|  |
| --- |
| PROJECT REPORT FOR VIRTUAL MEMORY MANAGEMENT |
|  | DATE: 26 APR 2018 |
|  | COURSE: COMP 3500 |
|  | AUBGID: DZT0021 |
|  |  |
|  | >> See README.md for More information and High-level Analysis << |
|  |  |
|  | -- SOLUTION DESCRIPTION -- |
|  | (1) How did you guarantee that each logical address is translated to the correct physical address? |
|  |  |
|  | Each logical address was bit-masked to extract the rightmost 16-bits. The following function prototypes were created and implemented to extract their values (declared in vmtypes.h and implemented in vmtypes.c): |
|  |  |
|  | // 32-Bit masking function to extract page number |
|  | int getPageNumber(int mask, int value, int shift); |
|  |  |
|  | // 32-Bit masking function to extract page offset |
|  | int getOffset(int mask, int value); |
|  |  |
|  | Once the page number and offset have been extracted from the logical address, their values are immediately stored and not manipulated. Once we find the frame number associated with the page, we then combine the frame number and offset to extract the physical address. Although the values associated with their variables change, the values are not modified more than once during each address translation. This ensures there are minimal errors and that such errors related to translation MUST be the result of some other factors (i.e. Memory leaks or otherwise). |
|  |  |
|  | Since we do not know the physical address beforehand, we can not be entirely certain of the translation's accuracy. However, this implementation produced no obvious errors or inconsistencies after multiple executions. |
|  |  |
|  | The physical address itself is retrieved using the left shift with the OR operator ((frame\_number << SHIFT) | offset\_number). This operation is used only once and for printing the results to the console. |
|  |  |
|  | (2) How did implement the page table, physical memory, and TLB? |
|  |  |
|  | Since the Page Table and TLB contains the same data, this implementation defined a Virtual Memory Addressing table that could be represented as either a TLB or Page Table. The type is defined as follows: |
|  |  |
|  | typedef struct vmTable\_t { |
|  | int \*pageNumArr; // page number array |
|  | int \*frameNumArr; // frame number array for this |
|  | int \*entryAgeArr; // Age of each index |
|  | int length; |
|  | int pageFaultCount; |
|  | int tlbHitCount; |
|  | int tlbMissCount; |
|  | } vmTable\_t; |
|  |  |
|  | It can be seen that this implementation uses dynamically allocated arrays such that the program would utilize storage on the heap and scale appropriately, depending on our parameters. |
|  |  |
|  | The physical memory itself is implemented as a 2D dynamically allocated array. These data structures are created by functions that are elaborated in file "vmtypes.c". |
|  |  |
|  | Below are their declarations as they appear in vm\_sim.c: |
|  |  |
|  | vmTable\_t\* tlbTable; // The TLB Structure |
|  | vmTable\_t\* pageTable; // The Page Table |
|  | int\*\* dram; // Physical Memory |
|  |  |
|  | (3) Does your program realistically and accurately simulate a virtual memory system? |
|  |  |
|  | Because this simulation uses solely software and algorithmic computations to address an abstraction for storage resources, it does not "realistically" simulate a virtual memory system. An actual MMU consists of H/W components which we simulate by creating data structures in memory. Furthermore, based on our current design specifications, it does not incorporate a replacement strategy for physical memory or the page table. Our design has enough space on the page table to accommodate the number of frames in simulated physical memory. It does, however, accurately show how resources are utilized to map memory addresses used by a program, into physical addresses in CPU memory. Also, this implementation allows main storage, as seen by a process or task, to appear as contiguous address space. |
|  |  |
|  |  |
|  | (4) Did you use the Java operators for bit-masking and bit-shifting? |
|  |  |
|  | Yes. Their implementation can be seen in the following two functions (as displayed in vmtypes.c): |
|  |  |
|  | /\* |
|  | 32-Bit masking function to extract page number |
|  | This function assumes a high order page number and |
|  | a low order page offset |
|  | @Param {mask} The int masking value we will use to perform AND operation w.r.t. value |
|  | @Param {value} The int value we wish to mask |
|  | @Param {shift} The relative number of bits we want to shift right after the bitwise operation |
|  | @Return {int} The int representation for Page Number |
