WIKI

이름: 문준영

학과: 컴퓨터소프트웨어학부

학번: 2022006135

Design:

Copy-on-Write:

- 메모리 낭비를 줄이기 위해 copy on write 기법을 사용합니다.
- Process가 다른 process를 fork할 때 새로운 물리 페이지를 할당 받지 않고 부모의 물리 페이지를 그대로 공유하게 합니다.
- fork 시 부모와 자식은 메모리 공유를 하지만 부모 프로세스에 반해 자식 프로세스는 write권한이 없습니다.

CoW_handler:

- write 권한이 없는 자식 프로세스가 부모 프로세스와 공유를 하는 페이지에 write를 시도를 하면 page fault가 발생합니다.
- xv6 운영체제 에서는 page fault에 대해 따로 핸들링 해주지 않기에 이를 처리해 주기 위해 Copy-on-Write handler를 page fault가 났을 때 수행합니다.
- Write를 시도하는 자식 프로세스에게 새로운 페이지를 할당해줍니다.
- 원래의 부모 프로세스가 가지고 있던 페이지를 그대로 새로운 페이지로 복사해 옵니다.
- 이를 자식의 page directory에 매핑시켜 줍니다.
- 자식 프로세스는 page fault가 handling된 후에 다시 page fault를 발생시켰던 instruction으로 돌아가 다시 명령을 수행하면 정상적으로 새로운 페이지에 write 명령이 수행이 됩니다.

Ref_count::

- 참조 횟수(reference count)를 하는 배열입니다.
- 배열의 크기는 전체 physical address의 페이지를 모두 포함할 수 있는 크기입니다.(PHYSTOP / PGSIZE)

- 처음 free page가 할당될 때 reference 값을 1로 만들어 줍니다.
- Page가 free될 때 그 페이지의 ref_count 값이 1이 될 때만 free해주도록 합니다

Incr_refc:

- 주어진 페이지의 ref_count값을 1 증가시킵니다.
- kmem.lock을 사용하여 incr_refc를 수행하는 동안 다른 프로세스들이 같은 페이지에 대해 접근하지 못하도록 막습니다.

Decr_refc:

- 주어진 페이지의 ref_count값을 1 감소시킵니다.
- Kmem.lock을 사용하여 decr_refc를 수행하는 동안 다른 프로세스들이 같은 페이지에 대해 접근하지 못하도록 막습니다.

Get_refc::

- 주어진 페이지의 ref_count값을 반환합니다.
- Kmem.lock을 사용하여 get_refc를 수행하는 동안 다른 프로세스들이 같은 페이지에 대해 접근하지 못하도록 막습니다.

Countfp:

- 시스템에 존재하는 free page의 총 개수를 반환합니다.
- Kmem.freelist를 돌면서 free page의 개수를 셉니다.

Countvp:

- Virtual address로 user memory에 할당된 가상 페이지들을 모두 순회하며 그 개수를 반환합니다.

Countpp:

- 페이지 디렉토리와 페이지 테이블을 순회하면서 프로세스에 할당된 물리 페이지를 모두 세어 그 수를 반환합니다.

Countptp:

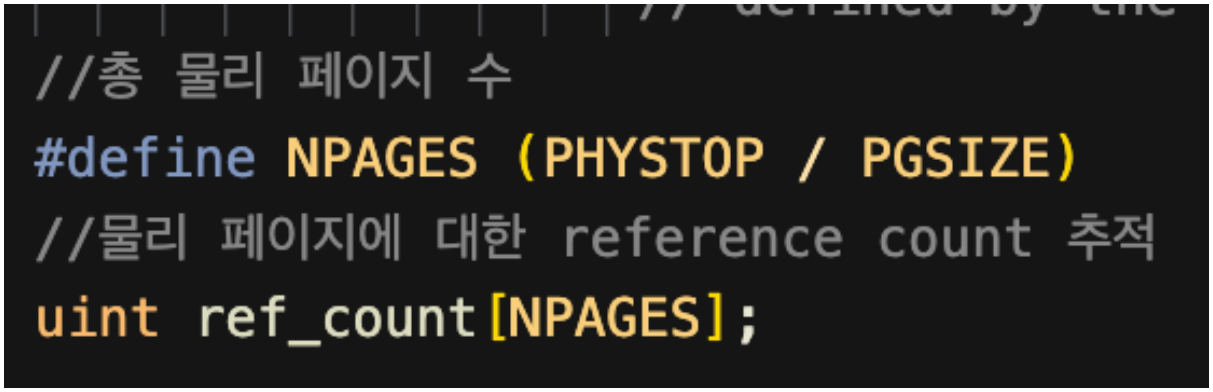
- 프로세스 내부에서 페이지 테이블에 의해 할당된 페이지의 개수를 반환합니다.
- xv6 운영체제 에서는 한 페이지 테이블의 크기가 PGSIZE와 동일하기에 페이지 디렉토리를 돌면서 유효한 page directory entry를 찾아 그 개수를 세어 반환합니

