



Project03

Copy-On-Write Fork

1. Design

COW는 fork를 할 때 page를 바로 복사해서 만들지 않고 공유한 채로 유지하다가 수정을 할 때 새로운 page를 만들어 복사하도록 합니다.

COW bit를 이용해 COW임을 설정해둔 후 수정을 해서 usertrap에서 걸리거나 copyout에서 COW인 경우 page를 복사하면 됩니다.

그리고 각각의 page를 참조 중인 프로세스의 개수를 확인하여야 하기에 따로 배열을 만들어 관리합니다.

2. Implementation

```
//riscv.h
#define PTE_V (1L << 0) // valid
#define PTE_R (1L << 1)
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // user can access
#define PTE_COW (1L << 8)
```

Cow bit를 이용하기 위해 PTE_COW를 설정해줍니다.

```
struct {
    struct spinlock lock;
    struct run *freelist;
    uint64 ref_count[PHYSTOP / PGSIZE];
} kmem;
```

먼저 kmem에 각 page를 공유하는 프로세스의 개수를 확인하기 위해 ref_count 배열을 만듭니다.

물리 주소의 최댓값인 PHYSTOP에서 page의 사이즈인 PGSIZE만큼 나눠서 모든 page의 개수만큼의 공간을 가지는 배열입니다.

kalloc/kfree

```
//kalloc.c
void *
kalloc(void)
{
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r) {
        kmem.freelist = r->next;
        kmem.ref_count[(uint64)r / PGSIZE] = 1;
    }
    release(&kmem.lock);

    ...
}
```

page를 새롭게 할당하게 되는 경우 그 메모리 주소의 ref_count를 1로 합니다.

```
//kalloc.c
void
kfree(void *pa)
{
    struct run *r;

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

    acquire(&kmem.lock);
    if(kmem.ref_count[(uint64)pa / PGSIZE] > 0)
        kmem.ref_count[(uint64)pa / PGSIZE]--;

    if(kmem.ref_count[(uint64)pa / PGSIZE] == 0) {
```

```

// Fill with junk to catch dangling refs.
memset(pa, 1, PGSIZE);
r = (struct run*)pa;
r->next = kmem.freelist;
kmem.freelist = r;
}
release(&kmem.lock);
}

```

page를 해제하는 경우에는 두 가지를 추가해야 합니다.

먼저 그 메모리 주소의 ref_count 값이 1 이상인 경우 1 감소 시킵니다. 그리고 ref_count 값이 0인 경우 page를 해제합니다.

만약 1 감소한 후에도 ref_count 값이 1 이상일 수 있습니다. 이 때는 여전히 참조 중인 프로세스가 있는 것이므로 해제해서는 안됩니다.

incref/decreef/getref

```

//kalloc.c
void
increef(void *pa)
{
    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("increef");

    acquire(&kmem.lock);
    kmem.ref_count[(uint64)pa / PGSIZE]++;
    release(&kmem.lock);
}

```

```

//kalloc.c
void
decreef(void *pa)
{
    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("decreef");

    acquire(&kmem.lock);
    if(kmem.ref_count[(uint64)pa / PGSIZE] > 0)

```

```

    kmem.ref_count[(uint64)pa / PGSIZE]--;
    release(&kmem.lock);
}

```

```

//kalloc.c
uint64
getref(void *pa)
{
    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("getref");

    uint64 ref = 0;
    acquire(&kmem.lock);
    ref = kmem.ref_count[(uint64)pa / PGSIZE];
    release(&kmem.lock);
    return ref;
}

```

특정 물리 주소의 ref_count 값을 증가, 감소 또는 반환하는 함수입니다.

uvmcopy

```

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

    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");

        *pte = (*pte & ~PTE_W) | PTE_COW;
        pa = PTE2PA(*pte);
    }
}

```

```

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

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

```

fork 함수에서 사용되는 함수로 부모의 page를 복사해서 자식의 page로 만들어 주는 함수입니다.

