Xv6 Kernel Optimization

Michael Tin

March 2025

Abstract

This project implements a collection of advanced system calls for the XV6 kernel. The goal is to expand on XV6's core functionality with file system traversal, inter-process communication, and optimized I/O operations. It is based on MIT's 6.1810 (formerly 6.828) Operating Systems course.

This project accomplishes the following objectives:

- Implements standard Unix user-level utilities: `sleep`, which pauses execution for a set number of ticks; `find`, which recursively searches directories for specific filenames; and `xargs`, which executes commands from standard input.
- Improves memory allocation by replacing statically declared arrays with a buddy allocator and implementing lazy page allocation for user-space heap memory.
- Optimizes the `fork()` system call by using a copy-on-write method that initially shares physical memory pages between parent and child processes, as opposed to directly duplicating them.
- Adds to the capabilities of the existing Xv6 file system by supporting much larger file sizes and implementing symbolic links.
- Implements the `mmap` and `munmap` system calls, allowing processes to dynamically map files directly into memory and share memory with other processes.

Contents

Unix Utilities

1 `sleep.c`

1.1 Objective

The goal of this exercise is to implement the UNIX `sleep` program for Xv6. The program pauses execution for a user-specified number of ticks. A tick represents a unit of time defined by the Xv6 kernel, specifically the interval between two interrupts from the timer chip. The implementation is placed in the file `user/sleep.c`.

1.2 Implementation

The implementation of the `sleep` program is straightforward.

1. Pass in the number of ticks as an argument (`argv[1]` in this case, as `argv[0]` is reserved for the command itself).
2. Convert `argv[1]`, which is a string, to an integer using `atoi()` function.
3. Call the `sleep` system call for the number of ticks passed in.

```
int main(int argc, char *argv[]) {
  if (argc < 2) {
    fprintf(2, "Usage: sleep <number of ticks>\n");
    exit();
  }

  int ticks = atoi(argv[1]);

  if (ticks <= 0) {
    fprintf(2, "Invalid number of ticks\n");
    exit();
  }
  sleep(ticks);
  exit();
}
```

2 `find.c`

2.1 Objective

The objective of this exercise is to implement a simplified version of the UNIX `find` program. The program searches through a directory tree and identifies all files that match a specified name. The implementation is placed in the file `user/find.c`.

2.2 Implementation

The code from the `ls` program can be reused for the `find` program because the functionality is very similar.

The purpose of the `ls` program is to list the contents of the directory where it is called from; it prints 1) all files and 2) all subdirectories and their contents. This is done through a single `switch` statement, where there is a case to print a file and a case to print a directory.

The case to print a directory inside the `switch` statement contains a `while` loop to explore all subdirectories. The `find` program needs to do this, but instead of printing all contents it prints only if a specific file name is found. This functionality is achieved by moving the `switch` statement to be inside of the `while` loop.

1. Pass in the path and the filename as arguments (`argv[1]` and `argv[2]`).
2. Check for errors at the beginning (cannot open or cannot stat).
3. Enter the `while` loop to begin reading the directory. If the directory name is `.` or `..`, do not explore and immediately `continue` the while loop.
4. Using `st.type`, determine the type of the object (`switch` statement).
    a. If it is a file, print it if it matches the passed in filename.
    b. If it is a directory, call the `find()` function again (recursion).

The goal is to recursively explore subdirectories until finding files; do not break the `while` loop until the type of the object is file, not directory.

```
void find(char *path, char* filename) {
  char buf[512], *p;
  int fd;
  struct dirent de;
  struct stat st;
  if((fd = open(path, 0)) < 0){
    fprintf(2, "find: cannot open %s\n", path);
    return;
  }
  if(fstat(fd, &st) < 0){
    fprintf(2, "find: cannot stat %s\n", path);
    close(fd);
    return;
  }
  strcpy(buf, path);
  p = buf+strlen(buf);
  *p++ = '/';
  while(read(fd, &de, sizeof(de)) == sizeof(de)){
    if(de.inum == 0)
      continue;
    // MT 1/18
    if ((strcmp(de.name, ".") == 0) || (strcmp(de.name, "..") == 0))
      continue;
    memmove(p, de.name, DIRSIZ);
    p[DIRSIZ] = 0;
    if(stat(buf, &st) < 0){
      printf("find: cannot stat %s\n", buf);
      continue;
    }
    // MT 1/18
    // printf("%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
    switch(st.type) {
      case T_FILE:
        // MT 1/18
        // printf("%s %d %d %l\n", fmtname(path), st.type, st.ino, st.size);
        if (strcmp(de.name, filename) == 0) {
          printf("%s\n", buf);
        }
        break;
      case T_DIR:
        // MT 1/18
        // if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
        //   printf("find: path too long\n");
        //   break;
        // }
        find(buf, filename);
        break;
    }
  }
  close(fd);
}
```

3 xargs.c

3.1 Objective

The goal of this exercise is to implement a simplified version of the UNIX `xargs` program. The program reads lines from standard input and executes a command for each line, passing the line as arguments to that command. The implementation is placed in the file `user/xargs.c`.

3.2 Implementation

The implementation of the `xargs` program is straightforward.

1. Declare an array of `char` pointers (`arrArgs`) which is size `MAXARG = 32`. This array stores the individual arguments as they are processed.
2. Declare a buffer array to store arguments (`buffer`) which is size `1024`.
3. The `for` loop reads the initial arguments (after `xargs`) and adds them to the argument pointers array.
4. After this, the remaining arguments are processed one character at a time. The characters are read into a temporary variable (`temp`) and then written to the buffer array (`buffer`).
5. When a space or newline is encountered, a null terminator (`'\0'`) is added to the buffer array (`buffer`). Then, the `char` pointer to the first letter in the string (`p`) is added to the array of char pointers (`arrArgs`).
6. When a newline is encountered, a child process is forked. The parent process then waits for the child process to finish (`wait` until `fork()`'s child exits).

```c
int main(int argc, char *argv[]) {
  if (argc < 2) {
    fprintf(2, "Usage: xargs <command>\n");
    exit();
  }
  char *arrArgs[MAXARG]; // 32
  char temp = '0';
  char buffer[1024];
  char *p = buffer;
  int position = 0;
  int numArgs = 0;
  for (int i = 1; i < argc; i++) {
    arrArgs[i - 1] = argv[i]; // Skip argv[0] which is xargs
    if (i == (argc - 1)) {
      numArgs = i;
    }
  }
  while (read(0, &temp, 1) != 0) {
    if ((temp == ' ') || (temp == '\n')) {
      buffer[position] = '\0';
      arrArgs[numArgs] = p;
      position++;
      numArgs++;
      if (temp == '\n') {
        p = buffer + position; // The space after the last arg
        if (fork() == 0) {
          exec(arrArgs[0], arrArgs);
        }
        else {
          wait();
        }
        numArgs = argc - 1;
      }
    }
    else {
      buffer[position] = temp;
      position++;
    }
  }
  exit();
}
```

4 Heap Allocator

4.1 Objective

Xv6 has only a page allocator and cannot dynamically allocate objects smaller than a page. To work around this limitation, Xv6 declares objects smaller than a page statically. For example, Xv6 declares an array of file structs, an array of proc structures, and so on. As a result, the number of files the system can have open is limited by the size of the statically declared file array, which has NFILE entries (see kernel/file.c and kernel/param.h).

