COMP30023: Computer Systems
Project 1: Virtual Memory Management

Released: Thursday, 27th March, 2025
Due: Sunday, 20th April, 2025 at 11:59pm

Weight: 15% of the final mark

1 Project Overview

In this project, you will write a program that simulates a virtual memory manager. The memory manager will rely on the use of a TLB and a page table to translate logical addresses to physical addresses and will manage the allocation of logical pages to physical frames. For simplicity and to avoid implementing any logic related to process management (e.g., process scheduling), you will assume memory management is done for a single process that runs for the duration of the simulation.

The input to your program will be an ordered sequence of memory accesses (see Section 7). For simplicity, all memory accesses are assumed to be *read* operations (i.e., your program does not need to manage memory accesses that update/write data). For each memory access, the memory manager will extract the page number and the offset from the logical address (see Section 3) and translate it into a physical address by using the TLB and/or page table (see Sections 4 and 6). In cases in which a page is being accessed but physical memory is full (i.e., all frames are allocated), your program will rely on a FIFO page replacement algorithm to decide which page to evict in order to load the new page into memory (see Section 5).

2 Virtual and Physical Memory Specifications

You will simulate a system with the following memory specifications:

Logical Address Space:

- Size: 2^{22} bytes (4MB)
- Page size: 2^{12} bytes (4KB)
- Number of pages: 2^{10} pages (1024 pages)

Physical Address Space:

- Size: 2^{20} bytes (1MB)
- Frame size: 2^{12} bytes (4KB)
- Number of frames: 2^{8} frames (256 frames)

TLB:

- Number of entries: 32

3 Task 1: Logical Address Parser

Logical addresses given as input to your program (as part of each memory access outlined in the input file) will need to be translated into physical addresses. For this purpose, your first task is to identify the page number and the offset associated with each of the given logical memory addresses.

In subsequent tasks, you will use this information to translate logical addresses into physical ones by relying on the data stored in the TLB and/or page table.

At the conclusion of this task, your program should be able to:

1. Extract the page number and offset from the logical address [**Task 1**]
2. Print an execution transcript as defined in Section 8.1 [**Task 1**]

Since the address space supported by your simulator is 2^{22} bytes, your program will need to parse logical addresses that are 22 bits long and include a 10-bit page number, and a 12-bit offset. Addresses given as input to your program, however, will be in the form of 32-bit unsigned integers. This means that you must mask the leftmost (most significant) 10 bits of each input address, and consider only the rightmost (least significant) 22 bits for parsing purposes, as illustrated in Figure 1.

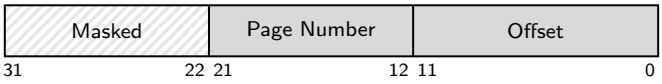

Figure 1: Structure of the 32-bit logical addresses given as input to the simulator

Below is an example of the page number and offset extracted from a 32-bit unsigned integer:

- 32-bit unsigned integer - input (decimal): 30023
- 32-bit unsigned integer - input (binary): 00000000000000000111010101000111
- Leftmost 10 bits - masked (binary): 0000000000
- Rightmost 22 bits (binary): 0000000111010101000111
- Page Number (binary): 0000000111
- Page Number (decimal): 7
- Offset (binary): 010101000111
- Offset (decimal): 1351

4 Task 2: Page Table Implementation and Initial Frame Allocation

Now that your program can extract the page number and offset from a logical address, you will implement the system's page table and use it to translate logical addresses to physical ones.

At the conclusion of this task, your program should be able to:

1. Extract the page number and offset from the logical address [Task 1]
2. Use the page number to identify the relevant page table entry [**Task 2**]
3. In case of a page fault, allocate the page to a free frame in increasing frame number and update the corresponding page table entry [**Task 2**]
4. Use the information in the page table entry and the offset to construct the corresponding physical address [**Task 2**]
5. Print an execution transcript as defined in Section 8.2 [**Task 2**]

Details relevant to this task are outlined below.

- The page table has 1024 entries (see Section 2), numbered from 0 to 1023.
- Each page table entry should contain the following:

1. A present/absent bit (1 if the page is mapped to a frame, 0 otherwise)
2. A frame number

- Physical memory is partitioned into 256 frames, numbered from 0 to 255.
- Pages should be allocated to frames in increasing frame number.
- At the start of the simulation, all physical memory is empty (i.e., all frames are free).
- For this task, you will assume that the number of pages referenced by the running process does not exceed the number of frames in the system (the process references a maximum of 256 pages). This means there will always be free frames, and you will not have to evict pages or implement a page replacement algorithm (this will be done in Task 3). Note that we will not test your implementation of Task 2 with inputs that reference more than 256 pages.

