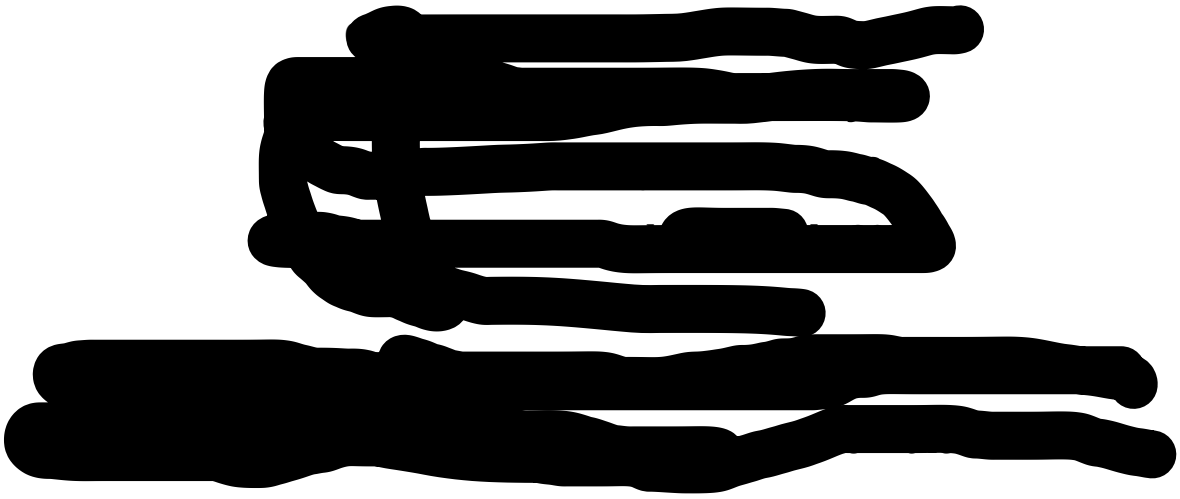
King Abdul Aziz University

Faculty of Computing & Information Technology



1. This project worth 10% of the overall module marks (100%).
2. This is a GROUP project.
3. Team leader must submit their project before 11:59 PM. Late submission will be penalized by deducting 20% of the score per one day.
4. Even though the Java language is not adequate for OS developments, however, in this project the focus is on illustrating the implementations aspects of memory management. **Hence, the language to be used for this project is Java.**

****

Part I

Memory management: Main memory

1. Designing and simulating a contiguous region of memory

Design and implement a program that simulates some of managing a contiguous region of memory of size MAX where addresses may range from 0 ... MAX − 1.

Your program must respond to four different requests:

- **a.** Request for a contiguous block of memory.
- **b.** Release of a contiguous block of memory.
- **c.** Compact unused holes of memory into one single block.
- **d.** Report the regions of free and allocated memory.

2. Allocating Memory

Your program will allocate memory using one of the three approaches: first fit, best fit, and worst fit depending on the flag that is passed to the RQ command. The flags are:

- **F—first fit**
- **B—best fit**
- **W—worst fit**

This will require that your program keep track of the different holes representing available memory. When a request for memory arrives, it will allocate the memory from one of the available holes based on the allocation strategy. If there is insufficient memory to allocate to a request, it will output an error message and reject the request.

Your program will also need to keep track of which region of memory has been allocated to which process. This is necessary to support the STAT command and is also needed when memory is released via the RL command, as the process releasing memory is passed to this command. If a partition being released is adjacent to an existing hole, be sure to combine the two holes into a single hole.

3. Compaction

If the user enters the C command, your program will compact the set of holes into one larger hole. For example, if you have four separate holes of size 550 KB, 375 KB, 1,900 KB, and 4,500 KB, your program will combine these four holes into one large hole of size 7,325 KB.

4. Implementation hints (Pat I)

Your program will be passed the initial amount of memory at startup. For example, the following initializes the program with 1 MB (1,048,576 bytes) of memory:

./allocator 1048576

Once your program has started, it will present the user with the following prompt:

allocator>

It will then respond to the following commands: RQ (request), RL (release), C (compact), STAT (status report), and X (exit).

A request for 40,000 bytes will appear as follows:

allocator>RQ P0 40000 W

The first parameter to the RQ command is the new process that requires the memory, followed by the amount of memory being requested, and finally the strategy. (In this situation, "W" refers to worst fit.)

Similarly, a release will appear as:

allocator>RL P0

This command will release the memory that has been allocated to process P0.

The command for compaction is entered as:

allocator>C

This command will compact unused holes of memory into one region.

Finally, the STAT command for reporting the status of memory is entered as:

allocator>STAT

Given this command, your program will report the regions of memory that are allocated and the regions that are unused. For example, one possible arrangement of memory allocation would be as follows:

Addresses [0:315000] Process P1

Addresses [315001: 512500] Process P3

Addresses [512501:625575] Unused

Addresses [625575:725100] Process P6

Addresses [725001] . . .

Part II

Memory management: Virtual Memory

5. 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 TLB and 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. Your learning goal is to use simulation to understand the steps involved in translating logical to physical addresses. This will include resolving page faults using demand paging, managing a TLB, and implementing a page-replacement algorithm.