The solution is to adopt the buddy allocator, which we have added to Xv6 in `kernel/buddy.c` and `kernel/list.c`. In `kernel/file.c`, the number of file structures should be limited by available memory rather than `NFILE`.

4.2 Implementation

The implementation of the Heap Allocator is straightforward and simple. The main task is to revise the implementation of `filealloc()`.

- The original code 1) acquires the lock protecting the file table, 2) loops through the array to find a free file structure, 3) once found, sets the reference count of the free file structure to 1, 4) returns a pointer to the allocated file structure and releases the lock.
- The new code 1) uses `bd_malloc()` to allocate a block of memory large enough to hold a `file`, 2) acquires the lock protecting the file table, 3) sets the reference count of the new `file` structure to 1, 4) releases the lock and returns a pointer to the allocated file structure.
- Additionally, we add `bd_free()` to the `fileclose` function to free memory that was allocated by `bd_malloc()`.
- With the implementation described here, `alloctest` successfully passes all tests.

4.2.1 filealloc() (kernel/file.c)

```
// Allocate a file structure.
struct file*
filealloc(void)
{
    struct file *f;
    f = (struct file *) bd_malloc(sizeof(struct file));
    acquire(&ftable.lock);
    // for(f = ftable.file; f < ftable.file + NFILE; f++){
    //   if(f->ref == 0){
    //     f->ref = 1;
    //     release(&ftable.lock);
    //     return f;
    //   }
    // }
    f->ref = 1;
    release(&ftable.lock);
    return f;
}
```

4.2.2 fileclose() (kernel/file.c)

```
// Close file f. (Decrement ref count, close when reaches 0.)
void
fileclose(struct file *f)
{
    struct file ff;
    acquire(&ftable.lock);
    if(f->ref < 1)
        panic("fileclose");
    if(--f->ref > 0){
        release(&ftable.lock);
        return;
    }
    ff = *f;
    f->ref = 0;
    f->type = FD_NONE;
    release(&ftable.lock);
    if(ff.type == FD_PIPE){
        pipeclose(ff.pipe, ff.writable);
    } else if(ff.type == FD_INODE || ff.type == FD_DEVICE){
        begin_op(ff.ip->dev);
        iput(ff.ip);
        end_op(ff.ip->dev);
    }
    bd_free((char *) f);
}
```

5 Lazy Page Allocation

5.1 Objective

One of the many neat tricks an O/S can play with page table hardware is lazy allocation of user-space heap memory. Xv6 applications ask the kernel for heap memory using the `sbrk()` system call. In the Xv6 kernel, `sbrk()` allocates physical memory and maps it into the process's virtual address space. However, there are programs that use `sbrk()` to ask for large amounts of memory but never use most of it, for example to implement large sparse arrays. To optimize for this case, sophisticated kernels allocate user memory lazily. That is, `sbrk()` doesn't allocate physical memory, but just remembers which addresses are allocated. When the process first tries to use any given page of memory, the CPU generates a page fault, which the kernel handles by allocating physical memory, zeroing it, and mapping it.

5.2 Implementation

The implementation of the Lazy Page Allocation is complex and involves modifications to multiple files. Initially, we delete page allocation from the `sys_sbrk()` system call implementation. The new `sys_sbrk()` simply increments the process's size (`myproc()->sz`) by `n` and returns the old size, without allocating memory.

5.2.1 `sys_sbrk()` (`kernel/sysproc.c`)

```
uint64
sys_sbrk(void)
{
    int addr;
    int n;
    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;
    // if(growproc(n) < 0)
    //   return -1;
    myproc()->sz += n; // Increment the process's size by n
    return addr;
}
```

We get the following result when attempting to run `echo hi`.

```
virtio disk init 0
hart 2 starting
hart 1 starting
init: starting sh
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
 sepc=0x000000000000124e stval=0x0000000000004008
va=0x0000000000004000 pte=0x0000000000000000
panic: uvmunmap: not mapped
```

We first have to fix `usertrap()` in `trap.c` so that `echo hi` can run in the shell again. The code should respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing.

The implementation of the modified `usertrap()` function is straightforward. We simply modify the `else` branch in the `usertrap()` function. This is the core idea of the Lazy Page Allocation.

1. Check whether a fault is a page fault (`r_scause() == 13` or `15`)
2. Check whether the address is out of bounds; if it is, kill the process.
3. Allocate a new page using `kalloc()`, and map the address to the new page using `mappages()`; if either of these operations fail, kill the process.

5.2.2 `usertrap()` (`kernel/trap.c`)

```
void
usertrap(void)
{
    ...
    else {
        if ((r_scause() == 13) || (r_scause() == 15)) {
            if ((r_stval() >= p->sz) || (r_stval() <= p->ustack)) {
                p->killed = 1;
                goto end;
            }
            uint64 a = PGROUNDDOWN(r_stval());
            char *mem = kalloc();
            if (mem == 0) {
                p->killed = 1;
                goto end;
            }
            memset(mem, 0, PGSIZE);
            if(mappages(p->pagetable, a, PGSIZE, (uint64)mem, PTE_W|PTE_X|
               PTE_R|PTE_U) != 0) {
                p->killed = 1;
                kfree(mem);
                // uvmdealloc(pagetable, a, oldsz);
                goto end;
            }
        }
        ...
    end:
    }
    ...
}
```

To check whether the address is out of bounds, we add `uint64 ustack` to the `proc` struct in `proc.h`. This is the bottom of the user stack.

5.2.3 proc{} (kernel/proc.h)

```
struct proc {
    ...
    // these are private to the process, so p->lock need not be held.
    uint64 kstack; // Bottom of kernel stack for this process
    uint64 ustack; // Bottom of user stack
    ...
};
```

5.2.4 fork() (kernel/proc.c)

```
int
fork(void)
{
    ...
    np->ustack = p->ustack;
    np->parent = p;
    ...
}
```

5.2.5 exec() (kernel/exec.c)

```
int
exec(char *path, char **argv)
{
    ...
    stackbase = sp - PGSIZE;
    p->ustack = stackbase;
    ...
}
```

This is not the expected behavior of XV6, so the OS will panic. We need to modify functions in `vm.c` so that we can avoid this.

