CA1 - Cache Management

Due 21 Nov 2021 by 23:59 **Points** 100 **Submitting** an external tool
Available 8 Nov 2021 at 0:00 - 30 Nov 2021 at 17:00

This assignment was locked 30 Nov 2021 at 17:00.

COMP517 Continuous Assessment Task 1

Assignment Number: 1 of 3

Deadline for submission: **16th November 2021, 17.00 (updated deadline)**

Submission Mode: Please provide your solutions electronically via Canvas. **You should submit a single .py file.**

Purpose of the assessment: This assignment will test the ability of the students to work with basic Python instructions, and to use and manipulate collections in Python.

Learning Outcomes Assessed:

LO1. Demonstrate knowledge of fundamental imperative programming concepts such as variables and assignment, conditional statements, loops and methods.

LO2. Be able to design and code applications in a suitable programming language.

Marking Criteria: Based on the marking descriptors of the University's Code of Practice on Assessment

Late submissions: For late submissions the standard penalties apply, according to UoL policy.

Plagiarism: Please provide individual answers and do not coordinate with other students to complete the assignment. Check the official UoL guidelines for plagiarism and academic integrity.
The submitted files will be checked using tools for plagiarism and collusion.

You may use the "Discussions" to ask clarification questions about the assignment, but please ***do not ask any questions that would give away the solutions or even parts of the solutions.***

Continuous Assessment Task 1 - Cache Management

Background:

In computers, memory is located in different parts of the computer; memory on disk is far away from the CPU and takes a long time to access, but offers a large amount of storage; cache memory is close to the CPU so it is faster to access, but does not offer a lot of storage space.

It is therefore wise to use the cache memory in a smart way, so that pages of memory that are more likely to be accessed are already in the cache. This means that there are fewer accesses to disk memory, and the memory transaction is ultimately quicker.

In this assignment, you will use Python to *simulate* a cache with two management techniques, which are explained below.

The cache memory operates in the following manner:

Pages from memory are requested. When a page from memory is requested, the cache memory is searched for that page.

- If the page is found in the cache memory, then this is called a "hit".
- If the page is not found in the cache memory, then this is called a "miss". When a miss happens, the page must be retrieved from the main memory (the disk memory) and must be placed in the cache memory. If the cache memory is not full, then the page is simply added to the cache memory. If the cache memory is full, then one of the pages that is in the cache memory will have to be replaced. In this case, we say that a new page is added to the cache memory, and the old page is *evicted*.

There are multiple ways in which we can choose which page to evict. Two of those are presented below, the "First in First Out (FIFO)" algorithm, and the "Least Frequently Used (LFU)" algorithm.

First in First Out (FIFO)

In a First in First Out (FIFO) cache memory, the page that is evicted is the one that has **the longest time since it was added.**

Least Frequently Used (LFU)

In a Least Frequently Used (LFU) cache memory, the page that is evicted is the **page that has had the fewest requests so far. In case of two pages having the same amount of requests, the lowest numbered page should be evicted. The number of requests that a page has had is maintained throughout the parsing of the whole set of requests, and it is not "forgotten" once a page has been removed from the cache memory.**

Specifics:

In this exercise, we will assume that

- every page is represented by a **positive integer**.
- the **capacity** of the cache memory is **8 pages**.

In particular, a request for a page will be indicated by a number, e.g., the number 3 means that we request page 3. If the requested page is in the cache memory, then this will be a hit. If the requested page is not in the cache memory, then this will be a miss. In this latter case, the page has to be retrieved from the main memory and placed into the cache memory. **If the cache memory is not full** (i.e., it has fewer than 8 pages already in it), then we can simply add the requested page to the cache memory. **If the cache memory is full** (meaning that it has exactly 8 pages in in), then we have to evict one page already in the cache memory, in order to bring the newly requested page in. Which page to evict is decided based on one of the two algorithms above, FIFO or LFU.

Program Structure

Your task is to write a program that implements the memory management operation, as described above.

Your program **must have** the following functions and variables:

- A global scope list called "`cache`"
- A global scope list called "`requests`"
- A function `fifo()` which will run the FIFO cache memory algorithm on `cache` and `requests`
- A function `lfu()` which will run the LFU cache memory algorithm on `cache` and `requests`

Clarification: After each method has been used for the given requests, the cache should be cleared, meaning that the list "`cache`" should not contain any elements.

