Programming Assignment 4
Memory Management

Part 1: Designing a Virtual Memory Manager

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 output the value of the byte stored at the translated physical address. The goal behind this project is to simulate the steps involved in translating logical to physical addresses, using single level paging.

Page Replacement and Handling Page faults

You will be given 32 bit Integer numbers, and use demand paging. On top of implementing a single level paging scheme, your physical memory is limited to 16 kilobytes.

This means that although you can keep all the page tables in 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 file, and reload it as needed later on. The strategy that you will use is the enhanced page replacement strategy described in section 9.4.5.3 on page 419 of the book. Since the last 8 bits represent the logical address, you will use the first 8 bits for the following purpose.

1) Keep a single bit to indicate whether the page is in memory or not.
2) Keep a single bit to indicate whether the page in memory is dirty or not.
3) Use two bits for the page replacement strategy described in the book.

Thus, each entry in the page table will be of 2 bytes only.

Specifics

Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must consider 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 9.33. Other specifics include the following:

- 2^8 entries in the page table
- Page size of 2^8 bytes
- Frame size of 2^8 bytes

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. Either the frame number is obtained from the page table or a page fault occurs. A visual representation of the address-translation process appears in Figure 9.34 (OS concepts Page 459)

Handling Page Faults

Your program will implement demand paging as described in Section 9.2. The backing store is represented by the file BACKING_STORE.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.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’s a read, print the signed byte along with the corresponding logical and physical address. If the corresponding page is not present in the memory, you need to retrieve it from the BACKING_STORE.bin.

If it’s a write, you need to print the corresponding logical and physical addresses. If the corresponding page is not present in the memory, you need to retrieve it from the BACKING_STORE.bin. You also need to update your page replacement strategy as needed.

How To Begin

First, write a simple program that extracts the page number and offset (based on Figure 9.33) 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

Your program should run as follows: ./run.out addresses.txt Your program will read in the file addresses.txt, which contains 1,000 logical addresses ranging from 0 to 65535. 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. The signed byte value stored at the translated physical address.

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

Statistics

After completion, your program is to report the following statistics:

1. Page-fault rate—The percentage of address references that resulted in page faults.
2. On each page fault, also print out “frame” or “page” corresponding to whether the fault was for a page, or for a frame, on a newline.

Output format:

Logical Address   Physical Address   Value   Page Fault

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:

```
0xFA1C                 0x841C                 0x45          Yes
```

Part 2: Two Level Paging

In part 2, you have logical addresses of size 24 bits, and thus the address space is 2^24. For any new details not specified in this part, you can assume that they match the requirements defined in the first part.

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.

Page Replacement and Handling Page faults

You will be given 32 bit Integer numbers, and use demand paging. 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 table – 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. You have to write the swapped out frame to a file, and reload it as needed later on. The strategy that you will use is the enhanced page replacement strategy described in section 9.4.5.3 on page 419 of the book.

Each entry in the page table is of 4 bytes. Byte 2 and Byte 1 store the 16-bit Frame number. You can use Byte 0 for the following purpose.

1) Keep a single bit to indicate whether the page is in memory or not.
2) Keep a single bit to indicate whether the page in memory is dirty or not.
3) Use two bits for the page replacement strategy described in the book.

Byte 3	Byte 2	Byte 1	Byte 0
unused	16-bit Frame Number		

Test File

We provide the file addresses.txt, which contains integer values representing logical addresses ranging from 0 – 16,777,216 (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's a read, print the signed byte at the corresponding address from BACKING_STORE.bin (size 16,777,216 bytes) on a newline. For writes, update your page replacement strategy as needed, but don't output anything or write anything on the address.

<u>Statistics</u>

After completion, your program is to report the following statistics:

1) Page-fault rate—The percentage of address references that resulted in page faults.
2) On each page fault, also print out "frame" or "page" corresponding to whether the fault was for a page, or for a frame, on a newline.

<u>Output format:</u>

Logical Address	**Frame Number of the Inner Page Table**	**Physical Address**	**Value**	**Page Fault because the desired page of the inner page table was not in physical memory**	**Page fault because the reference memory location was not in physical memory**

Suppose that upon a read for address 0x32FA1C, the desired inner page table was not in physical memory and resulted in page fault. After serving the page fault the corresponding page of the inner page table was copied in Frame 0x0024. Also, the inner page table indicates that the reference memory location is not in the physical memory, causing another page fault. After serving this page fault, Frame number 0x0200 is allocated to the page causing the page fault. The data at the logical address 0x32FA1C in the backing store is 0x45. print:

0x32FA1C	**0x0024**	**0x02001C**	**0x45**	**Yes**	**Yes**

<u>Submission Guidelines:</u>

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. No other formats except zip are acceptable.

Any deviation from this format will result in your assignment not being marked.
.