|  |  |
|  | \*/ |
|  | int getPageNumber(int mask, int value, int shift) { |
|  | return ((value & mask)>>shift); |
|  | } |
|  |  |
|  | /\* |
|  | 32-Bit masking function to extract physical memory offset |
|  | This function assumes a high order page number and |
|  | a low order page offset |
|  | @Param {mask} The int masking value we will use to perform AND operation w.r.t. value |
|  | @Param {value} The int value we wish to mask |
|  | @Return {int} The int representation for physical memory offset |
|  |  |
|  | \*/ |
|  | int getOffset(int mask, int value) { |
|  | return value & mask; |
|  | } |
|  |  |
|  | (5) When a TLB miss occurs, how do you decide which entry to replace? |
|  |  |
|  | The following function prototypes define that behavior: |
|  |  |
|  | // Function Prototypes |
|  | void translateAddress(); |
|  | void readFromStore(int pageNumber); |
|  | void tlbFIFOinsert(int pageNumber, int frameNumber); |
|  | void tlbLRUinsert(int pageNumber, int frameNumber); |
|  | int getOldestEntry(int tlbSize); |
|  |  |
|  | The user determines which algorithm they would like to execute a TLB insert. Each function has several comments detailing each logical step. For FIFO, we insert entries into their respective fields until the TLB can no longer fit newer entries. We then overwrite the last entry in the TLB with our newest entry and then programmatically shift the fields of every value in the table such that the first entry that made its way into the buffer is no longer in the table. For LRU, we enter in each new entry and then when the table is full we increment the age counter for all entries. The oldest entry gets replaced. If there are entries which have the same age, the first one is selected. |
|  |  |
|  | -- GENERALITY AND PERFORMANCE CRITERIA -- |
|  | (1) How general is your solution? |
|  |  |
|  | The solution is very general. All that needs to be modified are constants which define the system parameters for the virtual memory. These constants are defined as follows: |
|  |  |
|  | #define FRAME\_SIZE 256 // Size of each frame |
|  | #define TOTAL\_FRAME\_COUNT 256 // Total number of frames in physical memory |
|  | #define PAGE\_MASK 0xFF00 // Masks everything but the page number |
|  | #define OFFSET\_MASK 0xFF // Masks everything but the offset |
|  | #define SHIFT 8 // Amount to shift when bitmasking |
|  | #define TLB\_SIZE 16 // size of the TLB |
|  | #define PAGE\_TABLE\_SIZE 256 // size of the page table |
|  | #define MAX\_ADDR\_LEN 10 // The number of characters to read for each line from input file. |
|  | #define PAGE\_READ\_SIZE 256 // Number of bytes to read |
|  |  |
|  | There is no need to modify other values, as the program will execute on these values as long as the user is aware of its application. |
|  |  |
|  | (2) How easy would it be to change parameters such as the size of the TLB? |
|  |  |
|  | Very easy. See last question. |
|  |  |
|  | (3) Does your program only load pages from the backing store when they are needed? |
|  |  |
|  | Yes, there is absolutely NO reason to access the BACKING\_STORE if a frame number is found to be in the TLB or Page Table. It would be highly inefficient to access the backing store in such a case. The function translateAddress() handles whether or not backing store access is required. Additionally, this system implements a function that calculates the average time complexity for accessing the backing store. With the current input size, it takes an average of ~5 milliseconds to read from the backing store. However, as secondary memory and virtual memory size may vary, this time could have significant non-linear growth. Regardless of the time it takes, it still incurs a time penalty that would otherwise be unnecessary if we accessed it during every address translation. |
|  |  |
|  | (4) Does your solution allow the physical address space to be smaller than the virtual address space? |
|  |  |
|  | This implementation does allow for the physical address space to be smaller, due to the fact we use dynamic memory allocation. As a result, we always allocate the right amount of memory to hold the working set. The same applies for the page table and TLB despite the fact they are generally fixed-size. In any event, virtual memory is typically larger than physical memory - there wouldn't be much reason for virtual memory mappings if virtual memory and physical memory were the same size [source](https://stackoverflow.com/a/14347298). Furthermore, the user will define the size of their desired physical and virtual memory, as specified by the constants. |

* © 2019 GitHub, Inc.