

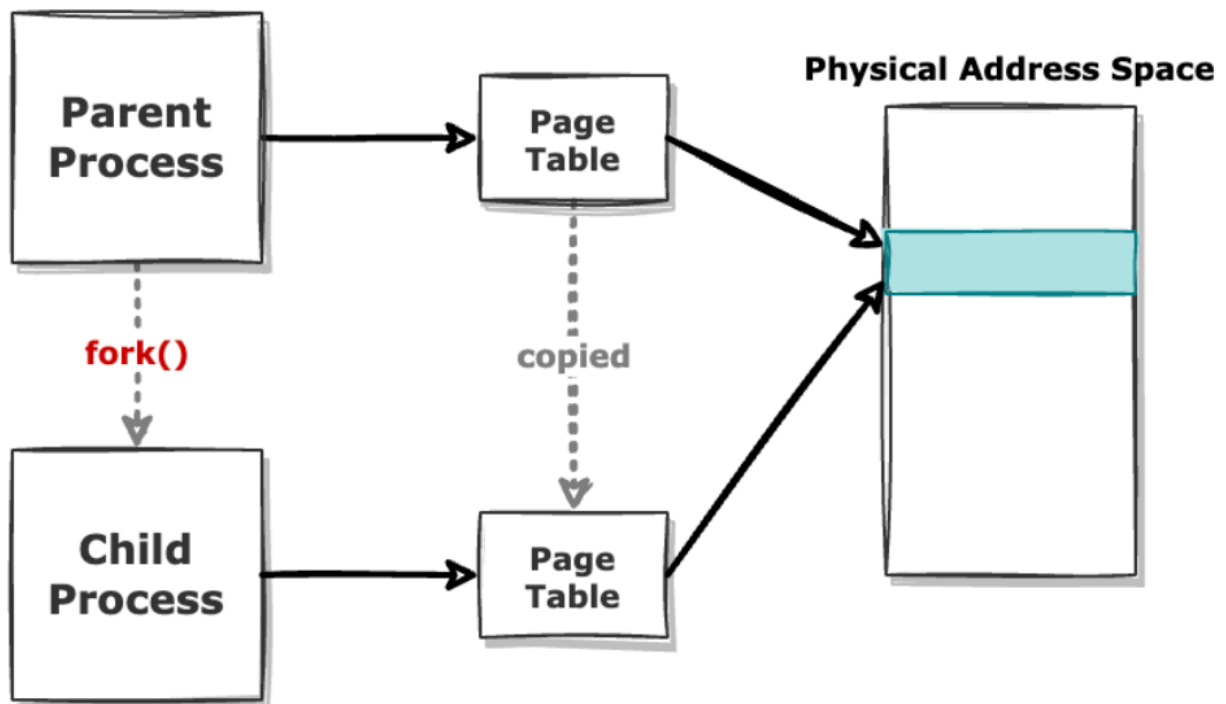
COW DESIGN

Copy on write 란?

COW는 다음 방식으로 작동한다. 일단 기본적으로 프로세스 하나만 해당 페이지를 가리키고 있는 상태는 **read/write**가 모두 가능한 상태며, 하나만 해당 페이지를 가리키고 있기 때문에 **reference count**를 1로 설정한다.

1. 만약 **fork**가 발생한다면 메모리의 특정 페이지를 가리키고 있는 부모 페이지 테이블 엔트리와 자식 페이지 테이블 엔트리를 **read only**로 표시하고 **reference count**를 1 증가시키며, **RSW**를 세팅한다. 원래부터 **read only**인지 COW 대상인지 구분을 하기 위해 **RSW**를 사용한다.
2. **read only** 페이지에 대한 **write** 요청이 들어오면 커널이 새로운 물리 페이지를 할당하고 변경을 요청한 프로세스의 페이지 테이블에 기존 매핑을 해제하고 새로운 물리 페이지를 새로 매핑해준다. 해당 페이지에 대한 **reference count**를 1 감소시킨다. 이때 만약 **reference count**가 1이 된다면 해당 물리 페이지를 가리키는 엔트리를 **read/write**모두 가능한 상태로 바꿔주고 **RSW**를 해제한다.
3. 새로 만들어진 물리 페이지는 **reference count**를 1로 세팅하고 **read/write**모두 가능한 상태로 세팅한다.

reference count가 2 이상인 물리 페이지가 있을 때 그 물리 페이지를 참조하는 프로세스의 자원을 회수할 때 로직에 수정이 필요할 것이다. **kfree**를 통해 해당 물리 페이지를 해제해버리면 이 물리 페이지를 참조하는 다른 프로세스들이 더 이상 참조할 수 없게 되버리기 때문이다. 따라서 프로세스가 종료되고 자원이 회수되는 시점에서 해당 물리 페이지의 **reference count**값이 2 이상인지 검사하고, 만약 2이상인 경우에는 페이지 테이블로부터 **uvmunmap**만 수행한다. 만약 1인 경우에만 **kfree**를 통해 자원을 해제해주자.



만약 **Read Only** 페이지에 대한 **write**가 수행된다면 **page fault**를 발생시켜 **page fault handler**를 실행시켜야된다. **SCAUSE** 레지스터는 다음과 같은 값들을 가질 수 있다. **page fault**와 관련된 코드는 아래와 같다. 우리의 관심사는 15번이다.

- 12: page fault caused by an instruction fetch
- 13: page fault caused by a read

- 15: page fault cause by a write

SCAUSE register

| Intr | Exception Code | Description |
|------|----------------|--------------------------------|
| 0 | 0 | Instruction address misaligned |
| 0 | 1 | Instruction access fault |
| 0 | 2 | Illegal instruction |
| 0 | 3 | Breakpoint |
| 0 | 4 | <i>Reserved</i> |
| 0 | 5 | Load access fault |
| 0 | 6 | AMO address misaligned |
| 0 | 7 | Store/AMO access fault |
| 0 | 8 | Environment call |
| 0 | 9-11 | <i>Reserved</i> |
| 0 | 12 | Instruction page fault |
| 0 | 13 | Load page fault |
| 0 | 14 | <i>Reserved</i> |
| 0 | 15 | Store/AMO page fault |
| 0 | >16 | <i>Reserved</i> 6 |

copy on write를 구현하기 위해 fork 함수를 확인해보자

fork

fork의 동작 방식은 다음과 같다

- `allocproc` 을 통해 프로세스를 생성
- `uvmcopy` 를 통해 부모 페이지 테이블을 자식으로 복사
- 자식의 sz 갱신, 부모의 트랩프레임 내용 복사
- fork 리턴값 0 설정
- 열린 파일들 복사
- pid 설정
- 부모 설정
- 상태 설정

```
int
fork(void)
{
    int i, pid;
```

```

struct proc *np;
struct proc *p = myproc();

// Allocate process.
if((np = allocproc()) == 0){
    return -1;
}

// Copy user memory from parent to child.
if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
}
np->sz = p->sz;

// copy saved user registers.
*(np->trapframe) = *(p->trapframe);

// Cause fork to return 0 in the child.
np->trapframe->a0 = 0;

// increment reference counts on open file descriptors.
for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
        np->ofile[i] = filedup(p->ofile[i]);
np->cwd = idup(p->cwd);

safestrcpy(np->name, p->name, sizeof(p->name));

pid = np->pid;

release(&np->lock);

acquire(&wait_lock);
np->parent = p;
release(&wait_lock);

acquire(&np->lock);
np->state = RUNNABLE;
release(&np->lock);

return pid;
}

```

우리가 수정해야될 부분은 `uvmcopy` 이다.

uvmcopy

`uvmcopy` 가 페이지 테이블을 복사하는 과정은 다음과 같다.

- **valid**한 페이지 테이블 엔트리들에 대하여 복사를 하는 **old** 페이지 테이블을 순회한다.

- `PTE2PA`를 통해 물리주소 **pa**를 구한다. `PTE_FLAGS`를 통해 **flags**를 구한다.
- `kalloc`을 통해 **mem**에 물리 프레임 할당을 담당한다.
- `memmove`를 통해 **pa**에 있는 물리 페이지를 **mem**으로 **PGSIZE**만큼 복사한다.
- `mappages`를 통해 **new**(새로운 페이지 테이블)에 **mem** 물리 프레임을 매핑해준다.

```
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, (uint64)mem, flags) != 0){
            kfree(mem);
            goto err;
        }
    }
    return 0;

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

pipe

`pipe`의 동작은 다음과 같다.

- 첫 번째 인자로 fdarray(유저 가상 주소)를 얻음
- 빈 디스크립터 슬롯에 엔드포인트 연결
- 사용자 메모리에 두 fd 값을 복사해서 돌려줌 -> 이때 **write** 발생한다.
- 성공적으로 두 개의 fd를 사용자에게 돌려줬으면 0 반환

```
uint64
sys_pipe(void)
{
    uint64 fdarray; // user pointer to array of two integers
```

```

struct file *rf, *wf;
int fd0, fd1;
struct proc *p = myproc();

argaddr(0, &fdarray);
if(pipealloc(&rf, &wf) < 0)
    return -1;
fd0 = -1;
if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
    if(fd0 >= 0)
        p->ofile[fd0] = 0;
    fileclose(rf);
    fileclose(wf);
    return -1;
}
if(copyout(p->pagetable, fdarray, (char*)&fd0, sizeof(fd0)) < 0 ||
    copyout(p->pagetable, fdarray+sizeof(fd0), (char *)&fd1, sizeof(fd1)) < 0){
    p->ofile[fd0] = 0;
    p->ofile[fd1] = 0;
    fileclose(rf);
    fileclose(wf);
    return -1;
}
return 0;
}

```

COW 없이 기본 xv6는 `fork` 시 부모의 페이지 테이블을 그대로 복사한다. 부모와 자식은 모두 동일한 물리 페이지 하나를 가리키는 PTE를 갖게되며 자식이 만약 `read(fds[0], buf, sizeof(i))` 를 호출하면, 커널의 `copyout(pagetable, buf, &i, 4)` 경로로 들어가서 커널이 가지고 있던 정수 `i`를 바로 자식의 사용자 가상 주소 `buf`에 복사해버린다. `copyout` 이 내부에서 별도의 페이지 복사를 하지 않고, 부모가 쓰던 그 물리 페이지에 곧바로 `i`를 덮어쓰게 되는 것이다. 따라서 `copyout` 은 추후에 수정되어야 한다.

COW Implementation

Copy on write를 구현하기 위해 **reference count**가 페이지별로 기록될 수 있도록 새로운 배열(**refcount**)를 만들고 **reference count**를 조정하기 위한 함수들을 정의하자. 다음 정의들은 `vm.c` 파일에 구현하였다.

KERNBASE부터 **PHYSTOP**까지의 크기를 **PGSIZE**로 나누면 물리 메모리 영역의 총 페이지 수가 된다. 이것으로 **refcount** 배열의 크기를 지정하자.

`pa2idx`의 역할은 물리 주소에 대응하는 인덱스 값을 반환해준다.

`cow_inc`의 역할은 해당 물리 주소에 대한 **refcount**를 1 증가시켜준다. `fork` 직후에 수행된다.

`cow_dec`의 역할은 해당 물리 주소에 대한 **refcount**를 1 감소시켜준다. **refcount**가 2 이상인 페이지에 대한 write가 수행되었을때, 또는 물리 페이지를 `uvmunmap`해줄 때, 또는 **refcount**값이 1인 상태에서 물리 페이지를 해제해줄 때 호출된다.

`cow_refcnt`는 현재 물리 페이지의 참조 횟수를 반환해준다.

