CSE 314 Assignment - 4
Enhancing the Memory Management System of xv6

Like many other components of xv6 OS, the memory management system isn't on par with modern-day standards. Your mission, should you choose to accept it would be to enhance and improve the memory management system of xv6 by doing the following tasks -

1. Updating the process details viewer
2. Developing a paging framework of xv6
3. Handling null-pointer protection

Note: Before starting the assignment, we strongly recommend you to go through the existing xv6 memory management system specially the page table code. Relevant files (but not limited to):
vm.c, mmu.h, kalloc.c, proc.c, proc.h, fs.c, memlayout.h, sysfile.c, trap.c, lapic.c, entry.S, defs.h

Resources:

1) Chapter 3 of xv6 Book
2) Structured Note on Chapter-3: Link
3) Virtual Memory and Paging Video Tutorial in xv6: Link
4) Class lectures, Tanenbaum Chapter 3 (3.1 - 3.5), Remzi Chapter - 18

Updating the process details viewer

Prior to enhancing the memory system of xv6, you will develop a tool to assist in testing the current memory usage of the processes in the system. To do so, enhance the capability of the xv6's process details viewer. Start by running xv6 and then press ctrl + P (^P) from within the shell. You should see a detailed account of the currently running processes. Try running any program of your liking (e.g., usertests/stressfs/forktest) and press ^P again (during and after its execution). The ^P command provides important information (what information is that?) on each of the processes. However, it reveals no information regarding the current memory state of each process. Add the required changes to the function handling ^P so that the memory state of all the processes in the system will also be printed as written below. The memory state includes the mapping from virtual pages to physical pages (for all the used pages), and also information regarding the used page tables.

For each process, the following should be displayed:
Process information currently printed with ^P
Page tables associated with the process
Page mappings - <virtual page number → physical page number>

You can follow any formatting to show the hierarchical structure of the Page tables but all relevant information should be present. A simple format is shown below:

For example, in a system with 2 processes, the information should be displayed as follows:
First process
1 sleep init 80104907 80104647 8010600a 80105216 801063d9 801061dd
Page tables:
    memory location of page directory = 800000
    pdir PTE 1, 63:
        memory location of page table = 258048
        ptbl PTE 1, 73, 299008
        ptbl PTE 2, 80, 327680

Page mappings:
1 -> 300
200 -> 500

Second process

2 sleep sh 80104907 80100966 80101d9e 8010113d 80105425 80105216
Page tables:
[etc etc..]
Page mappings:
1 -> 306
200 -> 500

Explanation:
Showing that the first process has 2 pages. Virtual page number 1 is mapped to physical page number 300 and virtual page number 200 is mapped to physical page number 500. The page directory of the first process is in physical memory address 800000, the first PTE is unused, the second PTE is used and its PPN is 63. This PPN points to physical memory address 258048. That page table has 2 used PTEs (the first is unused). The PPNs of these 2 page table PTEs are 73 and 80, which point to physical memory addresses 299008 and 327680, respectively.

*Please Note: * The values above are made up. * Don't print anything for pages or page tables that are currently unused. * Only print pages and page tables that are user pages (PTE_U).*

Developing the Paging Framework of xv6

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 physical memory. You have to implement a paging framework for xv6 which can take out pages and store them to disk. Also, the framework will retrieve pages back to the memory on demand. In your framework, each process is responsible for paging in and out its own pages.

File Framework

To keep things simple, we will use the file system interface supplied and create for each process a file in which swapped-out memory pages are stored. We supplied a framework for creating/ writing/ reading and deleting swap files. The framework was implemented in fs.c and uses a new parameter named swapFile that was added to the proc struct (proc.h). This parameter will hold a pointer to a file that will hold the swapped memory. The files will be named "/.swap" where the id is the process id. Review the following functions and understand how to use them.

A. int createSwapFile (struct proc *p) – Creates a new swap file for a given process p. Requires p- >pid to be correctly initiated.
B. int readFromSwapFile (struct proc *p, char* buffer, uint placeOnFile, uint size) – Reads size bytes into buffer from the placeOnFile index in the given process p swap file.
C. int writeToSwapFile (struct proc *p, char* buffer, uint placeOnFile, uint size) – Writes size bytes from buffer to the placeOnFile index in the given process p swap file.
D. int removeSwapFile (struct proc *p) – Delete the swap file for a given process p. Requires p->pid to be correctly initiated.

This is provided as a patch file to be applied in a fresh copy of xv6 before the start of your assignment. Read the README.txt about how to apply the patch.

