE){
    kmem.ref_count[V2P(p) >> PTXSHIFT] = 0; //initialize the ref_count to 0
    kfree(p);
  }
}
```

→ freerange는 kinit1과 kinit2에서 사용된다. 시스템이 시작될때 ref_count를 0으로 초기화 시키기 위해 freerange에서 각 페이지에 대해 0으로 초기화하는 코드를 추가해주었다. 여기서 V2P()는 virtual address를 physical address로 바꿔준다. (ref_count 배열은 가상 주소가 아닌 물리 주소를 기준으로 관리하기 때문이다.)

```
//PAGEBREAK: 21
// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
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);
  r = (struct run*)v;

  //edited
  uint pa = V2P(v);
  if(kmem.ref_count[pa >> PTXSHIFT] > 0){ //check if referencing process is left
    kmem.ref_count[pa >> PTXSHIFT]--; //decrement the number of processes referencing this page
  }
  if(kmem.ref_count[pa >> PTXSHIFT] == 0){
    // Fill with junk to catch dangling refs.
    memset(v, 1, PGSIZE);
    r->next = kmem.freelist;
    kmem.freelist = r;
  }

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

→ 프로세스가 더이상 페이지를 가리키지 않을 때에 대해 처리해주었다. 만약 참조하는 프로세스가 남아있는 경우, 즉 ref_count가 0보다 클 경우 참조 횟수를 감소해주었다. 만약 참조 횟수가 0일 때, 즉 더 이상 페이지를 참조하는 프로세스가 없을 경우에만 페이지를 free하고 freelist로 반환하도록 수정해주었다.

```
// 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;
    kmem.ref_count[V2P((char*)r) >> PTXSHIFT] = 1; // since allocated one
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

→ free page가 어떤 프로세스에 의해 할당되었을 때에 대한 상황을 처리해준다. 한 프로세스가 해당 페이지를 참조하기 시작했다는 의미로 페이지의 참조 횟수를 1로 설정해주었다.

```
void
incr_refc(uint pa){
  if(kmem.use_lock) // using lock
    acquire(&kmem.lock);
  kmem.ref_count[pa >> PTXSHIFT]++; // increasing ref_count
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

→ locking을 이용하여 참조 횟수의 동기화를 보장해주고, 인자로 받은 해당 페이지의 ref_count를 1 증가해주는 함수이다.

```
void
decr_refc(uint pa){
  if(kmem.use_lock) // using lock
    acquire(&kmem.lock);
  kmem.ref_count[pa >> PTXSHIFT]--; //decreasing ref_count
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

→ incr_refc()처럼 locking을 이용하였고, 인자로 받은 해당 페이지의 ref_count를 1 감소시켜주었다.

```
int
get_refc(uint pa){
  int refc;
  if(kmem.use_lock)
    acquire(&kmem.lock);
  refc = kmem.ref_count[pa >> PTXSHIFT]; //get the value of ref_count
  if(kmem.use_lock)
    release(&kmem.lock);
  return refc;
}
```

→ get_refc() 또한 locking을 이용하며 해당 페이지의 ref_count 값을 저장 후 리턴해주어 페이지를 참조 하고 있는 프로세스 수를 반환해준다.

위 세 함수들 모두 ref_count에 관련된 것이기 때문에 kalloc.c에서 정의해주었다.

다음은 fork에서 부모 프로세스의 페이지를 복제하는 것이 아닌 자식 프로세스와 페이지를 공유할 수 있도록 해주기 위해 vm.c를 수정해주었다. 앞서 설명하였듯이, fork에서는 copyuvm을 통해 프로세스의 페이지를 복제하였다. vm.c의 copyuvm을 수정하여 두 프로세스가 같은 페이지를 참조하도록 해줄 것이다.

