Assignment #6: Simulation of a Virtual-Memory Manager

CS201 Spring 2020
Part I: 10 points, due Tuesday, 4/14, 11:59 pm
Part II: 20 points, due Sunday, 4/26, 11:59 pm

1 Overview

You'll create a program that will translate logical addresses to physical addresses for a virtual address space of $2^{16} = 65536$ bytes. Your program will read from a file containing a sequence of logical addresses and will translate each logical 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. For extra credit, you can also model a translation look-aside buffer (TLB), which will assist in the address translation.

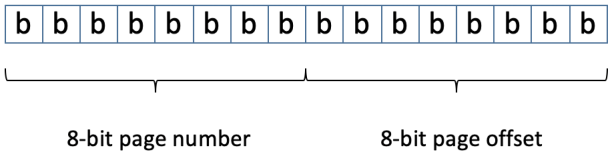
2 Details

Your program will read a text file containing a sequence of 16-bit integers that represent logical addresses. (The numbers in the file are in text format; they are integers in the range 0 to 65535, and so each numeric value can be represented by 16 bits.)

These 16 bits are divided into:

- top eight bits: the page number
- bottom 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 $256 * 256 = 65536$ bytes in size. Assume also that the physical memory is $256 * 256 = 65536$ bytes in size. The system will only read from logical addresses–it won't write to the logical address space.

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 protection).

When a page fault occurs, you'll follow the steps associated with demand paging, in Section 10.2 of the textbook (and slides 17-29 of Lecture #10). 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` and store it in 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 (by accessing the data in that frame).

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 size of the physical memory is the same as the size of the virtual memory: 65536 bytes. This means that you don't have to worry about page replacement during a page fault.

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.

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 & (255 << 8)) >> 8 = 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 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`. (Note: the original version of this doc had a typo here – it said "record $(20, 0)$ in the page table by setting `pageTable[20] = 0`.")

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

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

```
(62493 & (255 << 8)) >> 8 = 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 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`.

The physical memory address that corresponds to the logical 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 logical 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, unsigned char *buffer, int pageNumber);
// read bytes n to n+255, where n = 256*pageNumber;
// put the date 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.

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 logical 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. 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
```

Your program should produce be 1000 lines of output like this in total.

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.

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()`

6.3 Part III

This is for graduate students and for undergraduates who want 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 on my slides 20-21 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`.