OS project3 - Virtual Memory & File system

2021007138 서교빈

Design

💡

Outline of implementation approach

이번 프로젝트의 구현 목표는 3가지의 소문항으로 구분된다.

1 Copy-on-Write Fork

기존 시스템에선 fork 호출 시 부모 process의 메모리를 자식 process에 그대로 복사했다. 자식 process는 대부분의 경우에 exec을 호출하므로 비효율적인 방식이며 메모리 공간 낭비도 유발된다.

Copy-on-write를 구현하면 자식 process 생성 시 pagetable에 복사가 아닌 공유하도록 하며, 공유하는 page들의 flag에 COW bit를 추가하고 Write 권한을 없앤다. 공유된 page에 write 시도가 있을 경우에 page fault를 통해 독립된 page로 관리하도록 한다.

적절한 처리를 위해 수정해야 할 주요 사항은 다음과 같다.

- **vm.c/uvmcopy()**

---

page를 복사하지 않고 PTE의 flag를 수정한다.

- **trap.c/usertrap()**

쓰기 시도 시 cow page fault를 확인하고 새 page를 할당해 mapping한다.

- **vm.c/copyout()**

---

user page에 쓰기 시도 시 trap의 cow handling을 직접 처리한다.

- **kalloc.c/kalloc(), kfree()**

---

page 생성 시 reference 수를 관리한다.

2 Large files

기존 xv6 파일 시스템은 12개의 direct block과 1개의 single indirect block로 이루어져 있다. indirect는 256개의 block address로 이루어져 있고 block 당 1KB의 data를 저장할 수 있어 파일 크기는 최대 268KB로 제한되어 있다.

여기서 direct block 1개를 double-indirect block으로 구현할 경우 최대 268 -1 + 65,536 = 65,803KB 까지 파일 크기를 확장할 수 있다. singly-indirect block과 구조가 유사하기 때문에 참조해서 구현했으며 inode 구조체의 사이즈를 유지하기 위해 data block의 개수를 13으로 고정했다.

구현 사항은 다음과 같다.

- file system size를 기존 2000에서 2000000으로 수정한다.
- NDIRECT의 수를 11로 수정한다.
- MAXFILE 계산을 (NDIRECT + NINDIRECT + NDINDIRECT)으로 수정한다.
- 기존 addrs[] 배열의 선언을 12 + 1에서 11 + 2로 수정한다.
- **fs.c/bmap()**

---

doubly-linked의 address translation를 구현한다.

- **fs.c/itrunc()**

---

doubly-linked의 삭제를 구현한다.

3 Symbolic links

기존 xv6 시스템은 inode 기반의 hard link만 지원한다. 별개의 파일이라도 같은 inode를 공유하고 있다.

symbolic link를 구현하게 되면 symlink 파일일 경우 기존 파일 구조와 다르게 고유의 inode을 갖고 데이터로 다른 파일로의 경로를 저장한다.

주요 구현 사항은 다음과 같다.

- T_SYMLINK file type, O_NOFOLLOW flag를 추가한다.
- **sys_symlink()**

---

target과 path를 인자로 받아서 inode를 만들어 symbolic link를 만든다.

- **sys_open()**

---

symlink의 recursive loop에 대한 depth와 broken links에 대한 handling을 구현한다.

Code Implementation

💡

Key code modification and purpose

1 Copy-on-Write Fork

```
//kalloc.c

struct {
  struct spinlock lock;
  struct run *freelist;
  int refcount[(PHYSTOP - KERNBASE) / PGSIZE];
} kmem;

void increfcount(void *pa) {
  if ((uint64)pa >= PHYSTOP || (uint64)pa < (uint64)end)
    panic("increase ref count");
  acquire(&kmem.lock);
  kmem.refcount[PA2INDEX(pa)]++;
  release(&kmem.lock);
}

void decrefcount(void *pa) {
   if ((uint64)pa >= PHYSTOP || (uint64)pa < (uint64)end)
    panic("decrease ref count");
  acquire(&kmem.lock);
  kmem.refcount[PA2INDEX(pa)]--;
  release(&kmem.lock);
}

int getrefcount(void *pa) {
  if ((uint64)pa >= PHYSTOP ||  (uint64)pa < (uint64)end)
    panic("get ref count");
  int count;
  acquire(&kmem.lock);
  count = kmem.refcount[PA2INDEX(pa)];
  release(&kmem.lock);
  return count;
}

void *
kalloc(void)
{
  struct run *r; //freelist

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r) {
    memset((char*)r, 5, PGSIZE); // fill with junk
    acquire(&kmem.lock);
    kmem.refcount[PA2INDEX((void*)r)] = 1; //void* casting
    release(&kmem.lock);
  }
  return (void*)r;
}
```

