Operating System <Project03>

2023017856 한진호

1. Copy-on-Write

```
struct {
  struct spinlock lock;
  int ref[PHYSTOP / PGSIZE];
} page_ref_counts;
```

프로세스가 만들어졌을 당시에는 parent의 page를 공유하고 있어야 한다.

각 페이지를 공유하는 프로세스의 수를 알기 위해 Reference Counting 와 이를 보호하기 위한 lock을 member variable로 가지는 struct를 구성하였다.

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

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

  if(r){
    memset((char*)r, 5, PGSIZE); // fill with junk
    acquire(&page_ref_counts.lock);
    page_ref_counts.ref[(uint64)r / PGSIZE] = 1;
    release(&page_ref_counts.lock);
  }
  return (void*)r;
}
```

kalloc이 invoke 된 상황은 allocproc을 통해 방금 만들어진 상황일 것이다. 이 상황에서 우리는 COW가 되길 원하므로 page_ref_counts 의 ref 배열 내에 1이라는 값을 적어 이 프로세스가 parent 프로세스의 page 를 공유하고 있는 상황임을 알린다.

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

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

  acquire(&page_ref_counts.lock);
  if(--page_ref_counts.ref[(uint64)pa / PGSIZE] > 0){
    release(&page_ref_counts.lock);
    return;
  }
  page_ref_counts.ref[(uint64)pa / PGSIZE] = 0;
  release(&page_ref_counts.lock);

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

  r = (struct run*)pa;

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

page_ref_counts.ref 를 봤는데 0이 아니라면 현재 parent 와 공유하는 page가 있다는 뜻이다. 진짜 free가 되어야 하는 프로세스는 실행되면서 애초에 parent와 분리되었기에 ref가 0이였을 것이다. 따라서 이와 같은 경우는 free 시켜주는 것이 아닌 ref 를 0으로 설정해주고 끝나도록 하였다.

```
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");

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

    incref((void*)pa);

    // 쓰기 권한을 제거하고 COW 플래그를 설정
    flags = (flags & ~PTE_W) | PTE_COW;

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

    // 부모의 PTE도 읽기 전용으로 변경
    *pte = (*pte & ~PTE_W) | PTE_COW;
  }
  return 0;

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

parent의 page (PA)에 대해 incref()를 호출하여 reference count를 증가시킨다. 자식의 페이지 테이블에 부모와 동일한 PA를 mapping 하였다. 이후에 write를 시도한다면 page

fault를 발생시켜 이를 통해 page를 분리시키기 위해 parent, child 의 PTE에서 PTE_W 를 제거하고, PTE_COW 를 설정하였다.

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

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

  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call
    if(killed(p))
      exit(-1);
    p->trapframe->epc += 4;
    intr_on();
    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if(r_scause() == 15) { // Store/AMO page fault
    uint64 va = r_stval();
    pte_t *pte;

    if(va >= p->sz || (pte = walk(p->pagetable, va, 0)) == 0 || (*pte & PTE_U) ==
      setkilled(p);
    } else if((*pte & PTE_COW) == 0) {
      setkilled(p);
    } else {
      uint64 pa = PTE2PA(*pte);
      uint flags = PTE_FLAGS(*pte);
```

```
      char *mem;

      if((mem = kalloc()) == 0){
        setkilled(p);
      } else {
        memmove(mem, (char*)pa, PGSIZE);
        uvmunmap(p->pagetable, PGROUNDDOWN(va), 1, 1);
        if(mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)mem,
          kfree(mem);
          setkilled(p);
        }
      }
    }
  } else {
    printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pid
    printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
    setkilled(p);
  }

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

  if(which_dev == 2)
    yield();

  usertrapret();
}
```

if (r_scause() == 15)로 write에 의해 발생하는 page fault 인지를 확인하도록 하였다. PTE의 flag가 PTE_COW 인지 확인하여 COW 일 때 fault와 다른 상황에서의 page fault를 구별하도록 하였다. COW 일 때 fault가 맞다면, kalloc()으로 새로운 page를 할당하고, memmove로 원본 페이지 내용을 복사한 후, mappages 를 통해 새 page에 매핑하였다.

