Project 4: Xv6 VM Layout

Part A: Pointer Dereference

System Environment

Operating System: Xv6

Compiler/Development Environment: GCC (GNU Compiler Collection).

Languages Used: C

Overview

In this assignment, we modify the xv6 operating system to handle null pointer dereference differently. Currently, xv6 allows null pointer dereferences without throwing exceptions, leading to unexpected behavior. The goal is to alter the page table setup to leave the first three pages (0x0 - 0x3000) unmapped, forcing a trap when dereferencing a null pointer.

Additionally, a `nullpointer.c` file has been created to systematically test null pointer dereferencing and observe the modified behavior in the xv6 environment.

Approach

Null Pointer Dereference Program

- Developed a straightforward program that intentionally dereferences a null pointer.
- Tested the program on both Linux and xv6 to observe and compare the divergent behaviors.

Page Table Modification

- Explored how xv6 establishes and manages its page table.
- Updated the code to leave the first three pages unmapped, with the code segment now commencing at 0x3000.
- Investigated the exec() and userinit() functions to gain insight into the initialization of the address space.

fork() Functionality

- Inspected the fork() function to ensure that the child process inherits the modified address space.
- Updated pertinent sections of the code to guarantee the proper inheritance of the modified address space by the child process.

Kernel Parameter Handling

- Examined how the kernel manages parameters, with a focus on pointers.
- Implemented necessary checks to prevent issues related to bad pointers, ensuring robust parameter handling.

Makefile Changes

- Updated the makefile to reflect changes in the code segment's starting address.
- Altered the entry point to align with the new beginning of the code segment.

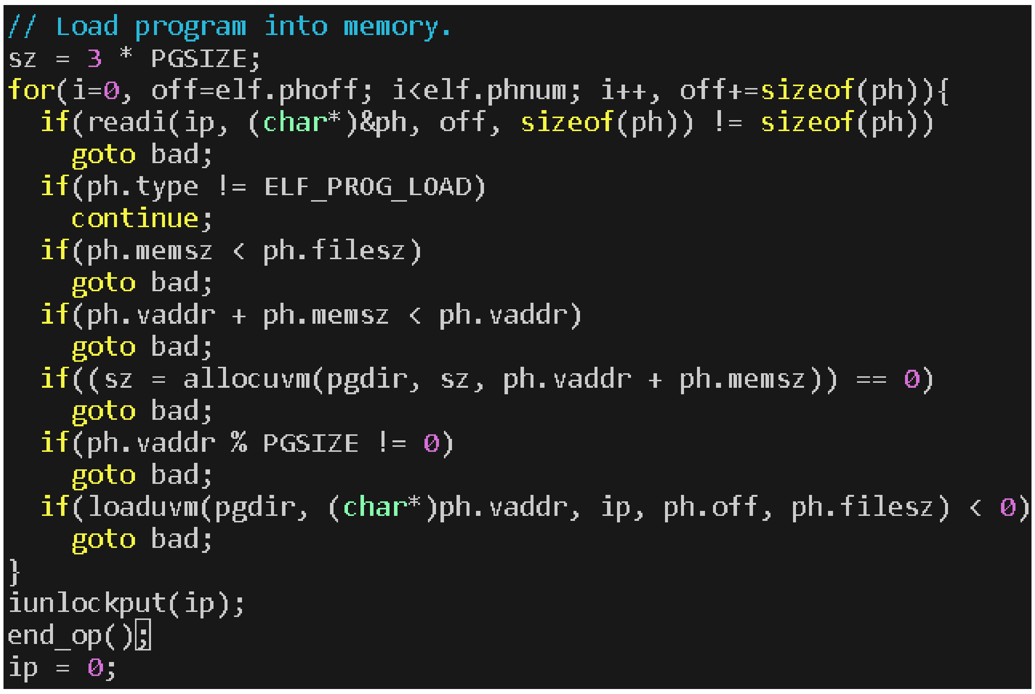
Building and Running

- Followed the standard xv6 build process, ensuring that modifications were incorporated.
- Compiled the program using gcc and executed it on Linux and xv6 to observe and compare results.
- Launched xv6 and executed various user programs to demonstrate the modified behavior regarding null pointer dereferences.

