

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 proceeded to fix 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.

```
ashley@ashley-virtual-machine: ~/Documents/xv6-riscv-labs
=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
cat        2 3 23944
echo       2 4 22760
forktest  2 5 13512
grep       2 6 27096
init       2 7 23592
kill       2 8 22704
ln         2 9 22560
ls         2 10 26112
mkdir      2 11 22824
rm         2 12 22816
sh         2 13 40832
stressfs   2 14 23808
usertests  2 15 150528
grind      2 16 37304
wc         2 17 24912
zombie     2 18 22088
sleep      2 19 22488
ps         2 20 23600
pstree     2 21 24592
pstest     2 22 23688
free       2 23 22568
memory-user 2 24 24040
private    2 25 23728
console    3 26 0
$ private
usertrap(): invalid memory address
$
```

```
513 // calculate start address
514 //HWM5-----calculate start address
515 start_addr = PGROUNDDOWN(p->cur_max - length);
516 newmmr->valid = 1;
517 newmmr->addr = start_addr;
518 newmmr->length = p->cur_max - start_addr;
519 newmmr->prot = prot;

579 }
580 // Get arguments and call munmap() helper function
581 uint64
582 sys_munmap(void)
583 {
584     uint64 addr;
585     uint64 length;
586     if(argaddr(0, &addr) < 0)
587         return -1;
588     if(argaddr(1, &length) < 0)
589         return -1;
590     return 0;
591 }

28 uint64 freemem(void);
29 //HWM5-----
30 void* mmap(void*, int, int, int, int, void*);
31 int munmap(void*, int);
32
33 // ulib.c
34 int stat(const char*, struct stat*);
```

*`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.

```

71 //Homework 5 (task 1)
72 } else if(r_scause() == 13 || r_scause() == 15){
73     //checking if the faulting address (stval register) is valid
74     if(r_stval() >= p->sz){
75         //check mapped region protection permits operation
76         for(int i=0; i<MAX_MMR; i++){
77             if(p->mmr[i].valid && p->mmr[i].addr < r_stval() && p->mmr[i].addr+p->mmr[i].length > r_stval()){
78                 if(r_scause() == 13){
79                     //read permission not set
80                     if((p->mmr[i].prot & PROT_READ) == 0){
81                         p->killed = 1;
82                         exit(-1);
83                     }
84                 }
85                 if(r_scause() == 15){
86                     //write permission not set
87                     if((p->mmr[i].prot & PROT_WRITE) == 0){
88                         p->killed = 1;
89                         exit(-1);
90                     }
91                 }
92             }
93         }
94     }
95 }
96 }
97 //hmu4--
98 }
99 //allocate physical frame memory
100 void *physical_mem = kalloc();
101 //if allocating memory was done correctly
102 if(physical_mem){
103     //maps virtual page to physical memory and inserts to pagetable
104     if(mappages(pi->pagetable, PGROUNDOWN(r_stval()), PGSIZE,
105         (uint64)physical_mem, (PTE_W | PTE_X | PTE_U)) < 0){
106         kfree(physical_mem);
107         printf("mappages didn't work\n");
108         p->killed = 1;
109         exit(-1);
110     }
111 }
112 }else{
113     printf("usertrap(): no more memory\n");
114     p->killed = 1;
115     exit(-1);
116 }
117 }
118 /*else{
119     printf("usertrap(): invalid memory address\n");
120     p->killed = 1;
121     exit(-1);
122 }*/
123 }
124 }
125 }
126 } else {
127     printf("usertrap(): unexpected scause %p pld=%d\n", r_scause(), p->pld);
128     printf("sepc=%p stval=%p\n", r_sepc(), r_stval());
129     p->killed = 1;
130 }

```

```

ashley@ashley-virtu
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
cat        2 3 23944
echo       2 4 22760
forktest  2 5 13512
grep       2 6 27096
init       2 7 23592
kill       2 8 22704
ln         2 9 22560
ls         2 10 26112
mkdir      2 11 22824
rm         2 12 22816
sh         2 13 40832
stressfs   2 14 23808
usertests  2 15 150528
grind      2 16 37304
wc         2 17 24912
zombie     2 18 22088
sleep      2 19 22488
ps         2 20 23600
pstree     2 21 24592
pstest     2 22 23688
free       2 23 22568
memory-user 2 24 24040
private    2 25 23728
console    3 26 0
$ private
total = 55

