Sean Cork & Dustin Hines
Professor Barker
Operating Systems
9 May 2018

Project 4 Writeup

Introduction

In this project, we implemented an external pager that maintains a page table for use by a MMU. The pager maintains a mapping of virtual addresses to pages in physical memory for each running process. The virtual pages are either stored on disk or in physical memory. It is the pagers responsibility to ensure that data being used by processes is located in memory. By using read/write faults determined by bits in the page table, the pager ensures that virtual pages will be in physical memory when either written to or read by a process. It is important to note that the pager is only evoked by calls from the MMU or when a fault occurs--if our pager needs to do any work it must ensure that a read or write fault occurs.

Design

Our first design issue that we dealt with was trying to figure out which data structures we were going to use in our project and how we were going to store the state of each page table entry. We decided first that we were going to store our page table entries in a struct that contained critical information about the page such as its virtual page, its physical page, whether it has been modified, whether it's extended but not zero filled, its reference bit and its resident bit. All of these bits that we are keeping track of were essential in the implementation of the pager. In the same struct we also kept track of the process it originated from by keeping a pointer to another struct that contains information about the current process. Next we created a struct that keeps track of information about the current process. This struct contained the current pid, a pointer the the processes page table and map that contains virtual page numbers mapped to pages. We kept a global struct that contains a map of pid's to process info struct pointers. This helped us keep track of which processes were active in the clock structure. We decided to use a queue as our clock structure because it was easy to implement a circular representation of the clock hand. The last two global structs that we used kept track of the number of disk blocks that the process could use and a vector of ints that contained the free blocks available and the other struct contained a queue of available physical pages.

