Assignment 03

Part II Programming (120 points)

Pair-programming up to two in a group, or work alone

Due: Beginning of the class, **Mar 22nd**

Demand Paging with Multilevel Page table and TLB

Demand Paging is a virtual memory management scheme that would only load and map pages from persistent storage to memory when needed, i.e., when a process tries to access a page that is not residing in the physical memory (which would result in a page fault). The OS will then try to allocate the physical frame, load, and map the page to the allocated physical frame as part of the page fault handling.

In this assignment, you will write a simulation of demand paging using a multi-level page tree/table with address translation caching (Translation lookaside buffer - TLB).

A **32-bit** virtual address space is assumed. The user will indicate how many bits are to be used for each of the page table levels, the size of the TLB cache (max number of page mappings in TLB cache), and a user-specified file containing hexadecimal addresses will be used to simulate virtual/logical address accesses and construct a page table tree.

Functionality

Upon start, your program creates an empty page table (only the level 0 node should be allocated). The program should read logical / virtual addresses one at a time from an input trace file. The trace file consists of memory reference traces for simulating a series of access attempts to logical / virtual addresses.

For each virtual/logical address read in, simulate the memory management unit (MMU) for translating virtual address to physical address as well as the Operating System demand paging process (See Figure 1 below):

1) Extract the full virtual page number (VPN), search the TLB then the page tree (in the case of a TLB miss) to find the Virtual Page Number (VPN) → Physical Frame Number (PFN) mapping information.
2) If the VPN → PFN mapping entry is found in TLB, use the found PFN for translation.
3) If the VPN → PFN mapping entry is NOT found in TLB (TLB miss), walk the page table tree:
    a. if the mapping is found in the page table, insert the mapping to the TLB cache; if the TLB is full, need to apply the cache replacement using the approximation of the LRU (Least Recently Used) policy (see TLB specification)

   b. if the mapping is not in the page table, insert the page to the page tree/table with an assigned frame index (starting at 0 and continue sequentially) that simulates the demand paging allocation of a physical frame to the virtual page brought into memory (**important**: you will assume an infinite number of frames are available and need not worry about a page replacement algorithm).
      i. after the demanding paging brings in the page and inserts the VPN → PFN mapping to the page table, also inserts the mapping to the TLB cache; if the TLB is full, need to apply the cache replacement (again, see TLB specification).
4) In all cases described in 2) and 3), update or insert the recently accessed pages which is used for cache replacement (**must** follow the TLB specification for updating the recently accessed pages).
5) If the page mapping is found in the TLB cache, increment the cache hit counter; if the page mapping is NOT found in the TLB (a TLB miss) and then the page mapping is found in the page table, increment a page table hit counter.
6) Note: TLB cache hits + page table hits + page table misses = total number of address accesses.
7) Print appropriate outputs to the standard output as specified below in "User Interface"

Important - Allocating (assigning) a frame number (PFN) to a VPN:

You use a sequential number for the PFN starting from 0.

At the beginning, there is nothing mapped in the page table. You read in first virtual address from the trace file, you extract the VPN information from that virtual address and insert the VPN to the multi-level page table (using the multi-level numbers of bits from the command line).

At the leaf level / node, you would assign (i.e., map) the frame 0 to that first VPN. Then increment the frame number to 1 to be assigned to the next VPN inserted to the page table. And you do this process for each new VPN inserted to the page table.

You are recommended to implement the multi-level paging without TLB first, then add the TLB simulation. Many autograding tests are testing the multi-level paging without TLB.

