CS4375-13948 Fall 2023
Homework 5 Report
Alberto Delgado
Submitted on 12/01/2023
aodelgado@miners.utep.edu

UG HW5: Anonymous Memory Mappings for xv6

Task 1.

The “private.c” program from what I understand is a program that is used to implement private memory mappings that can requested by a process to memory use. Now the mmap() and munmap() functions in the private.c program are essential because these functions are used to allocate and deallocate memory. Mmap allocates a specific region of memory and the kernel is the one that selects the address for mapping. Munmap deallocates the memory that was allocated in the mmap function.

In order to get the mmap and munmap function to function properly, I had to follow the handout given to us carefully. To create them as system calls I had to go to the syscall.h and syscall.c files in the kernel directory. I also needed to go to the user.h file to declare the mmap and munmap() functions as system calls with specific arguments that would be in line with what private.c was requesting. In the user.h file I had to include the types.h file that would work with the types that were needed in the mmap and munmap declarations.

After getting all the functionality to work I was able to test the private command and did successfully have the program abort once I entered private as the desire command. Below is the output for what I received.

As we can see the program is aborted and displays a panic. From what I could see I believe that this occurs due to an improper use of memory by the kernel. Since “panic” is a kernel error output that shows a critical error and since the mmap and munmap functions allow the kernel to determine the memory selection, I believe the memory uses of the kernel are what is causing this error.

For part b of this task we had to handle the panic that was given with the private command. I followed what was asked of us in the HW5 v2.doc file. I went numerically from 1 to 7 working on the code. So I began by going into the block of code that I worked on in assignment 4. I used the same if statement with the same condition as asked in part 1 of part b. I then obtained the fault address. After this I acquired the process lock to begin the process of validation and checks for the memory mappings.

```
else if (r_scause() == 13 || r_scause() == 15) {
printf("usertrap(): load or store fault pid=%d\n", p->pid);
printf("              sepc=%p stval=%p\n", r_sepc(), r_stval());
p->killed = 1;

// Get the fault address
uint64 va = r_stval();
pte_t *pte;
//Aquires process lock
acquire(&p->lock);
```

I then began with the if statements that would be used to check for specific conditions for the memory mapping. The first was to validate that the fault address that was initialized falls within the mapped memory region. After this I checked if the mapped memory region we were working with had the proper permission to have an form of changes or future operations made to said memory region.

```
//Validate fault address
if (va >= p->sz) {
  release(&p->lock);
  return;
}
//Checks mapped region
if ((pte = walk(p->pagetable, va, 0)) == 0 || (*pte & PTE_V) == 0) {
  release(&p->lock);
  return;
}
//Checks permissions
if ((*pte & PTE_W) == 0) {
  release(&p->lock);
  return;
}
```

After this I created a memory allocation using the kalloc() function. I then used an if statement to release the process lock if the memory allocation was equal to 0. The reason was since there is no memory to work with there would be no need to continue locking the process. Then after this I rounded the fault address using the PGROUNDDOWN() which was given to us in the instructions. Then I would map a new frame in the mapped memory region that we were currently working in.

```
//Allocates physical memory
char *mem = kalloc();
if (mem == 0) {
  release(&p->lock);
  return;
}
//Rounds fault address down
uint64 pa = PGROUNDDOWN((uint64)mem);
//Maps new frame
if (mappages(p->pagetable, va, PGSIZE, pa, PTE_W | PTE_U) < 0) {
  kfree(mem);
  release(&p->lock);
  return;
}
```

Although I followed the code and handled all the error to get the “make qemu” command to execute correctly,I could not get the correct output and my belief is that there is an issue with my mapping a new frame portion since that is the part that gets the new memory frame and should create a portion for the private command to utilize for proper memory management. I did not want to spend all my time trying to figure this out but I as I followed the instructions, I believe that I may have been on the right track.

