OS-344 Assignment-2

Instructions

- **Assignment has to be done by a group of 4 members.**
- **Group members should ensure that one member submits the completed assignment within the deadline.**
- **Each group should submit a report, with relevant screenshots/ necessary data and findings related to the assignments.**
- **We expect a sincere and fair effort from your side. All submissions will be checked for plagiarism through a software and plagiarised submissions will be penalised heavily, irrespective of the source and copy.**
- **There will be a viva associated with assignment. Attendance of all group members is mandatory.**
- **Assignment code, report and viva all will be considered for grading.**
- **Early start is recommended, any extension to complete the assignment will not be given.**

The goal of this lab is to understand memory management in xv6. This assignment consists of two individual parts and you will be required to start with new copy of xv6 each time.

Before you begin

➔ 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*** command 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)

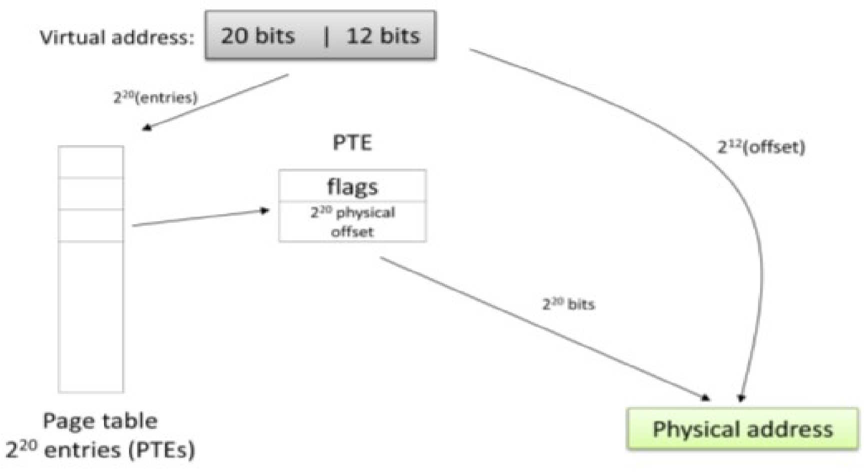
Start with a new copy of xv6. The patch that we had provided was only applicable to part A.

To help get you started, we strongly suggest you read this section while examining the relevant xv6 files (vm.c, mmu.h, kalloc.c, etc).

