OS-II: CS3523

Programming Assignment 5: xv6 Paging

## Task-1: Printing the page table entries:

Step 1: To define the system call number for the pgtPrint() system call, the following line was added to the syscall.h file.

#define SYS\_pgtPrint 22

Step 2: To specify the prototype of the pgtPrint() system call, the following lines were added to the prototypes of system calls in the syscall.c file.

extern uint64 sys\_pgtPrint(void);

[SYS\_pgtPrint] sys\_pgtPrint,

Step 3: The user level function for the pgtPrint() system call was defined by adding the following line to the user.h file.

int pgtPrint(void);

Step 4: The usys.pl file now has the following line added to specify the pgtPrint() system call.

entry("pgtPrint");

Step 5: Defined the system call sys\_pgtPrint() using the helper function pgtPrint() which is defined in the sysproc.c like the following.

Uint64

sys\_pgtPrint(void)

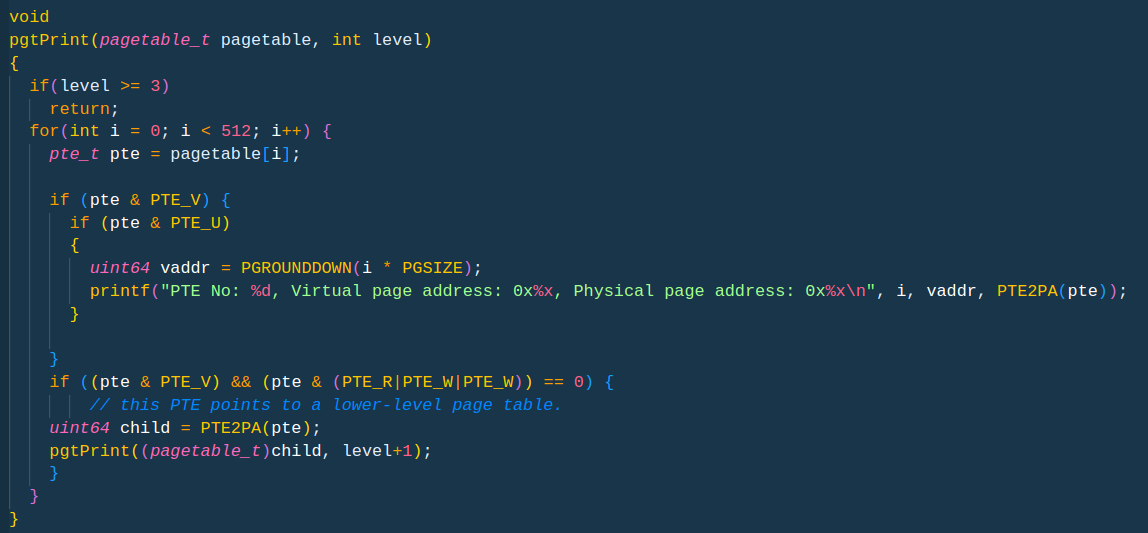
{

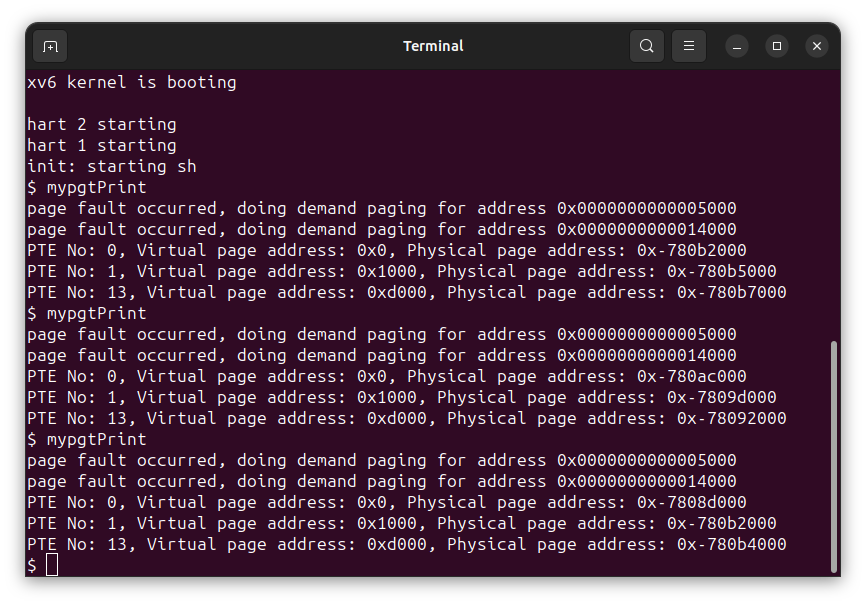
pgtPrint(myproc()->pagetable, 0);

return 0;

}

Step 6: To print the valid page table entries, the pgtPrint() function was defined in the sysproc.c file.

