EECS 343 Operating Systems -- Project 2 -- Virtual memory on xv6

Project Description | Submission Instructions

PROJECT 2: VIRTUAL MEMORY ON XV6

IMPORTANT DATES

Out: Wednesday, October 5th, 2016.

~~**Due:** Friday, October 14th, 2016 (11:59 PM CDT).~~

Due: Tuesday, October 18th, 2016 (11:59 PM CDT).
Since we are extending the deadline, NO LATE SUBMISSIONS WILL BE ACCEPTED FOR THIS PROJECT.

Submission instructions: Click the "Submission Instructions" tab above.

RESOURCES

- Some useful GDB and QEMU commands: **https://pdos.csail.mit.edu/6.828/2016/labguide.html**
- xv6 textbook: **https://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf**
  (note that we are using a version of the code from 2012)
- Guide to xv6 codebase: **https://pdos.csail.mit.edu/6.828/2012/xv6/xv6-rev7.pdf**
- Hexadecimal converter: **https://www.mathsisfun.com/binary-decimal-hexadecimal-converter.html**

PROJECT OVERVIEW

In this project you will become familiar with the xv6 virtual memory system and work to add a few features that are common in modern OSes. The project is composed of two parts, which you must complete in order. This project must be completed in groups of 2-3 people.

Part 1 (40 points)
In part 1, you will change the virtual memory space of user processes so that the address `0x0` is invalid. Currently, `0x0` (`NULL`) is a valid address for an xv6 user process. You will change xv6 so that dereferencing address `0x0` causes a page fault.

Part 2 (60 points)
In part 2, you will implement shared memory in xv6. This will be built upon your part 1 code. Currently, each process in xv6 has the same virtual address space, but each process's virtual address space maps to disjoint sets of physical pages. This prevents processes from accidentally (or intentionally) corrupting each other’s memory. You will change xv6 so that processes can share a limited number of physical memory pages.

Extra credit (20 points)
There is an optimization that can be added to your shared memory implementation for extra credit.

EDUCATIONAL OBJECTIVES

This project aims to achieve several educational objectives:

- Understand how virtual memory is implemented. In particular, understand how virtual addresses are mapped to physical addresses using a page directory and page tables.

- Understand how virtual memory is used to prevent processes from corrupting each other's memory.
- Learn how to implement shared memory and why it is useful.
- Practice writing your own test cases.
- Continue to hone your skills as an OS developer.

THE CODE

You can either build upon the code that you submitted in Project 1, or you can start from a clean version of the **original xv6 code**.

PART 1: MAKING THE NULL POINTER INVALID

40 points

In your prior C programming experience, you probably became somewhat familiar with the **null pointer**. In particular, you know that dereferencing a pointer whose value is `0x0` (`NULL`) causes a segmentation fault. The null pointer serves as a convenient **sentinel value** because, in most operating systems, address `0x0` is reserved for system use, which means that user programs should not be accessing that address anyway.

However, in xv6, address `0x0` is a valid virtual address. A user process's address space goes from `0x0` to `0xA0000` (`USERTOP`). This means that you can write a C program that dereferences a null pointer and run it on xv6 without causing a segmentation fault. Give it a try! You will be able to use this program later to test your work.

Your Task

Your task is to modify how xv6 manages the user virtual address space so that `0x0` is no longer a valid address. In particular, if a user program tries to dereference a null pointer, xv6 should trap and kill the process. Luckily for you, xv6 does this automatically for invalid memory accesses.

Your modifications must not prevent xv6 from functioning normally. In other words, don't break anything.

Guidance and hints

1. Make sure you understand how xv6 uses a page directory and page tables to map a process's virtual memory to physical memory. In particular, understand what the different bits in a Page Directory Entry and Page Table Entry mean. Chapter 2 of the **xv6 textbook** is a useful reference. How does a Page Table Entry differ for a valid page compared to an invalid page?
2. Look at the `exec()` function to understand what memory-related tasks are involved in executing a new process. Make a list of the memory-related functions that are being called, look at how they are implemented and decide which ones will need to change.
3. Look at the `fork()` function. Again, figure out which memory-related functions are being called and which ones need to change.
4. Keep digging through the rest of the code to find other places that need to change. There are several places where the kernel code makes checks on addresses and pointers. These checks are often based on the implicit assumption that the process's address space begins at `0x0`. For example, when a user program passes an argument in a syscall, the kernel checks the validity of the argument. Some of these checks may need to change.
5. Have a look at `user/makefile.mk`. This file dictates that user programs are compiled so that their entry point (first instruction) is loaded at address `0x0`. Since you are changing xv6 so that the first page is invalid, the entry point for user programs will have to change accordingly. For example, you can use the first address of the second page: `0x1000`.
6. Be bold. In Project 1, you simply had to add code to xv6, so you didn't have to worry about breaking the existing functionality. In this project, you actually have to modify the provided code. There is no way

around it. You will probably break something. Don't let this scare you. You will be able to repair the damage you cause.

7. When you type `ls` in xv6, you may have noticed a user program called `usertests`. This program runs a series of tests to make sure your kernel is working properly. The problem is that the changes you are making to the xv6 kernel in this project will invalidate some of the assumptions that underly the tests in `usertests`. Here is a slightly modified version of usertests that relaxes the invalid assumptions: **usertests_2_1** (You're welcome.)

   If you prefer, you can download the file from the command line using:
   ```
   $ wget http://www.aqualab.cs.northwestern.edu/component/attachments/download/703
   -O usertests_2_1.c.tar.gz
   ```

PART 2: IMPLEMENT SHARED MEMORY

60 points

xv6 is currently structured to map each process's virtual address space to disjoint sets of physical pages. In other words, every process's memory is non-overlapping with every other process's memory.

