CS4375-13948 Fall 2023
Denisse Loya Villalobos
dloyavilla@miners.utep.edu

Homework Report
Submitted on 11-13-2023

HW 4: Lazy Allocation for xv6

Task 1. freepmem() system call

To integrate the `freemem()` system call into the xv6 operating system, several key modifications were made across critical files. The implementation involved updating user-level and kernel-level components to ensure the proper execution of the new functionality.

User-Level Modifications:

- user.h: The user prototype for the `freemem()` system call was added to `user.h`. This file serves as the interface between user-level programs and the underlying operating system.
- usys.pl: An entry for the `freemem` system call was included in `usys.pl`. This file aids in generating system call stubs for user-level programs.

Kernel-Level Modifications:

- syscall.h: The system call number for `freemem` was defined in `syscall.h`. This file acts as a central repository for system call-related constants.
- syscall.c: The system call dispatcher in `syscall.c` was updated to recognize and dispatch the `freemem` system call.
- sysproc.c: In this file, the `sys_freemem` function was implemented. This function, in turn, calls another function named `freeCount`, responsible for counting the amount of free memory. This logic is encapsulated within the kernel.
- kalloc.c: The `freeCount` function, crucial for calculating free memory, was implemented in `kalloc.c`. This file was chosen due to its direct access to the linked list of memory in the `kmem.freelist`.

Additional User Commands:

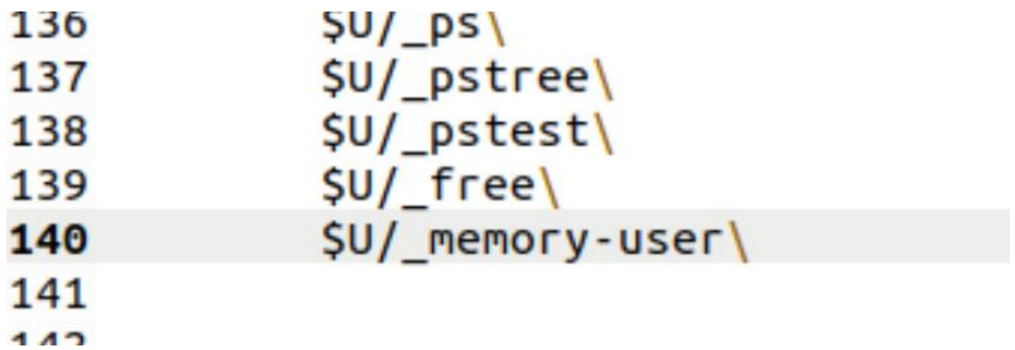
- Two new user commands, `free.c` and `memory-user.c`, were introduced to the `user` folder. These commands were subsequently added to the `Makefile` to enable their compilation and execution.

Testing and Results

- The implemented `freemem()` system call was thoroughly tested using the `free` command. Multiple invocations of the command were made to assess the accuracy of the free memory count.
- Additionally, the `memory-user.c` program, designed to interact with the `freemem` system call, was tested to ensure correct processing.

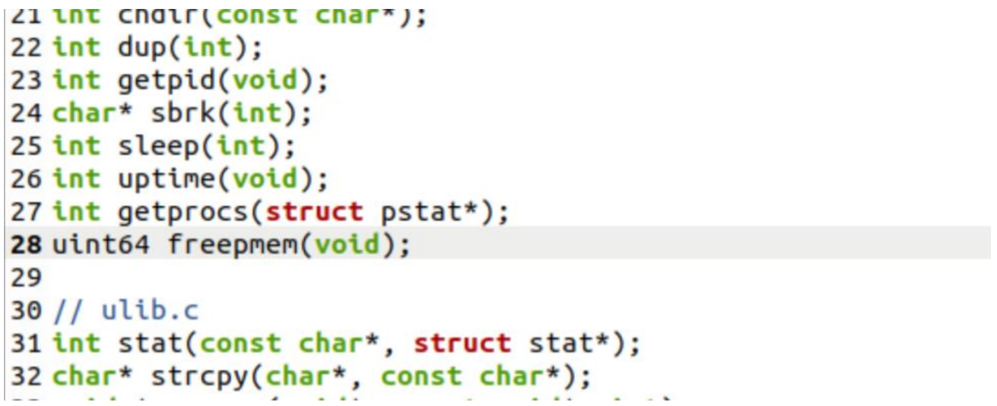
Learnings

- The task provided valuable insights into the intricacies of implementing a system call, requiring modifications across various files.
- Understanding the role of the linked list in `kalloc.c` and leveraging it for memory-related operations proved essential.



