CS352: Operating Systems, Spring 2018

Term Project

Due: May 30th, 2018 (23:55) on LMS

Note : You can do the project in groups of at most three students.

1. **(50 points) Virtual Memory Management Simulator.** The goal of the project is to simulate the steps involved address translation done by operating systems, and enhance of your understanding of virtual memory management concepts. You are going to write a program that translates logical addresses given to physical addresses for a virtual address space of 2^{16} = 65536 bytes. Your program will read from a text file containing logical addresses and, using a TLB as well as 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. You can write the program in either Java or C programming language.

   The program will read a file converting several 16-bit integer numbers that represent logical addresses. These 16 bits are divided into (1) an 8-bit page number, and (2) 8-bit page offset. Hence, there are 2^8 = 256 entries in the page table, and the page/frame size of the system is also 2^8 = 256. The size of the physical address space is equal to the size of the logical address space, i.e., 2^{16} = 65536 bytes. Consequently, the number of frames is also 2^8 = 256. Your program need only be concerned with reading logical addresses, translating them to their corresponding physical addresses, and outputting the byte stored at the physical address obtained. You do not need to support writing to the logical address space.

   Your program will translate logical to physical addresses using a TLB and page table as we covered curing lectures. First, the page number is extracted from the logical address, and the TLB is consulted. In the case of a TLB-hit, the frame number is obtained from TLB. In the case of a TLB-miss, the page table must be consulted. In the latter case, either the frame number is obtained from the page table or a page fault occurs. A visual representation of the address translation process is shown in Figure 1. You need to design and code appropriate data structures for TLB, page table and main memory of the system.

   Your program should implement demand-paging as described in the class. The secondary backing storage is represented by a file named BACKING_STORE.bin, a binary file of size 65536 bytes, and it will be provided to you. When a page fault occurs, you will read in a 256-byte page from the file BACKING_STORE.bin and store in an available page frame in physical memory. Once the frame is stored, first the page table and then the TLB should be updated, and any subsequent accesses to that page will be resolved by either the TLB or the page table. You should treat BACKING_STORE.bin as a random access file so that you can randomly seek to certain positions of the file for reading. It is suggested for you to use standard C library functions for performing I/O, including fopen(), fread(), fseek(), and fclose().

   Since the size of the physical memory is the same as the size of the virtual address space, i.e., 65536 bytes, you do not need to be concerned about page replacements during a page fault, because each physical frame stored on BACKING_STORE has a corresponding place in the physical memory.

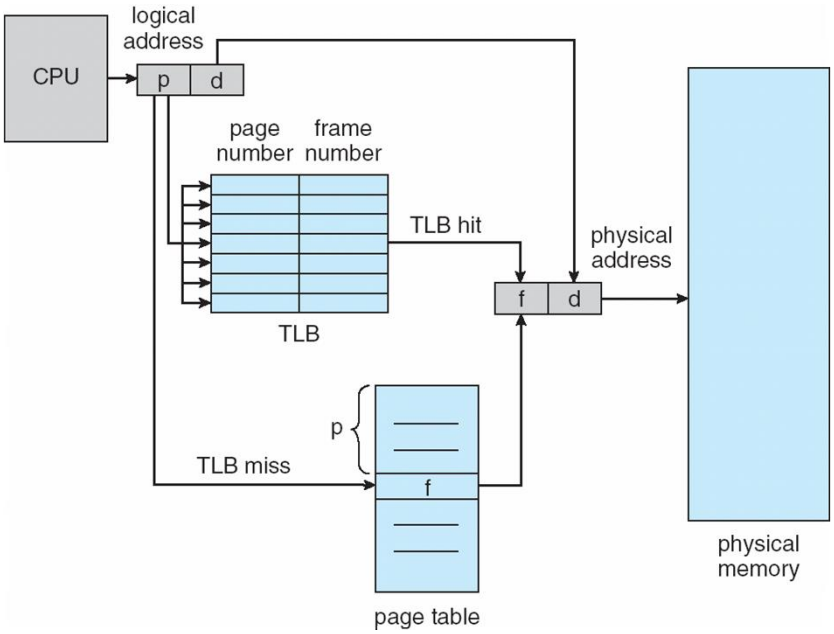

Figure 1 : A representation of address-translation process

On the other hand, since TLB has only 16 entries, you will need to use a replacement strategy when you update a full TLB. You should select the entry to be replaced via FIFO replacement strategy.

