Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Memory management xv6

The goal of this lab is to understand memory management in xv6.

Before you begin

- Download, install, and run the original xv6 OS code. 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. In the xv6 folder, run `make`, followed by `make qemu` or `make-qemu-nox`, to boot xv6 and open a shell.
- We have modified some xv6 files for this lab, and these patched files are provided as part of this lab's code. Before you begin the lab, copy the patched files into the main xv6 code directory.
- For this lab, you will need to understand the following files: `syscall.c`, `syscall.h`, `sysproc.c`, `user.h`, `usys.S`, `vm.c`, `proc.c`, `trap.c`, `defs.h`, `mmu.h`.
  - The files `sysproc.c`, `syscall.c`, `syscall.h`, `user.h`, `usys.S` link user system calls to system call implementation code in the kernel.
  - `mmu.h` and `defs.h` are header files with various useful definitions.
  - The file `vm.c` contains most of the logic for memory management in the xv6 kernel, and `proc.c` contains process-related system call implementations.
  - The file `trap.c` contains trap handling code for all traps including page faults.
- Learn how to write your own user programs in xv6. For example, if you add a new system call, you may want to write a simple C program that calls the new system call. There are several user programs as part of the xv6 source code, from which you can learn. We have also provided a simple test program `testcase.c` as part of our patched code. This test program is compiled by our patched `Makefile` and you can run it on the xv6 shell by typing `testcase` at the command prompt. We have also provided several test programs to test the xv6 code you write in this lab. Feel free to test your code with these, as well as with other test cases you write. Remember that any test program you write should be included in the Makefile for it to be compiled and executed from the xv6 shell. 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.

Part A: Displaying memory information

You will first implement the following new system calls in xv6.

- `numvp()` should return the number of virtual/logical pages in the user part of the address space of the process, up to the program size stored in `struct proc`. You must count the stack guard page as well in your calculations.
- `numpp()` should return the number of physical pages in the user part of the address space of the process. You must count this number by walking the process page table, and counting the number of page table entries that have a valid physical address assigned.

Because xv6 does not use demand paging, you can expect the number of virtual and physical pages to be the same initially. However, the next part of the lab will change this property.

Part B: Memory mapping

In this part, you will implement a simple version of the `mmap` system call in xv6. Your `mmap` system call should take one argument: the number of bytes to add to the address space of the process. You may assume that the number of bytes is a positive number and is a multiple of page size. The system call should return a value of 0 if any invalid inputs are provided. If a valid number of bytes is provided as input, the system call should expand the virtual address space of the process by the specified number of bytes, and return the starting virtual address of the newly added memory region. The new virtual pages should be added at the end of the current program break, and should increase the program size correspondingly. However, the system call should NOT allocate any physical memory corresponding to the new virtual pages, as we will allocate memory on demand. You can use the system calls of the previous part to print these page counts to verify your implementation. After the `mmap` system call, and before any access to the mapped memory pages, you should only see the number of virtual pages of a process increase, but not the number of physical pages.

Physical memory for a memory-mapped virtual page should be allocated on demand, only when the page is accessed by the user. When the user accesses a memory-mapped page, a page fault will occur, and physical memory should only be allocated as part of the page fault handling. Further, if you memory mapped more than one page, physical memory should only be allocated for those pages that are accessed, and not for all pages in the memory-mapped region. Once again, use the virtual/physical page counts to verify that physical pages are allocated only on demand.

To solve this lab, you must handle the page fault trap in `trap.c` and add code to allocate memory on demand when a page fault occurs. You can check whether a trap is a page fault by checking if `tf->trapno` is equal to `T_PGFLT`. Look at the arguments to the `cprintf` statements in `trap.c` to figure out how one can find the virtual address that caused the page fault. 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 processing of other traps. Remember, it is important to call `switchuvm` to update the CR3 register and TLB every time you change the page table of the process. This update to the page table will enable the process to resume execution when you handle the page fault correctly. One final hint: 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`.

We have provied a simple test program to test your implementation. This program invokes `mmap` multiple times, and accesses the memory-mapped pages. It prints out virtual and physical page counts periodically, to let you check whether the page counts are being updated correctly. Please write more such test cases to thoroughly test your implementation.

Submission instructions

- For this lab, you may need to modify some subset of the following files: `syscall.c`, `syscall.h`, `sysproc.c`, `user.h`, `usys.S`, `vm.c`, `proc.c`, `trap.c`, `defs.h`. You may also write new test cases, and modify the `Makefile` to compile additional test cases.
- Place all the files you modified in a directory, with the 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.