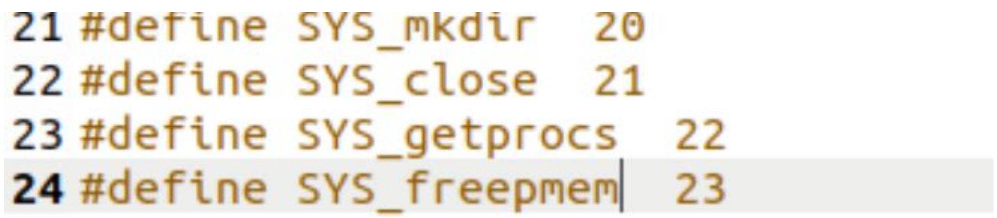
```
	$U/_ps\
	$U/_pstree\
	$U/_pstest\
	$U/_free\
	$U/_memory-user\

```

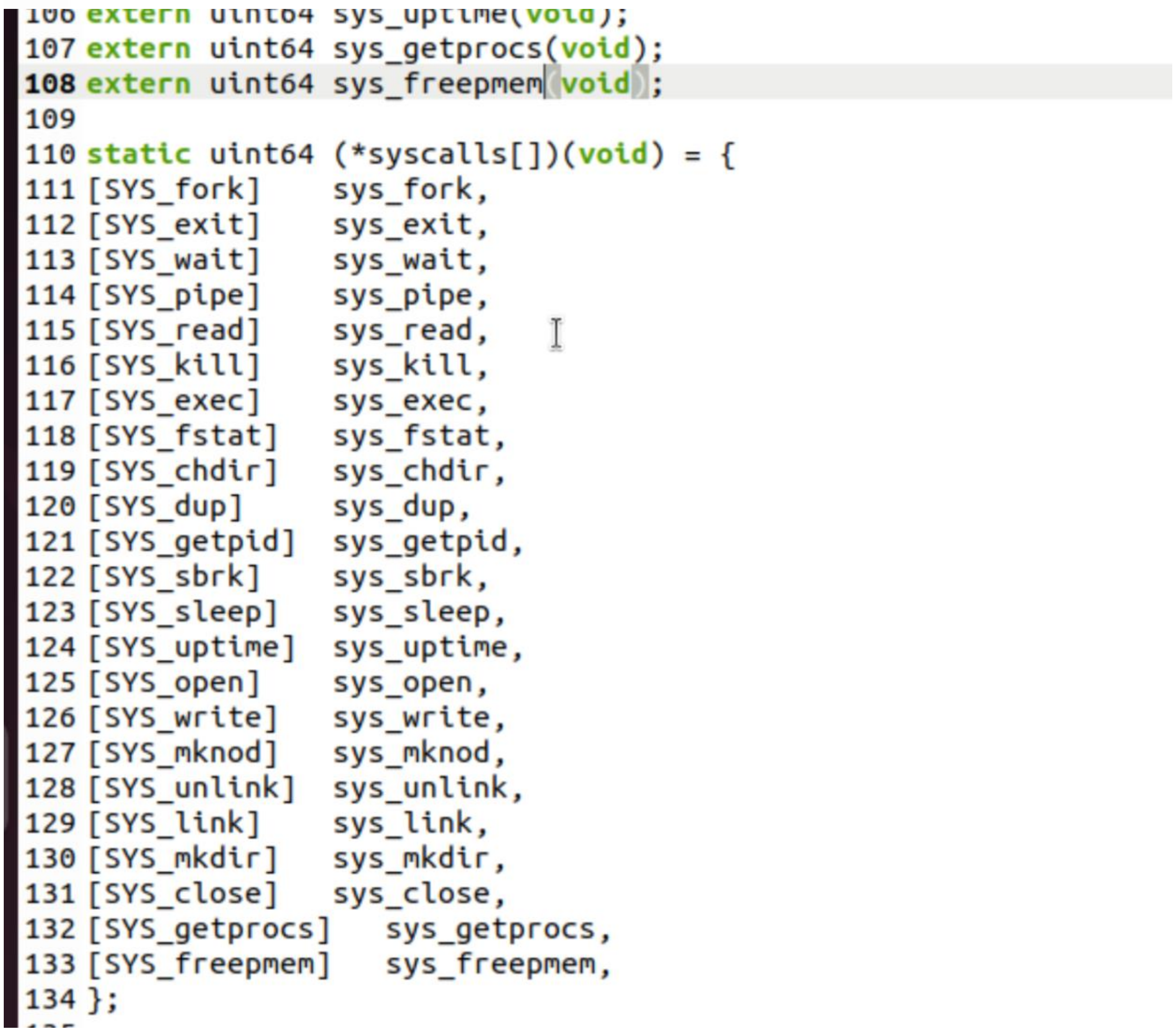


```
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int getprocs(struct pstat*);
uint64 freepmem(void);

// ulib.c
int stat(const char*, struct stat*);
char* strcpy(char*, const char*);
```

