CS 537 Spring 2019, Project 3: xv6 Virtual Memory

Teaming up!

For this project, you have the option to work with a partner. Read more details in Submitting Your Implementation and Collaboration.

Objectives

- 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 few features virtually every modern OS has. The first is to generate an exception when your program dereferences a null pointer; the second is to rearrange the address space so as to place the stack at the high end; the last is to support sharing memory between different processes. Sounds simple? Well, it mostly is. But there are a few details.

Project 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 to 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, and then change it to leave the first **four** pages (0x0 - 0x4000) unmapped. The code segment should be starting at 0x4000.

With all necessary modifications made, xv6 should trap and kill any process that tries to access a null pointer.

Additional Specifications

- Kill the process when it is accessing any address in the first four 'null' pages, though this behavior will be changed to support shared memory in Part C.

Hints

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?

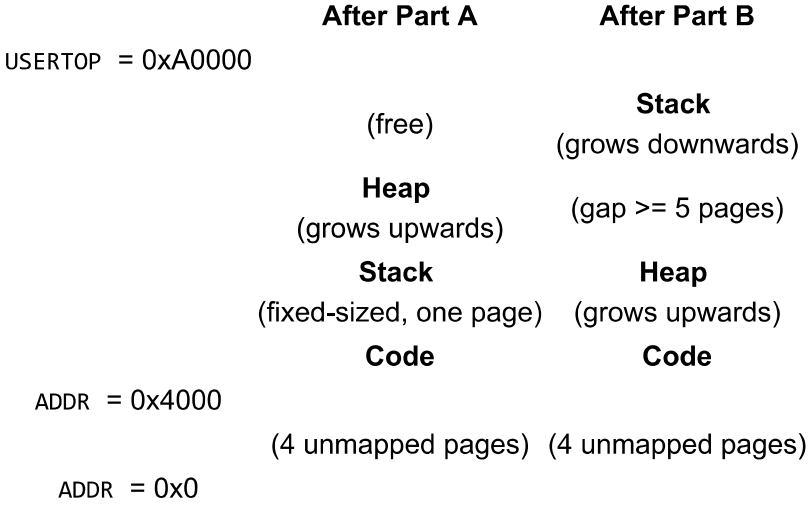
Remember that the first process is a little special and is constructed by the kernel directly? Take a look at `userinit()` and do not forget to update it too.

The rest of your task should focus on figuring out which parts of the code contain checks or assumptions on 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, which specifies the entry point (the memory location of the first instruction to execute) of all user programs. By default, the entry point is 0. If you change xv6 to make the first ~~two~~ **four** pages invalid, clearly the entry point will have to be the new beginning of the code segment. Thus, something in the makefile will need to change to reflect this as well.

Part B: Stack Rearrangement

In this part of the project, you'll rearrange the xv6 address space to make it more similar to a Unix process:



You should be wary of growing your heap so your stack will not be overwritten. In this project, you should always leave 5 unallocated (invalid) pages between the stack and heap.

The high end of the xv6 user address space is 640KB (see the `USERTOP` value defined in the xv6 code). Thus your first stack page should live at 636KB-640KB.

The one final challenging part of this project, is to automatically grow the stack backwards when needed. Doing so would require you to check if a page fault occurred on the page right below the stack bottom and then, instead of killing the offending process, you allocate a new page, map it into the address space, and continue to run.

Additional Specifications

- Stack can only grow one page at a time. If the page fault occurred more than one page below the current stack bottom, the process should be killed.

- The initial stack size should still be one page.
- You can assume the process will always have more than 5 pages of free memory after initialization.
- Don't worry about changing the memory structure of `userinit` in this part.

Hints

This will take a little more work on your part. First, you'll have to figure out where xv6 allocates and initializes the code, heap, and user stack; then, you'll have to figure out how to change the allocation so that the stack starts at the `USERTOP` of the xv6 user address space, instead of between the code and heap.

One thing you'll have to be very careful with is how xv6 currently tracks the size of a process's address space (with the `sz` field in the `proc` struct). There are a number of places in the code where this is used (e.g., to check whether an argument passed into the kernel is valid; to copy the address space). We recommend keeping this field to track the size of the code and heap, but use another field to track the stack. Accounting of this field should be added to all relevant code segments (i.e., where `sz` is modified).

Part C: 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 with each other through memory.

The basic process will be simple: there is a new system call you must create, called `void *shmget(int page_number)`, which should make a shared page available to the process calling it. The `page_number` argument can range from 0 through 2, and allows for three different pages to be used in this manner.

When a process calls `shmget(n)`, the OS should map the `n`-th shared physical page into the virtual address space of the caller, starting at the **second** page from the beginning 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 `shmget(n)`, it should also get the same 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).

