Project04 Wiki

한양대학교 컴퓨터소프트웨어학부 2020089007 김형민

목차

Design

System calls

wrapper functions

각 시스템 콜들의 wrapper 함수들의 동작은 동일하다.

시스템 콜들에서 반환한 값을 그대로 반환한다.

countfp

여기서 수행해야할 동작은 다음과 같다.

1. kmem.freelist를 순차적으로 순회하며 유효할 경우 카운트를 1 증가시킨다.
2. 유효하지 않은 값(0)을 만나면 순회에서 빠져나온 뒤 카운트 값을 반환한다.

countvp

여기서 수행해야할 동작은 다음과 같다.

1. 시작할 가상 주소를 0으로 설정한다.
2. 마지막 주소를 myproc()→sz에서 PGROUNDUP한 값으로 설정한다.
    a. PGSIZE+1 바이트만을 사용한다고 하더라도 2페이지가 필요하기 때문이다.
3. PGSIZE만큼 더해가면서 카운트를 1 증가시킨다.
4. 마지막 주소 이상이 되면 반복을 탈출하고 카운트 값을 반환한다.

countpp

여기서 수행해야할 동작은 다음과 같다.

1. 시작할 가상 주소를 0으로 설정한다.
2. walkpgdir을 통해 해당 가상 주소에 대한 물리 페이지 매핑이 존재하는지 확인한다.
3. PTE_P와의 bitwise and 연산을 통해 매핑이 존재하는지 확인한다.
4. 매핑이 존재한다면 카운트 값을 1 증가시키고, 가상 주소를 PGSIZE만큼 증가시킨다.
5. 매핑이 존재하지 않는다면 반복을 탈출하고 카운트 값을 반환한다.

countptp

여기서 수행해야할 동작은 다음과 같다.

1. 프로세스의 pgdir 엔트리들을 순차적으로 순회한다.
2. PTE_P 와의 bitwise and 연산을 통해 엔트리들이 존재하는지 확인한다.
3. 존재한다면 카운트 값을 1 증가시킨다.
4. NPDENTRIES만큼의 엔트리들을 전부 순회했다면 반복에서 빠져나온다.
5. pgdir 자체를 저장하기 위한 페이지 1개만큼의 분량을 더해 카운트 값을 반환한다.

Initial sharing

Data structure for tracking reference count

```
// kalloc.c
```

```
char pgrefcnt[PHYSTOP/PGSIZE];
```

페이지의 참조 횟수를 추적하기 위한 전역 변수이다. xv6에서는 PHYSTOP 위의 메모리는 유저 메모리를 사용하지 않는다. 또, 페이지 단위로 참조 횟수를 추적하므로 결론적으로는 PHYSTOP/PGSIZE 크기의 char 배열을 사용한다. 다른 조건을 변경하지 않았을 때 프로세스의 개수는 최대 64개 이므로 char 타입만으로도 페이지의 참조 횟수를 표현하는데 충분하다.

incr_refc

여기서 수행해야할 동작은 다음과 같다.

1. kmem.use_lock이 1이면 락을 획득한다.
2. 인자로 넘어온 주소의 페이지의 참조 횟수를 1 증가시킨다.
3. 앞서 락을 획득했다면 락을 해제한 뒤 리턴한다.

decr_refc

여기서 수행해야할 동작은 다음과 같다.

1. kmem.use_lock이 1이면 락을 획득한다.
2. 인자로 넘어온 주소의 페이지의 참조 횟수를 1 감소시킨다.
3. 앞서 락을 획득했다면 락을 해제한 뒤 리턴한다.

get_refc

여기서 수행해야할 동작은 다음과 같다.

1. kmem.use_lock이 1이면 락을 획득한다.
2. 인자로 넘어온 주소의 페이지의 참조 횟수를 리턴을 위한 변수에 저장한다.
3. 앞서 락을 획득했다면 락을 해제한 뒤, 앞서 확인한 참조 횟수를 반환한다.

kalloc

여기서 수행해야할 동작은 다음과 같다.

