**Part 3- Concepts**

**1.)**

Pages:

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

(i) How many page faults will occur for the following page replacement algorithms when 3 frames are allocated to the program?

(a) LFU

* Start: frames = [ ], freq = {}, faults = 0

1. Ref 201: miss, frames=[201], freq(201)=1, faults=1
2. Ref 302: miss, frames=[201,302], freq(302)=1, faults=2
3. Ref 203: miss, frames=[201,302,203], freq(203)=1, faults=3 (frames full now)
4. Ref 404: miss. All have freq=1 (201,302,203). Tie-break oldest is 201. Evict 201, insert 404. freq(404)=1, faults=4  
   frames=[404,302,203]
5. Ref 302: hit, freq(302)=2, faults=4
6. Ref 201: miss. Current freq:404(1),302(2),203(1). LFU are 404 & 203 (freq=1). Oldest between them is 203. Evict 203, insert 201, freq(201)=2, faults=5  
   frames=[404,302,201]
7. Ref 205: miss. freq now:404(1),302(2),201(2). LFU is 404(1). Evict 404, insert 205, freq(205)=1, faults=6  
   frames=[205,302,201]
8. Ref 206: miss. freq:205(1),302(2),201(2). LFU=205(1). Evict 205, insert 206, freq(206)=1, faults=7  
   frames=[206,302,201]
9. Ref 302: hit, freq(302)=3, faults=7
10. Ref 201: hit, freq(201)=3, faults=7
11. Ref 302: hit, freq(302)=4, faults=7
12. Ref 203: miss. freq before insert:201(3),302(4),206(1). LFU=206(1). Evict 206, insert 203, freq(203)=2 (it was used before), faults=8  
    frames=[203,302,201]
13. Ref 207: miss. freq:203(2),302(4),201(3). LFU=203(2). Evict 203, insert 207, freq(207)=1, faults=9  
    frames=[207,302,201]
14. Ref 206: miss. freq:207(1),302(4),201(3). LFU=207(1). Evict 207, insert 206, freq(206)=2, faults=10  
    frames=[206,302,201]
15. Ref 203: miss. freq:206(2),302(4),201(3). LFU=206(2). Evict 206, insert 203, freq(203)=3, faults=11  
    frames=[203,302,201]
16. Ref 302: hit, freq(302)=5, faults=11
17. Ref 201: hit, freq(201)=4, faults=11
18. Ref 302: hit, freq(302)=6, faults=11
19. Ref 203: hit, freq(203)=4, faults=11
20. Ref 206: miss. freq now:201(4),302(6),203(4). The LFU pages have freq=4 for both 201 and 203. Tie-break by oldest in memory. 201 has been in memory longer than 203. Evict 201, insert 206, freq(206)=3, faults=12.

**LFU 3-frames faults = 12**

(b) FIFO

Just keep a queue of pages in order of insertion.

* frames = [], faults=0

1. 201: miss, [201], faults=1
2. 302: miss, [201,302], faults=2
3. 203: miss, [201,302,203], faults=3 (full now)
4. 404: miss, evict 201, insert 404, [302,203,404], faults=4
5. 302: hit, [302,203,404], faults=4
6. 201: miss, evict 302, insert 201, [203,404,201], faults=5
7. 205: miss, evict 203, insert 205, [404,201,205], faults=6
8. 206: miss, evict 404, insert 206, [201,205,206], faults=7
9. 302: miss, evict 201, insert 302, [205,206,302], faults=8
10. 201: miss, evict 205, insert 201, [206,302,201], faults=9
11. 302: hit, [206,302,201], faults=9
12. 203: miss, evict 206, insert 203, [302,201,203], faults=10
13. 207: miss, evict 302, insert 207, [201,203,207], faults=11
14. 206: miss, evict 201, insert 206, [203,207,206], faults=12
15. 203: hit, [203,207,206], faults=12
16. 302: miss, evict 203, insert 302, [207,206,302], faults=13
17. 201: miss, evict 207, insert 201, [206,302,201], faults=14
18. 302: hit, [206,302,201], faults=14
19. 203: miss, evict 206, insert 203, [302,201,203], faults=15
20. 206: miss, evict 302, insert 206, [201,203,206], faults=16

**FIFO 3-frames faults = 16**

(c) Optimal page replacement strategy

Optimal evicts the page with the farthest future use.

