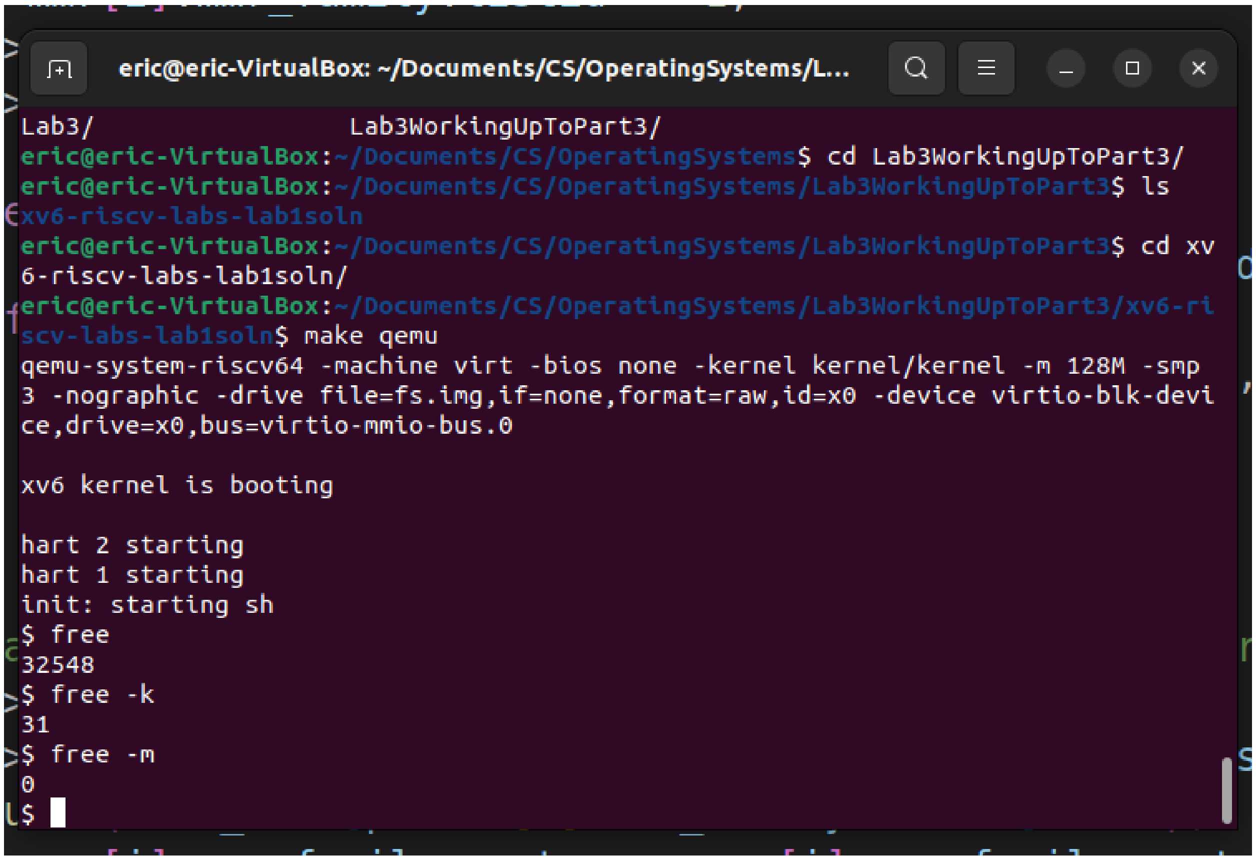
CS4375-17800 Fall 2022 Lab Report
Eric Smith Submitted on 11/7/2022
essmith1@miners.utep.edu

# Lab 3: Anonymous Memory Mappings for xv6

Task 1. free command

Show the result of running your free command with all the possible options and explain the results. Briefly describe your approach to implementing the free command.