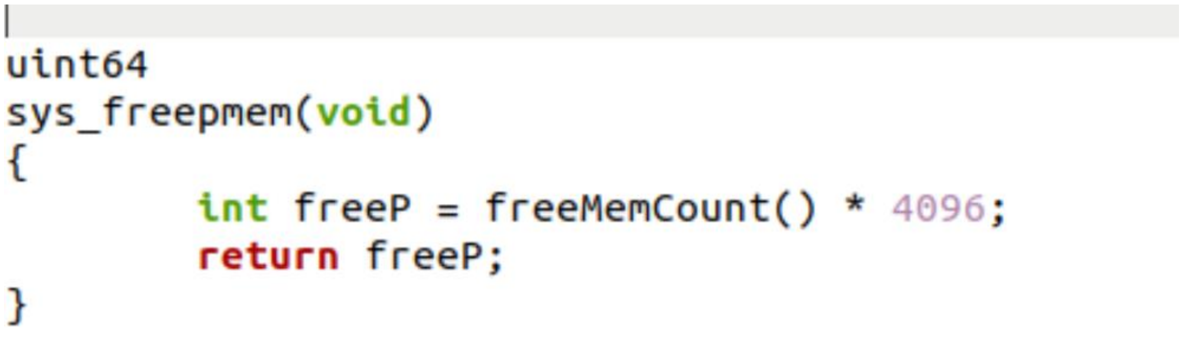


```
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_getprocs  22
#define SYS_freepmem  23
```

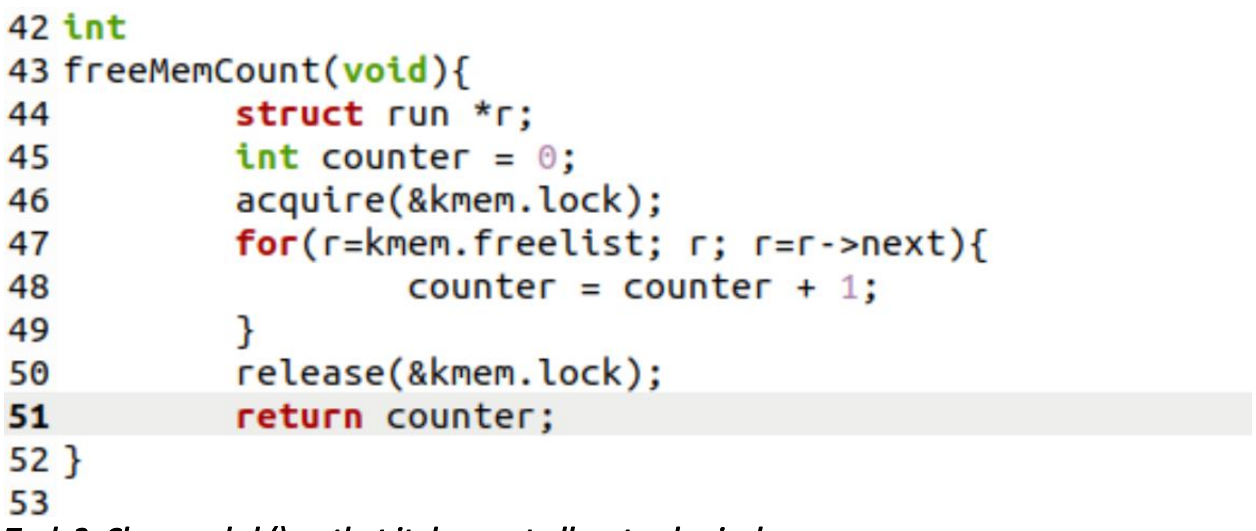


```
extern uint64 sys_uptime(void);
extern uint64 sys_getprocs(void);
extern uint64 sys_freepmem(void);

static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_getprocs]   sys_getprocs,
[SYS_freepmem]   sys_freepmem,
};
```



```
uint64
sys_freepmem(void)
{
        int freeP = freeMemCount() * 4096;
        return freeP;
}
```



```
int
freeMemCount(void){
        struct run *r;
        int counter = 0;
        acquire(&kmem.lock);
        for(r=kmem.freelist; r; r=r->next){
                counter = counter + 1;
        }
        release(&kmem.lock);
        return counter;
}

