SYSC4001 - Operating Systems
Assignment 3 – Concurrency, Shared Memory, Virtual Memory, Files
Student 1: Lina Elayed - 101297993
Student 2: Noor ElShanawany - 101302793

PART III:

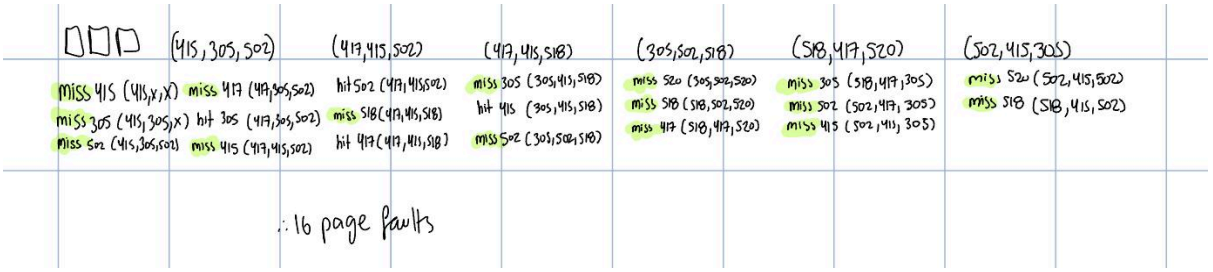
1. Consider the following page reference string in an Operating System with a Demand Paging memory management strategy:

415, 305, 502, 417, 305, 415, 502, 518, 417, 305, 415, 502, 520, 518, 417, 305, 502, 415, 520, 518

n = 20

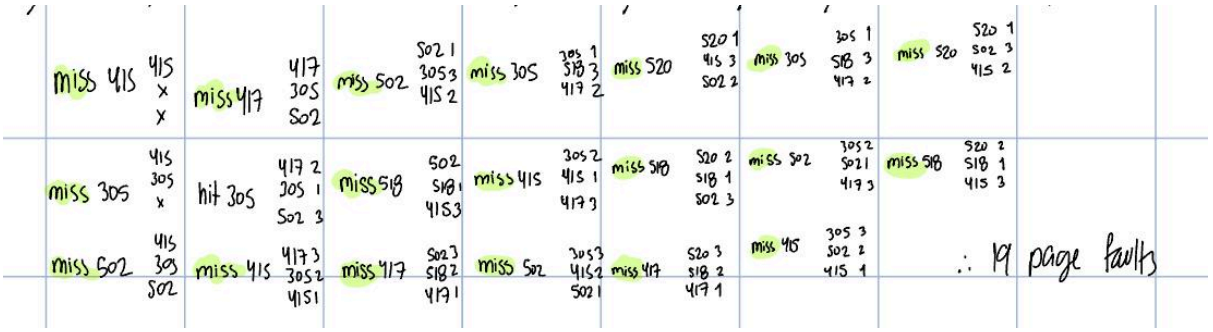
i. How many page faults will occur with 3 frames allocated to the program using the following page replacement algorithms

