CS 537 - Introduction to Operating Systems - Fall 2017

Site /

Project3b

Project 3b: xv6 VM Layout

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.

Updates

Read these updates to keep up with any small fixes in the specification.

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. Sounds simple? Well, it mostly is. But there are a few details.

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 two pages (0x0 - 0x2000) unmapped. The code segment should be starting at 0x2000. 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?

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 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.

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

Part B: Stack Rearrangement

The xv6 address space (after modifications from Part A) is set up like this:

```
USERTOP = 640KB
(free)
heap (grows towards the high-end of the address space)
stack (fixed-sized, one page)
code
(2 unmapped pages)
ADDR = 0x0
```

In this part of the xv6 project, you'll rearrange the address space to make it more similar to what we've discussed in class:

```
USERTOP = 640KB
stack (at end of address space; grows towards 0x0)
... (gap >= 5 pages)
heap (grows towards USERTOP)
code
(2 unmapped pages)
ADDR = 0x0
```

This will take a little 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.

Some tricky parts: 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).

You should also 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. These are reserved for supporting memory mapping segment in the future.

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 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.

The Code

The source code for xv6 can be found in **~cs537-1/ta/xv6/** . Everything you need to build and run and even debug the kernel is in there. Start with a fresh kernel, instead of using your old one with

the MLFQ scheduler.

You may also find the following readings about xv6 useful: xv6 book

Particularly useful for this project: Anything about fork() and exec(), as well as virtual memory (chapter 2).

Test Along The Way

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:

```
shell% ~cs537-1/ta/tests/3b/runtests testname
```

1. Move the code segment and leave the first two pages unmapped: `null, null2`
2. Checks syscall arguments with new assumptions about the address space: `bounds`
3. Move the stack to the top: `stack, bounds2, bounds_str`
4. Heap does not grow into stack: `heap`
5. Stack grows on page fault: `stack2, bounds3, stack3`
6. Stack does not grow into heap: `stack4, stack5`

Page last modified on October 15, 2017, at 09:10 PM