Please note that while you are not required to track the actual data stored in a frame (or in disk), you are required to maintain a data structure that allows you to track frames that are free (e.g., a free-frame list).

5 Task 3: Page Replacement Algorithm

In this task, the running process is allowed to reference more pages than there are available frames. This means that, unlike Task 2, all frames might be allocated when a page fault occurs. In such a case, you are required to evict a page from physical memory by following the FIFO page replacement algorithm, as covered in the lectures and the textbook.

At the conclusion of this task, your program should be able to:

1. Extract the page number and offset from the logical address [Task 1]
2. Use the page number to identify the relevant page table entry [Task 2]
3. In case of a page fault:
   (a) If there are free frames, allocate the page in increasing frame number [Task 2]
   (b) If there are no free frames, free up a frame according to the FIFO algorithm and allocate the referenced page to it **[Task 3]**
   (c) Update the page table accordingly **[Task 2 and 3]**
4. Use the information in the page table entry and the offset to construct the corresponding physical address [Task 2]
5. Print an execution transcript as defined in Section 8.3 **[Task 3]**

6 Task 4: TLB

In this task, you will incorporate a TLB into the address translation process. Your program will first check the TLB for an entry corresponding to the page that is being referenced by the memory access. If such an entry exists, your program will use the information in the TLB to construct the corresponding physical address. Otherwise, your simulator will access the page table and behave as specified in Tasks 2 and 3.

At the conclusion of this task, your program should be able to:

1. Extract the page number and offset from the logical address [Task 1]
2. Use the page number to look for a valid entry in the TLB **[Task 4]**, otherwise
3. Use the page number to identify the relevant page table entry [Task 2]
4. In case of a page fault:
   (a) If there are free frames, allocate the page in increasing frame number [Task 2]

(b) If there are no free frames, i) free up a frame according to the FIFO algorithm and allocate the referenced page to it [Task 3], and ii) flush (i.e., remove) the TLB entry corresponding to the evicted page if necessary **[Task 4]**

(c) Update the page table accordingly [Tasks 2 and 3]

5. Update the TLB according to a LRU policy **[Task 4]**
6. Use the information in the page table entry and the offset to construct the corresponding physical address [Task 2]
7. Print an execution transcript as defined in Section 8.4 **[Task 4]**

Details relevant to this task are outlined below.

- The TLB has 32 entries.
- Each TLB entry should contain the following:
  1. A page number
  2. A frame number
- After handling a TLB miss, the TLB must be updated to include an entry for the page that caused the TLB miss. If the TLB is full, then a LRU replacement strategy must be followed, where the least-recently-used entry in the TLB gets evicted to make room for the most recently accessed page.
- If there is an entry in the TLB for a page that has been evicted from memory (following the process explained in Section 5), then the TLB must be updated by flushing the corresponding entry. Flushing in the case of our program, refers to the process of eliminating an entry from the TLB because that entry is not valid anymore (i.e., the page is not allocated to a frame in memory).

7 Program Specification

Your program must be called `translate` and take the following command-line arguments:

```
Usage: translate -f <filename> -t <task>
```

`-f` *filename* will specify a valid *relative* or *absolute* path to the input file containing the memory accesses, following the format defined in Section 3. Each line (including the last) will be terminated with an LF (ASCII `0x0a`) control character.

`-t` *task* where *task* corresponds to one of the project's tasks and is one of {`task1`, `task2`, `task3`, `task4`}.

The arguments can be passed **in any order**, but you can assume that all the arguments will be passed correctly and each argument will be passed exactly once.

7.1 Input File

Each line in the input file contains a 32-bit unsigned integer representing a logical address. These addresses must be processed in the order in which they appear in the file.

Example:

Given the input file `addresses.txt` with the following information:

```
30023
10001
20007
```

Your program should simulate the management of three memory access; the first access is to logical address `30023`, the second one to logical address `10001`, and the third one to logical address `20007`.

No assumptions may be made about the number of lines in the input file. You can, however, assume that the input files we will use to test your program are such that simulations will complete in a reasonable amount of time.

In addition, no assumptions may be made about the length of the file name (`filename`).

You can read the whole file before starting the simulation or read one line at a time.