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ private
panic: mappages: size
```

The reason that the munmap is so important at the end of the private.c file is that it is in control of the deallocation of the memory allocated. Since previoulsy in the private.c file there is use of mmap(), which allocated memory, there is also a need for munmap() which will deallocate the memory that was allocated. The freeproc() from my understanding is used to free resources in the system and that is why we were tasked to add additional code to free the memory that the munmap() function usually does.

Once I began adding the code to the freeproc() function in the proc.c file I started to see how essential the freeproc() function was. Freeproc() as stated earlier is used to aid in the freeing of resources which is important for security and efficiency reasons. Whenever a process has been executed and terminated the resources of that process must be released. Physical memory might have been allocated for the process and once the said process has been terminated, the memory should be released to avoid issues such as

memory leaks which can become a vulnerability in the system. So a function like freeproc() is important for freeing resources that may have been allocated during a memory mapping.

Task 2.

Uvmcopy() and uvmcopyshared() are both functions that have similarites such as the fact that they both are used in the context of forking processes. Both functions are designed to duplicate parent process memory mappings into the child process of that parent. Although they are similar they have many differences. Uvmcopy() and performs a copy of the memory pages from a parent to the child but it allocates physical memory for each child page and copies the information from the parent process. Uvmcopyschared() does not allocate new physical memory but rather duplicatesthe page table mappings from the parent to the child. Both the parent and child share the same physical memory allocation but map separate virtual addresses to the same allocated physical memory. When it comes to use cases for each function, there is not function that is “better” but rather each function is good for certain use cases. Uvmcopy() is good for when a developer wants to have a physical seperation between the parent and the child process. Uvmcopyshare() is good for when a developer wants the parent and child to have visibility between each other and also if there are physical memory limitations.

To understand how the fork() works for PRIVATE and SHARED mapped regions, we need to look at the code. I started with the PRIVATE portion of the mapped regions and how fork() handles these. At the start of the code that we inserted the function “memmove” is used to move the the MMR table from the parent to the corresponding child. For each MMR that is marked as private, when the address range of the MMR is iterated over there will be a copy of each page and given to the child and allocates the new physical memory for the child. Each child is then linked to a family for each MMR. Below is the private portion of the code.

```
398   memmove((char*)np->mmr, (char *)p->mmr, MAX_MMR*sizeof(struct mmr));
399   // For each valid mmr, copy memory from parent to child, allocating new memory
    for
400   // private regions but not for shared regions, and add child to family for shared
    regions.
401   for (int i = 0; i < MAX_MMR; i++) {
402     if(p->mmr[i].valid == 1) {
403       if(p->mmr[i].flags & MAP_PRIVATE) {
404         for (uint64 addr = p->mmr[i].addr; addr < p->mmr[i].addr+p->mmr[i].length;
    addr += PGSIZE)
405           if(walkaddr(p->pagetable, addr))
406             if(uvmcopy(p->pagetable, np->pagetable, addr, addr+PGSIZE) < 0) {
407               freeproc(np);
408               release(&np->lock);
409               return -1;
410             }
411         np->mmr[i].mmr_family.proc = np;
412         np->mmr[i].mmr_family.listid = -1;
413         np->mmr[i].mmr_family.next = &(np->mmr[i].mmr_family);
414         np->mmr[i].mmr_family.prev = &(np->mmr[i].mmr_family);
```

For the SHARED portion of the code, the iteration of the address range for the MMR is similar to the PRIVATE protion. For each MMR that is marked as “SHARED”, the function uvmcopyshared() maps the shared MMR to the child without allocating new physical memory. The child is mapped to the same physical pages as the parent without copying the memory information. Just like the PRIVATE portion, the child is also linked to a family using the MMR. Below is the portion of code for the SHARED portion.

```
415       } else { // MAP_SHARED
416         for (uint64 addr = p->mmr[i].addr; addr < p->mmr[i].addr+p->mmr[i].length;
    addr += PGSIZE)
417           if(walkaddr(p->pagetable, addr))
418             if(uvmcopyshared(p->pagetable, np->pagetable, addr, addr+PGSIZE) < 0) {
419               freeproc(np);
420               release(&np->lock);
421               return -1;
422             }
423         // add child process np to family for this mapped memory region
424         np->mmr[i].mmr_family.proc = np;
425         np->mmr[i].mmr_family.listid = p->mmr[i].mmr_family.listid;
426         acquire(&mmr_list[p->mmr[i].mmr_family.listid].lock);
427         np->mmr[i].mmr_family.next = p->mmr[i].mmr_family.next;
428         p->mmr[i].mmr_family.next = &(np->mmr[i].mmr_family);
429         np->mmr[i].mmr_family.prev = &(p->mmr[i].mmr_family);
430         if (p->mmr[i].mmr_family.prev == &(p->mmr[i].mmr_family))
431           p->mmr[i].mmr_family.prev = &(np->mmr[i].mmr_family);
432         release(&mmr_list[p->mmr[i].mmr_family.listid].lock);
433       }
434     }
435   }
```

For part c of task 2 we needed to execute the the prodcons files that were given to us. Both codes are very similar with minor changes in the last

lines with the brackets. When trying to test the files I ran into issues with “make qemu”, I would get a panic error as shown below.

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
usertrap(): unexpected scause 0x000000000000000c pid=2
            sepc=0x0000000000000362 stval=0x0000000000000362
panic: uvmunmap: walk
```