1. 처음 할당이 일어나는 페이지의 참조 횟수를 1로 설정해준다.

kfree

여기서 수행해야할 동작은 다음과 같다.

1. 인자로 넘어온 주소의 페이지의 참조 횟수를 1 감소시킨다.
2. 페이지의 참조 횟수가 0이 되었다면 실제로 그 페이지를 할당 해제한다.
3. 참조 횟수가 1 이상이라면 단순히 참조 횟수만 감소시킨 뒤 리턴한다.

copyuvm

여기서 수행해야할 동작은 다음과 같다.

1. 자식 프로세스가 새로운 페이지 테이블을 할당받는다.
2. walkpgdir 통해 PTE의 주소를 획득한다.

3. PTE_P와의 bitwise and 연산을 통해 해당 PTE가 유효한지 확인한다.
4. 부모의 PTE의 쓰기 권한(PTE_W)를 해제한다.
5. 부모의 페이지 테이블을 재설치한다.
6. 해당 물리 페이지의 참조 횟수를 1 증가시킨다.
7. 자식의 페이지 테이블에 해당 물리 페이지를 매핑한다.
8. 새로 만든 자식의 pgdir을 반환한다.
9. 2부터 9까지의 과정을 페이지 단위로 매핑이 끝날 때 까지 반복한다.

Make a copy

T_PGFLT handling

여기서 수행해야할 동작은 다음과 같다.

1. trap.c에서 T_PGFLT를 캐치하기 위한 case 단락을 추가한다.
2. CoW_handler를 호출하여 exception을 핸들링한다.

CoW_handler

여기서 수행해야할 동작은 다음과 같다.

1. rcr2를 통해 페이지 폴트가 일어난 가상 주소를 획득한다.
2. PGROUNDDOWN으로 폴트가 일어난 페이지의 시작 주소를 획득한다.
3. walkpgdir을 통해 해당 PTE의 주소를 획득한다.
4. get_refc를 통해 해당 페이지의 참조 횟수가 현재 1이라면 쓰기 권한을 부여한다.
    a. 해당 물리 페이지를 가리키는 유일한 프로세스라는 뜻이기 때문이다.
5. 쓰기 권한 부여 후 페이지 테이블을 재설치하고 리턴한다.
6. 참조 횟수가 1이 아니라면 참조 횟수를 1 감소시킨다.
7. 새로운 물리 페이지를 할당한 뒤, 기존 페이지의 내용을 복사한다.
8. 앞서 얻은 PTE에 새로 할당한 물리 페이지에 쓰기 권한을 부여하여 매핑한다.
9. 페이지 테이블을 재설치한다.

Implement

System calls

countfp

```
// cow.c
int
countfp(void)
```

```
{
  ...
  r = kmem.freelist;
  while(r){
    rv++;
    r = r->next;
  }
  return rv;
}
```

앞서 디자인에서 설명한 대로 구현하였다. 간단하므로 추가적인 설명은 필요하지 않을 것이다.

countvp

```
// cow.c
int
countvp(void)
{
  ...
  a = (char*)0;
  last = (char*)PGROUNDUP(myproc()->sz);
  rv = 0;

  for(;;){
    if(a >= last)
      break;
    rv++;
    a += PGSIZE;
  }
  return rv;
}
```

앞서 디자인에서 설명한 대로 구현하였다. 간단하므로 추가적인 설명은 필요하지 않을 것이다.

countpp

```
// cow.c
int
countpp(void)
{
  ...
  a = (char*)0;
  rv = 0;

  for(;;){
    // 물리 페이지와의 매핑이 존재하는지 확인해야하므로 walkpgdir 호출
    if((pte = walkpgdir(myproc()->pgdir, a, 0)) == 0)
      return -1;
```

```
    // 매핑이 존재하지 않는다면 break
    if(!(*pte & PTE_P))
      break;
    rv++;
    a += PGSIZE;
  }
  return rv;
}
```

