Nachos-3: Caching and Virtual Memory

The third phase of Nachos is to investigate the interaction between the TLB, the virtual memory system, and the file system. We don't provide any new virtual memory code for this assignment. You will continue to use the stub file system. For this phase, you should run `gmake` and `nachos` in the `proj3` directory.

To help you to organize your code better, we provide you with a new package, `nachos.vm`, with two new classes, `VMKernel` and `VMProcess`. `VMKernel` extends `UserKernel`, and `VMProcess` extends `UserProcess`. `VMKernel` and `VMProcess` are the only classes you should have to modify for this project phase.

This phase of the project involve open-ended design problems. We will expect you to come up with a design that would make sense in a real system, and to defend your choices in the design review. For example, you will have some freedom to choose how to do software address translation on TLB misses, how to represent the swap partition, how to implement paging, etc.

The first design aspect to consider is the software-managed **translation lookaside buffer (TLB)**. Page tables were used in phase 2 to simplify memory allocation and to isolate failures from one address space from affecting other programs. For this phase, the processor knows **nothing** about page tables. Instead, the processor only deals with a software-managed cache of page table entries, called the TLB. Given a memory address (an instruction to fetch, or data to load or store), the processor first looks in the TLB to determine if the mapping of the virtual page to a physical page is already known. If the translation is in the TLB, the processor uses it directly. If the translation is not in the TLB (a "TLB miss"), the processor causes a trap to the OS kernel. Then it is the kernel's responsibility to load the mapping into the TLB, **the Nachos MIPS simulator does not have direct access to your page tables; it only knows about the TLB.** It is your job to write the code that manages a TLB miss.

The second design aspect of this project is **paging**, which allows physical memory pages to be transferred to and from disk to provide the illusion of an (almost) unlimited physical memory. A TLB miss may require a page to be brought in from disk to satisfy the translation. That is, when a TLB miss fault occurs, the kernel should check its own page table. If the page is not in memory, it should read the page in from disk, set the page table entry to point to the new page, install the page table entry, and resume the 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.

Performance of this mechanism depends greatly on the policy used to decide which pages are kept in memory and which are stored on 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 in memory those pages may be referenced soon. Another consideration is that if the replaced page has been modified, the page must first be saved to disk before the needed page can be brought in. (Of course, if the page has not been modified, it is not necessary to write it back to disk.)

To help you implement virtual memory, each TLB entry contains three status bits: **valid**, **used**, and **dirty**. If the valid bit is set, the virtual page is in memory and the translation can be used directly by the processor. If the valid bit is clear, or if the page translation is not found in the TLB, then the processor traps to the OS to perform the translation. The processor sets the **used** bit in the TLB entry whenever a page is referenced and sets the **dirty** bit whenever the page is modified.

Tasks:

I. 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 project phase 2.
- When you run Nachos from the `proj3` directory, the processor no longer deals with page tables (as it did in phase 2); 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 phase 2.) 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. 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; this will make Part 3 (see below) a lot easier.

- 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 for page replacement.

- 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). 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.

Testing Your Implementation

- Decrease number of physical pages to a very low value. Register a few processes in the kernel which use significantly high number of pages.
  - All the programs should run successfully but may take a long time because of so many page faults.
- How many TLB miss and hits?
- How many page faults?
- What is the distribution of the page faults for a physical page i.e., if there are 10 physical pages, what is the percentage of page faults due to physical page number i?
- Swap file should exist while programs are running. After termination, swap file will be deleted.
- Force any error for a program while it tries to read/write in swap file. It should eventually terminate.

Submission Deadline: Last week