Project 3
Wiki Documentation

This project focuses on the implementation of a virtual memory and file system in a RISC-V xv6 architecture. There are three subportions to this project that need to be implemented. First is Copy-on-Write (COW) Fork, which involves memory management optimization through lazy allocation. Next is Large Files, which involves file system extension with doubly-indirect blocks. Last is symbolic links, which involve advanced file referencing and path resolution.

I. Copy-on-Write Fork

Copy-on-Write (COW) complements the inefficient memory copying in fork by initially sharing pages between the parent and child and only copying pages when the pages are modified. This avoids failure due to insufficient memory. To implement this, I modified core functions: uvmcopy(), usertrap(), copyout(), kalloc(), and kfree() and attempted to maintain compatibility with existing code.

IMPLEMENTATION

Declare the function in the defs.h file

In order to add functions that are implemented across kernel files, I declared the function in the kernel/defs.h file. This handles synchronization for reference counts: add_ref adds 1, dec_ref subtracts 1, and pa_index is used to track reference counts for pages.

```
void            add_ref(void*);
void            dec_ref(void*);
int             pa_index(void*);
```

defs.h file

Add the code file to the makefile

Adding the code $U/_cowtest\ into the makefile ensures that the user program is properly compiled and linked once I run the makefile.

```
$U/_cowtest\
```

Makefile

Modifying kalloc.c functions

I first defined a new variable MAX_PAGES that calculates the maximum number of physical pages in xv6 architecture.

```
#define MAX_PAGES ((PHYSTOP - KERNBASE) / PGSIZE)
```

kalloc.c file (define MAX_PAGES)

I introduced a new lock called ref_lock in order protect page_refcount from race conditions when adding or decrementing the reference count. In order to properly use this lock, I initialized the ref_lock in kinit().

```
struct {
  struct spinlock lock;
  uint page_refcount[MAX_PAGES];
} ref_lock;
```

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

kalloc.c file (ref_lock)

Then, I introduced three new functions, pa_index, dec_ref, and add_ref. pa_index first checks if pa is within kernel's memory range and then calculates the index of the physical page. add_ref checks if the index is valid, then acquires the ref_lock and adds one to the reference count. dec_ref decrements the reference count by one and checks if the count reaches zero, which means that the page isn't being referenced anymore. Then, the physical page is freed via kfree.

```
void add_ref(void *pa) {
  int index = pa_index(pa);
  if (index == -1) {
    return;
  }
  acquire(&ref_lock.lock);
  ref_lock.page_refcount[index]++;
  release(&ref_lock.lock);
}
```

```
void dec_ref(void *pa) {
  int index = pa_index(pa);
  if (index == -1) {
    return;
  }
  acquire(&ref_lock.lock);
  ref_lock.page_refcount[index]--;
  int current_count = ref_lock.page_refcount[index];
  release(&ref_lock.lock);
  if (current_count== 0) {
    kfree(pa);
  }
}
```

```
int pa_index(void* pa) {
  if ((uint64)pa < KERNBASE || (uint64)pa >= PHYSTOP) {
    return -1;
  }
  return ((uint64)pa - KERNBASE) / PGSIZE;
}
```

kalloc.c file (pa_index, add_ref, and dec_ref functions)

I also modified kalloc to implement reference counting. It checks if a page was allocated and then calculates the index relative to the physical page number and intializes the reference count for this page to one.

```
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
    int index = ((uint64)r - KERNBASE) / PGSIZE;
    acquire(&ref_lock.lock);
    ref_lock.page_refcount[index] = 1;
    release(&ref_lock.lock);
  }
  return (void*)r;
}
```

kalloc.c file (kalloc function)

Lastly, I modified the kfree to support reference counting and implement shared page management. It checks if the reference count is greater than one, which means that the page is still being shared, and then proceeds to decrement it by one. Then, it proceeds to free the page by resetting the refcount to zero.

```
void
kfree(void *pa)
{
  struct run *r;
  int index = pa_index(pa);
  acquire(&ref_lock.lock);
  if(ref_lock.page_refcount[index] > 1){
    ref_lock.page_refcount[index] -= 1;
    release(&ref_lock.lock);
    return;
  }
  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);
  ref_lock.page_refcount[index] = 0;
  release(&ref_lock.lock);

  r = (struct run*)pa;

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

kalloc.c file (kfree function)

Modifying vm.c functions

I first set PTE_COW to be the eigth bit since it is unused. I modified the uvmcopy() to share pages instead of copying the parent memory immediately. pa extracts the physical address stored in pte and flags extracts the permission bits of pte. It then checks if the page is writeable, and if it is, it clears the "writeable" bit and sets the COW bit. This makes the parent page read-only and COW. It does the same for the child by inserting the same page into the child's pagetable and modifies the bits. Here, add_ref keeps track of how many processs share this physical page.

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