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

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

  acquire(&kmem.lock);
  kmem.refcount[PA2INDEX(pa)]--;
  if(kmem.refcount[PA2INDEX(pa)] > 0) { //refcount가 0보다 크면 free X
    release(&kmem.lock);
    return;
  }
  release(&kmem.lock);
  // 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;
  release(&kmem.lock);
}

//riscv.h

// pa to index
#define PA2INDEX(pa) (((uint64)(pa) - KERNBASE) / PGSIZE)
```

1. kmem 구조체에 refcount array 멤버를 추가해 page index별로 참조하는 수를 저장하도록 했다. array의 범위는 우선 KERNBASE부터 PHYSTOP까지 설정 후 riscv.h에 PA2INDEX라는 매크로를 추가해 page index를 계산하도록 했다.
2. page가 참조하는 수를 관리하기 위해 increfcount(), decrefcount(), getrefcount() 함수를 정의했다. pa의 범위를 확인하고, lock을 얻어 reference count를 조정한다.
3. kalloc()으로 page를 새로 할당할 경우 refcount array에 index를 계산해 1로 초기화한다.
4. kfree()에서는 우선 해당 index의 refcount를 1 감소시키고 reference 수가 0일 경우에만 즉, 자신만 참조 중일 경우에만 page를 해제한다.

```
//trap.c

void
usertrap(void)
{
  int which_dev = 0;
  uint64 fault_addr;
  pte_t *pte;
```

```
  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(killed(p))
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sepc, scause, and sstatus,
    // so enable only now that we're done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if (r_scause() == 15) {
    fault_addr = r_stval();
    pte = walk(p->pagetable, fault_addr, 0);

    if(!pte || !(*pte & PTE_V) || !(*pte & PTE_U) || !(*pte & PTE_COW)) {
      printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pid);
      printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
      setkilled(p);
    } else {
      uint64 pa = PTE2PA(*pte);

      //(2) 공유 중(refcnt > 1)이면 새 페이지 할당 및 복사
      if(getrefcount((void*)pa) > 1) {
        char *new_page = kalloc();
        if(!new_page) {
          setkilled(p);
        } else {
          memmove(new_page, (char*)pa, PGSIZE);
          // 새 페이지로 매핑, 쓰기 권한 추가, COW 비트 제거
          *pte = PA2PTE(new_page) | ((PTE_FLAGS(*pte) | PTE_W) & ~PTE_COW);
          decrefcount((void*)pa);
        }
      } else {
        // (3) 독점(refcnt==1)이면 PTE_W만 추가, COW 비트 제거
        *pte = (*pte | PTE_W) & ~PTE_COW;
```

```
      }
      sfence_vma(); // TLB flush
    }
  } else {
    printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pid);
    printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
    setkilled(p);
  }

  if(killed(p))
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

1. usertrap에서 COW page fault를 처리하도록 한다. 쓰기 page fault 발생 시 r_scause 값을 15를 받아 확인한다.
2. fault가 발생한 address를 받아 page에 참조 중인 수를 getrefcount로 확인하고, 만약 자신 외에 참조 중인 process가 없을 경우 flag bit만 수정한다.
3. 공유 중인 process가 있다면 새로운 page를 생성해 기존 page를 복사한다. 이후 pte의 매핑을 변경하며 flag를 수정해주고, 기존 page의 refcount를 감소시킨다.
4. TLB를 바로 flush해준다.

```
//vm.c

int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  //char *mem;

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    //if((mem = kalloc()) == 0)
    //  goto err;
    //memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, pa, (flags & ~PTE_W) | PTE_COW) != 0){
      //kfree(mem);
      goto err;
```

```
      }
      *pte = (*pte & ~PTE_W) | PTE_COW;
      increfcount((void*)pa);
    }
    return 0;

  err:
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;
}

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

    while(len > 0){
      va0 = PGROUNDDOWN(dstva);
      if(va0 >= MAXVA)
        return -1;
      pte = walk(pagetable, va0, 0);
      if(pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0)
        return -1;

      if(*pte & PTE_COW){
        uint64 pa = PTE2PA(*pte);
        if(getrefcount((void*)pa) > 1){
          np = kalloc();
          if(!np)
            return -1;
          memmove(np, (char*)pa, PGSIZE);
          *pte = PA2PTE(np) | ((PTE_FLAGS(*pte) | PTE_W) & ~PTE_COW);
          decrefcount((void*)pa);
        } else {
          //  refcount == 1
          *pte = (*pte | PTE_W) & ~PTE_COW;
        }
        sfence_vma();
      }

      // 이제 반드시 쓰기 권한이 있어야 함
      if((*pte & PTE_W) == 0)
        return -1;

      pa0 = PTE2PA(*pte);
      n = PGSIZE - (dstva - va0);
      if(n > len)
        n = len;
      memmove((void *)(pa0 + (dstva - va0)), src, n);

      len -= n;
      src += n;
```

```
      dstva = va0 + PGSIZE;
    }
    return 0;
}
```

