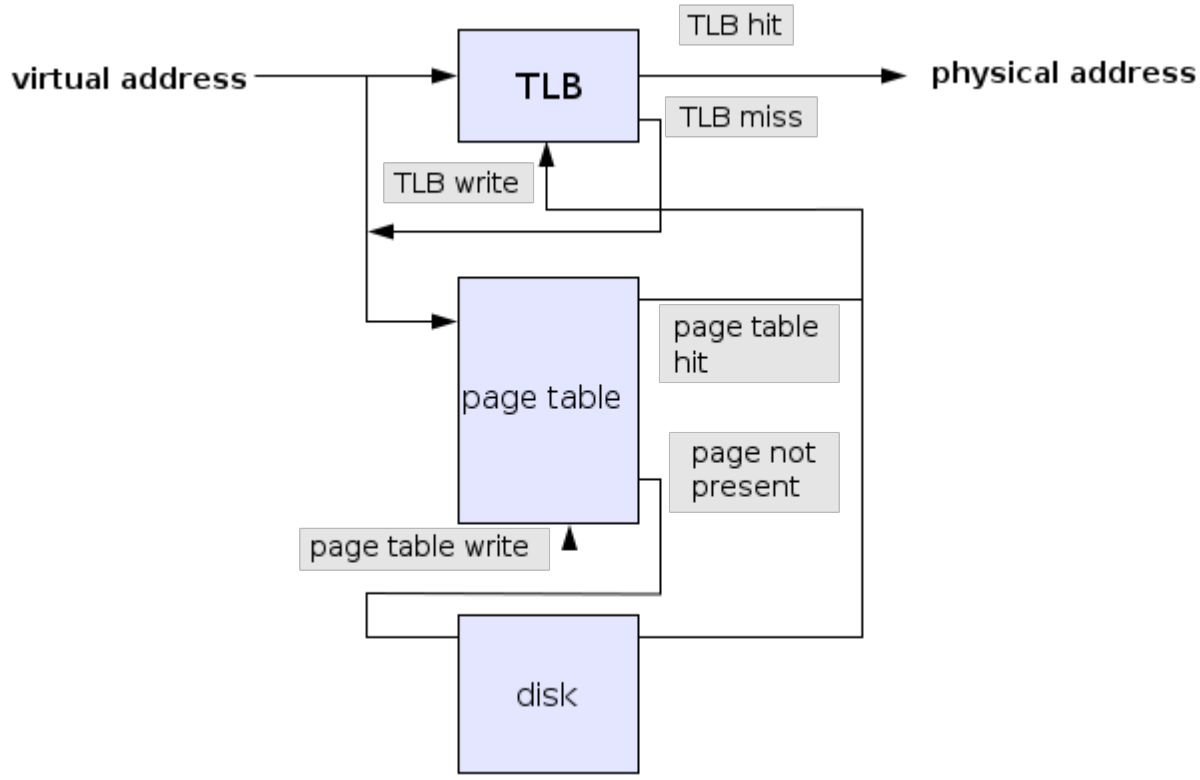
Program 3
Last updated: Feb 21, 2023
This assignment **may** be done in groups

Virtual Memory Simulator (memSim)

This assignment is about material covered in the Virtual Memory chapters (13-23) in the OSTEP book. You are well advised to read ahead to get started on this project. The project is *similar* to a project in the SGG (Dinosaur) book, *with some differences*. A copy of these pages is provided in Canvas. Read those pages and make sure you understand the background and diagrams, but keep in mind the modifications listed.



Designing a Virtual Memory Manager

As the problem in the book states, you have to translate logical memory to physical and simulate the operations of the main memory structures needed. Below are the units needed to accomplish this. We only care about read operations for this assignment. No need to implement writes.

- **TLB.** The Translation Lookaside Buffer is like a small cache for memory translation. An

entry in the TLB consists of a logical memory page number and its corresponding physical memory frame number. In this assignment, the TLB gets populated on a first-in-first-out (FIFO) basis, meaning the slot populated earliest, is the one that should be evicted first[1]. You will implement a TLB with 16 entries.

- **Page Table.** The page table is a much larger structure that keeps track of every page of a process, including those currently loaded and not currently loaded in main memory. It provides the same page to frame translation as the TLB. When there is a TLB miss, that induces a lookup of the page number in the page table. If the entry in the page table is not valid, we incur a page fault. Which means, we have to load the page from the backing store and then update both the page table and TLB. In this assignment the page size is 256 bytes.
- **Backing store**. This is where all pages can be originally found. This will be given to you in the form of a file called ***BACKING_STORE.bin*** (see Canvas) of length 65,536 bytes (64K), addressable by 256 byte blocks. Note: You should test with your own backing store as well as an example one provided. It will not be the same one you are graded on.
- **Physical memory.** This is where the actual content of main memory. It is also addressable in 256 byte frames. Most often there is not enough frames to store all the logically addressable pages for a process. Thus, we have to manage (using a page-replacement algorithm) which pages are loaded into which frames. In this assignment, the size of the physical memory (given as the number of frames) is passed into memSim via the command-line.
- **Reference sequence.** This is a sequence of logical address requests. A text file, containing ASCII integers (one per line), will be given on the command line. Each line is a logical address that needs to be translated to a physical address.
    - **See *addresses.txt* and *addresses_output.txt* for example input and output, based on the provided backing store file.**

Modifications to the book version

In the book version, the size of the physical memory is the same as the size of the logical memory. Therefore, the only kinds of page faults we can expect are the kind that result from empty pages. Once all pages are loaded into physical memory, there is no need ever to load any more pages or evacuate existing pages. That's boring. In this version, we will work with a variable physical memory size (in terms of frames). And, because it is possible to have a smaller physical memory, we will need to use a page replacement algorithm. Both of these will be passed in as command line arguments (See Executable section, below).

