CS 370
Programming Assignment 4
Memory Management

Due: December 12, 2021
By 11:55 PM

Part 1: Designing a Virtual Memory Manager using Single-level Paging (30 Points)

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size 2^{16} = 65,536 bytes. Your program will read from a file containing logical addresses and, using a page table, will translate each logical address to its corresponding physical address and will output the value of the byte stored at the translated physical address. The goal of this project is to simulate the steps involved in translating logical to physical addresses, using single level paging.

Specifics

Your program will read a file (**addresses.txt**) containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the rightmost 16 bits of each logical address. These 16 bits are divided into (1) an 8-bit page number and (2) 8-bit page offset. Hence, the addresses are structured as shown in Figure 1. Other specifics include the following:

- 2^8 entries in the page table
- Page size of 2^8 bytes
- Frame size of 2^8 bytes
- Physical memory of 16,384 bytes (i.e., 16 KB).

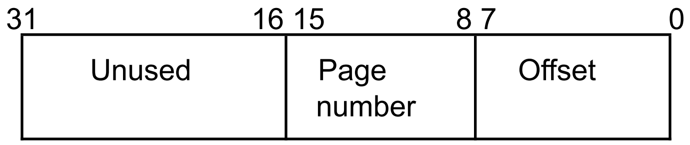

Figure 1. Address structure

Additionally, your program need only be concerned with reading logical addresses and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space.

Address Translation

Your program will translate logical to physical addresses using a page table as outlined in Section 8.5. First, the page number is extracted from the logical address and the page table is consulted to retrieve the frame number for logical to physical address translation. Either the frame number is obtained from the page table, or a page fault occurs if the referenced page is not present in the main memory.

Structure of Page Table Entry

A page table entry is of two bytes with the following structure:

- Bit_0 – Bit_7 – store the frame number
- Bit_8 – indicates indicate whether the page is in memory or not.
- Bit_9 – indicates indicate whether the page is in memory is dirty or not.
- Bit_{10}-Bit_{11} – are used as 2-bit counter to implement the enhanced second chance algorithm for page replacement as discussed below.

Figure 2 shows the structure of page table entry for Part 1.

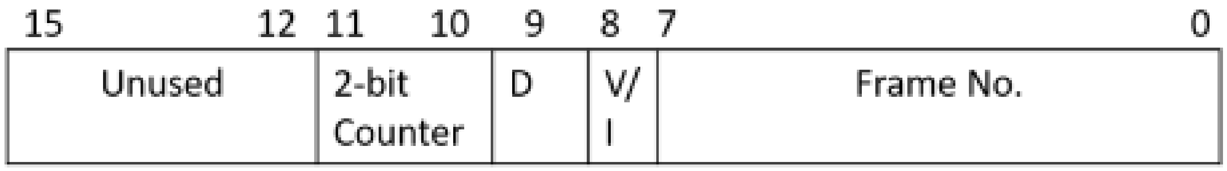

Figure 2. Page table entry structure for Part 1

Page Replacement Strategy

Since the logical memory is bigger than the physical memory, you will need to use a page replacement strategy to replace the frames as needed. You have to write the swapped out frame to a backing store file (BACKING_STORE_1.bin), and reload it as needed later on. The strategy that you will use is the enhanced page replacement strategy as described in **Section 10.4.5.3** (**Enhanced Second Chance Algorithm**) in the 10th edition of the Silberschatz's Textbook.

Handling Page Faults

Your program will implement demand paging as described in Section 10.2 of the textbook. The backing store is represented by the file BACKING_STORE_1.bin, a binary file of size 65,536 bytes. When a page fault occurs, you will read in a 256-byte page from the file BACKING STORE and store it in an available page frame in physical memory. For example, if a logical address with page number 15 resulted in a page fault, your program would read in page 15 from BACKING STORE (remember that pages begin at 0 and are 256 bytes in size) and store it in a page frame in physical memory. Once this frame is stored (and the page table is updated), subsequent accesses to page 15 will be resolved by the page table. You will need to treat BACKING_STORE_1.bin as a random-access file so that you can randomly seek to certain positions of the file for reading. We suggest using the standard C library functions for performing I/O, including fopen(), fread(), fseek(), and fclose().

Test File
We provide the file **addresses.txt**, which contains integer values representing logical addresses ranging from 0 – 65,535 (the size of the virtual address space). Additionally, each address will have a corresponding 0 or 1 written in the same line, representing whether the address is a read or a write. Your program will open this file, read each logical address and translate it to its corresponding physical address.

If it is a read, print the signed byte along with the corresponding logical and physical address **(see output format below).** If the corresponding page is not present in the memory, you need to retrieve it from the BACKING_STORE_1.bin.