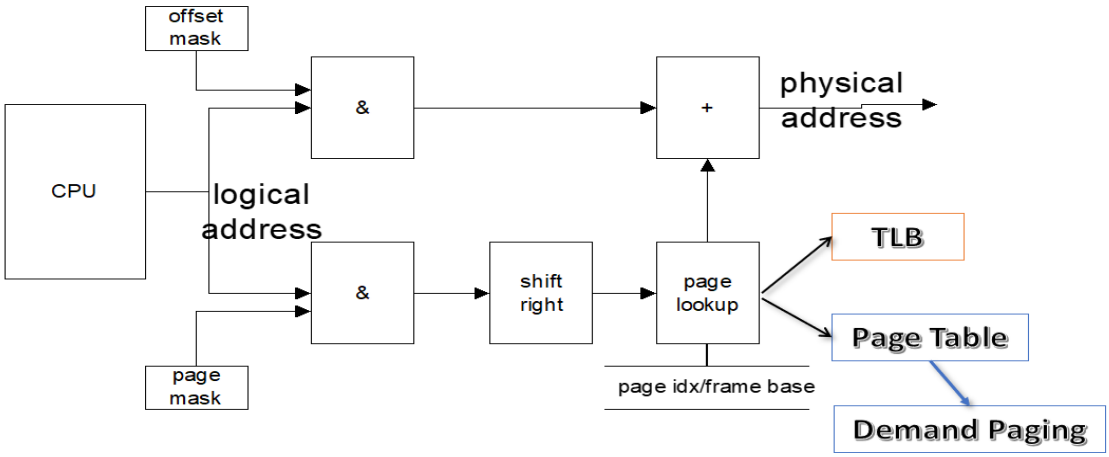

Figure 1. MMU Translation from Virtual / Logical address to Physical address

User Interface

When invoked, your simulator should accept the following optional arguments and have two or more mandatory arguments. **Implementing this interface correctly is critical to earning points.** Several functions are provided to help you put output in the correct format and are listed next to each of the output modes. These are located in output_mode_helpers.c, which will compile in C or C++. The function to use for each output mode is listed in parentheses after the mode explanation, see the function comments in the source code for details.

Do **NOT** implement your own output functions, autograding is strictly dependent on the output formats from the provided output functions.

Optional arguments:

-n N — Process only the first N memory accesses / references. Processes all addresses if not present.

-c N — Cache capacity of the TLB, i.e., max number of page mapping entries (N) in TLB. Default is **0** if not specified, meaning NO TLB caching.
Important:
- If an out-of-range number (< 0) is specified,
  print to the standard error output (stderr):
  Cache capacity must be a number, greater than or equal to 0
  then exit.

-o mode — output mode. Mode is a string that specifies what to be printed to the standard output:

bitmasks – Write out the bitmasks for each level starting with the highest level, one per line. In this mode, you do not need to actually process any addresses. Program prints bitmasks and exits. (Use **report_bitmasks**.)

virtual2physical – Show virtual address translation to physical address for every address, one address translation per line. (Use **report_virtual2physical**.)

v2p_tlb_pt - Show virtual to physical translation for every address, lookup TLB then pagetable walk if TLB misses, one address translation per line. (Use **report_v2pUsingTLB_PTwalk**.)

vpn2pfn – For every virtual address, show its virtual page numbers for each level followed by the frame number, one address per line. (Use **report_pagemap**.)

offset – Show offsets of virtual addresses, one address offset per line. (Use **hexnum**.)

summary – Show summary statistics. This is the default argument if -o is not specified. (Use **report_summary**.) Statistics reported include the page size, number of addresses processed, hit and miss rates for tlb and pagetable walk, number of frames allocated, total bytes required for page table (hint: use sizeof). You should get a roughly accurate estimate of the total bytes used for the page table including data used in all page tree levels. Note your calculated number may not match the number of total bytes in sample_output.txt (should be close though), as you may not have strictly the same data members in your structures as in the solution code, which is fine. But you should be aware that in general, with more paging levels, less total bytes would normally be used.

Mandatory arguments:

- The first mandatory argument is the name of the trace file consisting of memory reference traces for simulating a series of attempts of accessing virtual / logical addresses.
    - **trace.tr** is given for your testing.
    - Auto-grading on gradescope will use all or **part** of **trace.tr**.
    - Appropriate error handling should be present if the file is not existent or cannot be opened. It must print to standard error output (stderr) the following error messages:
      Unable to open <<trace.tr>>

- The traces were collected from a Pentium II running Windows 2000 and are courtesy of the Brigham Young University Trace Distribution Center. The files *tracereader.h* and *tracereader.c* implement a small program to read and print trace files. You can include these files in your compilation and use the functionality to process the tracefile. The file trace.tr is a sample of the trace of an executing process. See an example of reading trace file in *a3_programming_misctips.pdf*.

- The remaining mandatory arguments are the number of bits to be used for each level, **important**:
  - number of bits for any level MUST be greater than or equal to **1**; if not, print to the standard error output (stderr) with a line feed, e.g., level 0 has 0 bit specified, it should print:
    Level 0 page table must be at least 1 bit
    then exit.
  - total number of bits from all levels should be less than or equal to **28** (<= 28); if not, print to the standard error output (stderr):
    Too many bits used in page tables

Sample invocations (note these not necessarily will be used for the autograding test cases):

./pagingwithtlb trace.tr 8 12

Constructs a 2 level page table with 8 bits for level 0, and 12 bits for level 1. The remaining 12 bits would be for the offset in each page.