```
// 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");

    //edited for project4
    *pte &= (~PTE_W); //read-only for sharing page
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) { //mapping with the page
      goto bad;
    }
    incr_refc(pa); //increase ref_count of the page
  }

  lcr3(V2P(pgdir)); //flush TLB
  return d;

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

→ 자식 프로세스를 생성할 때 자식 프로세스는 새로운 페이지 테이블을 할당 받고, 부모 프로세스와 같은 물리 페이지를 가리켜야한다. 다수의 페이지 테이블이 물리 페이지를 mapping하기 때문에 해당 페이지의 ref_count를 증가시켜줘야한다. 페이지를 공유하기 때문에 페이지는 읽기 전용이어야한다. 그렇기에 PTE_W를 이용하여 읽기 전용 페이지로 만들어준다. 만약 나중에 쓰기 접근이 발생하면 커널로 trap되도록 해준다. 그리고 기존의 mem = kalloc()와 memmove() 코드를 지워 fork시 페이지를 복제하지 못 하도록 해주었다. 그 대신 mappages를 이용하여 parent process의 페이지 주소인 pa를 통해 매핑시켜주었다. 무사히 참조를 끝냈으면 incr_refc를 통해 pa의 ref_count를 1 증가해주었다. 마지막으로 TLB를 flush 해주기 위해 lcr3(V2P(pgdir))를 사용하였다. 이는 매핑에 실패했을 경우에도 이미 페이지 권한 변경으로 부모의 페이지 테이블이 수정되었으므로 TLB flush를 해주었다.

2. CoW - Make a Copy

새로운 프로세스가 생성되었을 때의 상황에 대한 코드 수정을 완료하였고, 이제 실제로 어떤 프로세스가 읽기 전용인 페이지에 쓰기 접근을 할 때에 대한 상황에 대해 구현할 것이다. 먼저,

읽기 전용 페이지에 쓰기를 시도하면 page fault가 일어난다. 이와 같은 exception handling은 trap.c에서 처리한다. 현재 T_PGFLT에 대한 예외 처리는 되어 있지 않으므로 trap.c를 수정해줄 것이다.

```
case T_PGFLT:
  CoW_handler();
  break;
```

→ 실습 때 진행한 lazy allocation에 대한 예외 처리를 해준 것처럼, 여기에서도 T_PGFLT에 대한 케이스를 추가하여 예외 처리를 해주는 함수를 불러주었다. CoW_handler()는 여러 static 함수 사용을 위해 vm.c에 직접 추가해주었다.

```
//edited for project4
void
CoW_handler(void){
  uint va = rcr2();
  pte_t *pte;
  uint pa, refc;

  if(va >= KERNBASE || va == 0){ // if va is not a valid user space address
    panic("CoW_handler: Invalid access");
    myproc()->killed = 1; //terminate the process
    return;
  }

  pte = walkpgdir(myproc()->pgdir, (void*)va, 0);
  if(pte == 0){
    cprintf("CoW_handler: pte should exit");
    myproc()->killed = 1; //terminate the process
    return;
  }
  if((*pte & PTE_P) == 0){
    cprintf("CoW_handler: pte is not present");
    myproc()->killed = 1; //terminate the process
    return;
  }
  if((*pte & PTE_U) == 0){
    cprintf("CoW_handler: pte is not a user page");
    myproc()->killed = 1; //terminate the process
    return;
  }
```

→ vm.c에 해당 함수를 추가해주었다. 원래는 copyuvm에서 부모 프로세스의 페이지를 복제하도록 한다. 그리하여 기존 copyuvm의 코드를 참고하여 CoW_handler를 구현했다. 먼저 페이지 폴트가 발생했으니 rcr2()를 통해 폴트가 발생한 가상 주소를 저장한다. 그 후 이 가상주소가 valid한 user address인지 확인하기 위해 va >= KERNBASE를 통해 커널 space에 포함되어있는지 확인하고, va이 null이 아닌지 확인해준다. 만약 잘못된 범위에 속해 있다면 접근 불가능한 범위이므로 에러 메시지와 함께 프로세스를 종료해준다. 그 다음 walkpgdir를 통해 va에 해당하는 page table entry를 찾는다. 만약 pte를 찾지 못한다면 가상 주소가 해당 페이지 테이블에 매핑되어 있지 않다는 의미이므로 에러 메시지와 함께 종료시킨다. 또한,