If it's a write, you need to first read the byte value at the given address, divide by 2, and write the resulting byte value back to the given memory address. Moreover, you need to print the corresponding logical and physical addresses as well as the resulting byte value after dividing by 2 **(see output format below).** In addition, the dirty bit in the corresponding page table entry will be set to '1'. If the corresponding page is not present in the memory, you need to retrieve it from the BACKING_STORE_1.bin. You also need to update the reference bit in the page replacement strategy as needed.

How To Begin
First, write a simple program that extracts the page number and offset from the following integer numbers:

1, 256, 32768, 32769, 128, 65534, 33153

Perhaps the easiest way to do this is by using the operators for bit-masking and bit-shifting.

Once you can correctly establish the page number and offset from an integer number, you are ready to begin.

How To Run Your Program

Compile your program by running Makefile: **make**
Your program should run as follows: **./run addresses.txt.**

Your program will read in the file addresses.txt, which contains 1000 logical addresses ranging from 0 to 65,535. Your program is to translate each logical address to a physical address and determine the contents of the signed byte stored at the correct physical address. (Recall that in the C language, the char data type occupies a byte of storage, so we suggest using char values)

Your program is to output the following values:

1. The logical address being translated (the integer value being read from addresses.txt).
2. The corresponding physical address (what your program translates the logical address to).
3. Whether the access was read or write

4. The signed byte value stored at the translated physical address.
5. Whether the access resulted in page fault or not.

Output format for both read and write:

If upon a read for address 0xFA1C, there's a page fault and Page number 0xFA is allocated to Frame 0x84, and the data at the logical address 0xFA1C in the backing store is 0x45. print:

Logical Address	Physical Address	Read/Write	Value	Page Fault
0xFA1C	0x841C	Read	0x45	Yes

We also provide the file sample.txt, which contains the first 15 sample output values for the file addresses.txt.

Statistics

After completion, your program is to report the page-fault rate — the percentage of address references that resulted in page faults (i.e. number of page faults divided by the number of memory addresses read).

Part 2: Two Level Paging
(70 Points)

In part 2, you have logical addresses of size 24 bits, and thus the address space is 2^{24} bytes (i.e., 16,777,216 bytes) . For any new details not specified in this part, you can assume that they match the requirements defined in Part 1.

Address Translation

You will use two levels of paging. The first 6 bits will define the offset in the Level 1 page table, the second 8 bits will define the offset in the Level 2 page table, and the final 10 bits will define the offset in the actual frame as shown in Figure 3.

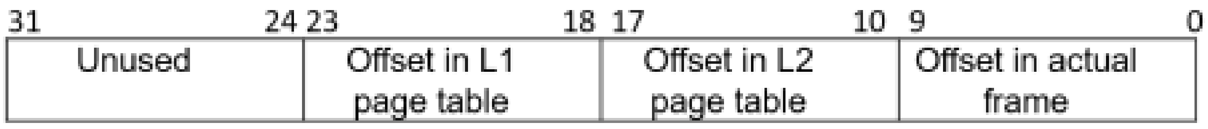

Figure 3. Address structure for Part 2

Page Replacement and Handling Page faults

On top of implementing a double level paging scheme, your physical memory is limited to 128 kilobytes. This means that you can neither keep all Level 2 page table, nor keep all the frames in the memory. Out of 128 kilobytes (128 frames), you can use **only 33 frames to store the page tables – 1 frame for storing the ENTIRE Level 1 page table and 32 frames for storing level 2 page table.**

NOTE: LEVEL 1 Page Table will always be present in memory.

Thus, you will need to use a page replacement strategy to replace the frames as needed. Depending on your implementation, you will have page faults in both L1 page table and L2 page table. You may store the remainder of the L2 page table on the backing store below the last instruction you will be processing (next page). The strategy that you will use is the enhanced page replacement strategy as described in **Section 10.4.5.3** (**Enhanced Second Chance Algorithm**) in the 10th edition of the Silberschatz's Textbook.

Structure of Page Table Entry for Part 2

A page table entry is of 4 bytes with the following structure:

- Bit_0 – Bit_{15} – store the frame number
- Bit_{16} – indicates indicate whether the page is in memory or not.
- Bit_{17} – indicates indicate whether the page is in memory is dirty or not. • Bit_{18}-Bit_{19} – are used as 2-bit counter to implement the enhanced second chance algorithm for page replacement as discussed below.
- Bit_{20}-Bit_{31} -- are unused.

Figure 4 shows the structure of page table entry for Part 2.