For example, assuming the three shared physical pages start at 0xfe2000, 0xfe1000, and 0xfe0000 respectively, calling `shmget(0)` then `shmget(2)` will change the virtual address of the process to as follows

	After `shmget(0)`	**After `shmget(2)`**
`ADDR` = 0x4000 - …	Code, heap, stack, …	Code, heap, stack, …
`ADDR` = 0x3000 - 0x4000	(unmapped)	(unmapped)
`ADDR` = 0x2000 - 0x3000	(unmapped)	**mapped to physical page 0xfe0000**
`ADDR` = 0x1000 - 0x2000	**mapped to physical page 0xfe2000**	mapped to physical page 0xfe2000
`ADDR` = 0x0 - 0x1000	(unmapped)	(unmapped)

Additional Specifications

- Calling `shmget()` with an invalid page number should return a null pointer.
- Within a single process, multiple calls of `shmget()` with the same number should return the same virtual address.
- The first call of `shmget()` should always map the shared pages to the second page, the second call with a different number should map to the third page, and so on.

- Similar to Unix, the child process created by `fork()` should have the same shared pages as the parent, whereas the program executed through `exec()` should not inherit shared pages from the original process.
- You should still kill the process when it is accessing any of the unmapped regions, however, unlike in Part A, your system should now support reading/writing to the mapped shared pages.

Hints

By now, you should be very familiar with how to create a new system call. But before doing that, first you'll need some other function to reserve three physical pages when the system starts up, and store their addresses in a data structure that is available to every process. Also, you'll need to figure out how to map a physical address to a virtual address in the process' page table. A good staring point is to look at how other functions (such as `fork()` and `exec()`) handle this.

One tricky thing about this part is that the shared memory should be handled differently in some cases. For example, when a process is killed, normally all its memory should be freed; however, if some of the memory contains a shared page, then obviously we should not free this shared page. Similarly, you should be very careful about shared pages when `fork()` is called.

Code

We suggest that you start from the source code of xv6 at ~cs537-1/xv6-sp19 , instead of your own code from previous projects.

```
cp -r ~cs537-1/xv6-sp19 ./
```

Testing

It is a good habit to get basic functionalities working before moving to advanced features. You can run an individual test with the following command:

```
/u/c/s/cs537-1/tests/p3b/runtests testname
```

The following three sets of tests are incremental. Please proceed with testing with later set of tests only after your previous set of tests have passed.

- **Tests to use after Part A implementation:**
    1. Move the code segment and leave the first four pages unmapped: *null*, *null2*, *null3*, *null4*
- **Tests to use after Part B implementation:**
    1. Checks syscall arguments with new assumptions about the address space: *bounds*
    2. Move the stack to the top: *stack*, *bounds2*, *bounds_str*
    3. Heap does not grow into stack: *heap*
    4. Stack grows on page fault: *stack2*, *bounds3*, *stack3*
    5. Stack does not grow into heap: *stack4*, *stack5*
- **Tests to use after Part C implementation:**
    1. shmget() returns a null pointer with invalid pagenumber: *shmem_invalidpg*
    2. multiple calls of shmget() returns same address; checks syscall arguments with shared-memory pages: *shmem_bound*

3. read write access to the shared-memory pages: *shmem_rw*
4. address space of forked child process with shared-memory pages in parent process: *shmem_fork*

Due Date

March 11, 2019, 11.59pm

Submitting Your Implementation

Please read the following information carefully. **We use automated grading scripts, so please adhere to these instructions and formats if you want your project to be graded correctly.**

If you choose to work in pairs, only **one** of you needs to submit the source code. But, **both** of you need to submit an additional `partner.login` file, which contains **one** line that has the **CS login of your partner** (and nothing else). If there is no such file, we are assuming you are working alone. If both of you turn in your codes, we will just randomly choose one to grade.

To submit your solution, copy all of the xv6 files and directories with your changes into `~cs537-1/handin/<cs-login>/p3/` . One way to do this is to navigate to your solution's working directory and execute the following command:

```
cp -r . ~cs537-1/handin/$USER/p3/
cd ~cs537-1/handin/$USER/p3/ && make && make clean
```

Consider the following when you submit your project:

- If you use any slip days you have to create a `SLIP_DAYS` file in the `/<cs-login>/p3/` directory otherwise we use the submission on the due date.
- Your files should be directly copied to `~cs537-1/handin/<cs-login>/p3/` directory. Having subdirectories in `<cs-login>/p3/` like `<cs-login>/p3/xv6-sp19` is **not acceptable**.

Collaboration

This project is to be done in groups of size one or two (not three or more). Now, within a group, you can share as much as you like. However, copying code across groups is considered cheating.

If you are planning to use `git` or other version control systems (*which are highly recommended for this project*), just be careful **not** to put your codes in a public repository.