CS4375-13948 Fall 2023
Ashley Magallanes
avmagallanes@utep.miners.edu

Homework Report 5
November 24, 2023

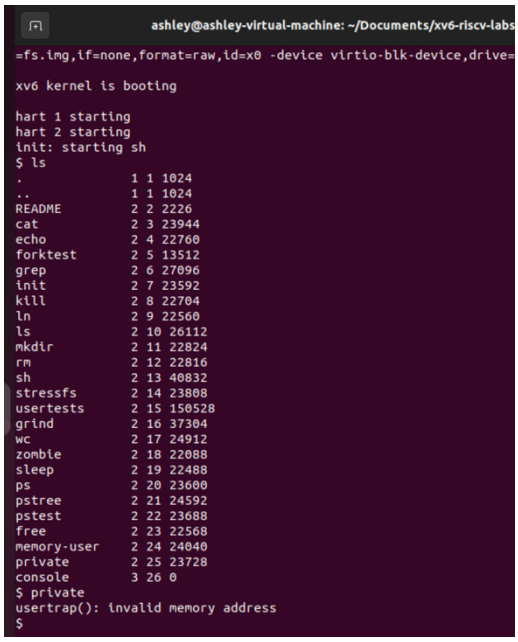
UG HW5: Anonymous Memory Mappings for xv6

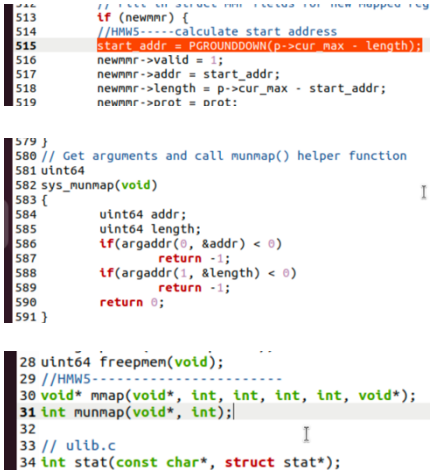
Task 1.

After including all the new code needed to execute the private program provided by Dr.Moore, I encountered some small errors and then procced to fixed them. I calculated the start_addr variable of the new mapped region and also implemented the function sys_munmap by getting the arguments and calling the helper function. Then I also had to include mmap() and munmap() in the defs.h file in order to complete them as system calls and make them available. In addition I also made sure to propagate cur_max to the child process in fork(), by setting the new process's current max to the parent process current max.

The result after all errors were fixed was and the private program was tested was an "exec private failed" error. The private program uses mmap() to map the buffer into memory. While the munmap() is used to release the buffer memory mapped previously by mmap(). The memory is mapped and then used by producer and consumer, then the total is printed and afterwards this memory is released. However, before modifying the usertrap() this program aborts due to the fact that the memory being accessed may be invalid.

The following showcase the result and some of the modifications done for this part of the task number one.





*mmap() and munmap() implementations were done by modifying all needed files

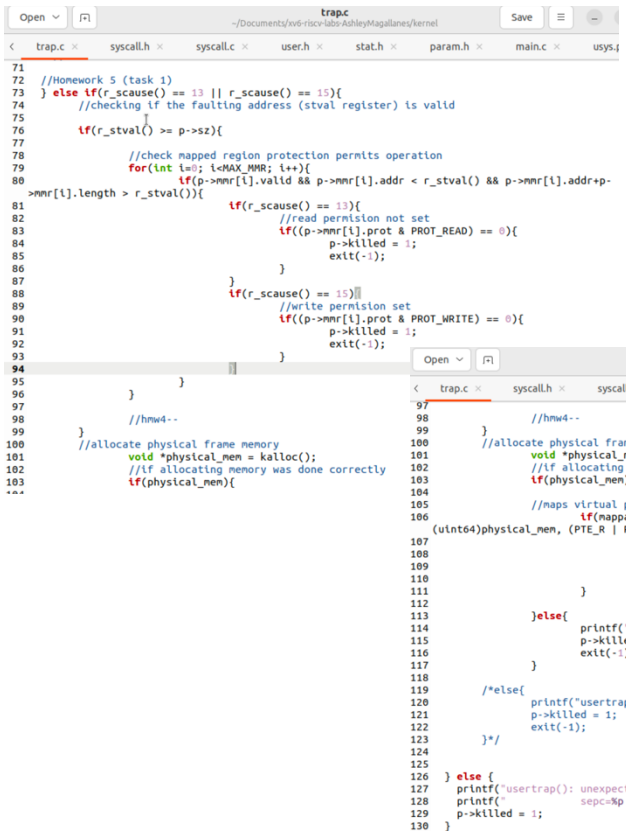
The first portion of this task resulted in the failure of the private command. I fixed this by modifying the previous usertrap() portion of code that I had added in

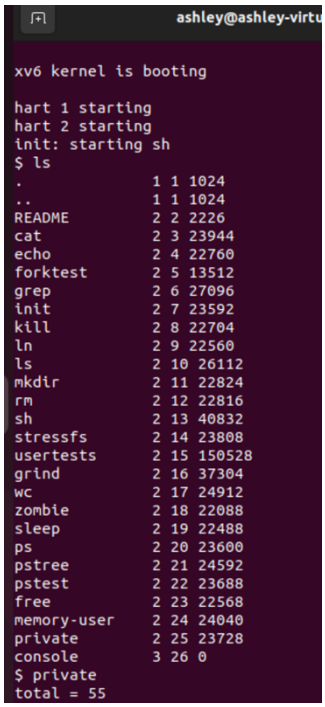
homework assignment number four. I had already checked the scause register and made sure if it was either a load or a store fault. I had also already gotten the fault address and checked if it properly fell within the mapped memory region.

