Project 5 (Bonus): Virtual Memory

Readings: Nachos Tutorial (Chapter 6)
TA: Qiang Xu (xuq22)
Due date: April 10th, 2015

March 3, 2015

In this project, you will further extend the memory management components in Nachos by implementing a software-based TLB and demand paging of virtual memory.

1 Task Requirements

I. (50%) Implement software-management of the TLB, with software translation via an inverted page table.

Since `VMProcess` extends `UserProcess`, you should not have to duplicate much code from `UserProcess`; that is, only override what's necessary and call methods in the superclass for everything else.

The way to handle TLB misses is to add code to `VMProcess.handleException` which deals with the Processor.exceptionTLBMiss exception. The virtual address causing the TLB miss is obtained by calling `Machine.processor().readRegister(Processor.regBadVaddr)`.

Some methods you will need to use are: `Machine.processor().getTLBSize()` (obtains the size of the processor's TLB), `Machine.processor().readTLBEntry()` (reads a TLB entry), and `Machine.processor().writeTLBEntry()` (writes a TLB entry). Note that TLB entries are of type TranslationEntry, the same class used for page table entries in Part I.

When you run Nachos from the proj3 directory, the processor no longer deals with page tables (as it did in Part I); instead, the processor traps to the OS on a TLB miss. This provides the flexibility to implement inverted page tables without changing anything about the processor simulation.

You will need to do make sure that TLB state is set up properly on a context switch. Most systems simply invalidate all the TLB entries on a context switch, causing entries to be reloaded as the pages are referenced.

Your TLB replacement policy can be random, if you wish. The only requirement is that your policy makes use of all TLB entries (that is, don't simply use a single TLB entry or something weird like that).

You should use a single global inverted page table for all processes. (This is a departure from the use of per-process page tables from Project 4) Because the inverted page table is global, each key used to look up an entry in the table will need to contain both the current process ID and the virtual page number. You may use the standard Java java.util.Hashtable class to implement the inverted page table, but do not depend upon the fact that this class is synchronized to avoid changes being made to it by multiple threads.

Only invalidate TLB entries when it is necessary to do so (e.g. on context switches).

Don't forget to set *used* and *dirty* bits where necessary in `readVirtualMemory` and `writeVirtualMemory`.

II. (50%) Implement demand paging of virtual memory. For this, you will need routines to move a page from disk to memory and from memory to disk. You should use the Nachos stub file system as backing store

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. You should consider using a core map, an array that maps physical page numbers to the virtual pages that are stored there.

The inverted page table must only contain entries for pages that are actually in physical memory. You will need to maintain a separate data structure for locating pages in swap.

The Nachos TLB sets the *dirty* and *used* bits, which you can use to implement the clock algorithm for page replacement. Alternately, you may choose to implement the nth chance clock algorithm as described in the lecture notes (see the textbook for more details on these algorithms).

Your page-replacement policy should not write any pages to the swap file which have not been modified (i.e. for which the *dirty* bit is not set). Also, do not unnecessarily increase the size of the swap file (by having unused space at the end or in the middle of the file). Thus, you will be required to keep pages around in swap even if they have been moved to physical memory.

Now that pages are being moved to and from memory and disk, you need to ensure that one process won't try to move a page while another process is performing some other operation on it (e.g., a `readVirtualMemory` or `writeVirtualMemory` operation, or loading the contents of the page from a disk file). You should not use a separate Lock for every page – this is highly inefficient.

We recommend that you use a single global swap file shared by all processes. You may use any format you wish for this file, but it should be rather simple as long as you keep track of where different virtual pages are stored in the file. You may assume that it's safe to grow the swap file to an arbitrary size; that is, you don't need to be concerned about running out of disk space for this file. (If a read or write operation on the swap file returns fewer bytes than requested, this is a fatal error.) To conserve disk space, you should reuse unallocated swap file space; a simple list of free swap file pages is sufficient for this.

The swap file should be closed and deleted when `VMKernel.terminate()` is called.

If a process experiences an I/O error when accessing the swap file, you should kill the process.

You should test the performance of your page-replacement algorithm by comparing it to a simpler algorithm, such as random replacement. A good way to test this is to write a MIPS C program which does a lot of paging; `test/matmult.c` is a good example. By counting the number of page faults, you can compare the performance of your algorithm with random replacement. We will grade your algorithm in part based on the page fault rate as compared to a simple replacement policy.

Note that it is possible to experience indefinite thrashing when the system has only a single physical page of memory if a process attempts to perform a load/store operation (convince yourself why!). Your implementation need not deal with this case.

2 Submission

You should **commit** your codes through the SVN repository. To avoid unintended changes to your working codes for Project 4, we suggest you to clone your Nachos directory and commit a separate copy of the Nachos source tree (e.g., Nachos2). Please inform the TA which folder to retrieve your code from the SVN repository before the deadline.

Under the nachos project directory, include a file called "report.pdf" details the data structure and algorithms used in your implementation. Grade of the project is divided as 40% for the key data structures/block diagram on the implementation and 60% for the correctness of your implemention with the respective percentage for each task.