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

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 original xv6 kernel, `sbrk()` allocates physical memory and maps it into the processs 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()**. The `sbrk(n)` system call grows the processs memory size by n bytes, and then returns the start of the newly allocated region (i.e., the old size). 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`. We have provided a patched `sysproc.c` file as part of this lab; you may use this file for eliminating this memory allocation.Your new `sbrk(n)` will just increment the processs size by n and return the old size. It does not allocate memory, because the call to `growproc()` is commented out. However, it still increases `proc->sz` by n to trick the process into believing that it has the memory requested. With the patched 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. 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:

- You can check whether a fault is a page fault by checking if `tf->trapno` is equal to `T_PGFLT` in `trap()`.
- Look at the `cprintf` arguments to see how to find the virtual address that caused the page fault.
- Reuse code from `allocuvm()` in `vm.c`, which is what `sbrk()` calls (via `growproc()`).
- Use `PGROUNDDOWN(va)` to round the faulting virtual address down to the start of 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.
- 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`. An easier option would be to write a new function to handle page faults within `vm.c` itself, and call this new function from the page fault handling code in `trap.c`.

If all goes well, your lazy allocation code should result in `echo hi` working.

Submission instructions

- For this lab, you will need to modify the following files: `defs.h`, `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.