

CS4375-13948 Fall 2023
Homework Report
Alberto Delgado
Submitted on 11/10/2023
adelgado@miners.utep.edu

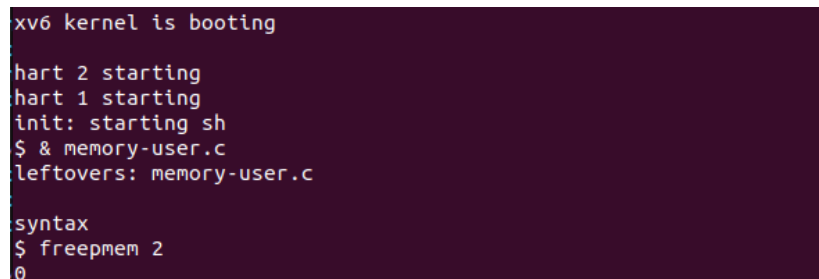
HW 4: Lazy Allocation for xv6

Please replace red text with your report text and any tables or figures, names of any accompanying files, etc. Remember to commit all the files for your lab submission, to put the URL for your private xv6 repo in the Teams assignment, to submit the Teams assignment, and to give the instructor and TA access to your repo.

Task 1. freemem() system call

In order to add freemem() as a system call to the xv6 I had to edit various files to get the freemem() function to perform correctly. The first thing I did was rename the free.c program given to us. The reason for this was because I kept getting an error that there was no rule to manage the target "user/_freemem". This was needed for the fs.img file and although I had declared freemem to be a user target, without the proper file, the program would not execute correctly. So I renamed the free.c file to freemem.c., this would fix my issue with the target rule. Secondly, I had to add a new function to kalloc.c, the change that was a necessary was to add a kfreemem() function. The function was created at the bottom of the kalloc.c file.

Another error that I encountered was an error of an implicit declaration of the kfreemem() function. The way that I fixed this was by defining the function in the kalloc.c portion of the defs.h file, this fixed the error that I was receiving. Apart from this, I entered the freemem() into all necessary system call files to get the freemem() function fully functional. Below is a screenshot of the output when using the freemem function.



```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ & memory-user.c
leftovers: memory-user.c

syntax
$ freemem 2
0
```

So, as requested in the instructions I ran the memory-user.c file in the background and ran the freemem function to see how long it would take until there was an error and the error happened after the second request.

Below is a screenshot of the output from the terminal.

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ & memory-user.c
leftovers: memory-user.c

syntax
$ freepmem 2
0
$ freepmem 10
usertrap(): unexpected scause 0x0000000000000005 pid=5
          sepc=0x0000000000000090 stval=0x0000000000003f98
panic: kfree
█
```

Task 2. Change sbrk() so that it does not allocate physical memory.

The edit the sbrk() system call I first went to the location of the system call. The location was found in the sysproc.c file in the kernel folder of xv6. For the most part I kept everything the same the changes that I made were to the if statement. To test out the function I commented out the if statement just for future reference. I then added a line, myproc()→sz +=n, The myproc() is a pointer to struct proc, sz is a member of the struct and n determines to size in bytes of the the address space of a process. So this is what I did to stop the allocation of physical memory. Once I ran the make qemu and used a wait() user command I got the following output:

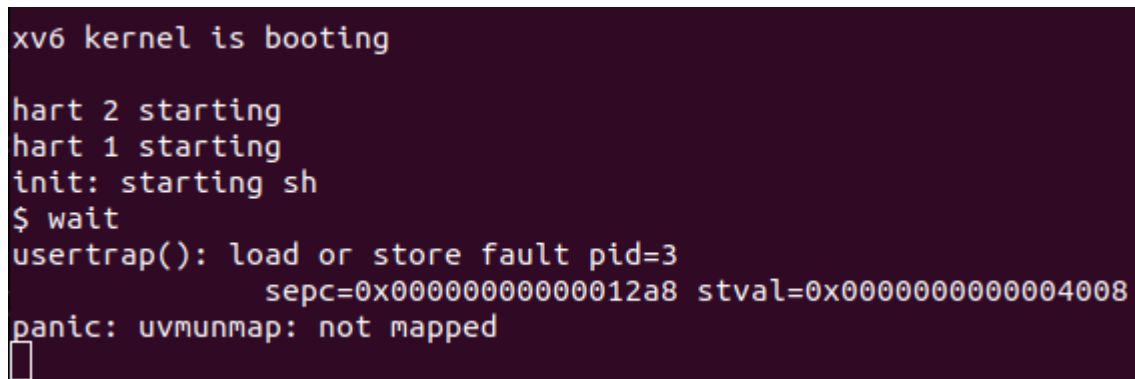
```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ wait
usertrap(): unexpected scause 0x000000000000000f pid=3
          sepc=0x00000000000012a8 stval=0x0000000000004008
panic: uvmunmap: not mapped
█
```

The output is showing that there is an issue when coming to mapping the memory space for the process. From my understanding, This is happening since there is no physical memory to be mapped for the process which is causing the issue in the wait() process. The process can not be mapped and cannot gain the appropriate physical memory to function properly.

Task 3. Handle the load and store faults that result from Task 2

To handle the load and store faults that resulted from the previous task I first had to access the `usertrap()` function from the `trap.c` file. Once I found the appropriate function, I went the conditional statements and this is where I would handle the load and store faults. In the default function there is if else statements, to follow the progression I inserted another else if statement. The else if statement that I would insert would handle both the load and store faults. The conditions that I used were `r_scause() == 13 || r_scause() == 15`. The reason I used these are because the `r_scause() == 13` statement in RISC-V checks if the trap cause s a load fault. On the other hand `r_scause() == 15` does the same thing but for the store fault in RISC-V. What I learned in this task is how trap causes and handling worked in RISC-V. Being able to handle faults are something that can be beneficial for any program. Below is a screenshot of the output after this task.



