Project 4 Wiki

Design

Cow (Initial Sharing)

- **페이지 참조횟수 배열**
  - 각 물리페이지마다 참조횟수를 확인해야하므로 전체할당가능한 메모리크기/페이지의 크기를 가진 정수형 배열을 생성하여 참조된 페이지의 횟수를 확인할 것이다.
- **pde_t* shareuvm(pde_t* pgdir, uint sz) 구현**
  - fork 함수 내부에서 변경해야할 부분은 copyuvm뿐이므로 copyuvm의 구조와 비슷하게 구현하지만 페이지 맵핑과정을 같은 물리메모리를 가리키도록 구현하고 권한을 변경하는 식으로 구현할 것이다.
- **공유 페이지 참조횟수 페이지 증가 제어**
  - shareuvm에서 각 물리메모리를 공유하기 전에 참조횟수를 증가시키는 방향으로 처리할 것이다

Cow (Make a Copy)

- **trap 처리**
  - T_PGFLT exception는 여러 가지 방식으로 발생할 수 있으므로 trapframe의 에러를 분석하여 write로 인한 trap만 처리할 수 있도록 할 것이다.
- **void Cow_handler(void) 구현**
  - 공유된 페이지가 1개인 경우
    - 권한 변경후 TLB flush
  - 공유된 페이지가 2개 이상인 경우
    - 권한 변경후 copyuvm의 방식과 비슷하게 새로운 메모리를 할당 받아서 페이지를 맵핑하는 방식으로 구현할 것이다.
- **공유 페이지 참조횟수 감소 제어**
  - 프로세스가 페이지를 더 이상 참조를 하고 있지 않다는 것은 kfree함수를 호출했다는 것을 의미하므로 kfree 함수 내부에서 페이지의 물리메모리의 참조횟수의 감소처리 및 해제를 수행할 것이다.

Common Function

- **void incr_refc(uint) , void decr_refc(uint), int get_refc(uint) 구현**
  - kalloc.c의 kmem의 lock을 공유하여 사용하고 페이지 참조횟수 배열을 활용하여 구현할 것이다.

System Call

- **int countfp(void)**
  - kmem의 freelist를 선형탐색하여 구할 것이다.
- **int countvp(void)**
  - 0~ proc→sz 까지의 주소 범위에 해당하는 가상 페이지에 해당하는 page table entry가 존재하며 user 권한 페이지인 경우를 개수를 구할 것이다.
- **int countpp(void)**
  - page directory → page table → page entry 구조로 탐색하여 해당 page table entry가 존재하면서 user 권한 페이지인 경우의 개수를 구할 것이다.
- **int countptp(void)**
  - 페이지 테이블에 의해 할당된 페이지의 수를 반환해야하므로 현재 프로세스의 page directory → page table의 페이지 개수 + 1(page directory page)로 구할 것이다.

Implement

Structure

- 페이지 참조횟수를 저장하기 위해서 모든 최대 물리메모리/페이지 크기에 해당하는 배열을 생성합니다.

```
//kalloc.c
struct {
    ...
    int refc[PHYSTOP/PGSIZE];
} kmem;
```

Cow (Initial Sharing)

- copyuvm대신 shareuvm을 호출하여 물리메모리를 공유합니다.

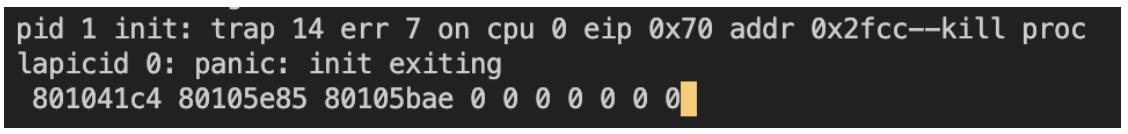
```
//proc.c
int
fork(void)
{
...
  if((np->pgdir = shareuvm(curproc->pgdir, curproc->sz)) == 0){
...
}
```

- **pde_t* shareuvm(pde_t* pgdir, uint sz)**
  - 기존 부모 페이지의 page table entry의 물리주소와 권한을 추출하고 권한은 읽기 권한으로 변경하고 물리주소를 자식 page directory에서 부모 페이지의 가상주소와 동일한 가상주소를 가리키는 곳에 맵핑합니다.

```
//vm.c
pde_t*
shareuvm(pde_t* pgdir, uint sz){
  pde_t* d; //new page directory
  pte_t* pte; //page table entry
  uint pa, flags;
  if((d=setupkvm())==0)
    return 0;
  for(uint i = 0; i < sz; i += PGSIZE){
    pte = walkpgdir(pgdir, (void*)i, 0);
    if(pte == 0)
      panic("shareuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("shareuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte) & ~PTE_W; //read only로 변경
    incr_refc(pa); //물리 페이지 참조횟수 증가
    *pte=pa|flags|PTE_P; //부모 페이지 권한 변경
    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0){ //자식 페이지 테이블에 매핑
        goto bad;
    }
  }
  lcr3(V2P(pgdir)); //페이지 테이블 재설치및 TLB flush
  return d;
bad:
  freevm(d);
  return 0;
}
```

Cow (Make a Copy)

- **Initial Sharing으로 발생하는 에러**
  - trap 14: page fault
  - err7: 2진수로 111(usermode, write, protection violation error)를 의미하므로 해당 trap과 error일때만 cow_handler를 처리해 주어야한다.



