CSE 314: xv6 Assignment 3

Developing a Paging Framework for 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 storing 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. 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.

Supplied file framework

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<id>' where the id is the process id. Review the following functions to understand their use.

- int createSwapFile (struct proc *p) – Creates a new swap file for a given process p. Requires p->pid to be correctly initiated.
- 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.
- 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.
- int removeSwapFile (struct proc *p) – Delete the swap file for a given process p. Requires p->pid to be correctly initiated.

Storing pages in files

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.

Retrieving pages on demand

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.

Process Termination

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

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.

Page replacement algorithms

Now that you have a paging framework, there is an important question which 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 only a few simple selection algorithms:

- FIFO, according to the order in which the pages were created
- Bonus – Second chance algorithm, according to the order in which the pages were created and the status of the PTE_A (accessed/reference bit) flag of the page table entry.

Important 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