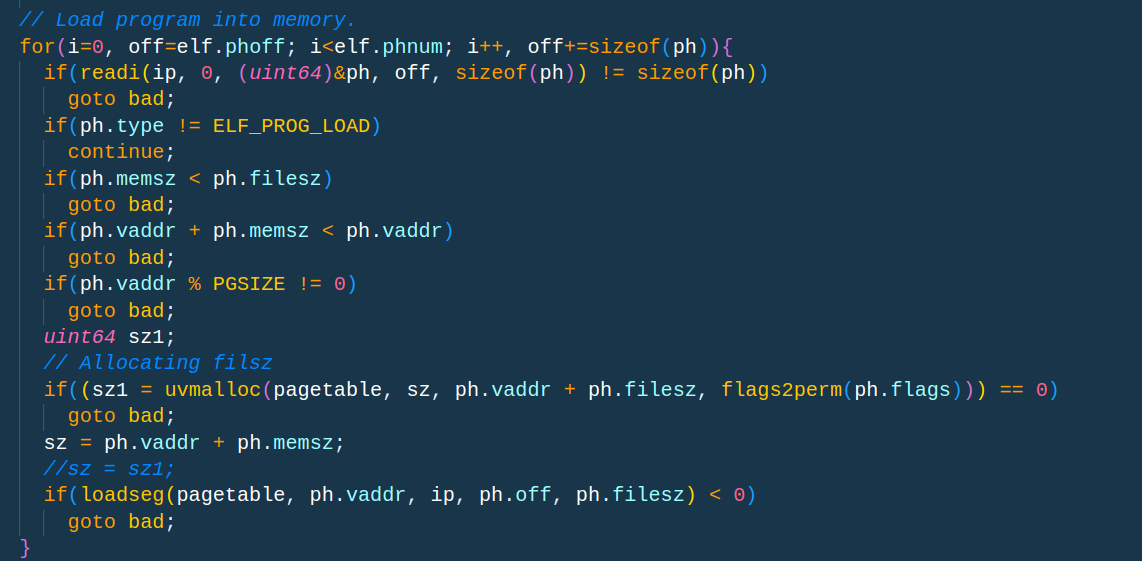


Output:

## Task-2: Implement demand paging:

Step 1: Modifications in exec():

It is possible for the filesz in a program section header to be smaller than the memsz, in which case the space between them should be filled with zeros (for C global variables) as opposed to being read from the file. We therefore change the exec() function to prevent the global variables from getting the pages until they are used if memsz is bigger than filesz. In reality, we are only allocating filesz size, thus we accomplish this by setting the fictitious allocating size memsz.

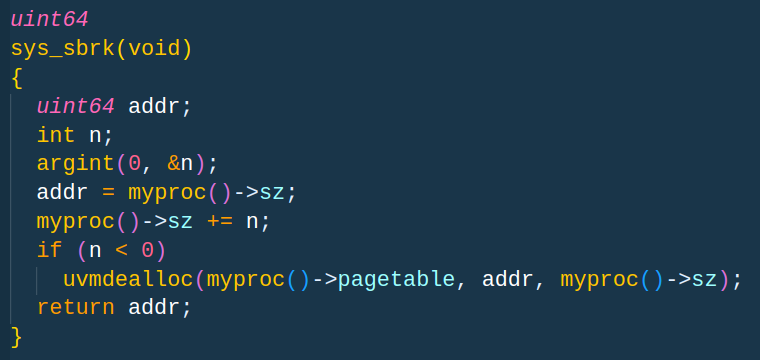


Step 2: Modifications in trap.c:

To trap the page storage and page load faults, the usertrap() function now has the following elseif condition.