다.

Implementation:

kalloc.c:

ref_count[NPAGES]:



Ref_count 배열을 만들기 위해 총 페이지의 개수를 정의하였습니다. 총 피지컬 address는 PHYSTOP까지 위치하고 있고 그 이상의 주소는 다른 I/O device를 위해 남겨두기에 PHYSTOP까지만을 physical page를 세는 데에 활용했씁니다. 전체 PHYSTOP을 PGSIZE로 나누어 페이지의 개수를 정의하였습니다. Ref_count배열을 총 physical page의 개수의 크기로 만들었씁니다.

kalloc.c(void):

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

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

Kalloc함수에서 처음에 freelist에서 페이지를 할당할 때 참조 횟수를 1로 설정해줍니다.

kfree(char *v):

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

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

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

  pa = V2P(v);
  if(ref_count[pa / PGSIZE] > 1){
    ref_count[pa / PGSIZE]--;
    if(kmem.use_lock){
      release(&kmem.lock);
    }
    return;
  }
  else{
    ref_count[pa / PGSIZE] = 0;
  }
  memset(v, 1, PGSIZE);
  r = (struct run *)v;
  r->next = kmem.freelist;
  kmem.freelist = r;

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

Kfree함수에서 memory를 초기화하기 위해서는 참조 횟수가 0이어야 함을 추가합니다.

incr_refc(uint pa):

```
void
incr_refc(uint pa){
  uint v = P2V(pa);
  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP){
    cprintf("incr_refc: out of range\n");
  }
  if(kmem.use_lock){
    acquire(&kmem.lock);
  }
  ref_count[pa / PGSIZE]++;
  if(kmem.use_lock){
    release(&kmem.lock);
  }
}
```

physical address를 받아서 그 주소의 참조 횟수를 증가시키도록 합니다. pa/PGSIZE를 통해 인자로 들어온 Physical address의 physical page frame 번호가 무엇인지 구합니다. 또한, kmem.lock을 이용하여 incr_refc를 수행하는 동안 다른 프로세스가 동일한 page의 참조 횟수

를 접근하지 못하도록 막습니다. Incr_refc에 들어온 physical address가 virtual address로 변환하였을 때 범위를 벗어나면 에러를 출력합니다.

decr_refc(uint pa):

```
void
decr_refc(uint pa){
  uint v = P2V(pa);
  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP){
    cprintf("decr_refc: out of range\n");
  }
  if(kmem.use_lock){
    acquire(&kmem.lock);
  }
  ref_count[pa / PGSIZE]--;
  if(kmem.use_lock){
    release(&kmem.lock);
  }
}
```

Incr_refc함수와 거의 동일하지만 reference count를 감소시킨다는 점만 차이가 있습니다.

get_refc(uint pa):

```
int
get_refc(uint pa){
  uint v = P2V(pa);
  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP){
    cprintf("get_refc: out of range\n");
  }
  int temp;
  if(kmem.use_lock){
    acquire(&kmem.lock);
  }
  temp = ref_count[pa/PGSIZE];
  if(kmem.use_lock){
    release(&kmem.lock);
  }
  return temp;
}
```

incr_refc함수나 decr_refc함수와 거의 동일하지만 리턴하는 값이 주어진 physical address가 속한 페이지의 reference count값입니다.

countfp(void):

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

free page의 개수를 셉니다. Kmem lock을 획득하고 freelist전체를 순회하면서 free list의 개수를 count합니다. 그리고 count를 리턴합니다.