6. Specific

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 mask the rightmost 16 bits of each logical address. These 16 bits are divided into (1) an 8-bit page number and (2) an 8-bit page offset. Hence, the addresses are structured as shown as:

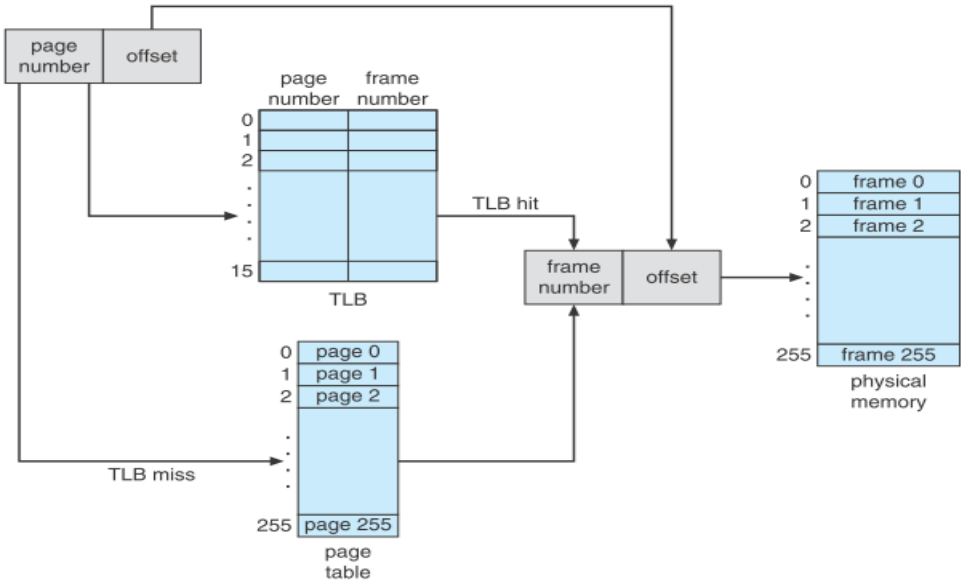
Other specifics include the following:

- 2^8 entries in the page table
- Page size of 2^8 bytes
- 16 entries in the TLB
- Frame size of 2^8 bytes
- 256 frames
- Physical memory of 65,536 bytes (256 frames × 256-byte frame size)

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.

7. Address Translation

Your program will translate logical to physical addresses using a TLB and page table as outlined in Section 9.3. 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 the 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:



8. Handling Page Faults

Your program will implement demand paging. 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 and TLB are updated), subsequent accesses to page 15 will be resolved by either the TLB or 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().

The size of physical memory is the same as the size of the virtual address space—65,536 bytes—so you do not need to be concerned about page replacements during a page fault. Later, we describe a modification to this project using a smaller amount of physical memory; at that point, a page-replacement strategy will be required.

9. Test File

You create the file addresses.txt, which contains integer values representing logical addresses ranging from 0 to 65535 (the size of the virtual address space). Your program will open this file, read each logical address and translate it to its corresponding physical address, and output the value of the signed byte at the physical address.

10. Implementation Hints (Part II)

a. How to Begin

First, write a simple program that extracts the page number and offset based on:

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. Initially, we suggest that you bypass the TLB and use only a page table. You can integrate the TLB once your page table is working properly. Remember, address translation can work without a TLB; the TLB just makes it faster. When you are ready to implement the TLB, recall that it has only sixteen entries, so you will need to use a replacement strategy when you update a full TLB. You may use either a FIFO or an LRU policy for updating your TLB.

b. How to Run Your Program

Your program should run as follows:

./a.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:

- **a.** The logical address being translated (the integer value being read from addresses.txt).
- **b.** The corresponding physical address (what your program translates the logical address to).
- **c.** The signed byte value stored in physical memory at the translated physical address.

We also provide the file correct.txt, which contains the correct output values for the file addresses.txt. You should use this file to determine if your program is correctly translating logical to physical addresses.

11.Statistics

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

a. **Page-fault rate**—The percentage of address references that resulted in page faults.
b. **TLB hit rate**—The percentage of address references that were resolved in the TLB.

Since the logical addresses in addresses.txt were generated randomly and do not reflect any memory access locality, do not expect to have a high TLB hit rate.

12.Page Replacement

Thus far, this project has assumed that physical memory is the same size as the virtual address space. In practice, physical memory is typically much smaller than a virtual address space. This phase of the project now assumes using a smaller physical address space with 128 page frames rather than 256. This change will require modifying your program so that it keeps track of free page frames as well as implementing a page-replacement policy using either **FIFO** or **LRU** to resolve page faults when there is no free memory.

****

Deliverables

- You should submit **one zip file (CPCS361ProjectYourTeamID.zip)** containing
  a. **Your report CPCS361GroupXPOurReport.docx** (Only word File is accepted) where **CPCS361** is your course name, X your group number, and P for project.

  b. **You report includes:**
     - A copy of a printout of your source code, containing a comment with your name, compiler name and version, hardware configuration, operating system, and version.
     - A copy of the program output on each test data file.
  c. **Your code CPCS361GroupXPOurCode.zip** where **CPCS361** is your course name, **X** your group number, and P for project.
     - source code [Par I and II]
     - executable file [Par I and II]
     - execution output for all test cases [Par I and II]
     - instructions for running your programs

- After Final presentation, you should submit **Your Presentation CPCS361GroupXP_OurPr.zip** (Only PowerPoint File is accepted) where **CPCS361** is your course name, **X** your group number, and **P** for project.

- Attend class for further details, answers to questions, and hints on this project. **You will be responsible for all that your Lab instructor says about this assignment in class.**

13.Scoring

Your project is will be scored based on the following:

Task	**Mark**
Input specification	5
Required Data	10
System requirements	15
Implementation of allocating memory and compaction strategies	20
Implementation of address translation, handling faults, page replacements strategies	20
Output Specification	5
Results	10
Presentation	15

Late submissions will be penalized by deducting 20% of the score per one day.