Christian Revilla

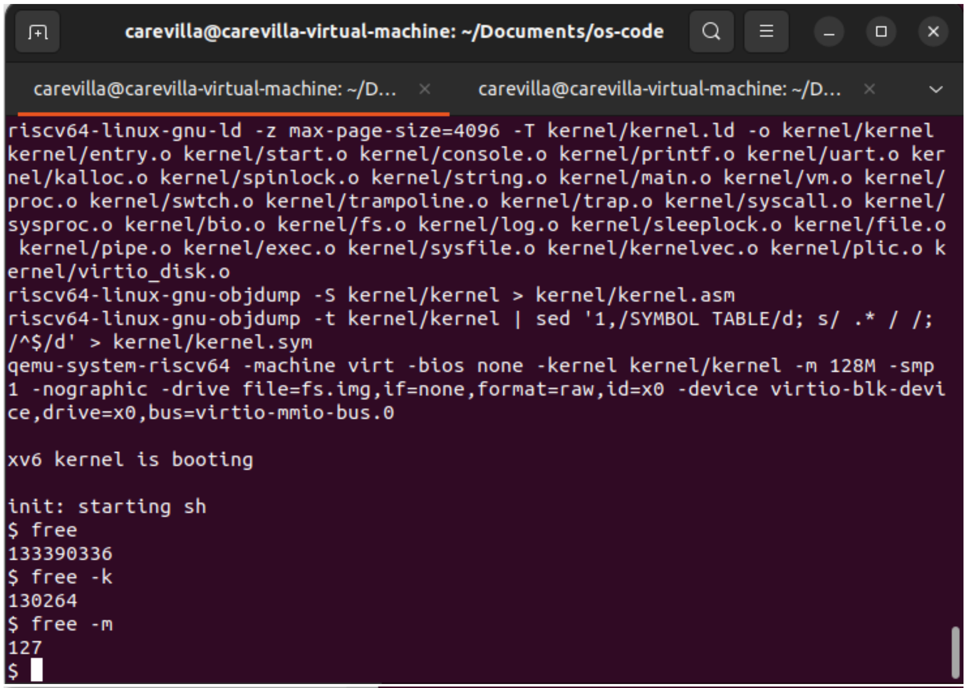
Dr. Moore

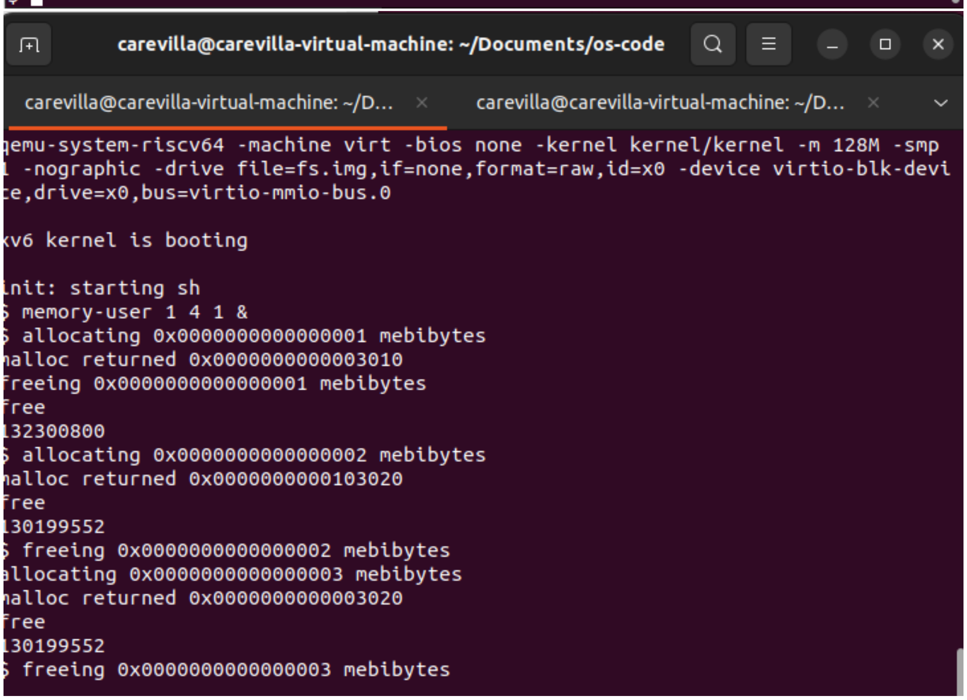
Homework #4

Lazy Allocation for xv86

Task 1: Freepmem() system call

In task 1 the objective was to successfully implement the freepmem system call to our xv86 operating systems. To begin solving task 1 I first created a new GitHub branch and named it Homework4 and pulled my most recent homework 3 code to it. Next, I downloaded the test files that were provided to us and added them to the Makefile so we can run them as test cases. With the code now in my system the first step was heading to user.h and adding the freepmem system call which will return an int and take in a void as a parameter. From here added an entry in usys.pl for the new system call. Once that was complete it was time to head to kernel code and first open up syscall.h and add a system call number for freepmem. Then open up syscall.c and define the freepmem call and add it to the system call array. In sysproc.c I defined a new function for the system call which will return the result of a helper function, to be discussed, multiplied by the PGSIZE back to the user. This will represent the total number of bytes available to be freed. The last thing I did was in kalloc.c I defined a helper function that will use the struct run to access the link list and keep traversing the link list keeping a count until a null is found. When the null is found simply release the lock and return the count. Below our sample runs of task 1 which match the expected outputs





Task 2: Change sbrk()

For the task 2 portion of homework 4 we were asked to modify the sbrk() system call so that it will now allocate virtual memory spaces. This task was quite simple and not much coding was needed. My approach was to head to *sysproc.c* and see the implementation of the current sbrk() system call. As the handout states, the current implementation calls