**Department Of CSE, IIT Jodhpur**
**CSL-3030 Operating Systems Lab**

**Mini Project: Topic 2**

**Memory management in xv6**

## Before you begin

- Make sure you have already downloaded, installed, and run the xv6 OS code given to you. 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.
- 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. You already know about them.
  - `mmu.h` and `defs.h` are header files with various useful definitions pertaining to memory management.
  - 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.
  - Understand the implementation of the `sbrk` system call that spans all of these files.
- 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 `mytest.c` as part of the custom xv6 tarball for this course. This test program is compiled by our modified `Makefile` and you can run it on the xv6 shell by typing `mytest` 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.

1. `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.

2. `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.

3. `getptsize()` returns the size of the page table of the process in terms of pages. That is, it returns the number of pages the OS allocates to the page table of the process. This count includes the outer page directory, as well as the inner page table pages allocated, to store all page table entries corresponding to the user and kernel parts of the virtual address space.

**Hints:**

- You can get a pointer to the PCB of the current process as follows.
  `struct proc *p = myproc();`
  From here, you can access all information about the process, e.g., `p->pgdir` is the pointer to the page directory, and so on.

- You can walk the page table of the process by using the `walkpgdir` function which is present in `vm.c`. You can find several examples of how to invoke this function within `vm.c` itself. To compute the number of physical pages in a process, you can write a function that walks the page table of a process in `vm.c`, and invoke this function from the system call handling code.

- The page directory of a process contains page table entries corresponding to the inner page table pages. You can access these entries using an array notation, e.g., `pgdir[i]`.

- You can check if a page table entry is present/valid or not by checking if the PTE_P flag is set. You can see an example of this usage in other functions like `walkpgdir`.

Please write your own test cases to test these system calls. Also, please understand the output and convince yourself that it makes sense.

## Part B: Memory mapping with `mmap` system call

In this part, you will implement a simple version of the `mmap` system call in xv6. The system call `mmap` takes 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 input is 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.

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. You can write more such test cases to thoroughly test your implementation.

Some helpful hints for you to solve this assignment are given below.

- Understand the implementation of the `sbrk` system call. Your `mmap` system call will follow a similar logic. In `sbrk`, the virtual address space is increased and physical memory is allocated within the same system call. The implementation of `sbrk` invokes the `growproc` function, which in turn invokes the `allocuvm` function to allocate physical memory. For your `mmap` im- plementation, you must only grow the virtual address space within the system call implementation, and physical memory must be allocated during the page fault. You may invoke `allocuvm` (or write another similar function) in order to allocate physical memory upon a page fault.

- The original version of xv6 does not handle the page fault trap. For this assignment, you must write extra code to handle the page fault trap in `trap.c`, which will allocate memory on demand for the page that has caused the page fault. You can check whether a trap is a page fault by check- ing 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 write code to 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.

So far in this assignment, we have assumed that the memory-mapped pages are allocated right after the program break, contiguous with the rest of the user memory image of the process. If you wish to make this problem more challenging, you can optionally extend your code to perform the memory-mapping at a user-specified address that is not necessarily contiguous with the rest of the memory image. You can add an extra argument `startaddr` to your system call, and then assign memory mapped pages starting at `startaddr`, if `startaddr` lies between the program break and `KERNBASE`, and no page is already mapped (virtual or physical) at this address. The system call should only allocate virtual memory, and physical memory should be allocated on demand, as before.

You will need to keep the following points in mind when implementing this extension.

- Now that your memory-mapped pages can be anywhere in the address space, and not necessarily up to the program break, you must identify such pages with a new flag. You can define a new flag `PTE_M` in `mmu.h`.
- You may have to change a few other system calls that make the assumption that the memory image extends only up to the program break. For example, when you count the number of virtual or physical pages for the system calls in part A, you must also count pages beyond the program break, to include pages that have the new flag set. In the fork system call, `copyuvm()` only copies pages up to the program break. Now, you must also copy memory-mapped pages between the program break and `KERNBASE`. Of course, you must allocate physical memory in the child for such pages only if the corresponding page has physical memory allocated in the parent.

We have not provided you any test cases to test this extension; please write them yourself using the template provided for the simpler `mmap` system call.

## Part C: Shared memory in xv6

In this part, you will be implementing a simplified IPC mechanism of shared memory between parent and child processes in xv6. The parent process memory maps a shared page into its virtual address space, and the child process inherits this page during fork. The shared page maps to the same physical memory in both parent and child processes, unlike other pages in the virtual address space that map to separate physical memory across both processes. The parent and child can then read and write into this shared page and see each other's changes.

This mechanism is implemented via the following system calls.

- `mapshared()` maps one page at the end of the user virtual address space of the parent process, and returns the starting virtual address of the new shared page. You may assume that the parent does not perform any `sbrk` system calls while using the shared page, so that the shared page always remains at the end of the address space. This shared page must be identified by a separate flag in the page table entry, say by defining the flag `PTE_S` in `mmu.h`. You must tag the shared page with such a flag in the page table.
- `getshared()` returns the starting virtual address of the shared page that is already present in the address space of a process. You may assume that there is only one shared page in the address space, and return the address of the page that has the shared flag set.
- `unmapshared()` unmaps the shared page from the address space of the process. It must return a negative value if there was some error during the unmap, and a non-negative value otherwise. This system call must remove the shared page from the address space of the invoking process, and also free up the physical memory of this page (if it was allocated). You can assume that this system call is invoked only once per shared page. Once one of the processes sharing the page unmaps it, you can assume that no other process will access this page, because their address space will have a pointer to an invalid page.

To correctly implement these system calls, you may have to change other functions in `vm.c` besides the code invoked by the system call itself. For example, you may need to change the code of `copyuvm` so that the child maps the same physical page as the parent for the shared page. You may also need to change the code in `deallocuvm` so that the shared pages are not freed up twice in the parent and child. If you free a page that was already freed up, you will see an error like this: `lapicid 0: panic: kfree`.

You are given two test cases, and a sample execution of your completed code should look like this.

```
$ test-shared1
child 42
parent 53
$ test-shared2
child 42
parent 53
child again 43
parent again 54
```

### Part D: Copy-on-Write Fork

In this part, you will implement the copy-on-write (CoW) variant of the `fork()` system call.

You will begin by adding a new system call to 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.

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 parent's 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.

- 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 a 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 data structure 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.
- 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.
- 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. 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.

- 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.
- 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.
- 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.
- 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 not provided any test cases and you can write your own.

Link for the code files: [Download Here]