The specifications for this assignment are as follows:

- 2^8 entries in the page table (same as the book's version)
- 16 entries in the TLB (same as book)

[1] If a page already exists in the TLB (a TLB hit), the entry should be moved to the end of the TLB eviction queue.

- Page size = frame size = block size = 256 bytes (same as the book)
- Variable number of frames (`FRAMES` given as command line argument)
- Support for multiple page replacement algorithms (given as command line argument)
- Physical memory of size in bytes is 256*`FRAMES`
- Page table is to have a “present” bit associated with it. If the bit is set, it means that page is currently valid, and is present in a physical memory frame.
- Your executable must be called **`memSim`**
- The output of the executable will have these components, printed to standard out.
    - For every address in the given addresses file, print one line of comma-separated fields, consisting of
        - The full address (from the reference file)
        - The value of the byte referenced (1 signed integer)
        - The physical memory frame number (one positive integer)
        - The content of the entire frame (256 bytes in hex ASCII characters, no spaces in between)
        - new line character
    - Total number of page faults and a % page fault rate
    - Total number of TLB hits, misses and % TLB hit rate

Executable

The usage of the main executable is this:

```
memSim <reference-sequence-file.txt> <FRAMES> <PRA>
```

- `reference-sequence-file.txt` contains the list of logical memory addresses
- `FRAMES` is an integer <= 256 and > 0, which is the number of frames in the system. Note that a `FRAMES` number of 256 means logical and physical memory are of the same size.
- `PRA` is “FIFO” or “LRU” or “OPT,” representing first-in first-out, least recently used, and optimal algorithms, respectively.
- `FRAMES` and `PRA` or just `PRA` may be omitted. If so, use 256 for FRAMES and FIFO for PRA.

Grading

For this assignment, I will run multiple tests. You are welcome to complete only some of the tests for the given number of points shown. For full credit, you must pass all the tests. For any credit, tests must pass on the CSL infrastructure (e.g. unix2.csc.calpoly.edu).

- Implementation without any modifications from the book. (*i.e.* hard-coded 256 frames, and no replacement algorithm): 10 points
- Support for a variable number of frames and FIFO page replacement algorithm: 4 points

- Support for LRU approximate (last used) page replacement algorithms: 4 points
- Support for OPT (optimal) page replacement algorithm: 2 points

Extra Points (Optional – Must Complete Required First)

By popular request, you may implement *optional* additional features/deliverables. **NOTE:** In order to receive extra credit, you must first make a credible attempt at completing all of the normal assignment deliverables listed above. For example, you may not skip LRU or OPT support and instead work on extra point options. Once this is satisfied, you may do any combination of the below:

- Support for Large Pages: 4 points
    - Large pages will be 4 frames in size (1024 bytes)
    - Use of large pages will be enabled via "1024" as a fourth argument to the program (which is usable only when both FRAMES and PRA arguments are given)
    - To receive full points, you must:
        - Implement the command-line argument
        - Implement large pages
        - Create two of your own address.txt files, one with a sequential memory access pattern and the other with a random access pattern.
        - **In the README** explain what effect large pages had (compared to normal size) for the input files you created.
- Visualize memory access patterns: 4 points
    - In class we will look at heatmaps showing memory access patterns. For example, a sequential memory scan looks like a diagonal lines, with addresses increasing as time passes.
    - Implement a visualization that takes in a file of the same format as memSim. It may be part of your memSim program, or a separate program.
    - Your visualization may be implemented using the same programming language you used for this assignment, or using Excel or R/RStudio (free, and commonly used for this kind of analysis).
    - To receive full points, you must:
        - Submit an image (chart, graph, etc.) based on the addresses.txt file provided with this assignment.
        - Submit the code used to create the visualization – though it may be constructed such that it only runs in your environment (it does not need to run on unix2)
        - **In the README** explain how the visualization was constructed, what it shows, and how this would be useful in understanding the behavior of a program and/or memory system.
- Implement a bad page replacement algorithm: 2 points
    - A "bad" algorithm will be worse than FIFO/LRU/OPT as demonstrated by one of the summary metrics output at the end of a run.
    - Your algorithm will be triggered using a <PRA> argument of **BAD**
    - To receive full points, you must:

- Implement your algorithm such that it is not simply ignoring or shortcutting the existence of memory or TLB memory frames (i.e. don't just evict frames immediately, or flush the TLB after each request). Let's see your creativity!
- **In the README,** describe your algorithm, and any observations you have about its effects on sequential and random memory access patterns.

Deliverables

Submit a gzip'd tar archive file with all your source code (no binaries). It must include the following:

1. Your source and header file(s).
2. A makefile (called Makefile) that will build the **memSim** executable. (This can be skipped for Python executables.)
3. A README file that contains:
    - Your name
    - Any special instructions.
    - Any other thing you want me to know while I am grading it.
    - The README file should be plain text, i.e, not a Word document, and should be named "README", all capitals with no extension.