Project03 Wiki

이름: 권도현

학번:2023065350

학과: 컴퓨터소프트웨어학부

---

1. Copy-on-write

PREVIEW

- `COW FORK` : Fork() 시, child와 parent가 physical page를 공유하고, write 시 physical page가 분리되는 방식
- `Initial state` : Fork() 직후의 상태
  - Child page table은 Parent's physical page를 가리킨다.
  - Child, parent process에서 모든 User PTE는 Read-only
  - COW bit로 RSW bit를 사용한다.
- `Write access`
  - 우선 Write 시도에는 Page fault가 발생해야 한다.
  - Kernel이 새로운 physical page를 할당한다.
  - 새로운 physical page에 기존 page를 복사한다.
  - PTE에 write 권한을 부여한다.
- `Reference counting`
  - 각 Page를 공유하고 있는 Process의 수를 추적한다.
  - Reference count = 0인 경우에만 page를 free한다.

수정해야 하는 함수들의 기존 구현을 살펴보자.

a. `uvmcopy()`

```
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){
    // Logical address에 해당하는 PTE를 찾음
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    // PTE 존재하고 유효한지 확인
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    // 해당 PTE에 맞는 physical address를 가져옴
    pa = PTE2PA(*pte);
    // 현재 PTE의 권한을 가져옴
    flags = PTE_FLAGS(*pte);
    // 새로운 Physical page allocation
    if((mem = kalloc()) == 0)
      goto err;
    // mem에 parent page 복사
    memmove(mem, (char*)pa, PGSIZE);
    // 새롭게 생성한 page를 page table에 매핑
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
  }
  return 0;

 err:
  // i: 0 ~ PGSIZE
  // 중간에 복사 실패 시, 이전까지 매핑한 페이지 해제
  uvmunmap(new, 0, i / PGSIZE, 1);
```

```
  return -1;
}
```

b. `usertrap()`

```
void
usertrap(void)
{
  int which_dev = 0;

  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();
    // system call 호출출
    syscall();
  } else if((which_dev = devintr()) != 0){
```

```
    // ok
  } else {
    printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p→
    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();
  // User space로 복귀
  usertrapret();
}
```

c. copyout()

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

  while(len > 0){
    // Page 단위 정렬
    va0 = PGROUNDDOWN(dstva);
    if(va0 >= MAXVA)
      return -1;
    // va0에 맞는 PTE 가져옴
    pte = walk(pagetable, va0, 0);
    // PTE 유효한 지 확인 + Write 권한 여부 확인
    if(pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0 ||
       (*pte & PTE_W) == 0)
      return -1;
    // PTE에 맞는 physical address 가져옴
    pa0 = PTE2PA(*pte);
```

```
    // Page에서 남은 공간
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    // Kernel space data src를 va에 매핑되는 physical memory로 옮김
    // pa0 + (dstva - va0): page allign x → 차이만큼 앞에서 시작
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

d. `kalloc()`

```
void *
kalloc(void)
{
  // free page가 linked list로 관리됨
  // run은 linked list의 Node
  struct run *r;

  acquire(&kmem.lock);
  // freelist의 가장 첫 번째 page를 가져옴
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    // page 전체를 0x5로 채움움
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

e. `kfree()`

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

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

  // Fill with junk to catch dangling refs.
  // 0x1로 memory를 채움
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  // pa에 해당하는 page를 freelist에 올림
  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}
```

DESIGN

해야 하는 작업

- Page가 memory에 없어 발생하는 page fault와 COW fork() 후 write access 때 발생하는 page fault를 구분해야 한다.
- Referece count 동기화

주의 해야 할 점

- Memory 범위를 벗어나지 않도록
- 기존 코드와의 호환성 유지

구현 방향성

- `uvmcopy`
  - 새로운 memory를 할당하고, parent의 memory를 복사하는 것을 제거한다.
  - fork()를 호출한 process가 공유 페이지가 아니라면 cow bit를 세팅하고 PTE_W = 0으로 세팅한다.
  - Child process의 page table이 parent의 physical address를 가리키도록 한다.
- `Max page number` :
  - 각 physical page마다 refCount를 관리해야 하므로, 배열의 최대 사이즈는 PHYSTOP / PGSIZE로, physical memory size를 page size로 나눈 값을 사용한다.
- `reference count`
  - 각 Page마다 reference count가 관리되어야 한다.
  - Max page number만큼의 배열로 관리하자.
- `COW bit`
  - RSW bit를 이용하며, 9번째 (index: 8) bit가 1이라면 COW이다.
- `usertrap`
  - User가 write를 시도할 때, COW_bit가 set 되어 있고, PTE_W bit가 0이라면 새로운 page를 할당 후 write 해야 한다.
- `copyout`
  - Kernel이 write를 시도할 때, COW_bit가 set 되어 있고, PTE_W bit가 0이라면 새로운 page를 할당 후 write 해야 한다.

IMPLEMENTATION

`riscv.h`

```
#define PTE_COW (1L << 8)
```

- Cow bit를 index가 8인 bit로 관리하기 위해 PTE_W, PTE_R과 같은 방식으로 설정한다.

uvmcopy

```
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){
    // Logical address에 해당하는 PTE를 찾음
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    // PTE 존재하고 유효한지 확인
    if((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");

    // pte의 write
    *pte = *pte & ~PTE_W;
    *pte = *pte | PTE_COW;

    // 해당 PTE에 맞는 physical address를 가져옴
    pa = PTE2PA(*pte);
    // Parent PTE의 권한을 가져옴
    flags = PTE_FLAGS(*pte);

    // page를 자식 page table에 매핑
    if(mappages(new, i, PGSIZE, pa, flags) != 0){
      goto err;
    }

    // 기존 Physical page reference count 증가
    incRefCount((void*)pa);
  }
  return 0;

 err:
```

```
  // i: 0 ~ PGSIZE
  // 중간에 복사 실패 시, 이전까지 매핑한 페이지 해제
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

- 먼저 PTE가 유효한지 확인한다.
- uvmcopy()를 호출한 부모 process가 공유 페이지가 아니라면, 공유 페이지가 되도록 세팅한다.
- parent process와 child process의 write 권한을 제거하고 cow bit를 세팅해야 한다.
  - 이를 위해 먼저 parent process의 pte의 write bit = 0, cow bit = 1로 세팅한다.
  - 위의 과정을 먼저 진행하고, child process의 pte는 parent process의 pte를 복사하여 사용한다.
- 부모의 physical page를 자식 page table에 매핑한다.

copyout

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

  while(len > 0){
    // Page 단위 정렬
    va0 = PGROUNDDOWN(dstva);
    if(va0 >= MAXVA)
      return -1;
    // va0에 맞는 PTE 가져옴
    pte = walk(pagetable, va0, 0);
    // PTE 유효한 지 확인 + Write 권한 여부 확인
    if(pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0) {
      return -1;
    }
```

```
if ((*pte & PTE_W) == 0 && (*pte & PTE_COW)) {
  pa = PTE2PA(*pte);
  flags = PTE_FLAGS(*pte);
  flags = flags | PTE_W; // write bit set
  flags = flags & ~PTE_COW; // cow bit 제거

  if((mem = kalloc()) == 0) {
    printf("kalloc is failed\n");
    return -1;
  }

  // 새롭게 할당한 매모리에 우선 기존 pa에 해당되는 메모리에 존재하는 값을 옮김
  memmove(mem, (char*)pa, PGSIZE);
  // 기존 va0과 이전 pagetable의 매핑을 해제
  uvmunmap(pagetable, va0, 1, 0);
  // 기존 Physical page reference count 감소
  // refCount = 0 인 경우에 할당 해제
  kfree((void*)pa);

  // 새로운 페이지 매핑
  if (mappages(pagetable, va0, PGSIZE, (uint64)mem, flags) != 0) {
    kfree(mem);
    printf("New page mapping failed\n");
    return -1;
  }
  // page table을 변경했으니 새로운 pte를 사용하도록
  pte = walk(pagetable, va0, 0);
}

// PTE에 맞는 physical address 가져옴
pa0 = PTE2PA(*pte);
// Page에서 남은 공간
n = PGSIZE - (dstva - va0);
if(n > len)
  n = len;
// Kernel space data src를 va에 매핑되는 physical memory로 옮김
// pa0 + (dstva - va0): page allign x → 차이만큼 앞에서 시작
```

```
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }

  return 0;
}
```