5.2.6 walk() (kernel/vm.c)

Do not panic on `MAXVA` check; simply comment this out.

```
static pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
    // if(va >= MAXVA)
    //   panic("walk"); // change this
    ...
}
```

5.2.7 mappages() (kernel/vm.c)

Do not panic on remap condition; simply move to the next page.

```
int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
    ...
    if(*pte & PTE_V) {
        // panic("remap");
        a += PGSIZE;
        if (a > last) {
            break;
        }
        continue;
    }
    ...
}
```

5.2.8 uvmunmap() (kernel/vm.c)

Do not panic if a page does not have a PTE or if its VA is not mapped; simply move to the next page.

```
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 size, int do_free)
{
    ...
    for(;;){
        if((pte = walk(pagetable, a, 0)) == 0) {
            // panic("uvmunmap: walk");
            a += PGSIZE;
            if (a > last) {
                break;
            }
            continue;
        }
        if((*pte & PTE_V) == 0) {
            // printf("va=%p pte=%p\n", a, *pte);
            // panic("uvmunmap: not mapped");
            a += PGSIZE;
            if (a > last) {
                break;
            }
            continue;
        }
        ...
    }
}
```

5.2.9 uvmcopy() (kernel/vm.c)

Do not panic if PTE does not exist or page is not present; simply continue loop.

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    ...
    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walk(old, i, 0)) == 0)
            // panic("uvmcopy: pte should exist");
            continue;
        if((*pte & PTE_V) == 0)
            // panic("uvmcopy: page not present");
            continue;
        ...
    }
}
```

We also comment out a `kalloc()` panic in the `sys_exec()` function.

5.2.10 sys_exec() (kernel/sysfile.c)

```
uint64
sys_exec(void)
{
    ...
    // if(argv[i] == 0)
    //   panic("sys_exec kalloc");
    ...
}
```

Additionally, we implement a check in `sys_sbrk()` to handle negative arguments.

5.2.11 sys_sbrk() (kernel/sysproc.c)

```
uint64
sys_sbrk(void)
{
    int addr;
    int n;
    if(argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;
    // if(growproc(n) < 0)
    //   return -1;
    myproc()->sz += n; // Increment the process's size by n
    if (n < 0) {
        uvmdealloc(myproc()->pagetable, addr, myproc()->sz);
        // uvmunmap(myproc()->pagetable, addr + n, addr - (addr + n), 1);
    }
    return addr;
}
```

The final modification we need to make is to handle the case in which a process passes a valid address from `sbrk()` to a system call such as `read` or `write`, but the memory for that address has not yet been allocated.

To handle this, we modify the `copyout()`, `copyin()`, and `copyinstr()` functions which can be found in `vm.c`. These functions handle read and write operations between user space and kernel space. We can use the Lazy Page Allocation that was implemented in `usertrap()` earlier.

In each of the three functions, there is a specific section of code.

```
pa0 = walkaddr(pagetable, va0);
if(pa0 == 0)
    return -1;
```

The function of this code is to check if the virtual address is mapped to a physical address in the page table; else, it returns `-1`, indicating an error. This function would be correct in the original implementation of XV6. However, because we are using Lazy Page Allocation, we actually want to allocate memory when we come across a virtual address that isn't mapped to a physical address. Hence, we modify the implementation of that code accordingly.

1. Check if the virtual address is mapped to a physical address.
2. If it isn't, allocate memory using `kalloc()`, and map the address to the new page using `mappages()`; if either of these operations fail, return `-1` to indicate an error.
3. Check if the virtual address is mapped to a physical address again. If it somehow isn't at this point, return `-1` to indicate an error.

Another subtle modification we make is that we stop casting the result of `PGROUNDDOWN(srcva)` to type `uint`, because unsigned integers aren't big enough to store the result (also, `va0` is of type `uint64`, so this does not make sense regardless).

Old

```
va0 = (uint)PGROUNDDOWN(srcva);
```

New

```
va0 = PGROUNDDOWN(srcva);
```

5.2.12 copyout() (kernel/vm.c)

```
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
    uint64 n, va0, pa0;
    while(len > 0){
        va0 = PGROUNDDOWN(dstva);
        pa0 = walkaddr(pagetable, va0);
        if(pa0 == 0) {
            char *mem = kalloc();
            if(mem == 0)
                return -1;
            memset(mem, 0, PGSIZE);
            if(mappages(pagetable, va0, PGSIZE, (uint64)mem, PTE_W|PTE_X|
               PTE_R|PTE_U)!= 0){
                kfree(mem);
                return -1;
            }
        }
        pa0 = walkaddr(pagetable, va0);
        if(pa0 == 0)
            return -1;
        ...
    }
}
```

5.2.13 copyin() (kernel/vm.c)

```
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
    uint64 n, va0, pa0;
    while(len > 0){
        va0 = PGROUNDDOWN(srcva);
        pa0 = walkaddr(pagetable, va0);
        if(pa0 == 0) {
            char *mem = kalloc();
            if(mem == 0)
                return -1;
            memset(mem, 0, PGSIZE);
            if(mappages(pagetable, va0, PGSIZE, (uint64)mem, PTE_W|PTE_X|
               PTE_R|PTE_U)!= 0){
                kfree(mem);
                return -1;
            }
        }
        pa0 = walkaddr(pagetable, va0);
        if(pa0 == 0)
            return -1;
        ...
    }
}
```

5.2.14 copyinstr() (kernel/vm.c)

```
int
copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
    uint64 n, va0, pa0;
    int got_null = 0;
    while(got_null == 0 && max > 0){
        va0 = PGROUNDDOWN(srcva);
        pa0 = walkaddr(pagetable, va0);
        if(pa0 == 0) {
            char *mem = kalloc();
            if(mem == 0)
                return -1;
            memset(mem, 0, PGSIZE);
            if(mappages(pagetable, va0, PGSIZE, (uint64)mem, PTE_W|PTE_X|
               PTE_R|PTE_U)!= 0){
                kfree(mem);
                return -1;
            }
        }
        pa0 = walkaddr(pagetable, va0);
        if(pa0 == 0)
            return -1;
        ...
    }
}
```

6 Background

The `fork()` system call in Xv6 copies all of the parent process's user-space memory into the child. If the parent is large, copying can take a long time. In addition, the copies often waste memory; in many cases neither the parent nor the child modifies a page, so that in principle they could share the same physical memory. The inefficiency is particularly clear if the child calls `exec()`, since `exec()` will throw away the copied pages, probably without using most of them. On the other hand, if both parent and child use a page, and one or both writes it, a copy is truly needed.

The goal of copy-on-write (COW) `fork()` is to defer allocating and copying physical memory pages for the child until the copies are actually needed, if ever.

> COW `fork()` creates just a pagetable for the child, with PTEs for user memory pointing to the parent's physical pages.
>
> COW `fork()` marks all the user PTEs in both parent and child as not writable. When either process tries to write one of these COW pages, the CPU will force a page fault.
>
> The kernel page-fault handler detects this case, allocates a page of physical memory for the faulting process, copies the original page into the new page, and modifies the relevant PTE in the faulting process to refer to the new page, this time with the PTE marked writeable.
>
> When the page fault handler returns, the user process will be able to write its copy of the page.

COW `fork()` makes freeing of the physical pages that implement user memory a little trickier. A given physical page may be referred to by multiple processes' page tables, and should be freed only when the last reference disappears.

7 Implement Page Reference Counter

7.1 Objective

In `kalloc.c`, a new structure should be created to record the reference count of each physical page. This can be implemented using a data structure such as a linked list or a fixed-length array. Any modifications to the reference count must be protected by a lock to ensure thread-safe updates. When the `kalloc()` function allocates a new physical page, the reference count for that page should be initialized to 1. Within the `kfree()` function, the reference count should be decremented, and the physical page should only be released when the count reaches 0. Additionally, a separate function should be implemented to increment the reference count when a page gains an additional reference.

7.2 Implementation

The implementation of the page reference counter is straightforward. As mentioned in the objective, we need to modify the functions in `kalloc.c`. We first modify the `kmem` struct to include a reference counter. This is a fixed length array that allows us to access the reference count of each physical page. The size of the array is defined as `PHYMEM / PGSIZE`, where:

- `PHYMEM = PHYSTOP - KERNBASE`. This is the last physical memory address minus the beginning of the kernel memory, which gives us the total physical memory. This is also equivalent to $128 \times 1024 \times 1024$.
  - `PHYMEM` is a custom definition.
- `PGSIZE` is the page size, which is 4096.

7.2.1 (kernel/kalloc.c)

