Implemented all assignments again in XV6-PUBLIC in DOCKER (in Windows 11)

Code logic, Output and Analysis:

Part A: Point Deference

null.c in linux (checked in MAC M1)

P is treated as an address and stored 100 into the address 0x0. Segmentation fault - core dumped is displayed. Null pointer deference as fault, OS is setting valid bit to 0 for this page table entry so an exception is raised and segmentation fault is thrown as I tried to access an invalid page. null deference can be added for debugging help.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int *p;
    p = 0;
    printf("%p\n",p);
    return 0;
}
```

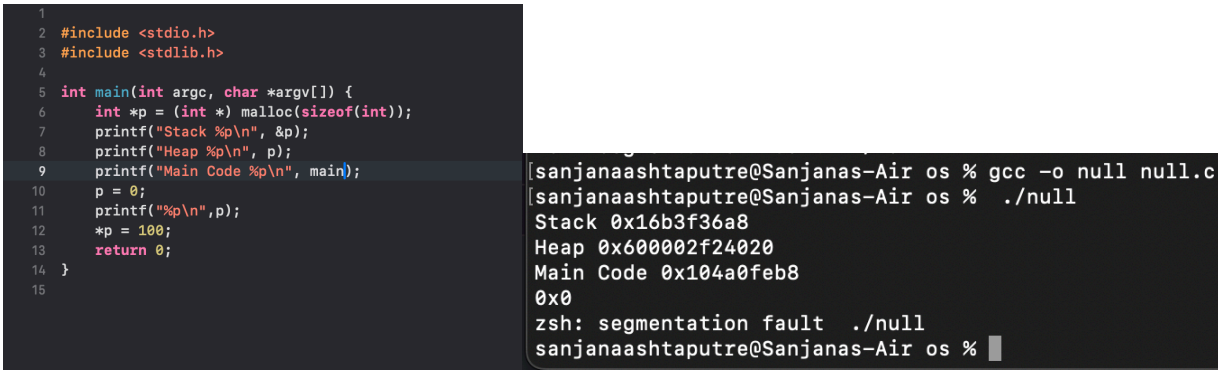
```
[sanjanaashtaputre@Sanjanas-Air os % gcc -o null null.c
[sanjanaashtaputre@Sanjanas-Air os %  ./null
0x0
sanjanaashtaputre@Sanjanas-Air os %
```

Stack allocated - 64 bit address space. Here address of main is treated like a pointer to a function. A value is displayed where main gets loaded into the virtual address space. xv6 address space: fills pages with code.. starts at VPN 0. After code is done, only one page stack, and a heap after that grows downwards to max of the address space. When the code is written to print what's at the virtual address 0, whatever the bits are for the instructions are displayed. First 4 bytes: entering main and setting up base n stack pointer. Instead of getting a debugging help, the system just prints out a value.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int *p;
    p = 0;
    printf("%p\n",p);
    *p = 100;
    return 0;
}
```

```
[sanjanaashtaputre@Sanjanas-Air os % gcc -o null null.c
[sanjanaashtaputre@Sanjanas-Air os %  ./null
0x0
zsh: segmentation fault  ./null
sanjanaashtaputre@Sanjanas-Air os %
```



```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int *p = (int *) malloc(sizeof(int));
    printf("Stack %p\n", &p);
    printf("Heap %p\n", p);
    printf("Main Code %p\n", main);
    p = 0;
    printf("%p\n",p);
    *p = 100;
    return 0;
}
```

```
[sanjanaashtaputre@Sanjanas-Air os % gcc -o null null.c
[sanjanaashtaputre@Sanjanas-Air os %  ./null
Stack 0x16b3f36a8
Heap 0x600002f24020
Main Code 0x104a0feb8
0x0
zsh: segmentation fault  ./null
sanjanaashtaputre@Sanjanas-Air os %
```

test_null.c in XV6 (XV6-public in docker on windows)

exec.c: It starts reading from the file that has the binary that's in elf format, if header doesn't look right then it goes to bad. Same thing for other conditions too. exec( ) creates a whole new address space and populate it with the program about to run with new stack and heap. It asks for page directory and creates a new page directory and starts populating page table and memory. It freezes the existing table. It creates new page table because if it fails and we might not be able to return to the address space to handle it so we need to keep a copy. It then loads the code into memory and map it into address space. It starts with size 0 which is the virtual address where it gets loaded into address space.

Uvmalloc is used to allocate space for new page table and loaded. Uvmalloc: Allocate PTEs and physical memory to grow process from oldsz to newsz, which need not be page aligned. Returns new size or 0 on error. It is basically making region of virtual space valid by going through the specified old and new size, allocate pages, set them to 0 with memset( ) for data protection, no overwriting on a page. Map pages makes the page table accessible and writable. It basically maps, updates the page table in a way that says virtual address 'a' maps to physical address mem and is writable part of our address space. So the sz is initialized to 0x3000. If we print the sz, vaddr and memsz, 2 line values are printed. Both start at virtual address 0 and each has some size. They are printed twice, the first is init, and the code size is 2512 bytes here. So, its going to map one page into the address space, fill in these many bytes into it from the binary. It calls fork and then exec to start the shell 'sh'.

Output before implementing null dereference.

```
pid : 3, 0 2312
Stack 0x0000000000002FC8
Heap 0x0000000000012FF0
Main Code 0x0000000000000000
0x0000000000000000
```

In Makefile By default location where the program memory will be loaded Is set to virtual address 0.

