

Operating System Fundamentals - EECS3221

Project 2

Design a Memory Management Unit (MMU)

Prepared by: Dr. Ruba Al Omari<sup>1</sup>

**Contents**

Instructions .....	1
Description .....	2
Specifications .....	2
Phase 1 – No Page Replacement .....	2
Address Translation .....	2
Handling Page Faults .....	3
Test File .....	4
How to Begin .....	4
Phase 2 – With Page Replacement .....	4
Page Replacement .....	4
How to Run Your Program .....	5
Statistics .....	5
How to Test Your Program .....	5
Deliverables .....	6
Grading .....	6
Grading Phase1 .....	7
Grading Phase2 .....	7
Rubric .....	7
Collaboration .....	8
Environment Setup .....	8

**Instructions**

- This is an individual project.
- Check eClass for the due date and time.

---

<sup>1</sup> This project is provided as a programming project in Operating System Concepts, 10th Edition, by Silberschatz et al. textbook.

## Description

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, 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.

## Specifications

Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need to be only 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 below:



Other specifics include the following:

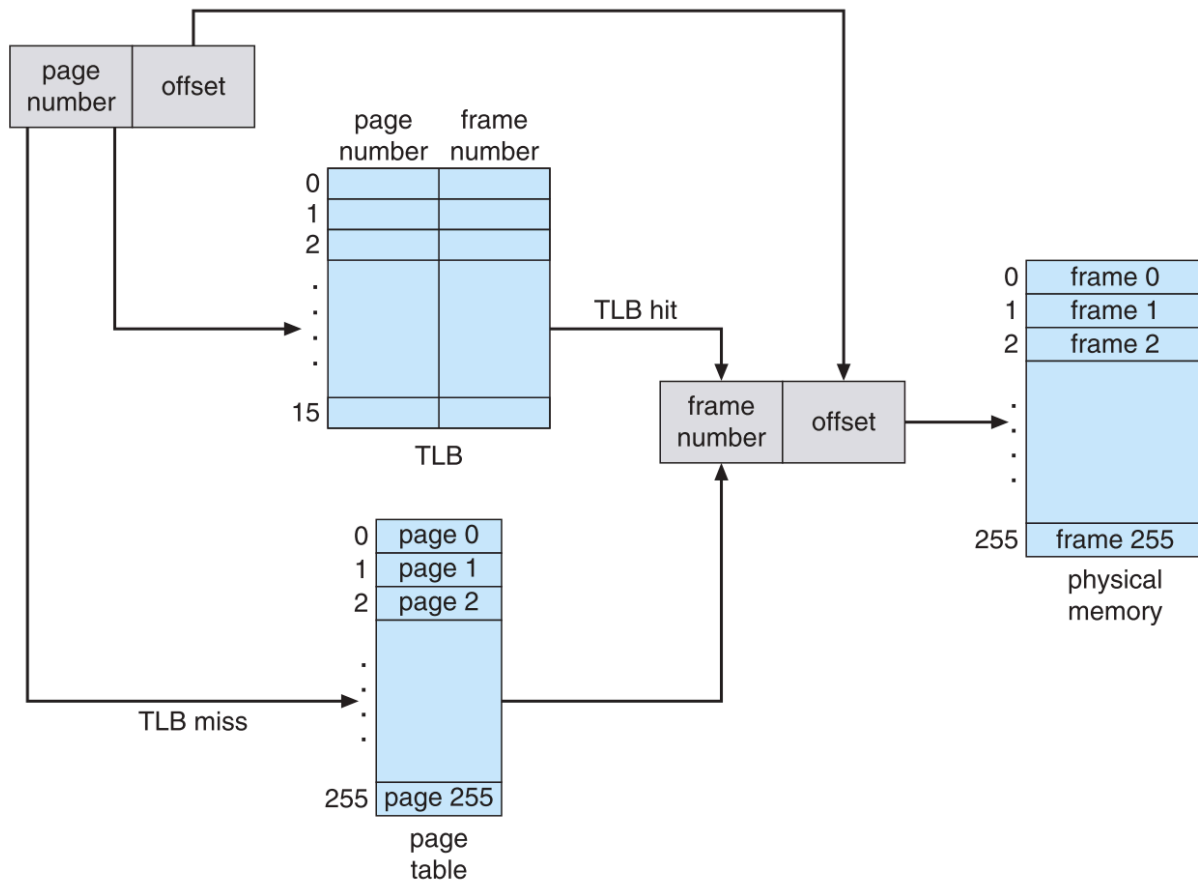
- $2^8$  entries in the page table
- Page size of  $2^8$  bytes
- 16 entries in the TLB
- Frame size of  $2^8$  bytes
- 256 frames
- Physical memory of 65,536 bytes (256 frames  $\times$  256-byte frame size)