pte가 존재하지 않는 경우와 유저 페이지 범위에 존재하지 않다면 잘못된 범위에 속해있는 것이므로 에러 메시지 출력과 동시에 프로세스를 종료해준다.

```
  pa = PTE_ADDR(*pte);
  refc = get_refc(pa);

  if(refc > 1){
    char *mem = kalloc(); // allocating new page
    if(mem == 0){
      cprintf("CoW_handler: Out of memory");
      return;
    }
    memmove(mem, (char*)P2V(pa), PGSIZE); // copy page contents
    *pte = V2P(mem) | PTE_P | PTE_W | PTE_U; // update page table entry
    decr_refc(pa); // decrement ref_count of original page
  }
  else if(refc == 1){ // if it's the last process, remove read-only restriction from the page
    *pte |= PTE_W;
  }
  lcr3(V2P(myproc()->pgdir)); // flush TLB
}
```

→ 가상 주소에 대한 예외 처리를 해준 후, 이제 해당 페이지의 복제를 진행해야 한다. 먼저 pte의 물리 주소를 구하고, 참조 횟수 값을 저장해주었다. 해당 페이지를 공유하는 프로세스 수가 여러개라면 각각 별도의 페이지 복사본을 생성해줘야한다. 그러기 위해 refc > 1을 통해 마지막 프로세스인지 확인 후 진행해주었다. 만약 마지막 프로세스가 아니라면, 기존 copyuvm의 코드처럼 kalloc을 통해 새로운 페이지를 생성하고, memmove를 통해 내용을 복사하여 복사본을 만들면 된다. 여기서 kalloc이 실패하는 경우에 대해서도 예외 처리를 해주었다. 그리고 복제를 완성하였으면 이제 쓰기도 가능하므로 그에 대한 page table entry를 수정해준다. 그리고 이제 더이상 기존 페이지를 참조하지 않으므로 decr_refc를 통해 참조 횟수를 1 감소해준다. 반면, 만약 페이지를 참조하는 프로세스가 마지막이라면, 즉 refc == 1이라면, 단순히 페이지의 쓰기 접근을 허용해준다. 이렇게 page table entry를 변경하였으므로 lcr3(V2P(myproc()->pgdir))를 통해 TLB flush를 해준다.

이제 Copy-on-Write에 대한 함수 구현 및 수정은 끝났다. 이제 유저 프로그램에 필요한 시스템 콜을 구현해보자.

```
int
countfp(void){
  int count = 0;
  if(kmem.use_lock)
    acquire(&kmem.lock);
  for(struct run *r = kmem.freelist; r; r = r->next){
    count++;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return count;
}
```

→ 시스템에 존재하는 free page의 수를 세어주고 값을 반환해준다. 그러기 위해 적절한 kmem 구조체의 freelist에 접근을 해야했고, 이를 위한 적절한 locking 사용을 위해 kalloc.c에 해당 함수를 추가해주었다. 위 코드를 보면 kmem.freelist가 끝날때까지 count 값을 늘려 개수를 세주었다.

countfp()를 제외한 다른 세 함수들은 walkpgdir와 같은 static 함수 사용을 위해 vm.c에 추가해주었다:

```
int countvp(void){
  int count = 0;
  struct proc *curproc = myproc();

  count = curproc->sz / PGSIZE;
  if((curproc->sz % PGSIZE) != 0) //increment count for any partial memory to count as a full page
    count++;

  return count;
}
```