Code Changes

exec.c

- sz = 0 is changed to sz = 3 * PGSIZE
- Updated the initial size of the process address space (sz) to be three times the page size (PGSIZE). This change ensures that the first three pages remain unmapped, aligning with the project's goal.

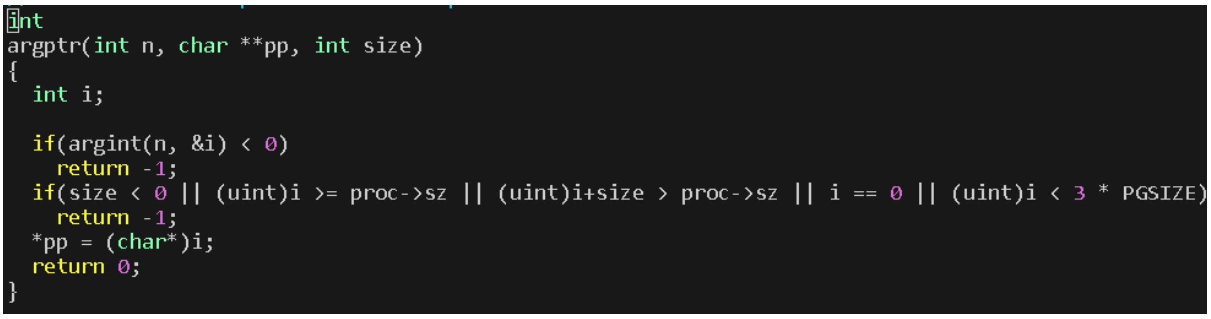


```
// Load program into memory.
sz = 3 * PGSIZE;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
iunlockput(ip);
end_op();
ip = 0;
```

syscall.c

- Added i == 0 to the Condition:
  i == 0 checks whether the argument i is equal to zero.
  This condition ensures that the pointer i is not pointing to a null address (address 0).
- Added (uint)i < 3 * PGSIZE to the Condition:

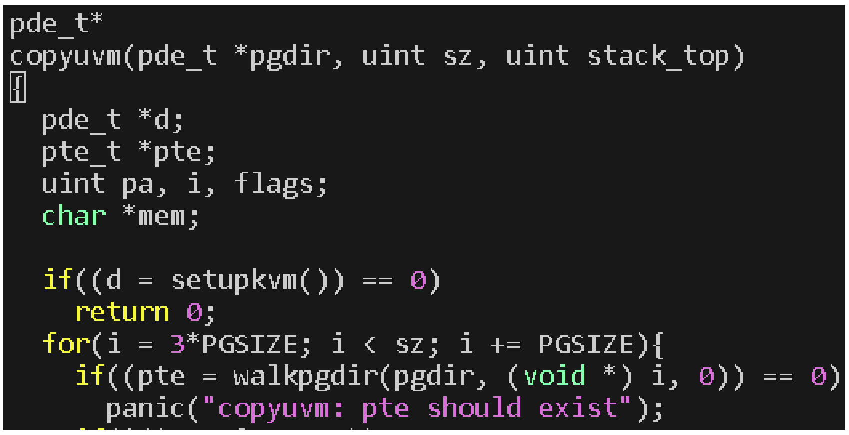
(uint)i < 3 * PGSIZE checks whether the argument i is less than three times the page size.
This condition ensures that the pointer i is not within the first three pages of the process's address space.



```
int
argptr(int n, char **pp, int size)
{
  int i;

  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= proc->sz || (uint)i+size > proc->sz || i == 0 || (uint)i < 3 * PGSIZE)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

vm.c

- Adjusted the initialization of the loop variable i in the copyuvm() function. The modification ensures that the loop starts from the beginning of the fourth page (3 * PGSIZE), aligning with the decision to leave the first three pages unmapped.

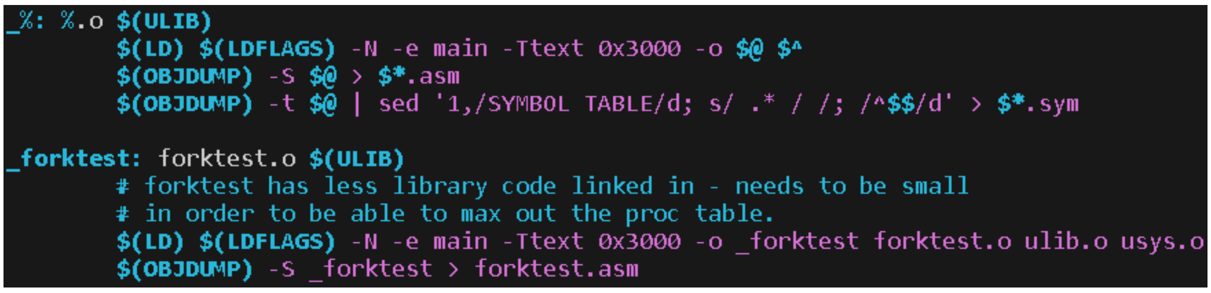