The code goes through and sets up this new page table and maps the code and sets up a stack and heap that grows downwards. Everything else in the address space that doesn't get mapped in here is by default not valid. So, the page address at 0, as long as its not loaded with anything, by default will not be valid.

```
$ test_null
copyuvm: sz = 20480
sz is 14844 before stack creation and 16384 is it's address
sp is 232054752 after executionincStack: sp = 232054752,  232030208 is the stack botton, 232026112 is the new bottom
sp out of boundpid 14 test_null: trap 14 err 4 on cpu 0 eip 0x3000 addr 0x0--kill proc
$
```

Null pointer dereference i,e. When a program tried to access memory at address 0x0. In linux, it throws segmentation fault which indicates a bug. SIGSEGV is triggered to terminate the program and core dump file is generated. We can check bug using the core dump to analyze the bug. In XV6, it resulted a trap to the kernel and the process is terminated with no core dump or other debugging information. If a null pointer is dereferenced in uniq head and ps code, it will cause a trap and terminate the process when trying to access address 0x0. In summary, resizing memory and stacks did not interfere with basic xv6 commands, implying the changes were done in a way that maintained compatibility with existing functionality. The OS core still works correctly even with modified memory allocation. Modifications did not break compatibility with code written for the old xv6 system.

Part B: Stack Rearrangement

Rearranging the address in the following manner:

USERTOP = 640KB

stack (at end of address space; grows backwards)

... (gap >= 5 pages)

heap (grows towards the high-end of the address space)

code

(3 unmapped pages)
ADDR = 0x0

exec.c: 5 unallocated pages are left between stack and heap as support memory mapping segments. Six pages are allocated at the top of user address space. The end of the stack is also saved after the stack is set *USERTOP* - *PAGES* * *PGSIZE* in a new variable.

In memlayout.h, a USERTO*P* is set for the OS.

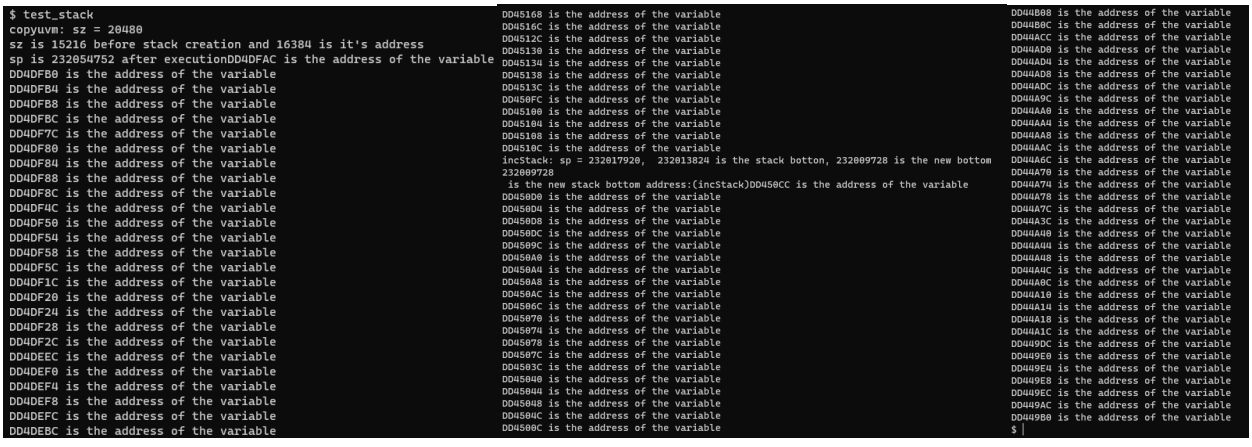
In proc.c, changes are made to userinit( ) and fork( ). The end of the stack is initialized to 0 and the process state is copied from parent into the child process if it is valid.

In trap.c, the process should be killed, when there is a page fault. A condition is added for the same.

In syscall.c, the threshold is checked as the address should not cross the USERTOP. The changes are made in argptr( ), fetchstr( ) and fetchint( ).

In vm.c, two new functions are created to check if new pages need to be added in order to build the stack if it is within the limit and to copy the child process memory. New memory is allocated and the old content is moved to the new location. If there are any unmapped pages, then the memory is set free. It works for both FCFS and priority based scheduler.

test_stack.c is the file used to test the stack rearrangement.



In summary, the unallocation decreases total use but having a guaranteed stack area adds control and stability. The combination allows customized memory use while still securing stack activities. Growing the stack or heap independently retains the unmapped pages, providing flexibility to tune memory usage except for those set unallocations. The separation enables customized growth without affecting the predefined unmapped area. Running old programs validates that the changes do not break support for existing software.

```
$ uniq uniqfile.txt
copyuvm: sz = 20480
sz is 15836 before stack creation and 16384 is it's address
sp is 232054736 after executionUniq command is being executed in user mode
I understand the operating system.
I love to work on OS.
Thanks xv6.
$
```

```
$ head headfile.txt
copyuvm: sz = 20480
sz is 15500 before stack creation and 16384 is it's address
sp is 232054736 after executionHead command is getting executed in user mode
This is sentence 1
This is sentence 2
This is sentence 3
This is sentence 4
This is sentence 5
This is sentence 6
This is sentence 7
This is sentence 8
This is sentence 9
This is sentence 10
This is sentence 11
This is sentence 12
This is sentence 13
This is sentence 14
```

```
$ head -n 5 headfile.txt
copyuvm: sz = 20480
sz is 15500 before stack creation and 16384 is it's address
sp is 232054720 after executionHead command is getting executed in user mode
This is sentence 1
This is sentence 2
This is sentence 3
This is sentence 4
This is sentence 5
$
```

```
$ ps
copyuvm: sz = 20480
sz is 14852 before stack creation and 16384 is it's address
sp is 232054760 after executionProcess ID         Process State           Process Name    Creation Time           End Time        Total Time
1               SLEEPING                init            0                       127417                          127417
2               SLEEPING                sh              7                       127417                          127410
13              RUNNING                 ps              127414                          127417                          3
$
```