```

<u>Task 2. Change sbrk() so that it does not allocate physical memory.</u>

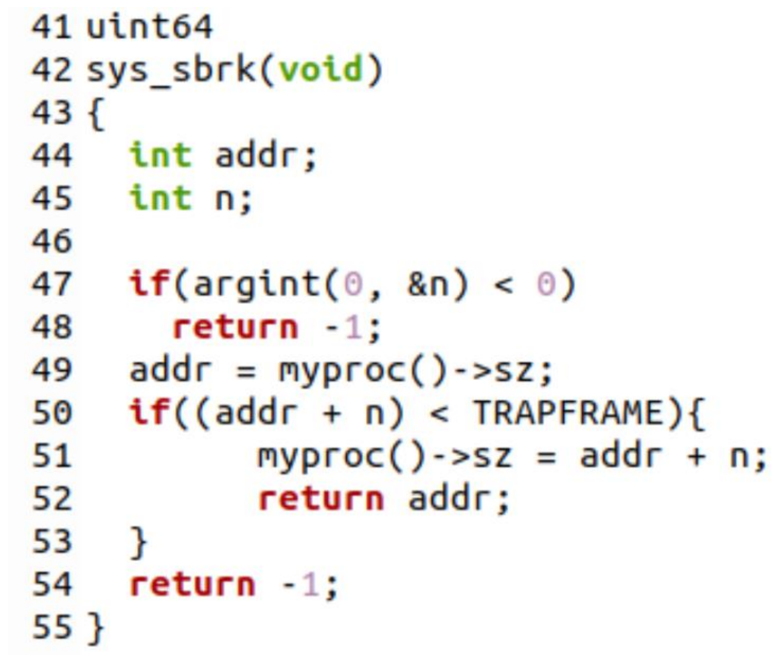
When you make the modification of avoiding calling growproc() and skip the allocation of physical memory, you might observe errors when running xv6 commands that use heap memory. If you attempt to run a command that uses heap memory (e.g., a command that performs dynamic memory allocation using `malloc()`), errors may occur. Because the `sbrk()` system call is responsible for growing the heap space in the process's virtual address space. By directly changing the `sz` field without allocating physical memory (skipping `growproc()`), you are essentially increasing the virtual memory space but not allocating corresponding physical memory pages. When the command attempts to access the newly "allocated" memory, it may result in a page fault or other memory-related errors since the physical memory pages are not actually allocated.

Learnings:

- This task highlights the distinction between virtual and physical memory allocation.
- Modifying `sbrk()` inside sysproc.c to allocate only virtual memory space without corresponding physical memory may lead to runtime errors.

Challenges:

- Understanding the consequences of allocating virtual memory without corresponding physical memory is crucial.
- Identifying and addressing the errors that arise when attempting to use the "allocated" memory is part of the learning process.