여기서 고민한 것이 있습니다. countvp, countpp, countptp함수는 모두 myproc()을 사용해 현재 프로세스에 접근해야 합니다. 하지만 kalloc.c파일에서는 myproc()에 대한 접근이 안됩니다. 또한, countfp를 구현하기 위해서는 freelist를 접근해야 하기에 kalloc.c에서 countfp를 구현했씁니다. 하지만 나머지 세 개의 count함수는 모두 vm.c파일에서 구현했습니다.

vm.c:

countvp(void):

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

  for(va = 0; va < sz; va += PGSIZE){
    pte_t *pte = walkpgdir(pgdir, (void *)va, 0);
    if(pte && (*pte & PTE_P)){
      count++;
    }
  }
  return count;
}
```

현재 프로세스의 할당된 가상 페이지의 수를 반환하는 함수입니다. 먼저 myproc()을 통해 현재 프로세스에 대한 접근을 합니다. 그리고 현재 프로세스의 sz를 PGROUNDUP을 통해 마지막 페이지 전체를 count할 수 있도록 합니다. (왜냐하면 va변수로 전체 가상 주소를 순회할 것인데 이를 PGSIZE단위로 증가시킬 것이기 때문입니다.) va를 PGSIZE단위로 증가시키면서 walkpgdir함수를 통해 해당 가상 주소를 통해 접근한 page table entry가 유효한지 확인합니다. 유효하다면 해당 가상 페이지는 할당된 것이므로 count값을 하나 증가시켜 줍니다. 이를 모든 가상 페이지에 대해 수행하고 count값을 반환합니다.

countpp(void):

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

  // page directory entry 순회
  for(int i = 0; i < PDX(KERNBASE); i++){
    if(pgdir[i] & PTE_P){
      pte_t *pgtab = (pte_t *)P2V(PTE_ADDR(pgdir[i]));
      // page table entry 순회
      for(int j = 0; j < NPTENTRIES; j++){
        if(pgtab[j] & PTE_P){
          count++;
        }
      }
    }
  }
  return count;
}
```

현재 프로세스의 페이지 테이블을 순회하면서 유효한 page table entry의 수를 반환하는 함수입니다. xv6 운영체제에서는 Demand paging 기법을 사용하지 않기에 countvp함수의 결과값과 countpp의 결과값이 서로 같아야 한다. 이 말인 즉슨, countvp에서 커널 영역을 고려하지 않고 할당된 가상 페이지의 개수를 셌으므로, countpp에서도 할당된 page table entry 중 kernel 영역을 제외한 entry를 셀 것이다.

먼저, 현재 프로세스의 페이지 디렉토리에 접근하여 그 페이지 디렉토리를 순회합니다. 순회는 NPDENTRIES만큼 순회하는 것이 아닌 PDX(KENBASE)까지 순회함으로써 커널 영역까지 접근을 하지 않도록 합니다.

해당 page directory entry가 유효하다면 이를 통해 page table의 주소를 알아내 page table을 순회합니다. Page table을 순회하면서 page table entry가 유효하면 count를 증가시킵니다. 최종적인 count값을 리턴합니다.

countptp(void):

