Queen's School of Computing
CISC324 – Fall 2022
Instructor: Dr. Anwar Hossain
Due Date: November 25th

Background and Outcomes

This lab will be the last in this course. In this lab, we will learn how to implement a simple multi-paging data structure that is being used nowadays in modern operating systems (Part 1). In addition, we will learn the implementation and the difference between three different page replacement algorithms (Part 2).

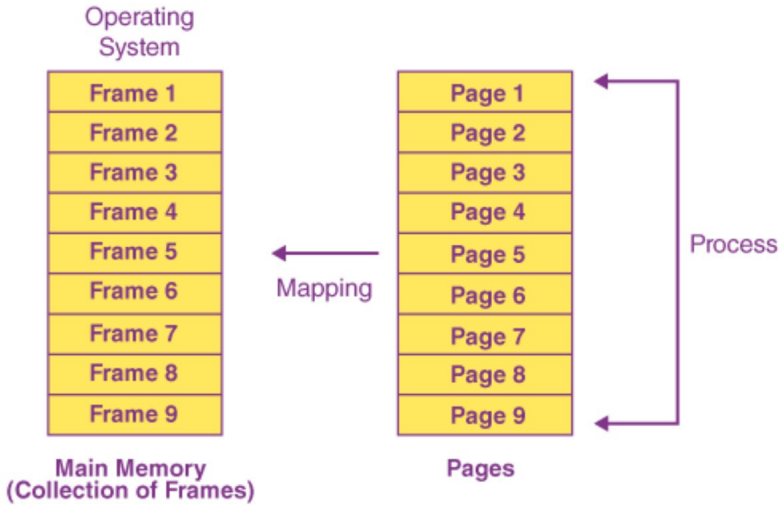
Paging is a method of gaining access to data more quickly. When a program requires a page, it is available in the main memory because the OS copies a set number of pages from the storage device into the main memory. Paging permits a process's physical address space to be non-contiguous. Paging refers to a memory management strategy that does away with the need for the allocation of contiguous physical memory.

What is Paging in the OS?

Paging is a storage mechanism used in OS to retrieve processes from secondary storage to the main memory as pages. The primary concept behind paging is to break each process into individual pages. Thus the primary memory would also be separated into frames.

One page of the process must be saved in one of the given memory frames. These pages can be stored in various memory locations, but finding contiguous frames/holes is always the main goal. Process pages are usually only brought into the main memory when they are needed; else, they are stored in secondary storage.

The frame sizes may vary depending on the OS. Each frame must be of the same size. Since the pages present in paging are mapped onto the frames, the page size should be similar to the frame size.



Queen's School of Computing
CISC324 – Fall 2022
Instructor: Dr. Anwar Hossain
Due Date: November 25th

Example

Assume we have a physical memory of 4 GB (2^{32} bytes) and an operating system that uses a multi-paging mechanism with a page size of 4KB (2^{12} bytes). The total number of frames in physical memory in this example can be easily calculated by dividing the total memory size by the page size according to equation (1)

$$numberOfFrames = \frac{TotalMemorySize}{PageSize} = \frac{2^{32}}{2^{12}} = 2^{20} frames\ (1\ Mega) \qquad (1)$$

When the CPU wants to execute an instruction, it processes the instruction's virtual address and with the help of the OS and the Memory Management Unit (MMU), it converts (maps) the virtual address to the corresponding physical address (frame number + offset).

The virtual address size is 32 bits because we have 2^{32} memory locations so we need 32 bits to represent them. The virtual address is split according to the following figure:

