Nachos Assignment 2: TLB and Virtual Memory

Sanzheng Qiao
Department of Computing and Software

October 22, 2007

Due: November 6, Tuesday, 11:59 pm.

In this assignment, we first solve the problem of memory management for multiprogramming. That is to allow multiple processes share the memory. Nachos uses pure paging. To support multiprogramming, we can no longer always load a program into contiguous pages. Thus we must load one page at a time without assuming linear mapping between virtual page numbers and physical page numbers. Note that the way the file system sees the executable file is different from the way the kernel sees the virtual space. The file system sees the executable as a stream of bytes stored on disk starting from the infile address 0, whereas the kernel sees the virtual space starting with the instruction at virtual address 0. Moreover, from programmer's view, an executable file is organized into segments, while in memory the file is divided into fixed size pages.

Also, in this assignment, we investigate the use of caching. We use caching for two purposes. First, we use a software-managed translation lookaside buffer (TLB) as a cache for page tables to provide the illusion of fast access to virtual page translation over a large address space. Second, we 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. We provide no new code for this assignment (the only change is that you need to compile with the "-DVM -DUSE_TLB" flags); your job is to write the code to manage the TLB and to implement virtual memory.

Page tables are used in Nachos 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, the DEC MIPS and the HP Snakes, a "TLB miss" simply causes a trap to the OS kernel, which does the translation, loads the mapping into 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.

The illusion of unlimited memory is provided by the operating system by using main memory as a cache for the disk. 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, a software page table 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.

This assignment is the following three items:

1. Multiprogramming and TLB.
   Find a way of loading one page at a time from disk into memory. You cannot assume linear mapping between virtual page numbers and physical page numbers. For this part, you implement the following function:

   ```
   LoadSeg(OpenFile *executable, int vAddr, int size, int fAddr)
   ```

   This function loads a segment (code or data), one page at a time, given the segment's virtual address, size, and infile address given by the header of the object file. Make you solution short and readable.
   Introduce the TLB into our system. Now that we have the TLB, the machine knows only the TLB but not the page table and the page fault exception during the address translation is really a TLB miss. When a TLB miss occurs, the exception handler checks the page table and copies an entry from the page table into the TLB. You must find a way of mapping the page table entries into the TLB entries, for example, direct mapping (1-way associative), 2-way associative, full-way associative, etc. For this part, you implement the following function:

   ```
   TLBMissHandler(unsigned int badVA)
   ```

   This function handles TLB miss given the virtual address which caused the TLB miss. In this item we assume:

   - the executable can fit into the memory, so we don't have to swap out pages during execution (not real virtual memory yet);
   - the executable is loaded into the memory when the address space is constructed (no real page fault).

2. Demand paging.
   In this part, we remove the second assumption listed above, that is, we implement demand paging. When we start a process, instead of loading the whole program, we mark all the entries in the page table invalid. So, when we start to run the program, we find that the requested page is not in the memory (a real page fault). For this, you will need a routine to move a page from disk to memory. For this step, we still keep the first assumption. That is no page replacement. Note, however, that the process space contains more than the object file. For example, it contains the stack, which is not a part of the executable. The stack is zero filled when the process is started, whereas the pages containing the code and data are loaded from the disk. So, you need status for each papge (for example: ZEROFILL, LOADFROMOBJ). For this item, you implement the following function:

```
PageFaultHandler(unsigned int badVA)
```

This function handles page fault given the virtual address which caused the page fault.

3. Virtual memory.
   In this part, we remove the first assumption listed in the first item, that is, we implement virtual memory. We create an illusion of an (almost) unlimited memory. For this, you will need a mechanism of replacing pages. So, you need a swap file for swapping pages, since some pages, for example, stack pages, are not in the object file. To test your code, run programs that require more space than the Nachos memory.