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

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  if((addr + n) < TRAPFRAME){
        myproc()->sz = addr + n;
        return addr;
  }
  return -1;
}
```

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

Modifications in `kernel/trap.c`:

- In `kernel/trap.c`, locate the `usertrap()` function.
- Check the value of `scause` to identify load or store faults.

Check for Load/Store Faults:

- If the exception is a load or store fault (`scause` code), proceed with additional checks.

Validate Faulting Address:

- Retrieve the faulting address from the `stval` register.
- Check if the faulting address is within the process's allocated virtual memory.

Handle the Fault:

- If the faulting address is valid but physical memory hasn't been allocated, allocate a physical memory frame using `kalloc()`.
- Install the page table mapping for the virtual page containing the faulting address using `mappages()`.

Explanation of Errors:

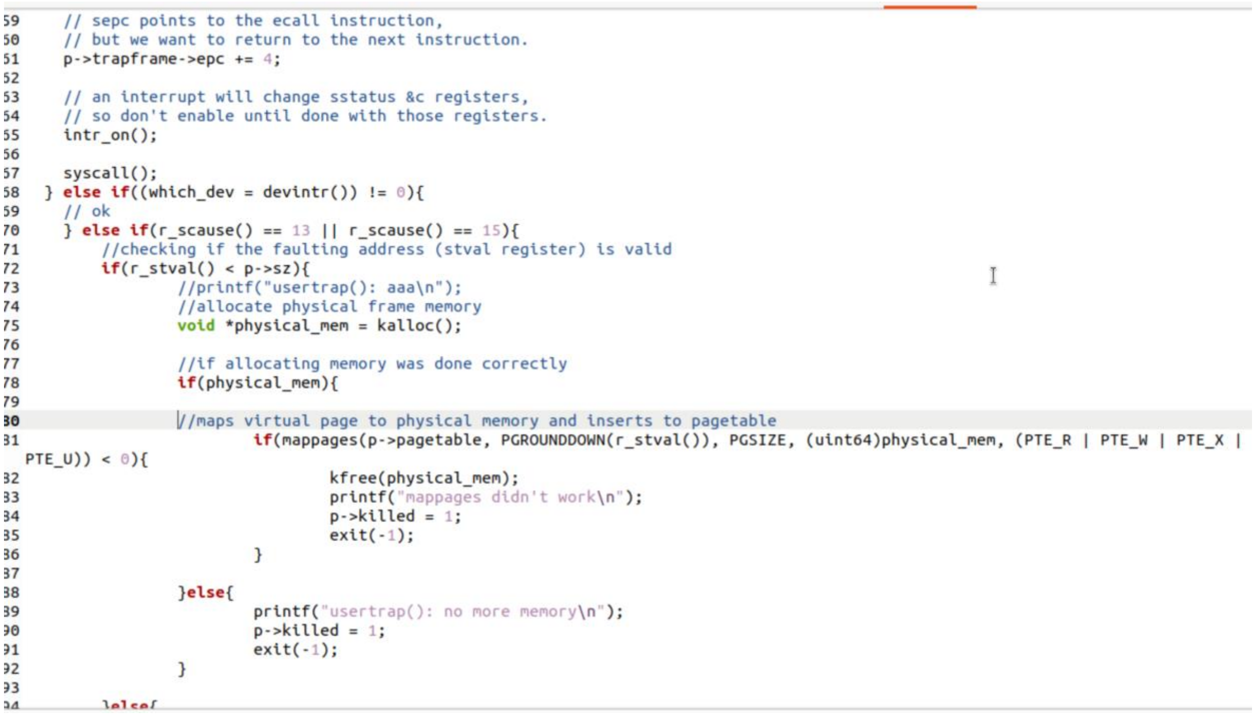
When you run xv6 and attempt to execute a user command, you may encounter errors due to accessing unallocated memory. The errors occur because the memory space accessed by the user command is not physically allocated, and the page table mapping is not installed.

Learnings:

- This task deepens your understanding of how the operating system handles load or store faults caused by accessing unallocated memory.
- It reinforces the importance of checking exception codes and validating addresses in the context of memory access.

Challenges:

- Challenges may arise in correctly identifying the type of exception and ensuring that the faulting address is properly validated.
- Debugging errors and ensuring that the memory allocation and page table mapping are performed accurately can be challenging.