- COW인 경우 기존 copyout과 다르게 동작해야 하기 때문에 COW인지 확인한다.
- Kernel에서 write하고자 하는 페이지를 공유 페이지가 아니라 전용 페이지로 만들기 위해, COW bit를 0으로 세팅하고 PTE_W를 1로 세팅했다.
- 기존 fork()처럼 kalloc()으로 새로운 메모리 페이지를 할당하고, 기존 pa에 해당하는 페이지의 메모리 내용을 복사한다.
- 기존에 child process의 va가 parnet process에 매핑되어있던 것을 해제한다.
- kfree()를 이용해 RefCount를 감소하고 refCount가 0인 경우 할당 해제한다.
- page table이 새롭게 할당된 memory 공간을 가리키도록 한다.

usertrap

```
void
usertrap(void)
{
  uint64 va, pa, flags;
  char *mem;
  pte_t *pte;
  int which_dev = 0;

  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();
  // system call 호출
  syscall();
}
else if((which_dev = devintr()) != 0){
  // ok
}
else if (r_scause() == 15) {
  va = PGROUNDDOWN(r_stval());

  pte = walk(p->pagetable, va, 0);

  if (pte == 0 || (*pte & PTE_V) == 0) {
    printf("Cannot find a pagetable\n");
    setkilled(p);
  }
  else if ((*pte & PTE_COW) && (*pte & PTE_W) == 0) {
    pa = PTE2PA(*pte);

    flags = PTE_FLAGS(*pte);
    flags = flags | PTE_W; // write bit set
```

```
      flags = flags & ~PTE_COW; // cow bit 제거

      // kalloc에서 refCount = 1까지 수행
      if((mem = kalloc()) == 0) {
        printf("usertrap: kalloc is failed\n");
        setkilled(p);
      }

      // 새롭게 할당한 매모리에 우선 기존 pa에 해당되는 메모리에 존재하는 값을 옮김
      memmove(mem, (char*)pa, PGSIZE);
      // 기존 va와 이전 pagetable의 매핑을 해제
      uvmunmap(p->pagetable, va, 1, 0);
      // 기존 Physical page reference count 감소
      // refCount = 0인 경우에 할당 해제
      kfree((void*)pa);

      // 새로운 페이지 매핑
      if (mappages(p->pagetable, va, PGSIZE, (uint64)mem, flags) != 0) {
        kfree(mem);
        panic("New page mapping failed\n");
      }
    }
    else { // page fault인데 cow가 아닌 경우
      printf("Non-cow fork occurs cow page fault\n");
      setkilled(p);
    }
  }

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

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();
  // User space로 복귀
  usertrapret();
}
```

- r_scause() == 15: trap의 원인이 store page fault(write)인지 확인한다.
- r_stval()을 통해 trap이 발생한 va를 확인한다.
- Cow일 시, vm.c의 copyout와 동일하게 동작해야 한다.
  - copyout에서 구현한 방식에 usertrap에 맞게 일부 수정 및 추가하였다.
- pte가 유효하고, 공유 페이지이며 write가 불가능한지 확인한다.
- 공유 페이지를 전용 페이지로 만들기 위해서, COW bit를 0으로 만들고, PTE_W bit로 세팅한다.
- kalloc()으로 새로운 메모리 페이지를 할당하고, 기존 pa에 해당하는 페이지의 메모리 내용을 복사한다.
- 기존 va가 page table에 매핑된 것을 해제한다.
- kfree()를 이용해 RefCount를 감소하고 refCount가 0인 경우 할당 해제한다.
- page table이 새롭게 할당된 memory 공간을 가리키도록 한다.

kalloc.h

```
#ifndef _KALLOC_H_
#define _KALLOC_H_

uint64 getRefCount(void* pa);
void incRefCount(void* pa);
void decRefCount(void* pa);

#endif // _KALLOC_H_
```

- uvmalloc에서 refCount를 incRefCount()를 이용하여 늘리고, kalloc.c 내에서 refCount를 관리하는 함수를 추가하기 위해 kalloc.h 헤더 파일을 만들었다.

kalloc.c

```
uint64 refCount[PHYSTOP / PGSIZE];
struct spinlock ref_lock;

uint64 getRefCount(void *pa) {
```

```
  int c;

  acquire(&ref_lock);
  c = refCount[(uint64)pa / PGSIZE];
  release(&ref_lock);

  return c;
}

void incRefCount(void *pa) {
  acquire(&ref_lock);
  refCount[(uint64)pa / PGSIZE]++;
  release(&ref_lock);
}

void decRefCount(void *pa) {
  acquire(&ref_lock);
  refCount[(uint64)pa / PGSIZE]--;
  release(&ref_lock);
}

void
kinit()
{
  initlock(&ref_lock, "refLock");
  initlock(&kmem.lock, "kmem");
  memset(refCount, 0, sizeof(refCount));
  freerange(end, (void*)PHYSTOP);
}
```

- 각 page가 refCount를 관리하게 만들고, refCount에 여러 process가 동시에 접근하지 못하게 lock을 구현한다.
  - lock을 초기화하기 위해 kinit()에 initlock(&ref_lock, "refLock")을 추가한다.
  - memset(refCount, 0, sizeof(refCount))을 통해 refCount 배열을 초기화한다.
- refCount를 늘리고, 줄이고, index에 맞는 값을 확인하기 위한 함수를 구현한다.

kalloc

```
void *
kalloc(void)
{
  // free page가 linked list로 관리됨
  // run은 linked list의 Node
  struct run *r;

  acquire(&kmem.lock);
  // freelist의 가장 첫 번째 page를 가져옴
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r) {
    acquire(&ref_lock);
    refCount[(uint64)r / PGSIZE] = 1;
    release(&ref_lock);
    // page 전체를 0x5로 채움
    memset((char*)r, 5, PGSIZE); // fill with junk
  }
  return (void*)r;
}
```

- 사용할 freelist의 첫 번째 page의 refCount를 1으로 초기화한다.
  - kalloc()은 process 하나에 page를 할당해주는 것이기 때문이다.