```
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)
    return -1;

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

    uint64 pa = PTE2PA(*pte);
    uint flags = PTE_FLAGS(*pte);
    char *mem;

    if((mem = kalloc()) == 0)
      return -1;

    memmove(mem, (char*)pa, PGSIZE);
    uvmunmap(pagetable, va0, 1, 1);
    if(mappages(pagetable, va0, PGSIZE, (uint64)mem, (flags & ~PTE_COW) |
      kfree(mem);
      return -1;
    }
  }

  pa0 = PTE2PA(*walk(pagetable, va0, 0));
  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;
}
```

read 와 같은 시스템 콜이 커널 공간에서 user의 공유된 COW 페이지로 데이터를 쓸 때 문제가 생기지 않도록 하기 위해 목적지인 페이지가 쓰기 불가능할 경우, PTE_COW flag가 있는가 확인하는 로직을 추가하였다. 만약 COW 페이지라면, 아래의 usertrap()과 동일한 방식으로 페이지를 복사하여 write가 가능한 페이지로 만들어준다.

```
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
```

2. Large files

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT + NINDIRECT*NINDIRECT)

struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses: 11 direct, 1 indirect, 1 doub
};
```

double indirect block pointer를 추가하기 위해 기존 12개의 직접 포인터 중 하나를 double indirect pointer용 공간으로 할당해야 하므로 NDIRECT 를 12에서 11로 바꿨으며 addrs 배열 크기는 1 증가 시켰다.

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

더 큰 파일을 저장하기 위해서는 file system 전체의 사용 가능한 블록 수도 증가해야 하기에 FSSIZE 를 2000에서 20000으로 늘려 주었다.

```
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;
}
bn -= NINDIRECT;

if(bn < NINDIRECT * NINDIRECT){
  // Doubly-indirect block
  uint *a2;
  struct buf *bp2;

  // Load the doubly-indirect block
  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;

  // Load the singly-indirect block from the doubly-indirect block
  if((addr = a[bn / NINDIRECT]) == 0){
    addr = balloc(ip->dev);
    if(addr == 0){
      brelse(bp);
      return 0;
    }
    a[bn / NINDIRECT] = addr;
    log_write(bp);
  }
  brelse(bp);
```

```
    // Load the data block from the singly-indirect block
    bp2 = bread(ip->dev, addr);
    a2 = (uint*)bp2->data;
    if((addr = a2[bn % NINDIRECT]) == 0){
      addr = balloc(ip->dev);
      if(addr){
        a2[bn % NINDIRECT] = addr;
        log_write(bp2);
      }
    }
    brelse(bp2);
    return addr;
  }

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

file이 single indirect block의 용량을 초과하면, double indirect block을 사용하여 data block address를 찾아야 한다. 이를 위해 기존의 direct, single indirect block 처리 다음에 double indirect block을 처리하는 부분을 추가하였다. 먼저 ip->addrs[NDIRECT+1]을 읽고, bn을 이용해 해당 블록 내에서 찾아야 할 single indirect block address를 계산한다. 그 후, 해당 single indirect block을 읽어 최종 데이터 블록의 주소를 찾고 각 단계에서 block이 할당되지 않았다면 balloc을 통해 새로 할당하도록 하였다.

```
void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp, *bp2;
  uint *a, *a2;