```
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);
    if (flags & PTE_W) {
      *pte = (*pte & ~PTE_W) | PTE_COW;
    }
    if(mappages(new, i, PGSIZE, pa, (flags & ~PTE_W) | PTE_COW) != 0)
      goto err;

    add_ref((void*)pa);
  }
  return 0;

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

vm.c file (uvmcopy function)

Next, I modified the copyout function. I first moved the (*pte&PTE_W ==0) if condition to the end, after checking if the pte is marked COW because in the previous step, PTE_W was cleared and the PTE_COW bit was introduced. If we were to check if the pte was writeable before checking the COW bit, it would automatically fail and reject actual COW pages.

After, allocating a new physical page, the data from the old shared page is copied to a new private page and the previous mapping is unmapped. The reference count is decremented and clears the cow bit and sets the new page as writeable.

```
if(pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0)
  return -1;

if (*pte & PTE_COW) {
  pa0 = PTE2PA(*pte);
  uint flags = PTE_FLAGS(*pte);
  char *mem = kalloc();
  if(mem == 0) {
    return -1;
  }
  memmove(mem, (void*)pa0, PGSIZE);
  uvmunmap(pagetable, va0, 1, 0);
  dec_ref((void*)pa0);
  if (mappages(pagetable, va0, PGSIZE, (uint64)mem, (flags | PTE_W) & ~PTE_COW) != 0) {
    kfree(mem);
    return -1;
  }
  pte = walk(pagetable, va0, 0);
  if (pte == 0)
    return -1;
}
if ((*pte & PTE_W) == 0) {
  return -1;
}
```

vm.c file (copyout function)

Modifying trap.c functions

The usertrap() function was modified to handle COW page faults (scause 13 or 15) which are triggered by copy-on-write faults. It first checks if the page table entry is valid or user-accessible. If it isnt, it kills the process. Then, it checks if the page is marked as COW and proceeds to allocate a new page and copy its contents. It updates the PTE to point to the new page then lastly, decrements the reference counter.

```
else if (r_scause() == 13 || r_scause() == 15) { //load/store page fault
  uint64 start_va = PGROUNDDOWN(r_stval());
  pte_t *pte = walk(p->pagetable, start_va, 0);
  if (pte == 0 || !(*pte & PTE_V) || !(*pte & PTE_U)) {
    setkilled(p);
    return;
  }

  if (*pte & PTE_COW) {
    char *pa = (char *)PTE2PA(*pte);
    char *mem = kalloc();
    if (mem == 0) {
      setkilled(p);
      return;
    }

    memmove(mem, (void *)pa, PGSIZE);

    uint flags = PTE_FLAGS(*pte); //update mapping
    flags |= PTE_W;
    flags &= ~PTE_COW;

    *pte = PA2PTE(mem) | flags;
    sfence_vma(p->pagetable, start_va);

    dec_ref(pa);
  }
  else {
    setkilled(p);
  }
}
```

vm.c file (usertrap function)

RESULT

After modifying several core functions, I booted the xv6 kernel and got the following results.

```
xv6 kernel is booting

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

execution results

As for the flow of operation, the program starts with the first part of the test, which is the simpletest() function. It tests basic COW memory allocation by allocating a large memory and calling fork. Since it only shared pages/created COW mappings, the parent shrinks back. The next test, threetest(), checks for multi-process COW testing with multiple children writing. It checks page fault handling, private copy creation, and proper freeing of memory. Lastly, filetest() tests the compatibility of copyout() with COW by using a pipe to send data between parent and child.

TROUBLESHOOTING

I ran into a pipe failed error in the filetest while running the program. It turns out that I hadn't modified the copyout() properly, so it was checking if the page is writeable before checking the COW bit. Since, the writeable bit was previously cleared, it returned -1. This was fixed by moving that portion of the if statement to the end, after checking the COW bit first.

```
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: pipe() failed
```

troubleshooting

II. Large Files

A problem in xv6 is the limited file size, which makes it insufficient for modern applications. A solution is to use doubly-indirect blocks, which allows file systems to handle large files by storing pointers to other blocks, which point to the actual data blocks. Its composed of 11 direct, one singly-indirect, and one doubly-indirect blocks. I implemented this by changing and modifying constants and structures and modifying core functions bmap() and itrunc().

IMPLEMENTATION

Modifying constants and structures

First, I changed the FSSIZE in param.h from 2,000 to 200,000.