kfree

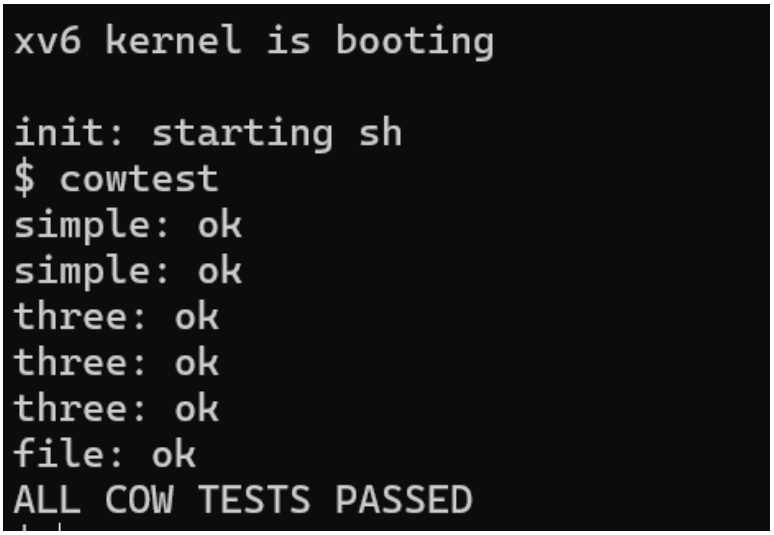
```
void
kfree(void *pa)
{
  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
      panic("kfree");
```

```
  if (getRefCount(pa) >= 1) decRefCount(pa);

  if (getRefCount(pa) == 0) {
    struct run *r;

    // Fill with junk to catch dangling refs.
    // 0x1로 memory를 채움
    memset(pa, 1, PGSIZE);

    r = (struct run*)pa;

    // pa에 해당하는 page를 freelist에 올림
    acquire(&kmem.lock);
    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
  }
}
```

- 공유 페이지가 아니라면 kfree()를 해야 하는 상황에서, refCount가 0이 아니면 실제로 page를 할당 해제하지 않아야 하므로, refCount를 감소하게 만든다.
- refCount가 0이라면 page를 할당 해제 한다.

RESULT

Test Code 실행 결과, 아래 사진과 같이 명세서의 결과와 동일하게 나왔다.



Simple test

- 굉장히 큰 메모리를 할당한 이후, fork()하여 cow fork() 방식이 아니면 메모리 공간으로 인하여 동작하지 않도록 한다.
- 정상적으로 실행되는 것으로 보아 cow fork() 방식대로 메모리를 공유하고 있는 것을 확인할 수 있다.

Three test

- COW fork() 이후 process가 write할 때, 자신만의 메모리 공간(page)에 write 할 수 있는지 확인한다.
- 정상적으로 실행되는 것을 보아, process가 write할 때 자신만의 메모리 공간을 가짐을 확인할 수 있다.

File test

- fork() 후에 copyout을 시도했을 때, 자신만의 메모리 공간(page)에 write 할 수 있는지 확인한다.
- 정상적으로 실행되는 것을 보아, process가 copyout할 때 자신만의 메모리 공간을 가짐을 확인할 수 있다.

TROUBLESHOTING

1. Shell 이 뜨지 않는 문제가 발생하였다.

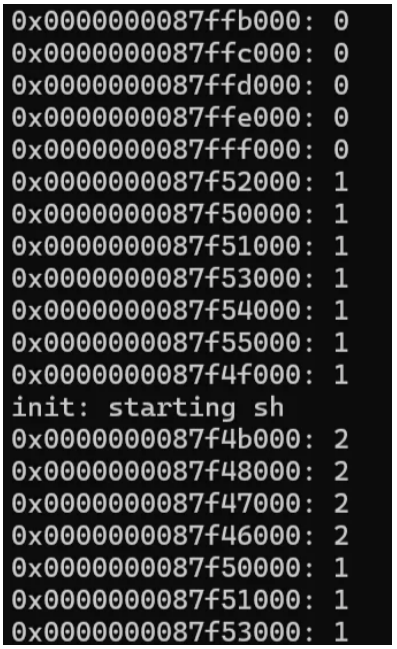
a. 디버깅 중, kfree에서 refCount 값이 -1로 무한 반복되는 것을 확인할 수 있었다.

b. 내 코드 구조 상, refCount는 항상 0보다 크거나 같아야 하는데 -1이 출력하는 이유를 확인할 수 없었다.

c. 기존에 별다른 확인 없이 refCount를 감소하던 코드를, refCount가 1보다 크거나 같은 경우에만 감소하도록 수정하였다.

d. (c)와 같이 수정하니 shell이 떴다.

e. 수정 후, 수정 전과 같은 디버깅 코드로 찍어보니 아래 사진과 동일하게 refCount = 0인 것이 수없이 출력 되었다.



f. init process를 생성할 때, kalloc()말고 다른 방식으로 memory를 할당하면 kalloc()에서와 같이 refCount = 1로 초기화하지 않기 때문에 발생한 문제라고 추정된다.

2. Remap 에러가 발생하였다.

a. 디버깅을 통해 `usertrap`이나 `copyout`에서 위 문제가 발생한 것을 확인할 수 있었다.

b. 두 코드에서는 새로운 메모리를 생성하고 매핑하는 작업을 수행한다.

c. 새로운 메모리를 기존 page table에 매핑할 때, 이전에 page table이 가리키던 것을 Unmapping 하지 않아서 발생한 문제임을 확인하고 uvmunmap 함수를 추가하여 해결하였다.

3. kfree() 구현 과정에서 kernel trap 발생
    a. 처음 구현 시, `copyout`과 `usertrap` 에서 decRefCount 함수를 이용하여 refcount를 감소하고, 이후에 kfree로 refcount가 0이라면 할당 해제하도록 구현하였다.
    b. kfree 내부적으로 refCount를 감소시키는 로직이 없었다.
    c. (a), (b)와 같이 구현하니, 내가 수정한 copyout 이나 usertrap 외의 함수에서 kfree시 refCount를 감소시키는 로직이 따로 존재하지 않게 되었다.
    d. copyout과 usertrap 외 함수가 kfree를 호출해 refCount를 감소시키고 refCount가 0이 되어 할당 해제되어야 하는데, refCount가 줄어드는 로직이 kfree 내부나 copyout과 usertrap 외 함수 구현 내부에 존재하지 않아 할당 해제되지 않은 현상이 발생했다.
    e. (d)로 인하여 메모리 공간이 부족해져 세 번째 “three: “ 부분에서 kernel trap이 발생하였다.
    f. 이를 해결하기 위해 kfree 내부에 refCount를 감소하는 로직을 넣고, refCount가 감소하는 경우에도 kfree, 할당 해제해야 하는 경우에도 kfree를 호출하도록 구현하였다.

---

2. Big files

PREVIEW

기존 12개의 Direct blocks와 1개의 Indirect block 형식 대신, 11개의 Direct blocks와 1개의 Indirect block, 1개의 doubly indirect block을 만들어서 가능한 최대 파일 크기를 증가하는 프로젝트

프로젝트를 진행하기 위해 기존 구현을 살펴보자

`FSSIZE` : File system 속 전체 Block의 개수

```
#define FSSIZE       2000  // size of file system in blocks
```

`NDIRECT` : Direct block의 개수

```
#define NDIRECT 12
```

`MAXFILE` : 최대 파일 Block 수

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

- BSIZE: Block의 Byte 수
- 한 Block에서 uint형 주소를 가질 수 있는 만큼의 수가 NIDIRECT가 가지는 Block의 개수이다.

`addrs[]` : inode struct가 직접 관리하는 block의 address

```
// 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+1];   // Data block addresses
};
```

- Direct block + Indirect block의 개수만큼 inode struct에서 block을 직접 관리한다.

