Project #3

1. Overview

In this project, you'll be adding two new features to xv6 virtual memory system; null pointer dereferences and shared memory.

1.1. Null pointer dereferences

In xv6, the virtual memory (VM) uses a simple two-level page table. A user code is loaded into the very first part of the address space. Thus, if you dereference a null pointer, you will not see an exception (as you might expect segmentation fault); rather, you will see the first bit of code in the program that is running.

Look at how `exec()` works to understand how address spaces get filled with code and initialized. Also, check `fork()` to understand how a child process is created by copying the address space of the parent. What needs to change in there?

The rest of your task will be completed by looking through the code to figure out where there are checks or assumptions made about the address space. Think about what happens when you pass a parameter into the kernel, for example; if passing a pointer, the kernel needs to be very careful with it; to ensure you haven't passed it a bad pointer. How does it do this now? Does this code need to change in order to work in your new version of xv6?

One last hint: you will have to look at the `Makefile`s of xv6 as well. It defines how user programs are compiled and also sets their entry point (where the first instruction is) as zero. If you change xv6 to make the first page invalid, the entry point will clearly have to be somewhere else (e.g., the next page, or 0x1000). Thus, something in the `Makefile` will need to change to reflect this as well.

When the null pointer dereferences happen, xv6 should trap and kill the process. The good news: this will happen without too much trouble on your part if you do the project in a sensible way, because xv6 already catches illegal memory accesses.

1.2. Shared Memory

You will need to implement two system call:

- `void *shmem_access(int page_number)`

  This call makes a shared page available to the process calling it. The `page_number` argument can range from 0 through 3, and allows for four different pages to be used in this manner (identify which page to be mapped instead of handle or name in conventional OSs).

  When a process calls this function, the OS should map the physical page into the virtual address space of the caller, starting at the very high end of the address space. This function should return the virtual address of that page to the caller.

  Although synchronization is important when communicating through shared memory, which is not your worry for this project.

- `int shmem_count(int page_number)`

  This call returns how many processes currently are sharing that pages.

Some hints:

- Failure cases: Bad arguments or address space already fully in use.
- How to handle `fork()`: Child process also can access the shared page, and reference count should be updated correctly.
- How to track reference counts of each page.

1.3. Hints

You need to modify/add codes mostly in the following files:

- `include/syscall.h`
- `kernel/defs.h`
- `kernel/exec.c`
- `kernel/makefile.mk`
- `kernel/proc.c`
- `kernel/syscall.c`

- `kernel/sysfunc.h`
- `kernel/sysproc.c`
- `kernel/vm.c`
- `user/makefile.mk`
- `user/user.h`
- `user/usys.S`

2. Deliverable

Make null pointer dereferences work. Create the following two system calls in xv6 kernel.

- `shmem_access()`
- `shmem_count()`

3. Grading Criteria

We will use 22 scripts for grading. You can check the descriptions below.

`null (10):` null pointer dereference should kill the process

`null2 (10):` any dereference from first page should fault

`bounds (10):` return value of syscall should be -1 when invalid arguments are given (e.g., null page)

`shmem_access_invalid_input (10):` return value of shmem_access should be NULL when invalid arguments are given

`shmem_access_return_value (10):` return value of shmem_access needs to be one of 4 pages of virtual address space; first access needs to return last virtual page, second access needs to return second to last virtual page, etc.

`shmem_access_double_call (10):` when shmem_access is called twice with the same page number and same physical page, it should not be mapped into two different places in the virtual address space

`shmem_access_double_call_fork (10):` shmem_access called once in parent process. It is then called again after fork in child process. The address received should be the same address the parent process received

`shmem_access_read_write (10):` shared pages should be readable and writable

`shmem_access_communication (10):` a process should be able to read shared pages previously written by the other process

`shmem_access_full_address_space (10):` shmem_access should fail if the entire address space is full

`shmem_access_full_address_space2 (10):` sbrk should not allocate when the entire address space is full

`shmem_access_persistent (10):` pages should not be deallocated when no one references them

`shmem_access_exec (10):` a new exec-ed program should not have access to shared pages when it begins

`shmem_access_exec2 (10):` a new exec-ed program should reallocate virtual

pages if shmem_access is called

shmem_access_syscall_args (10): you have to make sure that pointers from shared pages can be successfully passed to system calls

shmem_count_invalid_input (10): shmem_count should return -1 when it received invalid input

shmem_count_zero (10): shmem_count should return 0 if no one is referencing the page

shmem_count_zero2 (10): shmem_count should return 0 after child process accesses and exits

shmem_count_one (10): shmem_count should return 1 after 1 access

shmem_count_one2 (10): shmem_count should return 1 after 2 accesses to the same page

shmem_count_fork (10): shmem_count should return 2 after fork if the parent process has accessed a shared page

shmem_count_exec (10): Exec should decrement count for shared pages: Parent accesses a shared page, then it forks. Child performs exec to start a new program. In the new program, the count for the shared page should be 1

After finishing your implementation, compress your xv6 source codes using the commands below. Then, submit the compressed file on KLMS.

```
#First, move to a directory containing the xv6 directory
mv <DIRECTORY_CONTAINING_THE_XV6_DIRECTORY>

#Then, compress your xv6 directory
tar  -czvf  <Student ID>_proj3.tar.gz  xv6
```