a. FIFO



Hit ratio: hit/n = 4/20 = 0.20
= probability of having a hit is 20%

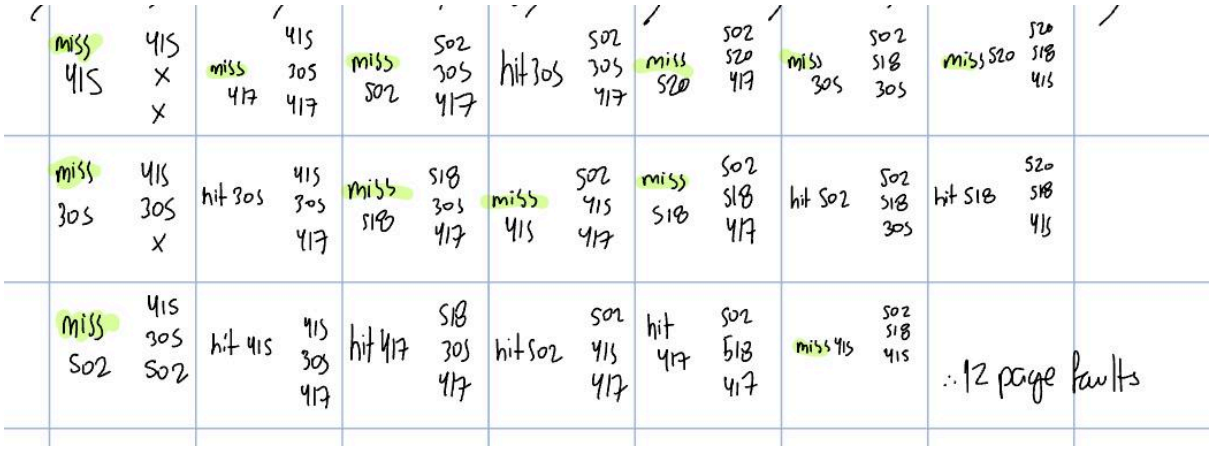
b. LRU



Hit ratio: hit/n = 1/20 = 0.05
= the probability of a hit is 5%

The Least Recently Used (LRU) page replacement algorithm removes the page that has gone unused for the longest amount of time. The idea is that recent behavior is a good predictor of near-future behavior, so a page that hasn't been referenced for a long time is less likely to be needed soon. To make this work, the operating system must track page-access history using a counter or a timer, which makes LRU more accurate than FIFO but also more complex to implement.

