Lecture Notes on Operating Systems

Lab: Memory management in xv6

In this lab, you will understand the memory management subsystem in xv6 by implementing two functionalities: a lazy heap memory allocation and a copy-on-write (CoW) fork.

Before you begin

- Download, install, and run the original xv6 OS provided to you. 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.
- After you install the original version of the code, copy the files in the xv6 patch provided to you into the original code folder. For each part of this lab, you must start with a clean install of the original code and copy the modified files of that part alone. That is, both parts of this lab must be solved independently.
- For this lab, you will need to understand the following files: `defs.h`, `kalloc.c`, `mmu.h`, `syscall.c`, `syscall.h`, `sysproc.c`, `trap.c`, `user.h`, `usys.S`, `vm.c`.
  - The files `sysproc.c`, `syscall.c`, `syscall.h`, `user.h`, `usys.S` link user system calls to system call implementation code in the kernel.
  - The file `defs.h` acts as the header file for several parts of the kernel code.
  - The file `trap.h` contains trap handling code, including page faults.
  - The file `mmu.h` contains various definitions and macros pertaining to virtual address translation and page table structure.
  - The files `vm.c` and `kalloc.c` contain most of the logic for memory management in the xv6 kernel.
- 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.

Part A: Lazy Memory Allocation

Most modern operating systems perform lazy allocation of heap memory, though vanilla xv6 does not. 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. In this lab, you will add support for this lazy allocation feature in xv6, by delaying the memory requested by `sbrk()` until the process actually uses it. You can solve this lab 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. We will only test your code to check that simple shell commands are executing properly, so you need not worry about various other corner cases in your implementation.

Part B: Copy-on-Write Fork

In this part, you will implement the copy-on-write variant of the `fork()` system call. Please begin this part with a clean installation of the original xv6 code and copy the modified files provided for this part.

You will begin by adding a few system calls to xv6. Add the following system calls to help you track memory usage of a process in xv6.

- The system call `getNumFreePages()` should return the total number of free pages in the system. This system call will help you see when pages are consumed, and can help you debug your CoW implementation. You must add code to maintain and track freepages in `kalloc.c`, and access this information when this system call is invoked.
- The system calls `getNumVirtPages()` and `getNumPhysPages()` should return the number of logical pages and physical frames in the user part of the memory image of a process respectively. You must count the number of logical pages starting from virtual address 0 up to the size of the virtual address space of the process, stored in the `struct proc` of the process. Your count must include the pages to store the code/data from the executable, the stack (and guard page for the stack), and the heap (if it exists). You need not count the pages that map to the kernel address space. You must count the number of physical pages by walking the process page table, and counting the number of page table entries that have a valid physical address assigned. (Note that because xv6 does not use demand paging, you can expect these two numbers to be the same. However, counting them separately will give you some practice in walking page tables!)
- The system call `getNumPTPages()` should return the number of pages consumed by the page table of a process. This count must include the page used by the page directory, as well as all the pages used to store the inner page tables of the process. You must consider the page tables that store both user-level PTE mappings as well as kernel page table mappings.

Next, you will start the copy-on-write fork implementation. The current implementation of the fork system call in xv6 makes a complete copy of the parents memory image for the child. On the other hand, a copy-on-write (CoW) fork will let both parent and child use the same memory image initially, and make a copy only when either of them wants to modify any page of the memory image. We will implement CoW fork in the following steps.

1. Begin with changes to `kalloc.c`. To correctly implement CoW fork, you must track reference counts of memory pages. A reference count of a page should indicate the number of processes

that map the page into their virtual address space. The reference count of a page is set to one when a freepage is allocated for use by some process. Whenever an additional process points to an already existing page (e.g., when parent forks a child and both share the same memory page), the reference count must be incremented. The reference count must be decremented when a process no longer points to the page from its page table. A page can be freed up and returned to the freelist only when there are no active references to it, i.e., when its reference count is zero. You must add a datastructure to keep track of reference counts of pages in `kalloc.c`. You must also add code to increment and decrement these reference counts, with suitable locking.

2. Understand the various definitions and macros in `mmu.h`, e.g., to extract the page number from a virtual address. Feel free to add more macros here if required.

3. The main change to the `fork` system call to make it CoW fork will happen in the function `copyuvm` in `vm.c`. When you fork a child, you must not make a copy of the parent's pages for the child. Instead, the child should get a new page table, and the page tables of the parent and the child should both point to the same physical pages. This function is one place where you may have to invoke code in `kalloc.c` to increment the reference count of a kernel page, becasue multiple page tables will map the same physical page.

4. Further, when the parent and child are made to share the pages of the memory image as described above, these pages must be marked read-only, so that any write access to them traps to the kernel. Now, given that the parent's page table has changed (with respect to page permissions), you must reinstall the page table and flush TLB entries by republishing the page table pointer in the CR3 register. This can be accomplished by invoking the function *lcr3(v2p(pgdir))* provided by xv6. Note that xv6 already does this TLB flush when switching context and address spaces, but you may have to do it additionally in your code when you modify any page table entries as part of your CoW implementation.

5. Once you have changed the fork implementation as described above, both parent and child will execute over the same read-only memory image. Now, when the parent or child processes attempt to write to a page marked read-only, a page fault occurs. The trap handling code in xv6 does not currently handle the *T_PGFLT* exception (that is defined already, but not caught). You must write a trap handler to handle page faults in `trap.c`. You can simply print an error message initially, but eventually this trap handling code must call the function that makes a copy of user memory.

6. The bulk of your changes will be in this new function you will write to handle page faults. This function can be written in `vm.c` and can be invoked from the page fault handling code in `trap.c`, because you cannot easily invoke certain static functions like `mappages` from `trap.c`. When a page fault occurs, the CR2 register holds the faulting virtual address, which you can get using the xv6 function call `rcr2()`. You must now look at this virtual address and decide what must be done about this page fault. If this address is in an illegal range of virtual addresses that are not mapped in the page table of the process, you must print an error message and kill the process. Otherwise, if this trap was generated due to the CoW pages that were marked as read-only, you must proceed to make copies of the pages as needed.

7. Note that between the parent and the child processes, any process that attempts to write to the read-only memory image (whether parent or child) will trap to the kernel. At this stage, you must allocate a new page and copy its contents from the original page pointed to by the virtual

address. However, you must make copies carefully. If N processes share a page, the first $N-1$ processes that trap should receive a separate copy of the page in this fashion. After the $N-1$ copies are made, the last process that traps is the only one that points to this page (as indicated by the reference count on the page). Therefore, this last process can simply remove the read-only restriction on its page and continue to use the original page. Make sure you modify the reference counts correctly, e.g., decrement the count when a process no longer points to a page by virtue of getting its own copy. Also remember to flush the TLB whenever you change page table entries.

8. Finally, think about how you will test the correctness of your CoW fork. Write test programs that print various statistics like the number of free pages in the system, and see how these statistics change, to test the correctness of your code. We have provided a few test programs and expected output for you as reference.

Submission instructions

- For part A, you will need to modify the following files: `defs.h`, `trap.c`, `vm.c`. For part B, you will need to modify the following files: `defs.h`, `kalloc.c`, `mmu.h`, `syscall.c`, `syscall.h`, `sysproc.c`, `trap.c`, `user.h`, `usys.S`, `vm.c`. Please place the modified files of each part in separate subfolders.
- 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.

Grading

We will run your code with possibly different 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.