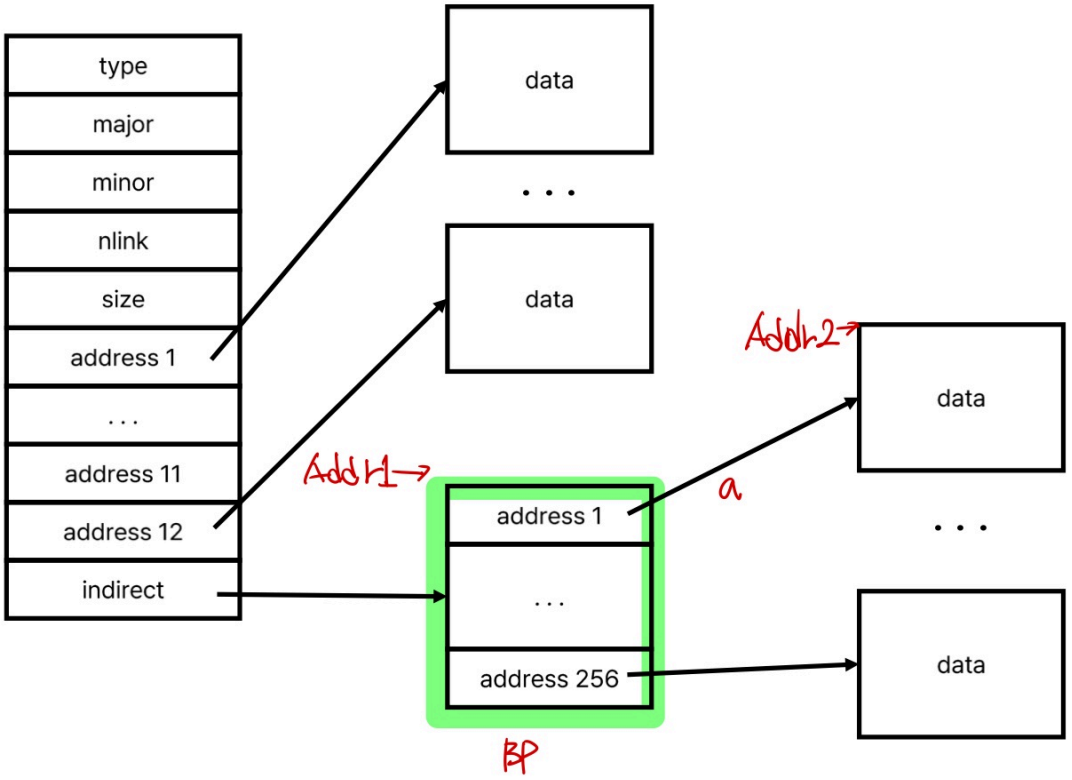
`bmap()`

```
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
```

```
struct buf *bp;
// bn이 Direct block 관련
if(bn < NDIRECT){
  // block이 실제로 할당되지 않았다면
  if((addr = ip->addrs[bn]) == 0){
    // block을 할당
    addr = balloc(ip->dev);
    if(addr == 0)
      return 0;
    ip->addrs[bn] = addr;
  }
  return addr;
}
bn -= NDIRECT;
// bn이 Indirect block이 관리하는 block
if(bn < NINDIRECT){
  // Load indirect block, allocating if necessary.
  if((addr = ip->addrs[NDIRECT]) == 0){
    // indirect block 할당
    addr = balloc(ip->dev);
    if(addr == 0)
      return 0;
    ip->addrs[NDIRECT] = addr;
  }
  bp = bread(ip->dev, addr); // Block을 읽어서 메모리로 가져옴
  a = (uint*)bp->data; // bp의 data를 uint pointer 배열로 변환
  if((addr = a[bn]) == 0){ // 실제 데이터에 해당하는 block이 비었다면
    addr = balloc(ip->dev);
    if(addr){
      a[bn] = addr;
      log_write(bp);
    }
  }
  brelse(bp);
  return addr;
}
```

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

- bn의 주소에 맞게 block이 없다면 할당하고 있다면 실제 주소를 반환한다.
- bn이 Direct block에 속하는 경우, Indirect block에 속하는 경우로 나누어 다르게 처리한다.
- Single Indirect block의 bmap 과정은 아래와 같다.



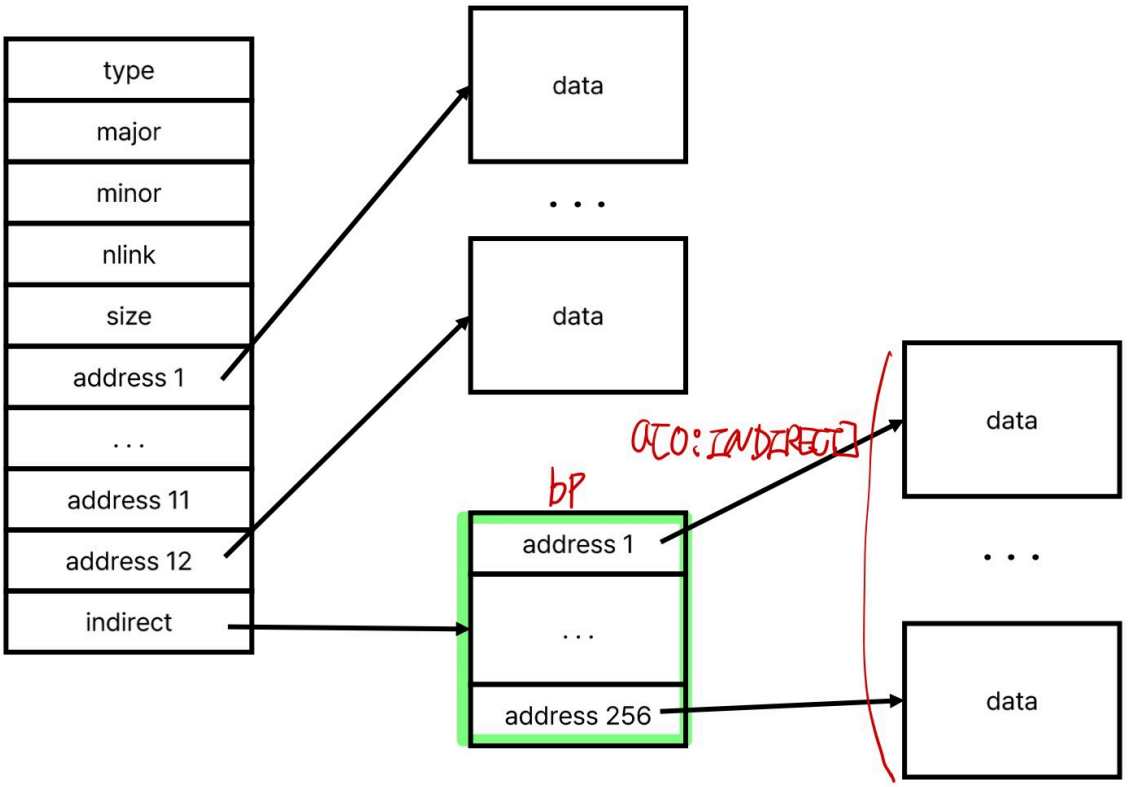
- inode[12]가 가리키는 block이 갖고 있는 data 배열 중, bn에 맞는 data의 시작 주소를 반환한다.