```
pde_t*
copyuvm(pde_t *pgdir, uint sz, uint stack_top)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 3*PGSIZE; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
```

makefile

- -Ttext 0x3000: This option tells the linker to set the starting address of the text (code) section in the linked executable to 0x3000. In other words, it determines where the machine code instructions of the program will be loaded into memory when the program is executed.

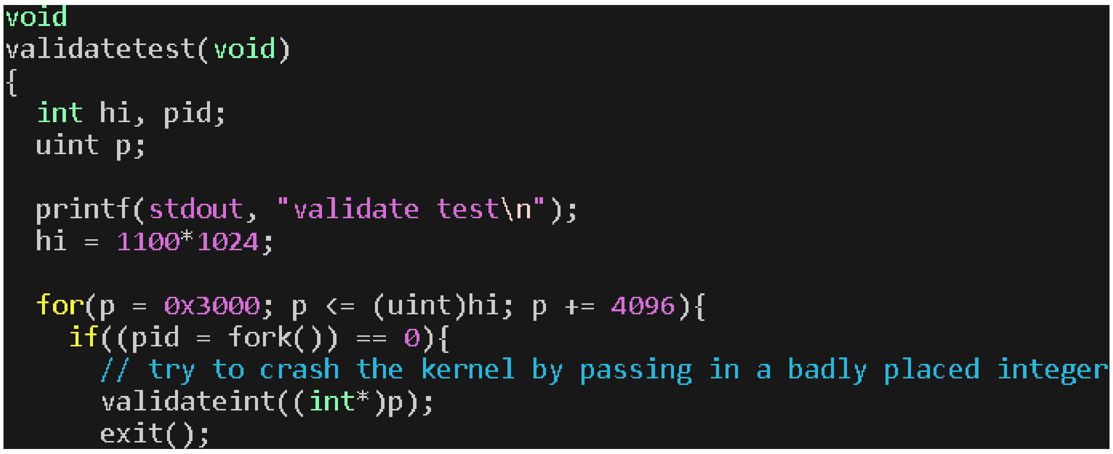
- 0x3000 is a specific memory address. In the context of xv6, this address is often used as the starting point for user-level programs. The choice of 0x3000 aligns with the xv6 memory layout, which typically leaves the first few pages of the address space unmapped to catch null pointer dereference errors.



```
_%: %.o $(ULIB)
        $(LD) $(LDFLAGS) -N -e main -Ttext 0x3000 -o $@ $^
        $(OBJDUMP) -S $@ > $*.asm
        $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym

_forktest: forktest.o $(ULIB)
        # forktest has less library code linked in - needs to be small
        # in order to be able to max out the proc table.
        $(LD) $(LDFLAGS) -N -e main -Ttext 0x3000 -o _forktest forktest.o ulib.o usys.o
        $(OBJDUMP) -S _forktest > forktest.asm
```

usertests.c

- In the original version, the loop started at p = 0, iterating over pages from the very beginning of the address space. However, in the modified version, the loop is adjusted to start at 0x3000, which corresponds to the new beginning of the code segment based on the earlier changes made in the project.

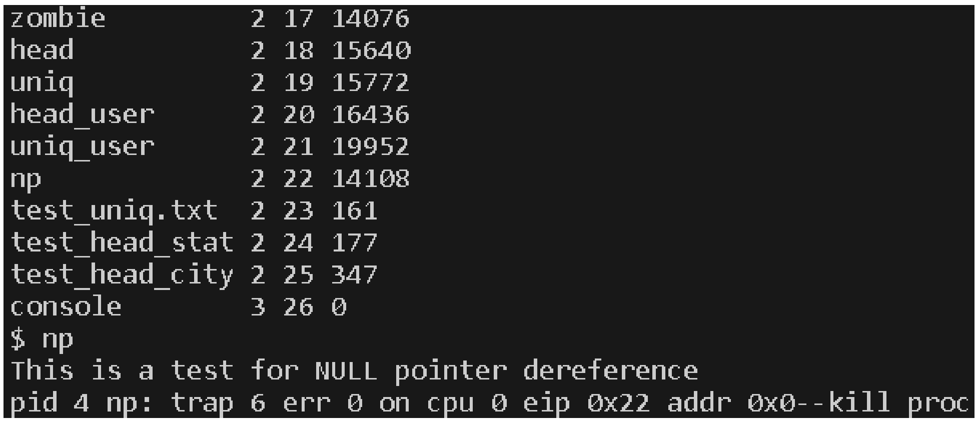