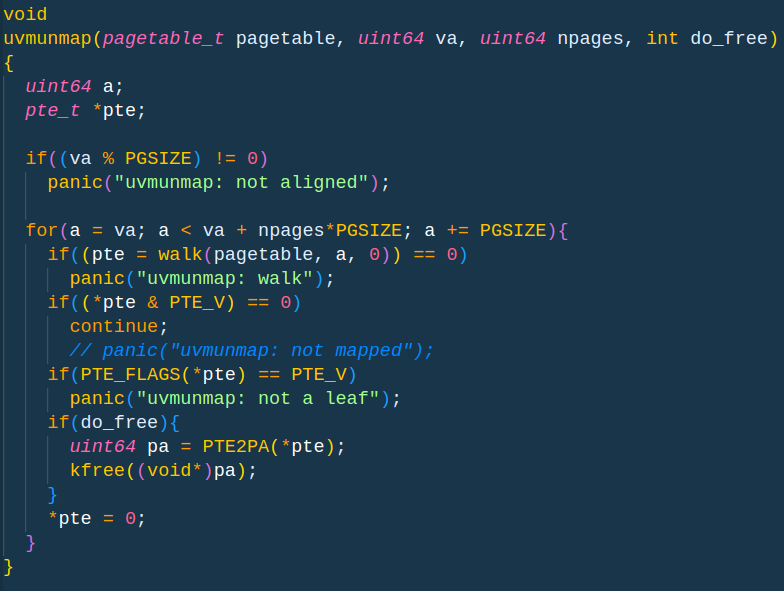


Step 3: Modified sbrk() function:



Step 4: Changes in uvmunmap:

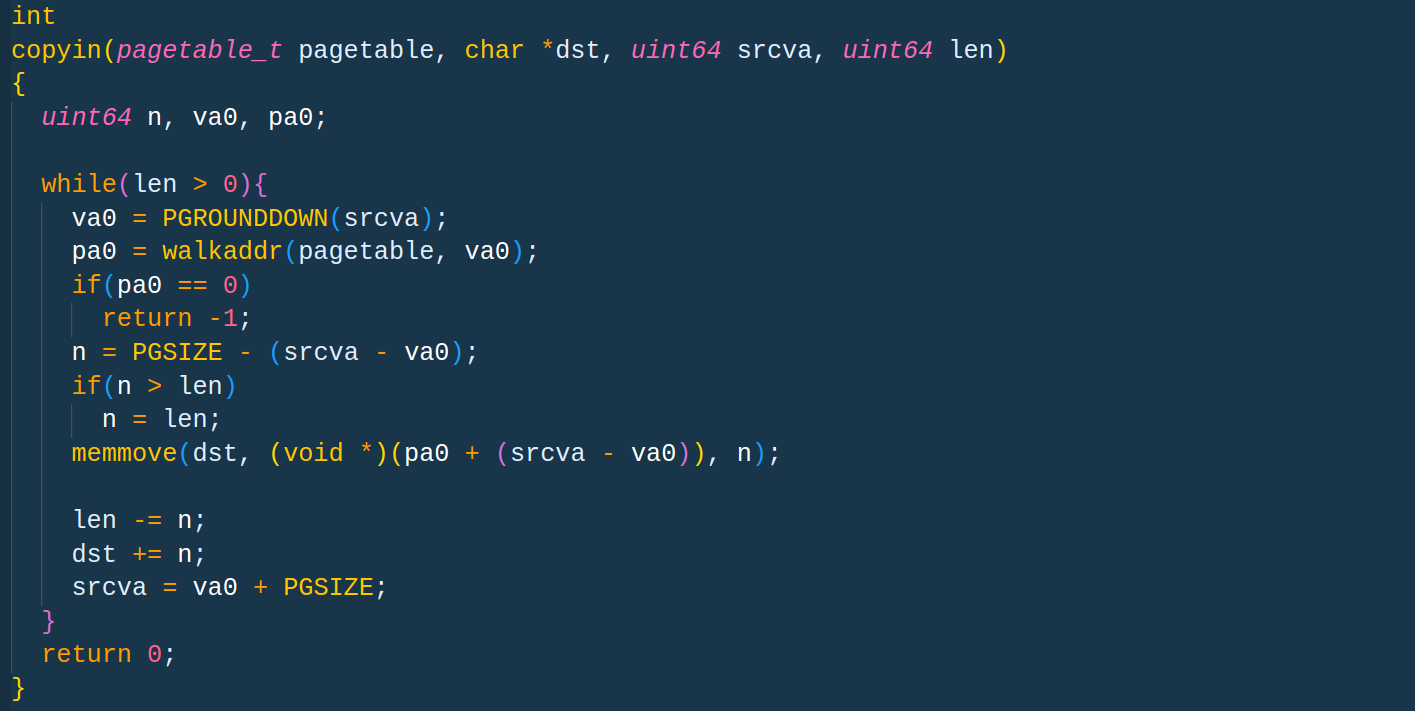
The panic("uvmunmap: not mapped"); in the vm.c file's uvmunmap() method have been commented out. in order to prevent the program from terminating in a panic if the page is not located in the page table.