A test file called `addresses.txt` is provided which contains integer values representing logical addresses ranging from 0 to 65535. You program will open this file, read each logical address and translate it to its corresponding physical address, and then output the value of the signed byte stored at that physical address. In C, `char` data type occupies one byte of storage, so you can use `char` values to represent physical memory contents.

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. TLB-hit rate: The percentage of address references that were resolved in the TLB.

2. (50 points) Sudoku Solution Validator. A *sudoku* puzzle uses a 9x9 grid in which each columns and row, as well as each of the nine 3x3 sub-grids must contain all of the digits 1…9. Figure 2 below presents an example of a valid Sudoku puzzle. You are required to write the program in C.

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Figure 2

This project consists of designing a multithreaded application that determines whether the solution to a Sudoku puzzle is valid or not. There are several different ways of multithreading this application. One suggested strategy is to create threads that check the following criteria:

1. A thread to check one column of the grid contains the digits 1 through 9, i.e., nine separate threads to check all columns
2. A thread to check one row of the gird contains the digits 1 through 9, i.e., nine separate threads to check all rows.
3. A thread to check that one of the 3x3 sub-grids contains the digits 1 through 9, i.e., nine threads to check all sub-grids.

With the above strategy there are 27 worker threads (plus a main thread) to check whether a given Sudoku grid is valid or not.

Your program should ask the user to enter 81 digits from the console (in row-major order, i.e., first the numbers in the first row are entered, then the numbers in the second row, etc.), and the main thread of your program should create the worker threads, passing each worker thread the locations within the grid that it must check. Note that the specific cells to be checked by column validators, row validators, and sub-grid validators are different from each other. You should design your worker threads according to their duties, ensure correct measures on how to obtain the location information from the main thread. In any case this will require passing several parameters to each thread. The simplest approach is to create a data structure using a struct `construct` of C. For example, a structure to pass the row and column where a thread must begin its validation would appear as follows:

```
/* structure for passing data to threads */
typedef struct
{
     int row;
     int column;
} parameters;
```

Both Pthreads and Windows programs can create worker threads using a strategy similar to that shown below:

```
parameters *data = (parameters *) malloc(sizeof(parameters));
data->row = 1;
data->column = 1;
/* Now create the thread passing it data as parameter */
…….
```

The data pointer will be passed to either the pthread_create() (Pthreads) function or the CreateThread() (Windows) function, which in turn will pass it as a parameter to the function to run as a separate thread.

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker thread has performed this check, it must pass its results back to the parent. One way to handle this is to create a global array of integer values that is visible to each thread. The i^{th} index in this array corresponds to the i^{th} worker thread. If a worker thread sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 would indicate otherwise. When all worker threads have completed, the main thread can check each entry in the result array to determine if the Sudoku puzzle is valid or not. After printing the

result to the console, the program should exit.
It is apparent from the above discussion that the main thread should wait for all threads to finish their duties before reaching a final decision. This means that the main thread should wait for all the worker threads to exit, and then makes its final decision. You can implement this with the `pthread_join` function of the Pthreads library (Linux, MAC OS) or via the `WaitForMultipleObjects (Windows)` function from the Win32 API, as we have seen during our lectures. Alternatively, you can achieve this step via another global integer variable, say `threads_completed`, initially set to zero, and incremented by every thread when that thread finishes its job, and before it exits. When the value of this variable reaches 27, it means that every thread finished its job, and the main thread can finalize its decision, output the result, and exit.

Please implement your program by using the second approach explained above (i.e., by using a global variable to indicate that all threads finished their job). Note that you will need to ensure proper synchronization mechanisms between the threads so that this approach works correctly. You will be using either Pthreads mutex or Windows Mutex objects depending on your development environment.

4. Submissions and Grading

a) Submissions on LMS of the fully-developed codes with all required features, before the deadline (%50)
b) Demonstrations and interview (%40) : Demonstrating your programs as a group, and answering the questions about your implementations. Your TA will arrange the demonstration and interview schedules.
c) In-line comments within your codes, giving enough information on your implementation details (%10).