```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ wait
usertrap(): load or store fault pid=3
          sepc=0x000000000000012a8 stval=0x00000000000004008
panic: uvmunmap: not mapped
█
```

The most difficult part for me was the conditional statement and trying to determine the values and what conditional statement to use in the else if statement. The way I was able to overcome this was by referring to the textbook and seeing how RISC-V handles faults. So research and testing various conditions are how I overcame this.

Task 4. Fix kernel panic and any other errors.

From my understanding, the reason we are getting a kernel panic and `usertrap()` error issues is due to an issue with memory allocation. To fix the errors and address these issues I went into the `sysproc.c` file to allow for the memory allocation in the `sbrk()` function. Since the error was an issue with the memory mapping, I had to implement the `growproc()` to allow the memory to allocate correct. Once I reinstated the `growproc()` then the errors were no longer displaying and the program functioned properly. Below is a screenshot of the terminal with a functional output.

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ wait
exec wait failed
$ 

```

Task 5. Test your lazy memory allocation.

The first test that I had worked on was the testing case that would not touch the memory allocation. I first created a char pointer that would allocated memory using the malloc() function. After this I used an if else statement that would return an error if the pointer was empty. After this if the if statement was not met then the memory would be freed using the free() function call.

The second test was working with the memory allocation that was going to be touched. So the code was the same as the first test case except the difference was that I entered various characters at various points of the memory allocation to have the touched portion of the memory since I was manipulating the memory.

Lastly, for the third test case I would keep the code very similar to the last two test cases but for this one, I used the page size and number of the pages to get an allocated memory space. I did 4000 byte size for this one since a page size is around this size in a 32 bit system. To get the actual size I multiplied the page size and number of pages and entered that value into the malloc() function. After this I would turn the touching portion of the second test to a for loop that would iterate through the pages and place a character in that portion. I would then free the allocated memory using the free() function. The reason that I used the free() function over the freepmem() function that I created was because my function have a void type in the arguments and could not take a value as an argument. Whenever I would use the freepmem() function the code would return an error stating that the value in the argument was not compatible with void.

```

(base) aodelgado@aodelgadolinu:~/Desktop/xv6-riscv-labs-riscv$ make qemu-test1
make: *** No rule to make target 'qemu-test1'.  Stop.
(base) aodelgado@aodelgadolinu:~/Desktop/xv6-riscv-labs-riscv$ make qemu test1
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-device,drive=x0,bus=virtio-mmio-bus.0
qemu-system-riscv64: -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0: Failed to get "write" lock
Is another process using the image [fs.img]?
make: *** [Makefile:165: qemu] Error 1
(base) aodelgado@aodelgadolinu:~/Desktop/xv6-riscv-labs-riscv$ make qemu test2
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-device,drive=x0,bus=virtio-mmio-bus.0
qemu-system-riscv64: -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0: Failed to get "write" lock
Is another process using the image [fs.img]?
make: *** [Makefile:165: qemu] Error 1
(base) aodelgado@aodelgadolinu:~/Desktop/xv6-riscv-labs-riscv$ make qemu test3
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-device,drive=x0,bus=virtio-mmio-bus.0
qemu-system-riscv64: -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0: Failed to get "write" lock
Is another process using the image [fs.img]?
make: *** [Makefile:165: qemu] Error 1
(base) aodelgado@aodelgadolinu:~/Desktop/xv6-riscv-labs-riscv$ 

```

When I would try to test all of the test files, I would get a make error. I would use the command make with all the test files and would get errors but was unable to determine the issues. I was unsure if the error was caused by the test files or the make command.

Extra Credit Task 6. Enable use of the entire virtual address space.

Although I was unable to finish both extra credit portions I was able to get a portion for both that I believe are how to start the task. For task 6 to enable use of the entire address space, I believe that we would modify the kernel space portion in the memlayout.h file in the kernel folder. Lines 47, 48 and 52 would need to be modified since the memory in the kernel is what we would need to interact with to get an entire virtual address space and we would need to change the limit of the address space. Another file that would need to be modified is the vm.c file which is the file in the kernel folder that deals with the virtual address space. Now I am unsure which portions to modify specifically but I do believe that vm.c is another that will need to be modified to achieve this task.

Extra Credit Task 7. Allow a process to turn memory overcommitment on and off.

Now task 7 was a bit tricky for me at first, but then I realized that for memory overcommitment I believe that we would need to follow similar steps as task 6. Since memory overcommitment is virtual memory giving more memory than physically available, this creates the illusion to the processes that they have more memory than what is available. To accomplish this we would need to change the physical address size as in task 6. The reason is because we would need to increase the size for the virtual address space. We would then also make modifications to the vm.c file which would be increasing the virtual address space for the processes. Again this is what I believe would need to be done in order to achieve this task or at minimum begin this task.

REFERENCES

Remzi H. Arpaci-Dusseau & Andrea Arpaci-Dusseau -
Operating Systems: Three Easy Pieces
<https://pages.cs.wisc.edu/~remzi/OSTEP/>

Brian Kernighan and Dennis Ritchie -
The C Programming Language, 2nd edition

Russ Cox, Frans Kaashoek, and Robert Morris -
xv6: a simple, Unix-like teaching operating system

Geeksforgeeks.org -
<https://www.geeksforgeeks.org/memory-management-in-operating-system/>
<https://www.geeksforgeeks.org/memory-mapping/>