Step 5: Chnages in copyin() and copyout():

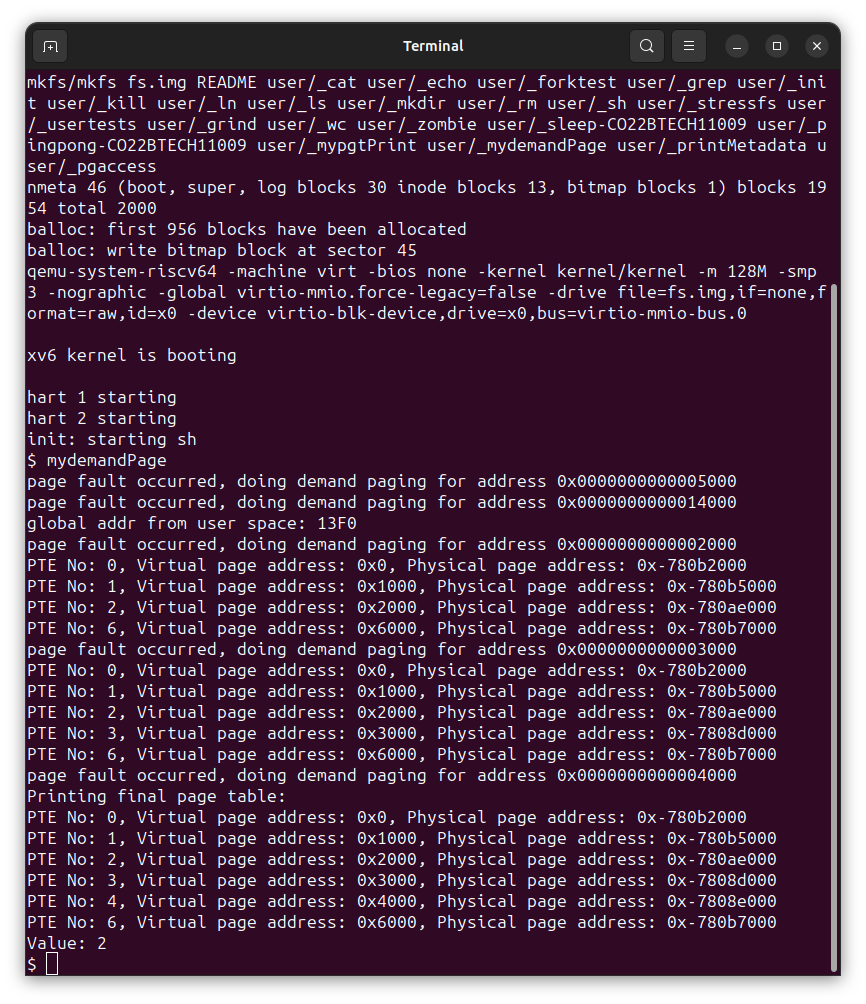
Address the situation where a process provides a legitimate address from sbrk() to a read or write system call, but the memory for that address hasn't been allocated yet.In the event that the physical address (PA) is invalid, they will first fail. If such a PA does not exist, then let's continue looping till we solve it. As the virtual address might be legitimate, the trap handler will assign further pages at a later time. The copy should be allowed to proceed. The following lines in the copyin() and copyout() routines of the vm.c file are changed to achieve this.