```

#define PHYCNT ((PHYSTOP - KERNBASE) / PGSIZE)
static uint refcount[PHYCNT];

```

```

// 물리 주소 pa에 대응하는 인덱스 계산
static inline int
pa2idx(uint64 pa) {
    return (pa - KERNBASE) / PGSIZE;
}

// COW fork 직후
void
cow_inc(uint64 pa) {
    int idx = pa2idx(pa);
    refcount[idx]++;
}

// COW 복사 또는 페이지테이블 unmap 또는 프레임 해제 시
void
cow_dec(uint64 pa) {
    int idx = pa2idx(pa);
    refcount[idx]--;
}

// reference count값 반환
int
cow_refcnt(uint64 pa){
    int idx = pa2idx(pa);
    return refcount[idx];
}

```

trap.c에서의 usertrap 부분에서 **scause**값이 15인 경우에 `cow_pagefault`를 발생시킨다.

```

void
usertrap(void)
{
    ...
    if(r_scause() == 8){
        ...
    } else if(r_scause() == 15){
        if(cow_pagefault(p->pagetable, r_stval()) != 0){
            panic("cow fault failed!");
        }
    } else if((which_dev = devintr()) != 0){
        // ok
    } 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);
    }
    ...

    usertrapret();
}

```

vm.c에 정의된 `cow_pagefault`는 다음과 같다.

본기문에서 `refcount[idx]`가 1인 상황은 다음과 같은 상황에서 발생한다.

- `fork`가 한 번 이상 처리된 페이지에 대해서는 **Read Only**로 바뀌며, 이후에 **write**가 지속적으로 발생한 후 해당 물리 프레임 참조하는 프로세스가 하나밖에 안 남은 경우 **write**가 발생한다면 저 분기문으로 들어간다.

`refcount[idx]`가 2이상인 상황은 `fork`가 이미 여러 번 실행되고 해당 페이지를 참조하는 프로세스가 여러 개 존재하는 상황에서 **write**가 실행되는 경우이다. 이 상황에서는 새로운 프레임을 할당한다. `kalloc`이 실행됨에 따라 해당 프레임에 대한 `refcount`는 1로 설정된다. 원래 프레임의 `refcount`는 1감소시켜준다. 페이지 테이블의 매핑 정보를 해제하고 `kalloc`을 통해 새로 만든 물리 페이지로 다시 매핑해준다. 이때 **write**를 세팅해주고 **RSW**를 해제해준다.

```
// 성공 시 0, 실패 시 -1 반환
int
cow_pagefault(pagetable_t pagetable, uint64 va)
{
    // va를 페이지 경계로 정렬
    va = PGROUNDDOWN(va);
    pte_t *pte = walk(pagetable, va, 0);
    if(!pte || !(*pte & PTE_V) || (*pte & PTE_W)){
        return -1; // 유효하지 않거나 이미 쓰기 가능
    }

    uint64 pa = PTE2PA(*pte);
    int idx = pa2idx(pa);

    if(refcount[idx] == 1){
        // 유일 참조
        *pte |= PTE_W;
        *pte &= ~PTE_RSW; // COW 예약 비트 해제
    } else {
        // 다중 참조
        char *newpa = kalloc(); // kalloc을 하는 순간 refcnt가 1 증가됨
        if(newpa == 0)
            panic("kalloc");

        memmove((void*)newpa, (void*)pa, PGSIZE); // 물리 프레임 복사
        cow_dec(pa); // 기존 프레임 refcount--

        uint64 old_flags = PTE_FLAGS(*pte);
        uvmunmap(pagetable, va, 1, 0);
        uint64 new_flags = old_flags;
        new_flags |= PTE_W; // 쓰기 허가 켜기
        new_flags &= ~PTE_RSW; // COW 예약 비트 끄기

        if (mappages(pagetable, va, PGSIZE, (uint64)newpa, new_flags) != 0) {
            panic("cow_pagefault: mappages failed\n");
        }
    }

    sfence_vma(); // TLB 플러시
    return 0;
}
```

```
}
```

sfence_vma

```
// flush the TLB.
static inline void
sfence_vma()
{
    // the zero, zero means flush all TLB entries.
    asm volatile("sfence.vma zero, zero");
}
```

`sfence.vma zero, zero`는 페이지 테이블을 수정한 뒤 CPU 내부의 TLB를 강제 동기화하기 위해 꼭 필요하다.

메모리 복사

uvmcopy

`fork`가 수행되면서 `uvmcopy`가 실행되고 원래 페이지 테이블에 대한 **entry**가 **Read Only**로 설정되고 새로운 페이지 테이블에 대한 **entry**도 **Read Only**로 설정된다. 이때 `cow_inc`를 통해 **reference count**를 1 증가시킨다.

```
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_W); // 해당 엔트리를 read-only로 만들
        *pte |= PTE_RSW; // COW 예약 비트를 켜 준 뒤

        pa = PTE2PA(*pte); // 엔트리로부터 물리 주소 추출
        flags = PTE_FLAGS(*pte); // flag 비트들만 추출

        if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){ // 새로운 페이지 테이블에도 해당 엔트리는
read only로 기록됨
            goto err;
        }

        cow_inc(pa); // reference count 1 증가
    }

    sfence_vma();
    return 0;
}
```



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

메모리 할당

kalloc

새로운 물리 페이지를 할당할 때 **refcnt**를 1로 설정한다.

```
void *
kalloc(void)
{
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r){
        kmem.freelist = r->next;
        cow_int(r); // 새로운 프로세스 생성 시 refcnt 1 증가
    }
    release(&kmem.lock);

    if(r)
        memset((char*)r, 5, PGSIZE); // fill with junk
    return (void*)r;
}
```

메모리 해제

kfree

물리 페이지를 해제할 때 **refcnt**를 감소시키고 만약 해당 물리 페이지에 대한 **refcnt**가 0이어서 해당 물리 페이지를 완전히 해제해야되는 경우에 원래 기본 **kfree** 로직을 수행한다.

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

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

    if(cow_refcnt(pa) > 0){
        cow_dec(pa); // 해당 물리 페이지를 참조하는 프로세스 수 하나 줄이기
    }

    if(cow_refcnt(pa) == 0){ // 해당 물리 페이지를 참조하는 프로세스가 존재하지 않는다면
        // 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);
}
}

```

유저 메모리에 쓰기

copyout

`pipe` 를 통해 `copyout` 하는 로직을 `pagefault`에서 했던 로직과 비슷하게 구현하면 된다. `copyout()` 함수가 바로 사용자 주소(`dstva`)에 데이터를 쓰는 순간 COW 복사를 수행하도록 구현하였다.

```

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

    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) // fork되면 PTE_W가 꺼져있을 수
        있으므로 이 부분 수정
            return -1;

        uint64 pa = PTE2PA(*pte);
        int idx = pa2idx(pa);
        if(refcount[idx] == 1){
            // 유일 참조
            *pte |= PTE_W;
            *pte &= ~PTE_RSW; // COW 예약 비트 해제
        }else{
            // 다중 참조
            char *newpa = kalloc();
            if(newpa == 0)
                panic("kalloc");
            memmove(newpa, (void*)pa, PGSIZE);
            cow_dec(pa);

            uint64 old_flags = PTE_FLAGS(*pte);
            uvmunmap(pagetable, va0, 1, 0);
            uint64 new_flags = old_flags;
            new_flags |= PTE_W; // 쓰기 허가 켜기
            new_flags &= ~PTE_RSW; // COW 예약 비트 끄기

```

```

    if (mappages(pagetable, va0, PGSIZE, (uint64)newpa, new_flags) != 0) {
        panic("cow_pagefault: mappages failed\n");
    }

    // TLB 동기화
    sfence_vma();
    // 다시 PTE를 가져와서 아래 쓰기 단계로 넘어감
    pte = walk(pagetable, va0, 0);
}

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;
}

```

COW Result

테스트에서 확인하고자 하는 부분들을 살펴보자.

- `simpletest` 를 한 번 호출한다.
- `threetest` 를 세 번 호출한다.
- `filetest` 를 한 번 호출한다.

각각이 어떤 내용을 담고 있는지 확인해보자.

```

int
main(int argc, char *argv[])
{
    simpletest();

    // check that the first simpletest() freed the physical memory.
    simpletest();

    threetest();
    threetest();
    threetest();

    filetest();

    printf("ALL COW TESTS PASSED\n");
}

```

```

    exit(0);
}

```

simpletest

이 함수는 시스템 물리 메모리의 절반 이상을 한 프로세스가 할당한 뒤, `fork` 를 호출하면 어떻게 되는지를 확인하기 위한 함수이다. 동작은 다음과 같다.

- `phys_size = PHYSTOP - KERNBASE` 는 사용 가능한 물리 메모리 총량을 의미한다.
- `int sz = (phys_size / 3) * 2;` 전체 물리 메모리의 2/3 만큼을 `sz` 변수에 저장한다.
- `char *p = sbrk(sz);` 를 통해 현재 프로세스의 브레이크 주소를 `sz` 만큼 늘려준다.
- `for(char *q = p; q < p + sz; q += 4096)` 를 통해 한 번도 접근하지 않았던 페이지마다 실제로 쓰기를 실행하여 페이지를 점유하도록 한다. `sbrk` 를 통해 물리 페이지가 바로 할당되는 것이 아니기 때문이다. 각 페이지마다 현재 프로세스 ID를 기록한다.
- `fork` 가 수행되면 자식에게도 똑같이 2/3 만큼 페이지를 줘야 하기 때문에 토탈 4/3 * 전체 메모리에 해당하는 물리 페이지가 할당 되어야 한다. COW(Copy-On-Write)가 제대로 구현되어 있다면 자식 프로세스는 메모리를 실제로 복사하지 않고 부모 페이지를 그대로 공유하게 되므로 `fork` 가 실패하지 않는다.
- `sbrk(-sz)` 를 통해 할당했던 메모리를 되돌린다.

```

// allocate more than half of physical memory,
// then fork. this will fail in the default
// kernel, which does not support copy-on-write.
void
simpletest()
{
    uint64 phys_size = PHYSTOP - KERNBASE;
    int sz = (phys_size / 3) * 2;

    printf("simple: ");

    char *p = sbrk(sz);
    if(p == (char*)0xffffffffffffL){
        printf("sbrk(%d) failed\n", sz);
        exit(-1);
    }

    for(char *q = p; q < p + sz; q += 4096){
        *(int*)q = getpid();
    }

    int pid = fork();
    if(pid < 0){
        printf("fork() failed\n");
        exit(-1);
    }

    if(pid == 0)
        exit(0);
}

```

```

wait(0);

if(sbrk(-sz) == (char*)0xffffffffffffffffL){
    printf("sbrk(-%d) failed\n", sz);
    exit(-1);
}

printf("ok\n");
}

```

threetest

이 함수는 세 단계(Fork)를 거쳐 세 프로세스가 모두 COW 메모리를 쓰도록 하고, 그 결과로 물리 메모리의 절반 이상이 안전하게 할당, 해제되는지를 확인한다. 동작 로직은 다음과 같다.