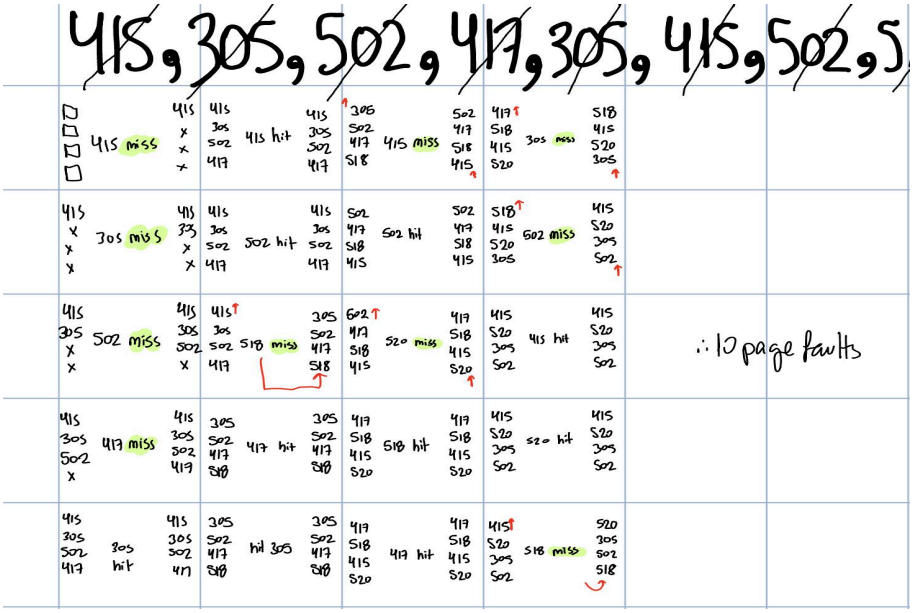
c. OPTIMAL



Hit ratio = 8/20 = 0.4
= so the probability of having a hit is 40%

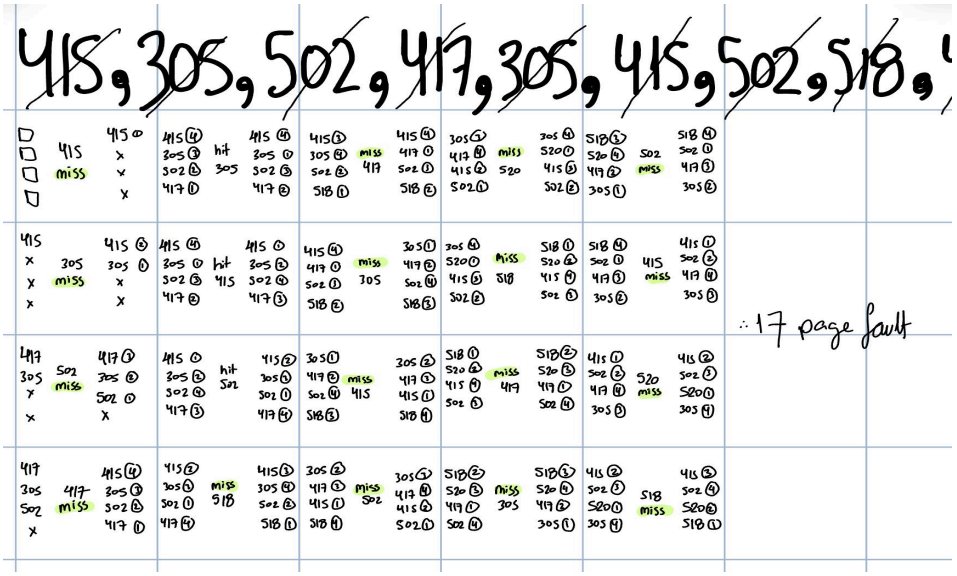
ii. Repeat all three algorithms from Part (i) with 4 frames allocated to the program. Calculate the hit ratio for each of the algorithms.

a. FIFO



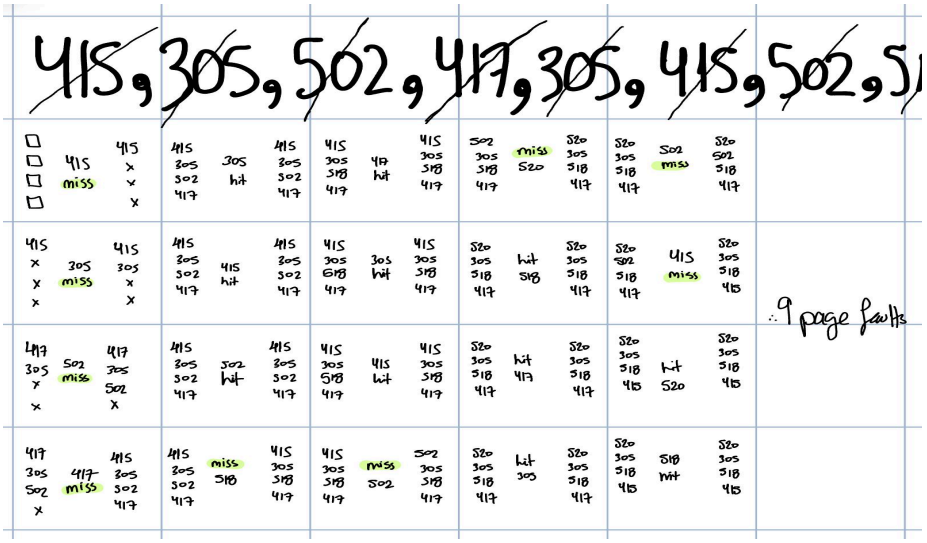
Hit ratio:10/20 =0.50
= the probability of having a hit is 50%

b. LRU



Hit ratio:17/20 =0.15
= the probability of having a hit is 15%

c. OPTIMAL



Hit ratio:11/20 =0.55 = the probability of having a hit is 55%

iii. Which algorithm performs best with 3 frames and why?

The best performance algorithm for the 3 frames (besides the optimal algorithm as it is a hypothetical algorithm with the best hit ratio due to the fact it uses future knowledge of the future memory references and the OS cant know this, making it a theoretical algorithm that we generally use as a “benchmark”) making the algorithm that performs the best is FIFO with a 20% hit rate. The higher ratio represents the higher chance of having the page we need in memory, which is good because we avoid page faults and if we get a page fault we will need to call a page fault ISR which takes time and is overhead. So FIFO is the best because the ordering of this reference string in this sequence happen to have many pages that are used again in the near future so in the queue they are used again before being left out, while the LRU algorithm’s counter sees that these algorithms that they have not been used in a while, forcing that page out, when in the reference string it is next to be used. So FIFO performs better in this case simply because the pattern of requests lines up more favorably with FIFO’s eviction order.

iii. Which algorithm performs best with 4 frames and why?

