Project #3
Virtual Memory and Paging

Assigned: Nov 20, 2023 Document version: 1.1
Due date: Dec 4, 2023

- This project will be done in groups of two students. You can do it individually as well. You will program in C/Linux. Programs will be tested in Ubuntu 22.04 Linux 64-bit.
- **Objectives/keywords**: Practice with paging, virtual memory, physical memory, virtual addresses, physical addresses, address translation, page replacement algorithms, single-level and two-level paging, backing store, swap space, random file I/O.

1. Project Specification (80 pts)

In this project you will develop a C program, a simulator, that will simulate the memory accesses and the paging behaviour of a simulated program. The simulated program will be defined by two files. One file will contain the memory references the program makes. The second file will contain the swapped out virtual pages of the program. This second file will act as the backing store. It will be like a swap space on a disk where all pages of a program resides.

The size of the virtual memory will be 64 KB (2^{16} bytes). We will assume that all pages in the virtual memory are defined and legal to access by the simulated program. You simulator will be called `memsim` and will be invoked with the following command line arguments.

memsim `-p` `level` `-r` `addrfile` `-s` `swapfile` `-f` `fcount` `-a` `algo` `-t` `tick` `-o` `outfile`

- The `-p` `level` option specifies the number of levels in the **page table**. It can be 1 or 2. If 1, it means single-level paging is used. If 2, two-level paging is used.
- The `-r` `addrfile` option is used to specify the name of a file that contains memory references, i.e., virtual addresses, made by the simulated program. Your simulator will read this file and will translate the virtual addresses in it to physical addresses and will perform the indicated read/write memory operations on the related pages in physical memory. The file is an ascii text file. There is no limit on the size of the file (i.e., on the number of memory references it includes).
- The `-s` `swapfile` specifies the name of a binary file that will store the pages (content) of the simulated program on disk. This file will act as the **backing**

store, i.e., swap space for the simulated program. The size of this file is fixed and is equal to the virtual memory size. It will contain 1024 pages in order: page 0, page 1, ..., page 1023. The size of each page is 64 bytes. If this file does not exist, it will be first created and initialized to all zeros. Then it will be opened (for reading and writing). A page that does not exist in memory will be loaded from this file when it is referenced.

- The `-f` `fcount` option specifies the number of frames that will be used by the simulated program. The minimum value of `fcount` can be 4, and the maximum value can be 128. Let F denote the value of the `fcount` argument. Then the size of the simulated physical memory will be $F \cdot PAGESIZE$. $PAGESIZE$ is 64 bytes. You will allocate that much space in your program with `malloc()` to act as physical memory. The range of physical addresses will be $[0, F \cdot PAGESIZE)$. We are assuming the whole physical memory (F frames) is used by the simulated program.
- The `-a` `algo` option specifies the **page replacement algorithm** to use. It can be one of the following strings: `FIFO`, `LRU`, `CLOCK`, `ECLOCK`. If `algo` is set to `ECLOCK`, then your program will simulate the Enhanced Clock algorithm that we have seen in the class (uses both M and R bits). If `algo` is set to `LRU`, then your program will simulate an exact LRU algorithm.
- The `-t` `tick` option specifies the timer tick period (in number of memory references done). After every `tick` many memory references, the R bits of the pages are cleared (in the page table entries).
- The `-o` `outfile` is used to specify the name of the output file that will be generated by your simulator and that will include information about virtual addresses, their translations, and page faults.

A **virtual address** (VA) will be 16 bits long (2 bytes). The offset part is the least significant 6 bits. In single-level paging, the remaining 10 bits will be used as virtual page number (VPN). In two-level paging, the remaining bits are further divided into two parts, P1 and P2, each 5 bits long. Hence address division scheme in two-level paging will be $[5, 5, 6]$.

A **physical address** (PA) will be 16 bits long, no matter what F (frame count) is.