In this project, you will allow processes to share up to four pages of memory with each other. Sharing memory allows processes to avoid making redundant copies of data, collaborate on a computation, and otherwise communicate with each other. The concept of shared memory falls within the more general topic of interprocess communication. You can read about both on Wikipedia:

- Shared memory: **https://en.wikipedia.org/wiki/Shared_memory**
- Interprocess communication: **https://en.wikipedia.org/wiki/Inter-process_communication**

Your Task

1. Create a new system call with the following signature:

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

   - The syscall MUST use this exact interface.
   - When a process calls this syscall, the kernel will make a shared page available to the process.
   - `page_number` can range from 0 to 3, allowing up to four different pages to be shared.
   - The four shared pages, if and when they are requested by a process, should be mapped to the highest four pages in the calling process's virtual address space.
   - The syscall will return the virtual address of the shared page. If a process calls this syscall twice with the same argument, the syscall should recognize that this process has already mapped this shared page and simply return the virtual address again.
   - This syscall will indicate failure by returning NULL.

2. Create another syscall with the following signature:

   ```
   int shmem_count(int page_number)
   ```

   - The syscall MUST use this exact interface.
   - This syscall returns the number of processes that are currently sharing the shared page specified by the `page_number` argument.
   - This syscall will indicate failure by returning -1.

3. Other requirements:

- If a user passes a bad argument to one of these syscalls, the syscall will indicate failure with its return value.
- When a process forks, the child process should automatically get access to the parent's shared pages, if any. Reference counts should be updated accordingly.
- If a virtual memory page is being used as shared memory, it cannot also be used as "normal" unshared memory and vice versa.
- No memory leaks. For example, suppose several processes have been sharing a page, but now process #42 is the last process left alive that was using that page. When process #42 terminates, the shared page must be freed.

Guidance and hints

1. Let's look at an example. Suppose process #3 calls `shmem_access(0)` and that this page is not currently being shared by any other processes. In this case, the syscall must allocate a new physical memory page, which will become the shared page, and map a virtual memory page to the physical address of the new physical memory page. Now, if process #4 calls `shmem_access(0)`, the syscall will map a virtual memory page in process #4's address space to the physical address of the existing shared page. Now the two processes can both read and write to the same memory. The virtual addresses returned in the two processes may be the same or may be different--it's up to you. However, both virtual addresses must be within the top four pages of the virtual address space.
2. You will have to revisit many of the changes you made in Part 1.
3. You will have to count references to each shared page.
4. You already learned how to add new system calls in Project 1--follow the same approach.
5. We are providing a user program called **`sharedmem_simpletests`** with a few simple test cases. However, these tests are deliberately not comprehensive. You should add your own test cases to convince yourself that your code is working properly. Be creative while thinking of all cases where things can go wrong. You should be able to think of at least one test case for every bullet point in the **Your Task** section above.
   (`$ wget http://www.aqualab.cs.northwestern.edu/component/attachments/download/707 -O sharedmem_simpletests.c.tar.gz`)
6. In part 1, we provided you a modified version of `usertests`. However, in part 2 we are invalidating even more of the assumptions that underly those tests. Here is yet another modified version that you can use for part 2: **usertests_2_2**

   (`$ wget http://www.aqualab.cs.northwestern.edu/component/attachments/download/704 -O usertests_2_2.c.tar.gz`)

EXTRA CREDIT:

20 points

Here is an extra credit assignment for those seeking a further challenge. Only attempt this if you are done with Parts 1 and 2.

Your Task

Depending on how you completed Part 2, you may have chosen to set aside four pages at the top of every process's virtual address space to be used as needed for shared memory only. If the processes are not sharing any memory with each other, this approach is wasting four pages of memory that could otherwise be used by the process for normal unshared purposes. Now you will make these four pages available for general purpose use (like the rest of the process's memory) *or* shared use, depending on how the process needs them. In other words, if the process never calls `shmem_access`, but it does need all 160 pages of user memory for unshared purposes, that should be allowed. Likewise, if the process needs 3 pages of shared memory and 157 pages of unshared memory, that should be allowed too.

Guidance and hints

1. The four pages that may be used for shared memory need some bookkeeping. (i.e. You want to be able to tell whether a given page is a shared page or not.) There are many ways to do this. Where do you keep metadata about a page in memory? Can we add to that?

2. Given the new constraints (the initial 4 pages can be either shared or general memory), you want to make sure you don't break anything that's already working from Part 1 and Part 2. Make sure you make xv6 check this in all memory-related functions.

3. You will have to revisit many of the changes you made in Parts 1 and 2.

4. Things can break while trying to make this work. Using version control is highly recommended for the purpose of this. (i.e. You can't ask for extension because you had working versions of part 1 and 2, and things broke while you were attempting to do the extra credit.) If you use a version control service such as github, make sure your repo is private.

5. We will try to provide a few test cases. Any tests cases we provide are deliberately not comprehensive. Again, be creative while thinking of all cases where things can go wrong and write your test cases accordingly.
   UPDATE: The test cases are now posted here: **https://piazza.com/class/id32r0aahkj1fo?cid=138**

6. Disclaimers: The TAs will focus their attention on helping groups with parts 1 and 2. You will mostly be on your own for the extra credit. Also, it bears repeating, make sure you complete parts 1 and 2 before attempting the extra credit.

Acknowledgements: This project is based on a similar project used by UW Madison in their OS course.

Attachments:

sharedmem_simpletests.c.tar.gz
usertests_2_1.c.tar.gz
usertests_2_2.c.tar.gz