Lab 4: Virtual Memory in NachOS

CMSC 442

Due: Friday, Nov. 10th, 2017 by 11:59:59pm

(Note: This lab, and all the other Nachos labs were derived from the original NachOS projects by Tom Anderson at UC Berkeley and the Duke Nachos Guide. They have been modified to fit our lab environment and changes in the compilation software since NachOS was originally published.)

The third phase of Nachos is to investigate the interaction between the TLB; the virtual memory system, and the file system. We provide no new virtual memory code for this assignment. The only change is that you need to compile with the "-DVM -DUSE_TLB" flags. This is done for you automatically if you compile from the "vm" folder.

When you compile NachOS this way, it will fetch pages in a completely different manner than it did when you ran it from userprog. This means that basically none of your test cases will work until you've completed Task 1.

In the original version of this project, you had to implement filesystem functions in addition to the virtual memory calls. I have simplified the project, so you can continue to use the "stub" version of the filesystem for this lab.

Virtual memory means that we use memory as a cache for disk, to provide an abstraction of an (almost) unlimited virtual memory size with performance close to that provided by physical memory. For this assignment, page translation allows us the flexibility to get pages from the disk as they are needed.

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. To implement virtual memory, you will extend your kernel's handling of the page tables to use three special bits in each page table entry (PTE):

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 in userprog/exception.cc (this is the same file in which you handled system calls).

The machine sets the **use** bit (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.

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 preserving its contents on disk. Once set, the dirty bit remains set until the kernel clears it.

As with any caching system, performance depends on the policy used to decide which things are kept in memory and which are stored only on the disk. On a page fault, the kernel must decide which page to replace . Ideally, it will throw out a page that will not be referenced for a long time, keeping pages in memory those that are soon to be referenced. Another consideration is that if the replaced page has been modified, the page must be first saved to disk before the needed page can be brought in. Many virtual memory systems (such as Unix) avoid this extra overhead by writing modified pages to disk in advance so that any subsequent page faults can be completed more quickly.

The first thing to consider is the software-managed translation lookaside buffer (TLB): the TLB is a cache of translations to provide the illusion of fast access to a large address space. Page tables were used in the previous lab to simplify memory allocation and to isolate failures in one address space from affecting other programs.

For this assignment, the **hardware** knows nothing about page tables. Instead, it deals only with a software-loaded cache of page table entries, called the TLB. On most modern processor architectures, a TLB is used to speed address translation. Given a memory address (of an instruction to fetch or a data value to load or store) the processor looks first in the TLB to determine if the mapping of virtual page to physical page is already known. If so (a "TLB hit") the translation can be done quickly. If not (a "TLB miss"), page tables and/or segment tables are used to determine a correct translation.

On several architectures, including Nachos, a "TLB miss" simply causes a trap to the O.S. kernel which does the translation, loads the mapping into the TLB, and returns control to the program. This allows the O.S. kernel to choose whatever combination of page table, segment table, inverted page table, etc., it needs to do the translation. On systems without a software-managed TLB, the hardware does the same thing as the software, but in this case, the hardware must specify the exact format for page and segment tables. Thus, software managed TLBs are more flexible at a cost of being somewhat slower for handling TLB misses. If TLB misses are infrequent, the performance impact of software-managed TLBs is minimal. Each entry in the TLB has a valid bit: if the valid bit is set, the virtual page is in memory. If the valid bit is clear or if the virtual page is not found in the TLB, software translation is needed to tell whether the page is in memory (with the TLB to be loaded with the translation) or the page must be brought in from the disk. In addition, the hardware sets the use bit in the TLB entry whenever a page is referenced and the dirty bit whenever the page is modified.

When a program references a page that is not in the TLB, the hardware generates a TLB exception, trapping to the kernel. The operating system kernel then checks its own page table. If the page is not in memory, it reads the page in from the disk, sets the page table entry to point to the new page and them resumes execution of the user program. Of course, the kernel must first find space in memory for the incoming page, potentially writing some other page back to disk, if it has been modified.

This provides the flexibility to implement inverted page tables without changing anything about the hardware. You will also need to do something about making sure the TLB state is set up properly on context switch. Most systems simply invalidate all TLB entries on a context switch; the entries get reloaded as the pages are references.

Task 0. Getting Started (To be completed by the group)

This lab builds upon your work in Lab 3. Your code from Lab 3 was supposed to be able to run programs from the test folder using the -x flag. In order for that to work correctly, you needed to have a working address space class, and implementations of at least the Exec, Exit, and Join system calls. You also needed to get the Read and Write system calls working at least enough for console I/O to work properly.