가상 주소 0부터 PGSIZE만큼 증사켜가면서 물리 페이지와의 매핑이 존재하는지 확인한다. xv6는 가상주소 0부터 순차적으로 쌓아가면서 페이지를 할당하기 때문에 매핑이 존재하지 않는 가상주소에 도달했다면 그 이상은 고려할 필요가 없다.

countptp

```
// cow.c
int
countptp(void)
{
  ...
  pgdir = myproc()->pgdir;

  for(i = 0; i < NPDENTRIES; i++){
    pde = &pgdir[i];
    if(*pde & PTE_P)
      rv++;
  }

  return rv + 1;
}
```

앞서 디자인에서 설명한 대로 구현하였다. xv6는 2레벨 페이지 테이블을 사용한다. 1단계 페이지 테이블을 페이지 pgdir, 2단계 페이지 테이블을 ptable이라고 명명하겠다.

- 반복문은할당된 ptable의 개수를 센다.
- 마지막 +1은 pgdir을 고려하여 더해주는 것이다.

Initial sharing

incr_refc

```
// cow.c

void
incr_refc(uint pa)
{
  if(kmem.use_lock)
```

```
    acquire(&kmem.lock);
  pgrefcnt[pa/PGSIZE]++;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

앞서 디자인에서 설명한 대로 구현하였다. 간단하므로 추가적인 설명은 필요하지 않을 것이다.

decr_refc

```
// cow.c

void
decr_refc(uint pa)
{
  if(kmem.use_lock)
    acquire(&kmem.lock);
  pgrefcnt[pa/PGSIZE]--;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

앞서 디자인에서 설명한 대로 구현하였다. 간단하므로 추가적인 설명은 필요하지 않을 것이다.

get_refc

```
// cow.c

int
get_refc(uint pa)
{
  ...
  if(kmem.use_lock)
    acquire(&kmem.lock);
  rv = pgrefcnt[pa/PGSIZE];
  if(kmem.use_lock)
    release(&kmem.lock);
  return rv;
}
```

앞서 디자인에서 설명한 대로 구현하였다. 간단하므로 추가적인 설명은 필요하지 않을 것이다.

kalloc

```
// kalloc.c

char*
```

```
kalloc(void)
{
  ...
  if(r){
    kmem.freelist = r->next;
    pgrefcnt[V2P((uint)r)/PGSIZE] = 1;
  }
  ...
}
```

물리 주소 단위로 추적하므로 V2P를 통해 물리 주소로 변환하여 참조 횟수를 1로 할당한다.

kfree

```
// kalloc.c

void
kfree(char *v)
{
  ...
  pa = V2P(v);

   decr_refc(pa);
   if(get_refc(pa) > 0)
     return;

  ...
}
```

1. 페이지의 참조 횟수를 1만큼 감소시킨다.
2. 이후 참조 횟수를 확인하여 0보다 크다면 아직 참조하고 있는 페이지가 존재하므로 할당해제 하지 않는다.
3. 감소시킨 이후 참조 횟수가 0이라면 해당 페이지를 참조하고 있는 프로세스가 더는 없다는 뜻이므로 실제로 그 페이지를 할당 해제한다.

copyuvm

```
// vm.c

pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  ...
  // Allocate page table
  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    ...
```

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

    // Clear write bit.
    flags &= ~PTE_W;
    *pte &= ~PTE_W;
    lcr3(V2P(pgdir));

    // Increase page's ref cnt by 1.
    incr_refc(pa);