growproc to allocate physical memory. I first began commenting the original code that calls the growproc function. I then create a new int variable addr and give it the original value of myproc()->sz which represents the current size on the heap. I have a simple if statement checking that if I were to increment the original value by the requested value that sum does not go over the max TRAPFRAME available. If the sum is within bounds simply assign the new size to myproc and return the original size. The scause error and code implementation are shown below.

```c
uint64
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;

  addr = myproc()->sz; // current size

  if((addr+n) > TRAPFRAME)
    return -1;

  myproc()->sz = addr+n;

  //if(growproc(n) < 0)
  //return -1;
  return addr;
}

uint64
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
```

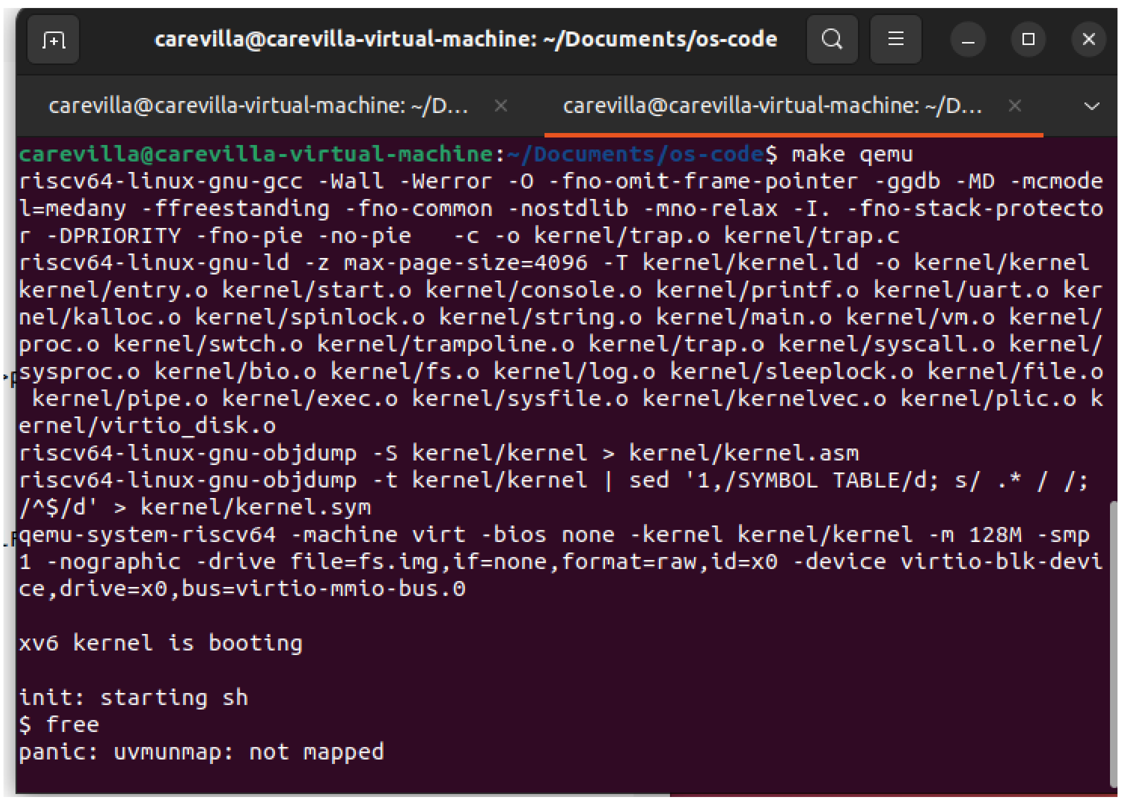
```
carevilla@carevilla-virtual-machine: ~/Documents/os-code
carevilla@carevilla-virtual-machine: ~/D...    carevilla@carevilla-virtual-machine: ~/D...
l=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protecto
r -DPRIORITY -fno-pie -no-pie   -c -o kernel/sysproc.o kernel/sysproc.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel
kernel/entry.o kernel/start.o kernel/console.o kernel/printf.o kernel/uart.o ker
nel/kalloc.o kernel/spinlock.o kernel/string.o kernel/main.o kernel/vm.o kernel/
proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/
sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o
 kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o k
ernel/virtio_disk.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > kernel/kernel.sym
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
1 -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

init: starting sh
$ free
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x00000000000012d0 stval=0x0000000000004008
panic: uvmunmap: not mapped
```

