4️⃣ Project04

학번 : 2018008068

이름 : 김재형

1️⃣Design

과제 명세

1. Copy on Write 구현
    a. 기존 copyuvm 수정
    b. Copy on Write Handler 구현
2. page 함수 count 구현
    a. free page 갯수 반환 함수
    b. user memory 에 할당된 가상 페이지 개수 구현
    c. 유효한 물리주소가 할당된 page table entry 수 반환
    d. 프로세스 페이지 테이블에 의해 할당된 페이지 수 반환(kernel + user memory)

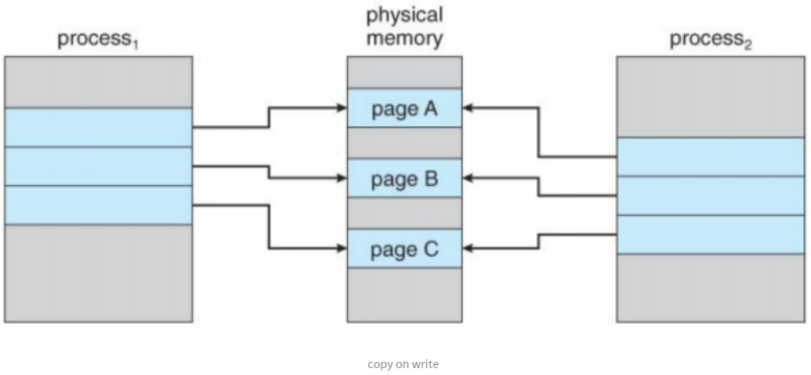
기존 xv6 분석

우선 기존 xv6는 Copy on Write 가 아닌 어떠한 방식으로 작동되는지부터 알아보자.

일반적으로 fork 이후 exec 을 호출하여 새 프로그램을 실행한다.

하지만, 이는 메모리 낭비라고 볼수있다.

복사를 하고 exec을 통해서 새로운 process를 만들어내는 것인데 write을 하지않고 read 만을 하는것이면 , 굳이 복사를 할 필요가 있는 것인가? 에 대한 의문이다.



copy on write

즉, 위 그림과같이 parent process를 child 에 복사하지 않고 parent process가 가지고 있는 phyiscal memory 의 주소를 공유하는 방식을 통해 child process를 사용하는 것을 말한다.

child process는 생성될시, parent process의 page들을 read-only 상태로 바꿔줘야한다.

(read 는 여러번 가능하다만, write는 한번에 하나만 가능하기에) … 만일 data가 실제 공유된 page에 write가 된다고 할시 page fault 가 일어난다!

즉 이를 통해 하나의 page 를 여러개의 process 가 사용하게 되므로 reference count 가 필요하다.

그렇다면 기존 fork함수에서는 어떻게 copy를 할까?

parent process 의 내용을 child process에게 복사하는 작업 copyuvm(curproc→pgdir, curproc→sz) 을 통해서 이루어진다.

Design 구현

- copyuvm 함수를 수정하여 , 기존에 physical memory 를 복사하는 부분을 수정할 필요가 있다.
  - 메모리 복사하는 부분 삭제
- CoW handler 함수를 구현하여 page fault 가 일어났을시, 처리하는 함수를 구현합니다.
  - 페이지 참조 횟수 구현
  - 참조횟수가 0일시 free 구현
  - TLB Flush 하는부분 삽입(페이지 권한이 변경되었기에 페이지 테이블을 재설치하고 CR3 레지스터에 페이지 포인터를 다시 선언하여야 한다.)
- Kalloc 함수를 수정해서 Reference count를 수정하여야한다.

2️⃣Implementation

copyuvm 함수 수정

1. copyuvm 함수 주석 처리
   a. 기존의 if(mem= kalloc()) 이라는 physical page를 할당하는 함수
   b. memmove(mem, (char*)P2V(pd), PSIZE) 라는 pa를 가상주소로 바꾼이후 mem에 복사하는 작업을 하는 부분

   위 a,b 부분 주석처리한다.
2. page table entry(PTE)에 writeable flag 를 disable 로처리
3. 기존 parent process의 page와 child process를 서로같은 mapping table 보도록 한다.
4. physical page의 reference count 증가
5. Lcr3(V2P(pgdir)) 호출하여 , TLB Flush 수행

코드는 다음과 같다.

```
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");

    *pte &= (~PTE_W);
    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(pgdir));
  return d;

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

CoW handler 구현

- trap.c 의 T_PGFLT 에 추가한다.

1. rc 를 호출하여 우선적으로 pagefault 가 발생한 가상주소를 가져온다.
   - walkpgdir는 이때 pgdir 과 가상주소를 줄시 해당 주소가 속한 pte를 반환한다.
2. pte를 통해서 physical address를 찾고, reference counter를 가져온다.
   a. refernce count 가 1보다 클시
      i. 새로운 페이지 할당받아서 복사
   b. referencecount 가 1일시
      i. 현재 pte 의 writeable flag 만 enable 시킨다.

코드는 다음과 같다.

