Assignment #5
Simulation of a Virtual-Memory Manager

CS201 Spring 2023
Part I: 10 points, due Wednesday, Apr. 19th, 11:59 pm
Part II: 20 points, due Saturday, Apr. 29th, 11:59 pm

1 Overview

You'll create a program that will translate logical (virtual) addresses to physical addresses for a virtual address space of size $2^{16} = 65536$ bytes. Your program will read from a file containing a sequence of virtual addresses and will translate each virtual address to its corresponding physical address and will write out the value of the byte stored at the translated physical address. The key data structure will be a page table. The goal of the project is to simulate the steps involved in address translation and paging. Graduate students, and undergraduates for extra credit, will also model a translation look-aside buffer (TLB), which will assist in the address translation.

You may work individually or with a partner.

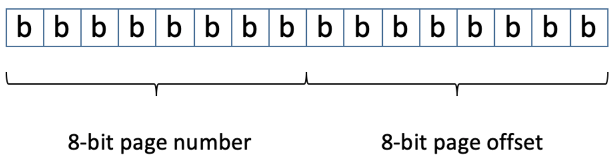
2 Details

Your program will read a text file containing a sequence of integers that represent virtual addresses. The numbers in the file are in ASCII format; they are integers in the range 0 to 65535, and so each equivalent numeric value can be represented using 16 bits.

These 16 bits are divided into:

- most-significant eight bits: the page number
- least-significant eight bits: the page offset

So here's what a 16-bit logical address looks like:



In addition:

- the page table has $2^8 = 256$ entries
- the page size is $2^8 = 256$ bytes
- the frame size is $2^8 = 256$ bytes

This means that the virtual memory is $256 \times 256 = 65536$ bytes in size. The physical memory will be smaller: `NUM_FRAMES` $\times$ 256 bytes in size. This means that not every virtual page can be mapped to a physical frame simultaneously, and so we will need a page-replacement policy.

The system will only read from logical addresses—it won't write to the logical address space.

Set `NUM_FRAMES` to 128.

2.1 Page replacement

Since there are more virtual pages than physical frames, it will be necessary to evict a page from the physical memory if all frames are occupied and a new page is requested.

Use a simple LRU scheme: keep track of when each page is accessed by using an array `int accessTime[NUM_FRAMES]`. The function that you call to access memory will also have an integer parameter `time`. So if frame 32 is accessed at `time=10`, then set `accessTime[32] = 10`. Initialize this array to all zeros.

When it is time to evict a page, evict the page from memory whose access time is the smallest—in other words, the page that was used the longest time ago.

3 Address Translation

The procedure for translating a logical address to a physical address is as follows:

- extract the page number from the logical address
- check to see whether the desired page is in the page table
- if it is, then read the data from the specified offset of the specified page in the page table
- otherwise, this is a page fault

There is no explicit check for whether the page is permitted to be read (we will ignore memory protection).

When a page fault occurs, you'll follow the steps associated with a page fault, as described in Lecture 9. For this program, $m = 16$ and $n = 8$. The backing store is represented by a binary data file `BACKING_STORE.dat`, of size 65536 bytes. When a page fault occurs, you will read in a 256-byte page from the file `BACKING_STORE.dat` and will store it in an available page frame in physical memory.

For example: if a logical address with page number 15 results in a page fault, then your program would read in page 15 from `BACKING_STORE.dat`, from the location `v` to `v+PAGE_SIZE-1`, where `v = PAGE_SIZE * 15`, and copy those `PAGE_SIZE` bytes to a free frame in physical memory. Once this page is stored (and the page table is updated), subsequent accesses to page 15 will be resolved by the page table, and data will be read from the correct frame in the physical memory. In other words, the reading of the page from `BACKING_STORE.dat` happens only during a page fault.

You will treat `BACKING_STORE.dat` as a random-access file, so that you can randomly seek to certain positions of the file and then read data. The first part of the assignment will consist of functions to read from the binary file.

The file `BACKING_STORE.dat` is in the class gitlab site under `CS201/Assignments/vmm`.

4 The Page Table

What needs to go into the page table? It is really just an integer array having `NUM_PAGES` entries. You should `#define NUM_PAGES 256` in your code. Also, `#define PAGE_SIZE 256`.

Suppose that the first logical address in the reference string (from the file of addresses) is 16916. This corresponds to page number 66 and page offset 20. Here is one way to calculate the values, using bitwise operations in C:

```
(16916 >> 8) & 255 = 66

16916 & 255 = 20
```

Another way to calculate the values uses integer division and mod:

```
16916 / 256 = 66

16916 % 256 = 20
```

Since this will be the first address you process, it will cause a page fault. Read page number 66 (bytes $66*256$ to $66*256+255$) from `BACKING_STORE.dat`, and write it into frame 0 of the physical memory. You then record $(66, 0)$ in the page table by setting `pageTable[66] = 0`. Also, set the access time for frame 0 to 0: `accessTime[0] = 0`.

The physical memory address that corresponds to this virtual address is $0 * 256 + 20 = 20$.