```
#define PHYMEM (PHYSTOP - KERNBASE) // PHYSTOP - KERNBASE = 128 * 1024 *
    1024
...
struct {
  struct spinlock lock;
  struct run *freelist;
  int refcnt[PHYMEM / PGSIZE]; // Fixed length array to access the reference
      count.
} kmem;
```

We add some functionality to the `kfree()` function. The reference count is decremented; then, we check if it has reached 0.

- If the reference count is greater than 0, we return and do not execute the rest of the code.
- If the reference count is 0, we release the physical page. Modification of the reference count is guarded by the lock.

7.2.2 kfree() (kernel/kalloc.c)

```
void
kfree(void *pa)
{
  ...
  // Decrement the ref count, release the physical page when it reaches 0.
  acquire(&kmem.lock); // Acquire lock.
  uint64 index = (((uint64)pa - KERNBASE) / PGSIZE); // Calculate the page
      index.
  kmem.refcnt[index]--; // Decrement the ref count.
  if (kmem.refcnt[index] > 0) { // If physical page has not reached 0,
      return early.
    release(&kmem.lock); // Release lock.
    return;
  }
  release(&kmem.lock); // Release lock.
  ...
}
```

Additionally, we add functionality to `kalloc()` to set all newly allocated physical page reference count to 1.

7.2.3 kalloc() (kernel/kalloc.c)

```
void *
kalloc(void)
{
  struct run *r;
  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r) {
    kmem.freelist = r->next;
    uint64 index = (((uint64)r - KERNBASE) / PGSIZE); // Calculate the page
        index.
    kmem.refcnt[index] = 1; // Newly allocated physical page ref count = 1.
  }
  release(&kmem.lock);
  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

We create two new functions to handle incrementing and decrementing of the reference count. Since modification of the reference count is guarded by the lock, adding these two new functions makes things easier when executing this code in other files.

The incrementing function, `refinc()`, is straightforward. It simply passes in a physical page address and then increments its reference count.

7.2.4 refinc() (kernel/kalloc.c)

```
// New function to increment the ref count.
void
refinc(uint64 pa)
{
  acquire(&kmem.lock); // Acquire lock.
  uint64 index = (((uint64)pa - KERNBASE) / PGSIZE); // Calculate the page
      index.
  kmem.refcnt[index]++; // Increment the ref count.
  release(&kmem.lock); // Release lock.
}
```

The decrementing function, `refdec()`, is also straightforward. It passes in a physical page address and decrements its reference count. However, we add one extra check to this function, which is to call `kfree()` on the physical page if the reference count reaches 0.

7.2.5 refdec() (kernel/kalloc.c)

```
// New function to decrement the ref count.
void
refdec(uint64 pa)
{
  acquire(&kmem.lock); // Acquire lock.
  uint64 index = (((uint64)pa - KERNBASE) / PGSIZE); // Calculate the page
      index.
  kmem.refcnt[index]--; // Decrement the ref count.
  if (kmem.refcnt[index] == 0) {
    release(&kmem.lock); // Release lock.
    kfree((void*)pa);
  }
  else {
    release(&kmem.lock); // Release lock.
  }
}
```

We add these new functions to the declarations for `kalloc.c` in `defs.h`.

7.2.6 (kernel/defs.h)

```
// kalloc.c
void* kalloc(void);
void kfree(void *);
void kinit();
void refinc(uint64);
void refdec(uint64);
```

8 Fix `uvmcopy()` function

8.1 Objective

The `uvmcopy()` function should be modified to support copy-on-write behavior. Instead of allocating a new physical page for the child process, the child's virtual page should be mapped to the same physical page used by the parent. The write permission (`PTE_W`) should be cleared from the page table entries of both processes to prevent direct modification of the shared page. A new privilege flag should also be defined to indicate that a page table entry represents a copy-on-write (COW) mapping; this flag can be defined in `riscv.h`. The reference counter for the shared physical page should be incremented to reflect that the page is now referenced by multiple processes.

8.2 Implementation

We first start by defining a privilege flag to record whether a PTE is COW mapping. This flag will be known as `PTE_COW`.

In Chapter 3.1 of the XV6 book, we can observe the RISC-V page table hardware. We see that Bits 0 - 7 are reserved for other functions, so we choose Bit 8 for the new `PTE_COW` privilege flag. We then add the privilege flag to `riscv.h`.

8.2.1 `(kernel/riscv.h)`

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

The modifications to `uvmcopy()` itself are as follows:

1. Clear `PTE_W` for the parent and child PTE.
2. Record `PTE_COW` for the parent and child PTE.
3. Map the parent's physical page to the child's virtual page.
4. Increment the page reference counter of this physical page.

We also remove the functionality of new page allocation.

8.2.2 uvmcopy() (kernel/vm.c)

```
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");
    // 1. Clear PTE_W for the parent and child PTE.
    *pte &= (~PTE_W);
    // 2. Record PTE_COW for the parent and child PTE.
    *pte |= PTE_COW;
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    // Remove the new page allocation.
    // if((mem = kalloc()) == 0)
    //   goto err;
    // memmove(mem, (char*)pa, PGSIZE);
    // if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
    //   kfree(mem);
    //   goto err;
    // }
    // 3. Map parent's physical page to child's virtual page.
    if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0)
      goto err;
    // 4. Increment the ref count of this physical page.
    refinc(pa);
  }
  return 0;
err:
  uvmunmap(new, 0, i, 1);
  return -1;
}
```

9 Fix `usertrap()` function

9.1 Objective

The `usertrap()` function should be modified to properly handle write page faults that occur on copy-on-write (COW) pages. When a page fault occurs, the handler should check whether the cause corresponds to a write fault (i.e., `r_scause()` returns 15) and verify that the faulting page is marked as a COW page using the newly defined privilege flag. If this condition is met, the kernel should allocate a new physical page and duplicate the contents of the original COW page into the new page, which can be performed using `memmove()`. After the copy is completed, the faulting virtual page should be remapped to the newly allocated physical page with the write permission (`PTE_W`) enabled. This remapping involves unmapping the original COW page and then mapping the virtual address to the new physical page with the updated permissions.