1. uvmcopy()에서 page를 새로 할당해 복사하던 기존과 달리, 새로운 pagetable에 page를 공유하도록 한다. parent와 child 각각에 쓰기 권한을 없애며 COW bit를 추가한다. increfcount()를 통해 page의 참조 수를 증가시킨다.
2. copyout()시 COW page에 대해 독립된 page로 관리하도록 한다.
3. 만약 자신만 참조 중일 경우 flag bit만 수정한다. 참조중인 수가 1보다 클 경우 새로운 page를 할당해 usertrap과 동일한 과정을 수행하도록 한다.

2 Large files

```
//fs.h

#define NDIRECT 11 //12->11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDINDIRECT NINDIRECT * NINDIRECT
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses 11+2
};

//file.h

// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+2];
};

//param.h
```

```
#define FSSIZE       200000  // size of file system in blocks
```

1. direct block의 수를 나타내는 NDIRECT를 11로 수정한다. MAXFILE의 크기를 doubly-indirect block만큼 늘린다. 이는 NINDIRECT를 제곱하여 NDINDIRECT로 정의했다.
2. addrs 배열의 크기를 수정한 NDIRECT만큼 늘려준다.
3. FSSIZE를 200000으로 늘려 큰 파일까지 허용한다.

```
//fs.c

static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;
  int index;

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0){
      addr = balloc(ip->dev);
      if(addr == 0)
        return 0;
      ip->addrs[bn] = addr;
    }
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0){
      addr = balloc(ip->dev);
      if(addr == 0)
        return 0;
      ip->addrs[NDIRECT] = addr;
    }
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      addr = balloc(ip->dev);
      if(addr){
        a[bn] = addr;
        log_write(bp);
      }
    }
    brelse(bp);
    return addr;
  }
  bn -= NINDIRECT;

  if(bn < NDINDIRECT) {
```

```
    if((addr = ip->addrs[NDIRECT + 1]) == 0) { //13th block
      addr = balloc(ip->dev);
      if(addr == 0)
        return 0;
      ip->addrs[NDIRECT + 1] = addr;
    } //double indirect block alloc

    bp = bread(ip->dev, addr); //read buffer(first indirect)
    a = (uint*)bp->data;
    index = bn/NINDIRECT; //index of second indirect
    if((addr = a[index]) == 0) {
      addr = balloc(ip->dev);
      if(addr) {
        a[index] = addr;
        log_write(bp);
      } //first indirect block alloc
    }
    brelse(bp); //release buffer(after bread)

    bp = bread(ip->dev, addr); //read buffer(second indirect)
    a = (uint*)bp->data;
    index = bn % NINDIRECT; //index of data address
    if((addr = a[index]) == 0) {
      addr = balloc(ip->dev);
      if (addr) {
        a[index] = addr;
        log_write(bp);
      }
    } //second indirect block alloc
    brelse(bp); //after bread

    return addr;
  }

  panic("bmap: out of range");

}

void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp;
  uint *a;
  struct buf *ibp;
  uint *ia;

  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }
```

```
  if(ip->addrs[NDIRECT]){
    bp = bread(ip->dev, ip->addrs[NDIRECT]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
      if(a[j])
        bfree(ip->dev, a[j]);
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT]);
    ip->addrs[NDIRECT] = 0;
  }

  if(ip->addrs[NDIRECT + 1]) {
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++) {
      if(a[j]) { //indirect block exist
        ibp = bread(ip->dev, a[j]); //indirect block buffer read
        ia = (uint*)ibp->data;

        for (int k = 0; k < NINDIRECT; k++) {
          if(ia[k]) { //second indirect block exist
            bfree(ip->dev, ia[k]);
          }
        }
        brelse(ibp);
        bfree(ip->dev, a[j]);
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT + 1]);
    ip->addrs[NDIRECT + 1] = 0;
  }

  ip->size = 0;
  iupdate(ip);
}
```

1. bn의 범위가 NINDIRECT보다 클 경우 double-indirect block의 위치를 확인하고 block이 없다면 새로 할당한다.
2. 첫 번째로 접근하는 indirect block을 버퍼로 읽어온다. 첫 번째 indirect block에서 읽을 block의 index를 bn/NINDIRECT로 계산한다. 만약 할당 전이면 block을 할당하고 버퍼 결과를 저장한다.
3. 첫 번째 indirect block에서 읽은 주소를 버퍼로 다시 읽어온다. data의 address에 해당하는 index를 bn % NINDIRECT로 계산한다. 앞선 과정과 동일하게 비어 있다면 block을 할당하고 그 주소를 반환한다.
4. itrunc()의 경우, double-indirect block이므로 내부 block부터 free하도록 한다.
5. 첫 번째 indirect block을 돌며 block이 존재할 경우 내부 block을 읽어와 free해준다. addrs의 마지막 index의 값을 0으로 초기화한다.