Additionally, your program needs to be only 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.

## Phase 1 – No Page Replacement

### 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:



## Handling Page Faults

Your program will implement demand paging as described in Section 10.2. The backing store is represented by the file `BACKING_STORE.bin`, a binary file of size 65,536 bytes located in the StarterKit directory.

When a page fault occurs, you will read in a 256-byte page from the file `BACKING_STORE.bin` 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.bin` (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()`.

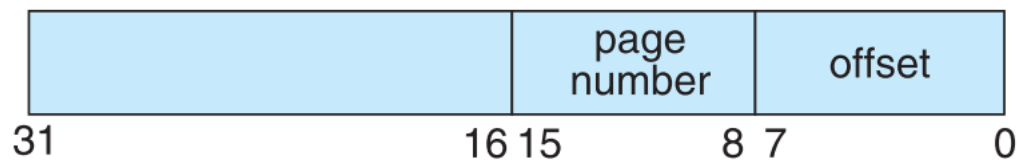
The size of physical memory is the same as the size of the virtual address space, i.e., 65,536 bytes, so you do not need to be concerned about page replacements during a page fault at this phase. Later, in phase 2, we describe a modification to this project, assuming a smaller amount of physical memory, for which a page-replacement strategy will be required.

### Test File

We provide 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.

### 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 16 entries, so you will need to use a replacement strategy when you update a full TLB. **FIFO** policy should be used for updating the TLB.

## Phase 2 – With Page Replacement

### 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. So at this phase, we have at most  $2^7$  valid entries in the page table (i.e., 128 pages). 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 **LRU** (Section 10.4) to resolve page faults when there is no free memory.

---

## How to Run Your Program

Your program should 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 a comma-separated values (csv) file that has three columns:

1. Column 1: the **logical address** being translated (the integer value being read from `addresses.txt`).
2. Column 2: the corresponding **physical address** (what your program translates the logical address to).
3. Column 3: the **signed byte value** stored in physical memory at the translated physical address.

We also provide the files `correct128.csv` and `correct256.csv`, which contain 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.

## Statistics

After completion, your program is to report the following statistics for both phase 1 and 2 at the end of the csv files:

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.

Please check the end of file `correct128.csv` and `correct256.csv` to see the correct format for reporting the statistics.

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.

## How to Test Your Program

The file `correct256.csv` is the correct output for `addresses.txt` for **phase 1** of this project.

You first need to complete the Makefile and then use `test.sh` script to test your project (i.e., you should edit the Makefile to reflect what files you created and how to compile them).

You don't need to perform error checking for out-of-bound values in the input file for this project.

Do **not** hardcode the file names `BACKING_STORE.bin` and `addresses.txt` in your source code. These are command-line arguments that will be passed along with the physical memory size to your program, as shown in the `test.sh` file.

During marking, the TAs will be passing other input files that are **not** necessarily named `BACKING_STORE.bin` or `addresses.txt` to your program. Your program should work regardless of the file name passed to it.

### Deliverables

Submit a **zip** file, `project_mmu_yourfirstname.zip`, containing all the files that are required to build and run your project, including:

- 1) `Makefile` (you should change/edit this file)
- 2) **All C source and header files**
- 3) `BACKING_STORE.bin` (do not change/edit this file)
- 4) `addresses.txt` (do not change/edit this file)
- 5) `test.sh` (do not change/edit this file)
- 6) `correct128.csv` (do not change/edit this file)
- 7) `correct256.csv` (do not change/edit this file)

Please do not submit object files (`*.o`) or compiled executables.