9.2 Implementation

The `usertrap()` function needs to 1) detect a page fault, 2) allocate physical memory, 3) copy the original page into the new page, and then 4) modify the relevant PTE to refer to the new page.

The implementation is as follows:

1. Add an "else if" branch to check for page faults.
   (a) `else if (r_scause() == 15)`
2. See if the virtual address is valid (if not, kill the process).
3. Traverse the page table and get a pointer to the PTE that corresponds to the virtual address. If the PTE is null, kill the process.
4. Check if the `PTE_COW` flag is set in the PTE.
5. Extract the privilege flags from the PTE.
6. Record the `PTE_W` privilege flag.
7. Clear the `PTE_COW` privilege flag.
8. Allocate a new page of physical memory using `kalloc()`. If `kalloc()` returns null, kill the process (memory allocation failed).
9. Extract the physical address from the original PTE.
10. Using `memmove()`, copy the contents of the original page to the new page.
11. Decrease the reference count of the original page.
12. Unmap from the original page.
13. Map to the new page.

9.2.1 usertrap() (kernel/trap.c)

```
void
usertrap(void)
{
  ...
  else if (r_scause() == 15) {
    uint64 va = PGROUNDDOWN(r_stval());
    if (va >= MAXVA) {
      p->killed = 1;
      goto end;
    }
    pte_t *pte = walk(p->pagetable, va, 0);
    if (!pte) {
      p->killed = 1;
      goto end;
    }
    if (*pte & PTE_COW) {
      uint flags = PTE_FLAGS(*pte);
      flags |= PTE_W;
      flags &= (~PTE_COW);
      char *mem = kalloc();
      if (!mem) {
        p->killed = 1;
        goto end;
      }
      uint64 pa = (uint64)PTE2PA(*pte);
      memmove(mem, (void*)pa, PGSIZE);
      refdec(pa);
      uvmunmap(p->pagetable, va, PGSIZE, 0);
      if (mappages(p->pagetable, va, PGSIZE, (uint64)mem, flags) != 0) {
        p->killed = 1;
        // refdec(pa);
        kfree(mem);
        goto end;
      }
    }
    end:
  }
  ...
}
```

10 Fix `copyout()` function

10.1 Objective

When copying data from the kernel to a user virtual address, the destination address may correspond to a copy-on-write (COW) page. In this case, the COW page must be handled in a manner similar to the logic used in the `usertrap` function. If the page is identified as a COW page, a new physical page should be allocated and the contents of the original COW page should be copied into the newly allocated page. After the copy is completed, the faulting virtual page should be remapped to the new physical page so that the write operation can proceed without modifying the shared COW page.

10.2 Implementation

The `copyout()` function has to be updated to handle the case where a page that is being copied is a COW page. We can reuse a lot of the code from `usertrap()`.

The implementation is as follows:

1. Set `va0` to the page-aligned starting virtual address of the page that contains the destination virtual address.
2. Traverse the page table and get the corresponding physical address; set `pa0` to the physical address. If `pa0` equals 0, return -1 (invalid address).
3. Traverse the page table to find the PTE for the virtual address. If no PTE is found, return -1 (error).
4. Check if the `PTE_COW` flag is set in the PTE.
5. Extract the privilege flags from the PTE.
6. Record the `PTE_W` privilege flag.
7. Clear the `PTE_COW` privilege flag.
8. Allocate a new page of physical memory using `kalloc()`. If `kalloc()` returns null, return -1 (error).
9. Using `memmove()`, copy the contents of the original page to the new page.
10. Decrease the reference count of the original physical page.
11. Unmap from the original page.
12. Map to the new page.
13. Update `pa0` to point to the new physical page.

10.2.1 copyout() (kernel/vm.c)

```
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;
  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    pte_t *pte = walk(pagetable, va0, 0);
    if (!pte) {
      return -1;
    }
    if (*pte & PTE_COW) {
      uint flags = PTE_FLAGS(*pte);
      flags |= PTE_W;
      flags &= (~PTE_COW);
      char *mem = kalloc();
      if (!mem) {
        return -1;
      }
      memmove(mem, (void*)pa0, PGSIZE);
      refdec(pa0);
      uvmunmap(pagetable, va0, PGSIZE, 0);
      if (mappages(pagetable, va0, PGSIZE, (uint64)mem, flags) != 0) {
        // refdec(pa0);
        kfree(mem);
        return -1;
      }
      pa0 = (uint64)mem;
    }
    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;
}
```

11 Large Files

11.1 Objective

In its current form, Xv6 files are limited to 268 blocks, or `268*BSIZE` bytes, where `BSIZE` is 1024 bytes in Xv6. This limitation arises because each Xv6 inode contains 12 direct block numbers and one singly-indirect block number. The singly-indirect block refers to a block that can hold up to 256 additional block numbers, resulting in a total capacity of 12 + 256 = 268 blocks.

To extend this limit, the Xv6 file system code should be modified to support a doubly-indirect block within each inode. This doubly-indirect block will contain 256 addresses of singly-indirect blocks, with each singly-indirect block containing up to 256 addresses of data blocks. With this structure, a file can consist of up to 65803 blocks, calculated as `256*256 + 256 + 11` blocks. The count will use 11 direct blocks instead of 12 because one of the direct block entries needs to store the address of the doubly-indirect block.

11.2 Implementation

The implementation initially requires us to modify some constant definitions.

Change `#define NDIRECT` from `12` to `11`.

11.2.1 `NDIRECT` (`kernel/fs.h`)

```
#define NDIRECT 11
```

Change `#define MAXFILE` accordingly.

11.2.2 `MAXFILE` (`kernel/fs.h`)

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

Modify the declaration of `addrs[]` in `struct dinode` from `uint addrs[NDIRECT + 1]` to `[NDIRECT + 2]`.

11.2.3 `dinode{}` (`kernel/fs.h`)

```
// On-disk inode structure
struct dinode {
  ...
  uint addrs[NDIRECT+2]; // Data block addresses
};
```

Modify the declaration of `addrs[]` in `struct inode` from `uint addrs[NDIRECT + 1]` to `[NDIRECT + 2]`.

