## ReadMe

## Table of Contents

1. [Overview](#overview)
2. [Step 1: Initial Memory Size Adjustment](#step-1-initial-memory-size-adjustment)
3. [Step 2: Stack Allocation](#step-2-stack-allocation)
4. [Step 3: Addition of stack\_bottom Field](#step-3-addition-of-stack_bottom-field)
5. [Step 4: Initcode Section Update](#step-4-initcode-section-update)
6. [Step 5: Constant USERTOP Addition](#step-5-constant-usertop-addition)
7. [Step 6: Null Pointer Dereference Test Program](#X9c21a41da5983c9fd05320d61647c96657d1f00)
8. [Step 7: Modifications in proc.c and proc.h](#step-7-modifications-in-procc-and-proch)
9. [Step 8: Changes in Other Files](#step-8-changes-in-other-files)
   * [1. stack\_test.c](#X83cec53fbd8b8515d98d8d6c81bdc85bc008193)
   * [2. syscall.c](#X4d0d42629bf19f7af965613e612ba153151dab9)
   * [3. trap.c](#Xe9e030f7ad4783c3a2d59cf3834337f58f9c0df)
10. [Step 9: copystack Function](#step-9-copystack-function)
11. [Step 10: incStack Function](#step-10-incstack-function)
12. [Step 11: Rubrics](#step-11-rubrics) (Answers)
13. Step 12: Screenshots

## 1. Overview

This repository documents modifications made to the operating system kernel code to implement specific functionalities, including memory layout adjustments, stack handling during process creation, and the introduction of features for testing scenarios such as null pointer dereference and stack rearrangement.

## 2. Step 1: Initial Memory Size Adjustment

Modified the initial memory size (sz) to 0x3000, leaving the first three pages (0x0 - 0x3000) unmapped to create a scenario where dereferencing a null pointer leads to a trap.

sz = 0x3000; // Set the initial memory size to 0x3000

## 3. Step 2: Stack Allocation

Allocated six pages for the stack at the top of the user address space (USERTOP - PAGES \* PGSIZE). Page table entries (PTEs) for the five pages before the stack were cleared to make them invalid.

sz = PGROUNDUP(sz);  
int PAGES = 6;  
int stack\_bottom = USERTOP - PAGES \* PGSIZE;  
  
if ((stack\_bottom = allocuvm(pgdir, stack\_bottom, stack\_bottom + PAGES \* PGSIZE)) == 0)  
 goto bad;  
  
for (int i = 0; i < PAGES - i; i++) {  
 // Clear 5 PTEs before the stack  
 clearpteu(pgdir, (char\*)(stack\_bottom - ((PAGES + i) \* PGSIZE)));  
}  
  
sp = stack\_bottom;

## 4. Step 3: Addition of stack\_bottom Field

Added a new field stack\_bottom in the proc struct to store the bottom of the stack. The value of the bottom of the stack (stack\_bottom) was assigned after setting up the stack.

p->sz = PGSIZE + 0x3000;  
p->stack\_bottom = 0;  
p->tf->esp = 0x3000 + PGSIZE;  
p->tf->eip = 0x3000;

## 5. Step 4: Makefile Section Update

Updated the initcode section by changing the starting text address to 0x3000 in the Makefile. Also, changed the starting text address for the kernel to 0x3000 and added new user programs (\_null\_pointer and \_stack\_test) to the UPROGS list.

## 6. Step 5: Constant USERTOP Addition

Added a new constant USERTOP with the value 0xDD4E000 in memlayout.h.

#define USERTOP 0xDD4E000

## 7. Step 6: Null Pointer Dereference Test Program

Created a new user-level program (null\_pointer.c) to test null pointer dereference. The program attempts to dereference a null pointer and prints a message.

#include "types.h"  
#include "user.h"  
  
int main(int argc, char \*argv[])  
{  
 int \*p = 0;  
 printf(1, "null pointer dereference test: %d\n", \*p);  
 exit();  
}

## 8. Step 7: Modifications in proc.c and proc.h

### Changes in proc.c:

1. **userinit function:** Modified the initialization of p->sz to include an offset (0x3000) and added the initialization of p->stack\_bottom to zero.

p->sz = PGSIZE + 0x3000;  
p->stack\_bottom = 0;  
p->tf->esp = 0x3000 + PGSIZE;  
p->tf->eip = 0x3000;

1. **fork function:** Added a new condition to check if the parent process has a valid stack\_bottom. If so, it uses the copystack function to copy the stack to the child process.

### Changes in proc.h:

Added a new field stack\_bottom to the struct proc.

uint stack\_bottom; // Bottom of stack

## 9. Step 8: Changes in Other Files

### 9.1. stack\_test.c

This file defines a simple program with a global variable and a recursive function decrement. The function prints the address of local variables, decrements the global variable, and calls itself recursively.

#include "types.h"  
#include "user.h"  
  
int global\_var = 4000;  
  
void decrement() {  
 int a,b,c;  
 a = 1;  
 b = 3;  
 c = 1;  
 // print the address of the local variables  
 printf(1, "Address of a: %p\n", &a);  
 if (global\_var < 0) {  
 return;  
 }  
 global\_var = global\_var - (b - (a+c));  
 return decrement();  
}  
  
int main(int argc, char \*argv[]) {  
 decrement();  
 exit();  
}

### 9.2. syscall.c

Changes have been made to the functions fetchint, fetchstr and argptr. The modifications involve replacing the process’s size (curproc->sz) with a constant value USERTOP.

