Lab 2: Memory management

Handed out Wednesday, Nov. 8, 2017
Walkthrough due Wednesday Nov 22, 2017
Due Wednesday Dec 6, 2017

Lab 2:

Objectives

There two parts to this assignment. Each part is mostly independent although the change of the memory layout will affect where you attach pages in shared memory if you use curproc->sz to mark the end of the address space as is currently done in xv6. This and similar changes are described in the assignment below.

- Modify memory layout and enable stack growth (Part 1: 65% + 5% bonus)
- Implement Shared Memory (Part 2: 35% + 10% bonus)

Preliminaries

For this assignment we will use some starter code which is needed for part 2 (and explained there). You can get the starter code from the lab2 repository on github.

Part 1: Growing Stack

Overview

In this part, you'll be making changes to the xv6 memory layout, and then adding support to enable the stack to grow (the default implementation provides only one page for the stack). 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
stack (fixed-sized, one page)
heap (grows towards the high-end of the address space)
```

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.

You could also be wary of growing your heap and overwriting your stack. In fact, you should always leave an unallocated (invalid) page between the stack and heap, although you should not do this if you are attempting the bonus part.

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.

Part 2: Implement shared memory

In this part of the assignment you are implementing support to enable two processes to share a memory page. This is implemented by having an entry in both process page tables point to the same physical page.

Start by looking at the user program shm_cnt.c. In this program we fork a process, then both processes open a shared memory segment with the same id using:

```
shm_open(1,(char **)(&counter));
```

For this system call, the first parameter gives an id for the shared memory segment, and the pointer is used to return a pointer to the shared page. By having this pointer be of type `shm_cnt`, we can access this struct off of this pointer and we would be accessing the shared memory page.

The code then proceeds to have both processes go through a loop repeatedly incrementing the counter in the shared page (acquiring a user level spin lock to make sure we dont lose updates; test your program without the lock and see if it makes a difference). The uspinlock is implemented in the starter code in uspinlock.c and uspinlock.h -- take a look.

At the end, each process prints the value of the counter, closes the shared memory segment and exits using `shm_close(1)`. One of them should have a value of 20000 reflecting updates from both the processes. Check your code without the spinlock to see if you lose updates.

Your task is to implement shm_open and shm_close. They are already added as system calls; you should write your code in shm.c

shm_open looks through the shm_table to see if this segment id already exists. If it doesn't then it needs to allocate a page and map it, and store this information in the shm_table. Dont forget to grab the lock while you are working with the shm_table (why?). If the segment already exists, increase the refence count, and use mappages to add the mapping between the virtual address and the physical address. In either case, return the virtual address through the second parameter of the system call.

shm_close is simpler: it looks for the shared memory segment in shm_table. If it finds it it decrements the reference count. If it reaches zero, then it clears the shm_table. You do not need to free up the page since it is still mapped in the page table. Ok to leave it that way.

Hints

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 -- we are allocating a new page the first time (similar to allocuvm) and adding it to the page table (similar to mappages).

Submission(s)

Like Lab1, there are two submissions, a walkthrough/design document and the final submission. The walkthrough tentatively consists of the following questions (may be subject ot modifications until Nov 12). Where an outline of an implementation is requested, you have to list all the major items clearly:

- 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.
- Show where in the code we can figure out the location of the stack.
- Outline how you would implement shm_open for a segment that already exists