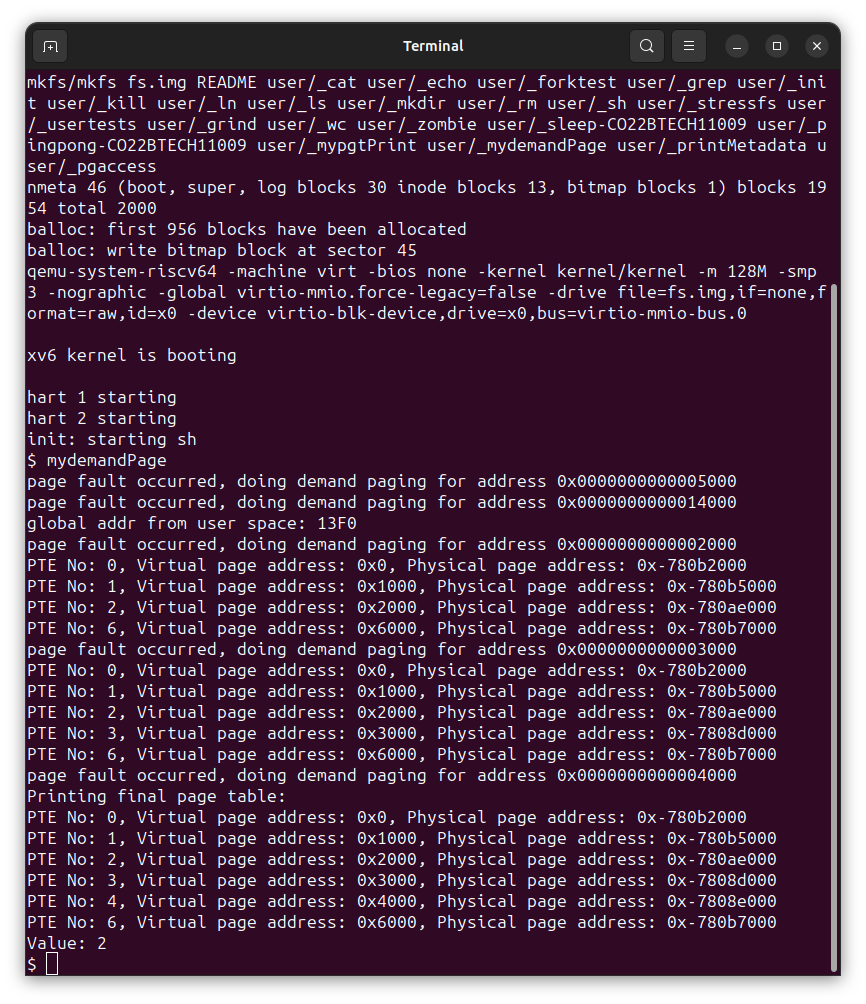


Output:

