**SYSC 4001**

**Assignment 3**

**December 6th, 2023**

Ali Abdollahian - 101229396

Nathan MacDiarmid - 101098993

**Part 1**

1. **i)**

**201,302,203,404,302,201,205,206,302,201,302,203,207,206,203,302,201,302,203,206**

1. **LRU:**

| | 201 | | --- | | 302 | | 203 | | | **404** | 404 | **205** | 205 | 205 | **201** | 201 | **207** | 207 | **302** | 302 | 302 | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 302 | 302 | 302 | **206** | 206 | 206 | **203** | 203 | 203 | 203 | 203 | 203 | | 203 | **201** | 201 | 201 | **302** | 302 | 302 | 302 | **206** | 206 | **201** | **206** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

There will be a total of 15 page faults using the LRU algorithm.

1. **FIFO:**

| | 201 | | --- | | 302 | | 203 | | | **404** | 404 | 404 | **206** | 206 | 206 | **203** | 203 | 203 | **302** | 302 | 302 | **206** | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 302 | **201** | 201 | 201 | **302** | 302 | 302 | **207** | 207 | 207 | **201** | 201 | 201 | | 203 | 203 | **205** | 205 | 205 | **201** | 201 | 201 | **206** | 206 | 206 | **203** | 203 | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

There will be a total of 16 page faults using the FIFO algorithm.

1. **Optimal:**

| | 201 | | --- | | 302 | | 203 | | | 201 | **205** | 205 | **201** | **203** | 203 | 203 | | --- | --- | --- | --- | --- | --- | --- | | 302 | 302 | 302 | 302 | 302 | 302 | 302 | | **404** | 404 | **206** | 206 | 206 | **201** | **206** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

There will be a total of 10 page faults using the Optimal algorithm.

**ii)**

1. **LRU:**

| | **201** | | --- | | **302** | | **203** | | **404** | | **205** | | | 201 | 201 | 201 |  |  | | --- | --- | --- | --- | --- | | 302 | 302 | 302 |  |  | | **206** | 206 | 206 |  |  | | 404 | **203** | 203 |  |  | | 205 | 205 | **207** |  |  | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

There will be a total of 8 page faults using the Last Recently

1. **FIFO**

| | **201** | | --- | | **302** | | **203** | | **404** | | **205** | | | **206** | 206 | 206 | 206 | 206 | | --- | --- | --- | --- | --- | | 302 | **201** | 201 | 201 | 201 | | 203 | 203 | **302** | 302 | 302 | | 404 | 404 | 404 | **203** | 203 | | 205 | 205 | 205 | 205 | **207** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

There will be a total of 10 page faults using the FIFO algorithm.

1. **Optimal**

| | **201** | | --- | | **302** | | **203** | | **404** | | **205** | | | 201 | 201 |  |  |  | | --- | --- | --- | --- | --- | | 302 | 302 |  |  |  | | 203 | 203 |  |  |  | | **206** | 206 |  |  |  | | 205 | **207** |  |  |  | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

There will be a total of 7 page faults using the Optimal algorithm.

2. We first access memory for the page table and frame number and then we access the byte in memory so in total 250 + 250 = 500
3. So in this case an 80 percent hit in the TLB means that we find the required page number in the TLB 80 percent of the time. Also we have 30 ns to search the TLB as a result there would be two situations; First, if we are able to find and the TLB contains the entry required, we have 30 ns overhead on top of the 250 ns memory access time. Second situation when we can not find page number in the TLB, we have an additional 250 ns to get it into the TLB:

(280 \* 0.80) + (530 \* 0.20) = 224 + 106 = 330

1. **Why adding another layer, TLB, can improve performance?**

Due to several reasons:

1- TLB provides a fast look up for translating virtual addresses to physical addresses which reduces memory access time.

2- In the case when TLB hits, it leads us to faster accessing the memory since we do not need to access the page table in memory for every address which is slow.

3- TLB is located between CPU and main memory and because the data is already in the cache it will boost the access to cached data.

Basically TLB speeds up by storing page-tables in faster memory.

**Are there situations where the performance may be worse with TLB than without TLB?**

Yes there are several situations such as:

1- If the CPU tries to access pages that are not in TLB, then CPU has to go to memory which is slow on top of the time it took CPU to search TLB.