* frames=[], faults=0

1. 201: miss, [201], f=1
2. 302: miss, [201,302], f=2
3. 203: miss, [201,302,203], f=3 (full)
4. 404: miss. Future use:
   1. 201 future use: step6
   2. 302 future use: step5
   3. 203 future use: step12,
   4. Farthest is 203. Evict 203, insert 404, f=4 frames=[201,302,404]
5. 302: hit
6. 201: hit
7. 205: miss. Future use now:
   1. 201 future: step10
   2. 302 future: step9
   3. 404 future: no future use Evict 404 (no future), insert 205, f=5  
      frames=[201,302,205]
8. 206: miss. Future use:
   1. 201: step10
   2. 302: step9
   3. 205: no future Evict 205, insert 206, f=6 frames=[201,302,206]
9. 302: hit
10. 201: hit
11. 302: hit
12. 203: miss. Future:
    1. 201: step17
    2. 302: step16
    3. 206: step14 Farthest is 201. Evict 201, insert 203, f=7  
       frames=[203,302,206]
13. 207: miss. Future:
    1. 203: step15
    2. 302: step16
    3. 206: step14, Farthest is 302 (used at step16 is actually the farthest in the future). Evict 302, insert 207, f=8 frames=[203,207,206]
14. 206: hit
15. 203: hit
16. 302: miss. Future:
    1. 203: step19
    2. 207: no future
    3. 206: step20 Evict 207 (no future), insert 302, f=9  
       frames=[203,302,206]
17. 201: miss. Future:
    1. 203: step19
    2. 302: step18
    3. 206: step20, Farthest is 206 (step20 is the farthest). Evict 206, insert 201, f=10, frames=[203,302,201]
18. 302: hit
19. 203: hit
20. 206: miss. Future: 203,302,201: no future uses left All same. Evict any, say 203. Insert 206, f=11  
    frames=[206,302,201]

**Optimal 3-frames faults = 11**

ii) Repeat (i) for 5 frames allocated to the program.

(a) LFU

* frames=[], faults=0

1. 201: miss, freq(201)=1, [201],f=1
2. 302: miss, freq(302)=1, [201,302],f=2
3. 203: miss, freq(203)=1, [201,302,203],f=3
4. 404: miss, freq(404)=1, [201,302,203,404],f=4
5. 302: hit, freq(302)=2,f=4
6. 201: hit, freq(201)=2,f=4
7. 205: miss, freq(205)=1, [201,302,203,404,205],f=5 (now full with 5 frames)
8. 206: miss. Need to evict LFU: freq so far:201(2),302(2),203(1),404(1),205(1)  
   LFU are 203,404,205 all freq=1. Oldest among them is 203. Evict 203, freq(206)=1, f=6  
   frames=[201(2),302(2),404(1),205(1),206(1)]
9. 302: hit, freq(302)=3,f=6
10. 201: hit, freq(201)=3,f=6
11. 302: hit, freq(302)=4,f=6
12. 203: miss. frames=[201(3),302(4),404(1),205(1),206(1)]  
    LFU=404(1),205(1),206(1). Oldest is 404. Evict 404, insert 203, freq(203)=2, f=7frames=[201,302,203,205,206]
13. 207: miss. freq:201(3),302(4),203(2),205(1),206(1)  
    LFU=205(1),206(1). Oldest is 205. Evict 205, insert 207, freq(207)=1, f=8  
    frames=[201,302,203,207,206]
14. 206: hit, freq(206)=2,f=8
15. 203: hit, freq(203)=3,f=8
16. 302: hit, freq(302)=5,f=8
17. 201: hit, freq(201)=4,f=8
18. 302: hit, freq(302)=6,f=8
19. 203: hit, freq(203)=4,f=8
20. 206: hit, freq(206)=3,f=8

No more evictions after step13.

