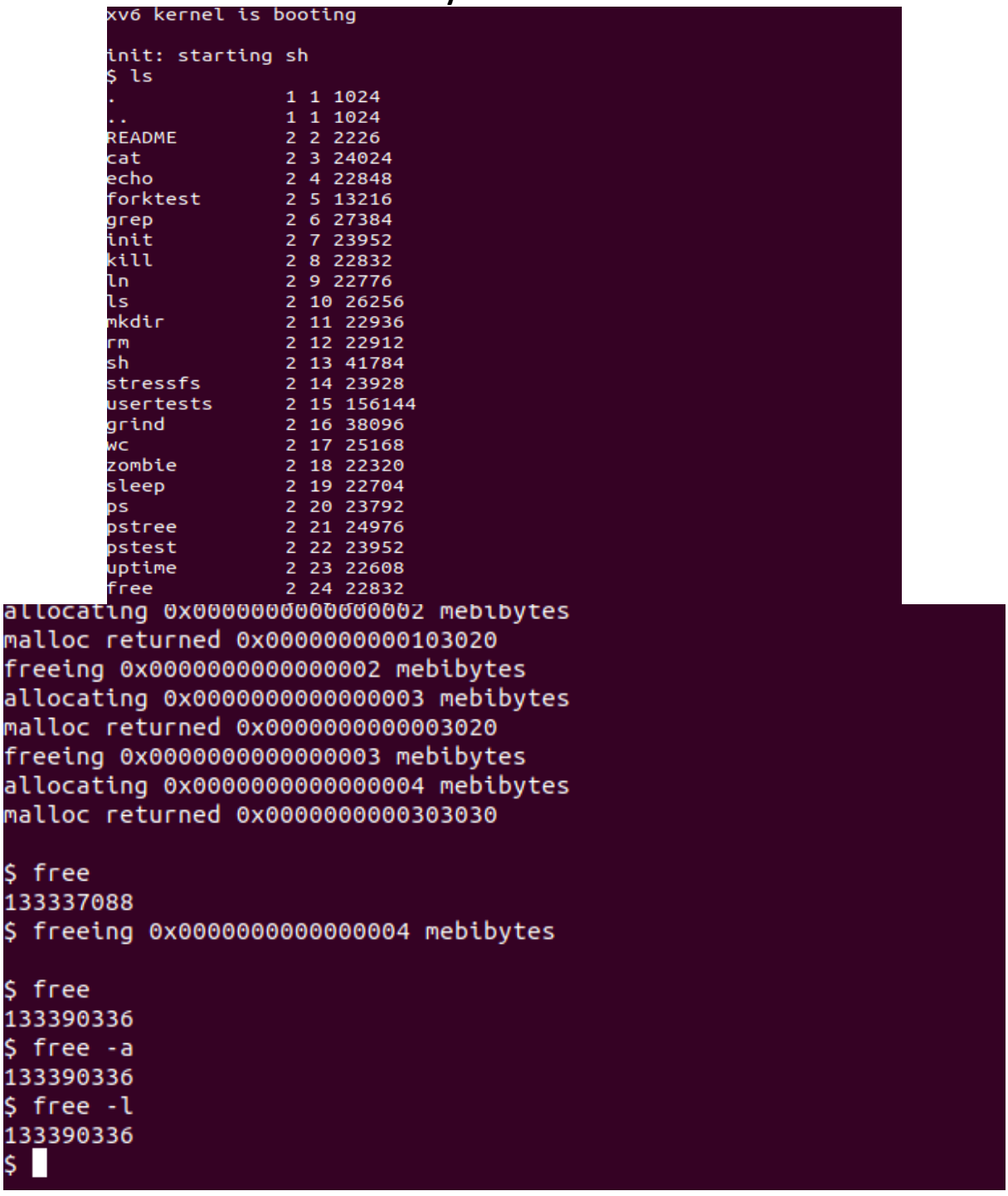
CS4375-13948 Fall 2023
Javier Venegas
Javenegas8@miners.utep.edu
https://github.com/javenegas8/OperatingSMoore

Homework 4 Report
Submitted on 11/10/2023

HW 4: Lazy Allocation for xv6



```
memory-user    2 25 24096
console        3 26 0
$ hart1 starting
exec hart1 failed
$ free -k
130264
$ free
133390336
$ free -m
127
$ memory-user 1 4 1 &
$ allocating 0x0000000000000001 mebibytes
malloc returned 0x0000000000003010
freeing 0x0000000000000001 mebibytes
allocating 0x0000000000000002 mebibytes
malloc returned 0x0000000000103020
freeing 0x0000000000000002 mebibytes
allocating 0x0000000000000003 mebibytes
malloc returned 0x0000000000003020
freeing 0x0000000000000003 mebibytes
allocating 0x0000000000000004 mebibytes
malloc returned 0x0000000000303030

$ free
133337088
$ freeing 0x0000000000000004 mebibytes
```

Task 1. freepmem() system call

Observing how much memory you can allocate using the memory-user program before malloc() fails and by understanding the relationship between calling freepmem() and the availability of free memory.

By carrying out this task, we gain insights into how the xv6 operating system handles memory allocation and deallocation, especially with the introduction of the freepmem() system call. Understanding the limitations of memory allocation and the impact on system performance is crucial for developing robust and efficient operating systems.

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

In summary, the errors occurring in this task likely stem from the changes made to the sbrk() system call, where physical memory allocation is bypassed. The absence of a usertrap() and potential issues with dynamic memory allocation could indicate that the system is not handling virtual memory operations and exceptions as expected. By carrying out this task, I learn more about the details of memory management, dynamic memory allocation, and the role of the sbrk() system call in xv6. Understanding these concepts is crucial for developing a robust and efficient operating system.

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

If the handling of load and store faults does not include the allocation of physical memory when needed, the system might not address the root cause of the faults, leading to persistent issues.

Also, without proper logging or debugging statements, it might be challenging to diagnose the flow of control and identify any issues during load and store fault handling.

This task provides a deeper understanding of how exceptions and traps are handled in the xv6 operating system, especially when dealing with virtual memory access violations.
The task highlights the intricacies of the interaction between the modified sbrk() implementation (Task 2) and the handling of load and store faults. Understanding this interaction is crucial for maintaining the integrity of virtual memory.

Some of the difficulties in this task were identifying the correct virtual memory range for the process and understanding how the faulting address relates to the heap, stack, and other segments can be challenging. Referencing the relevant documentation and existing code snippets helps overcome this difficulty.

Task 4. Fix kernel panic and any other errors.

Changes in

In conclusion, Task 4 provided a valuable learning experience by delving into the complexities of virtual memory management and debugging in the context of operating system development. The difficulties faced were addressed through careful analysis, systematic testing, and an iterative debugging process, contributing to a more robust understanding of the xv6 operating system.