→ countvp()는 현재 프로세스의 user memory에 할당된 가상 페이지의 수를 센다. 가상 주소 0부터 시작해서 현재 프로세스의 주소 공간 사이즈만큼까지 확인하기 위해 현재 프로세스의 user memory의 사이즈, 즉 curproc->sz를 페이지 사이즈인 PGSIZE로 나누었다. 그러면 총 페이지 수가 나오지만, 여기서 나눗셈은 나머지를 고려하지 않는다. 반면 메모리 상에서는 나머지 메모리 공간도 하나의 페이지를 차지하기 때문에 나머지가 있다면 count 수를 1 증가시켜 총 페이지 수에 자투리 메모리도 고려해준다.

```
int
countpp(void)
{
  int count = 0;
  struct proc *curproc = myproc();

  for (uint va = 0; va < curproc->sz; va += PGSIZE){
    pte_t *pte = walkpgdir(curproc->pgdir, (char *)va, 0);
    if (pte && (*pte & PTE_P))
      count++;
  }

  return count;
}
```

→ countpp()에서는 가상주소 0에서 시작해서 현재 프로세스의 유저 메모리 사이즈만큼까지 페이지 테이블을 탐색한다. 여기서 walkpgdir를 사용하여 page table entry를 얻고, 해당 pte가 유효한 물리 주소를 할당하고 있는지 확인한 후 count 값을 증가시켜준다. 이렇게 해주면 xv6에서는 demand paging을 사용하지 않아 유효한 페이지들은 모두 메모리에 올려두기 때문에 countvp와 동일한 결과를 얻게 될 것이다.

```
int countptp(void){
  struct proc *curproc = myproc();
  int count = 0;
  pde_t *pgdir = curproc->pgdir;

  count++; //count for the page directory entry page

  for(int i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P)
      count++;
  }
  return count;
}
```

→ countptp()에서는 현재 프로세스의 페이지 테이블에 의해 할당된 페이지 수를 반환한다. 여기서는 프로세스 내부 페이지 테이블 저장을 위한 모든 페이지와 페이지 디렉토리를 위한 페이지도 포함해야하므로 NPDENTRIES를 사용하여 for문을 작성해주었다. NPDENTRIE는 아래와 같이 memlayout.h에 정의되어 있고, freevm()에 쓰인 코드를 참고하여 작성하였다. 또한, 유저 레벨의 pte 뿐만 아니라 커널 페이지 테이블 매핑을 저장하는 페이지 테이블도 포함되어야 하므로 PTE_P로만 유효한지 확인하여 할당된 페이지 수를 세주었다. 또한, page directory entry를 저장하는 또 다른 페이지가 있기 때문에 이것에 대해 count 수를 추가로 1 증가시켜줬다.

```
// 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
```

함수들을 알맞는 파일에 작성 후 유저 프로그램에서도 시스템 콜로 부를 수 있도록 sysproc.c에 아래와 같이 각각 wrapper function을 작성해주었다.

```
//edited for project4
int
sys_countfp(void){
  return countfp();
}

int
sys_countvp(void){
  return countvp();
}

int
sys_countpp(void){
  return countpp();
}

int
sys_countptp(void){
  return countptp();
}
```

위에서 구현한 모든 함수들, 총 8개를 defs.h에 정의해주고, 위 4개를 시스템 콜로 등록하기 위해 Makefile, syscall.h, syscall.c, user.h, usys.S를 수정해주었다. 또한 Makefile에 테스트를 위한 파일들(test0, test1, test2, test3)을 등록해주었다.

III. Result

A. Copy-on-Write

Compile:

make → make fs.img → ./bootxv6.sh 순으로 명령어를 작성하여 컴파일하면 아래와 같이 실행되었다.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk...
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test0
```

모든 테스트 케이스 결과는 출력 예시와 똑같이 나왔다.

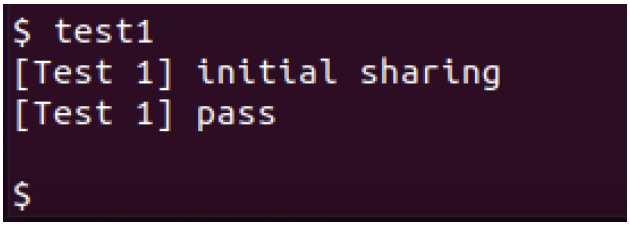
Test 1. test0.c

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

$
```