- `phys_size = PHYSTOP - KERNBASE` 는 사용 가능한 물리 메모리 총량을 의미한다.
- `sz = phys_size / 4`; 이며 `sbrk(sz)` 를 통해 현재 프로세스의 브레이크 주소를 **sz** 만큼 늘려준다
- `fork` 를 통한 자식1 -> 부모의 주소 공간을 COW로 공유한 뒤 1/4 sz의 1/2만큼 (= 전체 sz의 1/8) 페이지를 덮어써서 실제 할당 발생
- `fork` 를 통한 자식2 -> 자식1의 주소 공간(= 부모와 공유하던 페이지)을 COW로 공유한 뒤 1/4 sz의 4/5만큼 (= 전체 sz의 1/5) 페이지를 덮어써서 실제 할당 발생
- 부모 프로세스는 전체의 1/4만큼을 할당받게 됨. -> $1/4 + 1/8 + 1/5 \leq 1$ 이므로 정상적으로 할당된다. COW가 제대로 구현되어 있지 않다면 `fork` 를 수행할 때마다 **부모 복사본 + 자기 쓰기** 과정을 통해 메모리가 엄청 많이 할당되어 실패하게 된다.
- 최종적으로 `sbrk(-sz)` 를 통해 할당했던 메모리를 되돌린다.

```

// three processes all write COW memory.
// this causes more than half of physical memory
// to be allocated, so it also checks whether
// copied pages are freed.
void
threetest()
{
    uint64 phys_size = PHYSTOP - KERNBASE;
    int sz = phys_size / 4;
    int pid1, pid2;

    printf("three: ");

    char *p = sbrk(sz);
    if(p == (char*)0xffffffffffffffffL){
        printf("sbrk(%d) failed\n", sz);
        exit(-1);
    }

    pid1 = fork();
    if(pid1 < 0){
        printf("fork failed\n");
    }

```

```

    exit(-1);
}
if(pid1 == 0){
    pid2 = fork();
    if(pid2 < 0){
        printf("fork failed");
        exit(-1);
    }
    if(pid2 == 0){
        for(char *q = p; q < p + (sz/5)*4; q += 4096){
            *(int*)q = getpid();
        }
        for(char *q = p; q < p + (sz/5)*4; q += 4096){
            if(*(int*)q != getpid()){
                printf("wrong content\n");
                exit(-1);
            }
        }
        exit(-1);
    }
    for(char *q = p; q < p + (sz/2); q += 4096){
        *(int*)q = 9999;
    }
    exit(0);
}

for(char *q = p; q < p + sz; q += 4096){
    *(int*)q = getpid();
}

wait(0);

sleep(1);

for(char *q = p; q < p + sz; q += 4096){
    if(*(int*)q != getpid()){
        printf("wrong content\n");
        exit(-1);
    }
}

if(sbrk(-sz) == (char*)0xffffffffffffffffL){
    printf("sbrk(-%d) failed\n", sz);
    exit(-1);
}

printf("ok\n");
}

```

filetest

이 함수는 부모가 공유하는 **buf[]** 배열을 자식 프로세스들이 읽고 쓰는 동안, 자식이 부모의 메모리를 덮어쓰지 못하게 잘 보호되는 지를 확인하는 함수이다. 동작은 다음과 같다.

- **buf[0] = 99;**를 통해 첫 바이트를 99로 설정한다.
- 매 반복마다 `pipe(fds)` 를 호출하여 **fds[0]**(읽기 끝)과 **fds[1]**(쓰기 끝)을 초기화해준다.
- `fork` 후에 자식은 자신을 1틱 동안 잠재워서, 부모가 `write` 를 할 시간을 준다. 부모가 `write(fds[1], &i, sizeof(i))` 를 수행하면, 파이프에 **i** 의 4바이트가 들어간다. `read(fds[0], buf, sizeof(i))` 를 통해 자식은 파이프의 읽기 끝(**fds[0]**)에서 4바이트를 **buf** 배열의 앞 **buf[0..3]** 영역에 복사해 온다.
- 모든 루프가 끝나면 자식들을 수거한 뒤 부모의 **buf[0]** 값이 99로 잘 유지되고 있는지 확인한다.

`fork` 이후 부모/자식 페이지 테이블이 동일 물리 주소를 읽기 전용으로 가리키도록 PTE만 복사한다. 만약 COW가 없다면, `read` 가 유저 버퍼 **buf** 에 쓰기를 시도하면서 공유 중이던 부모 페이지(커널상 공유됨)를 곧바로 덮어쓰게 되어버린다.

```
// test whether copyout() simulates COW faults.
void
filetest()
{
    printf("file: ");

    buf[0] = 99;

    for(int i = 0; i < 4; i++){
        if(pipe(fds) != 0){
            printf("pipe() failed\n");
            exit(-1);
        }
        int pid = fork();
        if(pid < 0){
            printf("fork failed\n");
            exit(-1);
        }
        if(pid == 0){
            sleep(1);
            if(read(fds[0], buf, sizeof(i)) != sizeof(i)){
                printf("error: read failed\n");
                exit(1);
            }
            sleep(1);
            int j = *(int*)buf;
            if(j != i){
                printf("error: read the wrong value\n");
                exit(1);
            }
            exit(0);
        }
        if(write(fds[1], &i, sizeof(i)) != sizeof(i)){
            printf("error: write failed\n");
            exit(-1);
        }
    }
}
```

```

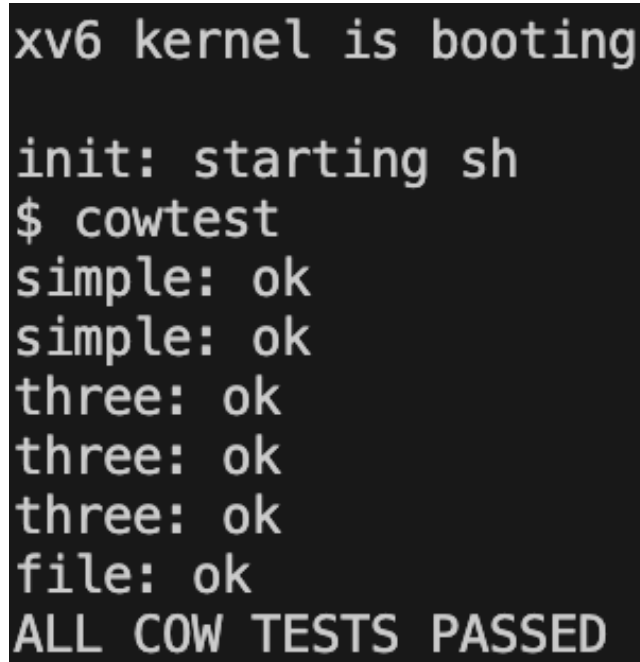
int xstatus = 0;
for(int i = 0; i < 4; i++) {
    wait(&xstatus);
    if(xstatus != 0) {
        exit(1);
    }
}

if(buf[0] != 99){
    printf("error: child overwrote parent\n");
    exit(1);
}

printf("ok\n");
}

```

모두 정상적으로 잘 실행된다.



```

xv6 kernel is booting

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

```

Large Files Design

write 시스템콜

`write` 시스템 콜의 동작 방식은 다음과 같다.

sys_write

- **struct file *f;** 는 쓰려고 하는 파일의 파일 포인터
int n; 는 쓰려는 바이트 수
uint64 p; 는 사용자 공간의 버퍼에 대한 포인터이다.
- `filewrite(f, p, n)` 를 호출한다.


```

uint64
sys_write(void)
{
    struct file *f;
    int n;
    uint64 p;

    argaddr(1, &p);
    argint(2, &n);
    if(argfd(0, 0, &f) < 0)
        return -1;

    return filewrite(f, p, n);
}

```

filewrite

- **write**대상이 **PIPE**인지 **DEVICE**인지 **INODE**인지에 따라 분기한다.
- **PIPE**의 경우 `pipewrite(f->pipe, addr, n)` 를 수행한다.
- **DEVICE**의 경우 **f->major** 디바이스 드라이버의 고유 식별자 번호의 유효성을 확인하고 `devsw[f->major].write` 디바이스 테이블을 확인하여 해당 디바이스가 쓰기를 지원하는지 검증한다. `devsw[f->major].write(1, addr, n)` 를 통해 유저 버퍼(addr)→장치로 데이터를 쓴다.
- **INODE**의 경우는 **n - i** 남은 바이트를 확인해가며 반복문을 수행한다. 있다면 `writei(f->ip, 1, addr + i, f->off, n1)` 가 호출되며 디스크 상 **inode**에 **n1** 바이트를 쓴다. 마지막에 `ret = (i == n ? n : -1);` 검사는 **i==n** 인 경우 모든 요청 바이트(n)를 정상 처리했다는 것이고, 아닌 경우 -1을 반환한다.

```

// Write to file f.
// addr is a user virtual address.
int
filewrite(struct file *f, uint64 addr, int n)
{
    int r, ret = 0;

    if(f->writable == 0)
        return -1;

    if(f->type == FD_PIPE){
        ret = pipewrite(f->pipe, addr, n);
    } else if(f->type == FD_DEVICE){
        if(f->major < 0 || f->major >= NDEV || !devsw[f->major].write)
            return -1;
        ret = devsw[f->major].write(1, addr, n);
    } else if(f->type == FD_INODE){
        // write a few blocks at a time to avoid exceeding
        // the maximum log transaction size, including
        // i-node, indirect block, allocation blocks,
        // and 2 blocks of slop for non-aligned writes.
        // this really belongs lower down, since writei()
        // might be writing a device like the console.
        int max = ((MAXOPBLOCKS-1-1-2) / 2) * BSIZE;

```

```

int i = 0;
while(i < n){
    int n1 = n - i;
    if(n1 > max)
        n1 = max;

    begin_op();
    ilock(f->ip);
    if ((r = writei(f->ip, 1, addr + i, f->off, n1)) > 0)
        f->off += r;
    iunlock(f->ip);
    end_op();

    if(r != n1){
        // error from writei
        break;
    }
    i += r;
}
ret = (i == n ? n : -1);
} else {
    panic("filewrite");
}

return ret;
}

```

우리가 구현하고자 하는 부분은 일반 파일이 대상이기 때문에 **INODE**로 분기하는 부분을 살펴봐야한다. 구체적으로 `writei`가 동작하는 방법을 확인해보자.

`writei`의 동작 원리는 다음과 같다.

- 오프셋이 파일 크기를 넘거나 오버플로우 발생 시 오류를 처리해준다.
- 루프를 순회하며, `bmap`을 통해 블록단위로 주소를 가져오고 `bread`를 통해 읽어온 블록은 **bp** 버퍼에 담는다.
- 블록에 쓸 수 있는 최대 바이트 수 계산하고 `either_copyin`을 통해 값을 기록한다.
- `log_write`를 통해 디스크에 반영하고 `brelease`를 통해 버퍼를 해제한다.
- 파일 크기 및 **inode**를 갱신한다.
- 실제로 쓴 바이트 수 반환한다.

```

int
writei(struct inode *ip, int user_src, uint64 src, uint off, uint n)
{
    uint tot, m;
    struct buf *bp;

    if(off > ip->size || off + n < off)
        return -1;
    if(off + n > MAXFILE*BSIZE)
        return -1;

```

```

for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    uint addr = bmap(ip, off/BSIZE); // off/BSIZE는 디스크 블록 번호
    if(addr == 0)
        break;
    bp = bread(ip->dev, addr);
    m = min(n - tot, BSIZE - off%BSIZE);
    if(either_copyin(bp->data + (off % BSIZE), user_src, src, m) == -1) {
        brelse(bp);
        break;
    }
    log_write(bp);
    brelse(bp);
}

if(off > ip->size)
    ip->size = off;

// write the i-node back to disk even if the size didn't change
// because the loop above might have called bmap() and added a new
// block to ip->addrs[].
iupdate(ip);

return tot;
}

```

우리의 `write` 대상 주소는 `bmap`을 통해 가지고 올 수 있는데, 구체적으로 어떻게 되는지 확인해보자.

bmap

inode 포인터 `ip`가 가리키는 파일의 `bn` 번째 블록에 대응하는 디스크 블록 주소를 가지고 온다.

- $bn < \text{NDIRECT}$ 는 **Direct** 블록 영역을 가리킨다.
- $\text{NDIRECT} \leq bn < \text{NDIRECT} + \text{NINDIRECT}$ 는 **single indirect block**영역을 가리킨다.



Direct 블록 영역의 경우 `ip->addrs[bn]` 을 통해 바로 디스크의 블록 주소에 접근한 뒤 해당 영역이 할당되어 있지 않다면 `balloc(ip->dev)` 을 통해 블록 영역을 할당한다.

single indirect block 영역의 경우 만약 **indirect block**이 없으면 `balloc` 을 통해 할당한다, 있다면 넘어간다. `bread` 를 통해 해당 **indirect block**을 **Buffer Cache**(간접 블록의 온-디스크 데이터를 메모리로 옮겨 담은 버퍼)로 읽어온다. `addr = a[bn]`을 통해 실제 디스크의 블록 단위 주소를 얻는다. 이때 해당 영역이 할당되어 있지 않다면 `balloc` 을 통해 할당해준다.