```
void
validatetest(void)
{
  int hi, pid;
  uint p;

  printf(stdout, "validate test\n");
  hi = 1100*1024;

  for(p = 0x3000; p <= (uint)hi; p += 4096){
    if((pid = fork()) == 0){
      // try to crash the kernel by passing in a badly placed integer
      validateint((int*)p);
      exit();
```

Results

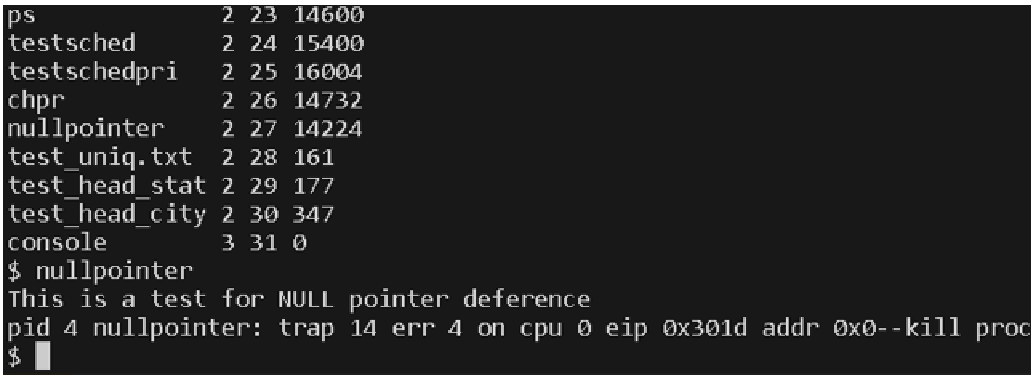
NULL Pointer Dereference: A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

- In the original xv6 operating system, unlike many Linux distributions, there is no specific function designed for checking null pointer dereference.



```
zombie          2 17 14076
head            2 18 15640
uniq            2 19 15772
head_user       2 20 16436
uniq_user       2 21 19952
np              2 22 14108
test_uniq.txt   2 23 161
test_head_stat  2 24 177
test_head_city  2 25 347
console         3 26 0
$ np
This is a test for NULL pointer dereference
pid 4 np: trap 6 err 0 on cpu 0 eip 0x22 addr 0x0--kill proc
```

- The core modification involves adjusting xv6 to initiate execution from the fourth page (virtual address 0x3000) instead of the first. This alteration addresses a specific issue in the original xv6 where starting from the first page results in null pointers pointing to valid addresses, thus circumventing null pointer exceptions. In the updated configuration, the system starts from 0x3000, leaving the first three pages unmapped to handle null pointer dereference differently.



