Lecture Notes on Operating Systems

Lab: Lazy Memory Allocation in xv6

Goal

The goal of this lab is to understand the memory management subsystem in xv6 by implementing lazy memory allocation.

Before you begin

- Download, install, and run the xv6 OS. You can use your regular desktop/laptop to run xv6; it runs on an x86 emulator called QEMU that emulates x86 hardware on your local machine.

- First, download the publicly available version of xv6, then copy the patch provided as part of this lab onto the original xv6 code.

- For this lab, you will need to understand the following files: defs.h, sysproc.c, trap.c, vm.c.

- Learn how to write your own test programs in xv6. We have provided a simple test program testcase.c as part of our patch. This test program is compiled by our patched Makefile and you can run it on the xv6 shell by typing ./testcase. You must be able to write other such test programs to test your code. Note that the xv6 OS itself does not have any text editor or compiler support, so you must write and compile the code in your host machine, and then run the executable in the xv6 QEMU emulator.

- The xv6 code patch provided to you implements a new system call called getNumPages. Read and understand this system call implementation from the modified files provided to you. This system call fills the locations pointed to by the arguments vp and pp with the number of physical pages and number of virtual pages respectively being used by the process, and returns the currently set process size. You will find this system call helpful in debugging your implementation.

Lazy Memory Allocation

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. In this lab, you will add support for this lazy allocation feature in xv6, in the following steps.

1. **Eliminate allocation from sbrk()**. Your first task is to delete page allocation from the `sbrk(n)` system call implementation, which is in the function `sys_sbrk()` in `sysproc.c`. The `sbrk(n)` system call grows the process 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 size 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.

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 can check whether a fault is a page fault by checking if `tf->trapno` is equal to `T_PGFLT` in `trap()`.
- If you think you need to call `mappages()` from `trap.c`, you will need to delete the `static` keyword in the declaration of `mappages()` in `vm.c`, and you will need to declare `mappages()` in `trap.c`.

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.
Testing your code

We have provided you with a simple testing/autograding framework to help you assess the correctness of your implementation. Untar and install the test code provided to you as part of this lab. Place your solution files in the folder `student_modified_files`. Next, run the script `generateOutput.sh`. This script patches the xv6 code to execute several testcases within the xv6 shell, and stores the output from these testcases in an output folder. The script `autograder.sh` can then be used to compare the expected output from these testcases (generated using our solution code) with your generated output, to verify the correctness of your code.

Note that the test cases provided by us are not exhaustive in any way, and you are encouraged to write your own testcases, beyond those provided as part of our scripts, to test your code.

Submission instructions

- For this lab, you will need to modify the following files: `defs.h, sysproc.c, trap.c, vm.c`
- Place all the files you modified in a directory, with directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Grading

We will use the autograding script (with possibly new testcases than those provided to you) to test the correctness of your code. We will also read your code to ensure that you have adhered to the problem specification in your implementation.