**LFU 5-frames faults = 8**

(b) FIFO

* frames=[], queue empty, faults=0

1. 201: miss, queue=[201],f=1
2. 302: miss, queue=[201,302],f=2
3. 203: miss, queue=[201,302,203],f=3
4. 404: miss, queue=[201,302,203,404],f=4
5. 302: hit
6. 201: hit (We still have one more free frame since we have 5 frames total)
7. 205: miss, queue=[201,302,203,404,205],f=5 (now 5 full frames)
8. 206: miss, evict front=201, queue=[302,203,404,205,206],f=6
9. 302: hit
10. 201: miss, evict front=302, queue=[203,404,205,206,201],f=7
11. 302: miss, evict front=203, queue=[404,205,206,201,302],f=8
12. 203: miss, evict front=404, queue=[205,206,201,302,203],f=9
13. 207: miss, evict front=205, queue=[206,201,302,203,207],f=10
14. 206: hit
15. 203: hit
16. 302: hit
17. 201: hit
18. 302: hit
19. 203: hit
20. 206: hit

No more faults after step13.

**FIFO 5-frames faults = 10**

(c) Optimal page replacement strategy

* frames=[], faults=0

1. 201: miss, [201],f=1
2. 302: miss, [201,302],f=2
3. 203: miss, [201,302,203],f=3
4. 404: miss, [201,302,203,404],f=4
5. 302: hit
6. 201: hit
7. 205: miss, [201,302,203,404,205],f=5 (5 frames full now)
8. 206: miss. Check future use: 404 no future use, 205 no future use. Evict one with no future. Evict 404. Insert 206,f=6 [201,302,203,205,206]
9. 302: hit
10. 201: hit
11. 302: hit
12. 203: hit
13. 207: miss. Future after step13: 205 no future use. Evict 205. Insert 207,f=7 [201,302,203,206,207]
14. 206: hit
15. 203: hit
16. 302: hit
17. 201: hit
18. 302: hit
19. 203: hit
20. 206: hit

No further faults after step13.

**Optimal 5-frames faults = 7**

**2.)**

a.

Without a TLB, for each memory reference in a system using a single-level page table located in memory, the following happens:

1. Access the page table to find the physical frame number. This is one memory reference = 250 ns.
2. Access the desired memory location using the obtained physical address. This is another memory reference = 250 ns.

So total time for a single memory reference = 250 ns (page table lookup) + 250 ns (actual memory access) = **500 ns**.

**Explanation**: To translate a virtual address to a physical address, we must first consult the page table, which resides in memory. Since each memory access takes 250 ns, and we need one memory access for the page table entry and another to actually get the data, the total doubles.

b.

Now we introduce a TLB (Translation Lookaside Buffer) that caches recent translations. The TLB imposes a 30 ns overhead on every memory reference, regardless of hit or miss.

* **On a TLB hit (80% of the time)**:  
  Steps:
  1. Check TLB: 30 ns overhead.
  2. Since it’s a hit, we have the physical address immediately, so just access memory: 250 ns.

Total time on a TLB hit = 30 ns + 250 ns = 280 ns.

* **On a TLB miss (20% of the time)**:  
  Steps:
  1. Check TLB: 30 ns overhead (still paid even on miss).
  2. Not in TLB, so access the page table in memory: 250 ns.
  3. Now access the desired memory location: another 250 ns.

Total time on a TLB miss = 30 ns + 250 ns + 250 ns = 530 ns.

