Assignment 3 - SYSC4001

Amit Kunkulol - 101260631
Kushal Poudel - 101298706

SYSC4001A - Lab Section L3

Professor Gabriel Wainer

2025-12-01

Part 3

1. Reference String: 415, 305, 502, 417, 305, 415, 502, 518, 417, 305, 415, 502, 520, 518, 417, 305, 502, 415, 520, 518

i)

a) FIFO, 3 Frames

Page	Frames After Access	Hit Or Fault
415	[415, -, -]	Fault
305	[415, 305, -]	Fault
502	[415, 305, 502]	Fault
417	[417, 305, 502] (remove 415)	Fault
305	[417, 305, 502]	Hit
415	[417, 415, 502] (remove 305)	Fault
502	[417, 415, 502]	Hit
518	[417, 415, 518] (remove 502)	Fault
417	[417, 415, 518]	Hit
305	[305, 415, 518] (remove 417)	Fault
415	[305, 415, 518]	Hit
502	[305, 502, 518] (remove 415)	Fault
520	[305, 502, 520] (remove 518)	Fault
518	[518, 502, 520] (remove 305)	Fault
417	[518, 417, 520] (remove 502)	Fault
305	[518, 417, 305] (remove 520)	Fault
502	[502, 417, 305] (remove 518)	Fault
415	[502, 415, 305] (remove 417)	Fault
520	[502, 415, 520] (remove 305)	Fault
518	[518, 415, 520] (remove 502)	Fault

Hits: 4

Page Faults: 16

Hit Ratio: 4/20 = 0.2 Hits/Page

b) LRU, 3 Frames

Page	Frames After Access	Hit Or Fault
415	[415, -, -]	Fault
305	[415, 305, -]	Fault
502	[415, 305, 502]	Fault
417	[417, 305, 502] (remove 415)	Fault
305	[417, 305, 502]	Hit
415	[417, 305, 415] (remove 502)	Fault
502	[502, 305, 415] (remove 417)	Fault
518	[502, 518, 415] (remove 305)	Fault
417	[502, 518, 417] (remove 415)	Fault
305	[305, 518, 417] (remove 502)	Fault
415	[305, 415, 417] (remove 518)	Fault
502	[305, 415, 502] (remove 417)	Fault
520	[520, 415, 502] (remove 305)	Fault
518	[520, 518, 502] (remove 415)	Fault
417	[520, 518, 417] (remove 502)	Fault
305	[305, 518, 417] (remove 305)	Fault
502	[305, 502, 417] (remove 518)	Fault
415	[305, 502, 415] (remove 415)	Fault
520	[520, 502, 415] (remove 305)	Fault
518	[520, 518, 415] (remove 502)	Fault

Hits: 1, Page Faults: 19
Hit Ratio: 1/20 = 0.05 Hits/Page

[Individual Submission - Student 1] What is the LRU algorithm?

- The least recently used algorithm is used to try and minimize the number of page faults, which occur when there are no free frames available. Essentially, the goal of all page replacement algorithms are to choose the best page to evict so that future page faults are minimized. For LRU, it assumes that a page that has not been used for the longest time is the least likely to be used soon, and replaces it. By replacing the least recently accessed page, we can perform close to the optimal algorithm.
- There are two possible implementations for this algorithm. The first way is using a counter and a clock. Every page table entry stores a timestamp of its last used time and on every memory access, the OS updates this timestamp. We replace the page with the smallest time value. Secondly, we can implement LRU using a stack. This involves the OS keeping a stack of pages in order of most recent use. Then, when a page is referenced, it is moved to the top of the stack. This way the bottom of the stack has the least recently used page, which we replace.
- Since this algorithm is expensive to implement, we use approximations like reference bit or clock algorithms.

c) Optimal, 3 Frames

Page	Frames After Access	Hit Or Fault
415	[415, -, -]	Fault
305	[415, 305, -]	Fault
502	[415, 305, 502]	Fault
417	[415, 305, 417] (remove 502)	Fault
305	[415, 305, 417]	Hit
415	[415, 305, 417]	Hit
502	[502, 305, 417] (remove 417)	Fault
518	[518, 305, 417] (remove 502)	Fault
417	[518, 305, 417]	Hit
305	[518, 305, 417]	Hit
415	[518, 415, 417] (remove 305)	Fault
502	[518, 502, 417] (remove 415)	Fault
520	[518, 520, 417] (remove 502)	Fault
518	[518, 520, 417]	Hit
417	[518, 520, 417]	Hit
305	[518, 520, 305] (remove 417)	Fault
502	[518, 520, 502] (remove 305)	Fault
415	[518, 520, 415] (remove 305)	Fault
520	[518, 520, 415]	Hit
518	[518, 520, 415]	Hit

