Assignment 03

Part II Programming (110 points)

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

Due: Beginning of the class, **Mar 26th**

Late Due: Beginning of the class, **Mar 28th.**

Demand Paging with Multilevel Page Table and Page Replacement

Demand Paging is a virtual memory management mechanism 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 find a free (unoccupied) physical frame, load, and map the page to the free physical frame as part of the page fault handling. When there is no free frame for allocation, a page replacement algorithm is used to select an existent mapped page frame to evict and bring in the page being accessed and map it to the freed frame.

In this assignment, you will write a simulation of demand paging using a multi-level page table with page replacement if necessary.

A **32-bit** virtual address space is assumed. The simulation will specify:

- a trace file with hexadecimal addresses used to simulate virtual/logical address accesses and construct a page table tree,
- number of bits used for each level of the multi-level page table,
- parameters for page replacement:
  - the number of available physical frames (optional)
  - number of memory accesses for the bitstring update interval for NFU with aging page replacement algorithm (optional)
- logging options (optional)
- see user interface section below for more details.

Functionality

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

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

1) Extract the virtual page number (VPN) based on paging levels, walk the page table tree to find the Virtual Page Number (VPN) → Physical Frame Number (PFN) mapping information.
2) If the VPN → PFN mapping entry is found, use the found PFN for translation.
3) If the VPN → PFN mapping entry is NOT found:
    a. If there is still free physical frame available:
        i. insert the page to the page table tree and map its VPN to the next available frame index (starts from 0 and continues sequentially) that simulates the demand paging allocation of a new physical frame (PFN) to the virtual page loaded into that frame.
    b. Else:
        i. Use a modified **NFU (not frequently used) with aging** algorithm (see a3specs.pdf) to find a victim frame to evict and map the VPN of the page being accessed to the victim frame as the PFN.
            i. **Remember**: **Update** the mapping of the replaced VPN (victim) to PFN in the page table to be **invalid**.
    c. Use PFN to **translate** the virtual address to physical address.
4) **In all cases** described in 2) and 3), **update or insert** to the page recent access history for potential later page replacement, see **a3specs.pdf**.
5) Print appropriate logs to the standard output as specified below in "User Interface".

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

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

At the beginning, there is nothing mapped in the page table. You simulate a memory access by reading in the first virtual address from the trace file, extract the VPN information from the virtual address (using the numbers of bits for the levels from the command line) and insert the VPN to the multi-level page table. At the **leaf** level / node, you would assign (or allocate) the physical frame 0 to the first VPN, then increment the frame number to 1 to be assigned (or allocated) to the next VPN inserted to the page table. Do this process for each new VPN inserted to the page table until all available physical frames are used, after which, if a virtual address with an unallocated VPN is accessed, use a modified **NFU with aging page replacement** algorithm to evict an existing mapped frame and assign the freed frame to the VPN being accessed (see a3specs.pdf). Note an evicted virtual page could be accessed again later, which would trigger page replacement to evict another frame/page.