itrunc()

```
void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp;
```

```
  uint *a;
  // Direct block 할당 해제제
  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }

  // Indirect block이 관리하는 block을 할당 해제
  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;
  }

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

- indirect block이 가리키는 data block들을 할당 해제한다.
- inode의 indirect block까지 할당 해제한다.
- Single Indirect block의 itrunc 과정은 아래와 같다.



create()

```
static struct inode*
create(char *path, short type, short major, short minor)
{
  struct inode *ip, *dp;
  char name[DIRSIZ];
    // 부모 file을 지정
  if((dp = nameiparent(path, name)) == 0)
    return 0;

  ilock(dp);
    // 같은 이름의 파일이 있는지 확인
  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;
```

```
  }
    // 위의 조건 통과 시, inode allocation
  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;
}
```

- 새로운 inode를 할당한다.

`inode` : 메모리에 올라와 있는 inode struct

```
// 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+1];
};
```

DESIGN

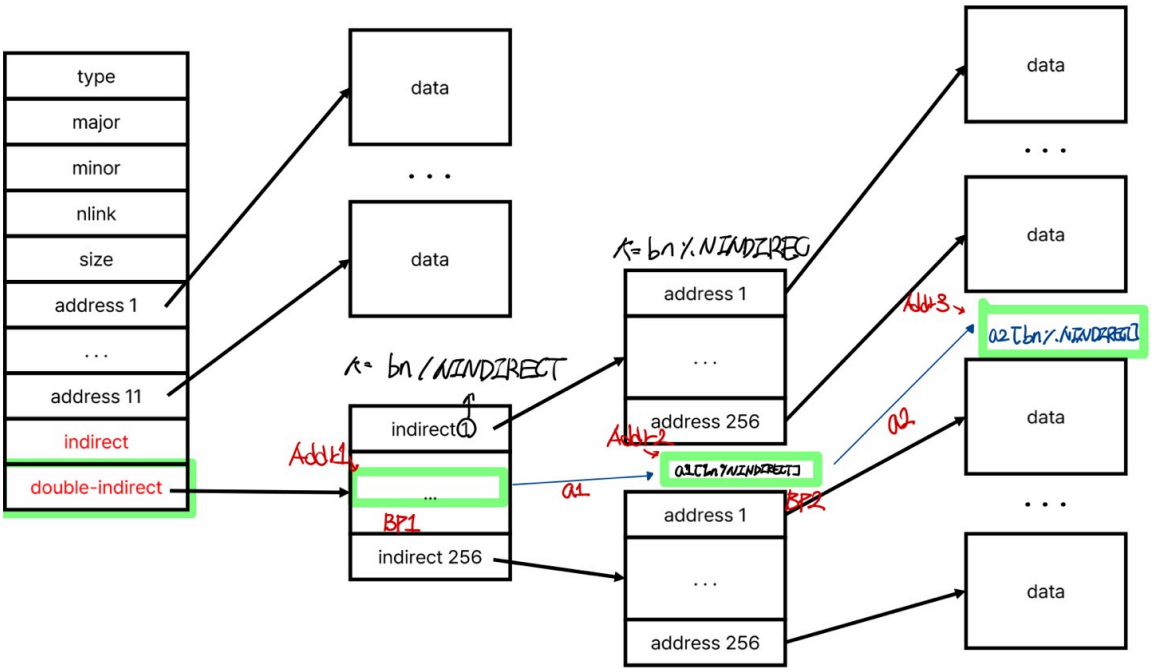
Single indirect block의 bmap, itrunc 동작 과정을 참고하여, Double indirect block를 구현한다.

Double indirect block에서는 single indirect block에서 한 층의 tree level이 추가되었기 때문에 이 부분을 신경 써서 구현한다.

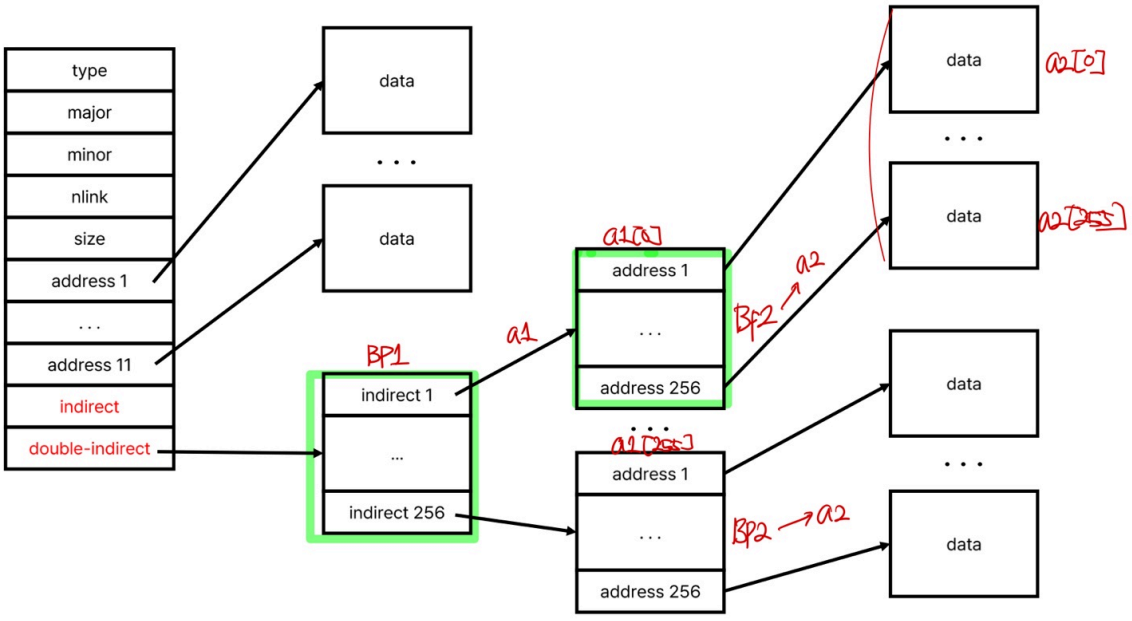
- Double indirect block은 Single indirect block과 다르게 중간에 각 block이 또 다른 NINDIRECT 개의 block을 가리킨다.

bmap, itrunc의 Double indirect block에서의 동작 과정은 아래와 같다.

1. `bmap`



2. itrunc



IMPLEMENTATION

명세서에 따른 수정

```
// 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];
};

// fs.h
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT + (NINDIRECT*NINDIRECT))

// 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
};
```

- 명세서에 따라 `MAXFILE`, `addrs array`, `NDIRECT`, `FSSIZE`를 수정하였다.

`bmap`

```
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, addr1, addr2, addr3, *a, *a1, *a2;
  struct buf *bp, *bp1, *bp2;
  // bn이 Direct block 관련
  if(bn < NDIRECT){
    // block이 실제로 할당되지 않았다면
    if((addr = ip->addrs[bn]) == 0){
      // block을 할당
      addr = balloc(ip->dev);
      if(addr == 0)
        return 0;
      ip->addrs[bn] = addr;
    }
    return addr;
  }
  bn -= NDIRECT;
  // bn이 Indirect block이 관리하는 block
  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0){
      // indirect block 할당
      addr = balloc(ip->dev);
      if(addr == 0)
        return 0;
      ip->addrs[NDIRECT] = addr;
    }
    bp = bread(ip->dev, addr); // Block을 읽어서 메모리로 가져옴
    a = (uint*)bp->data; // bp의 data를 uint pointer 배열로 변환
    if((addr = a[bn]) == 0){ // 실제 데이터에 해당하는 block이 비었다면
      addr = balloc(ip->dev);
      if(addr){
        a[bn] = addr;
        log_write(bp);
      }
    }
    brelse(bp);
```