Storing pages in a file

In any given time, a process should have no more than MAX_PSYC_PAGES (= 15) pages in the physical memory. Also, a process will not be larger than MAX_TOTAL_PAGES (= 30) pages. Whenever a process exceeds the MAX_PSYC_PAGES limitation, it must select enough pages and move them to its dedicated file. You can assume that any given user process will not require more than MAX_TOTAL_PAGES pages. To know which page is in the process' page file and where it is in that file (i.e., paging meta-data); you should maintain a data structure.

We leave the exact design of the required data structure to you. You may want to enrich the process struct with the paging meta-data. Be sure to write only the process' private memory. There are several functions already implemented in xv6 that can assist you – **reuse existing code**. Don't forget to free the page's physical memory.

Whenever a page is moved to the paging file, it should be marked in the process' page table entry that the page is not present. This is done by clearing the present (PTE_P) flag. A cleared present flag does not imply that a page was swapped out (there could be other reasons). To resolve this issue, we use one of the available flag bits in the page table entry to indicate that the page was indeed paged out. Now, whenever you move a page to the secondary storage set this flag as well.

Add the following line to mmu.h:

. #define PTE_PG 0x200 // Paged out to secondary storage

Now, whenever you move a page to the secondary storage set this flag as well.

Retrieving pages in a file

While executing, a process may require paged out data. Whenever the processor fails to access the required page, it generates a trap (interrupt 14, T_PGFLT). After verifying that the trap is indeed a page fault, use the %CR2 register to determine the faulting address and identify the page.

Allocate a new physical page, copy its data from the file, and map it back to the page table. After returning from the trap frame to user space, the process should retry executing the last failed command again (should not generate a page fault now). Don't forget to check if you passed MAX_PSYC_PAGES, if so another page should be paged out.

Comprehensive changes in existing parts

These changes also affect other parts of xv6. You should modify the existing code to properly handle the new framework. Specifically, make sure that xv6 can still support a fork system call. The forked process should have its own page file which is identical to the parent's file.

Process Termination

Upon process termination, the kernel should delete the page file, and properly free the process' pages that reside in the physical memory

Page Replacement Algorithms

Now that you have a paging framework, there is an important question that needs to be answered: Which page should be swapped out? As seen in the CSE 313 theory class, there are numerous alternatives to selecting which page should be swapped. For this assignment, we will limit ourselves to the following choices of algorithms:

A. FIFO – this is the default replacement algorithm that follows the order in which the pages were created.
B. NFU – we will approximate LRU in this algorithm.
C. NRU – this is another crude approximation of LRU.
D. Aging – Efficient approximation of LRU.

In the algorithm B,C and D, we will approximate LRU using the PTE_A (accessed/reference bit) flag of the page table entry.

Alas! So many algorithms! How am I going to do that?
-Well, to relieve your workload a bit, we devised a scheme for you to implement only **two** page replacement algorithms using your student ID:

((7-digit Student ID) mod 3) + 1	**Implementable Algorithms**
1	A + C
2	A + B
3	A+ D

Protection against Null Pointer

Many software exceptions occur when trying to dereference null points (accessing memory at location 0). While in certain situations it may be desirable to access the memory at address 0, it usually points to a bug in the code. For this reason, modern operating systems crash after dereferencing a null pointer, making it quicker to find bugs. xv6, on the other hand, does not offer such protection and allows access to address 0, which contains the program's text segment (go ahead and try to tamper with it).

Your task is to implement such functionality for user programs (and fork_test). To do so, make sure that the first memory page is never used (unallocated), and that the text segment begins at the second page. Find and update all the relevant places. Note that the Makefile needs to be updated as well, have a look at flag -T.

As a simple sanity test, write a small user program that prints out the contents of memory address 0. While this should work before the changes, when you have completed this task, accessing the memory at that location should cause an error.

Submission Guidelines

Deadline - 11/02/2022 (11:55 pm)

Start with a fresh copy of xv6 from the original repository. Make necessary changes for this offline. Don't commit. You can also create any additional files if need be. Then create a patch using the command line interface:

```
git diff HEAD > studentID.patch
```

Where studentID = your own seven-digit student ID (e.g., 1705XXX). Just submit the patch file. In the lab, during evaluation, we will start with a fresh copy of xv6 and apply your patch using the command line:

```
git apply studentID.patch
```

Please DO NOT COPY solutions from anywhere (your friends, seniors, internet, etc.). Implement the algorithms on your own with your style of coding. Any form of plagiarism (irrespective of source or destination), will result in getting -100% marks. You have to protect your code.