    // Map page table of child.
    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0)
      goto bad;
    lcr3(V2P(d));
  }
  return d;
}
```

1. 자식 프로세스에게 페이지 테이블을 할당한다.
2. 부모의 PTE와 자식의 PTE의 write flag를 비활성화한다.
3. 부모 프로세스의 페이지 테이블을 재설치한다.
4. 해당 페이지의 참조 횟수를 1 증가시킨다.
5. 자식의 페이지테이블에 해당 물리페이지를 매핑시킨다.
6. 자식 프로세스의 페이지 테이블을 재설치한다.
7. 자식의 pgdir을 리턴한다.

Make a copy

T_PGFLT handling

```
// trap.c
...
case T_PGFLT:
  CoW_handler();
  break;
...
```

CoW_handler

```
// cow.c

void
CoW_handler(void)
```

```
{
  ...
  // Read cr2 register to specify address that pgflt occurred.
  va_pgflt = rcr2();
  p = myproc();

  va = (char*)PGROUNDDOWN(va_pgflt);

  if((pte = walkpgdir(p->pgdir, va, 0)) == 0){
    ...
    return;
  }

  // 해당 물리 페이지를 참조하는 유일한 프로세스인 경우
  if(get_refc(PTE_ADDR(*pte)) == 1){
    *pte |= PTE_W;
    lcr3(V2P(p->pgdir));
    return;
  }

  mem = kalloc();
  if(mem == 0){
    ...
    return;
  }

  memmove(mem, (char*)P2V(PTE_ADDR(*pte)), PGSIZE);
  *pte = V2P(mem) | PTE_FLAGS(*pte) | PTE_W;
  lcr3(V2P(p->pgdir));
}
```

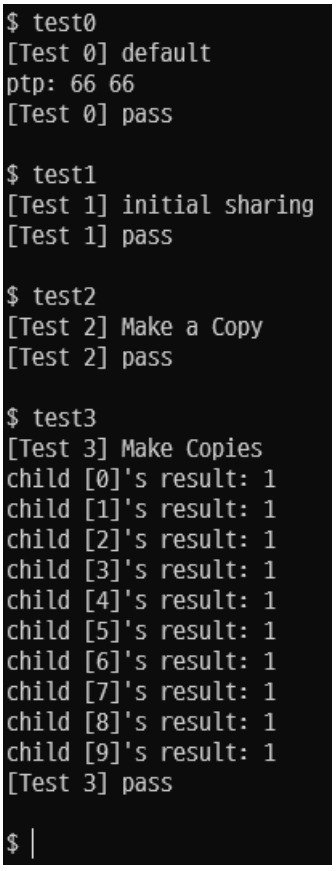
처리해야할 경우의 수는 2가지이다.

1. 페이지 폴트가 발생한 페이지의 참조 횟수가 1인 경우
2. 페이지 폴트가 발생한 페이지의 참조 횟수가 1보다 큰 경우

1의 경우에는 단순히 해당 PTE의 write flag를 활성화하고 페이지 테이블을 재설치한다.

2의 경우에는 새로운 물리 페이지를 할당해줘야한다. 따라서 kalloc으로 페이지 하나를 할당하고, memove로 기존의 내용을 새로 할당한 페이지에 복사한다. PTE의 경우에는 이미 존재하는 상황이므로 mappages를 호출할 필요 없이 PTE에 저장된 물리주소와 플래그들만 변경해주면 된다. 마찬가지로 write flag를 활성화시키고 페이지 테이블을 재설치해준다.

Result



문제 없이 모든 테스트를 마쳤다.

Trouble shooting

test1과 test3를 진행하는 과정에서 프리 페이지의 개수가 1개씩 모자라는 문제가 있었다. 관련된 함수들인 kalloc, kfree, CoW_handler, copyuvm에서 순차적으로 로깅을 하면서 문제를 추적해본 결과 스택이 반환되지 않아서 발생한 문제임을 확인하였다. 하나씩 대조해가면서 추적해보니 CoW_handler에서 새로운 물리 페이지를 할당한 후 기존 페이지의 참조 횟수를 1 감소시켰어야 하는데 그것을 하지 않은 것이 문제였다. 이 부분을 해결하니 정상적으로 작동하였다.