CMPSCI 377 Operating Systems **Fall 2013**

Lecture 13: October 16

Lecturer: Prashant Shenoy *Scribe: Armand Halbert*

13.1 Quick Review: TLB

A translation look-aside buffer, or TLB, is a fast, fully associative, memory that maps page to frame mappings. It is much smaller, but also much faster, than RAM. The TLB utilizes the 90/10 rule and the current mappings that are actually being used tend to fit within the TLB. It should also be noted that during a context switch the TLB should be saved in the PCB (process control block) and the TLB for the process that will be run next should be restored.

13.1.1 Costs of Using the TLB

The performance of using a TLB is measured using the hit ratio. A TLB hit is when the page that needs to be access is in the TLB. The opposite case is a TLB miss and requires the page mapping to be retrieved from the full page table that is stored in RAM. The relative costs are given below as formulas, but first we have to define some terminology:

- *ema*: effective memory access, the actual cost of retrieving something from memory (including accessing the page table).
- *ma*: memory access, the cost of accessing something in memory given a physical address.
- p: probability of a TLB hit.
- t: cost of a TLB access.

Without a TLB: $ema = 2 * ma$.
With a TLB: $ema = (ma + t) * p + (2 * ma + t) * (1 - p)$. Note that as p approaches 1, the $ema = (m + t)$.

13.2 Sharing

Paging allows sharing of memory across processes since the memory used by a process no longer needs to be contiguous. Different processes simply need to point to the same region in memory. This can also be used for sharing library code. This shared code must be reentrant, that means the processes that are using it cannot change it (e.g. no data in reentrant code). Sharing of pages is similar to the way threads share text and memory with each other. The shared page may exist in different parts of the virtual address space of each process, but the virtual addresses map to the same physical address. The OS keeps track of available reentrant code in memory and reuses them if a new process requests the same program. Sharing of pages across processes can greatly reduce overall memory requirements for commonly used applications.

13.3 Segmented Paging

In a pure paging based system like that described in the previous lecture, an application's virtual address space is divided up into equally sized pages. However, these page sized chunks do not match the logical way in which an ordinary process would be broken up—programmers think of a process's memory space being divided into regions for code, global variables, the stack, and the heap for dynamic data structures. *Segmentation* tries to match the programmer's view by dividing the address space of a process into multiple variable-length segments, one for each of the categories just described. This can simplify how addresses are created. For example, it may make sense to address an array by using an offset from the start of the global variables segment. If paging were used instead of segmentation, the address would need to be determined using an offset from the start of the process's entire address space.

To implement segmentation, a virtual address must consist of a segment number and an offset within the segment. Memory is still physically addressed with a single number. To obtain it, the processor looks up the segment number in a segment table to find a segment descriptor. The segment descriptor contains information such as (i) a flag indicating whether the segment is present in main memory, (ii) the address in main memory of the beginning of the segment (segment's base address) and (iii) the length of the segment. To translate an address, the OS verifies that the segment is present and that the offset is less than the segment length. If a segment is not present in main memory, a hardware interrupt is raised to the operating system, which may try to read the segment into main memory.

It is possible to combine segmentation and paging by dividing each segment into pages. In systems that combine them, virtual memory is usually implemented with paging, with segmentation used to provide memory protection. The virtual address space is treated as a collection of segments of arbitrary sizes, and the physical memory is treated as a sequence of fixed size page frames. A segment usually spans many pages, and these pages within a segment are mapped onto actual physical page frames. There may be a segment corresponding to each logical view of a process - heap segment, code segment, data segment, stack segment etc. This allows protection or sharing mechanisms to be applied at the granularity of segments, rather than individually for each page.

13.3.1 Addresses in a Segmented Paging System

While in a pure paging system each virtual address was divided into two portions, p and d, for the page number and offset respectively, implementing a segmented paging system requires an additional set of bits, s to be used to represent the segment for a memory address. Thus each virtual address consists of a segment number, a page number within that segment, and an offset within that page. The segment number indexes into the segment table which yields the base address of the page table for that segment. The remainder of the address, page number and offset, is checked against the limit of the segment. A trap is generated if this limit is violated. Otherwise, the page number is used to index the page table. The entry in the page table is the page frame. Finally, the physical address is obtained by adding the frame address and the offset.

For example, consider a system with memory size of 256 addressable words, a page table with 8 entries, a page size of 32 words, and 8 logical segments for each process. In this system, a physical address requires 8 bits total ($2^8 = 256$). In a pure paging system, the length of a physical address was always the same as the length of a virtual address; with segmented paging this may not be the case. For this example, 3 bits will be needed to index the page table ($2^3 = 8$), 3 bits will be needed for the segment, and 5 bits will be needed for the offset ($2^5 = 32$). This is a total of 11 bits for each virtual address, but only 8 bits for each physical one. Segmented paging adds a small amount of overhead to addresses due to the extra bits needed to specify the segment; however it provides greater convenience by creating more logically understandable addresses and improving sharing controls.

13.3.2 Sharing & Protecting Pages and Segments

In a naive system, two processes that are run from the executable would both load into memory identical segments for the code of the application. To eliminate this redundancy, it is desirable to let applications easily share their code segments with other processes in a safe (read only) way. This can be easily done by having two processes point to the same segment in their segment tables. This is much simpler than requiring each individual page to be explicitly shared. If needed, sharing can also be done on a page-by-page basis by copying page table entries between two processes.

Segmentation also supports protection and valid bits for both segments and pages. This can be used to force regions of memory to be read-only. Other bits can be used to specify whether a segment is "executable"; this can be used to specify whether a region contains code that the processor should be able to execute. Disabling the executable bit in segments such as the stack and the heap can be used to prevent viruses and malicious attacks which attempt to inject code into the system by overrunning buffers.

13.3.3 Segmentation Implementation

A segment table could be implemented in registers or in RAM. In practice, the segment and page tables are typically stored in RAM, and then the TLB is used to cache both types of tables. This can provide fast accesses at much lower cost than registers, assuming the code has some access locality.

In modern systems with large amounts of RAM, it can become necessary to do hierarchical levels of paging and segmentation. This can be used to control the growth of page tables for applications. When using single-level page tables, the full page table must be allocated immediately at startup with sufficient address space for the full RAM size of the system. In contrast, using hierarchical page tables can allow a process to be created with only a small page table initially; once that page table begins to fill up, some of its entries can be used to point to other page tables containing the other pages used by the process. This technique can be used for both page tables and segment tables. The *ema* cost, however, does grow as the number of levels in a page or segment table grows (worst case being $(N+1)m$ for N levels.

13.3.4 Costs and Benefits of Segmented Paging

Segmentation can improve process startup time since it is clear which segments need to be loaded first in order to begin running code. Segments also allow for simpler growth because it can be done per segment. Segmentation also allows for efficient coarse and fine grain sharing of pages and segments. On the downside, segmentation leads to slower address translation due to multilevel addressing, which can reduce performance. Context switches can also be more expensive in segmented systems since both the page table and segment table need to be saved during each context switch.

13.3.5 Inverted Page Tables

Inverted Page Tables are a technique that can be used to scale to even larger address spaces than can be efficiently handled using multi-level page tables. An inverted index changes page tables to be a key-value store. To translate a page, the page number is used as a key that is looked up in the store in order to find the corresponding frame number. The use of efficient hash tables can allow this to scale to very large systems, however each lookup may be slow. Fortunately, the TLB can still be used to cache address translations, reducing the performance overhead.