```

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
pctest      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
pctest      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/x6-riscv-lab0-AshleyMagallanes/kernel
proc.c  proc.h  Makefile  private.c  prodconst.c  prodconst2.c  prodconst3.c
157 static void
158 freeproc(struct proc *p)
159 {
160     if(p->trapframe)
161         kfree((void*)p->trapframe);
162     p->trapframe = 0;
163     //HMM5-----
164     for (int i = 0; i < MAX_MMR; i++) {
165         int dofrees = 0;
166         if (p->mmr[i].valid == 1) {
167             if (p->mmr[i].flags & MAP_PRIVATE)
168                 dofrees = 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                 dofrees = 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, dofrees);
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 from 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.

```

ashley@ashley-virtual-mach
hart 1 starting
hart 2 starting
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
cat        2 3 23944
echo       2 4 22760
forktest   2 5 13512
grep       2 6 27096
init       2 7 23592
kill       2 8 22704
ln         2 9 22560
ls         2 10 26112
mkdir     2 11 22824
rm         2 12 22816
sh         2 13 40832
stressfs   2 14 23808
usertests  2 15 150528
grind      2 16 37304
wc         2 17 24912
zombie     2 18 22088
sleep      2 19 22488
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
prodcons1  2 26 23976
prodcons2  2 27 23976
console    3 28 0
$ prodcons1
total = 55
$ prodcons2
total = 0
$

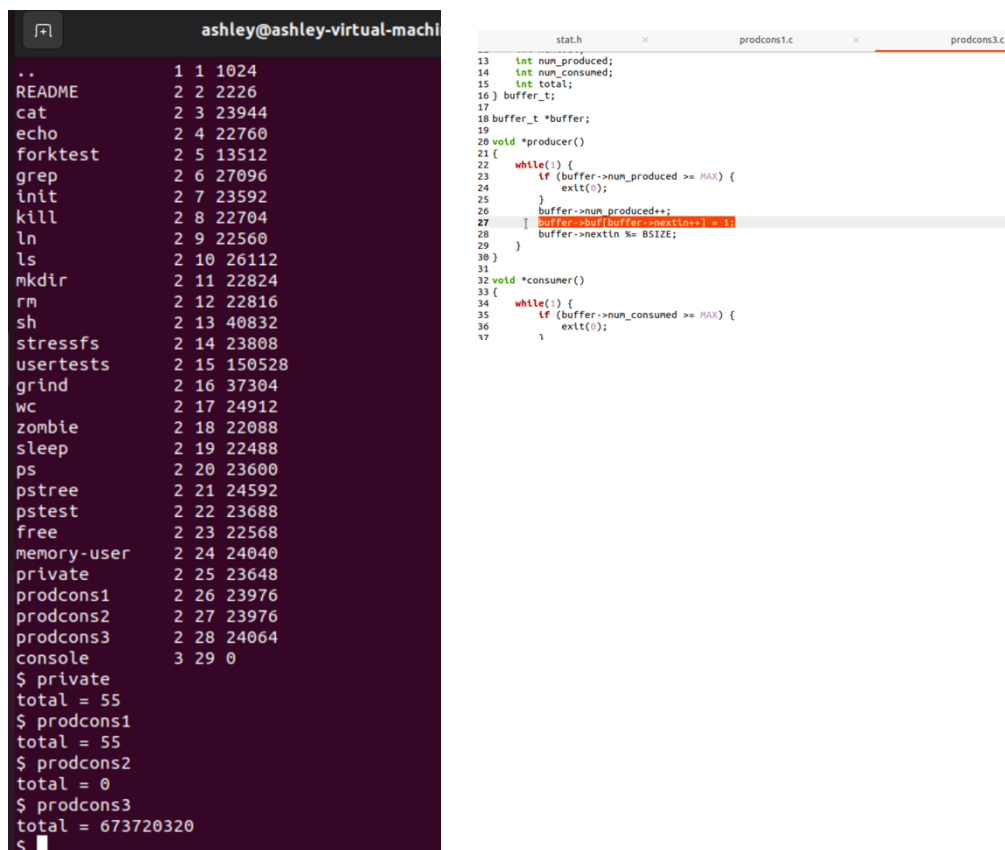
stat.h  prodcons1.c  prodcons2.c
30 }
31
32 void *consumer()
33 {
34     while(1) {
35         if (buffer->nun_consumed >= MAX) {
36             exit(0);
37         }
38         buffer->total += buffer->buf[buffer->nextout++];
39         buffer->nextout %= BSIZE;
40         buffer->nun_consumed++;
41     }
42 }
43
44 int main(int argc, char *argv[])
45 {
46     buffer = (buffer_t *) mmap(NULL, sizeof(buffer_t),
47                               PROT_READ | PROT_WRITE,
48                               MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
49     buffer->nextin = 0;
50     buffer->nextout = 0;
51     buffer->nun_produced = 0;
52     buffer->nun_consumed = 0;
53
54     if (!fork())
55         producer();
56 }
57
58 stat.h  prodcons1.c  prodcons2.c
32 void *consumer()
33 {
34     while(1) {
35         if (buffer->nun_consumed >= MAX) {
36             exit(0);
37         }
38         buffer->total += buffer->buf[buffer->nextout++];
39         buffer->nextout %= BSIZE;
40         buffer->nun_consumed++;
41     }
42 }
43
44 int main(int argc, char *argv[])
45 {
46     buffer = (buffer_t *) mmap(NULL, sizeof(buffer_t),
47                               PROT_READ | PROT_WRITE,
48                               MAP_ANONYMOUS | MAP_SHARED, -1, 0);
49     buffer->nextin = 0;
50     buffer->nextout = 0;
51     buffer->nun_produced = 0;
52     buffer->nun_consumed = 0;
53     buffer->total = 0;
54     if (!fork())
55         producer();
56 }