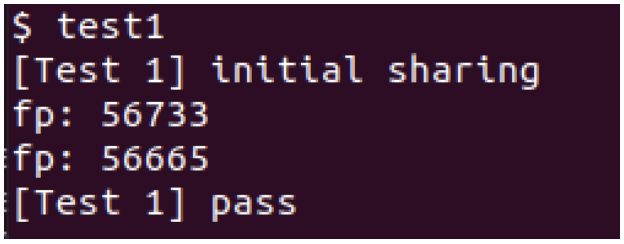
해당 테스트에서는 메모리 할당 전후의 free page 수, page table entry 수, 그리고 페이지 디렉토리에 사용된 페이지까지 포함된 수를 비교한다. countfp, countvp, countpp, countptp의 결과를 출력해본 결과, 아래 사진처럼 나온 것을 볼 수 있다. 이를 통해 demand paging 기법의 영향을 받지 않아 countvp와 countpp의 결과가 같다는 것을 볼 수 있고, 할당 전후의 countfp 결과, 즉 free page 수가 페이지 할당 후이기에 1만큼 차이나는 것을 볼 수 있다. 또한, ptp의 출력값이 테스트 명세에 나온 예시와 똑같이 66이 나온 것을 볼 수 있다. 그 결과, 테스트를 통과하였다.

```
$ test0
[Test 0] default
fp: 56733
vp: 3
pp: 3
ptp: 66
fp: 56732
vp: 4
pp: 4
ptp: 66
ptp: 66 66
[Test 0] pass

$
```

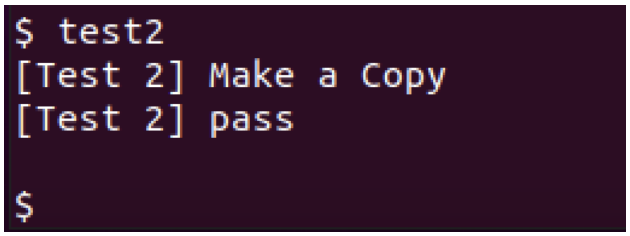
Test 2. test1.c



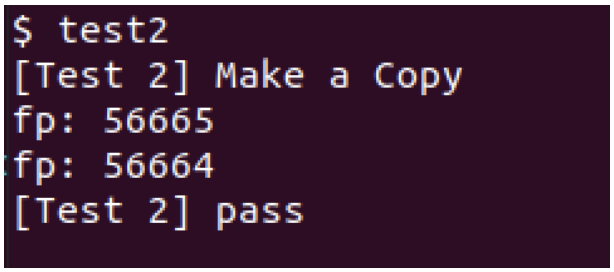
해당 테스트는 fork()를 통해 자식 프로세스를 생성하여 자식, 부모 모두 같은 물리 페이지를 가리키게 한다. fork에서 copyuvm을 통해 자식을 생성하며 페이지를 공유한다. 허나 공유를 하더라도 자식 프로세스는 자신의 페이지 디렉토리와 페이지 테이블이 있어야하므로 페이지 디렉토리, 커널 스택, 프로세스 control block과 같은 페이지를 필요로 한다. 그러므로 자식 프로세스 생성 후 countfp 결과의 차이가 68이 나야한다. 아래 free page 수를 보면 정확히 68 페이지가 줄어든 것을 것을 볼 수 있고, 결과적으로 테스트가 통과한 것을 볼 수 있다.