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

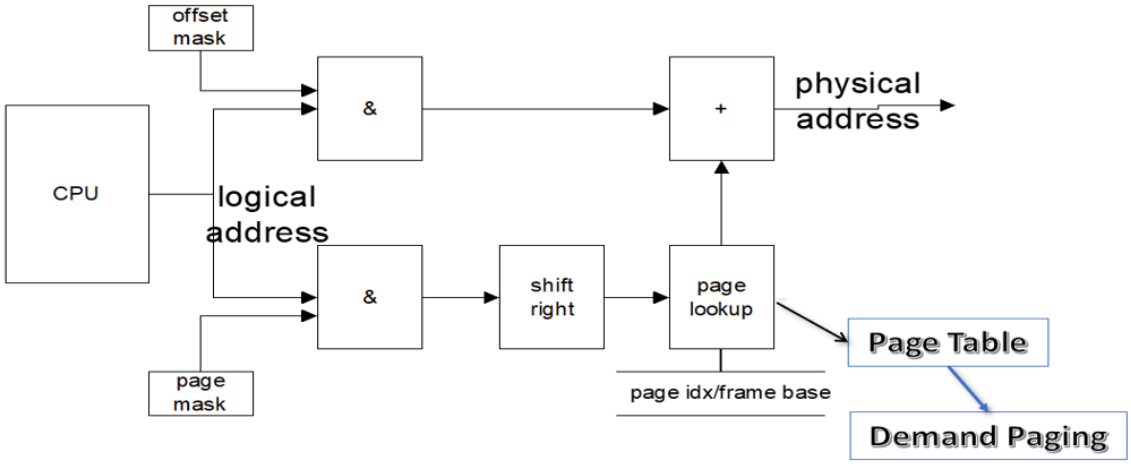
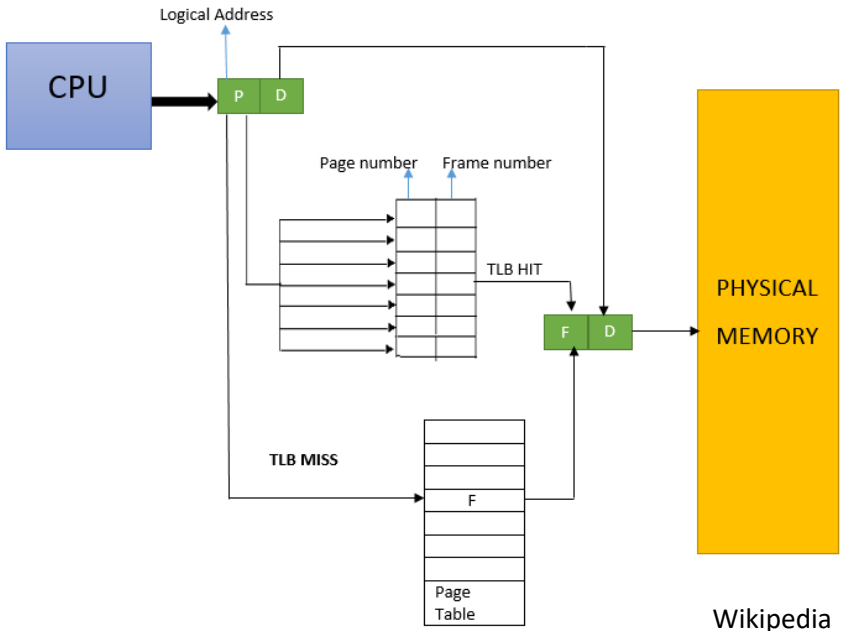


Figure 1. MMU Translation from Virtual / Logical address to Physical address



Wikipedia

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 print output in the correct format and are listed next to each of the log modes. These are in **log_helpers.h**, and **log_helpers.c** (for c++, change log_helpers.c to log_helpers.cpp), 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.

Important: 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.

Error handling:

- If an out-of-range number (< 1) is specified, print to the standard output in one line (or standard error stderr): *Number of memory accesses must be a number and greater than 0* then exit.

-f N — Number of available physical frames. If this argument is specified, page replacements may be simulated. **The default** is **999999** if not specified, simulating an infinite number of frames that no page replacement is necessary.

Error handling:

- If an out-of-range number (< 1) is specified, print to the standard output in one line (or standard error stderr): *Number of available frames must be a number and greater than 0* then exit.

-b N — number of memory accesses between bitstring updates, this is for simulating the bitstring update interval for NFU with aging, i.e., how often (every few memory accesses) the bitstring should be updated. **Default** is **10** if not specified.

Error handling:

- If an out-of-range number (< 1) is specified, print to the standard output in one line (or standard error stderr): *Bit string update interval must be a number and greater than 0* then exit.
- Note: do not worry about the upper bound of this parameter, you can assume autograder testing will **NOT** test an unreasonable big number for this argument like bigger than 2000 for example.

-l mode — Log 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 lowest tree level (root node is at level 0), one per line. In this mode, you do not need to process any addresses. The program prints bitmasks and exits. (Use the given **log_bitmasks** function.)

va2pa – Print virtual address translation to physical address for every address, one address translation per line. (Use the given **log_va2pa** function.)

vpns_pfn – For every virtual address, print its virtual page numbers for each level followed by the frame number, one address per line. (Use the given **log_vpns_pfn** function.)

vpn2pfn_pr – For every address, print vpn, pfn, replaced victim's vpn and bitstring if page replacement happened, and page hit or miss, one address translation per line. (Use the given **log_mapping** function.)

offset – Print page offsets of virtual addresses, one address offset per line. (Use the given **print_num_inHex** function.)

summary – Print summary statistics. This is the default argument if -l is not specified. (Use the given **log_summary** function.) Statistics reported include the page size, number of addresses processed, hit and miss rates for pagetable walk, number of page replacements, 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 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) or standard output 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 *vaddr_tracereader.h* and ***vaddr_tracereader.c*** (for c++, change the file name to ***vaddr_tracereader.cpp***) implement a small program to read and print virtual address 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 ***a3progtips.pdf***.