우리가 과제에서 구현해야하는 것이 `bmap` 함수 내부이다. 우리는 코드를 추가하고 기본적으로 정의된 필드들을 수정하여 **doubly indirect block**을 구현해야한다.

```
// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
// returns 0 if out of disk space.
static uint
bmap(struct inode *ip, uint bn)
{
    uint addr, *a;
    struct buf *bp;

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

```

`either_copyin`을 통해 어떻게 쓰기를 수행하는지 살펴보자.

동작 과정은 아래와 같다.

either_copyin

위에 `writel`에서 코드는 `either_copyin(bp->data + (off % BSIZE), user_src, src, m) == -1` 이렇게 실행되었었다. 전체적인 동작을 요약하면, 파일의 특정 바이트(off) 위치부터 블록 단위로 끊어가며, 사용자/커널 버퍼(src)에서 데이터를 꺼내와 해당 디스크 블록 버퍼(bp->data)에 덮어쓰는 로직이다.

```

// Copy from either a user address, or kernel address,
// depending on usr_src.
// Returns 0 on success, -1 on error.
int
either_copyin(void *dst, int user_src, uint64 src, uint64 len)
{
    struct proc *p = myproc();
    if(user_src){
        return copyin(p->pagetable, dst, src, len);
    } else {
        memmove(dst, (char*)src, len);
        return 0;
    }
}

```

read 시스템콜

`read` 시스템 콜의 동작 방식은 다음과 같다.

sys_read

- **struct file *f;** 는 읽으려고 하는 파일의 파일 포인터
int n; 는 읽으려는 바이트 수
uint64 p; 는 사용자 공간의 버퍼에 대한 포인터이다.
- `fileread(f, p, n)` 를 호출한다.

```
uint64
sys_read(void)
{
    struct file *f;
    int n;
    uint64 p;

    argaddr(1, &p);
    argint(2, &n);
    if(argfd(0, 0, &f) < 0)
        return -1;
    return fileread(f, p, n);
}
```

fileread

- **read**대상이 **PIPE**인지 **DEVICE**인지 **INODE**인지에 따라 분기한다.
- **PIPE**의 경우 `piperead(f->pipe, addr, n)` 를 수행한다.
- **DEVICE**의 경우 **f->major** 디바이스 드라이버의 고유 식별자 번호의 유효성을 확인하고 `devsw[f->major].write` 디바이스 테이블을 확인하여 해당 디바이스가 읽기를 지원하는지 검증한다. `devsw[f->major].read(1, addr, n)` 를 통해 장치 → addr(유저 버퍼)로 읽는다.
- **INODE**의 경우는 `readi(f->ip, 1, addr, f->off, n)` 를 통해 값을 **addr**(유저 버퍼)로 읽어온다.

```
// Read from file f.
// addr is a user virtual address.
int
fileread(struct file *f, uint64 addr, int n)
{
    int r = 0;

    if(f->readable == 0)
        return -1;

    if(f->type == FD_PIPE){
        r = piperead(f->pipe, addr, n);
    } else if(f->type == FD_DEVICE){
        if(f->major < 0 || f->major >= NDEV || !devsw[f->major].read)
            return -1;
    }
```

```

    r = devsw[f->major].read(1, addr, n);
} else if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, 1, addr, f->off, n)) > 0)
        f->off += r;
    iunlock(f->ip);
} else {
    panic("fileread");
}

return r;
}

```

`readi`의 동작 원리는 다음과 같다.

- 읽기 시작하는 오프셋이 파일 크기를 넘는 경우와 오버플로우에 관한 처리를 해준다.
- 루프를 순회하며, `bmap`을 통해 블록단위로 주소를 가져오고 `bread`를 통해 읽어온 블록은 `bp` 버퍼에 담는다.
- 블록에서 읽을 수 있는 최대 바이트 수를 계산하고 `either_copyout`을 통해 값을 복사한다.
- `brelse`를 통해 버퍼를 해제해준다.
- 실제로 읽은 바이트 수를 반환한다.

```

// Read data from inode.
// Caller must hold ip->lock.
// If user_dst==1, then dst is a user virtual address;
// otherwise, dst is a kernel address.
int
readi(struct inode *ip, int user_dst, uint64 dst, uint off, uint n)
{
    uint tot, m;
    struct buf *bp;

    if(off > ip->size || off + n < off)
        return 0;
    if(off + n > ip->size)
        n = ip->size - off;

    for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
        uint addr = bmap(ip, off/BSIZE); // off/BSIZE는 디스크 블록 번호
        if(addr == 0)
            break;
        bp = bread(ip->dev, addr);
        m = min(n - tot, BSIZE - off%BSIZE);
        if(either_copyout(user_dst, dst, bp->data + (off % BSIZE), m) == -1) {
            brelse(bp);
            tot = -1;
            break;
        }
        brelse(bp);
    }
    return tot;
}

```

```
}
```

Large Files Implementation

`params.h`에 있는 **FSSIZE**(파일 시스템의 크기)를 2000에서 200000으로 바꿔준다.

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

`fs.h`에 있는 **NDIRECT**를 12에서 11로 바꿔준다.

```
#define NDIRECT 11
```

`fs.h`에 **NDOUBLY_INDIRECT**를 **NINDIRECT*NINDIRECT**로 정의하고 **MAXFILE**을 (**NDIRECT + NINDIRECT + NDOUBLY_INDIRECT**)로 정의한다.

```
#define NDOUBLY_INDIRECT NINDIRECT*NINDIRECT
#define MAXFILE (NDIRECT + NINDIRECT + NDOUBLY_INDIRECT)
```

`file.h`에 있는 **inode**구조체의 **addrs[NDIRECT+1]**를 **addrs[NDIRECT+2]**로 수정해주자

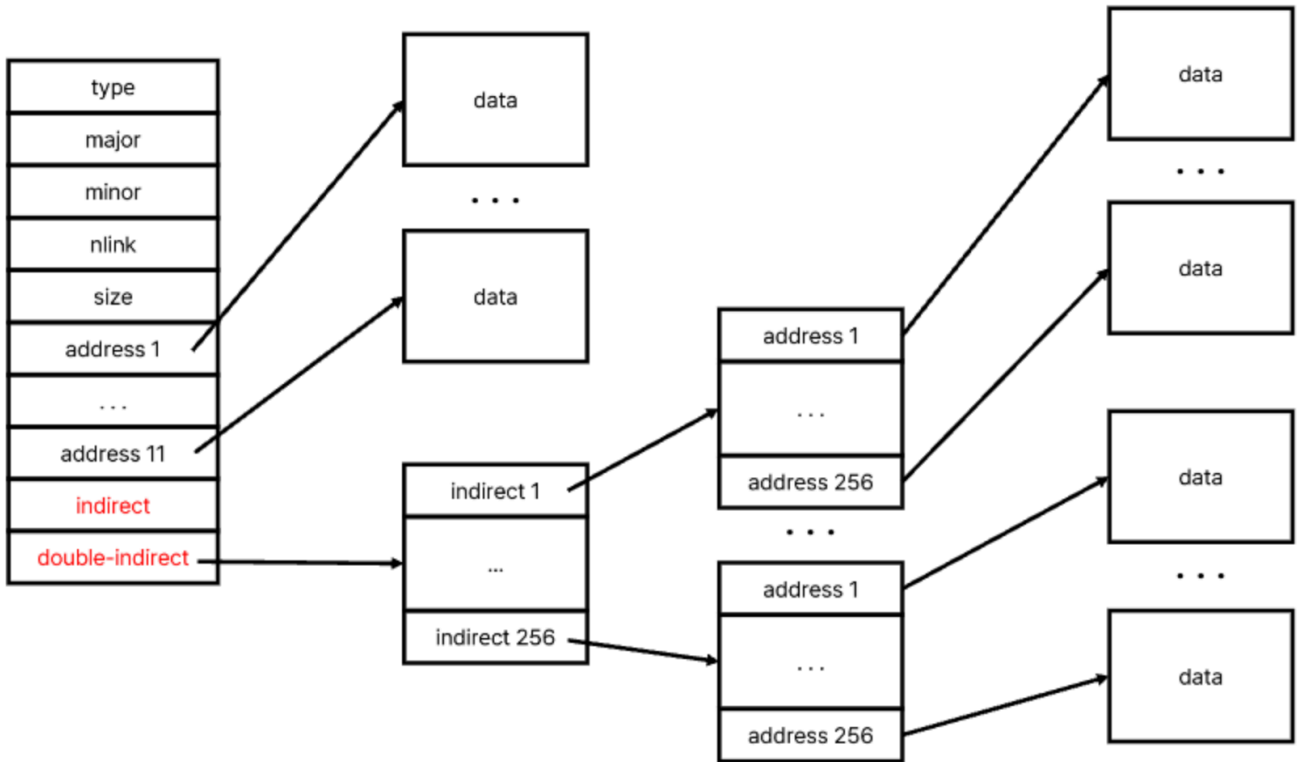
```
struct inode {
    uint dev;           // Device number
    uint inum;          // Inode number
    int ref;            // Reference count
    struct sleeplock lock; // protects everything below here
    int valid;          // inode has been read from disk?

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

`fs.h`에 있는 **dinode**구조체의 **addrs[NDIRECT+1]**를 **addrs[NDIRECT+2]**로 수정해주자

```
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
};
```

현재까지 수정한 필드들은 아래의 구조를 만족시키기 위해서 수정한 것이다.



다음은 수정된 `bmap`의 동작 로직이다. 구현된 **doubly-indirect** 블록 부분만 설명할 것이다.

- `addrs[NDIRECT+1]` 는 이중 간접 블록(double-indirect block)의 디스크 블록 번호를 가리킨다. 만약 할당되어 있지 않다면 `ballocc`을 통해 할당해준다.
- `bp = bread(ip->dev, addr)` 를 통해 이중 간접 블록 번호로부터 **버퍼 캐시** 상에 해당 블록을 읽어온다. `(uint*)bp->data` 를 `a` 로 캐스팅하여, 첫 번째 레벨 간접 블록들을 가리키는 배열처럼 활용하자. 마찬가지로 `a[double_index]`이 할당되어 있지 않다면 `ballocc`을 통해 할당한다.
- 여기서 **double_index**를 구할 때 다음과 같은 방식으로 계산한다 **double_index = bn / NINDIRECT**.

`bn` 은 “이중 간접 영역” 내에서의 논리 블록 인덱스($0 \leq bn < \text{NDOUBLY_INDIRECT}$)이다.

예를 들어 `NINDIRECT = 128` 이라면,

`bn = 0 ~ 127` 일 때 `double_index = 0`

`bn = 128 ~ 255` 일 때 `double_index = 1`

... 이렇게 $0 \leq \text{double_index} < \text{NINDIRECT}$ 범위를 가진다.

즉 두 번째 레벨 블록인 `a[]` 배열의 몇 번째 슬롯(인덱스)에 접근해야 하는지를 구해야 하기 때문에 이렇게 계산된다.

- `bp = bread(ip->dev, addr)` 에서 **addr**는 첫 번째 레벨 블록의 디스크 블록 번호이다. 마찬가지로의 방법이 사용되었으며 `a[pos]`는 최종 디스크 블록의 주소가 된다. 할당되어 있지 않다면 `ballocc`으로 할당해준다.