To be able to successfully complete this lab, you will need to fix anything that was broken in the previous lab. You should be able to at least begin work on the rest of this lab while you are debugging and correcting issues from Lab 3 as long as you are careful not to introduce lots of bugs that break compilation.

Task 1. Page Replacement and Demand Paging (to be implemented by the whole group)

Right now, your system loads the entire program into physical memory in the Addrspace constructor. This means that the number of processes we can concurrently run is limited by the size of Physical Memory. By default, NachOS supports 32 physical pages of 128 bytes each. That's only 4Kb of memory. If you were to run 4 processes, each of them could only have 1Kb of space (8 pages of physical memory). Obviously, this is undesirable.

To fix this, we are going to implement swapping. The basic idea is that your code and data will be stored on disk in a special "swap file" rather than directly in physical memory. Another way to think about this is we will store virtual memory in the file and use Physical Memory as a cache for it.

We are also going to implement "demand paging". Instead of loading all of the code and data immediately when we launch a program (either the main thread or through "Exec"), we are going to load only the pages we need when they are first accessed.

When a process needs to access a value in memory, the hardware will try to access the page it is in. If it discovers that the page is "invalid", it will generate a "page fault" exception that you can catch in the big if-statement in "exception.cc".

To implement this, you need to do several things:

1. Create (and open) a "swapfile" that can hold at least four times the amount of physical memory that NachOS provides. You can use the Create and Open functions calls from filesys/ for this (you should be familiar with these from the previous lab). Remember that the Create

function takes an initial size. While you probably set this to 0 to implement the Create system call, you will now want to specify a size.

You will need to read from and write to this file in several different places in your code, so it's probably a good idea to make the OpenFile* a global variable (perhaps in system.h/system.cc?)

2. Create a bitmap (or other data structure) to keep track of which "pages" of swap memory are being used. When we load pages into virtual memory, we will use this "free list" to find a "swap page" that isn't already used.

3. Create a global hash table (called the "inverted page table") that will allow you to look up the TranslationEntry struct for each page currently cached in Physical Memory. You will need to create your own Hashtable class for this (though you are welcome to use C++ STL templates such as "map" template if you find it helpful).

This is the critical step in this lab – understanding and implementing the inverted page table is what will allow you to make swapping work. This "inverted page table" will replace the per-process PageTable you implemented in the previous lab. Instead of each process managing its own separate page table, we are going to combine them all into a single global data structure.

In order to keep the size of this data structure sufficiently small, we will keep a mapping of physical pages to virtual pages instead of the other way around. This means that if we need to find virtual page 3 of process 2, we can search the inverted page table for "page 3, process 2" in the inverted page table. If we find it, we can directly access the memory. If not, we can load the page into physical memory.

4. Complete the gutting of your code to create an address space in the AddrSpace constructor: remove the code to allocate page frames and map virtual pages to physical pages and the code that reads pages from the executable. Instead, merely mark all the page table entries as invalid.

For this assignment, your kernel delays allocation of physical page frames until a process actually references a virtual page that is not already loaded in memory.

We will load the pages of code and data AS NEEDED when page faults occur (i.e. we will use "demand paging"). When a page fault occurs, you will first check to see if that page is already in physical memory (by looking for a mapping to it in your inverted page table). If it is not in memory, you should you should look for it in swap memory.

As an initial step, it is possible to load all pages from a program into swap memory in the addrspace constructor. However, you will eventually need to implement "demand paging" where pages only get pulled into virtual memory when they are used. No matter how you

decide to do this, be sure to handle the condition that there are not enough free page frames to hold the new process address space.

5. Implement *page replacement*, enabling your kernel to evict any virtual page from memory in order 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 can 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. This step should be done using software management of the TLB using an "inverted page table". Note that with the compile time flag -DUSE_TLB the **hardware** no longer deals with page tables. Instead, the hardware traps to the OS on a TLB miss, triggering code in userprog/exception.cc.

Modify your exception handler to catch the page fault exception and handle it by initializing the requested page on demand. If the memory access is to a valid page already in memory, NachOS will handle the memory access for us. However, if the page is not already loaded into physical memory, NachOS will place the virtual address it is trying to access in the register "BadVaddrReg" and generate a "PageFault" exception that we can handle in exception.cc.

Faults on different address space segments are handled in different ways. For example, a fault on a text page (i.e. a code page) should read the code from the executable file, but a fault on a stack or uninitialized data frame should create a new page which consists entirely of zeros. This step will likely require a restructuring of your AddrSpace initialization code as well as adding an additional "else if" in your ExceptionHandler function.

Extend your page fault handler to allocate a frame on-the-fly when a page fault occurs. If memory is full, it will be necessary to free up a frame by selecting a victim page to evict from memory. To evict a page, the kernel marks the corresponding PTE(s) invalid, then frees the page frame and/or reallocates it to another virtual page.