This invocation does not simulate TLB (as -c is not specified).

Process addresses from the entire file (as -n is not specified) and output the summary (as -o is not specified).

./pagingwithtlb -n 10000 -o *vpn2pfn* trace.tr 8 7 4

Constructs a 3 level page table with 8 bits for level 0, 7 bits for level 1, and 4 bits for level 2.
The remaining 13 bits would be for the offset in each page.

This invocation does not simulate TLB (as -c is not specified).

Processes the first 10,000 memory references from the file, and output the mapping of virtual page number to frame number for each address, one mapping for each address per line.

./pagingwithtlb -n 4800 -c 32 -o *v2p_tlb_pt* trace.tr 8 6 10

Constructs a 3 level page table with 8 bits for level 0, 6 bits for level 1, and 10 bits for level 2.
The remaining 8 bits would be for the offset in each page.

Simulates a TLB with 32 page mappings.

Processes the first 4,800 memory references from the file, and output virtual to physical address translation for every address, one address translation per line, with the tlb hit/miss, pagetable hit/miss information from the TLB and pagetable walk for finding the mapping information.

Additional examples and correct outputs can be seen in file sample_output.txt.

Compilation and execution

- You must create and submit a Makefile for compiling your source code. Refer to the Programming page in Canvas for makefile help.
- The make file must create the executable with a name as **pagingwithtlb.**
- Your code **must compile and run correctly on edoras.sdsu.edu**, regardless of where you wrote, compiled, and tested your program.

Code structure

You are **required** to **separate** the logic of page table operations, TLB operations, and the main flow of the functionality to separate code files, although you are certainly encouraged to use more if you see appropriate.

Autograding

About 3/5 test cases would be for testing the page table **without** TLB, the remaining test cases would be for testing page table **with** the TLB.

Turning In

For each pair programming group, submit the following artifacts by **ONLY ONE** group member in your group through **Gradescope**. Make sure you use the **Group Submission** feature in Gradescope submission to **add your partner** to the submission.

Make sure that all files mentioned below (Source code files, Makefile, Affidavit) contain each team member's name and Red ID!

- Program Artifacts
  - Source code files (.h, .hpp, .cpp, .C, or .cc files, etc.), Makefile
  - Do NOT **compress / zip** files into a ZIP file and submit, submit all files separately.
  - Do NOT submit any .o files or test files
- Academic Honesty Affidavit (no digital signature is required, type all student names and their Red IDs as signature)

- Pair programming Equitable Participation and Honesty Affidavit with the members' names listed on it. Use the pair-programmer affidavit template.
- If you worked alone on the assignment, use the single-programmer affidavit template.

Programming references

Please refer to Canvas Module "Course Resources – Programming" (particularly the FAQ) for coding help related to:

- Process command line arguments
- How can I have two classes or structures point to one another? (PageTable class and Level class point to each other)
- Code structure – what goes into a C/C++ header file
- Many other tips for c/c++ programming in Unix / Linux

Grading

Passing 100% auto-grading may NOT give you a perfect score on the assignment. The structure, coding style, and commenting will also be part of the rubrics for the final grade (**see Syllabus Course Design - assignments**). Your code shall follow industry best practices:

- Be sure to comment your code appropriately. Code with no or minimal comments are automatically lowered one grade category.
- Design and implement clean interfaces between modules.
- NO global variables.
- NO hard code – Magic numbers, etc.
- Have proper code structure between .h and .c / .cpp files, do not #include .cpp files.

Academic honesty

Posting this assignment to any online learning platform and asking for help is considered academic dishonesty and will be reported.

An automated program structure comparison algorithm will be used to detect code plagiarism.

- The plagiarism detection generates similarity reports of your code with your peers as well as from online sources. We will also include solutions from the popular learning platforms (such as Chegg, etc.) as part of the online sources used for the plagiarism similarity detection. Note not only the plagiarism detection checks for matching content, but also it checks the structure of your code.
- Refer to Syllabus for penalties on plagiarisms.
- Note the provided source code snippets for illustrating shared data structure would be excluded in the plagiarism check.