```
void CoW_handler(void){
  uint pagefault_va;
  pte_t *pte;
  uint rc, pa;
  if((pagefault_va = rcr2())<0){
    panic("wrong access");
    return;
  }
```

```
  // va가 속한 pte 가져옴.
  pte = walkpgdir(myproc()->pgdir, (void*)pagefault_va, 0);
  // pte -> pa
  pa = PTE_ADDR(*pte);
  rc = get_refc(pa);

  if(rc >1){
    char *mem;
    if((mem = kalloc()) == 0) return;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    //
    *pte = V2P(mem) | PTE_P | PTE_U | PTE_W;
    decr_refc(pa);
  }
  else if(rc == 1){
    // 현재 page table entry 에 writeable flag 만 enable
    *pte |= PTE_W;
  }
  lcr3(V2P(myproc()->pgdir));
}
```

Kalloc 함수 수정

1. 새로운 page 를 할당하고 해당 page 를 freelist 에서 뺴는 동작을 수행을 할시

array 로부터 해당 reference count 를 1을 감소시키는 역할과 , 1로 초기화시키는 코드 역시 추가하여야 한다. 코드는 다음과같다.

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

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

  numfreepages--;
  r = kmem.freelist;
  if(r){
    kmem.freelist = r->next;
    pgrefcount[V2P((char*)r) >> PGSHIFT] = 1;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

countfp, incr_refc, decr_refc, get_refc 구현

```
uint pgrefcount[PHYSTOP >> PGSHIFT];
```

다음과 같이 pgrefcount 선언이후

```
int countfp(void){
  return numfreepages;
}

void incr_refc(uint pa){
  ++pgrefcount[pa >> PGSHIFT];
}

void decr_refc(uint pa){
  --pgrefcount[pa >> PGSHIFT];
}

int get_refc(uint pa){
  return pgrefcount[pa >> PGSHIFT];
}
```

다음과 같이 구현하였다.

countvp 구현

- 가상공간의 시작주소 0부터 Kernel base 까지 순회하며 유효한 page 일시 count 를 +1 한다. 코드는 다음과같다.

```
int countvp(void) {
    struct proc *curproc = myproc();  // 현재 프로세스 구조체 가져오기
    pde_t *pgdir = curproc->pgdir;  // 페이지 디렉토리 포인터
    int count = 0;

    // 사용자 공간의 시작 주소는 0이고, 페이지 크기는 4096(0x1000)입니다.
    // KERNBASE 직전까지 모든 페이지를 순회합니다.
    for (uint va = 0; va < KERNBASE; va += PGSIZE) {
        pde_t *pde = &pgdir[PDX(va)];
        if (*pde & PTE_P) {
            pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
            pte_t *pte = &pgtab[PTX(va)];
            if (*pte & PTE_P) {
                count++;  // 유효한 페이지만 카운트
            }
        }
    }
    return count;
}
```

countpp 구현

- 현재 process의 pgdir pointer 를 반환받은 이후 , page table entry 에서 각 page 를 순회하며 유효한 phyisical page를 확인한다. 코드는 다음과같다.

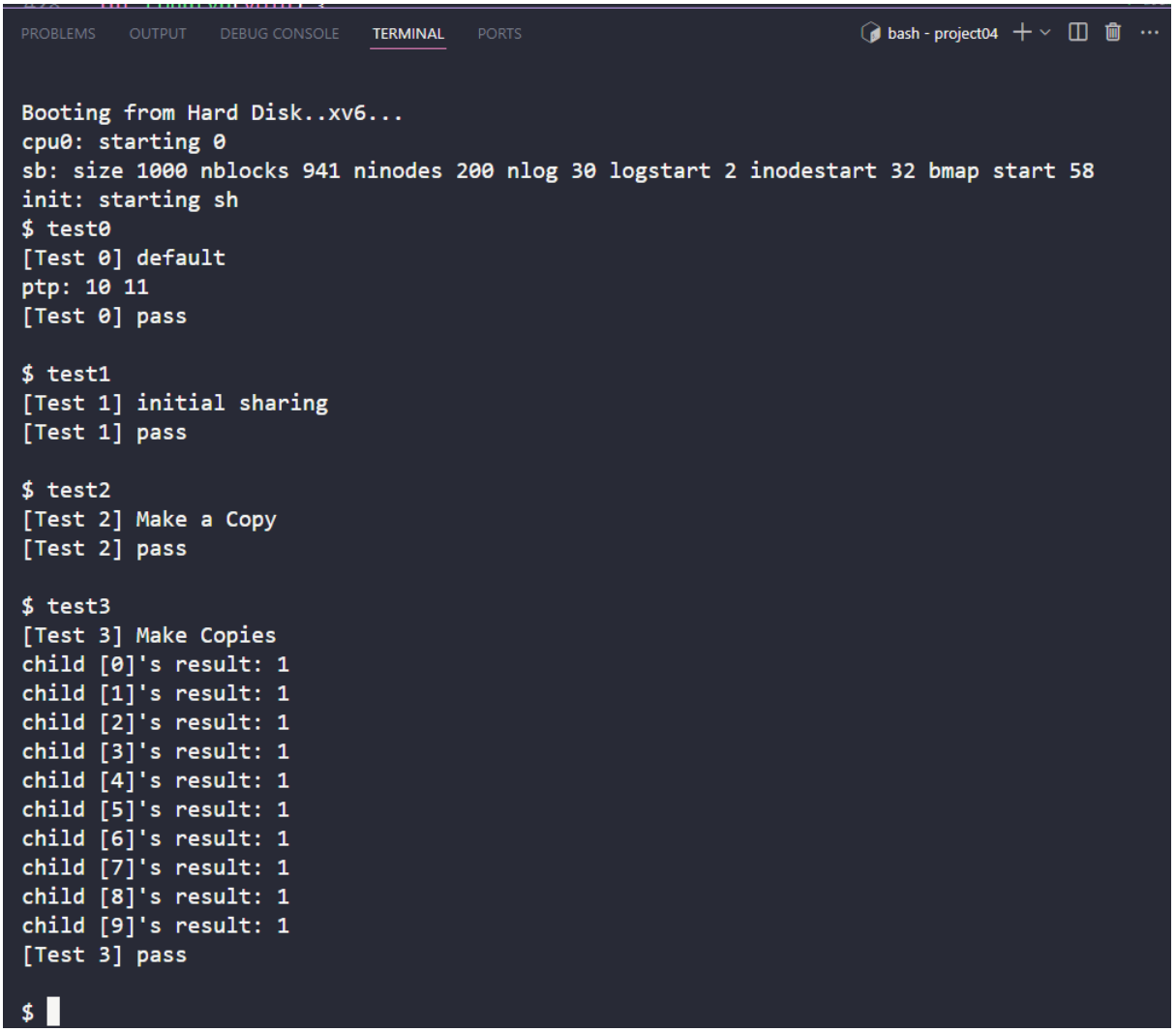
```
int countpp(void) {
    struct proc *curproc = myproc();
    pde_t *pgdir = curproc->pgdir;
    int count = 0;