The best performance algorithm for the 4 frames (besides the optimal algorithm as it is a hypothetical algorithm with the best hit ratio due to the fact it uses future knowledge of the future memory references and the OS cant know this, making it a theoretical algorithm that we generally use as a “benchmark”) making the algorithm that performs the best is FIFO. FIFO’s hit ratio is 50% meaning it's a 50/50 shot to not have a page fault, which is super close to our OPTIMAL algorithm if everything works out. That is because the ordering of this reference string in this sequence happen to have many pages that are used again in the near future so in the queue they are used again before being left out, while the LRU algorithm’s counter sees that these algorithms that they have not been used in a while, forcing that page out, when in the reference string it is next to be used. So FIFO performs better in this case simply because the pattern of requests lines up more favorably with FIFO’s eviction order.

iii. How do the results change when more frames are allocated? What is the relationship?

The relationship between adding more frames and the results of having a page fault vs hit is that increasing the number of frames has two main outcomes:

1- Adding more frames improves performance because more pages can stay in memory at once, leading to fewer page faults and a higher hit ratio, as we can see in our example here where our page faults after adding the 4th frame went from 16 to 10 in FIFO and 19 to 17 in LRU.

2- The other and negative scenario is Belday Anomaly which describes scenarios where our fourth frame leads to more page faults. We add the fourth frame where what we see is the Belady Anomaly where we ended up with more page faults. For example in scenarios like a FIFO algorithm where we only consider age not usefulness, the oldest page "#4" is removed because we haven't used it in the past 3 hits (yet it was the next hit! so we needed that page soon) yet it goes down as the next miss (page fault) even though we have added memory.

iii. Why is the Optimal algorithm impractical in real-world operating systems?

The Optimal algorithm is a theoretical algorithm that requires the knowledge of the future pages used in the reference page, which is something the OS can not possibly know due to the fact it focuses on a 1-process at a time before moving on to the next process, and even if we were to every find the data structures that implement this future acquiring algorithm, it would be very complicated, possibly requiring a lot of registers and memory making it expensive and decreases CPU utilisation with the overhead

Compare the performance of FIFO and LRU. When might FIFO be better or worse than LRU?

Algorithm:	FIFO	LRU
Performance comparison:	FIFO only considers age which leads to the removal of heavily used page just because it's old, which can cause extra faults - aka the Belady Analogy more than any other algorithm	The least recently used algorithm considers when was the last time that a page was used, which gives an indication/probability of pages that the process is no longer using vs pages that it will access soon. More complicated as it involves timers/counters
Cases where each is better	It is better in scenarios like our example, where the reference string has the group of pages that a program is actively using during a certain time period right after one another. Side note: but is also better if we only case about cost/simplicity as it is very simple making it cheap	Generally, LRU is designed to exploit temporal locality (as recently used pages are likely to be used again) and usually performs better than FIFO. It is important to note that for some specific patterns (like this reference string), the actual counts can make FIFO look better than LRU because LRU keeps pages that happen not to be reused soon, while FIFO accidentally retains better pages.

(practice exercise for the final: repeat with LFU) [Student 2: explain the LFU algorithm]
The Lest Frequency Used page replacement algorithm works by keeping track of how often each page in memory has been used. Whenever the system needs to bring in a new page but all memory frames are full, LFU removes the page that has been used the least number of times. The main idea is that pages that are used frequently are going to be needed again, while pages that are barely touched are safer to remove. LFU works well when a program has a steady pattern of accessing the same set of pages because the frequently used ones stay protected from replacement. LFU can struggle though when the program's behaviour suddenly changes. For example, a page that used to be popular may keep a high frequency count even if the program doesn't need it anymore, making the algorithm slow to adapt. Another potential struggle is that LFU needs the system to maintain counters for each page, which adds extra cost. Therefore, LFU tries to keep the most used pages in memory based on past use, but it is not always flexible enough for fast changing workloads.

2. [0.3 marks] Consider a system with memory mapping done on a page basis. Assume that the necessary page table is always in the main memory. A single main memory access takes 120 nanoseconds (ns).