```
ps             2 23 14600
testsched      2 24 15400
testschedpri   2 25 16004
chpr           2 26 14732
nullpointer    2 27 14224
test_uniq.txt  2 28 161
test_head_stat 2 29 177
test_head_city 2 30 347
console        3 31 0
$ nullpointer
This is a test for NULL pointer deference
pid 4 nullpointer: trap 14 err 4 on cpu 0 eip 0x301d addr 0x0--kill proc
$ 
```

As we can see the 'eip' points to 0x301d skipping the first 3 pages and throwing the trap error pointing out the null pointer dereference.

Part B: Stack Rearrangement

System Environment

Operating System: Xv6

Compiler/Development Environment: GCC (GNU Compiler Collection).

Languages Used: C

Overview

In this assignment, we rearranged the xv6 address space to accommodate a new configuration. Specifically, the goal is to place the stack at the high end of the xv6 user address space, followed by a gap, and then the heap. Additionally, three unmapped pages are to be preserved, creating a new layout that enhances the structure of the xv6 address space.

Approach

Identify Code Sections

Conducted an in-depth examination of the xv6 source code to pinpoint specific sections responsible for the initialization of the code, management of the heap, and allocation of the user stack.

Update Stack Placement

Altered the code to redefine the user stack's placement, shifting it to the high end of the xv6 user address space. This involved meticulous adjustments to the initial stack pointer (**sp**) and ensured the seamless functioning of stack-related operations in the modified configuration.

Introduce a Gap

Implemented strategic logic to introduce a minimum gap of five pages between the stack and the heap. This intricate adjustment was accomplished through precise modifications to memory allocation and page table entries, creating a designated unallocated region.

Modify Address Space Accounting

Carefully managed the **sz** field within the **proc** struct to accurately track the sizes of both the code and heap. Additionally, introduced a new accounting parameter (**stacktop**) to independently monitor the stack size, allowing for more granular control and adaptability.

Update Relevant Code

n critical files such as **exec.c** and **vm.c**, where the **sz** field is utilized, meticulously updated the code to seamlessly integrate the new accounting mechanism for stack size. Ensured that operations related to address space size remained in harmony with the modified accounting structure.

Automatic Stack Growth

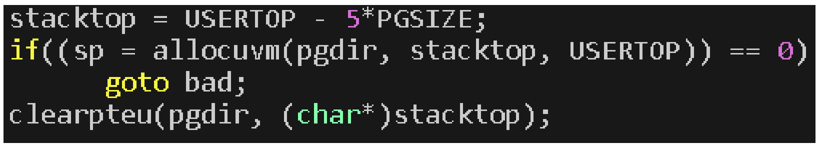
Implemented a robust logic system for the automatic growth of the stack:

- Detected faults on the page above the stack.
- Dynamically allocated a new page specifically for the stack.
- Effectively mapped the new page into the address space.
- Enabled the process to continue running without any interruptions or premature termination, contributing to a more resilient and adaptive system.

Code Changes

exec.c

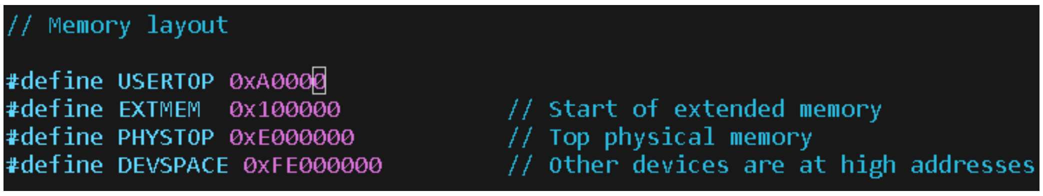
- Declare a variable named `stacktop` within the exec function. Allocate two pages at the next boundary, with the initial page marked as inaccessible and the subsequent page designated as the user stack. Ensure the process's top stack is set using the `stacktop` variable. Place the stack towards the higher end of the address space, followed by a gap of at least five pages, and then the heap.



```
stacktop = USERTOP - 5*PGSIZE;
if((sp = allocuvm(pgdir, stacktop, USERTOP)) == 0)
      goto bad;
clearpteu(pgdir, (char*)stacktop);
```

memlayout.h

- Establish a variable named `USERTOP` for the operating system.