A **page table entry** (PTE), no matter what kind of a page table it belongs to, will be 16 bits long (2 bytes). Of these 16 bits, k least significant bits (LSB) will be used as frame number, where $2^k = F$. Of the remaining bits, one bit will be used as R bit (Referenced bit), and one bit will be used as M bit (Modified/Dirty) bit, and one bit will be used as Validation bit (valid/invalid). Exactly which bits in a page table entry will be used as M, R, and Validation bits is up to you.

Initially all frames are empty. That means initially no pages are loaded into (physical memory) from the backing store yet. Pure **demand paging** is used. Pages are loaded when referenced. We are assuming that all pages in the backing store (in the virtual memory) are legal pages to access (we are assuming that they contain something useful for the simulated program, even though they may be all zeros).

Frames are allocated in increasing order with respect to their numbers. That means the smallest frame that is empty will be allocated to a new page that is to be loaded. If there is no empty frame, then page replacement will take place. In that case, a page (frame) will be selected as victim according to the page replacement algorithm. Then the victim page will be removed. If it is modified (M bit is 1), it has to be written back to the backing store before removal. After making a frame empty in this way, the page that caused the page fault is loaded from the swap file into the frame. The page table will be updated accordingly.

If single-level paging is used, the simulated program will start with a single page table where all entries are initialized to be invalid. If two-level paging is used, the program will start with only the top level page table. Again all entries in the top level table will be initially invalid. An inner table will be created when a related page is referenced.

Your program will generate an output file that will give information about the referenced virtual addresses and their translations. If a reference caused a page fault, this will also be indicated with the string `pgfault`. Each line of the output file will include information about a single virtual memory reference and its translation. The format of a line in the output file will be as follows.

VA PTE1 PTE2 OFFSET PFN PA PF

VA is a virtual address. PA is the corresponding physical address. PF is either a SPACE character or the string `pgfault`. PTE1 is the index of the top-level (outer) page table entry used in translation. PTE2 is the index of the second-level (inner) page table entry used in translation. OFFSET is the offset in the page, that corresponds to the virtual address. If single-level paging is used, then PTE2 will always be zero (0) and PTE1 will be the index of the single-level page table entry used in translation. PFN is the physical frame number corresponding to the page number (virtual address) that is translated. Example output lines are as follows. All numbers are hexadecimal.

```
0x0001 0x0 0x0 0x1 0x1 0x0041 pgfault
0x0001 0x0 0x0 0x1 0x1 0x0041
```

At the end of the output file, you will also write the total number of page faults happened.

The address file will contain the memory references (virtual addresses) made by the simulated program. It will have one memory reference per line. The line will include if the memory access is a read (r) or a write (w) operation. If the operation is a write operation, then the line will also include the value to write to the specified virtual address. A value will be one byte long, that means it can be an integer in range $[0, 255]$ (decimal). An example address file is shown below. Each byte of the virtual memory has a virtual address. In a line, `r` means read (a byte) operation, and `w` means write (a byte) operation. All numbers are hexadecimal.