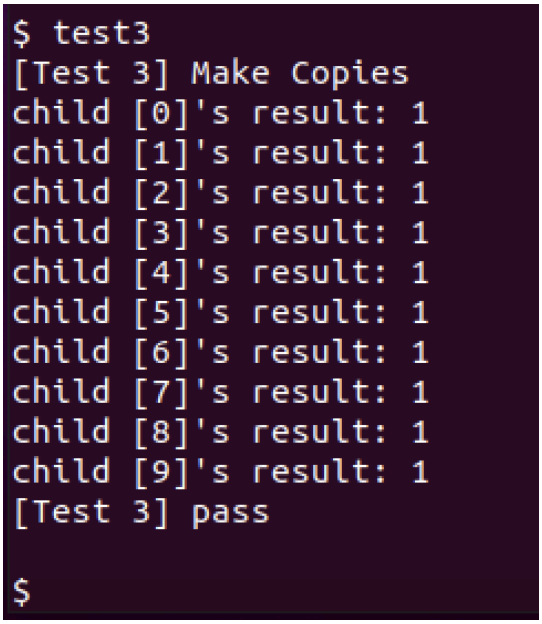
Test 3. test2.c



해당 테스트에서는 fork를 통해 자식 프로세스를 생성 후 자식 프로세스가 쓰기 접근을 하였을 때 새로운 물리 페이지를 할당 받는지 확인한다. initial_data라는 변수를 1로 수정하였을 때, 자식 프로세스는 공유 중이던 페이지를 복제 후 수정을 진행 한다. 그러므로 새로운 페이지를 하나 할당받아야하므로 수정 전후로 countfp의 결과 값이 1 차이 나야한다. 그 결과 아래와 같이 정확히 1개의 페이지가 새롭게 할당되어 1 차이 나는 것을 볼 수 있다. 따라서 해당 테스트도 통과했다.



Test 4. test3.c



fork로 자식 프로세스를 10개 생성 후 각각의 자식 프로세스가 부모 프로세스와 공유하는 변수에 대해 쓰기 접근을 한다. test2에서 확인하였듯이 공유변수에 대해 수정 시 알맞게 페이지가 할당되는 것을 볼 수 있다. 그렇다면 각 자식 프로세스가 종료할 때 해당 페이지에 대한 회수도 적절히 이루어져야한다. 아래 출력결과를 보면, 첫 fp는 부모 프로세스의 countfp 결과이고, 그 후는 각 프로세스에 대해 첫번째 fp는 수정 전후의 자식 프로세스의 countfp 결과이다. 예를 들어, 첫번째 자식 프로세스에 대해, 56053 → 56052로 바뀐 것으로 보아 적절해 페이지가 새롭게 할당되었다. 그 이후 exit을 통해 해당 프로세스가 종료되고, 다시 새로운 자식 프로세스가 생성되어 fp 출력값을 보면, 68+1=69만큼 free page 수가 늘어난 것을 볼 수 있다. 프로세스가 종료되며 free page에 대해 적절한 회수가 이루어진것이다. 그리고 이 과정이 반복되어 마지막 프로세스 종료까지 이루어지면, 처음 부모의 free page 수와 같은 것을 볼 수 있다. 이는 모든 프로세스에 대해 페이지 회수가 알맞게 진행됐다는 것이다. 이 반복 과정에서 하나의 프로세스가 완전히 종료된 후 새로운 프로세스가 생성되므로 시간차를 두고 자식 프로세스에 대한 결과가 하나씩 차례로 출력되었다. 이렇게 마지막 테스트도 통과하였다.

```
$ test3
[Test 3] Make Copies
fp: 56733
fp: 56053
fp: 56052
child [0]'s result: 1
fp: 56121
fp: 56120
child [1]'s result: 1
fp: 56189
fp: 56188
child [2]'s result: 1
fp: 56257
fp: 56256
child [3]'s result: 1
fp: 56325
fp: 56324
child [4]'s result: 1
fp: 56393
fp: 56392
child [5]'s result: 1
fp: 56461
fp: 56460
child [6]'s result: 1
fp: 56529
fp: 56528
child [7]'s result: 1
fp: 56597
fp: 56596
child [8]'s result: 1
fp: 56665
fp: 56664
child [9]'s result: 1
fp: 56733
[Test 3] pass
```