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

  // 페이지 디렉토리 자체의 페이지를 포함
  count++;

  for (int i = 0; i < NPDENTRIES; i++) {
    if (pgdir[i] & PTE_P) {
      // 페이지 디렉토리 엔트리가 존재하면 페이지 디렉토리 엔트리 카운트
      count++;
    }
  }

  return count;
}
```

현재 프로세스의 페이지 테이블에 의해 할당된 페이지의 수를 리턴하는 함수입니다. 즉 페이지 테이블을 위해 얼마나 많은 페이지가 활용되었는 지를 알아내는 함수입니다. xv6 운영체제는 32bit machine으로 2 level hierarchy의 page table로 이루어져 있습니다. 때문에 하나의 페이지 테이블이 차지하는 페이지는 한 페이지입니다. 따라서 page directory를 위한 페이지 수 한 개와 page directory를 돌면서 유효한 Page directory entry, 즉, page table의 개수를 세면 페이지 테이블에 의해 할당된 페이지의 수가 구해집니다.

Copy on Write:

Copy on write를 구현하기 위해 먼저 fork시에 자식 프로세스가 부모 프로세스의 메모리 공간을 복사 하지 않고 공유를 해야 합니다. Proc.c 파일에서 부모 프로세스의 메모리를 복사해오는 부분은 copyuvm함수입니다.

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return
  }

  // Copy                                    roc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
```

```
pde_t *proc::pgdir
Page table
pgdir
```

때문에 copyuvm에서 copy가 아닌 share를 하도록 수정해주었습니다.

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);
    flags = PTE_FLAGS(*pte);
    //페이지를 복사가 아닌 공유
    //읽기전용으로 만들어버림
    flags &= ~PTE_W;

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

인자로 들어오는 pgdir은 부모의 pgdir입니다.

-자식을 위한 새로운 페이지 디렉토리는 할당해줘야 하므로 setupkvm함수를 이용해서 자식 프로세스를 위한 새로운 페이지 디렉토리를 할당해줍니다.

-부모의 페이지 디렉토리를 돌면서 page table entry를 구해 부모 프로세스의 물리 페이지를 구합니다.

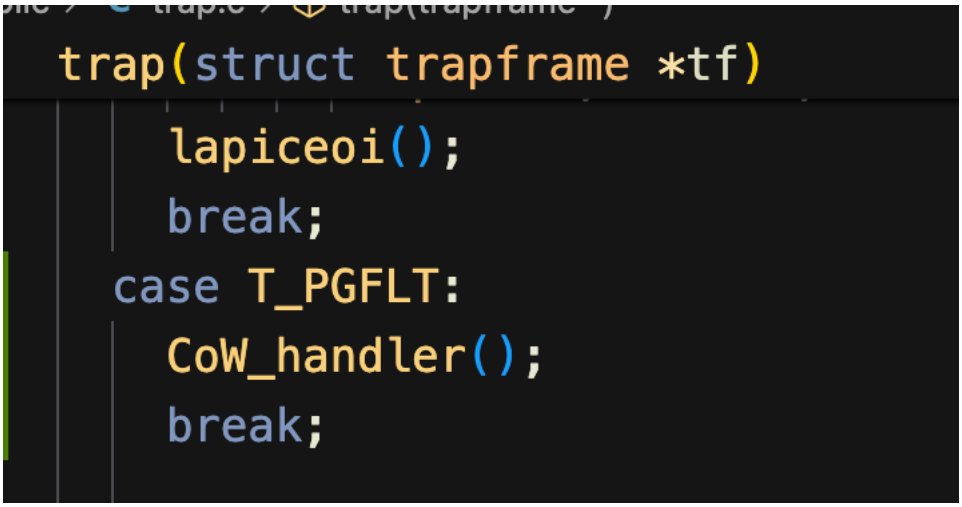
-구한 부모 프로세스의 물리 페이지의 flag에 write권한을 제외하고 이를 새로운 자식 프로세스의 페이지 디렉토리에 mappages함수를 사용해 매핑 시켜 줍니다. 매핑 시켜줄 때 해당 물리 페이지에 대한 참조 횟수가 증가하므로 incr_refc를 호출하여 해당 물리 페이지의 참조 횟수를 1 증가시킵니다.

-마지막에 페이지 디렉토리 d에 대해 TLB flush를 수행해줍니다. (page table entry에 대한 값이 변했기 때문)

이 과정을 통해 부모의 물리 페이지를 복사하지 않고 공유할 수 있습니다.