부모의 flag에 PTE_COW를 넣어주고 자식도 동일한 flag로 설정하여 매핑합니다. 기존의 코드에선 메모리를 새롭게 할당한 후 그곳에 매핑했지만 COW는 공유해야 하므로 메모리 할당을 없앴습니다.

그 후 incref를 통해 ref_count 값을 1 증가 시킵니다.

usertrap

```

//trap.c
else if(r_scause() == 15) {
    uint64 va = r_stval();
    va = PGROUNDDOWN(va);
    pte_t *pte = walk(p->pagetable, va, 0);
    if((*pte & PTE_COW) == 0) {
        p->killed = 1;
        return;
    }
    uint64 pa = PTE2PA(*pte);
    uint flags = PTE_FLAGS(*pte);

    if(getref((void*)pa) > 1) {
        char *mem;
        if((mem = kalloc()) == 0){
            panic("kalloc failed");
        }
    }
}

```

```

    }
    memmove(mem, (void*)pa, PGSIZE);
    decref((void*)pa);
    flags = (flags | PTE_W) & ~PTE_COW;
    *pte = PA2PTE(mem) | flags;
} else {
    *pte = (*pte | PTE_W) & ~PTE_COW;
}
}

```

write로 인한 page fault는 r_scause가 15입니다.

page fault가 발생하면 va를 통해 pte와 pa를 구하고 getref를 통해 ref_count 값을 확인합니다. 만약 1 초과인 경우 공유 중인 프로세스가 둘 이상이라는 것이기에 메모리를 새롭게 할당한 후 flage에서 PTE_COW를 제거합니다. 그리고 ref_count 값을 1 감소 시킵니다.

copyout

```

//vm.c
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
    uint64 n, va0, pa0;
    pte_t *pte;
    uint flags;
    char *mem;

    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 ||
           ((*pte & PTE_W) == 0 && (*pte & PTE_COW) == 0))
            return -1;
        if((*pte & PTE_W) == 0 && (*pte & PTE_COW)) {
            pa0 = PTE2PA(*pte);

            if(getref((void*)pa0) > 1) {

```

```

    if((mem = kalloc()) == 0) {
        return -1;
    }
    memmove(mem, (void *)pa0, PGSIZE);
    decref((void*)pa0);

    flags = (PTE_FLAGS(*pte) | PTE_W) & ~PTE_COW;
    *pte = PA2PTE(mem) | flags;
}
else {
    *pte = (*pte | PTE_W) & ~PTE_COW;
}
}
pa0 = walkaddr(pagetable, va0);
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;
}

```

PTE_W가 0인데 PTE_COW도 0인 경우 잘못 설정된 것이므로 return -1을 해줍니다. PTE_W가 0인데 PTE_COW가 1인 경우에는 usertrap처럼 메모리를 새로 할당해준 후 flag에서 PTE_COW를 제거하고 ref_count 값을 1 감소 시킵니다.

3. Results

```

init: starting sh
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED

```

문제 없이 잘 실행됩니다.

4. Troubleshooting

write로 인한 page fault를 구분하는 방법을 찾는데 시간이 좀 걸렸습니다.
또한 ref_count 배열을 어떻게 관리할지 생각하는 것이 힘들었습니다.

Large Files

1. Design

direct를 한 개 줄이고 대신 Doubly-Indirect를 추가합니다.
Indirect 안에 Indirect가 있는 것과 동일한 것이기에 배열 탐색을 Indirect에서 한 번 더 하도록 합니다.