- **trap 처리**
  - 위의 에러인 경우에만 Cow_handler 호출 아니면 xv6 default와 똑같이 처리하도록 한다.

```
void
trap(struct trapframe* tf)
{
...
case T_PGFLT:
  if(tf->err==7){ //111(usermode, write, protection violation error)
    Cow_handler();
    lapiceoi(); //성공적으로 interrupt의 종료를 표시
    break;
  }else{
    cprintf("pid %d %s: trap %d err %d on cpu %d "
          "eip 0x%x addr 0x%x--kill proc\n",
          myproc()->pid, myproc()->name, tf->trapno,
          tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
    break;
  }
...
}
```

- **void Cow_handler(void)**
  - 물리 페이지가 하나의 프로세스에서만 참조되는 경우 flags만 변경하여 TLB flush를 진행하고 두 개 이상의 프로세스에서 참조되는 경우 새로운 메모리를 할당하고 이전의 페이지를 page table에서 제거한 후에 새로운 메모리를 제거한 위치에 맵핑합니다.

```
//vm.c
//user memory의 복사본을 만드는 trap handler
void Cow_handler(void){
char* va=(char*)PGROUNDDOWN(rcr2());
struct proc* curproc=myproc();
pte_t* pte;
uint pa, flags;
char* mem;
if((pte=walkpgdir(curproc->pgdir,(char*)va,0))==0)
  panic("Cow_handler: pte should exist");
if(!(*pte & PTE_P))
  panic("Cow_handler: page not present");
pa= PTE_ADDR(*pte);
flags=PTE_FLAGS(*pte);
int sharecnt=get_refc(pa);
if(sharecnt==1){ //공유되지 않는 경우
  flags|=PTE_W; //쓰기 권한 부여
  *pte=pa|flags;
  lcr3(V2P(curproc->pgdir)); //TLB flush
  return;
```

```
  }else{ //공유된 경우
    if((mem=kalloc())==0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    flags|=PTE_W; //쓰기 권한 부여
    *pte= *pte & ~PTE_P; //page table entry 없음 처리
    lcr3(V2P(curproc->pgdir)); //TLB flush(안 해줄 경우 일정확률로 test case 2번 실패)
    if(mappages(curproc->pgdir, (void*)va, PGSIZE, V2P(mem), flags) < 0){
      cprintf("Cow_handler: fail to mappages\n");
      kfree(mem);
      goto bad;
    }
    kfree((char*)P2V(pa));
    lcr3(V2P(curproc->pgdir));
    return;
  }

bad:
  freevm(curproc->pgdir);
  return;
}
```

- **공유 페이지 참조횟수 감소 제어**
  - decr_refc를 따로 사용하지 않고 내부에서 감소처리하고 마지막 공유일때만 페이지를 free 시킨다.

```
//kalloc.c
void
kfree(char* v)
{
...
r = (struct run*)v;
if(kmem.refc[((uint)V2P(v))/PGSIZE]>1){
  kmem.refc[((uint)V2P(v))/PGSIZE]--;
  if(kmem.use_lock)
    release(&kmem.lock);
  return;
}

// Fill with junk to catch dangling refs.
memset(v, 1, PGSIZE);


r->next = kmem.freelist;
kmem.freelist = r;
kmem.refc[((uint)V2P(v))/PGSIZE]=0;
...
}
```

Common Function

- **void incr_refc(uint pa)**

```
void incr_refc(uint pa) {
  char* v=(char*)P2V(pa);
  if((uint)v%PGSIZE || v<end || V2P(v)>=PHYSTOP)
    panic("incr_refc");
  if(kmem.use_lock)
    acquire(&kmem.lock);
```

```
  kmem.refc[pa/PGSIZE]++; //참조횟수 증가
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

- **void decr_refc(uint pa)**
  - 구현은 되어 있지만 실제로 사용되지는 않는다.

```
//kalloc.c
void decr_refc(uint pa) {
  char *v=(char*)P2V(pa);
  if((uint)v%PGSIZE || v<end || V2P(v)>=PHYSTOP)
    panic("decr_refc");
  if(kmem.use_lock)
    acquire(&kmem.lock);
   kmem.refc[pa/PGSIZE]--; //참조횟수 감소
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

- **int get_refc(uint pa)**

```
int get_refc(uint pa) {
  char* v=(char*)P2V(pa);
  if((uint)v%PGSIZE || v<end || V2P(v)>=PHYSTOP)
    panic("get_refc");
  if(kmem.use_lock)
    acquire(&kmem.lock);
  int ret=kmem.refc[pa/PGSIZE]; //참조횟수
  if(kmem.use_lock)
    release(&kmem.lock);
  return ret;
}
```