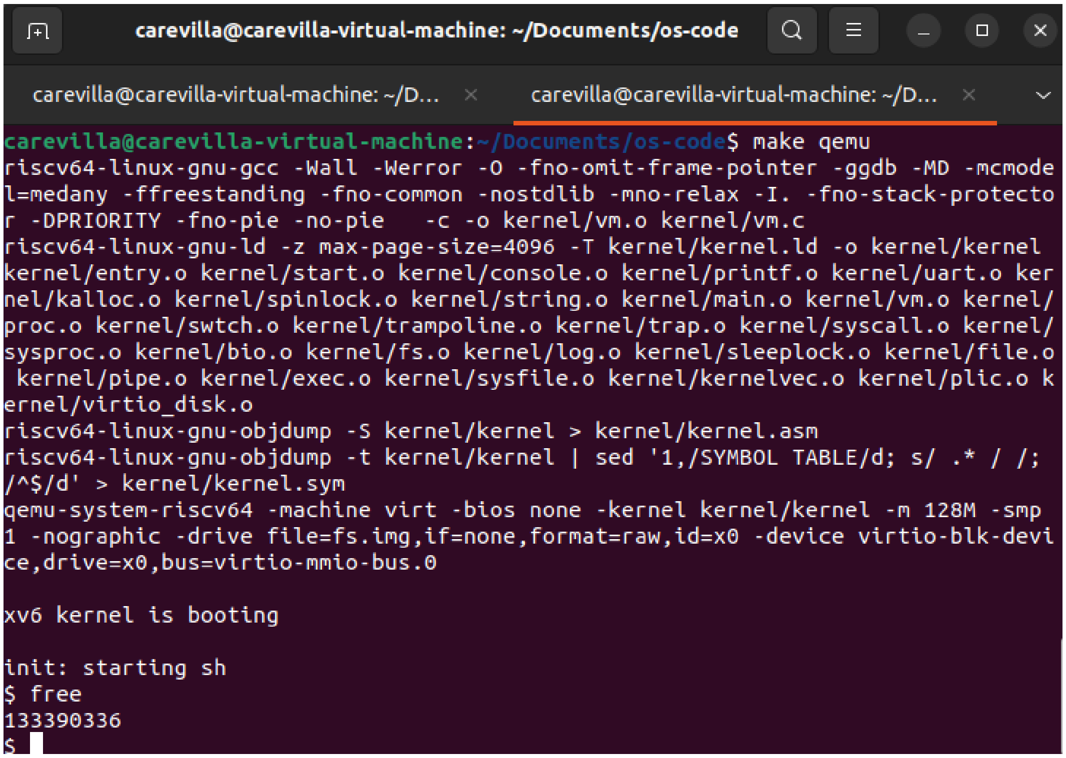
Task 3: Handle Load and Store Faults

For Task 3, we were asked to fix the faults that were caused by us modifying the sys_sbrk function from task 2. My first step in solving this problem was first learning how to reference the specific store and load faults. Prior to solving, when I would run the free command the error message indicated to hex values one being 15 and 13. With that knowledge I was able to properly assess the faults. In *kernel/trap.c* I modified the *usertrap()* function by adding another else if case checking for both hex values. If the faults were found, we check if the fault is within the virtual memory. If not as the handout states, we call *kalloc()* to begin allocating memory. After *kalloc()* we ensure the memory was properly allocated if not prompt an error. Else we call memset to clear the page and with mappages map the virtual page with the newly allocated memory. Below is a screenshot of the free command running, as we can see we no longer have the sfault.