Hits: 8
Page Faults: 12
Hit Ratio: 8/20 = 0.4 Hits/Page

ii)

a) FIFO, 4 Frames

Page	Frames After Access	Hit Or Fault
415	[415, -, -, -]	Fault
305	[415, 305, -, -]	Fault
502	[415, 305, 502, -]	Fault
417	[415, 305, 502, 417]	Fault
305	[415, 305, 502, 417]	Hit
415	[415, 305, 502, 417]	Hit
502	[415, 305, 502, 417]	Hit
518	[518, 305, 502, 417] (remove 518)	Fault
417	[518, 305, 502, 417]	Hit
305	[518, 305, 502, 417]	Hit
415	[518, 415, 502, 417] (remove 305)	Fault
502	[518, 415, 502, 417]	Hit
520	[518, 415, 520, 417] (remove 502)	Fault
518	[518, 415, 520, 417]	Hit
417	[518, 415, 520, 417]	Hit
305	[518, 415, 520, 305] (remove 417)	Fault
502	[502, 415, 520, 305] (remove 518)	Fault
415	[502, 415, 520, 305]	Hit
520	[502, 415, 520, 305]	Hit
518	[502, 518, 520, 305] (remove 415)	Fault

Hits: 10

Page Faults: 10

Hit Ratio: 10/20 = 0.5 Hits/Page

b) LRU, 4 Frames

Page	Frames After Access	Hit Or Fault
415	[415, -, -, -]	Fault
305	[415, 305, -, -]	Fault
502	[415, 305, 502, -]	Fault
417	[415, 305, 502, 417]	Fault
305	[415, 305, 502, 417]	Hit
415	[415, 305, 502, 417]	Hit
502	[415, 305, 502, 417]	Hit
518	[415, 305, 502, 518] (remove 417)	Fault
417	[415, 417, 502, 518] (remove 305)	Fault
305	[305, 417, 502, 518] (remove 415)	Fault
415	[305, 417, 415, 518] (remove 502)	Fault
502	[305, 417, 415, 502] (remove 518)	Fault
520	[305, 520, 415, 502] (remove 417)	Fault
518	[518, 520, 415, 502] (remove 305)	Fault
417	[518, 520, 417, 502] (remove 415)	Fault
305	[518, 520, 417, 305] (remove 502)	Fault
502	[518, 502, 417, 305] (remove 520)	Fault
415	[415, 502, 417, 305] (remove 518)	Fault
520	[415, 502, 520, 305] (remove 417)	Fault
518	[415, 502, 520, 518] (remove 305)	Fault

Hits: 3
Page Faults: 17
Hit Ratio: 3/20 = 0.15 Hits/Page

c) Optimal, 4 Frames

Page	Frames After Access	Hit Or Fault
415	[415, -, -, -]	Fault
305	[415, 305, -, -]	Fault
502	[415, 305, 502, -]	Fault
417	[415, 305, 502, 417]	Fault
305	[415, 305, 502, 417]	Hit
415	[415, 305, 502, 417]	Hit
502	[415, 305, 502, 417]	Hit
518	[415, 305, 518, 417] (remove 502)	Fault
417	[415, 305, 518, 417]	Hit
305	[415, 305, 518, 417]	Hit
415	[415, 305, 518, 417]	Hit
502	[502, 305, 518, 417] (remove 415)	Fault
520	[520, 305, 518, 417] (remove 502)	Fault
518	[520, 305, 518, 417]	Hit
417	[520, 305, 518, 417]	Hit
305	[520, 305, 518, 417]	Hit
502	[520, 502, 518, 417] (remove 305)	Fault
415	[520, 415, 518, 417] (remove 502)	Fault
520	[520, 415, 518, 417]	Hit
518	[520, 415, 518, 417]	Hit

Hits: 11
Page Faults: 9
Hit Ratio: 11/20 = 0.55 Hits/Page

iii)

Which algorithm performs best with 3 frames?

- Obviously, the optimal algorithm performs best because it is made by using the reference string (“seeing the future”). This is why it is simply used as a baseline for evaluating the performance of algorithms.
  Between FIFO and LRU, FIFO performed better with a better hit ratio. This is because LRU assumes pages used recently will be used again soon, but this reference string repeatedly jumps between pages before fully reusing them, so “recent past use” did not predict “near future use” well.

Which algorithm performs best with 4 frames?

- As mentioned, optimal will perform best no matter what. However, between FIFO and LRU, FIFO performed better in 4 frames as well. Adding one more frame helps both algorithms have fewer faults, but LRU still makes poor choices because the recent past is not a good guess for what will be used next (for this reference string). So it sometimes removes a page that will be needed again soon.

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

- Going from 3 -> 4 frames:
    - FIFO faults: 16 -> 10 (better)
    - LRU faults: 19 -> 17 (better)
    - Optimal faults:12 -> 9 (better)
- All three algorithms have fewer page faults when more frames are available. This is usually the case, that fault count doesn’t increase when you add frames. However, in FIFO, this isn’t the case sometimes due to Belady’s anomaly (more frames can actually cause more faults).

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

- Optimal needs to know the entire future page reference string to decide which page will be used farthest in the future at each replacement. However, since real-world operating systems don’t know the future memory accesses of processes, this is not possible to use in real-time. Instead, it is only used in theory, for giving a lower-bound on the number of faults to compare with real algorithms.