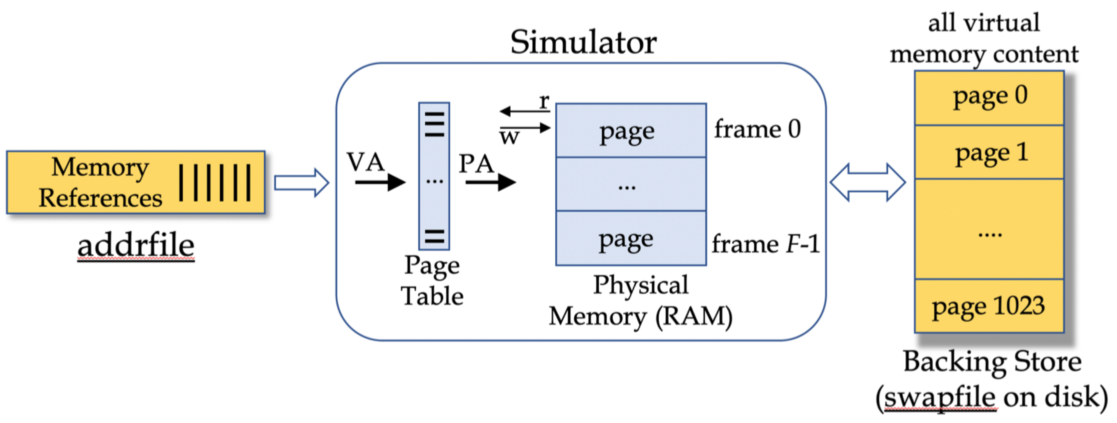

Figure 1: Diagram showing the run-time structure of the simulator program.

```
r 0x0000
r 0x0001
w 0x0000 0x5
r 0x0000
w 0x0001 0xff
r 0x0010
r 0x3d27
w 0x3d27
r 0x3d22
```

You program will read and process the address file one line at a time. After reading a line from the file, it will translate the respective virtual address to a physical address. This may cause a page fault. In this case the page fault needs to be handled. After the desired page is brought into memory, the related physical address will be accessed and the R bit of the page will be set to 1 (in the respective PTE). If the access type is w (write), then the specified byte value will be written to the respective offset in the page in physical memory and the M bit (dirty bit) will be set accordingly. The write operation will over-write the existing value in that offset in the page. After completing the operation indicated in the line, your program will also write an information line to the output. What exactly will be written is specified earlier.

After finishing processing the input file, your program will write (flush) the pages in physical memory to their respective locations in the swap file (backing store). In this way we will have an up-to-date version of the backing store. Then your program can terminate.

At the end of a simulation, we will have two files to check for correctness: 1) the output text file, 2) the backing store file (a binary file). We can check the binary file by using the `hexdump` utility program.

Example invocations of the simulator are shown below. All numbers are decimal.

```
./memsim -p 1 -r refs.txt -s bstore.bin -f 16 -a LRU -t 100 -o out1.txt
./memsim -p 2 -r refs.txt -s bstore.bin -f 16 -a LRU -t 100 -o o2.txt
./memsim -p 2 -r addr.txt -s swap.bin -f 8 -a ECLOCK -t 128 -o out.txt
```

2. Experiments (20 pts)

Design and conduct some experiments to observe the number of page faults for various algorithms and frame counts. For this you will need a large address file. You can write a simple program that will generate and put a large number of virtual addresses (memory references) into the address file. Plot your results in one or more graphs or put them into one or more tables. Try to interpret your results.

3. Submission

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both students will be written, separated by a dash '–'. In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include `README.txt`, `Makefile`, and program source file(s). We should be able to compile your program(s) by just typing `make`. No binary files (executable files or .o files) will be included in your submission. Then **tar** and **gzip** the directory, and submit it to **Moodle**.

For example, a project group with student IDs `21404312` and `214052104` will create a directory named `21404312-214052104` and will put their files there. Then, they will tar the directory (package the directory) as follows:

`tar cvf 21404312-214052104.tar 21404312-214052104`

Then they will gzip (compress) the tar file as follows:

`gzip 21404312-214052104.tar`

In this way they will obtain a file called `21404312-214052104.tar.gz`. Then they will upload this file to **Moodle**. For a project done individually, just the ID of the student will be used as a file or directory name.

4. Tips and Clarifications

1. Start early, work incrementally.
2. You will not simulate TLB.
3. Maximum filename length can be 63 characters.
4. A referenced virtual address can be in range $[0, 2^{16} - 1]$.

5. In two-level paging, an inner table will be created when needed. At that time, you will allocate space for the inner table using `malloc()`. Then you will initialize it properly.
6. You can assume that all options will present when the program is invoked. You can also assume that options will be entered in the order shown.
7. We need to specify and use a tick value even though the page replacement algorithm is specified as FIFO or LRU.
8. For a page that came new to the memory, the R bit is set to 1.
9. If, for example, the tick value is 10, then you will clear the R bits after the 10th reference (after it is completely processed), i.e., you will clear just before the 11th reference starts.