```
    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
    } else if(r_scause() == 13 || r_scause() == 15){
        //checking if the faulting address (stval register) is valid
        if(r_stval() < p->sz){
                //printf("usertrap(): aaa\n");
                //allocate physical frame memory
                void *physical_mem = kalloc();

                //if allocating memory was done correctly
                if(physical_mem){

                //maps virtual page to physical memory and inserts to pagetable
                        if(mappages(p->pagetable, PGROUNDDOWN(r_stval()), PGSIZE, (uint64)physical_mem, (PTE_R | PTE_W | PTE_X |
PTE_U)) < 0){
                                kfree(physical_mem);
                                printf("mappages didn't work\n");
                                p->killed = 1;
                                exit(-1);
                        }

                }else{
                        printf("usertrap(): no more memory\n");
                        p->killed = 1;
                        exit(-1);
                }

        }else{
```

Task 4. Fix kernel panic and any other errors.

Kernel Panic from `uvmunmap()`:

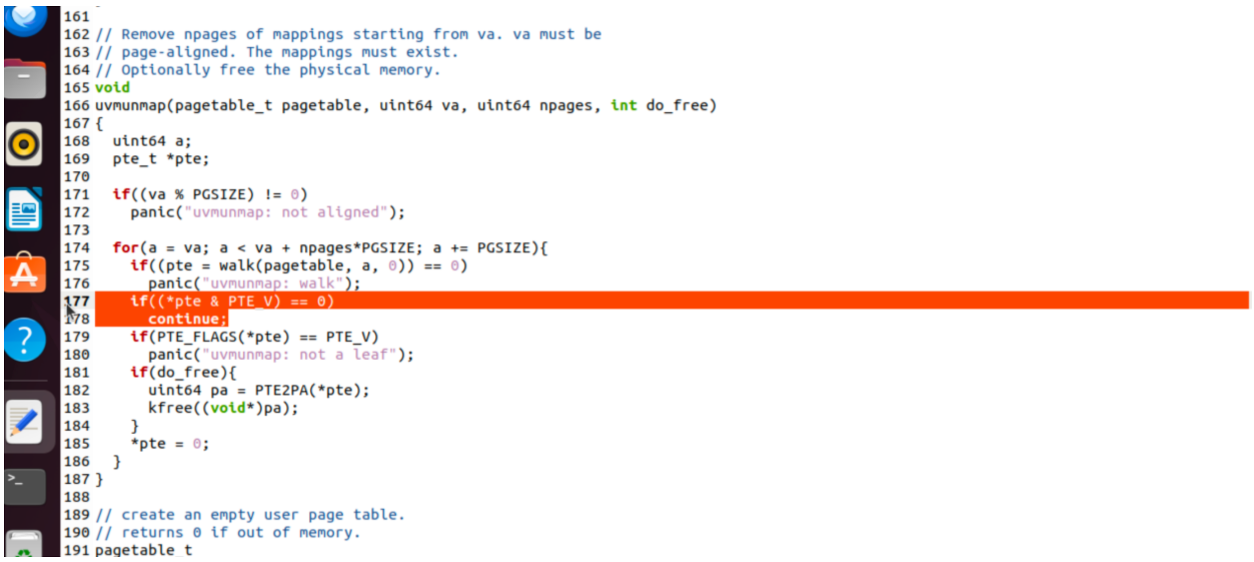
- The kernel panic likely occurs because `uvmunmap()` expects all virtual memory pages to have been mapped, but your modifications in Task 2 may have left some pages unmapped.
- Inspect the `uvmunmap()` function in `kernel/vm.c`.
- Ensure that it handles the case where not all virtual memory pages are mapped.

Learnings:

- This task reinforces your understanding of virtual memory management in an operating system.
- Debugging kernel panics and errors provides practical experience in identifying and resolving issues at the kernel level.

Difficulties:

- Difficulties may arise in pinpointing the exact locations of errors and understanding the interactions between different components.
- Using the `gdb` debugger, print statements, and thorough code review can help overcome these difficulties.