Given:
Main memory access time = 120 ns
Page table is in main memory
TLB overhead = 20 ns per reference

a. [0.1 marks] How long does a paged memory reference take in this system without a TLB? Explain your answer.

Without a TLB, each memory reference needs:
- 1 memory access to read the page table entry, to translate the logical address to physical
- 1 memory access to read the actual data from memory

Each access takes 120 ns:

$$\text{Time} = 2 \times 120 \text{ ns}$$
$$= 240 \text{ ns}$$

A paged memory reference takes 240 ns, because we need two main memory accesses: one to fetch the page table entry and one to fetch the data.

b. [0.1 marks] If we add a Translation Lookaside Buffer (TLB) that imposes an overhead of 20 ns on a hit or a miss. If we assume a TLB hit ratio of 95%, what is the Effective Memory Access Time? Explain your answer.

Now add a TLB that costs 20 ns on every reference.
On a TLB hit:
- 20 ns (TLB lookup)
- 120 ns (single memory access to fetch the data)
  = 140 ns

On a TLB miss:

- 20 ns (TLB lookup)
- 120 ns (memory access to get page table entry)
- 120 ns (memory access to get the data)
  = 260 ns

Hit ratio = 95%
|Miss ratio = 5%.

Effective Memory Access Time (EMAT):

$$\begin{aligned} \text{EMAT} &= 0.95 \times 140\text{ns} + 0.05 \times 260\text{ ns} \\ &= 133\text{ns} + 13\text{ns} \\ &= 146\text{ns} \end{aligned}$$

EMAT ≈ 146 ns.
This result shows that the TLB significantly reduces the average memory access time compared to the 240 ns required without a TLB. Because most accesses are hits, the system usually accesses memory in just one step instead of two.

c. [0.1 marks] Why does adding an extra layer, the TLB, generally improve performance? Are there situations where the performance may be worse with a TLB than without one? Explain all cases.

Why it usually improves performance:
Most programs show locality of reference: they repeatedly access a small set of pages over a short time. The TLB caches translations for these “hot” pages. With a high hit ratio (like 95%), we almost always pay the cheap hit time (140 ns) instead of the full two access time (240 ns). Overall, this lowers EMAT and speeds up the system.

When performance can be worse with a TLB:
If the hit ratio is low, the cost of the extra 20 ns TLB lookup plus extra misses can outweigh the benefit. An extreme example: if the program constantly jumps around and almost every reference is a TLB miss, we pay the overhead of:

$$20\text{ ns (TLB)} + 240\text{ ns (two memory accesses)} = 260\text{ ns}$$

which is slower than 240 ns with no TLB. Also, very large working sets or bad TLB replacement policies can cause frequent TLB misses and reduce the benefit.

Therefore, a TLB helps when access patterns have good locality and the hit ratio is high, but it can hurt when the hit ratio is poor.

3. [0.3 marks] Consider a system with a paged logical address space composed of 128 pages of 4 Kbytes each, mapped into a 512 Kbytes physical memory space. Answer the following questions and justify your answers.

    [0.1 marks] What is the format and size (in bits) of the processor's logical address?

Format = page number | offset

Page number =$\log_2$(# of pages)

$\log_2$(128) = 7

Offset = $\log_2$(page size in bytes)

1KB = 1024 byte

4KB = 4096 bytes

$\log_2$(4096) = 12

Format = 7 bit page | 12 bit offset

Size = 12+7 = 19 bit

[0.1 marks] What is the required length (number of entries) and width (size of each entry in bits, disregarding control bits) of the page table?

Physical memory: 512 KB = 512 × 1024 = 524,288 bytes

Page size: 4096 bytes

Number of frames = 524,288 / 4096 = 128 frames

Bits needed = $\log_2$(128) = 7 bits

Page-table width: 7 bits (ignoring control bits)

[0.1 marks] What is the effect on the page table width if now the physical memory space is reduced by half (from 512 Kbytes to 256 Kbytes)? Assume that the number of page entries and page size remain the same.