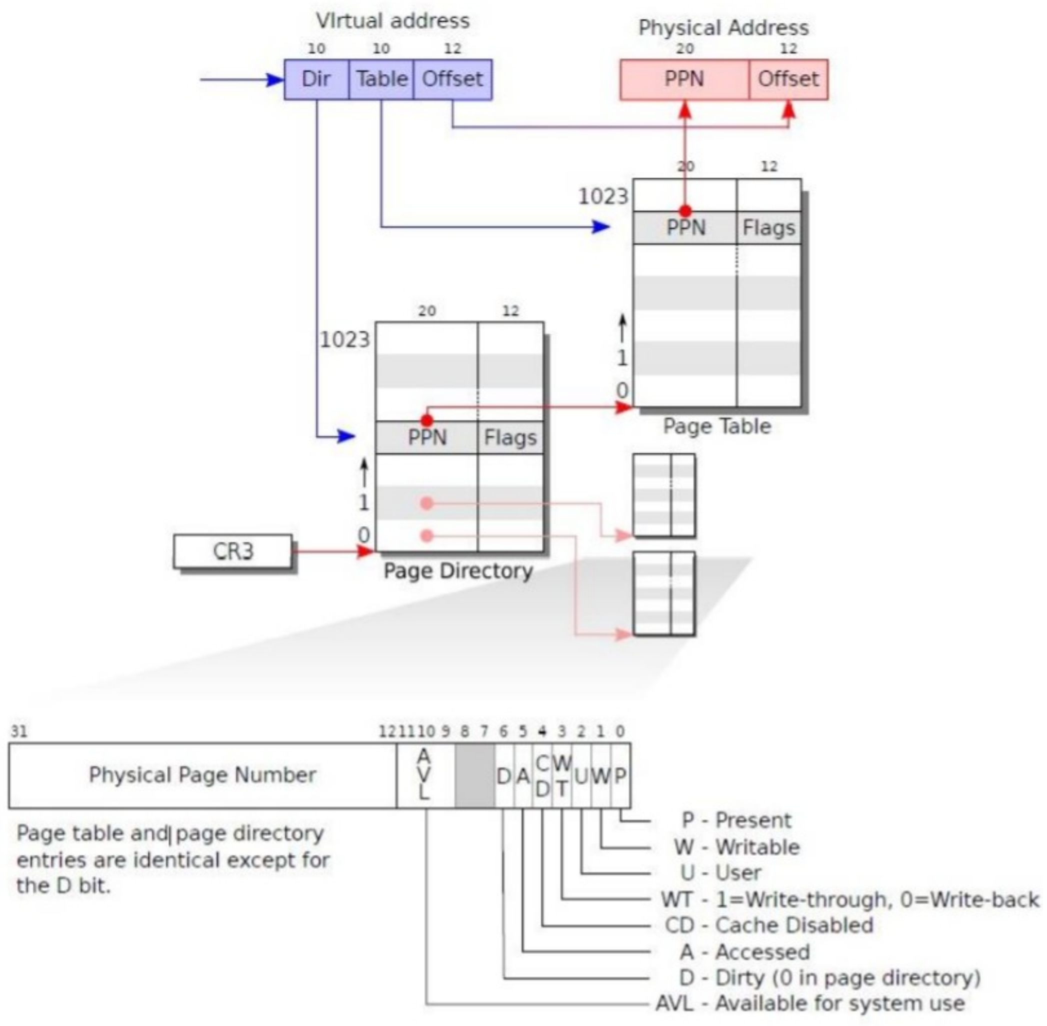
Xv6 memory

Memory in xv6 is managed in pages (and frames) where each such page is 4096 (= 2^{12}) bytes long. Each process has its own page table which is used to translate virtual to physical addresses.

The virtual address space can be significantly larger than the physical memory. In xv6, the process address space is 2^{32} bytes long while the physical memory is limited to 16MB only (see memlayout.h). When a process attempts to access some point in its memory (i.e. provides a 32 bit virtual address) it must first seek out the relevant page in the physical memory. Xv6 uses the first (leftmost) 20 bits to locate the relevant Page Table Entry (PTE) in the page table. The PTE will contain the physical location of the frame – a 20 bits page number (within the physical memory). These 20 bits will point to the relevant page within the physical memory. To locate the exact address within the page, the 12 least significant bits of the virtual address, which represent the in-page offset, are concatenated to the 20 bits retrieved from the PTE. Roughly speaking, this process can be described by the following illustration:



Maintaining a page table may require significant amount of memory as well, so a two-level page table is used. The following figure describes the process in which a virtual address translates into a physical one:



Each process has a pointer to its page directory (**line 59**, proc.h). This is a single page sized (4096 bytes) directory which contains the page addresses and flags of the second level table(s). This second level table is spanned across multiple pages which are very much like the page directory.

When seeking an address, the first 10 bits will be used to locate the correct entry within the page directory (by using the macro PDX(va)). The physical page number can be found within the correct index of the second level table (accessible via the macro PTX(va)). As explained earlier, the exact address may be found with the aid of the 12 LSB (offset).