Extend the eviction and initialization code to preserve and retrieve the contents of evicted pages as necessary. The system must be able to retrieve a victim page contents if the victim page is referenced at a later time; if the page is dirty, the system must save the page contents in backing store (or swap space), e.g., on local disk.

Use the Nachos file system interface to allocate and manage the backing store. You will need to write functions 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). Be sure to handle the dirty bits correctly when you evict and retrieve pages. In order to find unreferenced pages to throw out on page faults, you will need to keep track of all of the pages in the system which are currently in use, via the inverted page table.

In this way, your operating system will use main memory as a cache over a slower and cheaper backing store. As with any caching system performance depends largely on the policy used to decide which pages are kept in memory and which to evict. As you know, we are not especially concerned about the performance of your Nachos implementations (simplicity and correctness are paramount), so for now, you don't need to use anything fancy. You may use the FIFO replacement algorithm.

Another important design question is the handling of dirty pages. Naive kernels may allow processes to fill memory with dirty pages. Consider what can happen when memory is full of dirty pages. If a page fault occurs, the kernel must find a victim frame to hold the incoming page, but every potential victim must be written to disk before its frame may be seized. Cleaning pages (by saving them to disk) is an expensive operation, and it could even lead to deadlock in extreme cases, if for example a page is needed for buffering during the disk write. For these reasons, "real" operating systems retain a reserve of clean pages that can be grabbed quickly in a pinch. Maintaining such a reserve generally requires a paging daemon to aggressively clean dirty pages by pushing them to disk if the reserve begins to drain down. You do not need to worry about this issue, except to be sure that dirty pages are written to disk before being replaced.

After initializing the frame on a page fault, clear the exception by marking the PTE as valid, then resume execution of the user program at the faulting instruction so that it retries the memory access (which should now succeed). This means that you should NOT update the program counter following a page fault. If you have set up the page and page table correctly, then the instruction will execute correctly and the process will continue on its way, none the wiser.

Hints:

The "inverted page table" idea is used by PowerPC-based computers. It is essentially a hash table. We hash on the virtual address to find where it is stored in physical memory and compare it against the virtual page actually stored there. If the page in physical memory does not match (for example, if it is owned by a different process), we use "chaining" to check other locations in the page table. If no valid page in the table matches, we bring the in from the swap file on the disk and update the entry in the inverted page table.

For example, a simple hash function would be to just use the low-order bits of the virtual address to index the table using the formula *paddr = vaddr % size*, where size is the number of physical pages. In this case, each virtual page would have only one place where it can be loaded, which significantly simplifies implementation.

Task 2. Pre-fetching (To be completed by a single member of the group)

Pre-fetching is an optimization that takes advantage of spatial locality to speed up memory access. Spatial locality is the idea that if we access data on one page, we are fairly likely to access data on the subsequent page soon. We can take advantage of this by loading more than one page into memory at a time when a page fault occurs. In Nachos, we won't see a huge performance increase from this (in fact,

it might actually hurt performance), but on a real system with a physical hard drive, this can significantly reduce the number of page faults and lead to a big performance boost.

Extend your virtual memory system so that instead of loading a single page, it loads pages in pairs. For example, if virtual page 2 gets loaded into memory, virtual page 3 should also get loaded. If virtual page 5 gets loaded, so should virtual page 4. When loading an even page, you should load the next page at the same time. When loading an odd page, you should load the previous page.

Task 3. CLOCK (To be completed by a single member of the group)

Replace the FIFO page replacement algorithm with a version of CLOCK. *The Nachos TLB sets and clears dirty and use bits, which you can use for this purpose*, however you will need to keep track of the "next" pointer (the hand of the clock) and manage the communication with the TLB in such a way that the correct page is replaced.

Task 4. Statistics (To be completed by a single member of the group)

Add code that keeps track of how many page faults and page hits your virtual memory system incurs. Then add a system call named "Memstats()" with system call number 11 (add #define SC_MEMSTATS 11) that returns the current "hit rate". For example, if you have 20 hits and 12 misses, your hit rate would be 20/32 or about 62.5%. Your code should return the hit rate as an integer percent rounded down to an even value. In other words, with 20 hits and 12 misses, calling Memstats would return the value "62".

Task 5. Documentation (To be completed by the entire group)

In your base nachos directory, I would like you to create a file named LAB4-README.TXT that contains:

Your group name, your group submit name, the names of each member of your group, and a description of the contributions of each member (in paragraph form). Your description should be detailed, explaining which files you changed and which task they implemented. If you were unable to get part of a task working, you should be explicit about that and explain the problem in detail.

Submit as usual through the course web site.