Operating System

Project 5

2014-15703 송화평

1. 프로젝트 요약

본 프로젝트의 목적은 Copy-on-Write(COW)를 구현하여 프로세스를 fork를 통해 새로 생성할 때 메모리 공간과 이를 복사하는 시간을 줄임으로써 최적화하는 것이다. 이 효과를 극대화하기 위하여 프로그램이 execute 되는 시점부터 ELF 헤더의 정보를 읽어 각 program section이 어떤 정보를 가지고 있는 지 파악하여 code segment인지 data segment인지 구별하여 메모리에 로드하도록 하였다. 프로그램이 메모리에 로드되고 실행되어 프로세스를 이루면 이 프로세스는 fork를 통해 자식 프로세스를 생성할 수 있다. 이 때 code segment는 write할 수 없는 영역이므로 부모 프로세스와 공유하도록 하고, data, stack, heap segment는 최초에는 부모 프로세스와 공유하다 write 요청이 들어오면 page fault를 거침으로써 COW를 실행하여 다른 메모리 공간에 페이지 단위로 복사하도록 구현하였다. 이 과정은 복잡한 메모리 조작을 거치므로 여러 예외적 상황이 일어나게 마련이므로 이에 대해서도 처리를 할 수 있도록 하였고, 메모리 누수 없이 correct한 동작을 보장하였다.

2. 구현 내용

2.1. Program header 구분

시스템 프로그래밍에서 다루었듯이 ELF 포맷을 따르는 바이너리는 code와 data영역에 해당하는 프로그램 헤더를 따로 가지고 있어 각각 메모리의 다른 영역에 맵핑되도록 할 수 있다. 본 구현에서는 메모리에 올라갈 때 code segment에는 R-E, code segment에는 RW- 권한을 주어 부모 프로세스와 자식 프로세스가 code segment를 공유할 수 있도록 하였다. 이를 가능하게 해주는 가장 첫 걸음은 kernel/exec.c에서 일어난다.

```
int
exec(char *path, char **argv)
{
  // Load program into memory.
  sz = 0;
  sz_backup = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){

    ...

    if((sz = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
#ifndef SNU
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
#endif
    if((ph.flags & ELF_PROG_FLAG_READ) &&
```

```
        !(ph.flags & ELF_PROG_FLAG_WRITE) &&
        (ph.flags & ELF_PROG_FLAG_EXEC)
      ){
        if(changeflags(pagetable, sz_backup, ph.vaddr + ph.memsz, PTE_
          goto bad;
      }
      else if((ph.flags & ELF_PROG_FLAG_READ) &&
              (ph.flags & ELF_PROG_FLAG_WRITE) &&
              !(ph.flags & ELF_PROG_FLAG_EXEC)
              ){
        if(changeflags(pagetable, sz_backup, ph.vaddr + ph.memsz, PTE_
          goto bad;
      }
      else;

    if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;

    sz_backup = sz;
  }
    ...

  // Allocate two pages at the next page boundary.
  // Use the second as the user stack.
  sz = PGROUNDUP(sz);
  sz_backup = sz;
  if((sz = uvmalloc(pagetable, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  if(changeflags(pagetable, sz_backup, sz_backup + 2*PGSIZE, PTE_R | P
    goto bad;

    ...
```

위와 같이 uvmalloc() 함수로 segment가 프로세스의 페이지블에 매핑된 후, 각 segment의 종류에 따라 kernel/vm.c에 선언된 changeflags() 함수를 호출하여 해당 segment의 권한을 수정한다. 만약 해당 segment의 프로그램 헤더가 R-E 플래그를 가지고 있다면 메모리 위에는 PTE_R, PTE_X, PTE_U의 권한을 가진 상태로 올리고, RW-의 플래그를 가지고 있다면 메모리 위에는 PTE_R, PTE_W, PTE_U의 권한을 가진 상태로 올린다. 경계 페이지와 스택 페이지를 할당할 때도 같은 방식으로 권한을 주었다.

권한을 조정하는 방법에는 uvmalloc() 함수의 패러미터를 통해 권한을 주는 방법도 있었지만, 기존에 존재하는 함수를 수정하고 싶지 않아 changeflags() 함수를 새로 정의하였다. 이 함수의 동작 방식은 uvmalloc() 함수와 그 안에서 호출되는 mappages() 함수의 동작 방식과 정확히 같고 단지 권한만을 수정하고 종료한다는 점만 다르다. 즉 시작 지점부터 끝 지점까지의 매 페이지에 해당하는 pte를 불러와 그 권한을 원하는 대로 수정한다.