11.2.4 `inode{}` (`kernel/file.h`)

```
// in-memory copy of an inode
struct inode {
  ...
  uint addrs[NDIRECT+2];
};
```

We need to modify the `bmap()` function to support the doubly-indirect block. The `bmap()` function is responsible for translating a logical block number within a file to a physical block number on the disk.

The implementation is as follows:

1. Adjust the block number `bn` to account for the blocks already addressed by the singly-indirect block.
2. Check if the requested block number falls within the range addressable by the doubly-indirect block.
3. Check if the doubly-indirect block has been allocated; if not, allocate a new block using `balloc()`.
4. Read the doubly-indirect block from the disk into `bp` (buffer); then, cast the data into `a`.
5. Check if the respective singly-indirect block has been allocated; if not, allocate a new block using `balloc()`.
6. Release `bp` (buffer) for the doubly-indirect block.
7. Read the singly-indirect block from the disk into `bp` (buffer); then, cast the data into `a`.
8. Check if the respective direct block has been allocated; if not, allocate a new block using `balloc()`.
9. Release `bp` (buffer) for the singly-indirect block.
10. Return the address of the final data block.

11.2.5 bmap() (kernel/fs.c)

```
// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
static uint
bmap(struct inode *ip, uint bn)
{
    ...
    bn -= NINDIRECT;
    if (bn < (NINDIRECT * NINDIRECT)) {
        if ((addr = ip->addrs[NDIRECT + 1]) == 0) {
            ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);
        }
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;
        if ((addr = a[bn / NINDIRECT]) == 0) {
            a[bn / NINDIRECT] = addr = balloc(ip->dev);
            log_write(bp);
        }
        brelse(bp);
        bp = bread(ip->dev, addr);
        a = (uint*)bp->data;
        if ((addr = a[bn % NINDIRECT]) == 0) {
            a[bn % NINDIRECT] = addr = balloc(ip->dev);
            log_write(bp);
        }
        brelse(bp);
        return addr;
    }
    panic("bmap: out of range");
}
```

We also need to modify `itrunc()` accordingly to free all blocks of a file including doubly-indirect blocks. The `itrunc()` function is used to truncate a file, which effectively discards its contents.

The implementation is as follows:

1. Add a new `if` statement to handle the freeing of the doubly-indirect block.
2. Read the doubly-indirect block from the disk into `bp` (buffer); then, cast the data into `a`.
3. Iterate through the doubly-indirect block (outer loop). At each iteration...
    - Read the singly-indirect block from the disk into `bpi` (buffer); then, cast the data into `b`.
    - Iterate through the singly-indirect block (inner loop) to free all the data blocks using `bfree()`.
    - Release `bpi` (buffer) for the singly-indirect block.
    - Free the singly-indirect block itself using `bfree()`.
4. After the outer loop is complete...
    - Release `bp` (buffer) for the doubly-indirect block.
    - Free the doubly-indirect block itself using `bfree()`.
5. Clear the entry for the doubly-indirect block in the inode.

11.2.6 itrunc() (kernel/fs.c)

```
// Truncate inode (discard contents).
// Only called when the inode has no links
// to it (no directory entries referring to it)
// and has no in-memory reference to it (is
// not an open file or current directory).
static void
itrunc(struct inode *ip) {
    int i, j, k;
    struct buf *bp, *bpi;
    uint *a, *b;
    ...
    if (ip->addrs[NDIRECT + 1]) {
        bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
        a = (uint*)bp->data;
        for (j = 0; j < NINDIRECT; j++) {
            if (a[j]) {
                bpi = bread(ip->dev, a[j]);
                b = (uint*)bpi->data;
                for (k = 0; k < NINDIRECT; k++) {
                    if (b[k]) {
                        bfree(ip->dev, b[k]);
                    }
                }
                brelse(bpi);
                bfree(ip->dev, a[j]);
            }
        }
        brelse(bp);
        bfree(ip->dev, ip->addrs[NDIRECT + 1]);
        ip->addrs[NDIRECT + 1] = 0;
    }
    ip->size = 0;
    iupdate(ip);
}
```

12 Symbolic Links

12.1 Objective

Symbolic links, also known as soft links, refer to a linked file by its pathname. When a symbolic link is opened, the kernel follows the link and resolves it to the file it references. Symbolic links are similar to hard links, but there are important differences between them. Hard links are restricted to pointing to files on the same disk, whereas symbolic links can reference files across different disk devices.

12.2 Implementation

We first need to implement the `sys_symlink()` system call.

The implementation is as follows:

1. Declare arrays to store the target and path strings. Retrieve the target and path arguments from the system call (return -1 if there is an error with this).
2. Begin the file system transaction.
3. Create a new inode for the symbolic link and check if the creation was successful (return -1 otherwise).
4. Calculate the length of the target path string.
5. Write the length of the target path to the inode.
6. Write the target path string to the inode and finish it with a null terminator.
7. Copy the modified in-memory inode to disk.
8. Unlock and drop the reference to the inode.
9. End the file system transaction.

12.2.1 sys_symlink() (kernel/sysfile.c)

```
uint64
sys_symlink(void)
{
    //your implementation goes here
    char target[MAXPATH], path[MAXPATH];
    if ((argstr(0, target, MAXPATH) < 0) || (argstr(1, path, MAXPATH) < 0))
       {
        return -1;
    }
    begin_op(ROOTDEV);
    struct inode *ip = create(path, T_SYMLINK, 0, 0);
    if (ip == 0) {
        end_op(ROOTDEV);
        return -1;
    }
    // writei(ip, 0, (uint64)target, 0, (strlen(target) + 1));
    int len = strlen(target);
    writei(ip, 0, (uint64)&len, 0, sizeof(int));
    writei(ip, 0, (uint64)target, sizeof(int), len + 1);

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

We also need to modify the `sys_open()` system call to support symbolic links. To accomplish this, we will add a new `if` statement to the system call.

The implementation is as follows:

- Check if the inode represents a symbolic link, and that the `O_NOFOLLOW` flag is not set.
- Initialize the `depth` variable to `0`.
- Move into the `while` loop.
  - At the beginning of each iteration, check if `depth` exceeds `10`; if it does, we have encountered a cycle. In this case, we release the inode, end the operation, and return `-1` as an error code.
  - Read the length of the target path from the inode.
  - Read the target path string itself, including the null terminator at the end.
  - Unlock and drop the reference to the inode.
  - Resolve the path name and lock the inode.
  - Increment `depth`.

12.2.2 sys_open() (kernel/sysfile.c)