2- TLB size is way more limited than memory and if a program requires many new unique virtual pages that are not in TLB ends up with a high rate of misses.

3- During a context switch among different processes, the TLB needs to be flushed which is additional overhead.

1. We need two part firstly the page number to identify the page within the logical addresses so in our case here we need 5 bits to represent them which can be calculated as 2 to the power of n = 32; So n = 5 (2^5 = 32), Secondly we need offset of 11 bits that can be calculated as: 2^11 = 2K
2. The length and width is also can be calculated:

Length: is 32 pages to map

Width: is 9 bits because 2^20 = 1M -> 20 - 11 = 9 or in other word we can also say the page table size is the size of physical memory and we have a memory of 1 MByte and each page is 2k Bytes so there are 512 pages, 2^9=512

1. If the physical memory size is halved (0.5MBytes / 2K bytes = 256) the new width will become 2^8 = 256 which shows us that width is reduced by 1.
2. A race condition is when two pointers or pieces of a program are trying to use or edit the same piece of memory at the same time, usually a variable. For example, if you were to run a program with one variable called ‘count’ and the program has two different functions that both attempt to change the value at the same time, the system won’t know which one to choose, resulting in a race condition.
3. Disabling interrupts is not a good way to prevent race conditions as interrupts are what are enforcing race conditions not to happen. In a multiprocessor environment, interrupts are used to pass information to all processors in that environment, determining which processor can run next. If you disable interrupts, nothing will notify each processor when to run, resulting in one massive race condition in a multiprocessor environment.
4. A semaphore is used to ensure that a variable cannot be accessed by two or more instances, most likely functions, at the same time. Semaphores are implemented by using a producer/consumer model where one function produces a within a critical section and the other consumes within a critical section. The critical sections of each can only be run if the other is not running. This can be done by having some sort of boolean lock that is true if one function is running and false if the other is running. Semaphores are what are used to prevent race conditions.
5. The open() function used to search for the first entry point that matched the name of the file, but now instead calls an open-file table with an index, getting the file without any searching. Now, when the open() function is called, a user can pass the file name as well as mode information such as read-only, write, etc. Then the open() system call typically returns a pointer to the entry in the open-file table. The pointer is then used in all I/O operations, avoiding any further searching.

When several processes look to open the file simultaneously, a per-process open-file table and a system-wide open-file table are referenced. A per-process table tracks all files that a process has open whereas a system-wide table tracks all the files that the system has open. Each entry in the per-process table points to a system-wide table, there's only one system-wide table. The system-wide table contains process-independent information such as the location of the file on the disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an open() call, a new entry is simply added to the process’s open-file table pointing to the appropriate entry in the system-widge table. The open-file table usually has an open count associated with each file to indicate how many processes have the file opened. A close() call decreases this open count.

If a file is opened for APPEND in a hierarchical directory, The open() system call also accepts access mode information such as append; this mode is checked against the file’s permissions. If allowed then it's ready for the append process; APPEND mode means that data is written at the end of the file, and the file pointer is positioned at the end. Therefore, the file system needs to position the file pointer at the end of the file, ready for appending new data.

In order to use the append there is a write operation write\_next() which appends to the end of the file and advances to the end of the newly written material (the new end of the file).

2. Direct Blocks size : 12 blocks × 8 KB/block= 96 KB

Single Indirect Block size: 2048 pointers × 8 KB/pointer=16,384 KB= 16MB

Double Indirect Block size: 2048×2048×8 KB =33,554,432KB=32GB

Triple Indirect Block: 2048 × 2048 × 2048 × 8KB=68,719,476,736=which is approximately equal to 64TB

(12 \* 8 KB) + (2048 \* 8 KB) + (2048 \* 2048 \* 8 KB) + (2048 \* 2048 \* 2048 \* 8 KB) =67TB or approximately can be said 64TB

1. In our case if we have a file larger than the maximum size we could use the segmentation such that we break down the file into smaller parts then store them as a separate file so whenever we need to retrieve the file we would combine the segments again.

For instance if we have a file which is 75TB we could break this down into File A: Segment 1: 45TB and File B: Segment 2: 30TB.