```
static uint
bmap(struct inode *ip, uint bn) // 파일의 논리적 블록 번호bn을 실제 디스크 블록 번호로 매핑
{
    uint addr, *a;
    struct buf *bp;

    if (bn >= MAXFILE)
        panic("bmap: out of range");
```

```

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;

// NDIRECT <= bn < NDIRECT + NINDIRECT
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;

// NINDIRECT <= bn < NINDIRECT + NDOUBLY_INDIRECT
if(bn < NDOUBLY_INDIRECT){
    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;

    // 두 번째 레이어
    uint double_index = bn / NINDIRECT;
    if((addr = a[double_index]) == 0){
        addr = balloc(ip->dev);
        if(addr == 0)
            return 0;
        a[double_index] = addr;
    }
}

```

```

    log_write(bp);
}
brelse(bp);

// 디스크 블록
bp = bread(ip->dev, addr);
a = (uint*)bp->data;
uint pos = bn % NINDIRECT;
if ((addr = a[pos]) == 0) {
    a[pos] = addr = balloc(ip->dev);
    log_write(bp);
}
brelse(bp);
return addr;
}
panic("bmap: out of range");
}

```

`itrunc` 함수를 수정해보자. `itrunc` 는 파일이 가지고 있던 모든 블록(**direct, single indirect, double indirect**)을 순서대로 해제하고, **inode** 필드(**addrs** 와 **size**)를 초기화하여 파일 용량을 0바이트로 줄이는 역할을 수행한다. 기존에는 **double indirect**가 구현되어 있지 않았으므로 추가해주었다. 동작 로직은 다음과 같다.

- `ip->addrs[NDIRECT + 1]` 를 통해 두 번째 레벨 indirect 블록 자체가 할당이 되어있는지 검사한다.
- 두 번째 레벨 indirect 블록을 **a1**라는 배열이 역할을 대신하게 만든다. 배열을 순회하며 첫 번째 레벨 indirect 블록이 할당되었는지 확인한다.
- 만약 첫 번째 레벨 블록이 할당되어 있다면, 첫 번째 레벨 indirect 블록을 **a2**라는 배열이 역할을 하도록 한다.
- **a2** 배열을 순회하며 `bfree(ip->dev, a2[j])` 를 통해 실제 데이터 블록을 해제해준다.
- `bfree(ip->dev, a1[i]);`를 통해 첫 번째 레벨 블록을 해제해준다.
- `bfree(ip->dev, ip->addrs[NDIRECT + 1]);`를 통해 두 번째 레벨 블록을 해제해준다.

```

void
itrunc(struct inode *ip)
{
    int i, j;
    struct buf *bp1, *bp2;
    uint *a1, *a2;

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

    if(ip->addrs[NDIRECT]){
        bp1 = bread(ip->dev, ip->addrs[NDIRECT]);
        a1 = (uint*)bp1->data;
        for(j = 0; j < NINDIRECT; j++){
            if(a1[j])

```

```

        bfree(ip->dev, a1[j]);
    }
    brelse(bp1);
    bfree(ip->dev, ip->addr[NDIRECT]);
    ip->addr[NDIRECT] = 0;
}

if(ip->addr[NDIRECT + 1]){
    bp1 = bread(ip->dev, ip->addr[NDIRECT + 1]);
    a1 = (uint*)bp1->data;

    for(i = 0; i < NINDIRECT; i++){
        if(a1[i]){
            bp2 = bread(ip->dev, a1[i]);
            a2 = (uint*)bp2->data;

            for(j = 0; j < NINDIRECT; j++){
                if(a2[j])
                    bfree(ip->dev, a2[j]);
            }

            brelse(bp2);
            bfree(ip->dev, a1[i]);
        }
    }

    brelse(bp1);
    bfree(ip->dev, ip->addr[NDIRECT + 1]);
    ip->addr[NDIRECT + 1] = 0;
}

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

```

Large Files Result

테스트 코드를 살펴보면 다음과 같다.

연속적인 블록 할당 및 파일 확장

"big.file"이라는 새로운 파일을 생성하고 쓰기 모드로 연다.

buf 배열 크기는 BSIZE (1024바이트)이다.

buf 배열의 맨 앞 4바이트를 **blocks**(블록 번호)로 지정한다.

write를 호출하여 BSIZE만큼 기록하고(앞 4바이트는 블록 번호이며 나머지 부분은 쓰레기 값이 기록되어 있을 것이다. 그래도 그냥 DISK에 기록한다.)

블록을 모두 소모하면 write가 음수를 반환하여 cc값이 음수를 가지게 되고 break에 걸린다. (cc값은 디스크에 기록한 바이트 수이다.)

총 몇 개의 블록을 기록했는지 출력한다.

```
fd = open("big.file", O_CREATE | O_WRONLY);
if(fd < 0){
    printf("bigfile: cannot open big.file for writing\n");
    exit(-1);
}
blocks = 0;
while(1){
    *(int*)buf = blocks;
    int cc = write(fd, buf, sizeof(buf));
    if(cc <= 0)
        break;
    blocks++;
    if (blocks % 100 == 0){
        printf("blocks %d .", blocks);
    }
}

printf("\nwrote %d blocks\n", blocks);
if(blocks != 65803) {
    printf("bigfile: file is too small\n");
    exit(-1);
}

close(fd);
```

파일 재오픈 및 블록별 데이터 무결성(read) 확인

`close(fd)` 로 닫은 후, `open("big.file", O_RDONLY)` 로 파일을 읽기 모드로 연다.

`for(i = 0; i < blocks; i++)` 루프를 돌면서, 매번 `read` 를 통해 다음 블록(**B**SIZE 바이트)만큼 데이터를 읽는다.

쓸 때 `(int*)buf = blocks` 로 블록 번호를 저장했으므로, 읽을 때는 `*(int*)buf` 가 해당 블록 번호 `i` 와 동일해야 한다.

```
close(fd);
fd = open("big.file", O_RDONLY);
if(fd < 0){
    printf("bigfile: cannot re-open big.file for reading\n");
    exit(-1);
}
for(i = 0; i < blocks; i++){
    int cc = read(fd, buf, sizeof(buf));
    if(cc <= 0){
        printf("bigfile: read error at block %d\n", i);
        exit(-1);
    }
    if(*(int*)buf != i){
        printf("bigfile: read the wrong data (%d) for block %d\n",
            *(int*)buf, i);
        exit(-1);
    }
}
```

```
    }  
}  
  
printf("bigfile done; ok\n");  
  
exit(0);
```

65803 블록에 write를 수행하기 때문에 엄청 오래걸리지만 잘 출력된다.

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

Symbolic Links Design

우선 하드 링크와 심볼릭 링크의 차이에 대해서 살펴봐야한다.

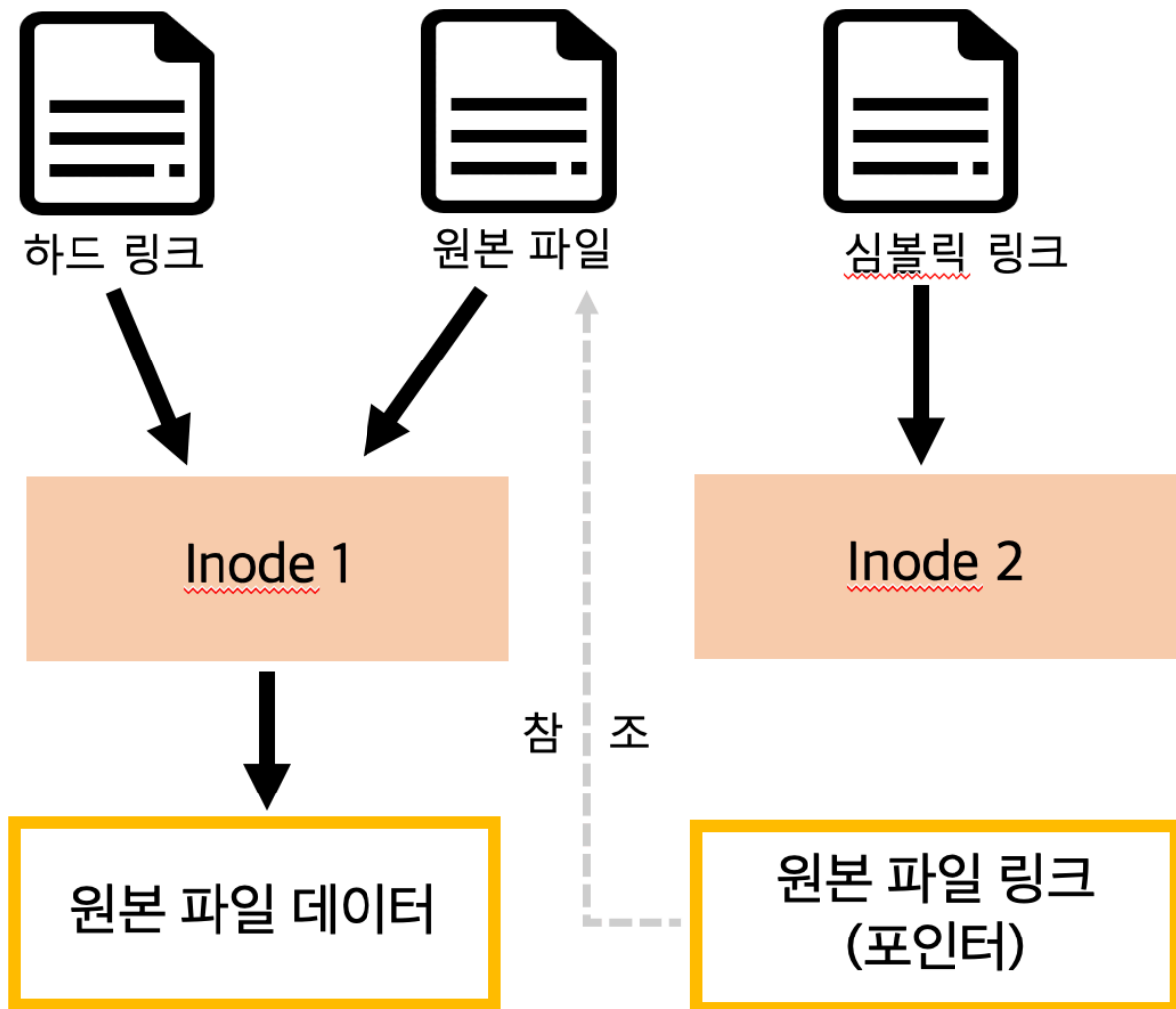
하드링크

하드 링크란 원본 파일과 동일한 **inode**를 가리키는 방식이다. (동일 **inode**번호를 가짐)

inode구조체가 가진 **reference count** 값을 통해 몇 개의 파일이 해당 데이터를 참조하고 있는지 관리한다.

심볼릭 링크

심볼릭 링크는 원본 파일과는 별도의 **inode**를 가리키고 해당



inode 구조체는 다음과 같은 정보들을 가지고 있다.

```
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];  
};
```

하드 링크 생성 시스템 콜

하드 링크는 `link` 시스템 콜을 통해서 생성된다.

- `namei(old)` 를 호출하여 **old** 경로에 해당하는 **inode**를 가져온다.

- `if(ip->type == T_DIR)` 에서 디렉터리인지 검사한다. 디렉터리에는 하드 링크를 만들 수 없다.
- **inode**의 링크 카운트 **nlink**를 증가시키고 `iupdate`를 통해 디스크 업데이트를 수행한다.
- `if((dp = nameiparent(new, name)) == 0)`를 통해 **dp**에 부모 디렉터리 **inode**를 할당한다.
- `if(dp->dev != ip->dev || dirlink(dp, name, ip->inum)`를 통해 **같은 물리 디바이스인지** 확인하고 **부모의 name에 대한 링크**를 생성한다.

