

UG HW5: Anonymous Memory Mappings for xv6

<https://github.com/Rcarmona/myxv6RubenCarmona.git>

Task 1.

- a. Here we added `PGROUNDDOWN(p->cur_max - length);` to potentially align it to a specific boundary or page size using the `PGROUNDDOWN` macro or function.

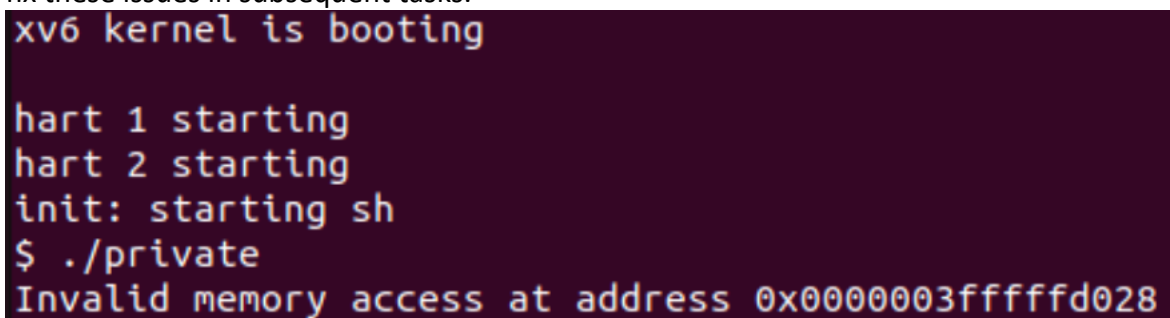
```
start_addr = PGROUNDDOWN(p->cur_max - length);
```

Here we define a system call function 'sys_munmap' that extracts two arguments, and address and a length, from the user's space, then calls the 'munmap' function with these arguments and returns its result.

```
// Get argument and call munmap() helper function
uint64
sys_munmap(void)
{
    uint64 addr;
    uint64 length;

    if (argaddr(0, &addr) < 0)
        return -1;
    if (argaddr(1, &length) < 0)
        return -1;
    return (munmap(addr, length));
}
```

Here we encounter an invalid memory access error, likely due to incorrect memory operations, uninitialized memory, or a kernel bug, as part of the task is to address and fix these issues in subsequent tasks.



```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ ./private
Invalid memory access at address 0x0000003fffffd028
```

- b. Describe how you fixed the `usertrap()` from part a. Show the result of running `private` after you fixed the problem. Describe any difficulties you encountered with this task and how you overcame them.
- This function is integral to an operating system's kernel, designed to respond when a user-mode program encounters issues like system calls or memory faults.

Essentially, it's like a gatekeeper that decides how the system should react to scenarios. Understanding 'usertrap' give us insight into how operating systems maintain stability and security in the face of unpredictable events during program execution. Below we can find the code that was needed to be fix, checking scause is either 13 (load fault) or 15 (store fault).

'usertrap' code

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

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

    w_stvec((uint64)kernelvec);

    struct proc *p = myproc();
    p->trapframe->epc = r_sepc();

    if (r_scause() == 0) {
        if (p->killed)
            exit(-1);

        p->trapframe->epc += 4;
        intr_on();

        syscall();
    } else if (which_dev == devintr()) != 0) {
        // ok
    } else if (r_scause() == 13 || r_scause() == 15) {
        // Check mapped region protection permits operation
        if (r_stval() >= p->sz) {
            for (int i = 0; i < MAX_MMIO; i++) {
                if (p->mmr[i].valid && p->mmr[i].addr < r_stval() && p->mmr[i].length > r_stval()) {
                    // Page fault load
                    if (r_scause() == 13) {
                        // Read permission
                        if ((p->mmr[i].prot & PROT_READ) == 0) {
                            p->killed = 1;
                            exit(-1);
                        }
                    }
                    // Page fault store
                    if (r_scause() == 15) {
                        // Write permission
                        if ((p->mmr[i].prot & PROT_WRITE) == 0) {
                            p->killed = 1;
                            exit(-1);
                        }
                    }
                }
            }

            void *physical_frame = kalloc();
            if (physical_frame) {
                if (mappages(p->pagetable, PADDR(r_stval()), PGSIZE, (uint64)physical_frame, (PTE_R | PTE_W | PTE_X | PTE_U) < 0)) {
                    kfree(physical_frame);
                    p->killed = 1;
                }
            }
        }
    } else {
        printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
        printf("             sepc=%p stval=%p\n", r_sepc(), r_stval());
        p->killed = 1;
    }

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

    if (which_dev == 2)
        yield();

    usertrapret();
}
```

Output

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ private
total = 55
```