```
uint64
sys_open(void)
{
    char target[MAXPATH], path[MAXPATH];
    ...
    if ((ip->type == T_SYMLINK) && ((omode & O_NOFOLLOW) == 0)) {
        int depth = 0;
        while ((ip->type == T_SYMLINK) && ((omode & O_NOFOLLOW) == 0)) {
            if (depth > 10) {
                iunlockput(ip);
                end_op(ROOTDEV);
                return -1;
            }
            int len = 0;
            readi(ip, 0, (uint64)&len, 0, sizeof(int));
            readi(ip, 0, (uint64)target, sizeof(int), len + 1);
            iunlockput(ip);
            ip = namei(target);
            if (ip == 0) {
                end_op(ROOTDEV);
                return -1;
            }
            ilock(ip);
            depth++;
        }
    }
    ...
    iunlock(ip);
    end_op(ROOTDEV);
    return fd;
}
```

MMAP

13 Objective

The `mmap` and `munmap` system calls allow UNIX programs to exert detailed control over their address spaces. They can be used to share memory among processes, to map files into process address spaces, and as part of user-level page fault schemes such as garbage-collection algorithms.

14 Implementation

The implementation initially requires us to add flags/definitions for the syscalls. To successfully compile the code without errors, we need to add "include guards" at the beginning of `file.h`, `fs.h`, and `sleeplock.h`.

Add `$U/_mmaptest\` in `Makefile` to add `mmaptest` test.

14.0.1 `Makefile`

```
+ $U/_mmaptest\
```

Add `entry("mmap"); entry("munmap");` in `user/usys.pl`.

14.0.2 `user/usys.pl`

```
+entry("mmap");
+entry("munmap");
```

Add the following function definitions in `user/user.h`.

14.0.3 `user/user.h`

```
+void* mmap (void*, unsigned int, int, int, int, unsigned int);
+int munmap (void*, unsigned int);
```

Add the following definitions in `kernel/syscall.h`.

14.0.4 `kernel/syscall.h`

```
+#define SYS_mmap 23
+#define SYS_munmap 24
```

Add the following mapping in `kernel/syscall.c`.

14.0.5 kernel/syscall.c

```
+[SYS_mmap] sys_mmap,
+[SYS_munmap] sys_munmap,
```

Add the following definitions in `kernel/syscall.c`.

14.0.6 kernel/syscall.c

```
+extern uint64 sys_mmap(void);
+extern uint64 sys_munmap(void);
```

Add the following definitions in `kernel/fcntl.h`.

14.0.7 kernel/fcntl.h

```
+#define PROT_READ 0x001
+#define PROT_WRITE 0x002
+#define MAP_PRIVATE 0x001
+#define MAP_SHARED 0x002
```

We need to define the VMA structure (recording start, length, permission, flags and file for each mapped memory range).

14.0.8 kernel/proc.h

```
+struct vma {
+ uint64 addr;
+ int len;
+ int prot;
+ int flags;
+ struct file *f;
+ int offset;
+ int valid;
+};
```

We need to add a table in `struct proc` of all the VMAs for a process. Since the `Xv6` kernel doesn't have a memory allocator in the kernel, it's OK to declare a fixed size array of VMAs and allocate from that array as needed. A size of 16 should be sufficient.

14.0.9 kernel/proc.h

```
+ struct vma vma_table[16];
```

We also need to initialize all members of the VMA struct for all VMAs in `allocproc()`.

14.0.10 `kernel/proc.c`

```
+ for(int i = 0; i < 16; ++i) {
+   p->vma_table[i].addr = 0;
+   p->vma_table[i].len = 0;
+   p->vma_table[i].prot = 0;
+   p->vma_table[i].flags = 0;
+   p->vma_table[i].f = 0;
+   p->vma_table[i].offset = 0;
+   p->vma_table[i].valid = 0;
+ }
```

We can now complete the implementation of `mmap()`.
The implementation is as follows:

1. Validate the input of all arguments (`addr`, `len`, `prot`, `flags`, `f`, and `offset`). Return an error if any of the arguments are negative.
2. Return an error if the file is not readable and read permission is requested.
3. Return an error if the file is not writable and write permission is requested.
4. Iterate through the VMA table to find a free slot. Return an error if no free slot is found.
5. Iterate through the VMA table to find an unused region in which to map the file. If there is overlap, adjust the virtual address to be after the end of the VMA in use.
6. Initialize all fields of the chosen VMA entry (which was found in Step 4).
7. Increase the file's reference count.
8. Return the starting virtual address of the mapped region.

14.0.11 kernel/sysfile.c

```
+uint64
+sys_mmap(void)
+{
+ uint64 addr;
+ int len;
+ int prot;
+ int flags;
+ struct file *f;
+ int offset;
+
+ if (argaddr(0, &addr) < 0) {
+   return -1;
+ }
+ if (argint(1, &len) < 0) {
+   return -1;
+ }
+ if (argint(2, &prot) < 0) {
+   return -1;
+ }
+ if (argint(3, &flags) < 0) {
+   return -1;
+ }
+ if (argfd(4, 0, &f) < 0) {
+   return -1;
+ }
+ if (argint(5, &offset) < 0) {
+   return -1;
+ }
+
+ if (!(f->readable) && (prot & PROT_READ)) {
+   return -1;
+ }
+ if (!(f->writable) && (prot & PROT_WRITE) && !(flags & MAP_PRIVATE)) {
+   return -1;
+ }
+
+ struct proc *p = myproc();
+ struct vma *vma = 0;
+ for (int i = 0; i < 16; ++i) {
+   if ((p->vma_table[i].valid == 0)) { // found one
+     vma = &p->vma_table[i];
+     break;
+   }
+ }
+ if (vma == 0) {
+   return -1;
+ }
+
+ uint64 va = 0x40000000; // starting address 0x40000000
+ for (int i = 0; i < 16; ++i) {
+   if (p->vma_table[i].valid == 1) { // 1 = used
+     uint64 start_va = p->vma_table[i].addr;
+     uint64 end_va = (p->vma_table[i].addr + p->vma_table[i].len);
+     if ((va >= start_va) && (va < end_va)) {
+       va = PGROUNDUP(end_va);
+     }
```

```
+    }
+  }
+
+  vma->addr = va;
+  vma->len = len;
+  vma->prot = prot;
+  vma->flags = flags;
+  vma->f = f;
+  vma->offset = offset;
+  vma->valid = 1;
+  filedup(vma->f);
+
+  return va;
+}
```

We can now complete the implementation of `usertrap()` (similar to Lazy Page Allocation). The implementation is as follows:

1. Identify that a page fault has occurred (`r_scause() = 13` or `15`).
2. Obtain the faulting address.
3. Iterate through the VMA table and find the relevant VMA (the VMA that contains the faulting address). Kill the process if the VMA is not found.
4. Round down the faulting address to the nearest page boundary.
5. Calculate the offset within the mapped file that corresponds to the faulting address.
6. Allocate a new physical page using `kalloc()`. Kill the process if allocation fails.
7. Initialize the new physical page to zero.
8. If the page fault was a result of a load/store operation (`r_scause() = 15`) and the mapping is set to `MAP_PRIVATE`, skip the file read.
9. Read 4096 bytes of the relevant file into the page using `readi()`.
10. Set the appropriate flags for the PTE (`PTE_R?`, `PTE_W?` + `PTE_U`).
11. Map the allocated physical page into the user's address space.

14.0.12 kernel/trap.c