20-bits	12-bits offset

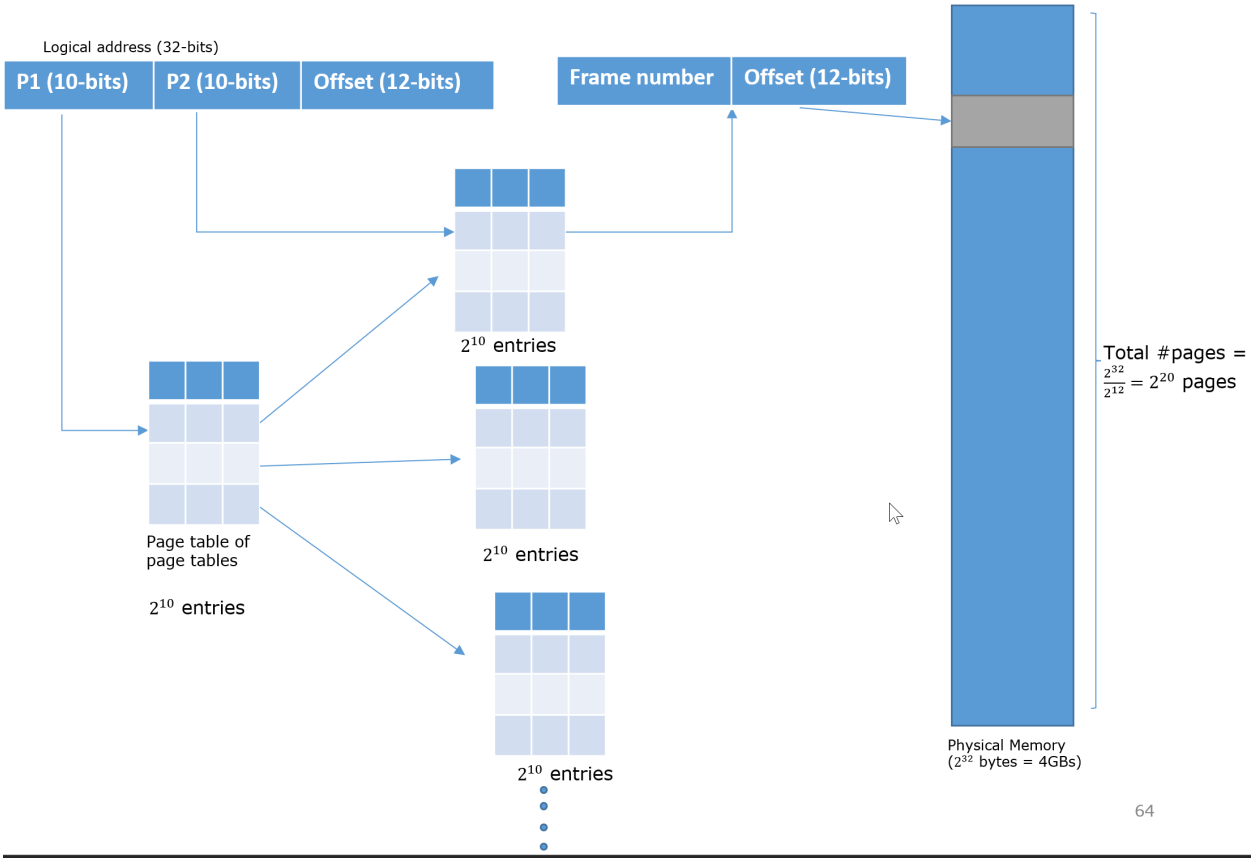
The most significant 20-bits will be used to index inside the page table to get the page table entry that contains the frame number the page in the virtual address should be mapped to.

Since we have 2^{20} frames then we need 2^{20} page table entries to map the whole physical space. But, each entry of this page table is of size 4 bytes because it contains the frame number thus, the whole page table size would be 2^{20} * 4 = 4MB. A page table of size 4 MB is huge because the whole page table will be consuming 4MB of physical memory the whole time.

We can split the page table into multiple page tables to save some space. We can think of designing a page table that consumes at most a single page of physical memory. In this case, we will have an outer page table with 2^{10} entries that make the page table size = $2^{10} * 2^2 = 2^{12}$ bytes. Now, the new page table design can fit inside a single page! Because $2^{12} = 4\ KB$ which is the page size we are using.

Now, each entry of the outer page table will point to another page table with another 2^{10} entries. Now let's check whether we did the paging design correctly or not in the figure below:

Queen's School of Computing
CISC324 – Fall 2022
Instructor: Dr. Anwar Hossain
Due Date: November 25th



The outer page table has 2^{10} entries, each entry points to a page table with 2^{10} entries as well. The total number of pages that can be mapped using this design can be calculated using the formula in equation (2).

$$TotalNumberOfPages = NumberOfEntriesInOuterPageTable * NumberOfEntriesInInnerPageTable$$

In this example, the equation boils down to $2^{10} * 2^{10} = 2^{20}$ pages and that's a correct design because we are actually having 2^{20} frames in total!