2.2. Copy-on-Write 구현

이 과정이 끝나면 프로그램은 메모리 위에 segment 단위로 로드된 상태가 된다. 이 때 해당 프로세스가 fork를 통해 자식 프로세스를 만들 때 code segment를 공유해야 한다. 이 작업은 kernel/vm.c의 uvmcopy() 함수에서 이루어진다. 기존 xv6의 uvmcopy() 함수는 부모 프로세스가 차지하는 메모리 크기만큼 새 메모리를 할당하고 그 내용을 복사하여 자식 프로세스의 페이지테이블에 맵핑하는 방식으로 구현되어 있었다. 하지만 상술했듯 이는 시간과 공간 측면에서 비효율적이다. 본 구현에서 uvmcopy() 함수는 메모리를 새로 할당하는

일 없이 자식 프로세스의 페이지테이블을 기존 부모 프로세스의 페이지테이블에 맵핑된 물리 주소에 다시 맵핑한다.

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    ...

  for(i = 0; i < sz; i += PGSIZE){

    ...

    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);

    if(!(flags & PTE_U));
      // printf("[DEBUG] uvmcopy: non-user segment at REL_PA %x\n", (u
    else if((flags & PTE_X) && !(flags & PTE_W) && (flags & PTE_R));
      // printf("[DEBUG] uvmcopy: code segment at REL_PA %x\n", (uint
    else {
      // printf("[DEBUG] uvmcopy: non-code segment at REL_PA %x\n", (u
      *pte = *pte & ~PTE_W;
      flags = PTE_FLAGS(*pte);
    }

    if(mappages(new, i, PGSIZE, pa, flags) != 0){
      kfree((void *) pa);
      goto err;
    }

    incr_refcnt((void *) pa);
  }
    ...
```

위 코드에서 볼 수 있듯, 기존 부모 페이지테이블에 맵핑된 물리 주소 프레임 각각에 대해 이를 복사하는 일 없이 non-code segment에 대해서만 write 권한을 제거해주는 일만 하고 자식 프로세스의 페이지테이블에 다시 맵핑해주고 있다. 여기서 write 권한을 제거해주는 이유는 uvmcopy() 함수가 실행되는 시점에서 최소 두 개 이상의 프로세스의 페이지테이블이 같은 물리 주소를 맵핑하고 있기 때문에 한 프로세스가 다른 프로세스 몰래 해당 페이지에 쓰기를 수행하는 것을 막기 위해서이다.

또한 incr_refcnt() 함수가 호출되는 것을 볼 수 있는데, 이 함수는 kernel/kalloc.c에 존재한다.
kernal/kalloc.c 파일에는 커널 레벨에서 물리 주소를 관리하는 함수와 자료구조가 정의되어 있다. 이 때 각 메모리 프레임을 맵핑하는 프로세스의 수를 파악하기 위해 아래와 같은 자료구조를 사용하였다.

```
struct {
  struct spinlock lock;
  struct run *freelist;
  uint16 refcnts[(PHYSTOP - KERNBASE) >> PGSHIFT];
} kmem;
```

xv6에서 물리 주소는 KERNBASE == 0x80000000부터 시작하고 PHYSTOP == 0x86400000에서 끝나며, 각 프레임의 크기는 4KB이므로 이에 해당하는 PGSIZE == 12만큼 우측으로 shift함으로써 모든 물리

프레임에 대한 프로세스 맵핑 개수에 대한 정보를 리스트 자료구조에 담을 수 있다. 이 때 이 리스트가 int가 아닌 uint16의 배열로서 선언된 이유는 단순히 공간을 절약하기 위함이다. 한 프레임을 256개 이상의 프로세스가 맵핑하는 것도 가능성이 커 보이지 않는 만큼 uint8로 선언하는 등의 방법을 통해 공간을 추가적으로 절약할 수도 있을 것이다. 유저 프로세스가 사용하는 가상 메모리 주소에 대응되는 물리 주소의 값은 KERNBASE와 PHYSTOP 사이의 값을 가지고, 관리해야 하는 메모리는 페이지 단위이므로 위와 같이 PGSHIFT만큼 비트를 당기게 되면 모든 물리 프레임의 맵핑 횟수를 셀 수 있게 된다.