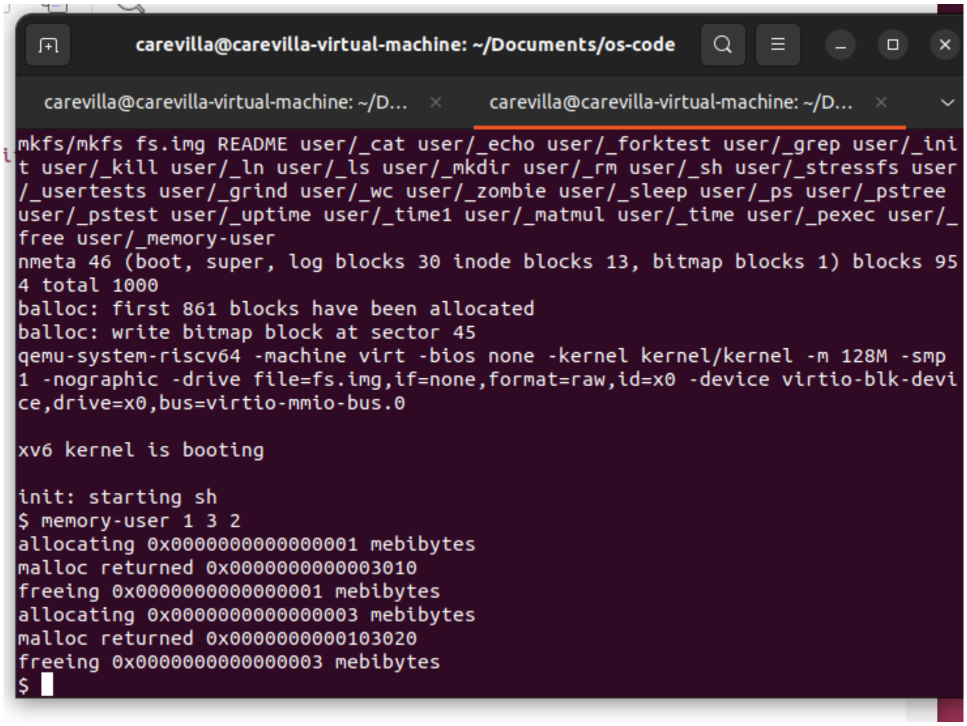
Task 4: Fix kernel panic and other errors

For task 4 part of the homework, we were asked to fix any ongoing problems or errors still left in the code. After my implementation in task 3 I was left with a panic usmunmap error that needed to be addressed. I traversed to the file *vm.s* and went to the functions uvmcopy and uvmunmap and modified an if statement there. Here is the part where the code was prompting the panic statement for the previous errors. To fix this I simply added a continue statement when checking if the pointer to pte exist and if the PTE_V (page table is valid) if that is zero then usually this means our page table is not present, so for now let's just continue thru this check. After all my efforts I no longer have any errors when running the free command as shown below.



Task 5: Test Your Lazy Memory Allocation

Finally, for the final task of homework 4 all we had to do was test our newly implemented lazy memory allocation. To achieve this, I headed back to user code and opened the test file that was given to us, memory-user.c. In this test file I un-commented all the previously commented code to be able to test my implementation. Below is a screenshot of a running program with no more errors.


carevilla@carevilla-virtual-machine: ~/Documents/os-code
carevilla@carevilla-virtual-machine: ~/D...
carevilla@carevilla-virtual-machine: ~/D...
mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_ini
t user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user
/_usertests user/_grind user/_wc user/_zombie user/_sleep user/_ps user/_pstree
user/_pstest user/_uptime user/_time1 user/_matmul user/_time user/_pexec user/_
free user/_memory-user
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 95
4 total 1000
balloc: first 861 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
1 -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
init: starting sh
$ memory-user 1 3 2
allocating 0x0000000000000001 mebibytes
malloc returned 0x0000000000003010
freeing 0x0000000000000001 mebibytes
allocating 0x0000000000000003 mebibytes
malloc returned 0x0000000000103020
freeing 0x0000000000000003 mebibytes
$