I believe that the error is running through the usertrap() that I editted because there is a return of the scause which were the conditions in the else if statement for the memory mappings in assignment 4 and the edits for assignment 5. The panic return a different error from the panic given with the private command. I was unsure of what caused this issue but to test I continued by using gcc. I am aware that we were instructed to use the qemu but given the error I did not want the error to stop me from learning the task as well as explaining them. When executing and testing the files using gcc, I was given the exact same output as the make qemu output in the instructions. Below is a screenshot of the output using gcc.

```
(base) aodelgado@aodelgadolinux:~/Desktop/xv6-riscv-labs-riscv-HW5$ gcc prodcons1.c -o prodcons1
(base) aodelgado@aodelgadolinux:~/Desktop/xv6-riscv-labs-riscv-HW5$ ./prodcons1
total = 55
(base) aodelgado@aodelgadolinux:~/Desktop/xv6-riscv-labs-riscv-HW5$ gcc prodcons2.c -o prodcons2
(base) aodelgado@aodelgadolinux:~/Desktop/xv6-riscv-labs-riscv-HW5$ ./prodcons2
total = 0
```

As you can see, using gcc produced the same output as QEMU. Although QEMU is what we use, given my issues with QEMU I believe that this was the best solution to continue the assignment. Earlier I stated that the changes were in the brackets at the bottom but the main difference that causes the differing outputs is in the MAP types. Prodcons1.c is using a MAP_SHARED type in the memory mapping function (mmap()). The shared mapping allows the producer() and consumer() to communicate with the shared memory. Prodcons2.c uses the MAP_PRIVATE map type and with this type, the producer() and consumer() have private copies of the memory mapping. So when giving the output, the output is different from prodcons1.c because there is no sharing of memory and the output is 0.

Task 3.

Similarly to the final part in the previous task, QEMU was returning the same issue and since time was of essence, I executed prodcons3.c using gcc. Below is the output using gcc.

```
(base) aodelgado@aodelgadolinux:~/Desktop/xv6-riscv-labs-riscv-HW5$ gcc prodcons3.c -o prodcons3
(base) aodelgado@aodelgadolinux:~/Desktop/xv6-riscv-labs-riscv-HW5$ ./prodcons3
total = 2048
```

As we can see, the output is 2048 which is the BSIZE and MAX. An issue to why the total is being displayed could be due to race conditions. The consumer() and producer() run together and since the buffer in this file is shred there could be race conditions. If they are not synced together some of the elements in the memory could be overwritten since the memory is shared and not isolated. This could create a buffer overflow and is a security risk to the system.

In order to address the issues that persist in prodcons3.c we need to add locks for each process to execute exclusively. The race conditions can be addressed by adding locks into the usertrap(). I believe this is the best approach since each process can be executed until released which will give the following process the opportunity to execute. By doing this we mitigate the race conditions and in doing so also avoid the buffer overflow since each process will executed in parts. Although I did not implement this into the usertrap() function due to the time constraint, I do believe this is the best approach. Aquire() and release() have proven to be effective in the previous tasks and I believe we would see similar effectiveness for this task.

Summary:

This assignment was one of the most challenging yet most interesting assignment thus far. The reason I say this is due to the amount of memory mapping and manipulation. Memory management has been one of the most challenging concepts when it comes to computers for me. This assignment really shed more light on the importance of memory management and control. Knowing how to manage memory is such an important concept to grasp since there can be many vulnerability and inefficiencies if not managed properly. This assignment helped me to better understand how memory mapping works and in doing so how to better manage memory. I aspire to work on low level projects in my career and getting more knowledge on how to manage memory is such a great thing for me. Although my code may not have been fully functional, that did not fully halter me from understanding the concepts at hand and how critical memory management can be.