```
void
incr_refcnt(void *pa){
  acquire(&kmem.lock);
  kmem.refcnts[INDEX(REL_PA(pa))]++;
  release(&kmem.lock);

  return;
}

void
decr_refcnt(void *pa){
  acquire(&kmem.lock);
  kmem.refcnts[INDEX(REL_PA(pa))]--;
  release(&kmem.lock);

  return;
}

int
get_refcnt(void *pa){
  return kmem.refcnts[INDEX(REL_PA(pa))];
}
```

이렇게 선언된 refcnts 배열의 값을 읽거나 여기에 값을 쓰기 위해서 세 가지 함수를 정의하였다. refcnts 배열에 값을 쓰는 함수는 incr_refcnt() 함수와 decr_refcnt() 함수로, 물리 주소를 받아 해당 물리 프레임의 맵핑 횟수에 1을 더하거나 빼는 함수이다. refcnts 배열이 kmem 구조체 안에 선언되었기 때문에 실제로 refcnts 리스트를 업데이트 하기 전에 kmem의 lock을 걸고, 업데이트가 끝나면 lock을 해제해주도록 구현하였다. get_refcnt() 함수는 물리 주소를 받아 해당하는 물리 프레임의 맵핑 횟수를 반환하는 함수이다. 이 함수는 업데이트를 하는 함수가 아니라 단순히 값을 받아오는 함수이기 때문에 lock을 걸고 해제하는 작업 없이 바로 값을 읽어오도록 하였다.

```
void *
kalloc(void)
{
  ...
  kmem.refcnts[INDEX(REL_PA(r))] = 1;
  ...
```

메모리를 사용하기 위해서는 어떤 프로세스가 어떤 함수를 부르든 결국에는 kalloc() 함수를 호출하게 된다. 이 때 kalloc() 함수가 비어있는 프레임을 내주기 전, 해당 프레임의 맵핑 횟수를 1로 수정해주어야 한다. 이 작업은 kalloc() 함수 내부에 한 줄만 추가함으로써 간단하게 수행하였다.

```
void
kfree(void *pa)
```

```
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PI
    panic("kfree");

  if(get_refcnt((void *) pa) > 1){
    decr_refcnt((void *) pa);
    return;
  }
  ...
```

또한 프로세스의 페이지테이블의 맵핑을 해제하거나 하는 등의 이유로 메모리를 반환해야 하는 경우가 생기면 kfree() 함수가 호출된다. 그런데 어떤 물리 프레임을 두 개 이상의 프로세스가 공유하고 있을 때 이 프레임의 메모리를 반환하게 되면 다른 프로세스에 문제가 생긴다. 따라서 해당 프레임에 맵핑하고 있는 프로세스가 두 개 이상일 때, 즉 get_ref() 함수의 반환값이 1보다 클 때는 단순히 해당하는 맵핑 횟수만 감소시키고 바로 함수가 끝나도록 구현하였다. 따라서 어떤 프로세스가 죽어 해당 페이지테이블의 맵핑이 사라지더라도 다른 프로세스의 페이지테이블은 correct한 상태를 유지할 수 있게 된다.

2.3. Page fault 처리

xv6에서 page fault는 기본적으로 instruction page fault, load page fault, store page fault의 세 가지가 존재한다. 본 프로젝트의 COW 구현을 통해서 발생할 수 있는 page fault는 store page fault이므로 이 page fault에 대해서만 처리하도록 하였다.

Store page fault는 다음과 같은 상황에서 발생한다.

> 1. Valid bit이 0인 페이지에 store 시도
> 2. Write bit이 0인 (또는 Execute bit이 1인) 페이지에 store 시도
> 3. User bit이 0인 페이지에 store 시도

위와 같은 이유로 store page fault가 발생하면 커널로 컨트롤이 옮겨가게 되고, usertrap()에서 해당 page fault를 처리하게 된다. 이 때 r_scause()로 exception의 종류를 알 수 있고, r_stval()로 그 exception을 발생시킨 가상 메모리 주소를 알 수 있다. 따라서 kernel/trap.c에서 다음과 같이 구현하여 store page fault가 발생하면 kernel/vm.c에 정의된 cowhandler() 함수를 호출하여 정말로 문제가 생긴 것인지, 아니면 단순히 Copy-on-Write를 실행하면 되는 것인지 판단하도록 하였다.

```
void
usertrap(void)
{
  ...
  } else if(r_scause() == 15){  // store page fault
    if(cowhandler(p->pagetable, r_stval()) < 0)
      p->killed = 1;
  }
  ...
```

cowhandler() 함수는 페이지테이블과 store page fault가 발생한 가상 주소를 받아, 실제로 문제가 발생한 경우이면 -1을 반환하고 Copy-on-Write을 적용해주면 되는 경우라면 0을 반환한다. 만약 -1을 반환하게 되는 경우이면 이를 호출한 usertrap에서 해당 프로세스를 죽일 것이다. 이렇게 되는 경우는 가상 주소가 KERNBASE보다 크거나 같은 경우 즉 유효하지 않은 범위이거나, 해당 가상 주소가 페이지테이블에 맵핑되지

않은 주소이거나, 가상 주소에 맵핑된 프레임이 유저 프레임이 아니거나, valid bit이나 write bit이 0이거나, execute bit이 1이거나, leaf page table이 아니거나, 메모리를 할당해줄 수 없는 경우이다.