2. Implementation

```
//param.h
#define NPROC      64 // maximum number of processes
#define NCPU       8  // maximum number of CPUs
#define NOFILE     16  // open files per process
#define NFILE      100 // open files per system
#define NINODE     50  // maximum number of active i-nodes
#define NDEV       10  // maximum major device number
#define ROOTDEV    1   // device number of file system root disk
#define MAXARG     32  // max exec arguments
#define MAXOPBLOCKS 10 // max # of blocks any FS op writes
#define LOGSIZE    (MAXOPBLOCKS*3) // max data blocks in on-disk log
#define NBUF       (MAXOPBLOCKS*3) // size of disk block cache
#define FSSIZE     200000 // size of file system in blocks
#define MAXPATH    128  // maximum file path name
#define USERSTACK  1    // user stack pages
```

FSSIZE를 2000에서 200000로 변경합니다.

```
//fs.h
#define NDIRECT 11
```



```
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDINDIRECT (NINDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
```

NDIRECT를 11로 변경하고 NDINDIRECT를 추가해줍니다.
MAXFILE은 NDINDIRECT도 포함하면 됩니다.

```
//file.h
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+1+1];
};

//fs.h
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+1+1]; // Data block addresses
};
```

inode와 dinode의 addrs의 사이즈를 1씩 늘려줍니다.

bmap

```

//fs.c
static uint
bmap(struct inode *ip, uint bn)
{
    uint addr, *a;
    struct buf *bp;

    ...

    if(bn < NDINDIRECT){
        if((addr = ip->addrs[NDIRECT+1]) == 0){
            addr = balloc(ip->dev);
            if(addr == 0)
                return 0;
            ip->addrs[NDIRECT+1] = addr;
        }
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;

        uint i = bn / NINDIRECT;
        uint j = bn % NINDIRECT;

        if((addr = a[i]) == 0){
            addr = balloc(ip->dev);
            if(addr){
                a[i] = addr;
                log_write(bp);
            }
        }
        brelse(bp);

        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;

        if((addr = a[j]) == 0){
            addr = balloc(ip->dev);
            if(addr){
                a[j] = addr;

```

```

        log_write(bp);
    }
}
brelse(bp);
return addr;
}

panic("bmap: out of range");
}

```

NINDIRECT의 숫자보다 더 큰 경우 NDINDIRECT를 탐색합니다.
NINDIRECT의 코드를 이용해서 중첩하여 한 번 더 탐색하도록 하였습니다.

itrunc

```

//fs.c
void
itrunc(struct inode *ip)
{
    int i, j;
    struct buf *bp, *bp2;
    uint *a, *b;

    ...

    if(ip->addrs[NDIRECT+1]){
        bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
        a = (uint*)bp->data;
        for(i = 0; i < NINDIRECT; i++){
            if(a[i]){
                bp2 = bread(ip->dev, a[i]);
                b = (uint*)bp2->data;
                for(j = 0; j < NINDIRECT; j++){
                    if(b[j])
                        bfree(ip->dev, b[j]);
                }
                brelse(bp2);
                bfree(ip->dev, a[i]);
            }
        }
    }
}

```

```

    }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT+1]);
    ip->addrs[NDIRECT+1] = 0;
}

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

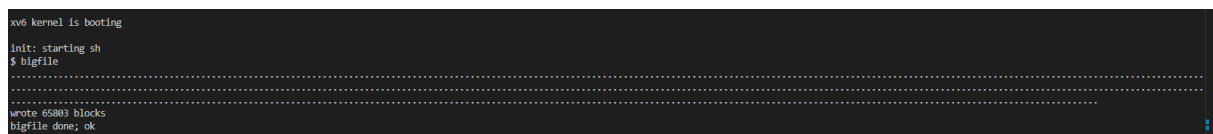
```

itrunc도 NINDIRECT의 부분을 참고하여 중첩하여 코드를 작성했습니다.
블럭을 탐색하면서 순차적으로 해제해줍니다.

create

create는 굳이 수정하지 않아도 되기에 기존 코드를 사용했습니다.

3. Results



```

xv6 kernel is booting
init: starting sh
$ bigfile
.....
wrote 65803 blocks
bigfile done; ok

```

결과는 문제 없이 잘 실행됩니다.

4. Troubleshooting

테스트 코드를 실행했더니 .이 너무 늦게 떠서 에러가 난 줄 알았습니다. 또한 bigfile done 이 뜨기까지 3시간 정도가 걸려서 구현의 문제로 멈췄다거나 에러가 발생한 줄 알았습니다.