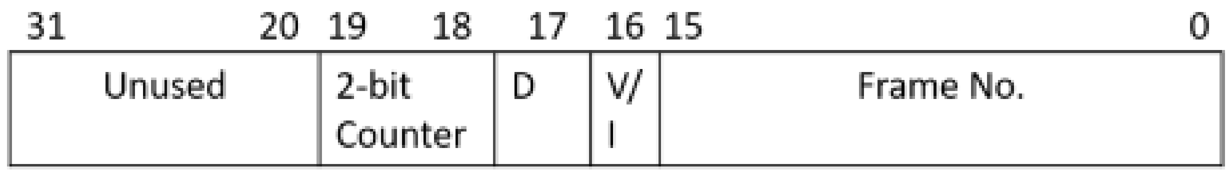

Figure 4. Page table entry structure for Part 2

Program Execution:
You are given a file BACKING_STORE_2.bin (size 16,777,216 bytes). In this file we have placed a binary form of a code (Recall from your CS-225 course that every code you write is first converted to its binary form before execution as computer only understands binary language). This binary code starts at (the first instruction is at) address 0x00C17C00 in BACKING_STORE_2.bin and ends at (the last instruction is at) 0x00C193E8. Your task is to extract this binary from BACKING_STORE_2.bin and execute this binary code using your 2-level paging system.

Note: After completion of the program, make sure that you write all the dirty pages back to the backing store.

Decoding binary code:
In this binary code each instruction is of 8 bytes and each memory address is of 3 bytes. There are only two types of instructions in the binary (**memory-memory instructions and memory value instructions**). In **memory-memory** instruction both arguments of the instruction are memory addresses so you need to read the value from both addresses and store the result of the operation in the first memory address. In **memory-value** instruction, the first argument of the instruction is a memory address and second argument is an immediate value. The result of the operation is to be stored at memory address.

As mentioned earlier each instruction is of 8 bytes starting from address 0x00C17C00. The first byte will tell you the op code of the instruction as well as the type of instruction (mem-mem or mem-value). The mapping is given in Table 1 below.

Table 1. Instruction mapping table

First byte of instruction (Hex)	Opcode	Instruction type
10	Add	memory-value
11	Add	memory-memory
20	Mov	memory-value
21	Mov	memory-memory
30	Sub	memory-value
31	Sub	memory-memory
40	Multi	memory-value
41	Multi	memory-memory
50	AND	memory-value
51	AND	memory-memory
60	OR	memory-value
61	OR	memory-memory
70	XOR	memory-value
71	XOR	memory-memory

The structure of the instruction is given below.

Memory-Memory Instruction structure:

Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0
Opcode	**First Address**			**Second Address**			**Unused**

Memory-Value Instruction structure:

Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0
Opcode	**First Address**			**Value**			

Note:
You have to use memory management system with 2 level page tables to execute this code. The addresses in the instructions are also from BACKING_STORE_2.bin.

Example Instruction:
For example, if the instruction is 0x601011e211111110, the opcode 60 indicates that the instruction is **OR** and it is a memory-value instruction. To execute this instruction, you need to read the value at address **0x1011e2** from BACKING_STORE_2.bin let's call the value **A**. Take bitwise OR of **A with 0x11111110** and store the result of the operation at address **0x1011e2.**

Expected output:
You are required to execute all the binary code instructions from **Address 0x00C17C00 to 0x00C193E8.**
On each instruction you need to print the following:

1. Type of instruction.
2. Whether the memory address access was a page hit or miss for both inner and outer page table.
3. The value at that address.

Output format

Instruction Type (memory memory or memory value)	Address 1 – Outertable Page table hit/miss	Address 1 – Innertable Page table hit/miss	Address 2 – Outertable Page table hit/miss	Address 2 – Innertable Page table hit/miss	Value at Address 1 before execution	Value at Address 1 after execution	Instruction read – opcode, first address, second address/value

We also provide the file sample.txt, which contains the first 15 sample output values for the file addresses.txt.

Submission Guidelines:
You have been provided with a zipped file. The zipped file contains two folders, for part 1 and part 2. DO NOT rename any of the existing files or folders. You can however change the ● Makefiles to add any new code. However, make sure the -o flag of gcc is not changed, and the name of the final executable is run.

- Zip both the folders, without adding them to a new folder, and upload the zipped file to LMS.

- The final zip file must be renamed as <R1>_<R2>.zip, where R1 and R2 are the roll numbers of the group members.
- Only one member in the group needs to submit the assignment.

General Tips (for both parts):

This assignment might seem daunting at first but the underlying concepts are not very hard. If you understand the concepts correctly, writing pseudo-code for the whole assignment would not take more than a couple of hours. For implementation, I recommend using a top-down programming approach. Try to use:

1. Global constants (e.g. PAGE_ENTRIES, PAGE_SIZE, MEM_SIZE, DIRTYBIT, COUNTBITS, VALIDBIT etc.). This would make your code cleaner, less error prone, and easier to understand for yourself
2. Modular design. Divide each task into separate procedures. Some obvious functions include functions for getting a free memory location, setting certain bits (dirty, invalid etc), getting a byte from a given address, exporting page, parsing instruction (for part 2), hexadecimal to decimal converter etc.

Once you have the pseudo-code written, you'll then just need to implement each function one by one. Try to test each function as you write it. If you test all of your functions at the end, it would be very hard for you to catch errors.