```
161
162 // Remove npages of mappings starting from va. va must be
163 // page-aligned. The mappings must exist.
164 // Optionally free the physical memory.
165 void
166 uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
167 {
168   uint64 a;
169   pte_t *pte;
170
171   if((va % PGSIZE) != 0)
172     panic("uvmunmap: not aligned");
173
174   for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
175     if((pte = walk(pagetable, a, 0)) == 0)
176       panic("uvmunmap: walk");
177     if((*pte & PTE_V) == 0)
178       continue;
179     if(PTE_FLAGS(*pte) == PTE_V)
180       panic("uvmunmap: not a leaf");
181     if(do_free){
182       uint64 pa = PTE2PA(*pte);
183       kfree((void*)pa);
184     }
185     *pte = 0;
186   }
187 }
188
189 // create an empty user page table.
190 // returns 0 if out of memory.
191 pagetable_t
```

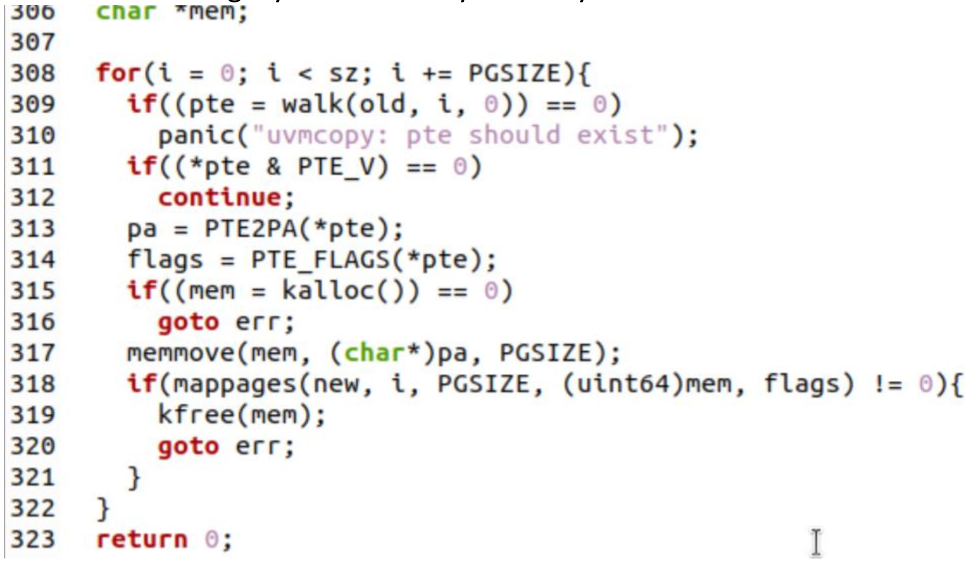
<u>Task 5. Test your lazy memory allocation.</u>

The provided test cases successfully demonstrate and validate the behavior of lazy memory allocation in the xv6 operating system, before doing a final test to the code a modification in line 312 in file vm.c had to be done, in order to fix the panic error. The test programs systematically allocate, touch, and free memory in various scenarios, allowing us to observe the dynamic memory management behavior. Here are the key takeaways from the test results:

- Displays memory usage information using the `free` command.
- The first line shows the free memory in kilobytes (`-k` option).
- The second line shows free memory in bytes.
- The third line shows free memory in megabytes (`-m` option).
- Executes the `memory-user` test program with the following arguments: `1 4 1`.
- The program allocates, touches, and frees memory in multiple steps.