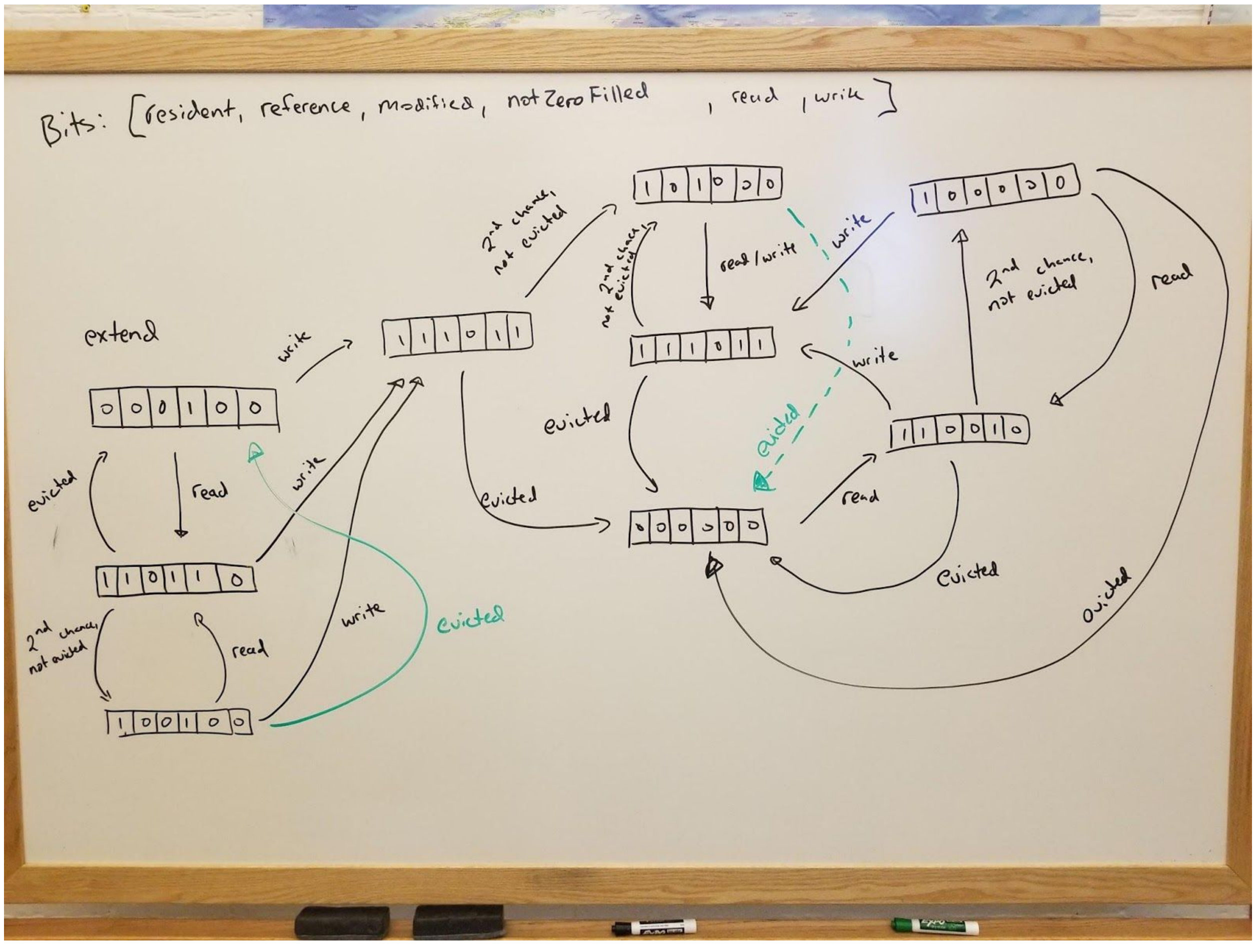

Figure 1: A State diagram showing the various states and transitions a virtual page can go through during its lifetime. We used six bits to keep track of state for each process.

Work deferral is accomplish in two major ways. First, we maintain a bit to keep track of whether an extended page has been zero filled. We only change this bit if a page is modified after extension. This means that an extended page can be read, evicted, and then read again without any disk reads or writes (which are costly). Instead, since we keep track of this page needing to be zero filled, we can just zero fill memory if is read after being evicted. But, if a extended page is written to, we cannot defer work by assuming we know what it contains. This leads to the second bit we use to defer work: the modified bit. This bit is set to 1 if a page has been modified since read from disk. This bit ensures that the pager doesn't initiate unnecessary disk writes when evicting a page--we will only write the page back to the disk if it has been changed. To evict pages, we use the simple version of the second chance clock algorithm; but, if we were to implement the enhanced version, then we would be able to defer even more work (the simple version was in line with the project specification). The reference bit is used for the second chance clock algorithm which simulates least recently used eviction policy. The resident bit is used to keep track of whether a page is resident (since a read and write bit of zero does not necessarily mean that a virtual page is not in physical memory).

To accomplish this work deferral along with the general function of the pager, we followed the following general principles when setting state bits. Since we needed to set either the read or write bit of a page to 0 in order to get a page fault, ensuring proper pager behavior required settings these bits properly. See figure 1 for the general bit settings, but our strategy for setting the read and write bits are as follows:

The bit settings break into 2 broader categories:

1. The page has been extended but not modified.
    a. Write bit: we always set the write bit of extended but not modified pages to zero
    b. Read bit: the read bit of an extended and unmodified page is zero whenever the reference bit of this page is 0 and the read bit is 1 otherwise.
2. The page has been extended and modified since extension.
    a. Write bit:
        i. The write bit is zero if a page is not resident.
        ii. The write bit of a page is zero when a page is resident and has not been modified since being read onto disk.
        iii. The write bit of a page is zero if it is resident and its reference bit is zero.
        iv. The write bit is 1 otherwise.
    b. Read bit
        i. The read bit of a page is 0 when the page is not resident.
        ii. The read bit is 0 when a page is resident and the reference bit is zero.

These rules for setting the read and write bits of virtual pages ensure that faults occur when the pager needs to update state information necessary for work deferral and for managing which virtual pages are in physical memory.

Architecture/Implementation

Second Chance: Second chance simulates least recently used eviction policy by going around the clock structure setting all reference bits that are equal to 1 to zero and setting the read and write enable bits on the page table to zero. When we reach a page that can be evicted we check to see if its modified and if so we write it to disk. We set all of the bits of the page to zero and return the ppage of the evicted physical page.

getPhysPage: First checks to see if there are any available free pages and returns the next free page. If there are no available free pages then it runs second chance and returns the physical page to be used.

Vm_init: Allocate global struct's to the heap and add all free physical memory pages and disk blocks to their respective queues before setting the page_table_base_register to NULL;

vm_create(pid): allocate to the heap a new struct to keep track of information about the current struct. Allocate a new page table. Set the information about the current process and add the current process to the global struct containing a map of all current running processes.

Vm_destroy: first go through the clock structure and remove anything that has the same pid as the current process. Then go through the pages and free their pages used by memory and add them back to the queue of free pages stored in the global struct. Do the same for disk blocks. Next use delete to deallocate the page table and the struct that contained the information about the current process. Next deallocate the current process information that is contained in the struct that contains all the information about all the current process.

