PartA: Null pointer dereference

When we try to dereference a null pointer, it gives an exception, usually segmentation fault.
For example, this line of code:

```
Int *p = 0;
```

To access virtual address 0, it will try to translate into physical address, so it will look up page table. In page table, valid bit says whether that page is accessible or not. Page 0 is not valid. So going to translate it into physical address gives segmentation fault.

While in xv6, it does not give segmentation fault. It prints out a hex address instead, like this “83e58955” . To examine the object file with the dereferencing program with the following command,

```
objdump -d testnull.o
```

```
Code segment :   0x34 : 0 55
                        1 89 25
                        2 83 e4 f0
```

We can see that it is the first few bytes of code segment. Since there is no protection , we can read those bytes at virtual address 0, physical address 0x34.
If we don’t have guard page on first page or first page set to null, that is what will happen.

So the goal is to set virtual address 0 to invalid. We need to change so code is loaded at some other address like 0x1000.
Two files needed to be change to reflect the segmentation fault.
In the **makefile** I change the entry point of the text section of code segment for user program to 0x1000 from 0x0

```
$(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o $@ $^
```

Also in the forktest target

```
$(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o _forktest forktest.o ulib.o usys.o
```

In **exec.c,** we allocate the pae 0 full of zeroes.

```
if((sz = allocuvm(pgdir, sz, sz + PGSIZE)) == 0)
        goto bad;
  clearpteu(pgdir, (char*)(sz - PGSIZE));
```

clearpteu creates a guard page beneath the user stack and makes it invalid.

Testing (nulltest)
When executing the test code , it tries to dereference a null pointer and it catches the segmentation fault as trap no. 14 and prints out an error message **"Segmentation fault at address %x\n",rcr2())**

PartB:Shared pages

a) <u>Adding two new system calls</u>

```
void*        shmem_access(int page_number);
int      shmem_count(int page_number);
```

Made necessary changes in user.h, usys.S, syscall.c, syscall.h and sysproc.c for the definition of system call. System calls retrieves the argument from user space passed on to kernel space.

During boot of process in main.c, a function initializes 4 virtual addresses that will be mapped to 4 physical pages later. The addresses are stored in a global array and another global array keeps track of for each page how many processes mapped this particular page.

A process needs to keep track of how many virtual pages it has access to and the total count of pages. So, the structure for process ,struct proc , needs two more fields to hold those values.

shmem_access(int page_number) function acquires a reference to current process and check if current process has already access to the given page. If it has access , it returns the address. Otherwise it has to check the address space is full or not before getting a new virtual address.

The first available address that can be allocated should be at the end of user address space, which is defined as KERNBASE. So from there we keep allocating shared pages and keep checking so growing heap doesn't collide with the shared pages (sz is the end of heap).

The shared pages also need to have the permission PTE_W bit set so that other processes can write to the page as well.

If the virtual to physical mapping is successful , store the va in process structure , increment global counter for that particular page number and return the virtual address.

shmem_count(int page_number) returns the value of global counter for that page_number. It can be used to test if fork() works or not for shared pages.

b)<u>Allocate and free shared memory</u>

Before allocating a new shared page, the size of process needs to be checked against the bound to return the leftover space after subtracting the already allocated space from KERNBASE.

Instead of freeing the page table and all the physical memory pages between 0 and KERNBASE, we free only the pages that are not shared pages.

c) Fork the process to create the child process

When fork() is called , child process inherit shared pages from parent, so global shared memory counter should increment for those pages. When child process exit but it is still entered on the process table, it becomes a zombie.The parent uses wait() to keep time when to remove the zombie child entry, and also decrement the shared memory counter.

So the basic idea is to copy shared pages from parent process to child process.

d) Exec to create a new process

Exec code when creating a new process should not free shared pages from old page directory. When creating the new process, the shared memory address array and count of pages also need to be initialized again.

e) Bad arguments in system call

When we are trying to fetch the system call argument, int, string, pointer, we need to fetch it from some address at user address space of the current process. This address needs to be checked against the lower and upper bound of available space between 0 and KERNBASE , counting the pages already allocated.

Testing (shmemtest)

1.Test cases to check allocating shared memory for page number 0 to 3, and write some value into the shared memory and read it.
2.Test case to check for trying to read from already allocated shared page and change the value in the shared page.
3. Test case to check if we can access an invalid shared page or not.
4.Test case to check shemem_count, to see for each valid page how many process currently access it(before fork).
5. Test case to check after fork, if child can access shared page inherited from parent and read from it.
6. Test case to check after fork, if the child process can change the value in shared page and parent , after calling wait(), should be able to read the value that child changed.
7. Test case to check if shmem_count print the correct value inside the child and parent process.
8.Test case to check if exec is successful or not and we can pass argument in the program exec is calling.