```
#define FSSIZE       200000
```

param.h file

Next, I modified constants that define the maximum file size in the xv6 file system. NDIRECT refers to the number of direct block pointers are stored in the inode, which in this case was changed from 12 to 11. NINDIRECT refers to the number of block numbers that can fit in one indirect block. NDINDIRECT refers to the number of block numbers that can fit in a double-indirect block. It holds NINDIRECT pointers to indirect blocks, and each holds a NINDIRECT pointer to data blocks. Then, I updated the MAXFILE calculation. Lastly, I modified the addrs[] array from +1 to +2 to include both an indirect and double-indirect blcok pointer

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDINDIRECT (NINDIRECT * NINDIRECT * 4)
#define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
```

```
uint addrs[NDIRECT+2];
```

fs.h file

Add the code file to the makefile

Adding the code $U/_bigfile\ into the makefile ensures that the user program is properly compiled and linked once I run the makefile.

```
$U/_bigfile\
```

Makefile

Modifying fs.c functions

I first updated the bmap() function. To implement doubly-indirect access translation. It first checks if a double indirect block exists. If it doesn't, it allocated a new block (balloc()) and store it in the inode. It then reads the block from the disk and computes the first level index d1, which determines which block we need. In case the first-level indirect block is missing, it's allocated and written back to disk.

Next, the first-level indirect block is read and the second-level index d2 is computed. In case its missing, its allocated and written to the disk. Lastly, this function returns the actual physical address of the data block.

```
bn -= NINDIRECT;

if (bn < NDINDIRECT) {
  if ((addr = ip->addrs[NDIRECT + 1]) == 0) {
    addr = balloc(ip->dev);
    ip->addrs[NDIRECT + 1] = addr;
  }

  bp = bread(ip->dev, addr);
  uint *indirect = (uint*)bp->data;
  uint d1 = bn / NINDIRECT;    //first-level
  if (indirect[d1] == 0) {
    indirect[d1] = balloc(ip->dev);
    log_write(bp);
  }
  brelse(bp);

  bp = bread(ip->dev, indirect[d1]);
  uint *dindirect = (uint*)bp->data;
  uint d2 = bn % NINDIRECT;    //second-level
  if (dindirect[d2] == 0) {
    dindirect[d2] = balloc(ip->dev);
    log_write(bp);
  }
  addr = dindirect[d2];
  brelse(bp);
  return addr;
}
```

fs.c file (bmap function)

I also updated the itrunc() function to free doubly-indirect blocks properly. If a double indirect block exists, it frees the data block from its second-level, first-level, then the double indirect block itself. Then, since the inode has no more data, the size is set to zero and written back to disk.

```
  if (ip->addrs[NDIRECT + 1]) {
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    uint *indirect = (uint*)bp->data;

    for (int i = 0; i < NINDIRECT; i++) {
      if (indirect[i]) {
        struct buf *bp2 = bread(ip->dev, indirect[i]);
        uint *dindirect = (uint*)bp2->data;
        for (int j = 0; j < NINDIRECT; j++) {
          if (dindirect[j]) {
            bfree(ip->dev, dindirect[j]);
          }
        }
        brelse(bp2);
        bfree(ip->dev, indirect[i]);
      }
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT + 1]);
  }
  ip->size = 0;
  iupdate(ip);
}
```

fs.c file (itrunc function)

RESULTS

After modifying several core functions, I booted the xv6 kernel and got the following results.

```
xv6 kernel is booting

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

execution results

As for the flow of operation, the program tests whether or not the file system can handle large files correctly. It writes repeatedly to the file, and writes the current block number to the beginning of the variable buf until the file system refuses further writes. Lastly it checks if it wrote 65803 blocks, which is the expected maximum file size supported with the inode layout.

TROUBLESHOOTING

I ran into a **panic: bmap: out of range** message when I ran this program. It turns out I was subtracting bn by NDIRECT twice, instead of doing NDIRECT and NINDIRECT once each.

III. Symbolic Links

Symbolic links, also known as soft links, are file systems that point to another file or folder. The xv6 architecture encounters a problem as it offers rigid file referencing. This means that links are inode-based and cannot be referenced across file systems. To solve this, I implemented the symbolic links architecture in xv6 by adding new components: T_SYMLINK file type, O_NOFOLLOW flag, and symlink() system call. I also modified sys_open() system call to implement link resolution logic and handle if the inode's type is a symbolic type, and I modified several kernel and user files as shown below.

IMPLEMENTATION

Register the system calls in the syscall.c file

I made a few changes to the syscall.c file, which added/registered the new functions as new system calls.

