EECE 315 – UBC

Assignment 4

NOTE: Solution to Assignment 3 and test cases for Assignment 4 will be made available on Friday, when demo for Assignment 3 is over. If you have completed Assignment 3 well, you should be good to go, otherwise you can use code provided in the Assignment 3 solution as a reference for Assignment 4. Also we will add some more test cases for you to test your code for assignment 4.

Overview

In this lab you will extend Nachos to support **demand paged virtual memory**. This new functionality gives processes the illusion of a virtual memory that is larger than the available machine memory.

You will implement and debug virtual memory in two steps. First, you will implement demand paging using page faults to dynamically load process virtual pages on demand, rather than initializing page frames for each process in advance at Exec time as you did in Assignment 3. Next, you will implement page replacement, enabling your kernel to evict a virtual page from memory to free up a physical page frame to satisfy a page fault. Demand paging and page replacement together allow your kernel to "overbook" memory by executing more processes than would fit in machine memory at any one time, using page faults to "juggle" the available physical page frames among the larger number of process virtual pages. If it is implemented correctly, virtual memory is undetectable to user programs unless they monitor their own performance.

The operating system kernel works together with the machine's memory management unit (MMU) to support virtual memory. Coordination between the hardware and software centers on the page table structure for each process. You used page tables in the previous assignment to allow your kernel to assign any free page frame to any process page, while preserving the illusion of a contiguous memory for the process. The indirect memory addressing through page tables also isolates each process from bugs in other processes that are running concurrently. In this project, you will extend your kernel's handling of the page tables to use three special bits in each page table entry (PTE):

- **Valid bit**: The kernel sets or clears the valid bit in each PTE to tell the machine which virtual pages are resident in memory (a valid translation) and which are not resident (an invalid translation). If a user process references an address for which the PTE is marked invalid, then the machine raises a page fault exception and transfers control to your kernel's exception handler.

- **Use bit**: The machine sets the use bit (aka reference bit) in the PTE to pass information to the kernel about page access patterns. If a virtual page is referenced by a process, the machine sets the corresponding PTE reference bit to inform the kernel that the page is active. Once set, the reference bit remains set until the kernel clears it.
- **Dirty bit**: The machine sets the dirty bit in the PTE whenever a process executes a store (write) to the corresponding virtual page. This informs the kernel that the page is dirty; if the kernel evicts the page from memory, then it must first "clean" the page by writing its contents to disk. Once set, the dirty bit remains set until the kernel clears it.

Task 1 – Demand Paging

In the first part, you will continue to pre-allocate a page frame for each virtual page of each newly created process at Exec time, just as in assignment 3. As before, return an error from the Exec system call if there are not enough free page frames to hold the process' new address space. But for this part, you need to make the following changes to AddrSpace:

1. In your AddrSpace initialization method, initialize all the PTEs as **invalid**.

2. In the same method, *remove* the code that (1) zeros out the physical page frames, (2) preloads the address space with the code and data segments from the file, and (3) prevents programs that require too much memory from proceeding. You will do this on demand when the process causes a page fault -- you will continue to allocate physical page frames in AddrSpace for each virtual page, but delay loading the frames with data until they are actually referenced by the process.

3. Handle page fault exceptions in ExceptionHandler. When the process references an invalid page, the machine will raise a page fault exception (if a page is marked valid, no fault is generated). Modify your exception handler to catch this exception and handle it by preparing the requested page on demand.

4. To prepare the requested page on demand, add a method to AddrSpace to page in the faulted page from the executable file. Note that faults on different address space segments are handled in different ways. For example, a fault on a code page should read the corresponding code from the executable file, a fault on a data page should read the corresponding data from the executable file, and a fault on a stack frame should zero-fill the frame. For pages that hold the arguments to the process, you will fault

on them when the child writes the arguments to its address space (using WriteMem) when the child thread first starts running in the Nachos kernel and before the program starts running.

5. Once you have paged in the faulted page, clear the page fault exception by marking the PTE as **valid**. Then let the machine restart execution of the user program at the faulting instruction -- when you return from the exception, do not increment the PC as you did when handling a system call so that the machine will re-execute the faulting instruction.

If you set up the page (by initializing it) and page table (by setting the valid bit) correctly, then the instruction will execute correctly and the process will continue on its way, none the wiser.

As you make the changes above, keep the following points in mind:

1. Remember, a virtual page may contain portions of two segments, such as the end of the code segment and the beginning of the data segment. A fault on that page will require you to load from both the code and data segments into that page. You need to handle this boundary case for all situations where two segments can overlap on a page (code, data, stack, and argument).

2. If you use ReadMem (or WriteMem) to implement a system call, it is entirely possible for those functions to reference a page that has yet to be loaded into memory (since you can give them an arbitrary address in the process virtual address space). If this happens, ReadMem will return FALSE because it triggered a page fault when it tried to access the address you gave it. You *should not* consider this an error. Instead, you should retry the operation assuming that the referenced page was successfully loaded. If it is FALSE again, then return an error.

