Lab Quiz 3

Goal

The goal of this assignment is to test your understanding of the xv6 operating system by building new features and functionality into the OS.

Note: This lab has two parts. For each part, start with a clean install of xv6 that is provided, make your changes to that code, and submit a patch that captures a diff of the code written in that part (alone) against the original code. Keep the patches for both your parts independent, so that we can grade both parts separately.

Part A: Displaying system and process statistics

In this part, we will implement system calls to get information about currently active processes, much like the `ps` and `top` commands in Linux do. Implement the following system calls in xv6:

- `getNumProc()` returns the total number of active processes in the system (either in running, runnable, sleeping, or zombie states).
- `getNumFreePages()` returns the total number of free pages in the free page list of the kernel.
- `getMaxPid()` returns the maximum PID amongst the PIDs of all active processes.
- `getProcInfo(pid, &p)` takes as arguments an integer PID and a pointer to a structure. This structure is an object in the program making the system call, and the system call returns information about the process with that PID by filling in various fields in this structrue. The information about the process that must be returned includes the parent PID, the number of times the process was context switched in by the scheduler, and the process size in bytes. Note that while some of this information is already available as part of the `struct proc` of a process, you may have to add new fields to keep track of some other extra information. If a process with that PID is not present, this system call must return -1 as the error code.

  Hint: As an example of how to pass a structure of information across system calls, you can see the code of the `ls` userspace program and the `fstat` system call in xv6. The `fstat` system call fills in a structure `struct stat` with information about a file, and this structure is fetched via the system call and printed out by the `ls` program. In a similar fashion, you can use the already defined `struct proc` or some other new structure that you declare in order to pass information. The `fstat` system call implementation also shows you how you can parse integer and pointer arguments given to the system call.

After implementing the above system calls, you will write a test program `proctest.c` that will showcase the functionality of your new system calls. Your test program must invoke each of the above system calls at least once and print out statistics about active processes. In order to print out information about all processes like `ps` does, you can obtain the maximum PID used via the appropriate system call, and repeatedly invoke `getProcInfo` for all PIDs starting from 1 until the maximum value, to get information about all the processes in the system. Your test program can also fork a few processes, and repeat this exercise, to verify that the information returned has been updated with the new process information.

Note that we will evaluate your code, both by running your test program above, as well by using our own test program that invokes the above system calls.

Part B: Lazy page allocation in xv6

Source and credits: Homework assignment of course 6.828, MIT

One of the many neat tricks an OS can play with page table hardware is lazy allocation of heap memory. Xv6 applications ask the kernel for heap memory using the sbrk() system call. For example, this system call is invoked when the shell program does a `malloc` to allocate memory for the various tokens in the shell command. In the xv6 kernel we have given you, sbrk() allocates physical memory and maps it into the process's virtual address space. However, there are programs that allocate memory but never use it, for example to implement large sparse arrays. Sophisticated kernels delay allocation of each page of memory until the application tries to use that page – as signaled by a page fault. You'll add this lazy allocation feature to xv6 in this exercise.

Part B Step 1: Eliminate allocation from sbrk()

Your first task is to delete page allocation from the sbrk(n) system call implementation, which is the function sys_sbrk() in sysproc.c. The sbrk(n) system call grows the process's memory size by n bytes, and then returns the start of the newly allocated region (i.e., the old size). Your new sbrk(n) should just increment the process's size (`proc->sz`) by n and return the old size. It should not allocate memory – so you should delete the call to growproc(). However, you should still increase `proc->sz` by n to trick the process into believing that it has the memory requested.

Make this modification to the code in sysproc.c, boot xv6, and type `echo hi` to the shell. You should see something like this:

```
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x12f1 addr 0x4004--kill proc
$
```

The "pid 3 sh: trap..." message is from the kernel trap handler in trap.c; it has caught a page fault (trap 14, or T_PGFLT), which the xv6 kernel does not know how to handle. Make sure you understand why this page fault occurs. The "addr 0x4004" indicates that the virtual address that caused the page fault is 0x4004.

Part B Step 2: Lazy allocation

Modify the code in `trap.c` to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing. That is, you must allocate a new memory page, add suitable page table entries, and return from the trap, so that the process can avoid the page fault the next time it runs. You should add your code just before the `cprintf` call that produced the "pid 3 sh: trap 14" message. Your code is not required to cover all corner cases and error situations; it just needs to be good enough to let the shell run simple commands like `echo` and `ls`.

Some helpful hints:

- Look at the cprintf arguments to see how to find the virtual address that caused the page fault.
- Steal code from allocuvm() in vm.c, which is what sbrk() calls (via growproc()).
- Use PGROUNDDOWN(va) to round the faulting virtual address down to a page boundary.
- Once you correctly handle the page fault, do `break` or `return` in order to avoid the `cprintf` and the `proc->killed = 1` statements.
- You will need to call mappages() from trap.c in order to map the newly allocated page. In order to do this, you'll need to delete the `static` in the declaration of mappages() in vm.c, and you'll need to declare mappages() in trap.c. Add this declaration to trap.c before any call to mappages():

  ```
  int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
  ```

- You can check whether a fault is a page fault by checking if `tf->trapno` is equal to T_PGFLT in trap().

If all goes well, your lazy allocation code should result in `echo hi` working. You should get at least one page fault (and thus lazy allocation) in the shell, and perhaps two.

To evaluate your submission for this part, we will check that your code lets the shell run simple commands. Then, we will check that you have indeed commented out the part that allocated memory in sbrk(), and that you have written the correct logic for page allocation in trap.c.

Submission and Grading

You must solve this lab individually, in the specified lab slot. No extensions will be allowed. Even if you haven't finished fully, please submit whatever you have when the time is up. Please create a tar gzip with your roll number as the filename and submit on Moodle. Your tarball must contain the following files.

- Your patches to xv6, including changes to the kernel code and test programs. Submit two separate patches for parts A and B, so that we can grade both parts independently.
- An optional readme.txt explaining how to patch your code, and any other information to aid evaluation.

To evaluate your submission, we will patch your code against the original xv6 code given to you (separately for each part), and test if you have implemented the functionality specified.