- Allocates 1 mebibyte of memory and prints the address returned by `malloc`.
- Displays the updated free memory after allocation.
- Touches and frees the previously allocated 1 mebibyte of memory.

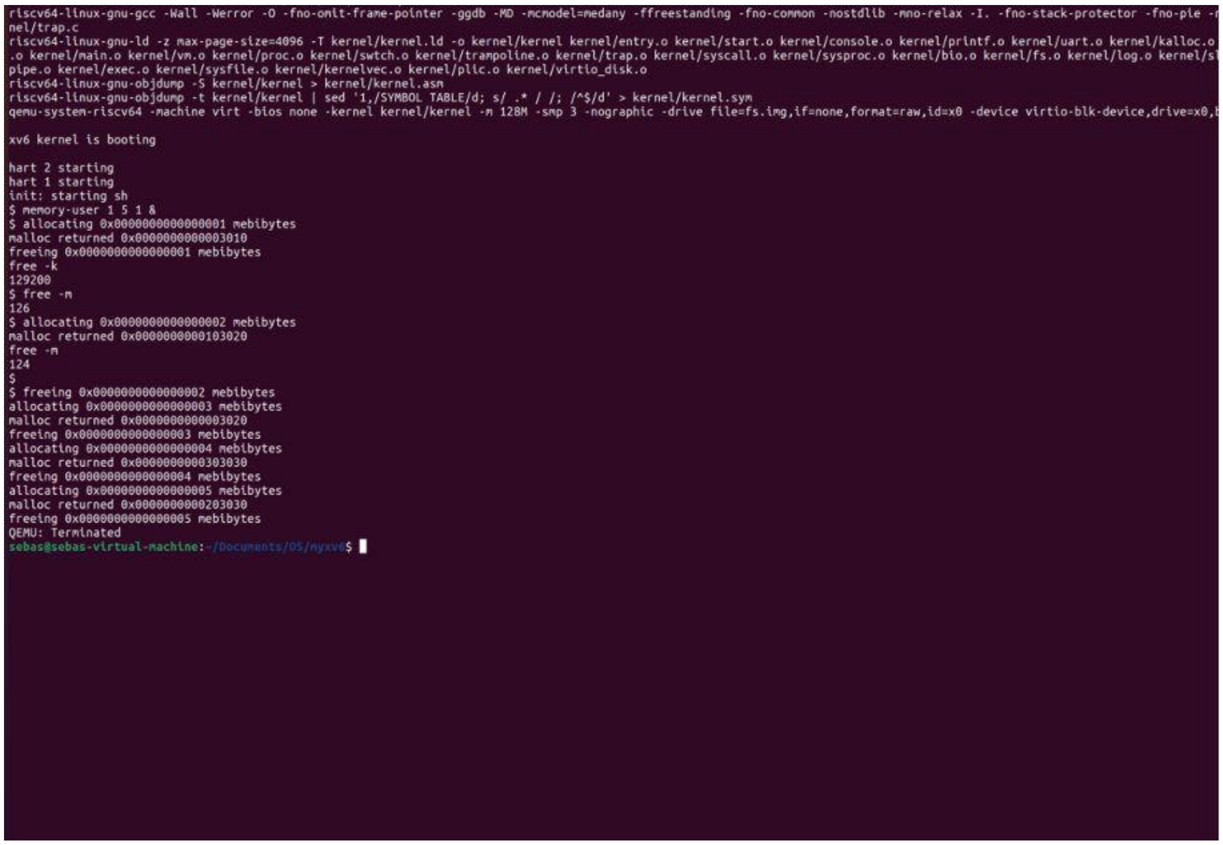
- Displays the updated free memory after freeing.

These tests provide confidence in the proper functioning of memory management features, and any difficulties or unexpected behaviors would likely require further analysis and debugging to maintain the integrity and reliability of the system.



```c
  char *mem;

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
      continue;
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
      kfree(mem);
      goto err;
    }
  }
  return 0;
```



```
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -
nel/trap.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 kernel/kalloc.o
.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/s
pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/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 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ memory-user 1 5 1 &
$ allocating 0x0000000000000001 mebibytes
malloc returned 0x0000000000003010
freeing 0x0000000000000001 mebibytes
free -k
129200
$ free -m
126
$ allocating 0x0000000000000002 mebibytes
malloc returned 0x0000000000103020
free -m
124
$
$ freeing 0x0000000000000002 mebibytes
allocating 0x0000000000000003 mebibytes
malloc returned 0x0000000000003020
freeing 0x0000000000000003 mebibytes
allocating 0x0000000000000004 mebibytes
malloc returned 0x0000000000303030
freeing 0x0000000000000004 mebibytes
allocating 0x0000000000000005 mebibytes
malloc returned 0x0000000000203030
freeing 0x0000000000000005 mebibytes
QEMU: Terminated
sebas@sebas-virtual-machine:~/Documents/OS/myxv6$
```