```
int
cowhandler(pagetable_t pagetable, uint64 va){
  uint64 pa;
  uint16 refcnt;
  pte_t *pte;
  char *mem;

  if(va >= KERNBASE)
    return -1;

  va = PGROUNDDOWN(va);
  if((pte = walk(pagetable, va, 0)) == 0)
    return -1;

  pa = PTE2PA(*pte);

  if(!(*pte & PTE_U) || !(*pte & PTE_V) || (*pte & PTE_W))
    return -1;

  if((*pte & (PTE_X | PTE_W | PTE_R | PTE_V)) == PTE_V)
    return -1;

  if(*pte & PTE_X)
    return -1;

  refcnt = get_refcnt((void *) pa);
  if(refcnt > 1){
    if((mem = kalloc()) == 0){
      kfree(mem);
      return -1;
    }

    memmove(mem, (char *) pa, PGSIZE);

    *pte = PA2PTE(mem) | PTE_FLAGS(*pte);

    decr_refcnt((void *) pa);
  }

  refcnt = get_refcnt((void *) pa);
  if(refcnt == 1){
    if((pte = walk(pagetable, va, 0)) == 0)
      return -1;

    if(!(*pte & PTE_U) || !(*pte & PTE_V) || (*pte & PTE_W))
      return -1;

    if(!(*pte & PTE_X))
      *pte = *pte | PTE_W;
    else return -1;
  }

  return 0;
}
```

이러한 경우가 아니라면 cowhandler() 함수는 먼저 페이지테이블에서 해당 가상 주소에 대응되는 pte와 물리 주소를 찾아 먼저 그 맵핑 횟수를 검사한다. 만약 해당 프레임이 2개 이상의 프로세스에 의해 맵핑되었다면 kalloc() 함수를 통해 프레임을 하나 받아와 물리 주소의 프레임을 새로 받아온 프레임으로 복사한 후 페이지테이블을 새로 받은 프레임을 가리키도록 수정하고, 원래 프레임의 맵핑 횟수를 1 감소시킨다. 이후 다시 맵핑 횟수를 검사하여 단 한개의 프로세스만 해당 프레임을 맵핑하고 있다면 그 프로세스는 이제 해당 페이지에 자유롭게 쓸 수 있어야 하므로 write bit을 1로 설정하고 종료하게 된다. 이러한 방식으로 Copy-on-Write가 수행되는 것이다.

```
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    cowhandler(pagetable, va0);
  ...
```

또한 프로그램을 메모리 위에 올리는 작업을 할 때 아규먼트 역시 해당 프로세스에 올라가야 한다. 이 때 kernel/vm.c 파일의 copyout() 함수를 호출하게 되는데, 이 때에도 cow가 일어날 수 있다. 따라서 위와 같이 cowhandler() 함수를 호출하는 한 줄을 삽입하여 해당 경우에도 올바르게 동작하도록 하였다.

2.4. Page frame의 해제

만약 어떤 물리 프레임의 유일한 맵핑이 해제된다면 해당 프레임은 free되어 추후 메모리를 요구하는 프로세스에게 줄 수 있어야 한다. 이 경우는 위에서 살펴본 kfree() 함수에서 if 조건이 맞지 않는 경우로, 이 때에는 정말로 해당 프레임을 해제해야 한다. 이는 xv6에서 이미 구현되어 있지만 다시 살펴본다면 아래와 같이 구현되어 있음을 알 수 있다.

```
void
kfree(void *pa)
{
  ...

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

  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
#ifdef SNU
  freemem++;
#endif
  release(&kmem.lock);
}
```

이 경우 커널은 해당 프레임을 모두 1로 채워 의미없는 값을 가지고 있도록 하며, 해당 프레임을 kmem 구조체의 freelist 리스트의 가장 첫 원소로 끼워넣음을 알 수 있다. 따라서 나중에라도 어떤 프로세스가 실제 메모리를 줄 것을 요구하면 해당 프레임은 그 프로세스에게 가장 먼저 주어지는 프레임이 될 것이다.

3. 마무리

위와 같이 본 프로젝트를 구현하여 프로세스 fork가 일어날 때 메모리 공간과 시간이 낭비되지 않도록 구현하였다. 본 구현에서는 어떠한 문제도 찾을 수 없었지만 만약 xv6 내부에 컴파일러를 탑재하여 다른 프로그램들을 자유로이 만들고 빌드할 수 있다면 또 다른 예측할 수 없는 문제가 생길 수 있을 것이다. 하지만 적어도 본 프로젝트에서 요구하는 경우들에 대해서는 correct한 작동을 보장하도록 구현하였다.