```
  return addr;
}

bn -= NINDIRECT; // single indirect block이 아닌 경우 double inderect를 확인해

if(bn < NINDIRECT*NINDIRECT){
  // Load double-indirect block, allocating if necessary.
  if((addr1 = ip->addrs[NDIRECT+1]) == 0){
    // double-indirect block 할당
    addr1 = balloc(ip->dev);
    if(addr1 == 0)
      return 0;
    ip->addrs[NDIRECT+1] = addr1;
  }
  // Double-indirect block에서 첫 번째 level indirect block을 확인
  bp1 = bread(ip->dev, addr1); // Block을 읽어서 메모리로 가져옴
  a1 = (uint*)bp1->data; // bp의 data를 uint pointer 배열로 변환
  if((addr2 = a1[bn / NINDIRECT]) == 0){ // 첫 번째 indirect block 중 어디에 속하
    addr2 = balloc(ip->dev);
    if(addr2){
      a1[bn / NINDIRECT] = addr2;
      log_write(bp1);
    }
  }
  brelse(bp1);

  // Double-indirect block에서 두 번째 level indirect block을 확인
  bp2 = bread(ip->dev, addr2); // Block을 읽어서 메모리로 가져옴
  a2 = (uint*)bp2->data; // bp의 data를 uint pointer 배열로 변환
  if((addr3 = a2[bn%NINDIRECT]) == 0){ // 두 번째 indirect block 중 어디에 속ㅎ
    addr3 = balloc(ip->dev);
    if(addr3){
      a2[bn%NINDIRECT] = addr3;
      log_write(bp2);
    }
  }
  brelse(bp2);
```

```
    return addr3;
  }

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

- DESIGN의 그림과 동일하게 구현하였다.
- Single indirect block을 매핑 하는 부분 아래에 Double indirect block을 매핑하는 부분을 처리하였다.
- 중간 Indirect block 중 어느 것을 사용해야 하는지는 bn / NINDIRECT를 통해 확인한다.
- 실제 데이터를 가리키는 block은 bn * NINDIRECT를 통해 확인한다.

itrunc

```
void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp, *bp1, *bp2;
  uint *a, *a1, *a2;
  // Direct block 할당 해제
  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }

  // Indirect block이 관리하는 block을 할당 해제
  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;
  }

  // Double - indirect block 할당 해제
  if (ip->addrs[NDIRECT+1]){
    bp1 = bread(ip->dev, ip->addrs[NDIRECT+1]);
    a1 = (uint*)bp1->data;

    for (j = 0; j < NINDIRECT; j++) {
      if(a1[j]) { // 두 번째 indirect block에서 가리키는 direct block 할당 해제
        bp2 = bread(ip->dev, a1[j]);
        a2 = (uint*)bp2->data;

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

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

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

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

- 중첩 반복문을 통해 각 중간 Indirect block이 가리키는 NINDIRECT 개의 block을 할당 해제한다.
- 위 작업을 모든 중간 Indirect block에 대해 반복한다.

create

- inode, dinode 구조체에서는 addrs를 수정하였다.
- 그러나 이는 NDIRECT를 수정해서 계산값만 수정한 것이고, 수정하기 전과 배열의 크기는 동일하였다.
- 따라서 수정 전과 동일한 구조체를 사용하였기 때문에 create는 딱히 수정할 부분이 없었다.

RESULT

이 Test는 Direct, Single indirect, Double indirect를 통해 65803개의 block을 지원할 수 있는지 확인한다.

Test 결과는 아래와 같다.

```
xv6 kernel is booting

init: starting sh
$ bigfile
.....................................................................................................................
.....................................................................................................................
.....................................................................................................................
.....................................................................................................................
.....................................................................................................................
wrote 65803 blocks
bigfile done; ok
```

- 명세서와 동일하게 658개의 점이 찍히는 것으로 보아, Direct, Single indirect, Double indirect block이 정상적으로 동작함을 확인할 수 있다.
- 점 개수는 아래 파이썬 코드의 결과로 확인하였다.

TROUBLESHOTING

명세서에 제시되어 있는 순서에 따라 구현하다 보니 큰 어려움은 없었다.

그러나, test를 하기 전에 컴퓨터를 재부팅하거나 실행 중이던 다른 프로그램을 종료하지 않으면 실행 결과를 확인하는데 너무 오랜 시간이 걸리는 문제가 있었다.

---

3. Symbolic links

PREVIEW

이 프로젝트는 기존 xv6의 hard link 기반의 file system을 symbolic link 기반의 file system으로 수정하여, file system의 유연성을 늘릴 수 있도록 하는 프로젝트이다.

프로젝트를 진행하기 전에, xv6의 file system의 동작 과정과 관련된 함수와 Symbolic link 구현에 필요한 함수들의 기존 구현을 살펴보자.

1. `sys_open` : 사용자가 접근하고자 하는 path에 있는 파일을 새롭게 만들거나 열고, 해당 파일을 kernel이 tracking할 수 있도록 file 구조체를 연결한 뒤, file descriptor를 반환 한다.

```
uint64
sys_open(void)
{
  char path[MAXPATH];
  int fd, omode;
  struct file *f;
  struct inode *ip;
  int n;
  // 두 번째 argumnet로 omode
  argint(1, &omode);
```

```
// 첫 번째 argument로 path
if((n = argstr(0, path, MAXPATH)) < 0)
  return -1;

begin_op();
// O_CREATE가 있다면 해당 path에 새로운 file 생성
if(omode & O_CREATE){
  ip = create(path, T_FILE, 0, 0);
  if(ip == 0){ // inode 생성 실패 시
    end_op();
    return -1;
  }
} else {  // O_CREATE에 없다면 기존 file을 열기 위해 시도
  // path에 맞는 inode 확인
  // 이때 namei() 이용 - 기존 파일 open시 name 이용 한다.
  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  ilock(ip);
  // path에 맞춰 가져온 inode가 directory인지 확인
  // directory라면 Read-Only여야 한다.
  if(ip->type == T_DIR && omode != O_RDONLY){
    iunlockput(ip);
    end_op();
    return -1;
  }
}
// Device file: 외부 I/O device를 위한 파일
// Device file인 경우 major가 유효한 지 확인 필요
if(ip->type == T_DEVICE && (ip->major < 0 || ip->major >= NDEV)){
  iunlockput(ip);
  end_op();
  return -1;
}
// file 구조체와 file descriptor를 할당
// 현재 file이 어떤 inode 구조체를 확인하고 있는지 표시한다.
if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
```

```
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }
  // kernel의 file이 현재 실제 inode를 추적하기 위한 정보 제공
  if(ip->type == T_DEVICE){
    f->type = FD_DEVICE;
    f->major = ip->major;
  } else {
    f->type = FD_INODE;
    f->off = 0;
  }
  // 이 부분이 Hard link
  // 여러 file 구조체가 같은 inode를 가리키게 된다.
  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;
}
```

2. `namex` : User가 path로 접근하고자 하는 file을 실제로 찾아줌

```
static struct inode*
namex(char *path, int nameiparent, char *name)
{
  struct inode *ip, *next;
```

```
  if(*path == '/') // 절대 경로인 경우에 root directory부터 시작
    ip = iget(ROOTDEV, ROOTINO);
  else // 상대 경로인 경우, 현재 diretory부터 시작
    ip = idup(myproc()->cwd);