```
extern uint64 sys_symlink(void);  [SYS_symlink] sys_symlink,
```

syscall.c file

Register the functions in the system call numbers

In addition to registering the functions in the syscall.c file, I also registered it in the syscall.h file by defining its system call number to the latest available number, which in this case is 22. This is done so that a number corresponds to each specific system call.

```
#define SYS_symlink 22
```

syscall.h file

Declare the function in the user.h file

In order to make the system call available at the user program, I declared the function in the user/user.h file.

```
int symlink(const char*, const char*);
```

user.h file

Add a new macro in the usys.pl file

Afterward, I added new macros in the usys.pl file so that it generates a system call stub that allows user programs to invoke these system calls. The usys.pl file overall defines how to preprocess when using the system call in a user program.

```
entry("symlink");
```

usys.pl file

Add the code file to the makefile

Adding the code $U/_symlinktest\ into the makefile ensures that the user program is properly compiled and linked once I run the makefile.

```
$U/_symlinktest\
```

Makefile

Adding new components: file type and flag

Added a new file type T_SYMLINK in kernel/stat.h and O_NOFOLLOW flag in kernel/fcntl.h. The flag number was decided in a way that it doesn't clash with other preexisting number.

```
#define O_NOFOLLOW 0x800
```

fcntl.h file

```
#define T_SYMLINK 4
```

stat.h file

Adding required specifications to the system calls in the sysfile.c file

Two major changes were made to the sysfile.c file. First of all, I created a new system call called symlink(target, path) that creates a new symbolic link that refers to the target at the path. It returns -1 if it fails and returns 0 if it successfully creates it. After checking for failure, the system call writes the symlink target path string into the inode's data blocks, in this case is ip.

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

  if((n = argstr(0, target, MAXPATH)) < 0 || argstr(1, path, MAXPATH) < 0)
    return -1;//failure

  begin_op();

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

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

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

sysfile.c file (symlink system call)

Next, the sys_open system call was modified to implement link/path resolution logic. It involves a recursive link with a depth of 10, and the while loop resolves symlinks by reading the target path from the inode data and calling namei() . This is done until it reaches a file that is not a symlink or exceeds the depth limit. The depth count also serves as a cycle detection to prevent infinite loops ("Should not be able to open b (cycle b->a->b->..)").

```
int depth = 0, length;
char next[MAXPATH+1];
if(!(omode & O_NOFOLLOW)){
  while(ip->type == T_SYMLINK && depth < 10){
    length = readi(ip, 0, (uint64)next, 0, MAXPATH);
    if(length < 0){
      iunlockput(ip);
      end_op();
      return -1;
    }
    next[length] = 0;
    iunlockput(ip);
    if((ip = namei(next)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);
    depth++;

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

sysfile.c file (open system call)

RESULTS

After implementing the symlink system call and modifying the open system call, I booted the xv6 kernel and got the following results.

```
xv6 kernel is booting

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

execution results

As seen in the photo, the program tests both the core functionality of symlink and concurrent testing.

As for the flow of operation, the program starts with file cleanup then moves on to the first part of the test, which is the testsymlink() function. It checks for basic and core functionality of symlinks, such as proper link creation and access, broken link handling, circular reference detection, non-existent target linking, and chain link resolution. Then, it moves onto the second test, which is the concur() function. It tests concurrency by looping

through the children of a base file z, and either create symlink or unlink. After, the file is checked to verify if it is a symlink using the O_NOFOLLOW. Overall, it checks race conditions on symlink creation and deletion under concurrency

TROUBLESHOOTING

I ran into two issues when running this program. First, I kept getting a **panic: virtio_disk_intr status** message whenever I tried running the program. It turns out that in the "large files" portion of the project, I had modified the addrs[] array in the dinode structure to add +1, but had fogotten to add +1 in the inode structure. Thus, once I modified the addrs[], it got rid of the panic message.

```
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];
};
```

File.h file

Moreover, I kept getting a **FAILURE: failed to stat b** error message when I ran the first half of the symlinktest (testsymlink() function). I fixed this issue by changing the O_NOFOLLOW flag number. It was originally set to 0x700, but I think it overlapped with an existing flag. Thus, I changed it to 0x800, and it fixed the error message and resulted in the desired results.