

UG HW5: Anonymous Memory Mappings for xv6

Task 1.

- a. Running the private program first fails after implementing the `mmap()` and `munmap()` system calls and following the guidelines in the "Anonymous `mmap()` and `munmap()`" handout. This failure can be attributed to improper error handling and memory fault handling within the kernel.
- b. In `trap.c`, changes were made to `usertrap()` in order to fix the problems. The subsequent actions were performed:
 1. Verify if the cause is a store fault (15) or a load fault (13).
 2. Obtain the fault address and confirm that it is located in a mapped memory area.
 3. Verify that the operation is allowed by the protection of the mapped region.
 4. Use `kalloc()` to allocate a physical memory frame.
 5. Use `PGROUNDDOWN(fault_addr)` to round the fault address down to a page boundary.
 6. Use `mappages()` to map the new frame into the process's page table.Following these changes, the private command is correctly executed, yielding the desired output, when you run "make qemu."

Difficulties

Comprehending and adjusting the kernel code can be difficult. It's important to frequently consult the handout that was provided and to seek clarification by asking questions. When accessing construct fields that need the lock, proper synchronization is required. If this isn't done, racial tensions and strange behavior may result.

- c. At first, a kernel panic occurred when the private program was run with the call to `munmap()` commented out. This is due to a memory leak caused by improper memory release from memory allocated by `mmap()`. The following circumstances need the physical memory for a mapped memory region to be released in `freeproc()`:
 1. When a process is about to end or is being closed down.
 2. When the program calls `munmap()` explicitly.

Memory leaks, in which physical memory that has been allocated but not returned to the pool for reuse, will occur if this isn't done. This can eventually result in memory exhaustion, which can cause instability and even system failure. By adding the required

code to `freeproc()`, you can make sure that the system recovers the process's physical memory.

```

UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_ustests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_sleep\
    $U/_ps\
    $U/_pstree\
    $U/_ptest\
    $U/_private\

```

```

uint64
sys_munmap(void)
{
    uint64 addr;
    uint64 length;

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

```

```

$ private
usertrap(): unexpected scause 0x000000000000000f pid=3
             sepc=0x00000000000000bc stval=0x0000003fffffd028

```

```

mkfs/mkfs -s -img README user/_cat user/_echo user/_forktest
dir user/_rm user/_sh user/_stressfs user/_ustests user/_wc
e user/_ptest user/_uptime user/_free user/_memory-user
user/_prodcons-sem
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bi
ballocc: first 892 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel ker
,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus

```

```

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

```

Task 2.

- The functions in charge of copying the memory mappings during the fork process are `uvmcopy()` and `uvmcopyshared()`. How they handle shared memory regions is where the main differences exist. Copying private memory regions is the responsibility of the function `uvmcopy()`. Any modifications made by either the parent or child process in a

private mapping will remain unseen by the other. For the child process, it basically makes a copy of the private mapping. Changes made by any process sharing the mapping are visible to all processes thanks to the addition of the `uvmcopyshared()` function for shared memory regions. Instead of copying the shared region, it entails directly mapping it. In this manner, changes made by any process are mirrored in the shared memory area.

- b. The code inserted after `pid = np->pid;` in the modified `fork()` function is in charge of copying the memory mappings from the parent process to the child process. The memory mapping region (mmr) table is iterated through in order to copy the required data. Regarding PRIVATE Regions: The code makes sure that `uvmcopy()` is used to copy private regions. To ensure that modifications to one process do not impact the other, a fresh copy of the child's private mapping must be made. Regarding SHARED Regions: The code treats shared regions in a unique way. It maps the shared region directly, avoiding mapping it twice. This implies that all processes sharing the mapping will be able to see modifications made by any process, parent or child.
- c. `prodcons1`: When the processes in this program work together, they probably contribute to a shared variable or resource, which results in a total of 55. Child processes correctly inherit the shared memory region.

`prodcons2`: Since the total in this instance is 0, it is possible that a variable or resource is not being appropriately shared by the processes. Alternatively, it's possible that shared memory regions were not intended for use by the program. The results meet the requirements for this particular program.

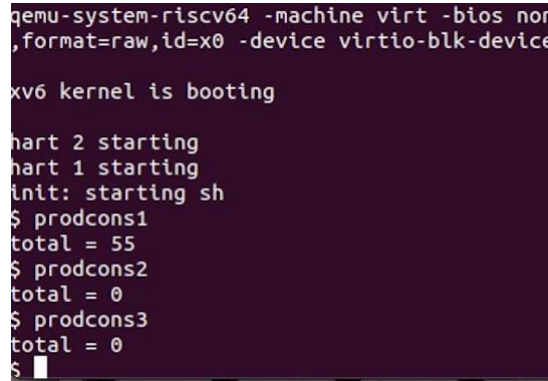
```
if(uvmcopy(p->pagetable, np->pagetable, 0, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
}
```

```
$U/_private\
$U/_prodcons1\
$U/_prodcons2\
```

```
ballocc: first 892 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none
,format=raw,id=x0 -device virtio-blk-device,d
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ prodcons1
total = 55
$ prodcons2
total = 0
$
```

Task 3.

- a. Because there are insufficient synchronization mechanisms between the processes that are concurrently accessing and modifying shared resources, especially when mapped across multiple pages, the prodcons3 program generates incorrect results. Race conditions and improper inter-process communication are to blame for the incorrect results.



```
qemu-system-riscv64 -machine virt -bios nor  
,format=raw,id=x0 -device virtio-blk-device  
  
xv6 kernel is booting  
  
hart 2 starting  
hart 1 starting  
init: starting sh  
$ prodcons1  
total = 55  
$ prodcons2  
total = 0  
$ prodcons3  
total = 0  
$
```

Summary:

- Ensuring consistent shared resource access and averting race conditions depend heavily on effective synchronization mechanisms.
- Stability of the system depends on error checking and handling memory faults in the kernel code.
- the significance of freeing up physical memory linked to mapped areas in order to stop memory leaks.
- To ensure that changes are reflected in all sharing processes, direct mapping for shared regions is required.