Symbolic Links

1. Design

파일 이름을 저장하여 다른 파일을 가리키는 Symlink를 구현해야 합니다. T_SYMLINK를 만들어 일반 file과 symlink를 구분해주고 symlink인 경우 데이터를 읽어 파일 이름을 확인 하고 그 파일을 열어줍니다.

2. Implementation

```
//stat.h
#define T_DIR    1 // Directory
#define T_FILE   2 // File
#define T_DEVICE 3 // Device
#define T_SYMLINK 4 // Symbolic link
```

일반 file과 symlink를 구분하기 위해 T_SYMLINK를 추가합니다.

```
//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
```

O_NOFOLLOW도 구분하기 위해 추가합니다.

symlink

```
//sysfile.c
uint64
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();
    if ((ip = create(path, T_SYMLINK, 0, 0)) == 0) {
        end_op();
        return -1;
    }

    writei(ip, 0, (uint64)target, 0, strlen(target));
}
```

```

        iupdate(ip);
        iunlockput(ip);
        end_op();
        return 0;
    }

```

create를 이용해서 T_SYMLINK로 file을 만든 후 target 문자열을 write하여 저장합니다.

sys_open

```

//sysfile.c
uint64
sys_open(void)
{
    char path[MAXPATH], target[MAXPATH];
    int fd, omode;
    struct file *f;
    struct inode *ip;
    int n;
    int symlink_count = 0;

    ...

    else {
        while(1){
            if((ip = namei(path)) == 0){
                end_op();
                return -1;
            }
            ilock(ip);
            if(ip->type != T_SYMLINK || omode & O_NOFOLLOW){
                break;
            }

            if(symlink_count++ >= 10){
                iunlockput(ip);
                end_op();
            }
        }
    }
}

```

```

        return -1;
    }

    int len = readi(ip, 0, (uint64)target, 0, MAXPATH);
    iunlockput(ip);
    if(len <= 0){
        end_op();
        return -1;
    }

    if(len >= MAXPATH)
        len = MAXPATH - 1;
    target[len] = '\0';

    safestrcpy(path, target, MAXPATH);
}

...

}

```

while문으로 계속해서 탐색하면서 symlink를 확인합니다. 만약 확인하는 파일의 type이 T_SYMLINK가 아니거나 O_NOFOLLOW인 경우 탐색을 멈추고 파일을 읽습니다. T_SYMLINK인 경우 내용을 읽은 후 path로 설정해주고 다시 확인합니다. 이때 symlink_count가 10 이상인 경우 더 이상 확인을 하지 않고 종료합니다. 또한 cycle이 생기더라도 10번까지만 symlink를 확인하므로 무한 루프는 방지하게 됩니다.

create

```

//sysfile.c
static struct inode*
create(char *path, short type, short major, short minor)
{
    ...

    if((ip = dirlookup(dp, name, 0)) != 0){
        iunlockput(dp);
    }
}

```

```

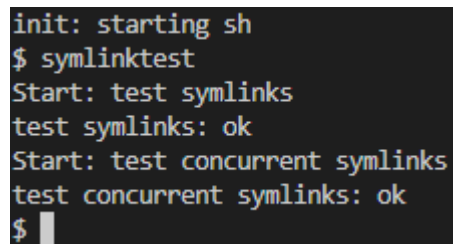
    ilock(ip);
    if(type == T_FILE && (ip->type == T_FILE || ip->type == T_DEVICE ||
        ip->type == T_SYMLINK))
        return ip;
    iunlockput(ip);
    return 0;
}

...
}

```

type을 확인할 때 T_SYMLINK도 확인하도록 합니다.

3. Results



```

init: starting sh
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$

```

결과는 문제 없이 잘 실행됩니다.

4. Troubleshooting

파일을 읽는 방식이 기존의 파일과는 아예 다르기에 구현을 어떻게 해야 할지 생각하는데 시간을 많이 썼습니다.