Nachos Assignment #3: Caching and Virtual Memory

Tom Anderson
CSE 451
Due date: Friday, December 10, 5:00 p.m.

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); you can continue to use the "stub" version of the file system.

It is important for you to note in addition to getting virtual memory working, this assignment is also meant to be 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 the freedom to choose how to do software translation on TLB misses, how to represent the swap partition, how to implement paging, etc. In each case, we will expect you to come to the design review armed with a defensible justification as to why your choices are reasonable. You should evaluate your design on all the available criteria: speed of handling a TLB miss, space overhead in memory, minimizing the number of page faults, simplicity, etc. As my advisor used to say, there is no single “right” answer, but there are plenty of wrong ones.

The first part 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 address space. Page tables were used in assignment 2 to simplify memory allocation and to isolate failures from one address space from affecting other programs. For this assignment, the hardware knows nothing about page tables. Instead it only deals with a software-loaded cache of page table entries, called the TLB. On almost all modern processor architectures, a TLB is used to speed address translation. 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 virtual page to physical page is already known. If so (a TLB “hit”), the translation can be done quickly. But if the mapping is not in the TLB (a TLB “miss”), page tables and/or segment tables are used to determine the correct translation. On several architectures, including Nachos and most modern architectures designed for UNIX, a “TLB miss” simply causes a trap to the OS kernel, which does the translation, loads the mapping into the the TLB and re-starts the program. This allows the OS kernel to choose whatever combination of page table, segment table, inverted page table, etc., it needs to do the translation. On systems without software-managed TLB's, 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 TLB's are more flexible, at a cost of being somewhat slower for handling TLB misses. If TLB misses are very infrequent, the performance impact of software managed TLB's can be minimal.

Second, you will need to use memory as a cache for disk, to provide the 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 disk as they are needed. 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 the page is in memory (with the TLB to be loaded with the translation), or the page must be brought in from 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 disk, sets the page table entry to point to the new page, and then resumes 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.

As with any caching system, performance depends on the policy used to decide which things are kept in memory and which are only 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 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.

For this assignment, you will need three data structures: (1) some way of translating in software from virtual page frames to physical page frames (hint: consider using a hash table -- note that we provide one in “lib”), (2) some way of translating from physical page frames back to virtual page frames (so that when you replace a page, you can invalidate its translation), and (3) some way of finding a page on disk if it is not in memory (hint: consider using a file per segment).

1. Implement software-management of the TLB, with software translation via 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. 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 a context switch. Most systems simply invalidate all the TLB entries on a context switch; the entries get re-loaded as the pages are referenced.

2. Implement 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 file system as backing store; this will make part 3 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. (Hint: Traditional UNIX uses something called a "core map" -- that maps physical page numbers to the virtual pages that are stored there -- for this purpose.)

The Nachos TLB sets and clears dirty and use bits, which you can use to implement the clock algorithm; alternatively, you can use the faults on TLB misses to keep a second chance list, as in VAX VMS.

Demonstrate the value of your paging algorithm by writing some programs that page, but page less frequently under your approach than say, using random replacement.

3. Implement memory-mapped files. The traditional way to access the file system is via Read and Write system calls, but the problem is that requires an extra level of copying between the kernel and the user level. A different interface is simply to "map" the file into the virtual address space; the program can then use load and store instructions directly on the file data. (An alternative way of viewing the file system is as “durable memory”; files just store data structures. If you access data structures in memory using load and store instructions, why not access data structures in files the same way?)

For this part, you need to implement an "mmap" system call, that takes a open file descriptor, and a location in the address space to put the file. You can assume the application programmer will pick an address aligned on a page boundary, that it will be in some unused part of the address space (in other words, won't overlap with any other mmap calls), and that the file has a fixed size that doesn’t grow. From part 2, you already have the mechanism to bring missing pages of the mapped file into memory -- in fact, you can use the same mechanism to bring in code and data pages from the executable on program startup.) Modifications to the file are committed to disk when the file is closed.

Note that the consequence of supporting mmap is that address spaces will be sparsely populated with (potentially lots of) segments, one per open file. For parts 1 and 2, you should design your software translation mechanisms with this in mind.

Implement a utility program (such as UNIX “cp”) that exploits the new interface.