Suppose then that the next virtual address is 62493. This corresponds to page number 244 and page offset 29:

```
(62493 >> 8) & 255 = 244

62493 & 255 = 29
```

And using integer division and the mod operator:

```
62493 / 256 = 244

62493 % 256 = 29
```

Since this will be the first time that page number 244 is accessed, this is also a page fault. Read page 244 (bytes $244 * 256$ to $244 * 256 + 255$) from `BACKING_STORE.dat` and put it in the next free frame, which is frame 1. Put $(244, 1)$ in the page table: set `pageTable[244] = 1`. Also, set `accessTime[1] = 1`.

The physical memory address that corresponds to the virtual address is then $1 * 256 + 29 = 285$.

5 How to Begin

Write this function:

```
int decodeAddress(int address, int *pageNumber, int *pageOffset);
// the function should take a four-byte integer in the range 0 to 256*256-1
// and compute the page number and page offset as described above;
// it should return 0 if the address argument is valid (in the range 0 to 256*256-1)
// and 1 otherwise
```

Use the following values to test your function. Here, I show the page number and the page offset for various virtual address values:

logical address	page number	page offset
0	0	0
1	0	1
256	1	0
32768	128	0
32769	128	1
128	0	128
65534	255	254
33153	129	129
16916	66	20
62493	244	29

Here is the other function you should write:

```
int readFromBackingStore(FILE *fp, char *buffer, int pageNumber);
// read bytes n to n+PAGE_SIZE-1, where n = PAGE_SIZE * pageNumber;
// put the data into the location pointed to by buffer
// return 0 if there was no error during the read; otherwise return 1
```

So for example, reading page 0 means reading bytes 0 to 255. Reading page 4 means reading bytes 1024 to 1279. Use `fread()` to read binary data.

You can testing this out by looking at the file `BACKING_STORE.asc`. This shows the contents of `BACKING_STORE.dat`, in readable format:

```
...
page number 133 page offset 252 value is 0
page number 133 page offset 253 value is 0
page number 133 page offset 254 value is 33
page number 133 page offset 255 value is 127
-----------------------------------------------------
page number 134 page offset 0 value is 0
page number 134 page offset 1 value is 0
page number 134 page offset 2 value is 33
page number 134 page offset 3 value is 128
page number 134 page offset 4 value is 0
page number 134 page offset 5 value is 0
page number 134 page offset 6 value is 33
page number 134 page offset 7 value is 129
page number 134 page offset 8 value is 0 ...
```

5.1 Opening a file for reading

Here's how to open a file for reading (the actual reading happens in subsequent calls to `fread()`):

```
char *fname = "BACKING_STORE.dat";
FILE *fp = fopen(fname, "r");
if (fp == NULL) {
  printf("ERROR: cannot open file '\%s' for reading\n", fname);
  exit(8);
}
```

It's good practice for a program that calls `fopen()` to call `fclose()` before it exits.

6 What to Do

6.1 Part I

Create the two functions shown above. Test them.

Put your code in a file named `vmm1.netid.c`. This file should have only your two functions (no `main()` function).

6.2 Part II

Implement the actual page-table mechanism. Use the file `addresses.txt` as your reference string (the sequence of virtual addresses for the VMM).

See the file `read-from-ascii-file-example.c` in the class github site for an example program that reads numbers from an ASCII file.

Your program should read the addresses in `addresses.txt`, use the page table to load relevant pages from `BACKING_STORE.dat`, and then "access" the specified byte in the physical memory and print out for each address, a line like this:

```
* Virtual address: 16916 [66, 20] Physical address: 20 [0, 20] Value: 0
```

The line above is the first line you should see. The "*" means that there was a page fault. Here are the next four lines:

```
* Virtual address: 62493 [244, 29] Physical address: 285 [1, 29] Value: 0
* Virtual address: 30198 [117, 246] Physical address: 758 [2, 246] Value: 29
* Virtual address: 53683 [209, 179] Physical address: 947 [3, 179] Value: 108
* Virtual address: 40185 [156, 249] Physical address: 1273 [4, 249] Value: 0
```

Eventually (after 128 different pages have been accessed), you'll get a page fault. This will happen for reference #172 (virtual address 48855). In this case, you need to select a page for eviction. The oldest page is page zero. Here's the output you should create:

```
  Virtual address: 34621 [135, 61] Physical address: 17981 [70, 61] Value: 0
* Virtual address: 51365 [200, 165] Physical address: 32677 [127, 165] Value: 0
  Virtual address: 32820 [128, 52] Physical address: 4148 [16, 52] Value: 0
  => EVICT! oldest frame is 0 (age = 0)
  => the page mapped to frame 0 is 66: page 66 is now unmapped (not in memory)
* Virtual address: 48855 [190, 215] Physical address: 215 [0, 215] Value: -75
```

And then, two addresses later, there will be another page fault:

```
* Virtual address: 51365 [200, 165] Physical address: 32677 [127, 165] Value: 0
  Virtual address: 32820 [128, 52] Physical address: 4148 [16, 52] Value: 0
  => EVICT! oldest frame is 0 (age = 0)
  => the page mapped to frame 0 is 66: page 66 is now unmapped (not in memory)
* Virtual address: 48855 [190, 215] Physical address: 215 [0, 215] Value: -75
  Virtual address: 12224 [47, 192] Physical address: 2752 [10, 192] Value: 0
  => EVICT! oldest frame is 2 (age = 2)
  => the page mapped to frame 2 is 117: page 117 is now unmapped (not in memory)
* Virtual address: 2035 [7, 243] Physical address: 755 [2, 243] Value: -4
```

From this, we can see that page in frame #1 was accessed more recently than the page in frame #2 (otherwise, the page in frame #1 would have been evicted).