- The remaining mandatory arguments are for the number of bits to be used for each page table level, starting from level 0, then level 1, so on and so forth:

- number of bits for any level MUST be greater than or equal to **1**; if not, print to the standard error output (stderr) or standard output with a line feed, e.g., if level 0 has 0 bit specified, it should print:
  Level 0 page table must have 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) or standard output with a line feed:
  Too many bits used for the page table
  then exit.

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 **pagingwithpr.**
- You are strongly recommended to set up your local development environment using a Linux OS (e.g., Ubuntu 20.04 or 22.04, or CentOS 8) to develop and test your code. You could also use Edoras (with CentOS Linux) to develop and test there. The gradescope autograder uses Ubuntu to compile and autograde your code.

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

./pagingwithpr trace.tr 12 8

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

This invocation does not simulate page replacement (as -f is not specified).

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

./pagingwithpr -n 1000 -l *vpns_pfn* trace.tr 6 8 8

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

This invocation does not simulate page replacement (as -f is not specified).

Processes the first 1,000 memory references from the file. With -l vpns_pfn, prints the mapping of virtual page numbers of all levels to frame number for each address, for each address per line.

./pagingwithpr -n 3500 -f 50 -l *vpn2pfn_pr* 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 page replacement with 50 available frames. NFU bitstring update interval (number of memory accesses between bitstring updates) would use the default 10 value (as -b is not specified).

Processes the first 3,500 memory references from the file. With -l vpn2pfn_pr, prints vpn, mapped pfn, page hit or miss, replaced pfn if page replacement happened, for every address per line.

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

Autograding

Again, many test cases (at least half of them) would be for testing the page table **without page replacement**, the remaining test cases would be for testing page table **with** the page replacement.

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 executable, object code (.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 group programmer affidavit template.
  - If you worked alone on the assignment, use the single programmer affidavit template.
- **Number of submissions**
  - The autograder counts the number of your submissions when you submit to Gradescope. For this assignment, you will be allowed a total of **10** submissions. As stressed in the class, you are supposed to do the testing in your own dev environment instead of using the autograder for testing your code. It is also the responsibility of you as the programmer to sort out the test cases based on the requirement specifications instead of relying on the autograder to give the test cases.
  - **Note as a group**, you would **ONLY have a total of 10 submissions** for the group. If the two members of a group submit separately and the total submissions together between

the two members exceed 10, a **30%** penalty on your overall a3 grade will be applied. This will be verified during grading.

Programming references

Please refer to ***a3progtips.pdf*** and refer to Canvas Module "Programming Resources" (particularly the FAQ) for coding help related to:

- Process command line arguments.
- How can I have two classes or structures point to one another? (if you choose to have 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 on your code appropriately. Code with no or minimal comments are automatically lowered one grade category.
    - Each **function in the header** should have **a comment section** above the function signature to give a brief description of the function, the inputs and output of the function.
- Design and implement clean interfaces between modules.
- Meaningful variable names.
- 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.

- Plagiarism detection generates **similarity reports of your code with your peer's code as well as code from online sources and submissions from past semesters**. It would be purely based on similarity check, two submissions being like each other could be due to both copied from the same source, whichever that source is, even the two students did NOT copy each other.
- We will also include solutions from the popular learning platforms (such as Chegg, github, etc.) as well as code from **ChatGPT** (see below) as part of the online sources used for plagiarism similarity detection. Note not only does the plagiarism detection check for matching content, but also it checks the structure of your code.
- **If plagiarism is found in your code**, your grade will automatically be zero for the assignment, it will also be reported following the SDSU academic dishonesty reporting

procedure. Further disciplinary action could be applied that can result in a course fail grade.

- Note the provided source code snippets or examples would be ignored in the plagiarism check.

SDSU's Center for Student Rights and Responsibilities have officially added plagiarism policies of using ChatGPT for academic work, as put below:

- "Use of ChatGPT or similar entities [to represent human-authored work] is considered academic dishonesty and is a violation of the Student Code of Conduct. Students who utilize this technology will be referred to the Center for Student Rights and Responsibilities and will face student conduct consequences up to, and including, suspension."