We will not give malformed input. If you want to reject malformed command-line arguments or input, your program should exit with a non-zero exit code per convention.

8 Expected Output

Your program must print an execution transcript to standard output (`stderr` will be ignored). The output expected for each of the project's tasks is defined below. Your program must conform to the expected output exactly, as we will use this output to verify that your code meets the project's specification through automated tests.

8.1 Task 1 - Output

Each time your simulator parses an input logical address, your program must print out a line in the following format:

```
logical-address=<laddress>,page-number=<pnumber>,offset=<poffset>
```

where:

- '`laddress`' refers to the input logical address (unsigned 32-bit integer without masking);
- '`pnumber`' refers to the page number extracted from the logical address;
- '`poffset`' refers to the offset extracted from the logical address.

Sample output:

```
logical-address=30023,page-number=7,offset=1351
logical-address=10001,page-number=2,offset=1809
logical-address=20007,page-number=4,offset=3623
```

8.2 Task 2 - Output

For each input address, this task expects the output of Task 1 to be printed first, followed by the output outlined below. See Section 8.5 for details on the ordering of output.

Each time your simulator translates a logical address to a physical address using the page table, your program must print out a line in the following format:

```
page-number=<pnumber>,page-fault=<pfault>,frame-number=<fnumber>,physical
-address=<paddress>
```

where:

- '`pnumber`' refers to the page number extracted from the logical address;
- '`pfault`' is 0 if the page was allocated in memory or 1 if a page fault occurred;
- '`fnumber`' refers to the number of the frame in memory to which the page was allocated;
- '`paddress`' refers to the physical address corresponding to the input logical address.

Sample output:

```
logical-address=30023,page-number=7,offset=1351
page-number=7,page-fault=1,frame-number=0,physical-address=1351
logical-address=10001,page-number=2,offset=1809
```

```
page-number=2,page-fault=1,frame-number=1,physical-address=5905
logical-address=20007,page-number=4,offset=3623
page-number=4,page-fault=1,frame-number=2,physical-address=11815
```

8.3 Task 3 - Output

For each input address, this task expects the output of Task 1 to be printed first, followed by the output outlined in this section if an eviction occurs, followed by the output of Task 2. See Section 8.5 for details on the ordering of output.

Each time a page is evicted from memory, your program must print out a line in the following format:

```
evicted-page=<epage>,freed-frame=<fframe>
```

where:

- '`epage`' refers to the number of the page that was evicted from memory;
- '`fframe`' refers to the number of the frame that was freed (i.e., where the page was evicted from).

Sample output:

```
logical-address=30023,page-number=7,offset=1351
page-number=7,page-fault=1,frame-number=0,physical-address=1351
...
logical-address=30023,page-number=7,offset=1351
page-number=7,page-fault=0,frame-number=0,physical-address=1351
...
logical-address=90048,page-number=21,offset=4032
evicted-page=7,freed-frame=0
page-number=21,page-fault=1,frame-number=0,physical-address=4032
```

8.4 Task 4 - Output

The output of this task must include the output for all other tasks as defined in Section 8.5, plus the output defined below.

For the following events, your program should print a line with the following format:

- After looking for an entry in the TLB:

  ```
  tlb-hit=<tlbhit>,page-number=<pnumber>,frame=<fnumber>,physical-
  address=<paddress>
  ```

  where:

  - '`tlbhit`' is 0 in case of a TLB miss or 1 in case of a TLB hit (a valid entry was found in the TLB);
  - '`pnumber`' refers to the page number extracted from the logical address;
  - '`fnumber`' refers to the frame number in case of a TLB hit or "`none`" in case of miss;
  - '`paddress`' refers to the physical address corresponding to the input logical address in case of a TLB hit or "`none`" in case of miss.

- After a TLB flush caused by a page eviction (i.e., after a page present in the TLB is evicted and the page is flushed (removed) from the TLB):

  ```
  tlb-flush=<pnumber>,tlb-size=<tsize>
  ```

  where:

  - '`pnumber`' refers to the page number stored in the TLB entry that was flushed;
  - '`tsize`' refers to the size (number of entries) of the TLB after flushing.

- After the TLB is updated following a TLB miss:

  ```
  tlb-remove=<rpnumber>,tlb-add=<apnumber>
  ```

  where:

  - '`rpnumber`' refers to the page number stored in the TLB entry that was removed or "`none`" if entry was free;
  - '`apnumber`' refers to the page number stored in the TLB entry that was added.

