CSE 314: Nachos Project 3: Virtual Memory (Demand Paging & Page Replacement)

Submission deadline: 12th week friday, 26th 2017 11.55PM

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 Project 2. 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 multiplex 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 Project 2 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.

Demand Paging:

- Implement demand paging. In the first part, you will continue to preallocate a page frame for each virtual page of each newly created process at Exec time, just as in Project 2. 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:
    - In your AddrSpace initialization method, initialize all the PTEs as **invalid**. This will cause the machine to trigger a page fault exception when the process accesses a page.
    - In the same method, set aside 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 content until they are actually referenced by the process.
    - 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.
    - 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 executale file, and a fault on a stack frame should zero-fill the frame.
    - For this part, adapt your executable file loading code from project 2. If the process faults on page 0, for example, then load the first page of code from the executable file into it. More generally, when you handle a page fault you will use the value of the faulting address to determine how to initialize that page: if the faulting address is in the code segment (the NOFF header has the virtual address base and size of the code segment), then you will be loading code; if the address is in the data segment (again use the NOFF header), then load data; if it is any other page, zero-fill it.

    - If you did the process argument extra credit in project 2, the process will fault on these pages 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. Handling these argument pages is optional for this project.
    - 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:
    - 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.
    - If you use ReadMem (or WriteMem) to implement a system call (e.g., when reading the string name argument for Exec), 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.
    - StartProcess and Exec originally closed the executable file after creating the address space. You now will have to keep it open during the life of the process so that you can read code and data pages on demand (add fields to the AddrSpace class to keep a pointer to the open executable file and its NOFF header).

- **Testing**: Start by testing with one process running at a time. During debugging, 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. Then test with multiple processes; this should work automatically since these changes should be independent of the number of processes running.

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.
    - 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.
    - 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 (read from the executable file or zero-filled it for uninitalized data and stack). In this part, allocate a physical page for the virtual page and use your code from part 1 above 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:
    - 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.
    - 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.
    - Read in the contents of the faulted page either from the executable file or from backing store (see below).
    - 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 create a backing store file, write pages from memory to backing store (for page out), and read from backing store to memory (for page in). You can define the class as you like, but here is a suggested interface:
    - ```
      /* Create a backing store file for an AddrSpace */
      BackingStore(AddrSpace *as);

      /* Write the virtual page referenced by pte to the backing store */
      /* Example invocation: PageOut(&machine->pageTable[virtualPage]) or */
      /*                     PageOut(&space->pageTable[virtualPage]) */
      void PageOut(TranslationEntry *pte);
      ```
      ```

```
/* Read the virtual page referenced by pte from the backing store */
void PageIn(TranslationEntry *pte);
```

- 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 set up 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:
  - 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. To be concrete, consider a page used for initialized data. On the first access (fault) on that page, you will read that page in from the executable file. If this page gets evicted, you should write it to the backing store. If the process faults on the page again, you should read the page in from the backing store, *not* from the executable file (the executable file contains the initial version of the page, the backing store contains the most recent version of the page).
  - You are not limited to one file for backing store for the entire system, and you can have a separate backing store file for each process. Doing so makes locating evicted pages convenient (virtual page *n* in the address space will be stored at offset *n * PageSize* in the file), and you can allocate a backing store object for each AddrSpace and store it in a field in AddrSpace. However, do not create backing store files at finer granularities; e.g., do not use one file per page.
  - Be sure to set the dirty bit to FALSE when you mark a PTE for the victim page as invalid.
  - When running multiple processes, you may select a victim page to evict from another process. As a result, you will need to update the PTE in the page table for that process, not the faulting one.
- 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 a particular code or data page. As described above, on the first fault on a page you need to read from the executable file (or zero-fill it in the case of an uninitData page), 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:

- Testing. In this project, programs can use more virtual memory than physical memory, and pages are brought into memory only if they are actually referenced by the user program. To show that you only initialize pages that are referenced on demand, write one test program that references all of the pages in memory, and a second test program that only references some of the pages. You can use the pagein and pageout counters (below) to convince yourself that you are only initializing the pages that are referenced by the process. For example, accessing a two-dimensional array selectively (e.g., all rows, some rows) can give different page reference behavior, and not calling a function will only execute a subset of the code in an executable file. We do not particularly care if the test programs do anything useful (like multiplying matrices), but they should generate memory reference patterns of interest.
- Your test programs should also demonstrate the page replacement policy and correct handling of dirty pages (e.g., allocating backing storage and preserving the correct contents for each page on backing store). Also implement test programs that (1) generate good locality (most references are to a small subset of pages), and (2) generate poor locality (everything is referenced repeatedly in a pattern), and (3) random locality (pages are effectively referenced randomly). If necessary, try reducing the amount of physical memory to ensure more page replacement activity.
- Here is an example test program that is larger than physical memory and uses the console to draw a "snake" wandering around on the screen: snake.c
- [15 pts] Maintain counters of page faults and pagein and pageout events. Print out these counters when Nachos exits. This will aid in debugging, and indicate how well the page replacement algorithm is working with the processes running in the system. You might also want to print out a DEBUG message when each of these events occurs as the process is running (and identify the process that caused the event so that you can disambiguate among multiple executing processes).
- In particular, add two new counters to the Statistics class in machine/stats.h, `numPageOuts` and`numPageIns`, and update the constructor and Print methods in stats.cc to initialize and report the values of those counters (ignore the top of the file where it says to not change the file). Increment the "PageOut" counter whenever you write a page to the backing store (note that "clean" pages should not cause a PageOut). Increment the "PageIn" counter whenever you read a page from (1) the original executable file and (2) your backing store. Print the values of those counters on the same line as the `numPageFaults` counter.
- With backing store implemented, 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 (correctness is paramount), but in this case we want you to experiment with one of the page replacement policies discussed in

class. For project 3, use a simple algorithm to implement page replacement such as FIFO or random replacement. For extra credit, you can implement LRU or an LRU approximation (see below). In any case, document your replacement algorithm choice in the project writeup.

- In your project writeup, create a table that reports the values of the paging counters for all of the test programs that you used to test your page replacement algorithm. If you implemented more than one replacement algorithm, report the values for all of the algorithms that you implemented. Here is an example:
- Physical memory size: 32 pages.

  Page replacement policy: Random.

Program		PageFaults	PageOuts	PageIns
halt	3	0	2	
matmult	112	45	74	
sort	755	624	722	
(more)				

- Note that the exact numbers of page faults and disk I/Os is implementation dependent, so do not be surprised if your implementation results in different values.

Aknowledgements:

http://cseweb.ucsd.edu/classes/fa14/cse120-b/projects/vm.html