```
// Create the path new as a link to the same inode as old.
uint64
sys_link(void)
{
    char name[DIRSIZ], new[MAXPATH], old[MAXPATH];
    struct inode *dp, *ip;

    if(argstr(0, old, MAXPATH) < 0 || argstr(1, new, MAXPATH) < 0)
        return -1;

    begin_op();
    if((ip = namei(old)) == 0){
        end_op();
        return -1;
    }

    ilock(ip);
    if(ip->type == T_DIR){
        iunlockput(ip);
        end_op();
        return -1;
    }

    ip->nlink++;
    iupdate(ip);
    iunlock(ip);

    if((dp = nameiparent(new, name)) == 0)
        goto bad;
    ilock(dp);
    if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
        iunlockput(dp);
        goto bad;
    }
    iunlockput(dp);
    iput(ip);

    end_op();

    return 0;

bad:
    ilock(ip);
    ip->nlink--;
```



```

iupdate(ip);
iunlockput(ip);
end_op();
return -1;
}

```

unlink 시스템 콜

unlink 시스템 콜은 주어진 경로(path)에 있는 파일 또는 빈 디렉터리 엔트리를 파일 시스템에서 제거(unlink)하는 역할을 한다. 동작은 다음과 같다.

- nameiparent(path, name)를 통해 dp에는 부모의 inode가 name에는 경로 말단의 파일의 이름이 들어간다.
- namecmp를 통해 .과 ..인지 확인한다. 파일 시스템 무결성을 위해 이 두 경우는 절대 unlink 하면 안된다.
- 참조 횟수(nlink)가 1보다 작다면 unlink 할 수 없으므로 panic이 호출된다.
- 디렉터리를 지우는 경우도 빈 디렉터리인 경우에만 삭제할 수 있도록 한다.
- memset(&de, 0, sizeof(de));를 통해 de에 0으로 세팅해준 뒤 writei(dp, 0, (uint64)&de, off, sizeof(de))를 통해 부모 inode의 해당 name에 대한 디렉터리 엔트리를 싹 날려준다. dirlookup(dp, name, &off)에서 off부분이 부모 디렉터리 블록에서 해당 엔트리(name에 대한)가 차지하는 byte offset이다.
- 삭제 대상이 디렉터리인 경우, 디렉터리의 ..이 부모로 링크 카운트를 하나 만들어주고 있었으므로 dp의 nlink를 하나 감소시켜준다.
- 대상 inode의 nlink를 하나 감소시켜준다.

```

uint64
sys_unlink(void)
{
    struct inode *ip, *dp;
    struct dirent de;
    char name[DIRSIZ], path[MAXPATH];
    uint off;

    if(argstr(0, path, MAXPATH) < 0)
        return -1;

    begin_op();
    if((dp = nameiparent(path, name)) == 0){
        end_op();
        return -1;
    }

    ilock(dp);

    // Cannot unlink "." or "..".
    if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
        goto bad;

    if((ip = dirlookup(dp, name, &off)) == 0)
        goto bad;
    ilock(ip);

```

```

if(ip->nlink < 1)
    panic("unlink: nlink < 1");
if(ip->type == T_DIR && !isdirempty(ip)){
    iunlockput(ip);
    goto bad;
}

memset(&de, 0, sizeof(de));
if(writei(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
    panic("unlink: writei");
if(ip->type == T_DIR){
    dp->nlink--;
    iupdate(dp);
}
iunlockput(dp);

ip->nlink--;
iupdate(ip);
iunlockput(ip);

end_op();

return 0;

bad:
iunlockput(dp);
end_op();
return -1;
}

```

디렉토리 생성 mkdir 시스템콜

mkdir 시스템 콜은 다음과 같다.

`create(path, T_DIR, 0, 0)` 를 통해 **path**에 실제 디렉터리 **inode**를 생성한다. 구체적으로 한 번 살펴보자.

```

uint64
sys_mkdir(void)
{
    char path[MAXPATH];
    struct inode *ip;

    begin_op();
    if(argstr(0, path, MAXPATH) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
        end_op();
        return -1;
    }
    iunlockput(ip);
    end_op();
    return 0;
}

```

```
}
```

create 함수의 동작 로직을 살펴보기 전에 **create** 함수에서 사용되는 함수들을 먼저 확인해보자.

nameparent, namex

namex의 역할은 다음과 같다.

- 주어진 경로 문자열(**path**)에 해당하는 **inode**를 찾아 반환해준다. 또한, 호출자가 **마지막 경로 요소의 부모 디렉터리 inode**를 원할 때 **nameparent == 1**, 그 부모 **inode**를 반환하도록 동작한다. 예를 들어 **path = "/usr/bin/ls"**, **nameparent = 1**이면, 최종적으로 **/usr/bin** 디렉터리의 **inode**를 반환하고, **name**에 **ls**를 저장한다.

namex의 동작 로직을 살펴보자.

- 처음에는 **path**가 '/'로 시작하는지 확인하여 상대 경로인지 절대 경로인지 확인한다. 포함되어 있다면 절대 경로이다. 여기서 **ip**는 다음 요소를 검색하기 전의 현재 디렉터리 inode 역할을 하게 된다.
- `while((path = skipelem(path, name)) != 0)`를 통해 **path** 문자열에서 가장 앞에 있는 다음 경로 요소(/로 구분된 이름)를 추출하여 **name** 버퍼에 복사하고 반환 값으로 남은 경로 문자열의 시작 주소를 반환한다.

예시: **path = "/usr/bin/ls"**로 시작할 때, 첫 호출 `skipelem("/usr/bin/ls", name) → name = "usr"`, 반환값 **path = "bin/ls"**

- 루프의 본문을 살펴보면 다음과 같다. 만약 **T_DIR**(디렉토리)가 아니라면 더 이상 하위 요소를 탐색할 수 없으므로 0을 반환한다. `if (nameparent && *path == '\0')`에서는 마지막 요소의 부모 디렉터리 **inode**(이 시점에서 **ip**는 부모 디렉터리의 **inode**이다.)만 얻고자 하며, **name**버퍼에는 마지막 요소 이름이 들어있다.

예시: **path = "/usr/bin/ls"**에서 **ip** (지금의 **/usr/bin** 디렉터리 **inode**)을 락 해제 후 반환하고 **name**에는 **ls**가 담겨 있게 된다.

- `next = dirlookup(ip, name, 0)`에서 `dirlookup`은 디렉터리 **ip**의 데이터 블록을 모두 읽어들이어서 디렉터리 항목 중 **name**과 일치하는 것이 있으면 해당 항목의 **inode** 번호를 얻은 뒤 `iget`으로 **inode**를 가져와 **next**에 넣는다.
- ip = next**를 하여 디렉토리 레벨 한 단계 아래로 내려가며 루프를 반복한다.
- 마지막은 부모 디렉터리 반환을 못하고 루프가 끝났을 때를 처리해준다.

```
struct inode*
nameparent(char *path, char *name)
{
    return namex(path, 1, name);
}

// Look up and return the inode for a path name.
// If parent != 0, return the inode for the parent and copy the final
// path element into name, which must have room for DIRSIZ bytes.
// Must be called inside a transaction since it calls iput().
static struct inode*
namex(char *path, int nameparent, char *name)
{
    struct inode *ip, *next;

    if(*path == '/')
        ip = iget(ROOTDEV, ROOTINO);
    else
```

```

ip = idup(myproc()->cwd);

while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
        iunlockput(ip);
        return 0;
    }
    if(nameiparent && *path == '\\0'){
        // Stop one level early.
        iunlock(ip);
        return ip;
    }
    if((next = dirlookup(ip, name, 0)) == 0){
        iunlockput(ip);
        return 0;
    }
    iunlockput(ip);
    ip = next;
}
if(nameiparent){
    iput(ip);
    return 0;
}
return ip;
}

```

create 함수의 동작 로직은 다음과 같다.

- `(dp = nameiparent(path, name))` 에서 마지막 경로 요소(**name**)와 그 부모 디렉터리(**dp**)의 **inode**를 구한다.
- `(ip = dirlookup(dp, name, 0)) != 0` 를 통해 이미 동일한 이름(**name**)으로 엔트리가 존재하는지 확인한다. 존재하면 **name**에 대한 **inode**를 반환한다.
- 구체적으로는 요청한 타입이 **T_FILE**(일반 파일)이고 실제로 존재하던 **ip->type**이 **T_FILE** 또는 **T_DEVICE**라면 (즉, 같은 파일 또는 동일한 종류의 디바이스를 재사용할 수 있다면) **inode(ip)**를 반환한다.
- 그게 아니라면 동일한 이름이 없으므로 `ip = ialloc(dp->dev, type)` 를 통해 새롭게 **inode**를 할당하고 아래와 같이 주요 필드를 초기화해준다.

`ip->major = major;` 주로 장치 파일인 경우 major 번호 설정

`ip->minor = minor;` 장치 파일인 경우 minor 번호 설정

`ip->nlink = 1;` 초기 링크 수 새롭게 생성되었으므로 1 (부모가 name으로 자식에 링크를 걸 것이기 때문)

그 다음 `iupdate(ip);` 를 통해 **inode** 정보를 디스크에 기록한다.(커널 메모리->디스크)

만약 생성하는게 디렉토리인 경우(`type == T_DIR`), `dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0` 를 통해

"."은 자기 자신으로의 링크,

".."은 부모 디렉터리를 가리키는 링크로 만든다.

`dirlink(dp, name, ip->inum)` 를 통해 부모 디렉터리(**dp**)에 **name**이란 이름으로 새 **inode(ip)**를 링크한다.

마지막으로 새롭게 디렉터리를 만든 경우, 부모 디렉터리의 링크 수를 증가시킨다.

dp->nlink++; “.” 엔트리가 생겼으므로 부모의 nlink를 +1

iupdate(dp); 변경된 부모 inode 정보 디스크에 기록

ip를 반환한다.

```
static struct inode*
create(char *path, short type, short major, short minor)
{
    struct inode *ip, *dp;
    char name[DIRSIZ];

    if((dp = nameiparent(path, name)) == 0)
        return 0;

    ilock(dp);

    if((ip = dirlookup(dp, name, 0)) != 0){
        iunlockput(dp);
        ilock(ip);
        if(type == T_FILE && (ip->type == T_FILE || ip->type == T_DEVICE))
            return ip;
        iunlockput(ip);
        return 0;
    }

    if((ip = ialloc(dp->dev, type)) == 0){
        iunlockput(dp);
        return 0;
    }

    ilock(ip);
    ip->major = major;
    ip->minor = minor;
    ip->nlink = 1;

    iupdate(ip);

    if(type == T_DIR){ // Create . and .. entries.
        // No ip->nlink++ for ".": avoid cyclic ref count.
        if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
            goto fail;
    }

    if(dirlink(dp, name, ip->inum) < 0)
        goto fail;

    if(type == T_DIR){
        // now that success is guaranteed:
        dp->nlink++; // for ".."
        iupdate(dp);
    }
}
```