```
// Memory layout

#define USERTOP 0xA0000
#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0xE000000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses
```

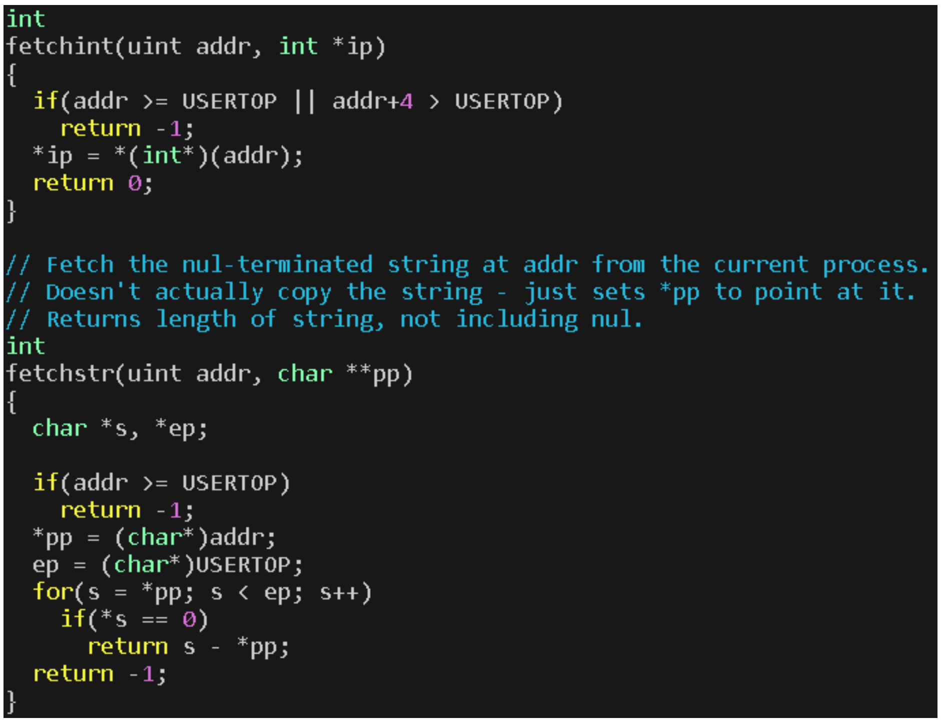
proc.c

- During the initialization of the initial user process, configure the `topstack` of the process to be 0. When creating a new process through forking, replicate the process state from p to the new process, incorporating the updated variable `proc->stacktop`.
- in **userinit():**
  - p->stacktop = 0;
- in **fork():**
  - // Copy process state from p.

```
if((np->pgdir = copyuvm(proc->pgdir, proc->sz, proc->stacktop)) == 0){
   kfree(np->kstack);
   np->kstack = 0;
   np->state = UNUSED;
   return -1;
}
np->stacktop = proc->stacktop;
```

syscall.c

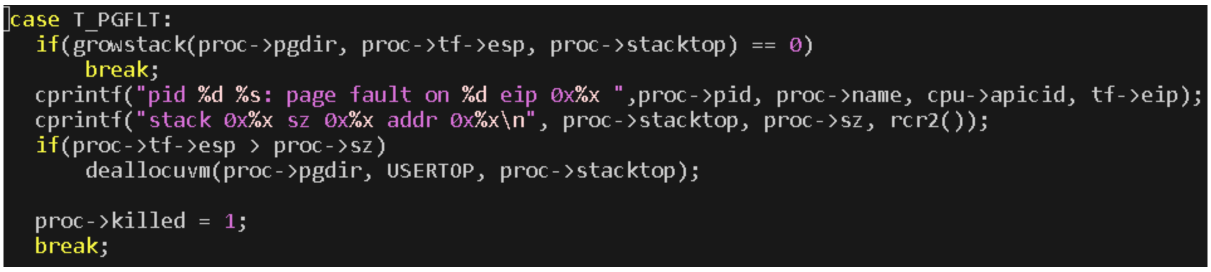
- Ensured that the current process's address remains below the USERTOP limit.



```
int
fetchint(uint addr, int *ip)
{
  if(addr >= USERTOP || addr+4 > USERTOP)
    return -1;
  *ip = *(int*)(addr);
  return 0;
}

// Fetch the nul-terminated string at addr from the current process.
// Doesn't actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
int
fetchstr(uint addr, char **pp)
{
  char *s, *ep;