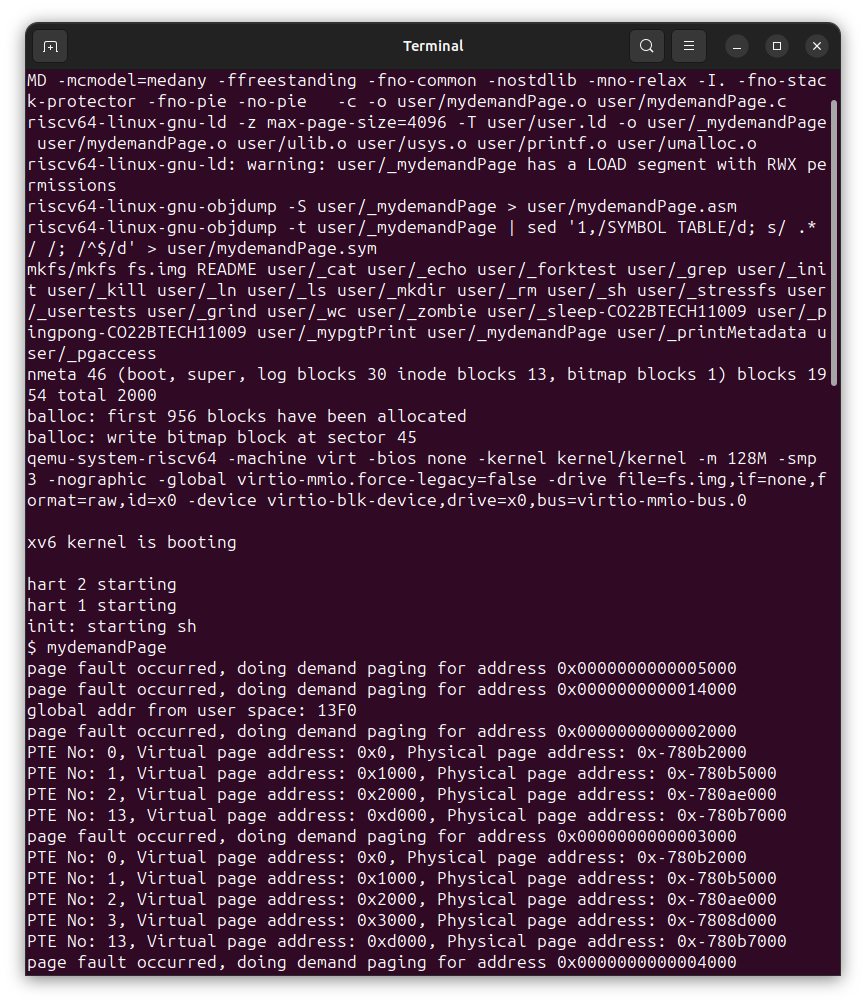
1. Global Array Size = 3000

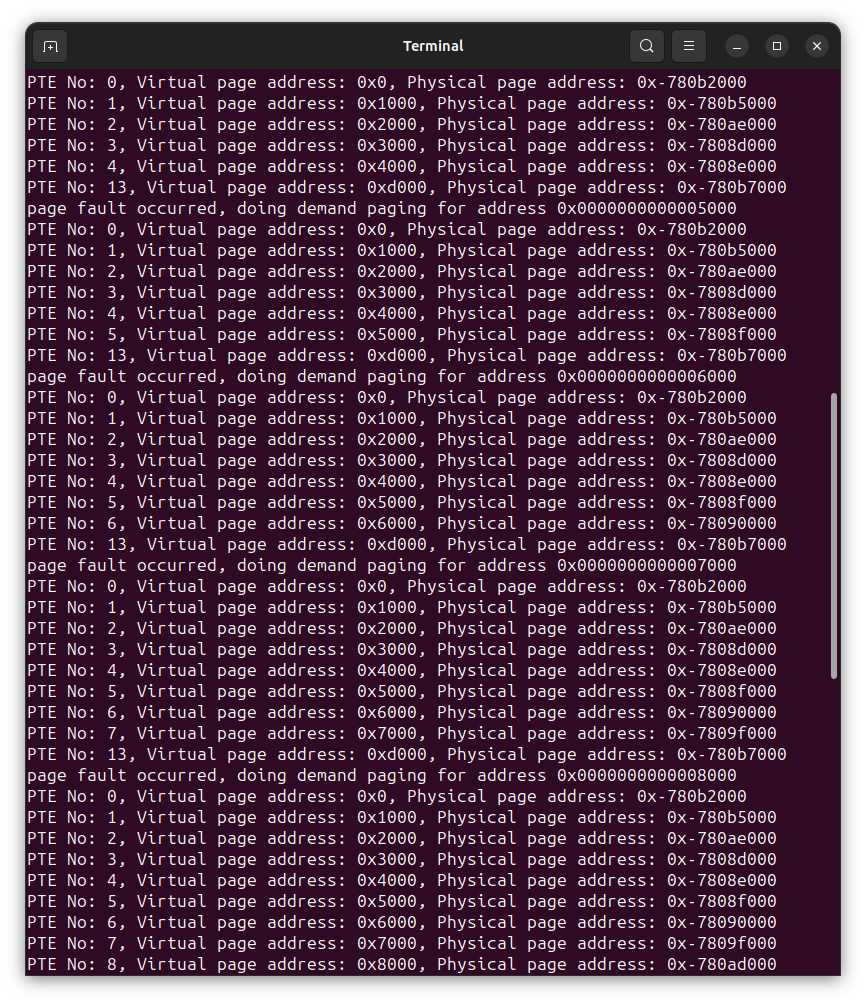


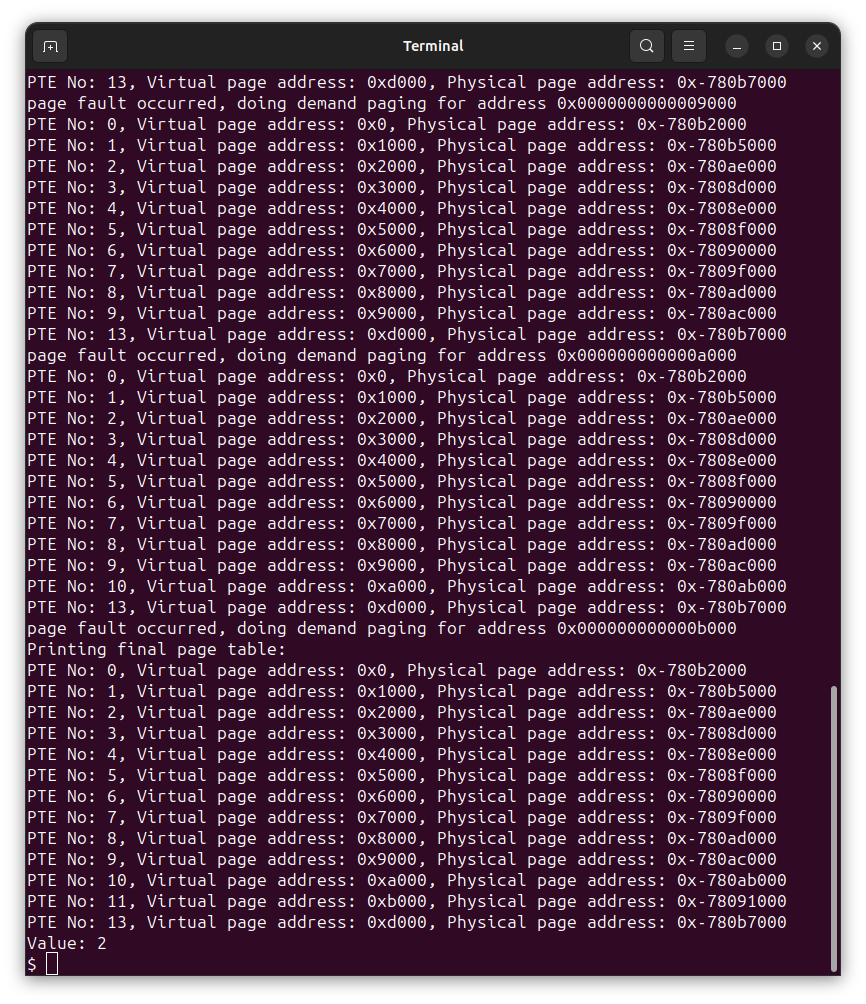
2. Global Array Size = 5000



3. Global Array Size = 10000







## Task-3: Implement logic to detect which pages have been accessed and/or dirty:

Step 1: To define the system call number for the pgaccess() system call, the following line was added to the syscall.h file.

#define SYS\_pgaccess 23