Constructing the list of requests: Your program should ask the user for an integer repeatedly until `0` is entered. As mentioned above, these integers represent requested pages, and `0` signifies the end of the request input. The inputted integers should be placed into the list `requests.`

Choosing the eviction algorithm: The user should then be presented with the following options: *press 1 for fifo*, or *press 2 for lfu,* or *press Q to quit the program*.

- If the user chooses `1,` then the function `fifo()` should be called.
- If the user chooses `2`, then the function `lfu()`) should then be called.
- If the user chooses `Q`, the program should terminate.

Whichever of the two functions is chosen, it should iterate over the requests in the list `requests`, and "request" each "page" (represented simply as an integer) from the cache memory.

- If the “page” is already in the cache memory `cache`, then this is a hit and the word “hit” should be printed to the screen.
- If the “page” is not in the cache memory `cache`, then this is a miss and the word “miss” should be printed to screen.

  In the event of a miss, the correct “page” should be evicted from the cache memory and the newly requested “page” should be correctly inserted into cache memory, according to the eviction algorithm that has been selected by the user earlier. In reality, this means that your program will remove an integer from `cache` and insert a new integer into `cache`.

  Important: The newly introduced page should be added to the end of the cache list. For example, if we have the cache [4, 6, 11, 77, 32, 5, 7, 8] and a new page to add, e.g., page 20, we evict some page according to our algorithm (e.g., page 32) and the cache now becomes [4, 6, 11, 77, 5, 7, 8, 20]. **If your list appears in a different order you might not pass some of the tests.**

At the end of processing all requests, you should then print the final state of cache (i.e., the contents of the list `cache`) and exit back to the main menu, where the user is prompted to enter the requests (recall that the cache will have been "cleared" after its final contents will have been displayed on the screen).

Examples

Consider the following list of requests [2,3,6,5,2,7], which have been inputted by the user, and consider a cache memory of size 4.

Remember, that each “page” of memory is simply an integer.

The cache management algorithm could proceed as follows:

Initially, *cache = []*

FIFO:

Request	Hit/Miss	Cache
2	Miss – add 2 to cache	[2]
3	Miss – add 3 to cache	[2,3]
6	Miss – add 6 to cache	[2,3,6]
5	Miss – add 5 to cache	[2,3,6,5]

2	Hit – already got 2 in cache	[2,3,6,5]
7	Miss - Evict 2, add 7 to Cache	[3,6,5,7]

LFU:

Request	Hit/Miss	Cache
2	Miss – add 2 to cache	[2 (1 request)]
3	Miss – add 3 to cache	[2 (1 request) ,3 (1 request)]
6	Miss – add 6 to cache	[2 (1 request) ,3 (1 request) ,6 (1 request)]
5	Miss – add 5 to cache	[2 (1 request) ,3 (1 request) ,6 (1 request) ,5 (1 request)]
2	Hit – already got 2 in cache so increment number of requests	[2 (2 requests) ,3 (1 request) ,6 (1 request) ,5 (1 request)]
7	Miss - Evict 3 as the smallest page number with lowest number of requests in the cache, add 7 to Cache	[2 (2 requests) , 6 (1 request) ,5 (1 request), 7 (1 request)]

CAUTION: The order of the elements of cache in your implementation will likely be different from that shown here. This is JUST for illustration purposes.

Instructions and Marking Criteria

Your program source file **should** contain a comment at the top with your full name and student number. If it does not, then a penalty will be imposed.

In your program you **should** follow the specified function names and signatures, If you do not follow the specified function names and signatures, then the correctness of your program may not be able to be correctly determined; you may also lose marks for not following the brief.

You **should not** use any libraries, and a penalty will be imposed if libraries are used.

You **should** submit a single file.

You **should** ensure that your program compiles and runs on a departmental lab computer (see CANVAS for more information).

Further to the guidelines on CANVAS, the mark will be calculated with the following proportions:

Correctness and Robustness: 70%

Design Choices: 30%

Note that your ability to provide presentable and well-structured code including comments and appropriate naming of the variables (besides those specified by the requirements of the assignment) is **not assessed**. That being said, you are strongly encouraged to still follow all the guidelines of good practice, as these will be assessed in subsequent assignments.