#### fetchint function:

if(addr >= USERTOP || addr+4 > USERTOP)

* **Explanation**: The boundary check for the address has been changed from curproc->sz to USERTOP. USERTOP seems to represent the top of the user space.

#### fetchstr function:

if(addr >= USERTOP)

* **Explanation**: Similar to fetchint, the boundary check has been changed to use USERTOP instead of curproc->sz.

#### argptr function:

if(size < 0 || (uint)i >= USERTOP || (uint)i+size > USERTOP)

* **Explanation**: The boundary check for the pointer has been modified to use USERTOP instead of curproc->sz.

### 9.3. trap.c

Added a condition after the default case in the switch statement. If the incStack() function returns 0 (indicating a stack overflow), it breaks out of the switch statement. Otherwise, it continues with the default behavior.

if (incStack() == 0)  
 break;

## 10. Step 9: copystack Function

This function copies the user stack (from stack\_bottom to USERTOP) to a new location specified by d (a new page directory). It allocates a new physical page (mem) in the kernel, copies the content from the old location to the new one, and then maps the new page into the new page directory (d). If any allocation or mapping fails, it jumps to the bad label and frees the allocated memory (freevm).

pde\_t\* copystack(pde\_t \*pgdir, pde\_t\* d, uint stack\_bottom)  
{  
 // ...  
 if(stack\_bottom == 0)  
 return 0;  
  
 for(i = stack\_bottom; i < USERTOP; i += PGSIZE){  
 // ...  
 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) {  
 kfree(mem);  
 goto bad;  
 }  
 }  
 return d;  
bad:  
 freevm(d);  
 return 0;  
}

## 11. Step 10: incStack Function

This function increments the stack of the current process. It checks if the stack pointer (sp) is within bounds, whether the page already exists, whether

the page is present, allocates a new page if needed, and updates the process’s stack\_bottom.

int incStack(void)  
{  
  
 if (sp > (stack\_bottom + PGSIZE))  
 {  
 cprintf("sp out of bound");  
 return -1;  
 }  
 if((pte = walkpgdir(pgdir, (void \*) new\_bottom, 1)) == 0)  
 {  
 cprintf("Page already exists\n");  
 return -1;  
 }  
 if(\*pte & PTE\_P)  
 return -1;  
  
 if(allocuvm(pgdir, new\_bottom, stack\_bottom) == 0){  
 cprintf("allocuvm failed\n");  
 return -1;  
 }  
 stack\_bottom = stack\_bottom - PGSIZE;  
 pte = walkpgdir(pgdir, (char \*)(stack\_bottom + PGSIZE), 0);  
 \*pte |= PTE\_U;  
 clearpteu(pgdir, (char \*)stack\_bottom );  
 myproc()->stack\_bottom = stack\_bottom;  
 cprintf("incStack: stack\_bottom = %d\n", stack\_bottom);  
 return 0;  
}

## 12. Step 11: Rubrics

### Part A

#### a) Null Pointer Dereference in Linux and xv6

In both Linux and xv6, dereferencing a null pointer leads to a segmentation fault. A null pointer is a pointer that does not point to any valid memory location. When the program attempts to access or modify the memory at address 0 (null pointer), the operating system detects this invalid memory access and triggers a segmentation fault. The segmentation fault is a protective mechanism to prevent unauthorized access to memory, ensuring the stability and security of the system.

#### b) Dereferencing 3 Pages

During the phase of dereferencing 3 pages in xv6, basic functionalities such as cat, mkdir, and simple hello world programs should work. This implies that the modification made to the initial memory size (sz) and the allocation of six pages for the stack do not interfere with the fundamental operations of the operating system. The changes introduced maintain the integrity of the system’s core functionalities.

#### c) Running Older Codes on New xv6

Running older codes on the new xv6 involves testing programs or code snippets that were developed for the original version of xv6 with the modifications made in Part A and Part B. The findings from this testing phase would provide insights into the backward compatibility of the modified xv6 kernel. Any discrepancies, errors, or unexpected behavior in running older codes would need to be documented.

### Part B

#### a) Stack Rearrangement

##### a. a) 3 Pages are Unmapped, but the Stack is Fixed

With three pages unmapped but the stack fixed, it means that a region in the user address space is intentionally left unallocated. This can have implications for memory usage efficiency and security. The fixed stack ensures that a specific portion of the address space is reserved for the stack, providing a predictable and controlled environment for stack operations.

##### a. b) Stack and Heap Grows, but 3 Pages are Not Unmapped

When the stack and heap grow but three pages are not unmapped, it suggests that additional memory is allocated for the stack and heap without interfering with the initially unmapped pages. This design choice allows for flexibility in memory management while maintaining the presence of the three unmapped pages.

#### b) Running Older Code with this New xv6 (Part A + Part B)

Running older code on the new xv6, which includes the modifications from Part A and Part B, tests the overall compatibility and robustness of the modified kernel. It ensures that the introduced changes do not negatively impact the execution of existing programs and that the system can still handle older code seamlessly.

## 13. Screenshots

All the required screenshots are included in the screenshots folder.