At this point you should go over the xv6 documentation on this subject:

http://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf (chapter 2)

- ➔ Tip: before proceedings we strongly suggest you go over the code again. Now, attempt to answer questions such as:
  - ➔ How does the kernel know which physical pages are used and unused?
  - ➔ What data structures are used to answer this question?
  - ➔ Where do these reside?
  - ➔ Does xv6 memory mechanism limit the number of user processes
  - ➔ If so, what is the lowest number of processes xv6 can ‘have’ at the same time (assuming the kernel requires no memory whatsoever)?
- ➔ A very good and detailed account of memory management in the Linux kernel can be found here: http://www.kernel.org/doc/gorman/pdf/understand.pdf
- ➔ Another link of interest: “What every programmer should know about memory” http://lwn.net/Articles/250967/

Paging

An important feature lacking in xv6 is the ability to swap out pages to a backing store. That is, at each moment in time all processes are held within the main (physical) memory. In this task, you are to implement a paging mechanism for xv6 which is capable of swapping out pages and storing these to disk. Each swapped out page will be saved on a dedicated file whose name is the same as the process' pid and 20 MSB of virtual address (e.g., “<pid>_<VA[20:]>.swp”).

- ➔ Note: the xv6 file system supports file names of maximum 14 letters length.

- ➔ Note: there are good reasons for not writing to files from within kernel modules but in this assignment, we ignore these.
- ➔ For a few “good reasons”, read this: http://www.linuxjournal.com/article/8110
- ➔ And finally, if you really want to understand how swapping is done:

  http://www.kernel.org/doc/gorman/pdf/understand.pdf

Task 1: kernel processes:

In order to implement paging mechanism, we will use a kernel process (i.e., the processes that whole their “life” reside inside kernel mode). Therefore, you must implement following function:

*void create_kernel_process(const char *name, void (*entrypoint)());*

This function must create a kernel process and add it to the processes queue.

Task 2: swapping out mechanism:

Whenever a kernel tries to allocate memory for a process and fails (due to the lack of free physical memory), the process must be suspended from execution (i.e., the process state must be changed to SLEEPING). Next, a request (task) for a free page must be submitted to the kernel swapping out process. When a kernel swapping out process will receive this task, it will save the content of one of currently allocated pages (according to Least Recently Used (LRU) policy described at class) to the disk, remove the corresponding present bit from a PTE, mark the page as swapped out and finally mark this page as a free page.

- ➔ Note: the swapping out process must support a request queue for the swapping requests.
- ➔ Note: whenever there are no pending requests for the swapping out process, this process must be suspended from execution.
- ➔ Note: whenever there is exists at least one free physical page, all processes that were suspended due to lack of physical memory must be woken up.
- ➔ Note: only user-space memory can be swapped out (this does not include the second level page table)

Task 3: swapping in mechanism:

When the kernel detects a page fault, it must check if the cause of this page fault is in swapping out mechanism. If the page was actually swapped out, the kernel must suspend current process from execution and submit a swapping in task to corresponding kernel process. When a swapping in kernel process will receive a task, it will allocate a single page of physical memory, fill it with the corresponded content (that was swapped out) and finally map this page to a given virtual address.

- ➔ Note: when a page fault occurs not due to swapping out, the corresponding process must be safely terminated (the OS must continue running).
- ➔ Note: the swapping in process must support a request queue for the swapping requests.
- ➔ Note: whenever there are no pending requests for the swapping in process, this process must be suspended from execution.
- ➔ Note: when the swapping in process finish to map the physical page to virtual address, it must wake up corresponding process.
- ➔ Tip: a special attention must be put on the synchronization issues
- ➔ Tip: you may want to enrich the process struct with the swapping meta-data.

Task 4: Sanity test

Write a user-space program named memtest. The main process will fork 20 child processes. Each child process executes a loop with 10 iterations; at each iteration the process will allocate 4KB of memory using malloc system call; next it will fill the memory with values obtained from a simple function (you can choose the function you mostly like) and finally validate the content of the memory using the same function.

- ➔ Tip: in order to check the paging mechanism, you can reduce the value of PHYSTOP. This can prevent the kernel to be able to hold all processes memory in RAM.

Submission instructions

- Place all the files including Makefile that you modified for each part into separate subfolders named as Part_[A or B] and put both of them into a zip file, with the name being your group number (say, G10.zip).
- report.pdf should contain a detailed description of your new memory management policies that you had implemented for both the parts.
- Further, you must describe your test cases in some detail, and the observations you made from them.

--End of Assignment-3--