The first time an evicted page is read back is for the reference to address 64181. You should see that this causes a page fault:

```
  Virtual address: 585 [2, 73] Physical address: 17225 [67, 73] Value: 0
  Virtual address: 51229 [200, 29] Physical address: 32541 [127, 29] Value: 0
  => EVICT! oldest frame is 18 (age = 18)
  => the page mapped to frame 18 is 9: page 9 is now unmapped (not in memory)
* Virtual address: 64181 [250, 181] Physical address: 4789 [18, 181] Value: 0
```

I've put my output in the file `vmm.out`, in the gitlab repo.

Put all of your code in a file named `vmm2.netid.c`. Your Part II code should use your Part I functions and should have a `main()` function so that it can run all by itself.

6.3 Structures

Put all relevant page-table information in a structure:

```
typedef struct {
  int pageTable[NUM_PAGES]; // the actual page table
  int accessTime[NUM_FRAMES]; // the most recent time that a frame was accessed
  unsigned char freeFrame[NUM_FRAMES]; // 0 if a frame is occupied; otherwise 1
} PageTableInfo;
```

Initialize the `accessTime[]` array to all -1. Initialize the `freeFrame[]` array to all 1.

You can also create a global variable to represent the physical memory:

```
char physicalMemory[NUM_FRAMES * PAGE_SIZE];
```

Make a function of this form:

```
int getFrameNumber( PageTableStruct *pageTableInfo,
                    int logicalPageNumber,
                    int accessTime,
                    int *pageFault);
```

This takes a virtual page number as input, and an access time. If this page is in memory, then it sets `pageFault` to 0 and returns the frame number. It choses a frame, returns that frame number, and sets `pageFault` to 1. This is the function that will manage the page table.

If the page does cause a page fault, then you need to read data from the backing store into memory.

You'll use your function from before to do the actual reading from the binary file:

```
int readFromBackingStore(FILE *fp,
                         char *buffer,
                         int pageNumber);
```

When you evict a page (to reuse an in-use frame during a page fault), mark the page being evicted as "unmapped"; *i.e.*, no longer in memory. To do so, set the `pageTable` entry for that page to -1: if you are reusing frame `f`, then you'll have to look through the page table to find the page `p` that is mapped to frame `f`. Then set `pageTable[p]` to -1.

6.4 Read loop

Your `main()` should have a while loop:

```
accessTime = 0
read a virtual address from addresses.txt
while (read was successful) {
  convert the text address into an integer address
  parse the address, to get the page number and the page offset
  call getFrameNumber() to get the frame number
  if this is a page fault {
    read PAGE_SIZE bytes from the backing store,
        from bytes PAGE_NUMBER * PAGE_SIZE to PAGE_NMBER * PAGE_SIZE + 255
    save those bytes in memory, starting at location frameNumber * PAGE_SIZE
  }
  get the actual byte from memory[frameNumber * PAGE_SIZE + PAGE_OFFSET]
  print info in the format shown
  accessTime = accessTime + 1
  read the next virtual address from addresses.txt
}
```

Note that you are reading from two different files, and that these two different files have two different kinds of data:

- `addresses.txt`: this is ASCII (text) data; you will read numbers from this file as text, using `fgets()`
- `BACKING_STORE.dat`: this is binary data; you will read binary data (in 256-byte chunks) from this file, using your `readFromBackingStore()` function, which calls `fread()`

6.5 Part III

This is for graduate students and for undergraduates who would like a little extra credit.

Implement a TLB to assist with the page-table lookup. Use a FIFO replacement strategy for the TLB.

The TLB will have 16 entries.

Figure 9.12 in the book shows how the TLB assists the page-table lookup. This is also shown in Lecture 9 Part II.

Keep the TLB as a structure having two integer arrays:

```
int logicalPageNumber[TLB_SIZE];
int frameNumber[TLB_SIZE];
```

And you should `#define TLB_SIZE 16`

Implement the TLB as a circular buffer: the first time you put a `(logicalPageNumber, frameNumber)` pair in the TLB, put it at index = 0. The next time, put it at index = 1, etc. After you write a `(logicalPageNumber, frameNumber)` pair at index = TLB_SIZE-1, you'll write the next entry at index = 0 (because of the circular nature of the buffer). It makes sense to keep a "current index" variable with the TLB to indicate the position where you'll write the next entry to the TLB.

Put all of your code in a file `vmm3.netid.c`.