Vm_switch: make sure that you are switching to a valid process id and then set the page_table_base_register as a pointer to that pid and return.

Vm_fault: translate the virtual page number from the address and make sure the addr you are faulting is valid. Then check to see if the page is not resident. If the page isn't resident and its is extended not zero filled call a helper program that deals with not zero fille page faults. If the page is zero filled call a function that deals with real page faults. If the page is resident return a function that deals with read / write faults.

normalPagefault: First get a free physical page running the clock algorithm if you have to. Then read from the disk the memory. If fault was on a write the page was not modified and it should not be write enabled. If the page was a write that means it was modified and is write enabled. Set the resident, reference bits and in the page table set the returned physical page in memory to the ppage and the read enable bit to 1. Return and int to declare that the fault was successful.

notZeroFilledPageFault: Get a free physical page from the queue. Zero fill the entire space that is allocated to the virtual address. If the fault was a read set the modified to zero and in the page table set the virtual address write enable to zero. If the fault was a write set the bit to say that it was zero filled and set that it was modified and set the write enable in the page table. After that set the write enable and then in the struct containing information about the page set the bits for resident, reference ppage and then finally in the page table set the ppage of the virtual address. Return an int indicating success.

readWriteFault: if the page is not zero filled. If the fault is a write set the appropriate reference modified, write enable and read enable bits. If the ppage is a read. Set the reference to 1 and the read_enable to 1 then check to see if the page is modified. If it is write enable is 0 else the write enable is 1. If the page is zero filled. If the page is a write modify the zerofilled bit the reference bit and the modified bit. Also set the read enable and write enable bit to 1. If the page is a read only modify the reference bit and the read enable bit. Return an int indicating success.

Vm_extend: If there are no more virtaula dresses in the arena return null. If there are no more free disk blocks return Null. Set the end of the virtual address space and the beginning.

Create a new entry for the page table entry t. Allocate a struct containing information about the page to the heap. Set the virtual page the the ppa:qge number set the not zero filled bit to true and every other bit about the page to zero. Add the page to the map of pagges and info about the pages. Return the address of the beginning of the new valid virtual address space.

Vm_syslog: Get the virtual address. Check to see if the virtual address is outside the arena. Then check to see if the length of the message is inside the arena. For the length the message make sure that the page exists if the page isn't read enabled call fault, otherwise add the information contained at the physical memory address to the output string. Print the output string and return an int indicating success.

<u>Evaluation</u>

We first tested our pager by creating our own test cases. We used our working knowledge of proper pager operation to judge whether our pager was properly handling the test cases. Syslog was useful to judge the correctness of our pager in test cases: printing a string is more intuitively correct than looking at the bit settings of the pages as the pager runs. However, we discovered upon submission to the autograder that a lot of the test cases we thought our pager handled correctly were actually exposing bugs in our pager! To get around this, we created more test cases and found many bugs in functionality, particular with segmentation faults. After 3 days of submitting as we fixed many bugs and did not passing additional test cases, we realized that we must be making errors in the less obvious areas of the pager functionality, particularly the bit settings of the virtual pages. First, we fixed a major conceptual error with how we handled extended and unmodified pages. This change allowed up to pass all but 1 of the autograder tests. We then created specific test cases and found that there were some areas where we still not setting bits correctly (we failed to adequately treat syslog calls as a normal application read of the page). We spent a lot of time looking at our pager output and checking to see that the correct pagers were in physical memory and that things were generally working as expected. In general, this was a difficult lab to evaluate for correctness without first developing a strong understanding of the correct pager functionality--developing this understanding took some time and trial and error!

<u>Conclusion</u>

Overall, this was a fairly involved project that involved some frustration (to the point of writing “smile” on our todo list to maintain better spirits) but that was ultimately rewarding and sometimes fun. A big take-away from this project is the importance of having a correct state diagram before launching into coding. At a certain point, debugging will not always get you closer to the correct solution if you are making conceptional mistakes. We really felt the effects of this when we went from passing 7 test cases to 27 from fixing the way we handled extended but not modified pages. The plus side of this is that we finished much faster than we expected based on autograder feedback; since we fixed almost all of our bugs trying to figure out what we were doing wrong before we re-evaluated our state diagram, once we fixed our state diagram we were almost done. A strategy used by the other groups that we could have taken more advantage of was working on the test cases first. We definitely increased our understanding of how the pager is supposed to work when we created test cases. In closing, this was a challenging and rewarding project that emphasized the importance of locking down the conceptual details of a problem before jumping into writing the implementation.