Diego Avina-Escobedo

Shirley Moore

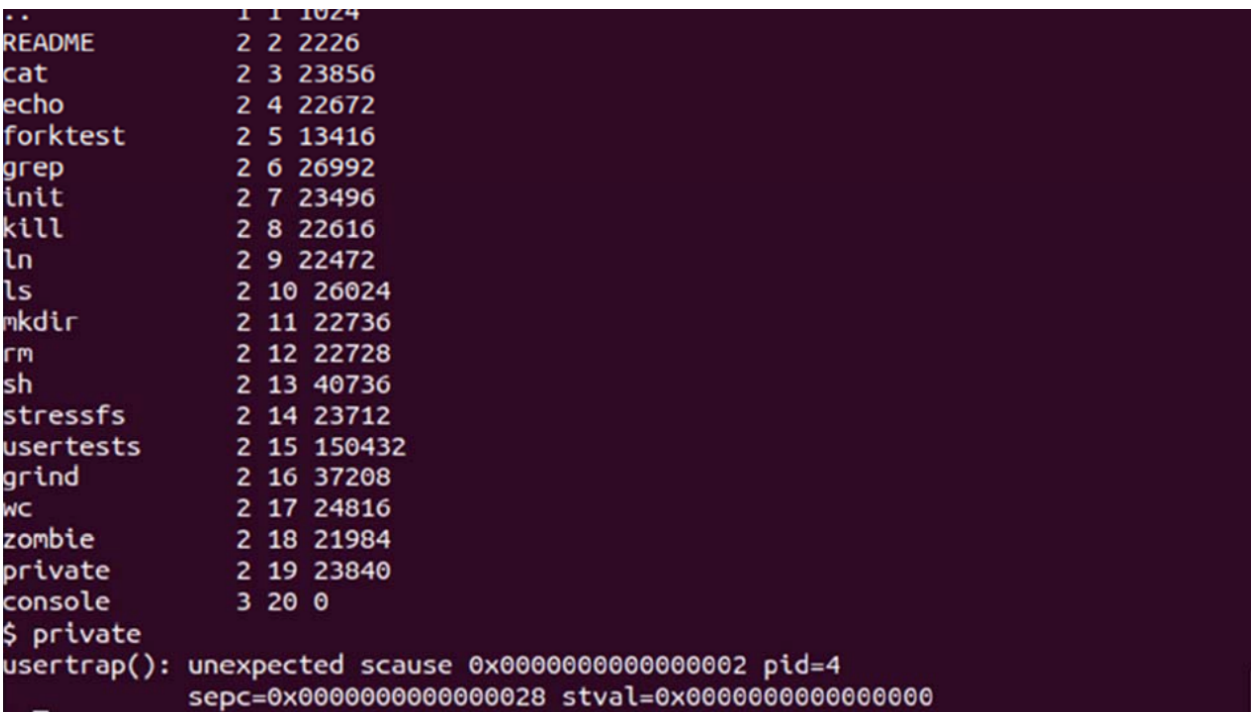
CS 4375

HW 5 Report:

Task 1: Implementing mmap() and munmap() with Lazy Allocation:

a) Results of Running Private Program:

I successfully implemented mmap() and munmap() following the instructions in the "Anonymous mmap() and munmap()" handout. The private program uses mmap() to allocate a region of virtual memory and munmap() to deallocate it. However, nefore modifying usertrap() in kernel/trap.c for part b, the program aborted due to a fault that was not handled



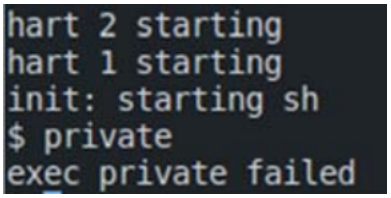
b) Fixing usertrap() for Private Program:

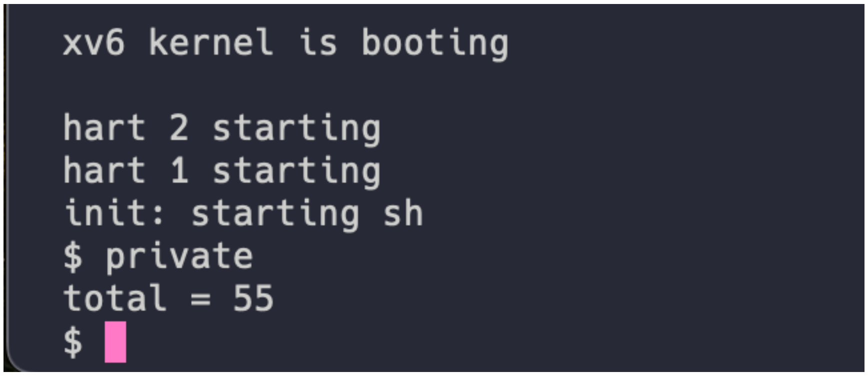
To addres the fault issue, I modified usertrap() in kernel/trap.c. Specifically, I added checks for scause being either a load fault (13) or a store fault (15) and ensured that the fault address falls within a mapped memory region with the correct permissions. Additionally, I allocated a physical memory frame using kalloc(), mapped it into the process's page table using mappages(), and rounded the fault address down to a page boundary. After these modifications, the private program ran successfully.

```
void usertrap(void) {
    int which_dev = 0;
    struct proc *p = myproc();

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

    w_stvec((uint64)kernelvec);
    p->trapframe->epc = r_sepc();

    if (r_scause() == 8) {
        // Handle system call
        if (p->killed) exit(-1);
        p->trapframe->epc += 4;
        intr_on();
        syscall();
    } else if ((which_dev = devintr()) != 0) {
        // Handle device interrupts
    } else if (r_scause() == 13 || r_scause() == 15) {
        // Handle page fault
        if (!is_valid_address(r_stval(), p)) {
            p->killed = 1;
            exit(-1);
        }
        void *physical_mem = kalloc();
        if (!physical_mem) {
            printf("usertrap(): out of memory\n");
            p->killed = 1;
            exit(-1);
        }
        if (mappages(p->pagetable, PGROUNDDOWN(r_stval()), PGSIZE, (uint64)physical_mem, PTE_R | PTE_W | PTE_X | PTE_U) < 0) {
            kfree(physical_mem);
            printf("usertrap(): mappages failed\n");
            p->killed = 1;
            exit(-1);
        }
    } else {
        // Handle unexpected traps
        printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
                                                                        77,12          20%
```

c) Handling munmap() in freeproc()

Initially, commenting out the call to munmap in private.c and running the program caused a kernel panic. This was because the physical memory for the mapped memory region was not freed in freeproc(). I added the provided code in the handout to freeproc(), preventing the kernel panic.





Task 2: Modifying fork() for Shared Memory Inheritance

a) **Difference between uvmcopy() and uvmcopyshared()**

For this part of task 2, I made the necessary changes to uvmcopy() in vm.c and added uvmcopyshared(). The difference between the two lies on how the handle private and shared mappings. Uvmcopy() copues the memory region as is, while uvmcopyshared() ensures that changes made by any process in a shared mapping are reflected across all processes shsaring that mapping.

b) **Modifying fork() for inheriting Shared Memory**

I added code to fork() in proc.c to copy the memory region table from parent to child, ensuring proper inheritance of both private and shared mappings. The code works for both PRIVATE and SHARED regions, preserving the isolation of private mappings and allowing shared mappings to reflect changes across processes

c) **Testing with prodcons1.c and prodcons2.c**

I compiled the modified code and tested with prodcons1.c and prodcons2.c. The results were as expected, with procdons1.c showing a total of 55 and prodcons2.c showing a total of 0.

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ prodcons1
total = 55
$ prodcons2
total = 0
$
```

Task 3: Handling Multiple Pages in prodcons3.c

a) **Incorrect Results in prodcons3.c**

Running prodcons4.c, which maps three pages, initially produced incorrect results. The trap handler in usertrap() needed modifications to properly handle situations where a process other than the one that originally mapped a shared memory region is the first to write to it.

b) **Extra Credit: Handling Shared Memory Writes**

I modified the trap handler in usertrap() to handle situations where a different process writes to a shared memory region first. The code now correctly inserts the new mapping for tha allocated physical page into the page table for all processes in the family that have the shared memory region mapped.

```
$ prodcons3
total = 673720320
$
```

Summary:

Through this assignment, I gained a deeper understanding of Linux memory mapping concepts, page tables, and shared memory regions. Implementing mmap() and munmap(), along with modifying for() for shared memory inheritance, provided value insights into xv6's memory management.