Sample output (refer to test cases for more scenarios):

```
logical-address=30023,page-number=7,offset=1351
tlb-hit=0,page-number=7,frame=none,physical-address=none
tlb-remove=none,tlb-add=7
page-number=7,page-fault=1,frame-number=0,physical-address=1351
```

8.5 Ordering of Output

Note that, as defined in each output section above, the final execution transcript produced by your program (which is what is assessed in the automated tests) corresponds to the cumulative output of tasks: Task 1 only prints output for Task 1; Task 2 prints output for Tasks 1 and 2; Task 3 prints output for Tasks 1, 2, and 3; and Task 4 prints output for Tasks 1, 2, 3, and 4.

The execution transcript must follow the order dictated by the algorithm, i.e.:

- Task 1 (Section 8.1), `logical-address=...`
- Task 4 (Section 8.4), `tlb-hit=...`
- Task 3 (Section 8.3), `evicted-page=...`
- Task 4 (Section 8.4), `tlb-flush=...`
- Task 4 (Section 8.4), `tlb-remove=...`
- Task 2 (Section 8.2), `page-number=...`

The above order must be followed strictly.

Each task (as specified by `-t <task>`) should have the behaviour of the current and preceding tasks only. e.g., `-t task3` must not involve accessing a TLB in the background.

Only relevant lines (as specified for each Task) should be printed per address in the input.

If the input file is empty, no output should be printed.

9 Marking Criteria

The marks are broken down as follows:

Task	**Marks**
Task 1: Logical Address Parser (Section 3)	2
Task 2: Page Table Implementation and Initial Frame Allocation (Section 4)	4
Task 3: Page Replacement Algorithm (Section 5)	3
Task 4: TLB (Section 6)	4
Build quality	1
Quality of software practices	1
TOTAL	15

Assessment of Tasks 1-4 Tasks 1, 2, 3, and 4 will be assessed through automated testing. We will compile your code and run it against a set of test cases. We will compare the output produced by your program against the expected output of each test case. This is why it is essential that you follow the specification and produce output exactly as outlined in Section 8.

You will be given access to a subset of the test cases used to assess your project as well as their expected outputs. Half of the marks for Tasks 1 - 4 will be awarded based on this subset of visible test cases. The other half of the marks will be determined based on a different subset of hidden test cases not available to you.

Because we compile and run your code, it is a requirement that your code **compiles and runs on the provided VMs** and produces deterministic output. Code that does not meet these requirements cannot be assessed and hence will receive 0 marks (at least) for Tasks 1 - 4.

Build quality

- The repository must contain a `Makefile` that produces an executable named "`translate`", along with all source files required to compile the executable. Place the `Makefile` at the root of your repository, and ensure that running `make` places the executable there too.
- Make sure that all source code is committed and pushed.
- Running `make clean && make -B && ./translate <...arguments>` should execute the submission.
- Compiling using "`-Wall`" should yield no warnings.
- Running `make clean` should remove all object code and executables.
- Do not commit `translate` or other executable files (see Practical 1). Scripts (with `.sh` extension) are exempted.
- The automated test script expects `translate` to exit with status code 0 (i.e. it successfully runs and terminates).

The mark calculated for "Build quality" will be visible on CI (see Section 11).

Quality of software practices Factors considered include:

- **Proper use of version control**, based on the regularity of commit and push events, their content and associated commit messages (e.g., repositories with a single commit and/or non-informative commit messages will lose 0.5 marks).
- **Quality of code**, based on the choice of variable names, comments, formatting (e.g. consistent indentation and spacing), and structure (e.g. abstraction, modularity).
- **Proper memory management**, based on the *absence* of memory errors and memory leaks.

Further deductions may be applied to inappropriate submissions, e.g. catching segmentation faults, hard-coding the output into the code.

10 Submission

Programming language All code must be written in C (e.g., it should not be a C-wrapper over non C-code).

Use of libraries Your code will likely rely on data structures to manage processes and memory. You may write your own code, reuse any existing code you might have written (e.g., in other subjects), or use external libraries for this (and only this) purpose.

External libraries cannot be used in your project for any other purpose.

You may use standard libraries (e.g. to read files, sort, parse command line arguments[1] etc.).

[1] https://www.gnu.org/software/libc/manual/html_node/Getopt.html

GitHub The use of `GitHub` is mandatory. Your submission will be assessed using the code in your Project 1 repository (`proj1-⟨username⟩`) under the subject's organization.