New physical memory: 256 KB = 262,144 bytes

Number of frames = 262,144 / 4096 = 64 frames

Bits needed = $\log_2$(64) = 6 bits

The page-table width decreases from 7 bits to 6 bits because decreasing physical memory decreases the number of frames so it reduces frame-number field by 1 bit.

What is the physical memory space:

Physical memory space refers to the actual hardware memory inside the computer (the RAM). Physical memory is limited in size, so the OS must manage it efficiently. It divides memory into fixed-size blocks called frames, and each frame can hold one page from a process. When a process needs to run, only the required pages are loaded into these frames, the rest remain on disk. Because physical memory is faster and more expensive than disk storage, the operating system uses hardware support and memory-management algorithms (like FIFO, LRU, or Optimal) to decide which pages stay in RAM and which pages should be removed when memory becomes full. It is important to make the calculations we have about the size of the physical memory space as it helps us determine how many frames exist and therefore how many bits are needed in each page-table entry to identify a frame.

What is the logical memory space:
The logical memory space is the set of memory addresses that a program believes it has access to when it runs. This is also called the program's virtual address space. Instead of dealing directly with physical RAM, the program works with these logical addresses, and the operating system later translates them into real physical locations behind the scenes. This allows each program to feel like it has its own private and organized memory, even though the actual physical memory may be shared, limited or scattered. The logical memory space makes programs easier to run and safer to manage because every process gets an isolated view of memory, and the ability to use more memory than what physically exists. With the example given in our assignment, it says that the system has 128 pages of 4 KB each, which describes the program's entire logical memory space, regardless of how large the actual physical memory is.

4. [0.1 marks] Explain, in detail, the sequence of operations and file system data structure accesses that occur when a process executes the lseek(fd, offset, SEEK_END) system call. Consider a system using a hierarchical directory structure and assume the file described by the file descriptor (fd) is not currently open by any other process.

When a process executes lseek(fd, offset, SEEK_END), the system first switches from user mode to kernel mode to process the system call. The kernel uses the file descriptor fd to find the file entry in the process open file table. This entry points to a system wide open file table entry, which contains a reference to the inode of the file. The inode was originally located when the file was opened by walking through the hierarchical directory structure, so the directories do not need to be searched again during lseek. The inode stores important information such as the file size and the location of its data blocks. Because SEEK_END is used, the kernel reads the file size from the inode and adds the given offset to compute the new file position. This new position is then saved in the process open file table for that file descriptor. No file data is read or written during this operation, since lseek only changes the file pointer. Since no other process has the file open, no coordination with other processes is needed. Finally, the kernel returns the new file position to the process and switches back to user mode.

5. File Sys Organization:
   [0.1 marks] (from Silberschatz) Consider a file system that uses inodes to represent files. Disk blocks are 8Kb in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

Each disk block is 8 KB in size and each pointer to a disk block is 4 bytes, so one indirect block can store 8192 divided by 4, which equals 2048 pointers. The inode contains 12 direct block pointers, one single indirect pointer, one double indirect pointer, and one triple indirect pointer.

This means the total number of data blocks that can be addressed is 12 for the direct blocks, plus 2048 from the single indirect level, plus 2048 squared from the double indirect level, and plus

2048 cubed from the triple indirect level. The total number of addressable data blocks is therefore $12 + 2048 + 2048^2 + 2048^3$.

Since each block is 8 KB, the maximum file size is 8 KB multiplied by this total. This gives a value slightly larger than 64 terabytes. Therefore, the maximum file size that can be stored in this file system is a little over 64 TB.

> [0.1 marks] Explain what you can do in case (a) if you need to store a file that is larger than the maximum size computed. Give an example showing how you can define a larger file, and what the size of that file would be.

If a file larger than the maximum size computed in part (a) needs to be stored, one simple solution is to divide the large logical file into multiple regular files and store each one using a separate inode. For example, if the maximum file size is about 64 TB, a very large file could be split into two parts called file_part1 and file_part2, where each part could be up to 64 TB in size. The application would treat these two files as one logical file by reading from the first file and then continuing into the second. In this case, the effective file size would be about 128 TB. Another possible solution is to redesign the file system to use larger block sizes or a different structure such as extents, which would allow a single inode to describe a much larger file directly.