```

}

iunlockput(dp);

return ip;

fail:
// something went wrong. de-allocate ip.
ip->nlink = 0;
iupdate(ip);
iunlockput(ip);
iunlockput(dp);
return 0;
}

```

파일 열기 open 시스템 콜

`sys_open` 함수의 동작 로직은 다음과 같다.

- `if(omode & O_CREATE)` 를 통해 **O_CREATE** 플래그를 확인하여 설정되어 있다면 `create` 를 통해 파일을 새로 만들고 **O_CREATE** 플래그가 설정이 되어있지 않다면 `namei(path)` 를 통해 해당 **path**에 대한 최종 **inode**를 가져와 기존 파일/디렉터리를 연다.(경로 중간에 디렉터리가 없거나 접근 권한이 없으면 0 반환)
- `if (ip->type == T_DIR && omode != O_RDONLY)` 를 통해 디렉터리를 쓰기 모드로 열려고 하는지 검사한다. (디렉터리 쓰기 모드는 허용되지 않는다)
- **inode**가 **T_DEVICE**(디바이스 노드)인데, **major** 번호가 유효 범위를 벗어나면 에러를 처리한다.
- `(f = filealloc()) == 0 || (fd = fdalloc(f))` 를 통해 파일 구조체와 파일 디스크립터를 할당해준다.
- 할당한 파일 구조체를 초기화해준다.

`if (ip->type == T_DEVICE)` -> 특수 장치 파일인 경우

- `f->type = FD_DEVICE`
- `f->major = ip->major` (장치 구분 번호)

그 외(대부분 일반 파일이거나 디렉터리)인 경우

- `f->type = FD_INODE`
- `f->off = 0` 으로 초기 파일 오프셋을 0으로 설정

공통적으로

- `f->ip = ip;` **file** 구조체가 참조하는 실제 **inode**를 연결
- `f->readable = !(omode & O_WRONLY);` **O_WRONLY** 비트가 설정되어 있으면 읽기 불가
- `f->writable = (omode & O_WRONLY) || (omode & O_RDWR);` **O_WRONLY** 혹은 **O_RDWR** 플래그가 있는 경우 쓰기 권한
- `if ((omode & O_TRUNC) && ip->type == T_FILE)` 는 **O_TRUNC** 비트를 주고 파일을 열면 열자마자 파일 내용을 모두 삭제하고 크기를 0으로 줄인다는 의미이다.
- 성공적으로 파일이 열리면 파일 디스크립터를 반환한다.

uint64

```

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

    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);

        if(ip->type == T_DIR && omode != O_RDONLY){
            iunlockput(ip);
            end_op();
            return -1;
        }
    }

    if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
        iunlockput(ip);
        end_op();
        return -1;
    }

    if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
        if(f)
            fileclose(f);
        iunlockput(ip);
        end_op();
        return -1;
    }

    if(ip->type == T_DEVICE){
        f->type = FD_DEVICE;
        f->major = ip->major;
    } else {

```

```

    f->type = FD_INODE;
    f->off = 0;
}
f->ip = ip;
f->readable = !(omode & O_WRONLY);
f->writable = (omode & O_WRONLY) || (omode & O_RDWR);

if((omode & O_TRUNC) && ip->type == T_FILE){
    itrunc(ip);
}

iunlock(ip);
end_op();

return fd;
}

```

Symbolic Links Implementation

sybolic link를 구현하기에 앞서서 **target**경로를 어디에다가 저장할지 정해야한다. 가장 편리한 방법으로 **inode**가 가리키는 데이터 블록에다가 경로를 저장하고 읽으려고 한다(`writel`와 `readl` 활용하기).

T_SYMLINK 필드를 추가한다.

```

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

struct stat {
    int dev;        // File system's disk device
    uint ino;       // Inode number
    short type;     // Type of file
    short nlink;    // Number of links to file
    uint64 size;    // Size of file in bytes
};

```

O_NOFOLLOW 필드를 추가한다. 이 부분은 **open**시스템 콜 도중 **inode**가 심볼릭 링크인 경우 **O_NOFOLLOW**플래그가 있을 때 그 심볼릭 링크를 따라가지 않고(follow하지 않고), **inode** 자체를 접근하도록 한다.

```

#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
#define O_TRUNC   0x400
#define O_NOFOLLOW 0X800

```

create를 수정하자. **type == T_SYMLINK**에 대하여 이미 **name**에 대한 **inode**가 존재하는 경우 **ip**를 반환한다. 없는 경우에는 **ialloc**을 통해 생성해주며 기본적인 필드들을 초기화해준다. 추가로 구현된 이 부분은 **sys_symlink**에서 링크를 호출하는 주체의 말단 **inode**를 반환하기 위해 추가된다.


```

static struct inode*
create(char *path, short type, short major, short minor)
{
    struct inode *ip, *dp;
    char name[DIRSIZ];

    if((dp = nameiparent(path, name)) == 0)
        return 0;

    ilock(dp);

    if((ip = dirlookup(dp, name, 0)) != 0){
        iunlockput(dp);
        ilock(ip);
        if(type == T_FILE && (ip->type == T_FILE || ip->type == T_DEVICE))
            return ip;
        //추가추가
        if(type == T_SYMLINK) {
            return ip;
        }
        //
        iunlockput(ip);
        return 0;
    }
    if((ip = ialloc(dp->dev, type)) == 0){
        iunlockput(dp);
        return 0;
    }

    ilock(ip);
    ip->major = major;
    ip->minor = minor;
    ip->nlink = 1;

    iupdate(ip);

    if(type == T_DIR){ // Create . and .. entries.
        // No ip->nlink++ for ".": avoid cyclic ref count.
        if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
            goto fail;
    }

    if(dirlink(dp, name, ip->inum) < 0)
        goto fail;

    if(type == T_DIR){
        // now that success is guaranteed:
        dp->nlink++; // for ".."
        iupdate(dp);
    }

    iunlockput(dp);

```

```

    return ip;

fail:
    // something went wrong. de-allocate ip.
    ip->nlink = 0;
    iupdate(ip);
    iunlockput(ip);
    iunlockput(dp);
    return 0;
}

```

sys_symlink 시스템콜을 구현하자. 심볼릭 링크를 수행하는 주체가 심볼릭 링크 대상에 대한 주소를 `writei` 를 통해 디스크에다가 저장하는 방식을 택했다. 심볼릭 링크를 수행하는 주체의 **inode**는 `create` 를 통해 **T_SYMLINK** 타입을 지니게 된다.

```

uint64
sys_symlink(void)
{
    char target[MAXPATH], path[MAXPATH];
    struct inode *ip;
    int len, total;
    char buf[sizeof(len) + MAXPATH + 1];

    // target(심볼릭 링크를 걸 대상)경로와 path(심볼릭 링크를 수행하는 주체)경로를 가져옴
    if (argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0)
        return -1;

    begin_op();

    // T_SYMLINK 타입으로 inode를 만들거나 가져옴
    ip = create(path, T_SYMLINK, 0, 0);
    if (ip == 0) {
        end_op();
        return -1;
    }
    // offset 0번에 target 길이를 기록
    len = strlen(target);
    total = sizeof(len) + len + 1;
    memmove(buf, &len, sizeof(len)); // len
    memmove(buf + sizeof(len), target, len + 1); // target에다가 마지막 null
    if (writei(ip, 0, (uint64)buf, 0, total) != total) { //디스크에 기록
        iunlockput(ip);
        end_op();
        return -1;
    }
    // 디스크에 반영한다
    iupdate(ip);
    iunlockput(ip);
    end_op();
    return 0;
}

```

`sys_open`을 다음과 같이 수정하였다. 추가된 부분의 로직은 다음과 같다.

- 타입이 `T_SYMLINK`이며 `O_NOFOLLOW`가 세팅이 안되어 있는 경우에 심볼릭 링크를 따라가서 실제 대상(target) 파일로 열게 된다.
- `while (ip->type == T_SYMLINK && count < 10)` 을 통해 반복을 수행할 것인데, **count**는 b->a->b.. 처럼 서로가 서로에 대해 **symbolic link**를 만든 경우에 대하여 순환이 발생할 경우 제한을 둔 것이다.
- `readi`를 통해 디스크 블록에 저장된 심볼릭 링크 대상 경로의 길이와 경로 정보를 가져온다.
- inode**값을 심볼릭 링크 대상으로 갱신하고 다시 while루프를 수행한다.
- 만약 **count**값이 10 이상이 되는 경우 사이클이 발생한 것이므로 -1을 반환한다.

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

    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);

        // 추가추가
        if(ip->type == T_SYMLINK && !(omode & O_NOFOLLOW)){
            int count = 0;

            while(ip->type == T_SYMLINK && count < 10){
                char buf[sizeof(int) + (MAXPATH+1)];
                int total = ip->size;
                int len;
                if(readi(ip, 0, (uint64)buf, 0, total) != total){
                    panic("readi failed!");
                }
            }
        }
    }
}
```

```

    memmove(&len, buf, sizeof(int));

    char *target = buf + sizeof(int);
    target[len] = '\0'; //마지막 null

    iunlockput(ip);

    // 경로 따라가기, 해당 경로가 존재하지 않으면 -1 리턴
    ip = namei(target);
    if(ip == 0){
        end_op();
        return -1;
    }
    ilock(ip);
    count++;
}

// 사이클이 발생한 경우
if(count >= 10){
    iunlockput(ip);
    end_op();
    return -1;
}
//

if(ip->type == T_DIR && omode != O_RDONLY){
    iunlockput(ip);
    end_op();
    return -1;
}

if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
    iunlockput(ip);
    end_op();
    return -1;
}

if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
        fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
}

if(ip->type == T_DEVICE){
    f->type = FD_DEVICE;
    f->major = ip->major;
} else {
    f->type = FD_INODE;

```

```

    f->off = 0;
}
f->ip = ip;
f->readable = !(omode & O_WRONLY);
f->writable = (omode & O_WRONLY) || (omode & O_RDWR);

if((omode & O_TRUNC) && ip->type == T_FILE){
    itrunc(ip);
}

iunlock(ip);
end_op();

return fd;
}

```

Symbolic Links Result

테스트 코드를 살펴보면 다음과 같다.

`main` 함수에서 `cleanup`, `testsymlink`, `concur`, `exit` 을 차례대로 호출한다.

```

int
main(int argc, char *argv[])
{
    cleanup();
    testsymlink();
    concur();
    exit(failed);
}

```

`cleanup` 은 테스트를 시작하기 전, `/testsymlink` 디렉터리와 그 내부에 남아 있을 수 있는 이전 테스트 결과물(파일 또는 링크)을 모두 삭제한다. 이렇게 함으로써 이후 테스트가 항상 깨끗한 상태에서 실행되도록 만든다.

```

static void
cleanup(void)
{
    unlink("/testsymlink/a");
    unlink("/testsymlink/b");
    unlink("/testsymlink/c");
    unlink("/testsymlink/1");
    unlink("/testsymlink/2");
    unlink("/testsymlink/3");
    unlink("/testsymlink/4");
    unlink("/testsymlink/z");
    unlink("/testsymlink/y");
    unlink("/testsymlink");
}

```

`mkdir("/testsymlink")` 를 통해 `/testsymlink` 디렉터를 만들고, 이후 모든 테스트 파일/링크를 이 안에 생성한다.