System Call

- **int countfp(void)**

```
//kalloc.c
//free page의 총 개수 반환
int countfp(void){
  if(kmem.use_lock)
    acquire(&kmem.lock);
  int cnt=0;
  struct run* r=kmem.freelist;
  while(r){
    cnt++;
    r=r->next;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return cnt;
}
```

- **int countvp(void)**
  - PTE_U(user page table), PTE_P(page table present) bit로 점검하여 할당된 가상페이지의 수를 반환합니다.

```
//vm.c
//현재 프로세스에 할당된 가상 페이지 수를 반환하는 함수
int countvp(void){
  struct proc* curproc=myproc();
  int cnt=0;
  for(uint i=0;i<curproc->sz;i+=PGSIZE){
    pte_t* pte=walkpgdir(curproc->pgdir,(char*)i,0);
    if(pte!=0 && (*pte & PTE_U) && (*pte & PTE_P))
      cnt++;
  }
  return cnt;
}
```

- **int countpp(void)**
  - 1개의 page directory → NPDENTRIES 만큼의 page table 존재, 1개의 page table → NPTENTRIES만큼의 page table entry 존재하므로 해당 구조를 따라 탐색하면서 PTE_U, PTE_P bit로 점검하여 유효한 물리주소의 개수를 카운트합니다.

```
//vm.c
//현재 프로세스에 할당된 유효한 물리주소가 가리키는 페이지 수를 반환하는 함수
int countpp(void){
  struct proc* curproc=myproc();
  int cnt=0;
  pde_t* pgdir=curproc->pgdir;
  for(int i=0;i<NPDENTRIES;i++){
    if((pgdir[i] & PTE_U) && (pgdir[i] & PTE_P)){
      pte_t* pgtab=(pte_t*)P2V(PTE_ADDR(pgdir[i]));
      for(int j=0;j<NPTENTRIES;j++){
        if((pgtab[j] & PTE_P) && (pgtab[j] & PTE_U)){
          cnt++;
        }
      }
    }
  }
  return cnt;
}
```

- **int countptp(void)**
  - page directory를 저장하는 페이지와 그 페이지 directory에서 존재하는 모든 페이지 page table을 탐색하여 더 합니다. 이 과정에서 PTE_U를 검사하지 않으므로 커널과 유저 레벨의 페이지 테이블을 모두 고려할 수 있습니다.

```
//vm.c
//프로세스의 페이지 테이블에 의해 할당된 페이지의 수를 반환
int countptp(void){
  struct proc* curproc=myproc();
  int cnt=0;
  pde_t* pgdir=curproc->pgdir;
  cnt+=1 ; //페이지 디렉토리
  for(int i=0;i<NPDENTRIES;i++){
    if((pgdir[i] & PTE_P) && (pgdir[i])){
      cnt+=1; //페이지 테이블
    }
  }
  return cnt;
}
```

Result

- test0
  - vp와 pp의 개수가 같고 ptp가 66개인 것으로 보아 정상적으로 수행된 것으로 확인된다.

```
$ test0
[Test 0] default
fp:56733, vp:2, pp:2, ptp:66
fp:56732, vp:3, pp:3, ptp:66
ptp: 66 66
```

- test2
  - copy를 진행했을때 성공적으로 다른 메모리가 사용되는 것을 알 수 있다.

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

- test1
  - 성공적으로 메모리를 공유함을 알 수 있다.

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

- test3
  - 10개의 자식을 생성했을때에도 Make a Copy가 정상적으로 수행됨을 알 수 있다.

```
$ 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
```

Trouble shooting

- panic(”remap”) 문제
  - shareuvm에서 발생
    - 만약에 mappages를 수행할때 page directory에 va에 해당하는 기존에 있던 페이지를 따로 처리하지 않고 다시 페이지 할당을 하면 발생하는 문제로 PTE_P flag를 해제하여 페이지를 없애 해결할 수 있다.
  - Cow_handler에서 발생
    - rcr2()에서 받은 가상메모리 주소에 대해서 mappages를 했더니 해당 문제가 발생하였다.
    - 발생이유는 mappages는 va~va+PGSIZE-1까지의 메모리를 맵핑하게되는데 주어진 가상메모리가 페이지의 가상메모리의 시작주소가 아닌 경우 페이지를 침범하게 되면서 발생한 문제가 PGROUNDOWN(rcr2())처리를 하여 해당 문제를 해결하였다.
- initial sharing 이후 무한 재부팅 문제
  - 처음 시작과정에서 init:starting sh라는 문구가 나오고 계속 재부팅이 되던 문제가 있었다. 이때 trap handler를 구현했음에도 handler로 넘어가긴 하지만 계속 재부팅이 되던 문제가 있었다.
  - 처음에는 권한 부족으로 인한 문제라고 생각되어 아래와 같은 코드를 trap.c 에 tvinit에 추가하여 부팅하는데에 성공은 했지만 위에서 언급한 panic문제가 발생하였고 panic 문제를 해결함과 동시에 해당 문제가 해결되었고 이후 아래 코드를 제거를 진행하였다.

```
SETGATE(idt[T_PGFLT], 1, SEG_KCODE<<3, vectors[T_PGFLT], DPL_USER);
```