    // KERNBASE의 페이지 디렉토리 인덱스를 계산
    uint kernbase_pdx = PDX(KERNBASE);

    for (int i = 0; i < kernbase_pdx; i++) {  // KERNBASE 이전까지만 검사
        if (pgdir[i] & PTE_P) {
            pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pgdir[i]));  // 해당 엔트리의 페이지 터
            for (int j = 0; j < NPTENTRIES; j++) {
                if (pgtab[j] & PTE_P) {
                    count++;
                }
            }
        }
    }
    return count;
}
```

3️⃣Result

결과는 다음과 같다.



4️⃣Trouble Shooting

Countptp 구현 이슈....

countptp 함수의 구현 명세는 다음과 같다.

1. 프로세스 페이지 테이블 (ptable) 을 순회하면서 할당된 페이지 갯수를 확인한다.
    - 각 프로세스의 pgdir을 순회하면서 유효한 page 의 갯수를 count 한다.
2. Kernel 페이지 테이블을 순회하면서 유효한 페이지 갯수를 확인한다.
    - 유효한 페이지 갯수를 count 한다.

일단 결과값은 다음과같다.

```
[Test 0] default
ptp: 10 11
[Test 0] pass
```

옳은 결과값은 66 이 나와야한다.

에러 원인

- User page table 의 갯수는 잘 세었지만, kernel page 테이블의 갯수확인부분에서 메모리를 조금 부정확하게 센것으로 확인된다.

구현코드는 다음과같았다.

```
int countptp(void) {
    struct proc *p;
    int count = 0;
    pde_t *pgdir;
    pte_t *pgtab;


    uint i;

    // 커널 페이지 테이블 순회
    pde_t *kpgdir = (pde_t*)KERNBASE;  // KERNBASE에 매핑된 페이지 디렉토리 포인터
    for (i = PDX(KERNBASE); i < PDX(0xFE000000); i++) {
        if (kpgdir[i] & PTE_P) {
            pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(kpgdir[i]));
            for (uint j = 0; j < NPTENTRIES; j++) {
                if (pgtab[j] & PTE_P) {
                    count++;  // 할당된 커널 페이지 카운트
                }
            }
        }
    }

    acquire(&ptable.lock);  // 프로세스 테이블 락을 획득합니다.
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state == UNUSED) continue;  // 사용되지 않는 프로세스는 건너뜁니다.

        pgdir = p->pgdir;  // 프로세스의 페이지 디렉토리 주소를 가져옵니다.
        if (!pgdir) continue;

        // 사용자 공간의 시작 주소는 0이고, KERNBASE 직전까지 모든 페이지를 순회합니다.
        for (uint va = 0; va < KERNBASE; va += PGSIZE) {
            pde_t *pde = &pgdir[PDX(va)];
            if (*pde & PTE_P) {
                pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
                pte_t *pte = &pgtab[PTX(va)];
                if (*pte & PTE_P) {
                    count++;  // 유효한 페이지만 카운트
                }
            }
        }
    }
```

```
    release(&ptable.lock);  // 프로세스 테이블 락을 해제합니다.

    return count;  // 할당된 페이지의 총 수를 반환합니다.
}
```