All your files should be in one folder, meaning, do not create a folder called `StarterKit`.

Instead your zip file should have all the files directly in the Zip.

### Grading

The TAs will use `test.sh` bash script to grade your project. The script will be edited to use different file names in place of the `addresses.txt`, `correct128.csv`, and `correct256.csv`.

As can be seen in the `test.sh`, we first **make** your project using your submitted **Makefile**

Sample `test.sh` output:

```

red1:/eecs/home/alomari/3221/project_mmu_alomari
File Edit View Search Terminal Help
red1 321 % test.sh
Compiling and Buidling the MMU
rm -rf *.o
rm -rf mmu
gcc -Wall -o mmu mmu.c

----- Phase 1 -----
Running Phase 1
Grading Phase 1 ...
Good Job -- 8 points

----- Phase 2 -----
Running Phase 2
Grading Phase 2 ...
Good Job -- 12 points

red1 322 %

```

### Grading Phase1

After running make as mentioned above, we run:

```
./mmu 256 BACKING_STORE.bin addresses.txt
```

in which, your executable, i.e., **mmu**, will be given 3 parameters: 256 as the size of the physical memory, **BACKING\_STORE.bin** and **addresses.txt**, which contains the logical input addresses.

At this phase, your program should create a file called **output256.csv**, which will be compared against the **correct256.csv**.

### Grading Phase2

For phase 2, we change the size of the physical memory to 128 by running:

```
./mmu 128 BACKING_STORE.bin addresses.txt
```

Here, your program should create a file called **output128.csv**, which will be compared against the **correct128.csv**.

**Note:** Ensure you run **test.sh** before submitting your file to ensure all is running properly. For statistics, you should set the floating-point precision to 2.

### Rubric

Item	Points
Correct <b>output256.csv</b> file for no page replacement	6

Correct statistics for no page replacement (1 point for TLB-hit rate and 1 point for page-fault rate)	2
Correct <b>output128.csv</b> file for page replacement	9
Correct statistics for page replacement (1.5 points for TLB-hit rate and 1.5 points for page-fault rate)	3
<b>Sum</b>	<b>20</b>

TAs will evaluate your project on one of the Department's servers with a specific configuration. You need to configure your account on one of the department servers according to this specific configuration. To do so, check the guidelines under the Environment Setup section.

### Collaboration

Here are the allowed activities while working on this project:

1. **Discussing Problem-Solving Approaches:** You may discuss strategies for solving a problem. However, to preserve the learning experience of others, avoid revealing key solutions directly and instead guide them toward discovering the solutions themselves.
2. **Syntax and Bug Discussions:** You can discuss syntax issues and bugs in your code, provided you do not show your code to others. For example, verbally discussing issues is allowed, but screen sharing via tools like Zoom is not.
3. **Using Small Code Snippets:** Using brief, publicly available code snippets for solving minor problems, such as iterating through an array, is permitted. These must be cited in comments within your code.

While working on this project, it is not allowed to be:

1. **Viewing Another Student's Code:** You may not view another student's project code to understand an idea or solve a problem, nor share your own code for this purpose.
2. **Possessing Unauthorized Code:** You may not possess project solution code you didn't write yourself, nor another student's code, in any form (electronic, physical, etc.), even for debugging purposes. Sharing such code is equally prohibited.

**Important Note:** Your code will be run through code plagiarism checkers.

### Environment Setup

This section provides guidance for setting up EECS servers for remote development in C. You will need to configure the server and set up a client IDE (e.g., Visual Studio Code) to use the server as a backend for development.

You can use any EECS department server (e.g., red, red1, crimson, etc.) to write your code.