Then I had to make sure that the mapped region protection was permitted by the operation. I did this by modifying the code. I added a for loop that traverses the MAX_MMR and for each I checked either the read or the write permissions. If it is a load then the read permission must be present else the process needs to be killed. If it is a store then the write permission must be present and if it is not then it also needs to be killed. However, I first made sure that the memory was valid or that the space is not empty. I did this by using .valid field of mmr and also making sure that the address is less than the stval register and also the address plus the length is still less. I used the already defined values PROT_READ and PROT_WRITE to check the permission.

Then I continued with the rest of the process needed, which was mainly covered in homework number four. This included the allocation of the memory frame, rounding the fault address and the mapping of the new frame of memory into the process's page table. As mentioned, this was all done in the previous assignment. This resulted in a functioning 'private' command, which outputs 'total = 55' when compiled.

This portion gave me a difficult time because I wasn't too sure on how to check the mapped region permitted the operation but after some guidance it made sense. The following are the images of the trap.c modifications in usertrap() along with the result of private.





When I commented out the call to munmap() near the end of private.c and then attempted to run qemu and the command again the terminal output a kernel panic. The specific output was: 'panic: freewalk: leaf.' This might had happened because the memory mapped is never being unmapped after the execution of private. Munmap is in charge of releasing the virtual memory previously mapped in the kernel, that is why commenting this line of code results in a panic.

After including the new code provided by Dr.Moore in freeproc(), this panic was resolved and the terminal didn't shut down anymore after running the private command. Freeproc() is in charge of freeing this memory that was allocated for the private process. The physical memory for a mapped memory region needs to be freed in freeproc() when the process is terminated, in this case it was freed when private was all done, and due to this the panic was not thrown anymore. However, the memory needs to be freed when its not needed anymore, and when it was allocated beforehand.

```
pstree          2 21 24592
pstest          2 22 23688
free            2 23 22568
memory-user     2 24 24040
private         2 25 23648
console         3 26 0
$ private
total = 55
panic: freewalk: leaf
```

```
ps              2 20 23600
pstree          2 21 24592
pstest          2 22 23688
free            2 23 22568
memory-user     2 24 24040
private         2 25 23648
console         3 26 0
$ private
total = 55
$ QEMU: Terminated
```

```
59
50      //munmap(buffer, sizeof(buffer_t));
51
```

proc.c
~/Documents/xv6-riscv-labs-AshleyMagallanes/kernel

Open Save

proc.c | proc.h | Makefile | private.c | prodcons1.c | prodcons2.c | prodcons3.c

```
157 static void
158 freeproc(struct proc *p)
159 {
160   if(p->trapframe)
161     kfree((void*)p->trapframe);
162   p->trapframe = 0;
163   //HMW5-------------------------------------
164   for (int i = 0; i < MAX_MMR; i++) {
165         int dofree = 0;
166         if (p->mmr[i].valid == 1) {
167         if (p->mmr[i].flags & MAP_PRIVATE)
168         dofree = 1;
169         else { // MAP_SHARED
170         acquire(&mmr_list[p->mmr[i].mmr_family.listid].lock);
171         if (p->mmr[i].mmr_family.next == &(p->mmr[i].mmr_family)) { // no other family members
172         dofree = 1;
173         release(&mmr_list[p->mmr[i].mmr_family.listid].lock);
174         dealloc_mmr_listid(p->mmr[i].mmr_family.listid);
175         } else { // remove p from mmr family
176         (p->mmr[i].mmr_family.next)->prev = p->mmr[i].mmr_family.prev;
177         (p->mmr[i].mmr_family.prev)->next = p->mmr[i].mmr_family.next;
178         release(&mmr_list[p->mmr[i].mmr_family.listid].lock);
179         }
180         }
181         // Remove region mappings from page table
182         for (uint64 addr = p->mmr[i].addr; addr < p->mmr[i].addr + p->mmr[i].length; addr += PGSIZE)
183         if (walkaddr(p->pagetable, addr))
184         uvmunmap(p->pagetable, addr, 1, dofree);
185         }
186   }
187   //-----------------------------------------
188   if(p->pagetable)
189     proc_freepagetable(p->pagetable, p->sz);
190   p->pagetable = 0;
```

Task 2.