Step 2: To specify the prototype of the pgaccess() system call, the following lines were added to the prototypes of system calls in the syscall.c file.

extern uint64 sys\_pgaccess(void);

[SYS\_pgtPrint] sys\_pgaccess,

Step 3: The user level function for the pgaccess() system call was defined by adding the following line to the user.h file.

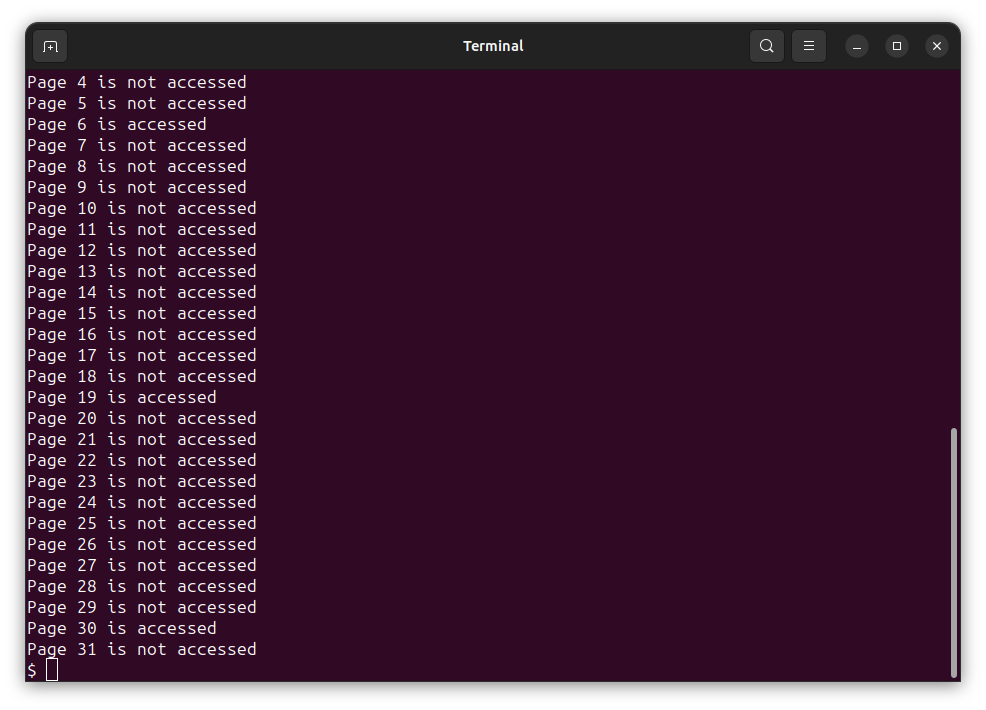
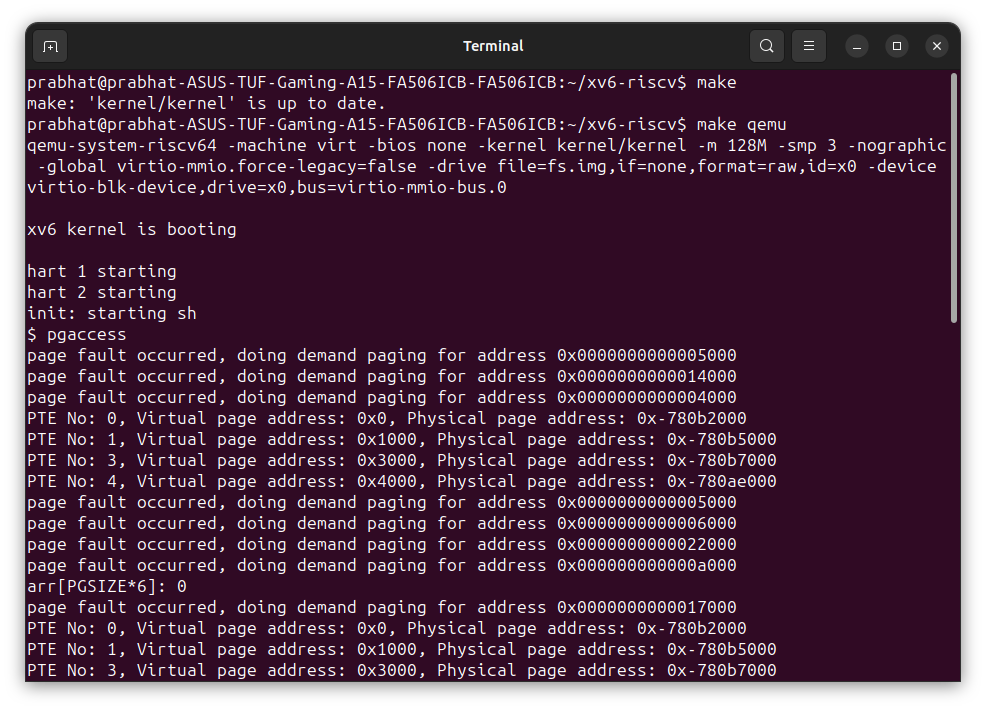
int pgaccess(void\*, int, uint64\*);