We strongly encourage you to commit your code at least once per day. Be sure to `push` after you `commit`. This is important not only to maintain a backup of your code, but also because the git history may be considered for matters such as special consideration, extensions and potential plagiarism. Proper use of `git` will have a positive effect on the mark you get for quality of software practices.

Submission To submit your project, please follow these steps carefully:

1. Push your code to the repository named `proj1-⟨username⟩` under the subject's organization, https://github.com/feit-comp30023-2025.

   Executable files (that is, all files with the executable bit that are in your repository) **will be removed** before marking. Hence, ensure that none of your source files have the executable bit.

   Ensure your code **compiles and runs on the provided VMs**. Code that does not compile or produce correct output on VMs will typically receive very low or 0 marks.

2. **Submit the full 40-digit SHA1 hash** of the commit you want us to mark to the **Project 1 Assignment** on the LMS.

   You are allowed to update your chosen commit by resubmitting the LMS assignment as many times as desired. However, only the last commit hash submitted to the LMS before the deadline (or approved extension) will be marked without a late penalty.

3. Ensure that the commit that you submitted to the LMS is correct and accessible from a fresh clone of your repository. An example of how to do this is as follows:

   ```
   git clone git@github.com:feit-comp30023-2025/proj1-<username> proj1
   cd proj1
   git checkout <commit-hash-submitted-to-lms>
   ```

   Please be aware that we will only mark the commit submitted via the LMS. It is your responsibility to ensure that the submission is correct and corresponds to the commit you want us to mark.

Late submissions Late submissions will incur a deduction of 2 marks per day (or part thereof). For example, a submission made 1 hour after the deadline is considered to be 1 day late and carries a deduction of 2 marks.

We strongly encourage you to allow sufficient time to follow the submission process outlined above. Leaving it to the last minute usually results in a submission that is a few minutes to a few hours late, or in the submission of the incorrect commit hash. Either case leads to late penalties.

The submission date is determined **solely** by the date in which the LMS assignment was submitted. Forgetting to submit via the LMS or submitting the wrong commit hash will result in a late penalty that will apply regardless of the commit date.

We will not give partial marks or allow code edits for either known or hidden cases without applying a late penalty (calculated from the deadline).

Extension policy For extensions between 1-3 business days, you must:

1. Have an AAP or fill in FEIT's short extension declaration form before the project's deadline
2. Submit an extension request via the following form: https://forms.office.com/r/hRjkf82uxR

For extensions of more than 3 business days, you must:

1. Apply for an extension via the special consideration portal before the assessment deadline
2. Receive a successful outcome for your application

3. Submit the outcome of your application via the following form: https://forms.office.com/r/SYNsEGB3kG

Further details are available on the "FEIT Extensions and Special consideration" page on Canvas (under the Welcome module).

11 Testing

You will be given access to several test cases and their expected outputs. However, these test cases are not exhaustive and will not cover all edge cases. Please be aware that the test suite (visible and hidden cases combined) will aim to cover **all of the business rules** that are written in the specification. Hidden test cases will also be **more difficult**. Hence, you are strongly encouraged to write tests to verify the correctness of your own implementation.

Testing Locally: You can clone the sample test cases to test locally, from: `feit-comp30023-2025/project1`.

Continuous Integration Testing: To provide you with feedback on your progress before the deadline, we have set up a Continuous Integration (CI) pipeline on `GitHub` with the same set of test cases.

Though you are strongly encouraged to use this service, the usage of CI is not assessed, i.e., we do not require CI tasks to complete for a submission to be considered for marking.

The requisite `ci.yml` file has been provisioned and placed in your repository, but is also available from the `.github/workflows` directory of the `project1` repository linked above.

12 Collaboration and Plagiarism

The program you submit as part of this assignment must be your own work.

You cannot copy work from another student or from any other source, such as online repositories. Do **not** share your code, and do **not** ask others to give you their programs. Do **not** post your code on the subject's discussion board Ed. The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program. See https://academicintegrity.unimelb.edu.au for more information.

Note that soliciting solutions by any means (e.g., via posts to online forums), whether or not there is payment involved, is also Academic Misconduct.

You should not post your code to any public location (e.g., in your GitHub account) before receiving the final marks for the subject. This ensures you are not breaching the Academic Integrity Policy by enabling other students in the same semester to complete their project. You are not allowed to post the project's specification in any public forum at any stage, as it is copyright of the University.

The use of AI tools is strictly prohibited. This project must be completed without the help of AI, with the final product expected to be your own work.