- `open("/testsymlink/a", O_CREATE | O_RDWR)` 를 통해 `/testsymlink/a` 라는 일반 파일을 새로 생성하고 읽기·쓰기 모두 가능하도록 연다.
- `symlink("/testsymlink/a", "/testsymlink/b")` 를 통해 `/testsymlink/b` 라는 이름의 심볼릭 링크를 생성한다. 이 링크는 내부적으로 실제 대상(target)을 `/testsymlink/a` 로 기록하고 이후에 `/testsymlink/b` 를 열면 운영체제는 자동으로 `/testsymlink/a` 를 가리키게 된다.
- `write(fd1, buf, 4)` 를 통해 `/testsymlink/a` 에 4바이트(a b c d)를 기록한다.
- `stat_slink("/testsymlink/b", &st)` 를 통해 `O_NOFOLLOW` 플래그를 주고 `open()` + `fstat()` 을 내부적으로 호출하여, `/testsymlink/b` 가 실제 심볼릭 링크 자체인지 검사한다.
- `open("/testsymlink/b", O_RDWR)` 후 `read(fd2, &c, 1)` 를 호출하여 심볼릭 링크를 통해 실제 파일 `/testsymlink/a` 에 접근한다. 링크 **b**가 가리키는 대상이 **a**이므로, `read(fd2, &c, 1)` 은 곧 `/testsymlink/a` 의 첫 바이트('a')를 읽어와야 한다.
- `unlink("/testsymlink/a")` 를 통해 `/testsymlink`와 그 자식인 **a**와의 링크를 해제하고, `/testsymlink/b`를 통해 다시 열어보려 시도한다. 이 시점에 **b**가 갖고 있는 경로 정보 `/testsymlink/a`가 유효하지 않으므로, 심볼릭 링크 **b**는 깨진 링크 상태가 된다. 따라서 `open("/testsymlink/b", O_RDWR)` 은 실패해야 한다. 오픈이 성공한다면 테스트 실패로 처리한다.
- `/testsymlink/a -> /testsymlink/b` 이제 **a**는 다시 심볼릭 링크가 된다, `symlink` 시스템 콜에서 `create`를 통해 **a**에 대한 정보를 다시 만들고 **b**로의 심볼릭 링크를 형성한다. 이제 "**b** -> **a** -> **b**"로 cycle이 형성된다. 이 상태에서 `open("/testsymlink/b", O_RDWR)` 를 시도하면, 운영체제는 순환 참조를 감지하여 실패해야 한다. 만약 실패하지 않고 열리면 테스트 실패.
- `symlink("/testsymlink/nonexistent", "/testsymlink/c")` 에서 타깃이 없어도 심볼릭 링크 생성 자체는 허용된다. 따라서 최종 0을 반환하여 `r=0` 이 될 것이다. 추후에 `open` 을 통해 없는 경로에 접근할 경우 **-1**을 반환하게 될 것이다.
- `/testsymlink/1 -> /testsymlink/2`
`/testsymlink/2 -> /testsymlink/3`
`/testsymlink/3 -> /testsymlink/4` 다음과 같은 심볼릭 체인을 만드는데 이때 `/testsymlink/4`는 아직 존재하지 않으므로, 링크로만 연결되는 상태이다. 체인의 마지막 노드인 `/testsymlink/4`를 `O_CREATE | O_RDWR`로 열어 실제 파일로 만들고 `/testsymlink/1`을 `O_RDWR`로 열면, 내부적으로 `1 -> 2 -> 3 -> 4` 체인을 따라가서 결국 `/testsymlink/4` 파일 디스크립터를 얻는다.
`write(fd2, &c, 1)` → `/testsymlink/1` 통해 체인 마지막인 `/testsymlink/4`에 1바이트를 쓴다. 실제로 `/testsymlink/1`을 열면 `/testsymlink/4` 파일의 FD가 반환되므로, 여기서는 최종적으로 `/testsymlink/4`에 대입되는 셈이다. 그 뒤 `read(fd1, &c2, 1)` → `fd1` 역시 `/testsymlink/4` 파일 디스크립터이므로, 동일한 파일에서 읽게 된다. 쓴 값과 읽은 값이 동일하면 성공이다.

```
static void
testsymlink(void)
{
    int r, fd1 = -1, fd2 = -1;
    char buf[4] = {'a', 'b', 'c', 'd'};
    char c = 0, c2 = 0;
    struct stat st;

    printf("Start: test symlinks\n");

    mkdir("/testsymlink");
```

```

fd1 = open("/testsymlink/a", O_CREATE | O_RDWR);
if(fd1 < 0) fail("failed to open a");

r = symlink("/testsymlink/a", "/testsymlink/b");
if(r < 0)
    fail("symlink b -> a failed");

if(write(fd1, buf, sizeof(buf)) != 4)
    fail("failed to write to a");

if (stat_slink("/testsymlink/b", &st) != 0)
    fail("failed to stat b");
if(st.type != T_SYMLINK)
    fail("b isn't a symlink");

fd2 = open("/testsymlink/b", O_RDWR);
if(fd2 < 0)
    fail("failed to open b");
read(fd2, &c, 1);
if (c != 'a')
    fail("failed to read bytes from b");

unlink("/testsymlink/a");
if(open("/testsymlink/b", O_RDWR) >= 0)
    fail("Should not be able to open b after deleting a");

r = symlink("/testsymlink/b", "/testsymlink/a");
if(r < 0)
    fail("symlink a -> b failed");

r = open("/testsymlink/b", O_RDWR);
if(r >= 0)
    fail("Should not be able to open b (cycle b->a->b->..)\n");

r = symlink("/testsymlink/nonexistent", "/testsymlink/c");
if(r != 0)
    fail("Symlinking to nonexistent file should succeed\n");

r = symlink("/testsymlink/2", "/testsymlink/1");
if(r) fail("Failed to link 1->2");
r = symlink("/testsymlink/3", "/testsymlink/2");
if(r) fail("Failed to link 2->3");
r = symlink("/testsymlink/4", "/testsymlink/3");
if(r) fail("Failed to link 3->4");

close(fd1);
close(fd2);

fd1 = open("/testsymlink/4", O_CREATE | O_RDWR);
if(fd1<0) fail("Failed to create 4\n");
fd2 = open("/testsymlink/1", O_RDWR);
if(fd2<0) fail("Failed to open 1\n");

```

```

c = '#';
r = write(fd2, &c, 1);
if(r!=1) fail("Failed to write to 1\n");
r = read(fd1, &c2, 1);
if(r!=1) fail("Failed to read from 4\n");
if(c!=c2)
    fail("Value read from 4 differed from value written to 1\n");

printf("test symlinks: ok\n");
done:
close(fd1);
close(fd2);
}

```

testsymlink에서 stat_slink("/testsymlink/b", &st)에 대한 코드이다.

경로(pn)를 읽기 전용(O_RDONLY) 및 O_NOFOLLOW으로 연다. O_NOFOLLOW 플래그를 추가함으로써 pn이 심볼릭 링크라도 그 링크 자체만 열라는 의미가 된다. fstat은 그 링크 메타데이터를 st에 채워 넣는 과정이다.

```

// stat a symbolic link using O_NOFOLLOW
static int
stat_slink(char *pn, struct stat *st)
{
    int fd = open(pn, O_RDONLY | O_NOFOLLOW);
    if(fd < 0)
        return -1;
    if(fstat(fd, st) != 0)
        return -1;
    return 0;
}

```

concur에 대한 코드에서 테스트하고자 하는 것은 다음과 같다. 두 자식 프로세스가 거의 동시에 /testsymlink/y 심볼릭 링크를 만들기도 하고, 삭제하기도 하면서 심볼릭 링크를 만들 때 **inode allocation** 및 **디렉터리 엔트리 insertion**과 삭제할 때 **디렉터리 엔트리 제거 & 하드 링크 카운트 감소** 작업이 서로 엉키지 않고 올바르게 처리되는지 확인하는 것이다. 즉 **race condition**이 발생하는지 확인하는 코드이다.

- open("/testsymlink/z", O_CREATE | O_RDWR);를 통해 z파일을 하나 생성한다. O_CREATE | O_RDWR 플래그를 줘서, 없으면 만들고 읽기·쓰기 모드로 연다.
- for(int j = 0; j < nchild; j++) fork();에서 두 번 fork를 호출해서 총 두 개의 자식 프로세스를 만들어 낸다. 각 자식 프로세스는 pid == 0 부분의 코드를 실행한다.
- int m = 0; unsigned int x = (pid ? 1 : 97); pid 값이 0이되면 x=97이고, 그렇지 않으면 x=1이 된다. pid는 자식 입장에서 항상 0이므로 x=97이 들어가지만, 두 자식 모두 별개로 랜덤 시드를 초기화한다는 의도이다.
- for(i = 0; i < 100; i++)에서 총 100번 반복하면서, 난수 x = x * 1103515245 + 12345;를 통해 x를 갱신하고, if(x % 3 == 0) -> 1/3 확률로 symlink(심볼릭 링크 생성) 아니면 2/3 확률로 unlink를 시도한다. 존재하지 않아도 unlink는 내부적으로 문제없이 그냥 넘어간다.
- 100번 반복 후 자식 프로세스는 exit(0)으로 정상 종료된다.
- for(int j = 0; j < nchild; j++)에서 wait(&r)을 통해 자식을 수거한다.


```

static void
concur(void)
{
    int pid, i;
    int fd;
    struct stat st;
    int nchild = 2;

    printf("Start: test concurrent symlinks\n");

    fd = open("/testsymlink/z", O_CREATE | O_RDWR);
    if(fd < 0) {
        printf("FAILED: open failed");
        exit(1);
    }
    close(fd);

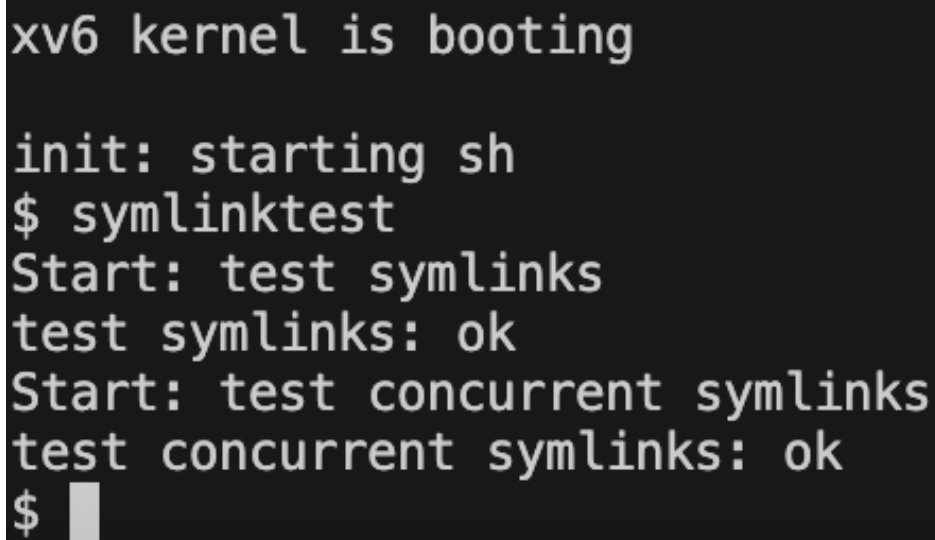
    for(int j = 0; j < nchild; j++) {
        pid = fork();
        if(pid < 0){
            printf("FAILED: fork failed\n");
            exit(1);
        }
        if(pid == 0) {
            int m = 0;
            unsigned int x = (pid ? 1 : 97);
            for(i = 0; i < 100; i++){
                x = x * 1103515245 + 12345;
                if((x % 3) == 0) {
                    symlink("/testsymlink/z", "/testsymlink/y");
                    if (stat_slink("/testsymlink/y", &st) == 0) {
                        m++;
                        if(st.type != T_SYMLINK) {
                            printf("FAILED: not a symbolic link %d\n", st.type);
                            exit(1);
                        }
                    }
                } else {
                    unlink("/testsymlink/y");
                }
            }
            exit(0);
        }
    }

    int r;
    for(int j = 0; j < nchild; j++) {
        wait(&r);
        if(r != 0) {
            printf("test concurrent symlinks: failed\n");
            exit(1);
        }
    }
}

```

```
}  
printf("test concurrent symlinks: ok\n");  
}
```

결과는 정상적으로 동작한다.



```
xv6 kernel is booting  
  
init: starting sh  
$ symlinktest  
Start: test symlinks  
test symlinks: ok  
Start: test concurrent symlinks  
test concurrent symlinks: ok  
$
```

Troubleshooting

`copyout` 호출 시 `fork` 이후에 read only가 된 페이지에 대하여 -1을 반환하는 조건문 때문에 `copyout` 이 제대로 동작하지 않는 오류가 있었다. `if(pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0)` 를 통한 수정을 통해 해결할 수 있었다.

처음에 페이지 폴트가 발생하는 케이스만 고려하다가 `pipe` 를 통한 `read` `write` 을 통해 부모 페이지가 오염될 수 있는 케이스를 확인해볼 수 있었다. `copyout` 에 COW 로직을 추가하여 해결할 수 있었다.

`open` 했을 때 `O_NOFOLLOW` 플래그를 줬음에도 여전히 따라가 버리는 현상이 있었다. `open()` 안의 symlink follow 루프에서 `O_NOFOLLOW` 플래그 검사 위치를 잘못 설정하여 플래그가 제대로 전달되지 못하는 것을 확인하였다.