Step 4: The usys.pl file now has the following line added to specify the pgaccess() system call.

entry("pgaccess");

Step 5: Defined the system call sys\_pgtPrint().

Output:



# Understanding System Calls and Lessons Learned

### System Calls:

System calls provide a way for user-level processes to request services from the operating system kernel. They act as a bridge between user space and kernel space, enabling processes to perform privileged operations such as accessing hardware, managing files, and controlling processes. When a system call is invoked, the CPU switches from user mode to kernel mode, allowing the kernel to execute the requested operation on behalf of the process. Once the operation is completed, control returns to the user space.

### **Lessons Learned:**

1. Kernel Modification: This assignment involved modifying the xv6 kernel to implement new system calls, pgtPrint() and pgaccess().
2. Understanding Page Tables: Implementing pgtPrint() required understanding page tables and how to traverse them to extract information about page table entries.
3. Demand Paging: Implementing demand paging involved modifying the kernel's behavior to allocate memory pages on demand, rather than upfront during process creation.
4. Accessing and Modifying Page Table Entries: Implementing pgaccess() required understanding how to access and manipulate page table entries to detect page access and modification.
5. Debugging and Experimentation: Throughout the assignment, debugging techniques and experimentation were crucial for understanding the behavior of the modified kernel and user-level programs.

# Observations and Reasoning for Experiments

### Experiment 1: Global Array Size

* Observation: Increasing the size of the global array arrGlobal led to changes in the number of valid page table entries.
* Reasoning: Larger global arrays consume more memory, requiring additional page table entries to map the corresponding virtual addresses to physical memory. Thus, the number of valid entries increased as the array size grew.

### Experiment 2: Local Array Size

* Obseravation: Changing the size of the local array arrLocal within the main function did not affect the number of valid page table entries.
* Reasoning: Local variables are allocated on the stack, and their size does not impact the page table entries directly. Therefore, the number of valid entries remained constant regardless of the array size.

### Experiment 3: Repeated Execution

* Observation: Across multiple executions of the user program, the number of page table entries remained the same.
* Reasoning: The size of the code and data segments of the user program remained constant between executions, resulting in the same memory layout and thus the same number of page table entries. Additionally, xv6 likely reuses existing page table entries for unchanged memory regions, contributing to the consistency across executions.