만약, 자식 프로세스가 여기서 부모와 공유하고 있는 물리 페이지에 write를 수행하려고 한다면, 즉, exec같은 함수를 수행하려고 한다면 그 때 write하려고 하는 페이지를 복사하고 복사된 페이지에 write권한을 부여해 그곳에 write하도록 해야 합니다. 이를 구현하기 위해 page fault에 대해 handling을 해야합니다. 왜냐하면, write 권한이 없는 페이지에 대해 write하려고 시도하면 Page fault가 발생하기 때문입니다.

Trap.c 파일을 보면 page fault에 대한 핸들링 작업이 없습니다. 때문에 page fault가 발생하면 default로 넘어가 panic이 발생합니다. 때문에 다음과 같이 trap 함수에 추가해 page fault 발생시 CoW_handler가 호출되게 합니다.



CoW_handler(void):

```
void
CoW_handler(void){
  uint pgflt_addr = rcr2();
  pde_t *pgdir = myproc()->pgdir;
  pte_t *pte;
  uint currpage = PGROUNDDOWN(pgflt_addr);
  char *mem;
  //완전히 새로운 페이지를 할당해야 하므로 round up
  pte = walkpgdir(pgdir, (void *)currpage, 0);

  //page fault가 발생한 주소가 page table에 매핑되어 있지 않은 잘못된 범위에 속해 있는 경우 panic 발생
  if(!pte || !(*pte & PTE_P)){
    panic("Page fault");
  }

  //Write 권한이 없는 경우
  if(!(*pte & PTE_W)){
    uint old_pa = PTE_ADDR(*pte);
    uint flags = PTE_FLAGS(*pte);


    if((mem = kalloc()) == 0){
      panic("kalloc error");
    }
    memmove(mem, (char *)P2V(PTE_ADDR(*pte)), PGSIZE);


    //reference 값 하나 낮춤
    decr_refc(PTE_ADDR(old_pa));
    *pte = V2P(mem) | flags | PTE_W;
    lcr3(V2P(pgdir));
    return;
  }
}
```

Copy on Write 기법에서 page fault가 발생했을 때 이를 handling하는 함수입니다.

-rcr2()함수를 이용해서 page fault가 일어난 주소를 구합니다.

-우리는 page fault가 난 주소보다는 그 페이지에 더 중점을 두고 있기에 PGROUNDDOWN 매크로를 이용해 page fault address의 page를 구해줍니다.

-해당 페이지는 현재 가상 주소로 되어 있으므로 walkpgdir을 이용해서 해당 페이지에 매핑 되어 있는 물리 페이지를 가리키는 page table entry를 구해줍니다.

-만약 page table entry가 유효하지 않다면 접근하면 안되는 page에 접근한 것이므로 panic을 발생시킵니다.

-만약 해당 물리 페이지에 대한 write권한이 없어서 Page fault가 났으면 다음을 수행합니다.

1. kalloc을 이용해 새로운 물리 페이지를 할당받습니다.

2. 새롭게 할당받은 물리 페이지에 page fault가 났던 페이지를 복사해 옵니다

3. 새롭게 할당받은 물리 페이지에 write 권한을 줍니다.

