# Project 3b: The Null Pointer and Shared Memory

## Only section 1 does this part!

## Objectives

There are two objectives to this assignment:

- To familiarize you with the xv6 virtual memory system.
- To add a few new VM features to xv6 that are common in modern OSes.

## Overview

In this project, you'll be changing xv6 to support a feature virtually every modern OS does: causing an exception to occur when your program dereferences a null pointer. Sound simple? Well, it mostly is. But there are a few details.

You'll also be adding a new simple facility to allow different processes to share a memory page. Also sound simple? Good! At least you think things sound simple.

## Details

### Part A: Null-pointer Dereference

In xv6, the VM system uses a simple two-level page table as discussed in class. As it currently is structured, 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); rather, you will see whatever code is the first bit of code in the program that is running. Try it and see!

Thus, the first thing you might do is create a program that dereferences a null pointer. It is simple! See if you can do it. Then run it on Linux as well as xv6, to see the difference.

Your job here will be to figure out how xv6 sets up a page table. Thus, once again, this project is mostly about understanding the code, and not writing very much. Look at how `exec()` works to better understand how address spaces get filled with code and in general initialized. That will get you most of the way.

You should also look at `fork()` , in particular the part where the address space of the child 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'll have to look at the xv6 makefile as well. In there user programs are compiled so as to set their entry point (where the first instruction is) to 0. If you change xv6 to make the first page invalid, clearly the entry point will 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.

You should be able to demonstrate what happens when user code tries to access a null pointer. When this happens, 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.

## Part B: Shared Pages

In most operating systems, there are some different ways for processes to communicate with one another. In this part of the project, you'll explore how to add a shared-memory page to processes that are interested in communicating through memory.

The basic process will be simple: there is a new system call you must create, called **void *shmem_access(int page_number),** which should make 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.

When a process calls **shmem_access(0),** the OS should map a physical page into the virtual address space of the caller, starting at the very high end of the address space. The call then returns the virtual address of that page to the caller, so it can read/write it. If another process then calls **shmem_access(0),** it should also get the page mapped into its virtual address space (possibly at a different virtual address), and also be able to read/write it. In this way, the processes can communicate, by reading/writing shared memory. Of course, to use such memory carefully, one has to think about synchronization, but that is not your worry (for this project).

One other call is needed: **int shmem_count(int page_number).** This call tells you, for a particular shared page, how many processes currently are sharing that page.

Some things to think about:

- Failure cases: Bad argument to system call, address space already fully in use (large heap).
- How to handle fork(): Upon fork, must also make sure child process has access to shared page, and that reference counts are updated appropriately.
- How to track reference counts to each page so as to be able to implement shmem_count().

# The Code

The source code for xv6 (and associated README) can be found in **~cs537-1/ta/xv6/** . Everything you need to build and run and even debug the kernel is in there.

Might be good to read the xv6 book a bit: Here .

**Particularly useful for this project: Chapter 1 .**