```
+ else if ((r_scause() == 13) || (r_scause() == 15)) {
+   uint64 va = r_stval();
+   struct vma *vma = 0;
+
+   for (int i = 0; i < 16; ++i) {
+     if (p->vma_table[i].valid == 0) {
+       continue;
+     }
+     uint64 start_va = p->vma_table[i].addr;
+     uint64 end_va = (p->vma_table[i].addr + p->vma_table[i].len);
+     if ((va >= start_va) && (va < end_va)) {
+       vma = &p->vma_table[i];
+       break;
+     }
+   }
+   if (vma == 0) {
+     p->killed = 1;
+     goto end;
+   }
+
+   uint64 fault_va = PGROUNDDOWN(va);
+   int offset = vma->offset + (fault_va - vma->addr);
+   pte_t *pte = walk(p->pagetable, fault_va, 0);
+
+   char *mem = kalloc();
+   if (mem == 0) {
+     p->killed = 1;
+     goto end;
+   }
+   memset(mem, 0, PGSIZE);
+
+   if ((r_scause() == 15) && (vma->flags & MAP_PRIVATE)) {
+     if (pte) {
+       goto skip;
+     }
+   }
+
+   struct file *f = vma->f;
+   begin_op(f->ip->dev);
+   ilock(f->ip);
+   readi(f->ip, 0, (uint64)mem, offset, PGSIZE);
+   iunlock(f->ip);
+   end_op(f->ip->dev);
+   int flags = PTE_U;
+   if (vma->prot & PROT_READ) {
+     flags |= PTE_R;
+   }
+   if (vma->prot & PROT_WRITE) {
+     if (!(vma->flags & MAP_PRIVATE)) {
+       flags |= PTE_W;
+     }
+   }
+
+ skip:
+   if (mappages(p->pagetable, fault_va, PGSIZE, (uint64)mem, PTE_W|PTE_R|
    PTE_U) != 0) {
```

```
+      p->killed = 1;
+      kfree(mem);
+      goto end;
+    }
+  end:
+  }
```

We can now complete the implementation of `munmap()`. The implementation is as follows:

1. Validate the input of all arguments (`addr`, `len`). Return an error if any of the arguments are negative.
2. Iterate through the VMA table and find the relevant VMA (the VMA that overlaps with the region being unmapped). Return an error if the VMA is not found.
3. If the VMA was mapped with the `MAP_SHARED` flag, write the contents of the unmapped page back to the mapped file.
4. Unmap the specific pages using `uvmunmap()`.
5. If `munmap` has removed all pages of a previous `mmap`, decrement the reference count of the corresponding `struct file`.
6. Return a success (0) if the function completes successfully.

14.0.13 kernel/sysfile.c

```
+uint64
+sys_munmap(void)
+{
+ uint64 addr;
+ int len;
+
+ if (argaddr(0, &addr) < 0) {
+   return -1;
+ }
+ if (argint(1, &len) < 0) {
+   return -1;
+ }
+
+ struct proc *p = myproc();
+ struct vma *vma = 0;
+ for (int i = 0; i < 16; ++i) {
+   if (p->vma_table[i].valid == 0) {
+     continue;
+   }
+   uint64 start_va = p->vma_table[i].addr;
+   uint64 end_va = (p->vma_table[i].addr + p->vma_table[i].len);
+   if ((addr >= start_va) && (addr < end_va)) {
+     vma = &p->vma_table[i];
+     break;
+   }
+ }
+ if (vma == 0) {
+   return -1;
+ }
+
+ if (vma->flags & MAP_SHARED) {
+   for (uint64 va = addr; va < (addr + len); va += PGSIZE) {
+     pte_t *pte = walk(p->pagetable, va, 0);
+     if (pte) {
+       int offset = (va - (vma->addr + vma->offset));
+       begin_op(vma->f->ip->dev);
+       ilock(vma->f->ip);
+       writei(vma->f->ip, 1, addr, offset, len);
+       iunlock(vma->f->ip);
+       end_op(vma->f->ip->dev);
+     }
+   }
+ }
+
+ uvmunmap(p->pagetable, addr, len, 1);
+
+ if (addr == vma->addr) {
+   if (len == vma->len) {
+     fileclose(vma->f);
+     vma->valid = 0;
+   }
+ }
+
+ return 0;
+}
+
```

We also need to add some additional code to `exit()` and `fork()` in order to pass the entirety of `mmaptest`.

Modify `exit` to unmap the process's mapped regions as if `munmap` had been called.

14.0.14 kernel/proc.c

```
+ for (int i = 0; i < 16; ++i) {
+   if (p->vma_table[i].valid == 1) {
+     uvmunmap(p->pagetable, p->vma_table[i].addr, p->vma_table[i].len, 1);
+     if (p->vma_table[i].f) {
+       fileclose(p->vma_table[i].f);
+     }
+     p->vma_table[i].valid = 0;
+   }
+ }
```

Modify `fork` to ensure that the child has the same mapped regions as the parent. Increment the reference count for a VMA's `struct file`.

14.0.15 kernel/proc.c

```
+ for (int i = 0; i < 16; ++i) {
+   if (p->vma_table[i].valid == 1) {
+     np->vma_table[i] = p->vma_table[i];
+     if (p->vma_table[i].f) {
+       filedup(p->vma_table[i].f);
+     }
+   }
+ }
```

To avoid panics due to Lazy Allocation, we also implement modifications to `uvmcopy()` and `uvmunmap()`. In both functions, we simply comment out the panics and continue iterating through the loop.

14.0.16 kernel/vm.c

```
+ if ((pte = walk(pagetable, a, 0)) == 0) {
+   // panic("uvmunmap: walk");
+   a += PGSIZE;
+   if (a > last) {
+     break;
+   }
+   continue;
+ }
+ if ((*pte & PTE_V) == 0) {
+   // printf("va=%p pte=%p\n", a, *pte);
+   // panic("uvmunmap: not mapped");
+   a += PGSIZE;
+   if (a > last) {
+     break;
+   }
+   continue;
+ }
```

14.0.17 kernel/vm.c

```
+ if((pte = walk(old, i, 0)) == 0) {
+   // panic("uvmcopy: pte should exist");
+   continue;
+ }
+ if((*pte & PTE_V) == 0) {
+   // panic("uvmcopy: page not present");
+   continue;
+ }
```

References

[1] Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2023). *Operating Systems: Three Easy Pieces* (Version 1.10). Arpaci-Dusseau Books.

[2] Cox, R., Kaashoek, F., & Morris, R. (2020). *xv6: A simple, Unix-like teaching operating system*. MIT CSAIL. `https://pdos.csail.mit.edu/6.S081/2020/xv6/book-riscv-rev1.pdf`

[3] Lions, J. (2000). *Lion's commentary on UNIX 6th edition*. Peer-to-Peer Communications.