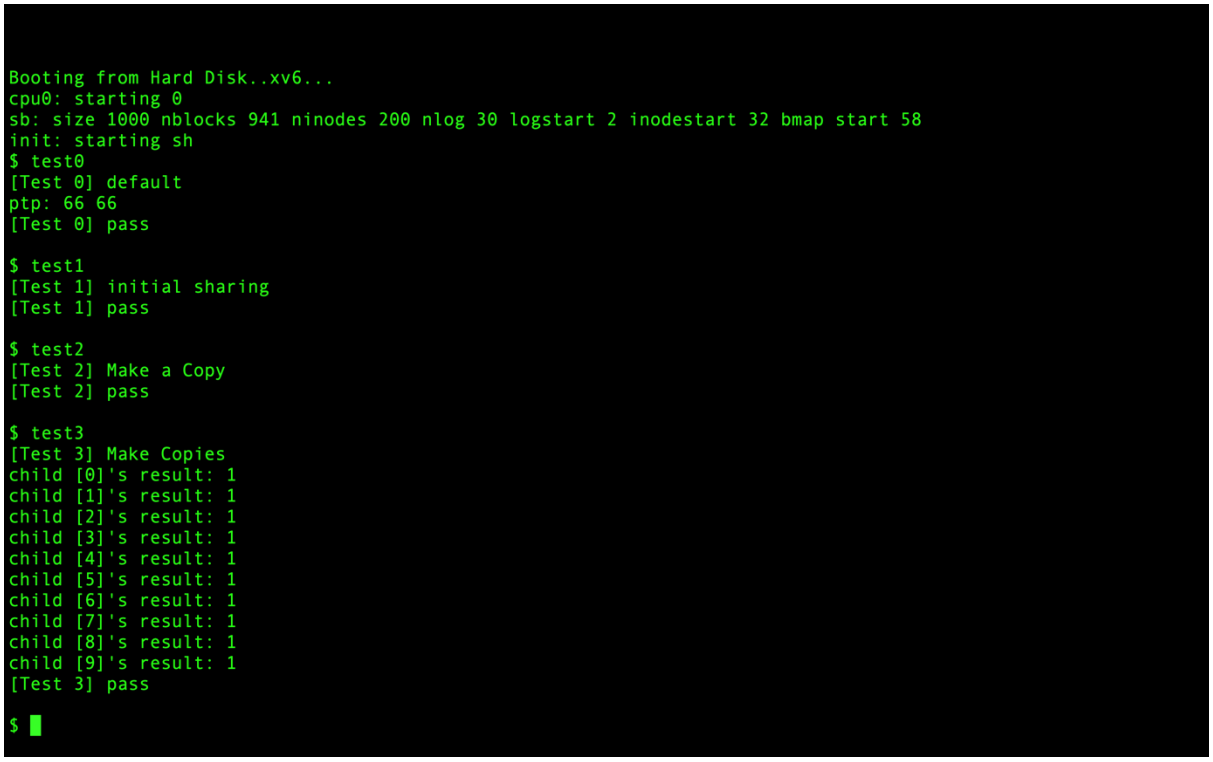
4. page table entry를 새롭게 할당받은 물리 페이지로 변경합니다.

-최종적으로 page table entry를 수정해줬으므로 TLB flush를 해주고 함수를 종료합니다.

이와 같이 CoW_handler를 수행하고 나면 다시 원래의 page fault가 발생했던 instruction부터 수행합니다. 때문에 write권한이 주어졌기에 정상적으로 프로세스가 진도를 나갈 것입니다.

Result:

Test0~Test3 파일의 수행 결과입니다.



Test0에서 sbrk에 대해 page table entry의 수가 잘 변동한다는 것을 알 수 있고 countptp 역시 정상적으로 작동한다는 것을 알 수 있씁니다.

Test1에서 fork 시 initial sharing을 함을 알 수 있습니다.

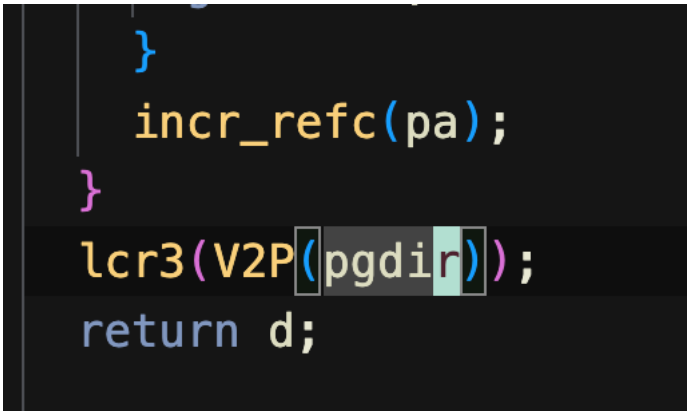
Test2와 Test3에서 copy가 잘 만들어지는 것으로 보입니다.