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

struct inode*
namei(char *path)
{
  char name[DIRSIZ];
  return namex(path, 0, name);
}
```

3. create

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

  // path에 맞는 부모 directory를 가져옴
  if((dp = nameiparent(path, name)) == 0)
    return 0;

  ilock(dp);
  // 부모 directory에 만들고자 하는 파일과 같은 이름을 가진 파일이 이미 존재하는지
  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;
  }
  // 같은 이름의 파일이 없을 때, allocation
  if((ip = ialloc(dp->dev, type)) == 0){
    iunlockput(dp);
    return 0;
  }

  ilock(ip);
  ip->major = major;
  ip->minor = minor;
  ip->nlink = 1; // 기본 Link
  iupdate(ip);
  // .은 현재 file에 link하고 ..은 부모 directory에 link
  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;
  }
  // 부모 directory에 현재 inode link
  if(dirlink(dp, name, ip->inum) < 0)
    goto fail;
  // 생성할 파일이 directory인 경우 부모 directory에 nlink++
  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;
}
```

4. dirlink

```
int
dirlink(struct inode *dp, char *name, uint inum)
{
  int off;
  struct dirent de;
  struct inode *ip;

  // Check that name is not present.
  // 한 directory 내에 같은 이름의 파일 존재 불가
  if((ip = dirlookup(dp, name, 0)) != 0){
```

```
    iput(ip);
    return -1;
  }

  // Look for an empty dirent.
  // 빈 directory entry가 있는지 확인
  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      break;
  }
  // Directory entry에 쓸 entry를 만들어 놓음
  strncpy(de.name, name, DIRSIZ);
  de.inum = inum;
  // 찾은 directory entry에  write
  if(writei(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
    return -1;

  return 0;
}
```

5. sys_link

```
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();
  // 기존 file의 inode 확인
  if((ip = namei(old)) == 0){
    end_op();
```

```
    return -1;
  }

  ilock(ip);
  // ip가 directory이면 무시
  if(ip→type == T_DIR){
    iunlockput(ip);
    end_op();
    return -1;
  }
  // link 수 증가
  ip→nlink++;
  iupdate(ip);
  iunlock(ip);
  // new의 부모 directory를 찾음
  if((dp = nameiparent(new, name)) == 0)
    goto bad;
  ilock(dp);
  // 같은 file system에 존재하는지 확인한다.
  // dirlink로 새로운 directory entry 생성하고 같은 inode num을 가리키도록 설정
  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;
}
```

기존 `Hard link`의 동작 방식은 아래와 같다.

- Hard link의 경우 다른 path가 같은 inode를 가리킬 수 있도록 하는 방식이다.
- 이를 위해 동일한 file system 내에서 부모 directory에서 다른 name이 같은 inode를 가리킬 수 있도록 해야 한다.
- 이를 위해 sys_link 내에서 old의 inode를 찾고, new의 부모 directory 속 new가 old의 inode를 가리키도록 구현해야 한다.

DESIGN

기존 `Hard link` 기반 코드를 `Symbolic link` 기반으로 동작하도록 구현해야 한다.

`Symbolic link` 는 아래와 같은 특징을 갖는다.

- 하나의 inode가 path를 가리킨다.
- 하나의 inode를 open하였을 때, 원본 file이 아니라 symbolic link에 해당하는 경우 실제 file을 담고 있는 inode를 찾아야 한다.

이를 위해선 sys_link와 비슷하게 sys_symlink를 구현하되, create를 통해 새로운 inode를 생성하고 path를 가리키도록 해야 한다.

또한 파일은 open 할 때, symbolic link 라면, 원본 파일를 recursively 하게 찾아와야 한다.

- 특정 inode가 원본 파일인지, Symbolic link인지 구별하기 위해 `T_SYMBOLIC`을 사용한다.

IMPLEMENTATION

1. `T_SYMBOLIC / O_NOFOLLOW`

```
#define T_SYMLINK 4   // Symbolic link

#define O_NOFOLLOW 0x1000
```

- O_NOFOLLOW는 이전 값이 0x400 인 것을 보고 넉넉하게 0x800이 아닌 0x1000을 사용하였다.

2. `sys_symlink`

```
uint64
sys_symlink(void)
{
  char target[MAXPATH], path[MAXPATH];
  int n; // argument 정상적으로 받아오는지 확인용
  struct inode* ip;

  if((n = argstr(0, target, MAXPATH)) < 0)
    return -1;

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

  begin_op();
  // symbolic link는 같은 path를 가리키는 새로운 파일을 만들어야한다.
  // 또는 같은 이름의 link 존재하면 그 link 사용
  // create 사용
  ip = create(path, T_SYMLINK, 0, 0);
  if (ip == 0) {
    printf("create is failed\n");
    end_op();
    return -1;
  }

  // 새로운 파일을 생성하므로 nlink를 따로 관리 하지 않는다.
  // create에서 nlink = 1
  // 기존 inode 사용하는 경우에는 nlink++ 필요 없다.
  // path기반이므로 directory를 가리켜도 상관없다.
```

```
  // Symbolic link는 같은 file system인지는 확인하지 않아도 된다.
  // dirlink 대신 이 작업 수행
  // ip가 target path를 가리키도록 한다.
  if(writei(ip, 0, (uint64)target, 0, sizeof(target)) != sizeof(target)) {
    printf("writei is failed\n");
    iunlockput(ip);
    end_op();
    return -1;
  }

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

- sys_link를 참고하여 구현하였다.
- DESIGN에 작성한 사항에 맞게 sys_link 함수를 수정하거나 추가하였다.
- 더욱 자세한 것은 주석을 통해 확인할 수 있다.

3. create

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

  // path에 맞는 부모 directory를 가져옴
  if((dp = nameiparent(path, name)) == 0)
    return 0;

  ilock(dp);
  // 부모 directory에 만들고자 하는 파일과 같은 이름을 가진 파일이 이미 존재하는지
  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;
  // 이미 같은 이름의 symbolic link가 존재하는 경우
  if (type == T_SYMLINK) {
    return ip;
  }
  iunlockput(ip);
  return 0;
}
// 같은 이름의 파일이 없을 때, allocation
if((ip = ialloc(dp->dev, type)) == 0){
  iunlockput(dp);
  return 0;
}

ilock(ip);
ip->major = major;
ip->minor = minor;
ip->nlink = 1; // 기본 Link
iupdate(ip);

// .은 현재 file에 link하고 ..은 부모 directory에 link
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;
}
// 부모 directory에 현재 inode link
if(dirlink(dp, name, ip->inum) < 0)
  goto fail;
// 생성할 파일이 directory인 경우 부모 directory에 nlink++
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_link에서 현재 확인 해야하는 inode가 T_SYMBOLIC임을 명시했다.
- create에서는 name이 충돌하는 경우만 아니라면, 기존 create 방식을 그래도 사용하고 sys_link에서 writei 하도록 구현하였다.
- name이 충돌하는 경우, 해당 inode가 T_SYMBOLIC이라면 그냥 해당 inode를 반환하도록 구현하였다.

4. `sys_open`

