Lab 3: Memory management

Handed out Friday, Feb. 14, 2020
Due Monday March 2, 2020

Objectives

- Modify memory layout to move stack to top of address space (70%)
- Implement stack growth (30%)

Preliminaries

You can get the starter code from the lab2 repository on github.

Part 1: Changing memory layout

Overview

In this part, you'll be making changes to the xv6 memory layout. Sound simple? Well, there are a few tricky details.

Details

In xv6, the VM system uses a simple two-level page table. If you do not remember the details, read Section 20.3 of OS 3 easy steps. However, you may find the description in Chapter 1 of the xv6 manual sufficient (and more relevant to the assignment).

The xv6 address space is currently set up like this:

```
code  lowest address
stack (fixed-sized, one page)
heap (grows towards the high-end of the address space)
```

In xv6, heap is not actually present. Stack is just 1 page. It starts 2 page sizes above the code+data block. 1 page size is for the stack itself and other page size is the guard page. Everything above stack and below kernbase is called "heap" in xv6. In this lab, we want to have multiple pages for stack.

In this part of the xv6 project, you'll rearrange the address space to look more like Linux:

```
code
heap (grows towards the high-end of the address space)
... (gap)
stack (at end of address space; grows backwards)
```

You can see the general map of the kernel memory in memlayout.h; the user memory starts at 0 and goes up to KERNBASE. **Note that we will not be changing the kernel memory layout at all, only the user memory layout**

Right now, the program memory map is determined by how we load the program into memory and set up the page table (so that they are pointing to the right physical pages). This is all implemented in exec.c as part of the exec system call using the underlying support provided to implement virtual memory in vm.c. To change the memory layout, you have to change the exec code to load the program and allocate the stack in the new way that we want.

Moving the stack up will give us space to allow it to grow, but it complicates a few things. For example, right now xv6 keeps track of the end of the virtual address space using one value (sz). Now you have to keep more information potentially e.g., the end of the bottom part of the user memory (i.e., the top of the heap, which is called brk in un*x), and bottom page of the stack.

Once you figure out in exec.c where xv6 allocates and initializes the user stack; then, you'll have to figure out how to change that to use a page at the high-end of the xv6 user address space, instead of one between the code and heap.

Some tricky parts: Let me re-emphasize: one thing you'll have to be very careful with is how xv6 currently tracks the size of a process's address space (currently 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 doing some other accounting to track the stack, and changing all relevant code (i.e., that used to deal with `sz`) to now work with your new accounting. Note that this potentially includes the shared memory code that you are writing for part 2.

Part 2: Growing the Stack

The final item, which is challenging: automatically growing the stack backwards when needed. Getting this to work will make you into a kernel boss, and also get you those last 10% of credit. Briefly, here is what you need to do. When the stack grows beyond its allocated page(s) it will cause a page fault because it is accessing an unmapped page. If you look in traps.h, this trap is T_PGFLT which is currently not handled in our trap handler in trap.c. This means that it goes to the default handling of unknown traps, and causes a kernel panic.

So, the first step is to add a case in trap to handle page faults. For now, your trap handler should simply check if the page fault was caused by an access to the page right under the current top of the stack. If this is the case, we allocate and map the page, and we are done. If the page fault is caused by a different address, we can go to the default handler and do a kernel panic like we did before.

Bonus (5%): Write code to try and get the stack to grow into the heap. Were you able to? If not explain why in detail showing the relevant code.

Hints

IMPORTANT Check the survival guide posted to walk you through this assignment.

Particularly useful for this project: Chapter 1 of xv6 + anything else about fork() and exec(), as well as virtual memory.

Take a look at the `exec` code in exec.c which loads a program into memory. It will be using VM functions from vm.c such as allocuvm (which uses mappages). These will be very instructive for implementing shm_open (in lab 4) -- we are allocating a new page the first time (similar to allocuvm) and adding it to the page table (similar to mappages).

It may be helpful to try to answer these questions to yourself:

- Read chapter 2 in the xv6 book. Briefly explain the operation of allocuvm() and mappages() and Figure 1-2. Check how exec uses them for an idea.
- Explain how you would given a virtual address figure out the physical address if the page is mapped otherwise return an error. In other words, how would you find the page table entry and check if its valid, and how would you use it to find the physical address.
- Find where in the code we can figure out the location of the stack.