Trouble Shooting:

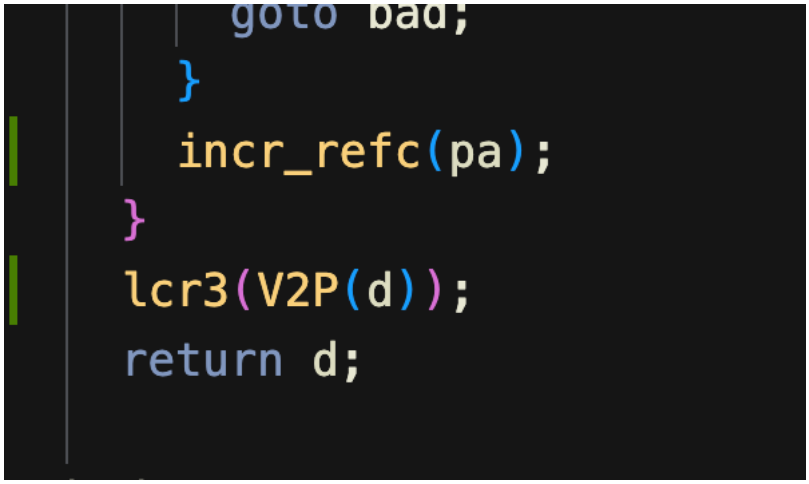
이번 과제를 하면서 3가지 정도의 문제를 겪었습니다.

가장 첫째로 겪은 문제는 xv6 운영체제가 부팅이 안되는 에러였습니다. 처음에는 이 문제가 kmem.lock이 deadlock이 걸린 줄 알고 이를 열심히 수정했습니다. 하지만 그럼에도 부팅이 되지 않기에 찾아보니 xv6 부팅시에 cprintf와 같은 함수를 사용하면 부팅이 안된다고 하더군요. 때문에 kfree, walkpgdir 등등 초기 부팅시 사용되는 함수에서 cprintf와 같은 함수를 지우니 부팅이 되었습니다.

둘째로 make a copy가 안됐던 상황입니다. CoW_handler의 구현은 위와 같이 완벽하게 구현하였고, 위와 같이 부모 프로세스와 페이지 공유 상태에서 페이지 복사 상태로 변강하였습니다. 하지만 xv6를 부팅해서 ls와 같은 간단한 명령어를 입력한 결과 아무런 결과가 출력되지 않았습니다. Cprintf()를 사용하면서 디버깅을 해본 결과 init process와 shell process의 경우에는 exec가 호출이 되지만 제가 실행하는 프로세스, 즉, 세 번째 프로세스부터는 page fault가 올바르게 handling된 후에 exec가 다시 실행되지 않는 것이었습니다. 이 문제점을 찾아보니 문제는 copyuvm에서 발생했었습니다. Copyuvm함수에서 저는 page table entry를 수정하였기에 TLB flush를 수행해줬습니다. 하지만 여기서 저는 부모



프로세스의 page directory에 대해 TLB flush를 진행한 거였습니다. 사실 copyuvm 수행 중 부모 프로세스의 page table entry는 변한게 없습니다. 제가 구현한 바에 따르면 단순히 부모 프로세스의 페이지를 자식 프로세스에 복사를 해주는 과정만 있을 뿐이죠. 때문에 TLB flush는 자식 프로세스의 page table entry가 변했으므로 자식 process의 page directory에 대해 진행해줘야 합니다.



이와 같이 자식 프로세스에 대해 TLB flush를 하니 정상적으로 프로세스가 exec를 수행했습니다.

마지막으로 겪은 문제는 countptp의 명세 이해를 잘 못했기에 발생했습니다. 명세를 잘못 이해해서 그냥 프로세스의 해당하는 모든 페이지를 다 count하였습니다. 하지만 결과가 너무나도 예상 테스트의 결과가 다르기에 명세를 좀 더 자세히 읽어보았습니다. 명세를 읽어보니 "페이지 테이블에 의해 할당된 페이지"라고 했습니다. 이 말이 좀 이해가 안 가서 곰곰이 생각해보니 page table 역시 저장하기 위해 공간이 필요한데 그 공간을 말하는 것임을 깨달았습니다. 이렇게 마지막 문제 역시 해결하였습니다.