- c. Explain why running the private program with the call to munmap() commented out initially caused a kernel panic before you added the code for part c to freeproc(). Under what conditions does the physical memory for a mapped memory region need to be freed in freeproc() and what happens if this isn't done?
 - The kernel panic occurs because 'munmap()' is responsible for unmapping a region of memory that was previously mapped with 'mmap()', when we comment this line

of code the memory remains mapped even after the process is completed, which can lead to state where the kernel tries to access or free memory that is no longer valid or has been reassigned to another process. The code added fixes the kernel panic by properly handling the unmapping of both private and shared memory regions upon process termination. By ensuring that all memory mappings are appropriately deal with, the kernel avoids trying to access invalid memory, which would otherwise cause a panic.

Commenting 'munmap'

```
int
main(int argc, char *argv[])
{
    buffer = (buffer_t *) mmap(NULL, sizeof(buffer_t),
                               PROT_READ | PROT_WRITE,
                               MAP_ANONYMOUS | MAP_PRIVATE,
                               -1, 0);

    buffer->nextin = 0;
    buffer->nextout = 0;
    buffer->num_produced = 0;
    buffer->num_consumed = 0;
    buffer->total = 0;

    producer();
    consumer();

    printf("total = %d\n", buffer->total);

    //munmap(buffer, sizeof(buffer_t));

    exit(0);
}
```

Panic output

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ private
total = 55
panic: freewalk: leaf
```

Code added to 'freeproc'

```
freeproc(struct proc *p)
{
    if(p->trapframe)
        kfree((void*)p->trapframe);
    p->trapframe = 0;

    for (int i = 0; i < MAX_MMR; i++) {
        int dofrees = 0;
        if (p->mmr[i].valid == 1) {
            if (p->mmr[i].flags & MAP_PRIVATE)
                dofrees = 1;
            else { // MAP_SHARED
                acquire(&mmr_list[p->mmr[i].mmr_family.listid].lock);
                if (p->mmr[i].mmr_family.next == &(p->mmr[i].mmr_family)) { // no other family members
                    dofrees = 1;
                    release(&mmr_list[p->mmr[i].mmr_family.listid].lock);
                    dealloc_mmr_listid(p->mmr[i].mmr_family.listid);
                } else { // remove p from mmr family
                    (p->mmr[i].mmr_family.next)->prev = p->mmr[i].mmr_family.prev;
                    (p->mmr[i].mmr_family.prev)->next = p->mmr[i].mmr_family.next;
                    release(&mmr_list[p->mmr[i].mmr_family.listid].lock);
                }
            }
        }
        // Remove region mappings from page table
        for (uint64 addr = p->mmr[i].addr; addr < p->mmr[i].addr + p->mmr[i].length; addr += PGSIZE)
            if (walkaddr(p->pagetable, addr))
                uvmmunmap(p->pagetable, addr, 1, dofrees);
    }

    if(p->pagetable)
        proc_freepagetable(p->pagetable, p->sz);
    p->pagetable = 0;
    p->sz = 0;
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->chan = 0;
    p->killed = 0;
    p->xstate = 0;
    p->state = UNUSED;
}
```

Output after fixing the panic

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ private
total = 55
```

Task 2.

- a. The 'uvmcopy' function is used to create a new page table by copying the memory regions from an old page to a new one, allocating new physical memory for each page, and copying the contents. 'uvmcopyshared' is designed for sharing the physical pages between the old and new page tables without allocating new memory and copying contents. Instead, it maps the new page table entries to the same physical addresses as the old ones, likely with the intention of creating shared memory regions between processes.

'uvmcopy()'

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 start, uint64 end)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;
    char *mem;

    for(i = start; i < end; 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((mem = kalloc()) == 0)
            goto err;
        memmove(mem, (char*)pa, PGSIZE);
        if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
            kfree(mem);
            goto err;
        }
    }
    return 0;

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

'uvmcopyshared()'

```
// Copies the parent process's page table to the child
// Duplicates the page table mappings so that the physical memory is shared
// Returns 0 on success, -1 on failure
int
uvmcopyshared(pagetable_t old, pagetable_t new, uint64 start, uint64 end)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;

    for(i = start; i < end; 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(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){
            goto err;
        }
    }
    return 0;

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

- b. In the modified 'fork()' function, for each valid memory-mapped region in the parent process, if the region is private 'MAP_PRIVATE', 'uvmcopy()' is used to copy the memory to the child, allocating new memory for the child's private use. This ensures that modifications by the child do not affect the parent. For shared regions 'MAP_SHARED', 'uvmcopyshared()' is invoked instead, which creates mappings in the child's page table to the same physical frames as the parent, without allocating new memory. This allows both parent and child processes to see the same data and any changes made by one will be visible to the other.
- c. Show the results of running the prodcons1 and prodcons2 programs. Explain the results, including why they produce different results.

Output for prodcons1.c and prodcons2.c