```
uint64
sys_open(void)
{
  char path[MAXPATH], target[MAXPATH];
  int fd, omode;
  struct file *f;
  struct inode *ip;
  int n, depth = 0;
  // 두 번째 argumnet로 omode
  argint(1, &omode);
  // 첫 번째 argument로 path
  if((n = argstr(0, path, MAXPATH)) < 0)
    return -1;

  begin_op();
  // O_CREATE가 있다면 해당 path에 새로운 file 생성
  if(omode & O_CREATE){
```

```
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){ // inode 생성 실패 시
      end_op();
      return -1;
    }
  } else {  // O_CREATE에 없다면 기존 file을 열기 위해 시도
    // path에 맞는 inode 확인
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);
    // path에 맞춰 가져온 inode가 directory인지 확인
    // directory라면 Read-Only여야 한다.
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      end_op();
      return -1;
    }
  }
  // symbolic link라면 path가 아닌 실제 inode를 찾아야 한다.
  // symbolic link를 타고 재귀적으로 찾아야 한다.
  if (ip->type == T_SYMLINK) {
    while (1) {
      // depth: cycle 방지
      // O_NOFOLLOW: 이 flag 세팅 시, target으로 이동하면 안 된다.
      if (ip->type != T_SYMLINK || (omode & O_NOFOLLOW) || depth >= 10) {

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

      iunlockput(ip);
      // 다음 loop를 위해 ip update
      if ((ip = namei(target)) == 0) {
        end_op();
```

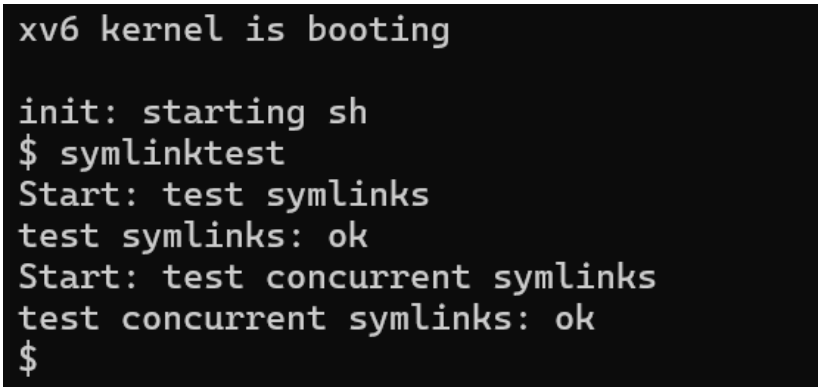
```
      return -1;
    }
    depth ++;
    ilock(ip);
  }

  if (depth >= 10) {
    iunlockput(ip);
    end_op();
    return -1;
  }
}

// Device file: 외부 I/O device를 위한 파일
// Device file인 경우 major가 유효한 지 확인 필요
if(ip→type == T_DEVICE && (ip→major < 0 || ip→major >= NDEV)){
  iunlockput(ip);
  end_op();
  return -1;
}

// file 구조체와 file descriptor를 할당
// 현재 file이 어떤 inode 구조체를 확인하고 있는지 표시한다.
if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
  if(f)
    fileclose(f);
  iunlockput(ip);
  end_op();
  return -1;
}
// kernel의 file이 현재 실제 inode를 추적하기 위한 정보 제공
if(ip→type == T_DEVICE){
  f→type = FD_DEVICE;
  f→major = ip→major;
} else {
  f→type = FD_INODE;
  f→off = 0;
}
```

```
  // 이 부분이 Hard link
  // 여러 file 구조체가 같은 inode를 가리키게 된다.
  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;
}
```

- open하고자 하는 inode가 T_SYMBOLIC이라면 원본 파일이 존재하는 위치까지 recursively하게 확인해야 한다.
- inode가 T_SYMBOLIC인 경우, 무한 루프를 돌면서 T_SYMBOLIC이 아닌 inode가 나오거나 depth가 10보다 작을 때까지 찾도록 구현하였다.

RESULT

명세서의 결과와 동일한 출력 결과를 얻었다.



- `testsymlink()` 가 정상적으로 실행된 것을 보아 기본적인 Link 생성과 접근, Broken link 처리, Circular reference detection, 존재하지 않는 file linking, chain link 해결 모두 정상적으로 동작함을 확인할 수 있다.

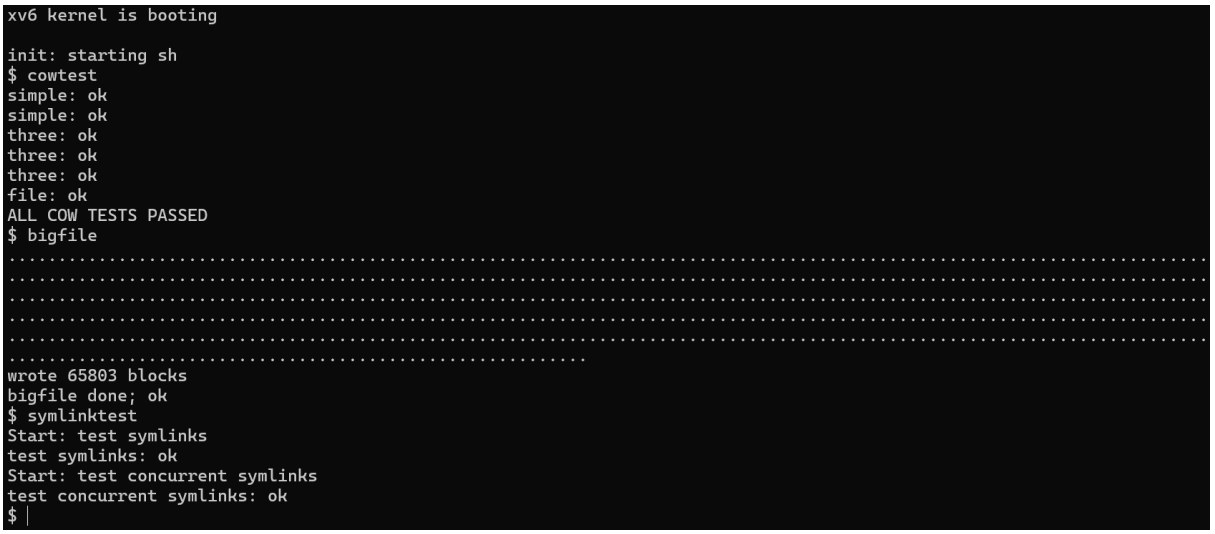
- `concur()`이 정상적으로 실행된 것을 보아같은 symbolic link에 한 동시 다발적인 접근도 처리하는 것을 확인할 수 있다.

TROUBLESHOTING

`testsymlink` 에서 “failed to open b” 메세지가 계속 출력되었다.

- 디버깅 결과, `sys_open` 속에 T_SYMLINK 처리하는 코드 내에서 namei()가 실패하는 것을 확인했다.
- Test code를 확인하면 b를 open 할 때, b→a (Symbolic link) 상태임을 확인할 수 있다.
- 하지만 a는 T_SYMBOLIC이 아니다.
- 기존 내 코드에서 while(1) 내부에 추가적으로 현재 ip(a)의 type이 T_SYMLINK인지 확인하는 코드가 없어 a가 T_SYMBOLIC이 아님에도 불구하고 namei를 호출하여 문제가 발생한 것이었다.
- while(1) 내부에서 무한 루프를 종료할 조건에 현재 ip의 type이 T_SYMLINK인지 확인하는 코드인지 확인하는 조건을 추가하여 해결하였다.

Total results



- `cowtest`, `bigfile`, `symlinktest` 순서로 실행해도 정상적으로 실행되는 것을 확인할 수 있다.