IV. Trouble Shooting

A. Booting again infinitely

모든 코드를 수정 및 구현 하고 처음 부팅을 하여 실행했을 때, 부팅이 연속적으로 계속 되는 현상이 있었다. 처음에는 당황하였지만 수정한 코드를 찬찬히 살펴보니 ref_count에 대해 아무 조건없이 접근한 실수를 발견하였다. 아래와 같이 ref_count의 수가 0보다 클 때만 감소할 수 있도록 조건문을 덧붙여주니 해결되었다. 부주의로 인해 생긴 에러였다.

```
if(get_refc(V2P(v)) > 0){ //edited
  decr_refc(V2P(v));
```

B. Editing fork() directly

과제 명세를 읽은 후 구현 구상을 할때, copyuvm을 통해 페이지 복제를 하니 단순히 copyuvm을 사용하는 기존 코드를 지우고 직접 fork를 수정하면 될 것이라 생각했다. 하지만 작업을 하다보니 아래처럼 walkpgdir와 같은 static 함수를 사용해야했고, 이 때문에 proc.c에 있는 fork를 직접 수정하는 것은 불가하다는 것을 깨달았다. 그리하여 기존 코드를 사용하되 vm.c에서 copyuvm을 수정하기로 결정하였다.

```
//  np->pgdir = curproc->pgdir; // share parent's page table with child
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

/*  for(i = 0; i < np->sz; i += PGSIZE){
    pte_t *pte = walkpgdir(np->pgdir, (void*)i, 0);
    if(pte && (*pte & PTE_P)){
      uint pa = PTE_ADDR(*pte);
      incr_refc(pa); // increment ref_count for shared pages
      *pte &= PTE_W; // set pages as read-only
    }
  }
*/
```

C. ref_count 변수 위치

처음 구상 단계에서는 kalloc.c에서 각 페이지마다 관리를 위해 run structure에 필드를 추가하고자 했다. 하지만 이를 실제로 구현 후 부팅하여 유저 프로그램을 실행하니 바로 종료가 되거나 무한대기를 하는 등의 문제가 있었다. decr_refc, incr_refc, 그리고 get_refc를 실행할 때마다 ref_count의 수를 출력하였을 때, 그 결과가 굉장히 컸고, 이는 로직에 맞지 않다고 생각하여 문제점을 발견할 수 있었다. 그리하여 전체적으로 페이지를 관리하기 위해 하나의 배열을 사용하기로 결정했고, 해당 크기는 전체 페이지 수로 만들어서 관리하게 했다. Locking 또한 적절히 사용하기 위해 kmem에 배열을 추가해주었다. 이렇게 수정하니 ref_count 출력값이 로직에 맞게 나와 해당 문제를 해결하였다.

D. 무한대기 **- Deadlock**

ref_count에 대한 문제는 해결하였지만, 무한대기 문제는 해결되지 않았다. 정확한 상황을 말하자면, 부팅 후 첫 테스트케이스는 바로 종료가 되었고, 그 후 두번째 테스트케이스를 실행하면 무한대기를 하는 현상이 있었다. 아무래도 deadlock이 걸린 듯 싶어 ref_count에 접근하는 코드들을 살펴보았다. 하지만 locking에 대한 acquire, release 문제도 잘 해결되어 있었고, 로직에는 문제가 없어 보였다. 그래서 혹시 몰라 ref_count를 정의한 kalloc.c에서는 incr_refc, decr_refc, 그리고 get_refc를 사용하지 않고 직접 ref_count에 접근하여 수정하도록 해주니 문제가 해결되었다. 아마 어떠한 상황에서 로직이 꼬이는 문제가 있었던 것으로 예상된다. 이렇게 모든 문제를 해결하여 위에서 본 것과 같이 모든 테스트를 통과할 수 있었다.