  if(addr >= USERTOP)
    return -1;
  *pp = (char*)addr;
  ep = (char*)USERTOP;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}
```

trap.c

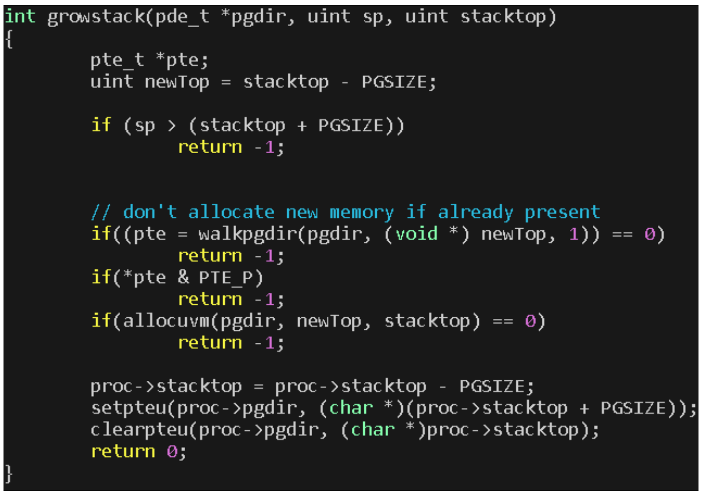
- Implemented a scenario within the operating system to handle page faults. In the event of a page fault, where a process attempts to access a region of memory that is not currently in physical RAM, the system takes appropriate measures, resulting in the termination of the respective process. This addition ensures that the operating system effectively manages page faults, maintaining system stability and preventing potential issues associated with memory access violations.



```
case T_PGFLT:
  if(growstack(proc->pgdir, proc->tf->esp, proc->stacktop) == 0)
      break;
  cprintf("pid %d %s: page fault on %d eip 0x%x ",proc->pid, proc->name, cpu->apicid, tf->eip);
  cprintf("stack 0x%x sz 0x%x addr 0x%x\n", proc->stacktop, proc->sz, rcr2());
  if(proc->tf->esp > proc->sz)
      deallocuvm(proc->pgdir, USERTOP, proc->stacktop);

  proc->killed = 1;
  break;
```

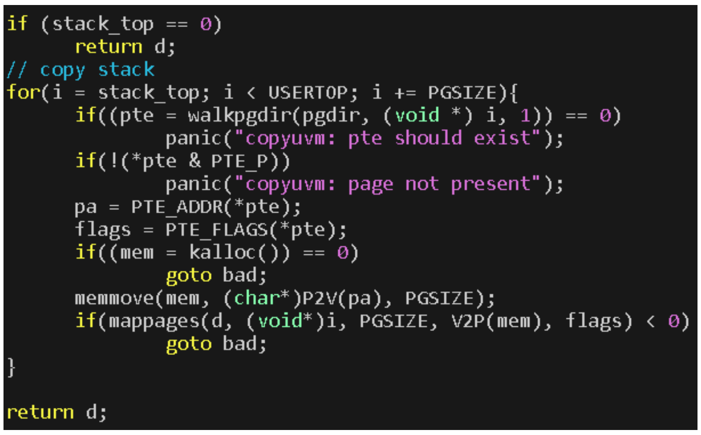
vm.c

- To facilitate the expansion of the stack, a dedicated function was developed to cater specifically to this functionality.



```
int growstack(pde_t *pgdir, uint sp, uint stacktop)
{
        pte_t *pte;
        uint newTop = stacktop - PGSIZE;

        if (sp > (stacktop + PGSIZE))
                return -1;


        // don't allocate new memory if already present
        if((pte = walkpgdir(pgdir, (void *) newTop, 1)) == 0)
                return -1;
        if(*pte & PTE_P)
                return -1;
        if(allocuvm(pgdir, newTop, stacktop) == 0)
                return -1;

        proc->stacktop = proc->stacktop - PGSIZE;
        setpteu(proc->pgdir, (char *)(proc->stacktop + PGSIZE));
        clearpteu(proc->pgdir, (char *)proc->stacktop);
        return 0;
}
```

- To replicate memory for a child process, introduced code into the **copyuvm** function. This modification ensures that the child process inherits the necessary memory from its parent during the duplication process.



```
if (stack_top == 0)
        return d;
// copy stack
for(i = stack_top; i < USERTOP; i += PGSIZE){
        if((pte = walkpgdir(pgdir, (void *) i, 1)) == 0)
                panic("copyuvm: pte should exist");
        if(!(*pte & PTE_P))
                panic("copyuvm: page not present");
        pa = PTE_ADDR(*pte);
        flags = PTE_FLAGS(*pte);
        if((mem = kalloc()) == 0)
                goto bad;
        memmove(mem, (char*)P2V(pa), PGSIZE);
        if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0)
                goto bad;
}