```
xv6 kernel is booting

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

- Prodcons1 program uses 'MAP_SHARED' in its 'mmap()' call, allowing the producer and consumer child processes to share the memory with the parent process; thus, modifications by children affect the parent's memory space, resulting in a cumulative 'total' of produced items (55). Conversely, 'prodcons2' employs 'MAP_PRIVATE', creating a separate copy-on-write memory for children; modifications by the consumer child do not reflect in the parent's memory, leading the parent to report a total of 0 as it cannot see the consumed items added by the consumer child.

Task 3.

- Explain why the prodcons3 program produces incorrect results before you implement part b.

Output for prodcons3

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ prodcons3
total = 673720320
```

- The 'prodcons3' program yields incorrect results due to unsynchronized access to a shared buffer between concurrently running producer and consumer processes, causing race conditions and potential buffer overflows. The 'MAP_SHARED' flag exacerbates the issues by allowing both child and parent processes to modify the buffer, which can lead to more items produced than intended.
- (Extra credit) In my solution page faults in shared memory regions in usertrap(), I tackled the key challenge of synchronizing shared memory access across multiple processes. When a process attempts to write to a shared memory segment and triggers a page fault, I allocated a new physical frame and then mapped this frame to the page tables of

all processes sharing this memory segment. This requires iterating through a list of these processes and using functions like `mmap`(`&`) for each to update their page tables.

```
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() == 0) {
        if (p->killed)
            exit(-1);

        p->trapframe->epc = 4;

        intr_on();

        syscall();
    } else if ((which_dev = devintr()) != 0) {
        // ok
    } else if (r_scause() == 13 || r_scause() == 15) {
        // Page fault handling
        // Check mapped region protection permits operation
        if (r_stval() >= p->sz) {
            for (int i = 0; i < MAX_MMR; i++) {
                if (p->mmr[i].valid && p->mmr[i].addr <= r_stval() && p->mmr[i].addr + p->mmr[i].length > r_stval()) {
                    // Handle page fault store (write operation)
                    if (r_scause() == 15) {
                        // Check write permission
                        if ((p->mmr[i].prot & PROT_WRITE) == 0) {
                            p->killed = 1;
                            exit(-1);
                        }

                        // Allocate a new frame for the shared memory
                        void *physical_frame = kalloc();
                        if (!physical_frame) {
                            p->killed = 1;
                            break; // Exit the for loop
                        }

                        // Insert the new mapping for the current process
                        if (mmap(p->pagetable, PGROUNDDOWN(r_stval()), PGSIZE, (uint64)physical_frame, PTE_FLAGS(p->pagetable[PGROUNDDOWN(r_stval()) >> PTE_SHIFT] | PTE_U) < 0) {
                            kfree(physical_frame);
                            p->killed = 1;
                            break; // Exit the for loop
                        }

                        // Insert the new mapping for all other processes in the family
                        struct mmr_list *mmr_list = get_mmr_list(p->mmr[i].mmr_family.listid);
                        acquire_lock(&(mmr_list->lock)); // Acquire the lock before accessing the list

                        struct mmr_node *current = mmr_list->head;
                        do {
                            if (current->proc != p) { // Ensure not to remap for the current process
                                if (mmap(current->proc->pagetable, PGROUNDDOWN(r_stval()), PGSIZE, (uint64)physical_frame, PTE_FLAGS(current->proc->pagetable[PGROUNDDOWN(r_stval()) >> PTE_SHIFT] | PTE_U) < 0) {
                                    // Handle error in mapping
                                    // Depending on the implementation, you might want to rollback previous mappings here
                                }
                            }
                            current = current->next;
                        } while (current != mmr_list->head);

                        release_lock(&(mmr_list->lock)); // Release the lock after modifying the list
                    }
                }
            }

            // ... rest of the page fault handling ...
        }
    } else {
        printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
        printf("      %p stval=%p\n", r_sepc(), r_stval());
        p->killed = 1;
    }

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

    if (which_dev == 2)
        yield();

    usertrapret();
}
```

Summary:

Through the implementation of a subset of Linux memory mapping using `mmap`(`&`) and `munmap`(`&`) system calls for private and shared anonymous mappings, this assignment covered essential concepts such as memory mapping, page tables, and lazy allocation, providing insight into how modern operating systems manage memory. I gained practical experience in implementing system calls, handling data types to the xv6 operating system, and managing errors effectively.