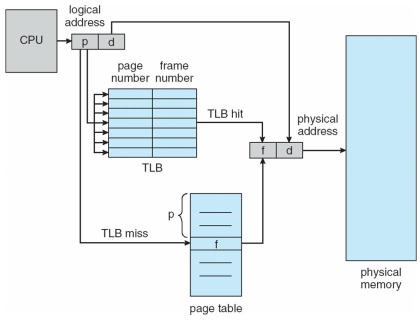


Virtual Memory Manager

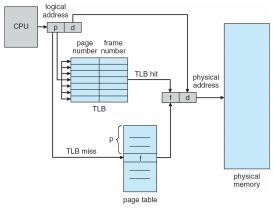
Prepared for: TC2008 Class

Prepared by: Abelardo López Lagunas,Ph.D.

October 6, 2014

Version number: 1.1

Table of Contents



Description

Objective

The goal of this project is to implement a simulator for a virtual memory manager. The simulator reads a file with memory references, called a *memory trace*, and then proceeds to simulate the memory access in a system that consists of a small Translation LookAhed Buffer (TLB), a one-level page table, and a block of memory. The simulator reports the total number of TLB misses, page faults in the form of disk reads and writes, total number of memory frames used, and the *average access time* for the entire trace. To verify you read all events in the trace it is recommended that you also keep track of the total number of events (memory accesses) in the trace.

Goals

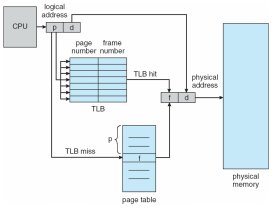
The implementation of the virtual memory simulator will test your comprehension of the interactions between elements in a simple memory hierarchy, as well as verify that the student is capable of using basic data structures, file management, and general programming methodologies.

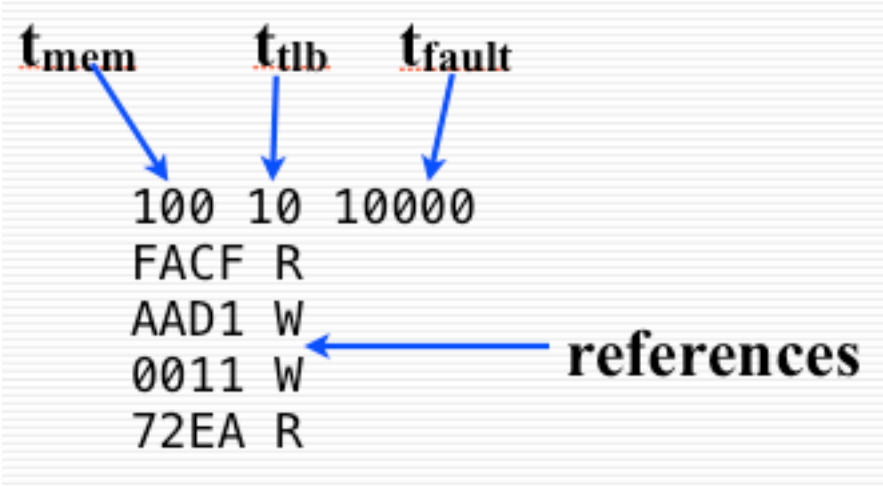
Problem Statement

You are required to implement a *virtual memory simulator* for a system with 16-bit virtual addresses (64Kbytes total) and 8Kbytes of physical memory. Furthermore, assume that the pages and frames are 512-bytes each. To reduce the access time the system has an 8-entry TLB. The TLB uses the Least Recently Used (LRU) replacement policy to evict entries. To simplify your implementation you can use a single-level page table for the virtual to physical address translation. Given the previous conditions your system will have 128 entries in the page table and a total of 16 frames in memory.

Your simulator must keep track of what pages are loaded into memory. As it processes each memory event from the trace, it should check if the reference is in the TLB and also check if the corresponding page is in memory or not. If not, it should choose a frame to remove from memory. Of course, if the frame to be replaced is dirty, it must be saved to disk. Finally, the new frame is loaded into memory from disk, and the page table is updated.

Your program will take as input a *text* file *with* an **arbitrary** number of memory references. The first line in the file has the main memory access time ($\mathbf{t_{mem}}$), the TLB access time ($\mathbf{t_{tlb}}$), and the page fault penalty ($\mathbf{t_{fault}}$). All times are in nano seconds. After the first line, each line will have a 16-bit address followed by the access type (read or write). As an example consider the following four memory references:



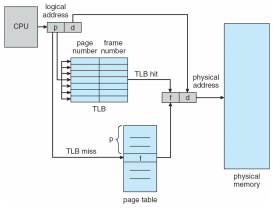


Note that your program should read three *positive* integers followed by a newline character and then for each line you should read a 16-bit address followed by a space and then the access type which is either **R** (read) or **W** (write).

The trace file that will be provided to you has 1,000,000 virtual memory references and for each reference you must perform the translation from virtual to physical memory and depending on whether the reference is in the TLB or the page table you will keep track of the average access time for each reference.

Note that when you are evicting an entry in the TLB or from a frame in memory you need to check if the entry is *dirty* (i.e. the reference was a **write**). If it's **dirty** you need to simulate a memory **write** (writes the dirty page into disk), followed by a memory **read** (reads the new page from disk into memory). Note that the TLB has a *subset* of the memory references contained in the frame table, simplifying your coding significantly[1].

[1] If you do not understand why this is the case re-read the virtual memory management section in the class notes



Expected outputs

The simulator reports the TLB hit rate, the page faults in the form of disk reads and writes, total number of memory frames used, and the *average access time* for the entire trace. Note that the final trace is over 1,000,000 entries long so you will be provided with a smaller trace for testing purposes

Proposed solution

Summary

Although you do not need to use dynamic memory for this project, it would be better if the implementation uses *malloc* to allocate the memory for the TLB, Frame table, and Page Table. Note that the implementation of the Least Recently Used algorithm is the same for both the TBL and Frame Table.

Finally, although the problem has defined the size of the virtual addresses, physical addresses, and TLB your code should be written in such a way that it would be trivial to increase or decrease their sizes.