3. StartProcess and Exec closed the executable file after creating the address space. You no longer have this luxury, and will have to keep it open during the life of the process.

Testing:

Start by testing with one process running at a time. During debugging, you will probably want to print out the arguments that you are giving to ReadAt and bzero when initializing a page during a page fault to make sure that you are loading the correct parts of the executable file into the virtual page.

Expected output:

1. All the test programs should be running without any fault.
2. Print the relevant debug statements showing the memory allocation on page fault.

Task 2 – Page Replacement

Now implement demand paged virtual memory with page replacement. In this second part, not only do you delay initializing pages, but now you delay the allocation of physical page frames until a process actually references a virtual page that is not already loaded in memory.

1. Start by completing the gutting of your code that creates an address space. In part one, you removed the code that initialized the virtual pages. Now, remove the code that (1) allocates page frames and (2) preinstalls virtual-physical translations when setting up the page table. Instead, merely mark all the PTEs as invalid.

2. Extend your page fault exception handler to allocate a page frame on-the-fly when a page fault occurs. In part one, you just initialized the virtual page when a page fault occurred. In this part, allocate a physical page for the virtual page and use your code from part 1 to initialize it, mark the PTE as valid, and return from the exception.

You can get the above two changes working without having page replacement implemented for the case where you run a single program that does not consume all of physical memory. Before moving on, be sure that the two changes above work for a single program that fits into memory (e.g., array).

Now implement page replacement to free up a physical page frame to handle page faults:

1. Extend your page fault exception handler to evict pages once physical memory becomes full. First, you will need to select a victim page to evict from memory; for now, keep this simple and just choose a convenient page. Then mark the PTE for that page as invalid.

2. Evict the victim page. If the page is clean (i.e., not dirty), then the page can be used immediately; you can always recover the contents of the page from disk. If the page is dirty, though, the kernel must save the page contents in backing store on disk.

3. Read in the contents of the faulted page either from the executable file or from backing store (see below).

4. Implement a BackingStore class to handle page in and page out operations. An important part of this project is to use the Nachos file system interface to allocate and manage the backing store. Implement methods to allocate space on backing store, locate pages on backing store, push pages from memory to backing store (for page out), and pull from backing store to memory (for page in).

5. Use the FileSystem class to create files for backing store (see filesys/filesys.h). After creating the backing store file, use the FileSystem class to open it. Opening the file will return an OpenFile object, which allows you to do reads and writes (see filesys/openfile.h). The Makefiles are setup to compile with FILESYS_STUB defined, so be sure to look at that version of the classes.

As you implement the above operations, keep the following points in mind:

1. As in the first part of the project, the first time a page is touched it needs to be initialized (from the executable file for code and data, bzero'd for bss and stack, or initialized when writing arguments; see part 1 above). If this page is subsequently evicted to backing store, it will be read from there on further page faults.

2. You are not limited to one file for backing store for the entire system, and might find it more convenient to have a backing store file for each process (doing so makes locating evicted pages convenient). However, do not create backing store files at finer granularities; e.g., do not use one file per page.

3. Be sure to clear the dirty bit when you mark a PTE for the victim page as invalid.

4. When running multiple processes, you may select a victim page from another process. As a result, you will need to update the PTE in the page table for that process, not the faulting one.

5. Finally, you should only do as many page reads and writes as necessary to execute the program, and as dictated by your page replacement algorithm. You will soon discover that the first page fault is different than subsequent ones on code and data pages. On the first fault you need to read from the executable file, and on the second you need to read from the backing store. Your implementation needs to be able to handle this situation. It might be tempting to just copy the pages from the executable file to the backing store when the process is first created, or on a page-by-page basis when the first fault occurs, but both of these cases introduce extra unecessary disk I/Os and should not be used.

Testing:

Reduce the memory size of physical memory by modifying ***NumPhyPages*** *variable in* ***machine/machine.h*** *file.*

This way you reduce the memory size, therefore running a larger program will cause the memory conflicts, therefore your code to clear pages from memory will be executed.

Start by testing with one process running at a time. During debugging, you will probably want to print out the arguments that you are giving to ReadAt and bzero when initializing a page during a page fault to make sure that you are loading the correct parts of the executable file into the virtual page.

Expected output:

1. All the test programs should be running without any fault.
2. Print the relevant debug statements showing the memory allocation and de-allocation.

Evaluations

Task 1

Correct Implementation 50%

(Your output should print relevant debug statement for memory allocation)

Task 2

Correct Implementation 50%

(Your output should print relevant debug statement for memory allocation and de-allocation)

Submission Guidelines

You need to submit all the files that you will be modifying for this assignment:

Create and submit the zip file as an attachment to the eece315term2@gmail.com with the subject as follows:
“Assignment4: <Group No.>”

Once you submit the assignment, you will receive an automated reply from the system. You need to show that reply to the TA before they could assess your assignment in the lab, to make sure that you have submitted your assignment correctly.