3 Symbolic links

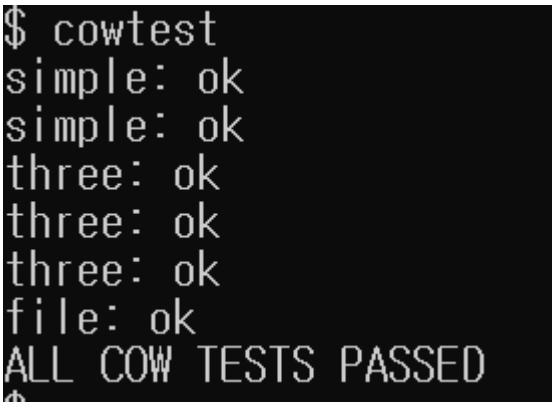
```
//fcntl.h

#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
#define O_TRUNC   0x400
#define O_NOFOLLOW 0x800

//stat.h

#define T_DIR     1   // Directory
#define T_FILE    2   // File
#define T_DEVICE  3   // Device
#define T_SYMLINK 4   // symlink

//sysfile.c

int
sys_symlink(void) {
  char target[MAXPATH], path[MAXPATH];
  struct inode *ip;
  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
    return -1;

  begin_op();
  ip = create(path, T_SYMLINK, 0, 0);
  if (ip == 0) {
    end_op();
    return -1;
  }

  //ilock(ip);
  writei(ip, 0, (uint64)target, 0, strlen(target) + 1);
  iunlockput(ip);

  end_op();
  return 0;

}

uint64
sys_open(void)
{
  char path[MAXPATH];
  int fd, omode;
  struct file *f;
  struct inode *ip;
  int n;
  int depth = 0;

  argint(1, &omode);
  if((n = argstr(0, path, MAXPATH)) < 0)
```

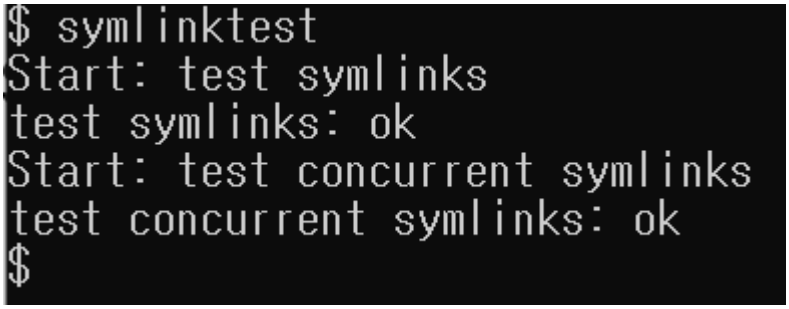
```
    return -1;

  begin_op();

  if(omode & O_CREATE){
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
  } else {
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);

    while(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)) {
      if(++depth > 10) { // 순환 참조 방지
        iunlockput(ip);
        end_op();
        return -1;
      }
      char target[MAXPATH];
      int m = readi(ip, 0, (uint64)target, 0, MAXPATH);
      iunlockput(ip);
      if(m <= 0) {
        end_op();
        return -1; // broken link
      }
      target[MAXPATH-1] = '\0'; // 안전하게 널 종료
      ip = namei(target);
      if(ip == 0){
        end_op();
        return -1; // broken link
      }
      ilock(ip);
    }
    ...
    ...
    ...
}
```

1. O_NOFOLLOW를 추가해 symbolic link 구현에 활용한다.
2. T_SYMLINK 타입을 추가해 symlink 파일을 관리하도록 한다.
3. symlink file을 create하고 target path를 저장한다.
4. symlink file일 경우 while을 돌며 target을 추적한다. depth를 1씩 증가시켜 최대 depth(==10)을 넘지 않도록 한다.
5. broken link에 대해선 예외 처리하여 -1을 반환한다.

Results

Compliation process, program flow and working code





TroubleShooting

Encountered problem, applied solutions

symlink 시스템 콜을 구현하기 위해 새 inode를 생성하고 target path를 write하는 과정에서 lock을 할당하도록 구현했었다. test 프로그램 실행 시 ilock() 도중 멈추는 현상이 확인되었고 create()에서 lock을 얻은 상태로 inode를 반환하는 것을 발견했다. lock을 중복으로 얻는 코드를 삭제해 해결했다.