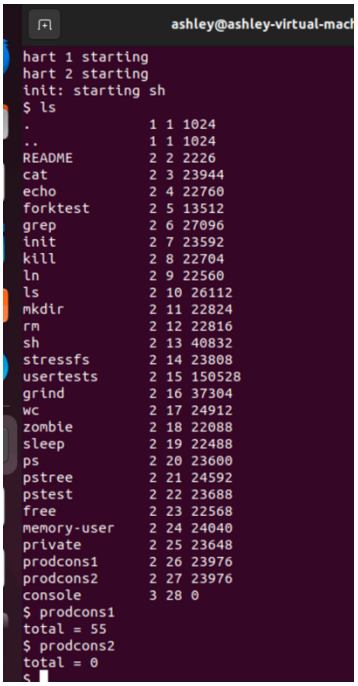
Although uvmcopy() and uvmshared() have the same parameters they both have different functionality. Uvmcopy() allows private mapping parent or child changes to not be shared with all the process, and uvmcopyshared() is almost like the opposite, this method handles mappings that do need to be shared and seen by any other processes. Uvmshared() copies the parent process's page table to the child, and then dupliactes the page table mappings so that the physical memory is shared. Then uvmcopy() also copies the parent's process page table to the child but doesn't duplicate to allow sharing, instead it includes a kfree() for the memory.

The new code added to fork() works for both private and shared mapped regions. This allows the copying of the mmr table form parent to child however it handles both cases. If the mapped region is private, meaning that child process needs to be given a private space of memory since the mapped region is private then each valid

mmr will copy memory form parent to child and allocate new memory for it. For the mapped regions that are shared then the child process np needs to be added to family. The defined value MAP_PRIVATE is used to check for this distinguishment, then based on the output of this value the corresponding method uvmcopy() or uvmcopyshared() is called.

When running prodcons1 this resulted in an output of a total of 55 which is the same output as the private command. While the result of the command prodcons2 was zero. Prodcons1 and prodcons2 result differ because although both c files have the producer and consumer functionalities and seem to be exactly the same there is a small difference. Both prodcons1 and prodcons2 use two if-else statements, if there is no forking being done then the producer is called, else the process waits. Then if there is still no forking being done the consumer is called, else the process still needs to wait. This is the same for both commands, however when initializing the buffer Prodcons1 uses MAP_SHARED as an argument for mmap() and Prodcons2 uses MAP_PRIVATE. These values are both defined in stat.h and they both handle the privacy of the process.

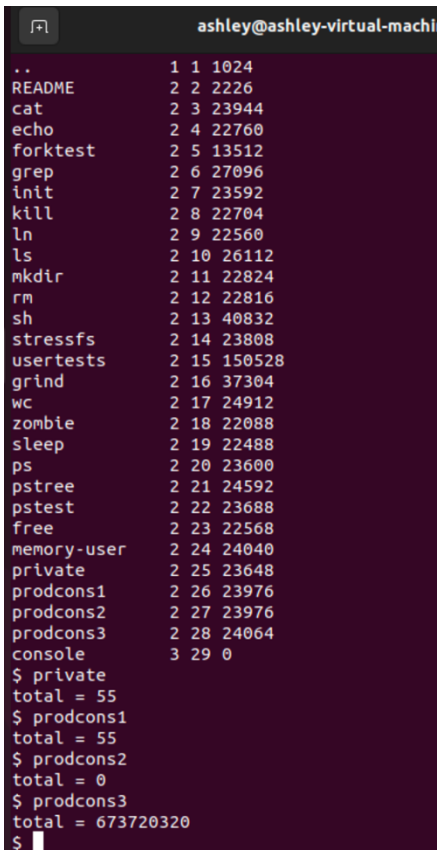
MAP_SHARED is set to 0x01 and it allows the changes between the process to be shared, or public to each other. MAP_PRIVATE is set to 0x02 and it makes sure that the changes between the processes are private. Therefore, prodcons2 ends up with a total of zero compared to prodcons1 which had 55. The MAP_PRIVATE didn't allowed the changes between the producer and consumer to be shared which made no progress in the total at the end, but prodcons1 did. The following image shows this result.



Task 3.

Unlike prodcons1 and prodcons2, prodcons3 is used to map three pages. The result for prodcons3 command is a total of 673720320. This is an incorrect result because in the producer when the buffer nextin is being assigned it is being set to one. Which even if the memory mapped is shared, the producer is never actually updating the next index for the buffer. The kernel implementation so far is only handling the situation in which the process that originally maps a shared memory is the first to write but in this case it doesn't properly work.

I believe this is fixed in the extra credit portion by modifying the usertrap() once more and making sure that it inserts new mapping for the allocated physical page into the page tables for all the processes in the "family" that have the shared memory region mapped.



Summary: This homework was more extensive than the ones assigned before, and although at first, I wasn't fully sure of how to implement the different elements at the end I understood everything in dept. I now understand why some processes have shared memory while others don't. In addition I now know that the usertrap() is a main focus of the kernel and it is really important when it comes to memory mapping!