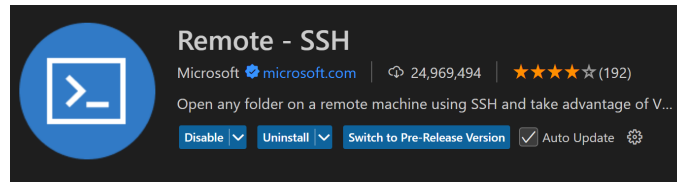
There are several ways you can use the servers:

**Option 1:** Physically visit one of the EECS labs, log in locally to the machine and start developing your code.

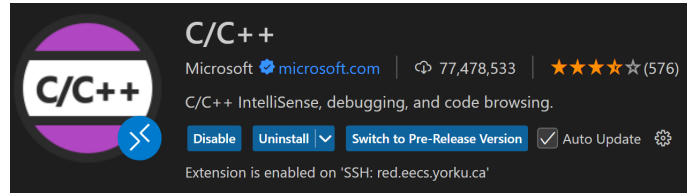
**Option 2:** If you can't be in the lab, you can still log in remotely to <https://remotelab.eecs.yorku.ca> and start developing your code.

**Option 3:** You can set up your local IDE to connect to the remote server following the example steps below:

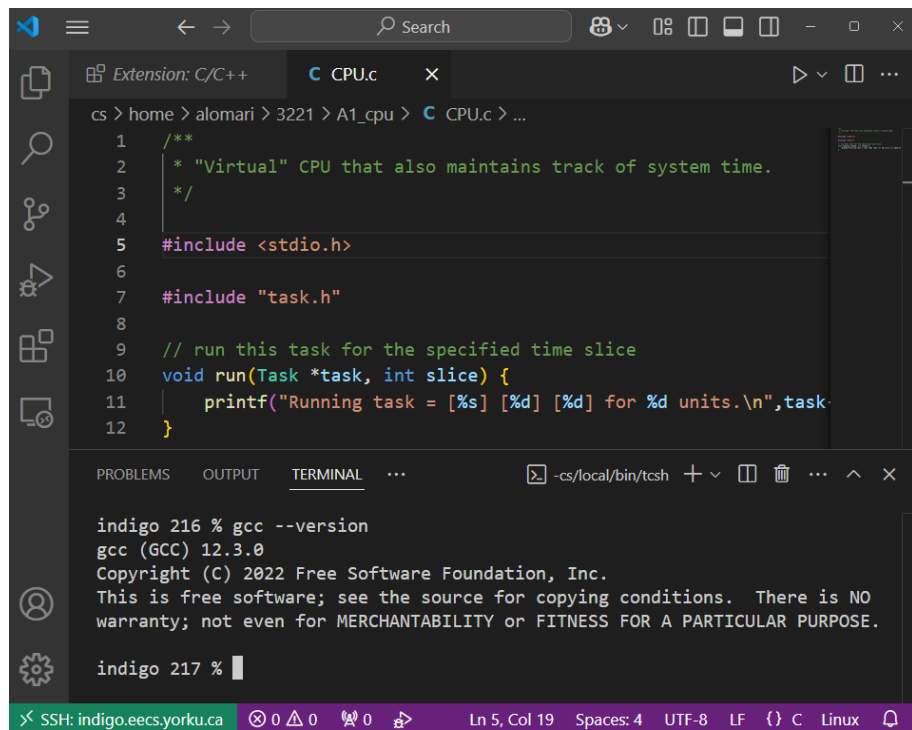
1. Download and install Visual Studio Code on your local machine.
2. Install Required Extensions:
  - Remote - SSH Extension: Allows remote connections to servers.



- C/C++ Extension: Enables C/C++ development.



3. Create a remote connection to your, e.g., red1, server.
4. Open the terminal inside VSCode and verify the versions of GCC and make.



The screenshot shows a Visual Studio Code editor window with a C file named `CPU.c`. The code is a simple program that prints the task name and slice. The terminal at the bottom shows the output of the `gcc --version` command, indicating that GCC 12.3.0 is installed.

```
cs > home > alomari > 3221 > A1_cpu > CPU.c > ...
1  /**
2   * "Virtual" CPU that also maintains track of system time.
3   */
4
5  #include <stdio.h>
6
7  #include "task.h"
8
9  // run this task for the specified time slice
10 void run(Task *task, int slice) {
11     printf("Running task = [%s] [%d] [%d] for %d units.\n", task->name, task->id, task->priority, slice);
12 }
```

```
indigo 216 % gcc --version
gcc (GCC) 12.3.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

indigo 217 %
```

You are all set to start the development of your project code.

If you encounter login or configuration issues, contact EECS IT support.