For task1, we were assigned with inserting the free command into xv6. The free command is a user command which displays the amount of free memory within a system. This command is helpful for when you want to evaluate how much memory a certain process is using and troubleshooting for conditions such as memory leaks. There are a few options which could be used as flags, mainly the -k and -m flags which will reduce the outputted number to either display the amount of kilobytes or megabytes. By default, the program will output the total amount of bits which are free.



```
Lab3/                     Lab3WorkingUpToPart3/
eric@eric-VirtualBox:~/Documents/CS/OperatingSystems$ cd Lab3WorkingUpToPart3/
eric@eric-VirtualBox:~/Documents/CS/OperatingSystems/Lab3WorkingUpToPart3$ ls
xv6-riscv-labs-lab1soln
eric@eric-VirtualBox:~/Documents/CS/OperatingSystems/Lab3WorkingUpToPart3$ cd xv
6-riscv-labs-lab1soln/
eric@eric-VirtualBox:~/Documents/CS/OperatingSystems/Lab3WorkingUpToPart3/xv6-ri
scv-labs-lab1soln$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-devi
ce,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ free
32548
$ free -k
31
$ free -m
0
$
```

Task 2.

For task 2, we were required to implement mmap() and munmap() functions with lazy allocation. We were provided with much of the code and needed to make sure to add all the system call requirements for xv6. As part of the code we needed to add, there was a variety of MMR structures which we added to proc.h. These MMR structures defined the layout of a memory region within a process' structure. The basic MMR structure contained necessary information such as address, length, flags, protection, valid bit, and a file pointer which was not used for this lab.

The mmap() implementation starts out with some basic argaddr and argint commands to pass arguments into the mmap system call. The function returns -1 if one of these arguments was not passed correctly. Next, the function will start traversing through the MMR list. Once a free location is found within the MMR list, we start allocating fields of a new MMR structure. We then call mapvpages if a new page table is needed.

The munmap() function works similarly to mmap(). We traverse the list trying to find a address given as a parameter. Once we find this address, we check that no other processes have been mapped to the memory region and then free it if this condition is true.

a. Show the results of running the private program. Explain how the private program uses mmap() and munmap().

The mmap() implementation starts out with some basic argaddr and argint commands to pass arguments into the mmap system call. The function returns -1 if one of these arguments was not passed correctly. Next, the function will start traversing through the MMR list. Once a free location is found within the MMR list, we start allocating fields of a new MMR structure. We then call mapvpages if a new page table is needed.

The munmap() function works similarly to mmap(). We traverse the list trying to find a address given as a parameter. Once we find this address, we check that no other processes have been mapped to the memory region and then free it if this condition is true.

b. Explain why the program trapped into the kernel before you modified usertrap() in kernel/trap.c.

Private would cause a trap because it was trying to map to pages which haven't been allocated yet. We solve this by checking if the fault address falls within a valid memory region of the page, and then allocate those pages if possible. If it is not possible to allocate within the specified memory location, then we just let the trap propagate.

c. Explain why running the program initially caused a kernel panic before you added the code for part c to freeproc().

The program would initially cause a kernel panic because the program would fail to unmap. This would leave unmapping up to the job of the operating system which is something which was not implemented yet. Adding the code to freeproc meant that we are freeing the memory and pages once a process finishes. This allows programs to not have to specifically deallocate with munmap.

d. 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.

We only free the physical memory if it has no protection flags or it has no other family members. This means that vital processes are never freed or processes with shared memory do not get freed. If we do not do these checks, then vital processes may fail, or a process may try to grab a page no longer allocated which will either be a security risk or cause a fault.

e. Describe any difficulties you ran into with this task and if/how you overcame them.

My main difficulty with this task was understanding how the page tables are structured within the given implementation. Once I got a walk through of the general structure and how to work within the page tables, the implementation got a lot easier.

Task 3.
Show the results of running the prodcons1 and prodcons2 program. Explain the results, including why they produce different results.

Task 4.

a. Explain why the prodcons3 program produced incorrect results before you implemented part b.
b. Show the result of running prodcons3 after you have completed Task 4b. Describe any difficulties you ran into with this task and if/how you overcame them.

**Summary:**
Describe what you learned from doing this lab.