**Effective Memory Access Time (EMAT)**:

* Probability of hit = 0.8
* Probability of miss = 0.2

EMAT = (0.8 \* 280 ns) + (0.2 \* 530 ns)  
= (0.8 \* 280) + (0.2 \* 530)  
= 224 ns + 106 ns  
= **330 ns**.

**Explanation**: By having a high hit ratio (80%), most accesses benefit from the shorter 280 ns time. Only a minority (20%) incur the longer 530 ns penalty, resulting in an average (effective) time of 330 ns, which is an improvement over the 500 ns without a TLB.

c.

**Why does adding a TLB improve performance?**  
A TLB acts as a high-speed cache for address translations. Most programs exhibit locality of reference, meaning they frequently access the same set of pages over short periods. By caching these translations, the system avoids repeated accesses to the page table in memory. When the TLB hit rate is high, the average access time significantly decreases.

**When could performance be worse with a TLB?**  
If the hit ratio is very low (i.e., most references miss in the TLB), then every access pays the additional TLB overhead (30 ns in this case) and still ends up doing a full page table lookup plus a memory access. This adds extra overhead compared to not having a TLB at all. In other words, the benefit of the TLB only manifests if its hit ratio is sufficiently high to offset the overhead it introduces. If the workload results in poor locality and constant TLB misses, the TLB overhead makes performance worse than not having a TLB.

**3.)**

a.

**Find the format of the processor’s logical address:**

* **Offset calculation:** Each page is 2 KB = 2048 bytes.  
  2048 = 2^11, so we need 11 bits for the offset within a page.
* **Page number calculation:** There are 32 pages in the logical address space.  
  32 = 2^5, so we need 5 bits for the page number.

**Logical address format:**

* Page number: 5 bits
* Offset: 11 bits

Total logical address length = 5 + 11 = 16 bits

**Answer:** The logical address has a 5-bit page number and an 11-bit offset, for a total of 16 bits.

b.

**Determine the length and width of the page table (before the physical memory reduction):**

* **Page Table Length (Number of Entries):** There is one page table entry per page in the logical space. Number of pages = 32, so there are 32 entries in the page table.
* **Width of Each Entry (Number of Bits Needed for the Frame Number):** With 1 MB of physical memory and a 2 KB frame size, we have 512 frames (since 1 MB divided by 2KB is 512). 512 = 2^9, so we need 9 bits to uniquely identify each frame.

**Answer:** The page table has 32 entries in length, and each entry is 9 bits wide (disregarding control bits)

c.

**Effect on the page table if the physical memory is reduced by half:**

* **New Physical Memory Size:** Half of 1 MB = 0.5 MB = 512 KB = 524,288 bytes.
* **Number of Frames After Halving:** Each frame is still 2 KB. Number of frames = 524,288 bytes / 2,048 bytes per frame = 256 frames. 256 = 2^8, so now we need only 8 bits to represent the frame number.
* **Impact on the Page Table:** The number of pages (and thus the number of page table entries) does not change because the logical address space and page size are unchanged. The only change is in the width of each entry. Instead of 9 bits, now 8 bits are sufficient to address all frames.

**Answer:** The number of page table entries remains at 32 entries in length, but each entry now only needs 8 bits width to store the frame number (disregarding control bits).

**4.)**

When a process issues a write operation to a file, the operating system (OS) must handle several steps to ensure the correct data is recorded. In the case of a file opened for **APPEND** mode within a hierarchical directory structure, the process is as follows:

1. **Validate File and Permissions**:  
   The OS first checks that:
   * The file is currently open by the requesting process.
   * The file is open in a mode that allows writing. If the file is not open or not writable, the operation fails.
2. **Determine the File Position**:  
   When a file is opened for APPEND, every write operation should place data at the end of the existing file content. This means the OS must:
   * Retrieve the file’s current size
   * Set the file’s write pointer (offset) to the end of the file, regardless of any previously set offset.
3. **Locating the Data Blocks**:  
   Files in a hierarchical directory structure are stored in data blocks on disk, and their location is recorded in metadata structures. The OS must:
   * Use the file’s metadata to determine which disk blocks correspond to the current end of the file.
   * If the write operation extends beyond the current end, the OS may need to allocate additional free blocks from the filesystem’s free space pool.
   * Update the file’s metadata structures to include these newly allocated blocks if necessary.
4. **Performing the Write to Cache/Buffers**:  
   Typically, writes go through a system buffer cache for efficiency. The OS will:
   * Copy the user-supplied data from the process’s buffer into kernel-managed buffers.
   * Mark these buffers as they contain modified data that should be written to disk.
5. **Updating File Size and Metadata**:  
   Once the data is prepared for writing:
   * The file’s size is updated in its metadata to reflect the appended data.
   * The file’s modification time (mtime) and possibly its change time (ctime) are updated.
   * If the directory structure or indexes (like a directory entry or index structure) need to be updated due to block allocations, these updates occur as well.
6. **Deferred or Immediate Disk Write**:  
   Depending on the OS’s write policy, the data may be:
   * Written to the disk immediately (synchronous write).
   * Scheduled for write-back later (asynchronous write), allowing the OS to optimize disk I/O.
7. **Returning Control to the Process**:  
   After the OS has completed the necessary updates (at least to the cache and metadata), it returns control to the user process. The return value of the write operation typically indicates how many bytes were successfully written.

In APPEND mode, the critical difference is that the write pointer is always set to the end of the file prior to writing. This ensures that any new data is added after the existing content, preventing overwriting existing data and maintaining the file as a continuously growing structure. If multiple processes are writing concurrently in APPEND mode, the OS may need to lock the file to ensure that the end-of-file pointer is managed consistently and safely.

**5.)**

a.

* Block size = 8 KB = 8192 bytes
* Pointer size = 4 bytes
* Each indirect block holds 8192/4 = 2048 pointers.
* There are 12 direct pointers, and one each of single, double, and triple indirect.

Calculations:

1. Direct blocks: There are 12 direct pointers, each pointing directly to a data block.  
   Each data block = 8 KB; 12 \* 8 KB = 96 KB
2. Single indirect: The single indirect block contains 2048 pointers, each pointing to a data block; 2048 blocks \* 8 KB = 16 MB
3. Double indirect: The double indirect block points to N=2048N = 2048N=2048 single indirect blocks. Each single indirect block references 2048 data blocks.  
   Total data blocks via double indirect = N×N = 2048^2 = 4,194,304 blocks.  
   Each block = 8 KB; (2048 \* 2048 blocks) \* 8 KB = 32 GB
4. Triple indirect: The triple indirect block points to 2048 double indirect blocks.  
   Each double indirect block covers 2048^2 data blocks. Total data blocks via triple indirect = 2048^3 = 8,589,934,592 blocks.  
   Each block = 8 KB.  
   Total triple indirect space = 8,589,934,592×8 KB, Let’s express this in powers of two: 2048 = 2^11, 2048^3 = 2^33, A block is 8 KB = 2^13 bytes. Thus, triple indirect space = 2^33×2^13= 2^46 bytes = 64 TiB, with 1 TiB being 2^40 bytes

**Add them all up.**

* Direct: ~96 KB
* Single indirect: ~16 MB
* Double indirect: ~32 GB
* Triple indirect: ~64 TiB

The triple indirect contribution dominates the total, so the maximum file size is approximately **64 TiB** plus some smaller additions from the other parts. For completeness:

b.

To store a file larger than 64 TiB, you could add another level of indirection. For example, a quadruple-indirect pointer and with :

* Quadruple indirect: 2048^4 blocks \* 8 KB = ≈ 128 PiB

This would allow a much larger file size.

Therefore, to store larger files than the original maximum, we can modify the file system design, for example by adding another layer of indirect pointers. For instance, adding a quadruple-indirect pointer would allow for a file size of around 128 PiB.