return d;
```

Analysis of `uniq`, `head`, and `ps` commands

After null pointer dereference

uniq command

- Executing the **uniq** command with intentional null pointer dereference appears to proceed without errors or abnormal terminations.
- The implementation of null pointer dereference in the **uniq** command does not appear to cause any noticeable alterations or disturbances, as the output remains consistent with its standard behavior.
- The stack and memory layout within xv6 remain stable, suggesting that the intentional null pointer dereference in **uniq** is handled appropriately without causing layout distortions.

head command

- After the successful implementation of null pointer dereference, the **head** command operates smoothly without encountering any errors.
- The modifications made to the null pointer dereference do not result in any noticeable changes in the output or behavior of the **head** command. The intentional null pointer dereferencing within the head command does not visibly affect its standard functionality.
- The null pointer dereference changes in the **head** command do not seem to have any impact on the stack and memory layout in xv6, much like the behavior of uniq.

ps command

- After the deliberate implementation of a null pointer dereference, the **ps** command executes without any runtime problems.
- The **ps** command's behavior remains unchanged from its pre-modification state, and there are no observed disruptions that deviate from the expected outcome. Even with intentional null pointer dereferencing, there are no visible alterations in the output or execution of the **ps** command.
- The **ps** command in xv6 appears to handle modifications related to null pointer dereference without causing any significant disruptions to the stack and memory layout.

After Stack Rearrangement

uniq command

- The execution of the **uniq** command after the rearrangement of the stack implementation proceeds smoothly, without any visible errors or abnormal terminations.
- The **uniq** command maintains its standard functionality without any visible changes or disruptions in its execution even after the stack rearrangement. The output and behavior of the command remain consistent throughout.

- The arrangement of the stack and memory layout in xv6 is deliberately reorganized, resulting in the stack being located at the end of the address space and expanding in the opposite direction.

head command

- The execution of the post-stack rearrangement implementation of the **head** command proceeds smoothly and seamlessly, devoid of any errors or interruptions.
- The **head** command continues to exhibit consistent output and behavior after the modification, with no noticeable changes in its standard functionality. The rearrangement of the stack does not visibly affect the head command's performance.
- The arrangement of the stack and memory layout in xv6 is deliberately reorganized, positioning the stack at the conclusion of the address space and expanding in the opposite direction**.**

ps command

- After the rearrangement of the stack has been implemented, executing the **ps** command does not result in any errors or abrupt terminations.
- The **ps** command's performance and output remain unaffected by any alterations in the stack arrangement, and there are no discernible disturbances in its execution.
- The arrangement of the stack and memory layout in xv6 is deliberately reorganized, with the stack located at the end of the address space and expanding in the opposite direction.

The intentional reordering of the stack in **uniq**, **head**, and **ps** commands does not lead to any noticeable interruptions or mistakes while executing the commands. The arrangement of the stack and memory layout in xv6 reflects this deliberate reordering, with the stack now positioned at the end of the address space and expanding in the opposite direction.

Analysis of FCFS and Priority Scheduling

FCFS Scheduling

- The modified xv6 version was utilized to execute a series of processes using FCFS scheduling, which involved intentionally dereferencing a null pointer.
- Despite the intentional null pointer dereference, FCFS scheduling functioned as expected.
- There were no noticeable alterations in the order of process execution or the time taken for completion.
- The modifications made to the layout, as well as the intentional changes such as null pointer dereference, did not seem to impact the behavior of FCFS.
- The rearrangement of the stack, layout modifications, and null pointer dereference did not appear to have any influence on FCFS scheduling.
- The scheduling algorithm seamlessly adapted to the modified memory layout.

Priority Scheduling

- The modified xv6 version was utilized to execute a series of processes using Priority scheduling. This included deliberately dereferencing a null pointer.
- Despite this intentional modification, the Priority scheduling mechanism operated as expected. There were no discernible alterations in the order of process execution based on their priority levels.
- It appears that the intentional modifications, such as the null pointer dereference, did not have any noticeable impact on the behavior of Priority scheduling.
- Furthermore, the rearrangement of the stack, changes in layout, and null pointer dereference did not seem to exert any apparent influence on the effectiveness of the scheduling algorithm within the modified memory layout.

Resources Used

- **https://www.youtube.com/watch?v=NWicJxVENjg**
- **MIT PDOS Xv6 Handbook**
- **https://stackoverflow.com/questions/4007268/what-exactly-is-meant-by-de-referencing-a-null-pointer**
- **https://pdos.csail.mit.edu/6.828/2008/lec/l-threads.html**