```

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.



The image shows a terminal window on the left and a code editor on the right. The terminal window, titled 'ashley@ashley-virtual-mach...', displays the output of the 'ps' command, listing various processes and their memory usage. The processes listed include 'private', 'total = 55', 'prodcons1', 'total = 55', 'prodcons2', 'total = 0', 'prodcons3', and 'total = 673720320'. The code editor on the right shows the source code for 'prodcons3.c'. The code defines a producer and a consumer function. The producer function increments a counter and updates the buffer. The consumer function decrements the counter and reads from the buffer. The code is written in C and uses standard library functions like 'printf' and 'exit'.

```
.. 1 1 1024
README 2 2 2226
cat 2 3 23944
echo 2 4 22760
forktest 2 5 13512
grep 2 6 27096
init 2 7 23592
kill 2 8 22704
ln 2 9 22560
ls 2 10 26112
mkdir 2 11 22824
rm 2 12 22816
sh 2 13 40832
stressfs 2 14 23808
usertests 2 15 150528
grind 2 16 37304
wc 2 17 24912
zombie 2 18 22088
sleep 2 19 22488
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
prodcons1 2 26 23976
prodcons2 2 27 23976
prodcons3 2 28 24064
console 3 29 0
$ private
total = 55
$ prodcons1
total = 55
$ prodcons2
total = 0
$ prodcons3
total = 673720320
$
```

```
13 int num_produced;
14 int num_consumed;
15 int total;
16 buffer_t;
17
18 buffer_t *buffer;
19
20 void *producer()
21 {
22     while(1) {
23         if (buffer->num_produced >= MAX) {
24             exit(0);
25         }
26         buffer->num_produced++;
27         buffer->buf[buffer->nextin++] = 1;
28         buffer->nextin %= BSIZE;
29     }
30 }
31
32 void *consumer()
33 {
34     while(1) {
35         if (buffer->num_consumed >= MAX) {
36             exit(0);
37         }
38     }
39 }
```

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!