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

```
    ip->addrs[i] = 0;
  }
}

// Free singly-indirect blocks
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;
}

// Free doubly-indirect blocks
if(ip->addrs[NDIRECT + 1]){
  bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
  a = (uint*)bp->data;
  // For each entry in the doubly-indirect block
  for(j = 0; j < NINDIRECT; j++){
    if(a[j]){
      // read the singly-indirect block it points to
      bp2 = bread(ip->dev, a[j]);
      a2 = (uint*)bp2->data;
      // For each entry in the singly-indirect block
      for(int k = 0; k < NINDIRECT; k++){
          if(a2[k])
              bfree(ip->dev, a2[k]); // free data block
      }
      brelse(bp2);
      bfree(ip->dev, a[j]); // free the singly-indirect block
    }
  }
  brelse(bp);
  bfree(ip->dev, ip->addrs[NDIRECT + 1]); // free the doubly-indirect block
```

```
    ip->addrs[NDIRECT + 1] = 0;
  }

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

파일을 삭제할 때 double indirect block과 그에 연결된 모든 하위 블록들(중간 단계의 single indirect block들, 실제 데이터 블록들)을 해제하지 않으면 디스크 공간에 leak가 발생하게 된다. 이를 피하기 위해 single indirect block을 해제한 다음에 double indirect block을 해제하도록 하였다. 이중 for문을 돌며 double indirect block이 가리키는 single indirect block을 찾고, 각 single indirect block이 가리키는 모든 데이터 블록을 먼저 bfree로 해제한 후 중간 단계의 single indirect block들을 해제하고, 마지막으로 double indirect block 자체를 해제하였다.

```
$ bigfile
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................
wrote 65803 blocks
bigfile done; ok
```

3. Symbolic Link

```
uint64
sys_symlink(void)
{
  char target[MAXPATH], path[MAXPATH];
  struct inode *ip;

  if(argstr(0, target, MAXPATH) < 0 || argstr(1, path, MAXPATH) < 0) {
    return -1;
  }

  begin_op();
```

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

  if(writei(ip, 0, (uint64)target, 0, strlen(target) + 1) < 0){
    iunlockput(ip);
    end_op();
    return -1;
  }

  iunlockput(ip);
  end_op();

  return 0;
}
```

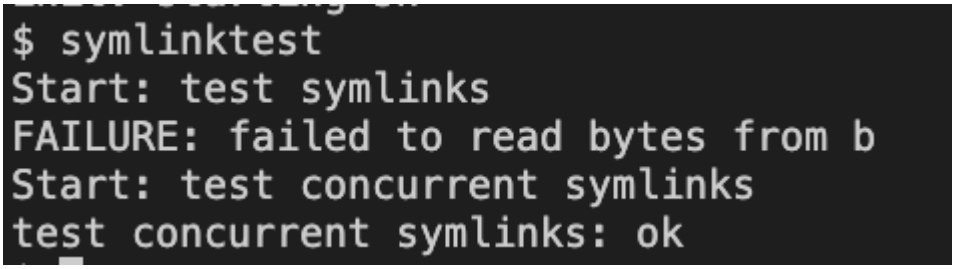
begin_op() 를 호출하여 atomic 하게 하였다. 기존의 create를 재사용하여 T_SYMLINK 라는 새로운 타입의 inode를 생성하였다. create 함수는 내부적으로 새로운 inode를 할당하고, path에 명시된 이름으로 부모 디렉터리에 연결하였다. 일반 파일이 사용자 데이터를 저장하는 것과 달리, symbolic link는 그 내용물로 목표 경로 자체를 가지기에 writei 함수를 호출하여 symbolic link의 핵심 데이터인 target 경로 문자열을 방금 생성한 inode의 데이터 블록에 기록하였다. 모든 작업이 성공적으로 완료되면 iunlockput 으로 inode의 잠금 및 참조를 해제하고, end_op()를 호출하여 최종적으로 commit하였다.

test 를 실행해보았을 때 panic: virtio_disk_intr status 가 발생하였다.

panic이 발생하게 된 경로를 찾아보니 close(fd2) → fileclose(f) → iput(ff.ip) → itrunc(ip) → bp2 = bread(ip→dev, a[j]) → virtio_disk_rw(b,0) 였다.

로그를 보았을 때는 쓰레기 값이 들어있어서 발생한 panic 으로 보인다.

초기 sys_open 에서 for 문으로 depth 까지 Recursive link following 만 남겼을 때는 프로그램 실행 중에는 panic 이 뜨지 않고 아래와 같이 로그가 보였다.


$ symlinktest
Start: test symlinks
FAILURE: failed to read bytes from b
Start: test concurrent symlinks
test concurrent symlinks: ok