Queen's School of Computing
CISC324 – Fall 2022
Instructor: Dr. Anwar Hossain
Due Date: November 25th

Requirements (Part 1)

You are asked to implement a Java program that can perform the following functions:

1. Your program shall prompt the user on the screen to enter the physical memory size (in bytes). In the example above, the physical memory size is 2^{32} bytes (already done for you).
2. Your program shall prompt the user to enter the page size (in bytes). In the example above, the page size is 4096 bytes (already done for you).
3. Your program shall prompt the user to enter the number of bits for the page offset. In the example above, the number of bits for the page offset is 12-bits (already done for you).
4. Your job in this part is to display some information including **(modify class "Part1\PagingStatisticalProcessor.java" in the boilerplate code)**:
    a. The total number of frames.
    b. Print the number of paging levels. In the example above, we have two-level paging. Always bear in mind that each page table can fit in a single page at most and that differs according to the size of the page, memory size, and page offset you are given in the previous points.
    c. The number of bits required to access each page table level. For instance, in the example above, we have two-level paging, thus we have 10-bits for each level. According to the input, it is possible that you are given specs that require an n-level paging design.
    d. The number of entries of the page table at each paging level.
    e. The size in bytes of the page table at each paging level.
    f. You'll be given a program size in bytes, and you are required to display the following:
        i. The number of pages required for this program.
        ii. The number of page tables required to map this program to the physical space.
    g. You'll be given a virtual address in bits ("01101011101…."), and your job is to display the index of each page table entry **in decimal** and the page offset in decimal. For instance, in the example above, a virtual address of the form "01101100011101010111010000101111" is split to 3 parts:

0110110001	1101010111	010000101111

The output should be in decimal:

433	855	1071

433 will be used to index the outer page table, then 855 will be used to index the inner page table and finally 1071 will be used as a displacement inside the physical memory frame.

Queen's School of Computing
CISC324 – Fall 2022
Instructor: Dr. Anwar Hossain
Due Date: November 25th

Requirements (Part 2)

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when a new page comes in.

Page Fault: A page fault happens when a running program accesses a memory page that is mapped into the virtual address space but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of a page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Page Replacement Algorithms:

1. First In First Out (FIFO): This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example 1: Consider page reference string 1, 3, 0, 3, 5, 6, 3 with a memory of size 3-page frames. Find the number of page faults. Check Figure 1.

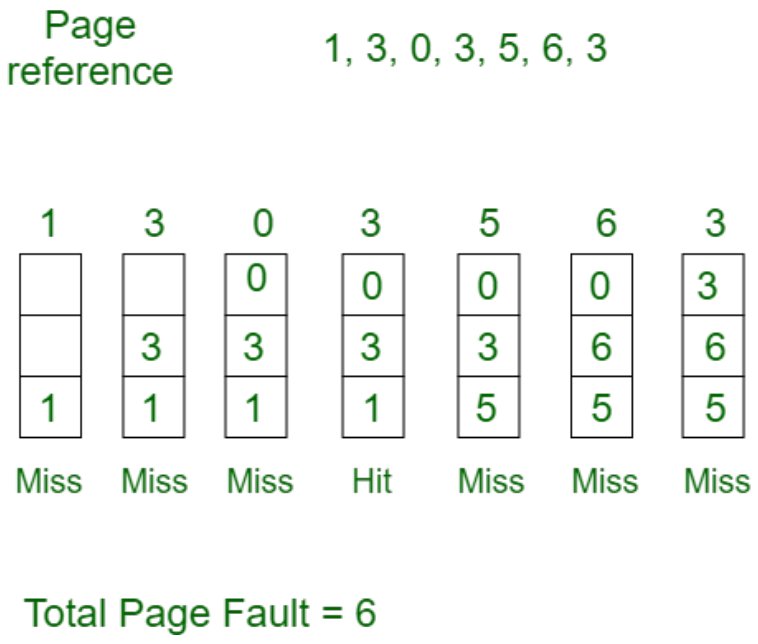


Figure 1 FIFO page replacement algorithm

Queen's School of Computing
CISC324 – Fall 2022
Instructor: Dr. Anwar Hossain
Due Date: November 25th

2. Least Recently Used (LRU): is a Greedy algorithm where the page to be replaced is the least recently used. The idea is based on the locality of reference, the least recently used page is not likely

Let's say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially, we have 4 page slots empty.

Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults

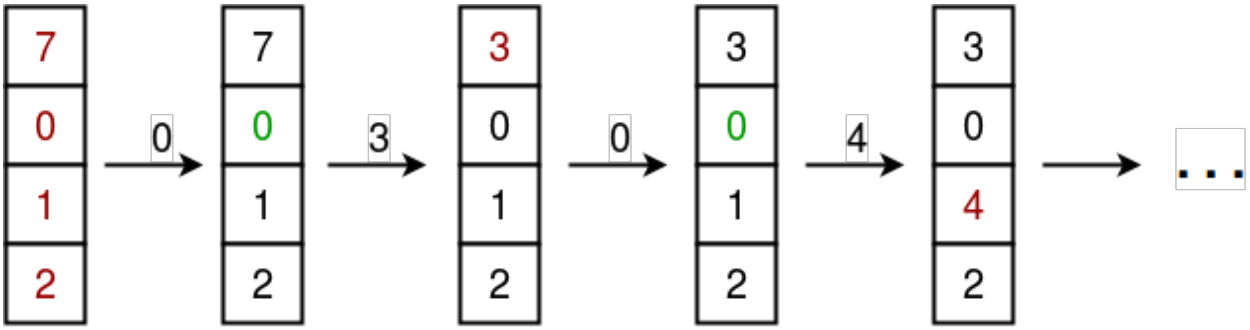
0 is already their so —> 0 Page fault.

when 3 came it will take the place of 7 because it is least recently used —>1 Page fault

0 is already in memory so —> 0 Page fault.

4 will takes place of 1 —> 1 Page Fault

Now for the further page reference string —> 0 Page fault because they are already available in the memory.



Total Page faults = 6

3. Most Frequently Used (MFU): is a Greedy algorithm where the page to be replaced is the most frequently used.

The idea is that the most recently used things will remain in the primary cache, giving very quick access. This reduces the "churn" that you see in an MRU cache when a small number of items are used very frequently. It also prevents those commonly used items from being evicted from the cache just because they haven't been used for a while.

MFU works well if you have a small number of items that are referenced very frequently, and a large number of items that are referenced infrequently. A typical desktop user, for example, might have three or four programs that he uses many times a day and hundreds of programs that he uses very infrequently. If you wanted to improve his experience by caching in memory programs so that they will start quickly, you're better off caching those things that he uses very frequently.

Queen's School of Computing
CISC324 – Fall 2022
Instructor: Dr. Anwar Hossain
Due Date: November 25th

Actually MFU algorithm thinks that the page which was used most frequently will not be needed immediately so it will replace the MFU page.

For part 2, you are required to implement the three previously mentioned algorithms (LRU, FIFO, and MFU). The user will enter the following:

- The number of frames for physical memory (max of 10), let's call it n.
- The number of pages for a particular program (max of 50), let's call it m.
- Then, loop m times and prompt the user to enter the pages' numbers.

Finally, print the page fault count for each algorithm.

You don't have to worry about handling user input, we've already implemented that part in the starter code. All you have to do is write the code for the algorithms in **"FirstInFirstOut.java", "LeastRecentlyUsed.java", and "MostFrequentlyUsed.java"** files (just look for "//TODO" words).

What to submit?

1. Place all your source codes in a folder named in the following format: 324-1234-Lab4 where 1234 stands for the last 4 digits of your student ID, e.g.: If a student ID is 20196072, the folder should be named 324-6072-Lab4.
2. Place a ReadMe.txt file in the same folder above.
3. Compress the above folder (324-1234-Lab4) using Zip (the extension must be.zip)
4. Log into OnQ, locate the lab's dropbox, and upload your zip-folder

What to check during submission?

1. Check that you are not submitting an empty folder.
2. Check that you are not submitting the bytecodes (i.e., compiled).
3. Check that you are not submitting the wrong files.
4. Check that you are not submitting to the wrong dropbox.
5. Check that you are submitting before the deadline.

References

[1] Page replacement algorithms.

[2] Paging in Operating Systems