Compare the performance of FIFO and LRU

- While FIFO is cheap to implement and simple, it still has some disadvantages. It can throw out a heavily used page just because it was loaded long ago. It can also lead to Belady’s anomaly (more frames can actually cause more faults).
- LRU relies on the idea that recently used pages are likely to be used again, which can be better depending on the workload. However, it’s more expensive to implement.

When might FIFO be better or worse than LRU?

- FIFO is better than LRU when the access pattern breaks the assumption that "recently used" is a good predictor of "soon to be used". This can occur in cyclic patterns, like the reference string used in this question.
- FIFO can be worse than LRU in workloads where programs tend to reuse the same pages within a short time (most real-world examples). In these situations, LRU usually keeps the working set in memory, while FIFO can remove hot pages just because they are "old".

[Individual Submission - Student 2] What is the LFU algorithm?

- LFU is a 'cache-replacement' algorithm that removes the item that has been used the least number of times. (Least Frequently Used)
- Each item in the cache keeps a usage count, and whenever a new item is to be inserted, the least used item gets removed. If there is a tie between the frequencies, it depends on implementation. It's good for when you have popular items that have and will stay popular for a long time.

2.

a) Without a TLB, we must go to the page table in main memory. For a paged memory reference, the CPU must read the page table entry (PTE) from memory, then read the actual data from memory (two total main memory accesses).
120ns x 2 = 240ns
Therefore, a paged memory reference takes 240 ns without a TLB.

b) For a TLB Hit: 20ns to look up TLB and find the PTE, then only one memory access for the actual data.

- 20 ns + 120 ns = 140 ns

For a TLB Miss: 20ns to look up TLB and fail, then 120ns to read the PTE from memory, then 120ns to read the actual data.

- 20 ns + 120 ns + 120 ns = 260 ns

Then compute EMAT: (P=Probability, T=Time)

- EMAT = $(P_{Miss} \times T_{Miss}) + (P_{Hit} \times T_{Hit})$
  EMAT = (0.05 x 260 ns) + (0.95 x 140 ns)
  EMAT = 146 ns

Therefore, the EMAT is 146 ns with the TLB and a 95% hit ratio.

c) A TLB generally improves performance because the TLB is a cache for recebt page table entries. And for programs that tend to access the same pages repeatedly (loops, arrays, stack, etc.), we hit the TLB frequently. With these hits, we avoid the extra memory access for the page table. For example, in (b), we avoided an extra 120ns for a hit.

However, there are some situations where having a TLB can be worse. When the hit ratio is very low, you have the TLB overhead plus two memory accesses. Another possibility is if the process keeps getting pages that never stay in the TLB (unique pages). This will cause constant misses, adding overhead.

3.

a) Logical address space: 128 pages
Page size: 4096 bytes = 2^{12} -> 12 bits for offset
Number of pages: 128 = $2^7 \rightarrow$ 7 bits for the page number

Size = page number + offset
Size = 7 + 12 = 19 bits

The processor's logical address format is 7 bits for page number, 12 bits for offset. The size is 19 bits.

b) The page table has one entry per logical page. Therefore, since logical pages = 128, page table length = 128 entries.

Width = number of bits needed to store the frame number
Width = physical memory size / frame size
Width = 512 Kb / 4Kb
Width = 128 frames
128 = 2^7. Therefore, 7 bits are needed to uniquely number all frames.

The required length is 128 entries, and the width is 7 bits per entry.

c) If the physical memory becomes 256 KB:
Width = number of bits needed to store the frame number
Width = physical memory size / frame size
Width = 256 Kb / 4Kb
Width = 64 frames
64 = 2^6. Therefore, 6 bits are needed to number all frames.

The page table width decreases from 7 bits to 6 bits per entry, because fewer bits are needed to index 64 frames instead of 128.

[Individual Submission - Student 1] Explain what the physical memory space is

- The physical memory space is the physical RAM. This is the real hardware storage where instructions and data reside. It consists of real physical addresses and is managed by the OS in fixed-size frames.
- In physical memory, there's many things like the user process code, the OS, page tables, device drivers, shared libraries, stacks, heaps, global variables, etc.

[Individual Submission - Student 2] Explain what the logical memory space is

- The logical memory space is the space that the program thinks it has. This is independent from the real amount of physical RAM it may have. Generally, every program is given it's own logical memory space started at address 0, then it uses these address spaces. The operating system translates this to physical addresses in RAM. This makes it so the programs think they have a large amount of continuous memory, even though the OS may be fragmenting, sharing, or using a smaller amount of space for the data.

4. When a process executes the lseek(fd, offset, SEEK_END) system call:

- The kernel saves the current user context
- Trap is issued to switch to kernel mode
- The kernel gets the arguments: fd, offset, and whence = SEEK_END
- The kernel checks that fd is a valid index in the PCB and refers to an open file
  - If invalid, it returns -1
- If valid, that table entry points to a system-wide open file table entry.
  - Note: The open file table entry stores things like the current file offset, open mode/flags, and a pointer to the file's inode (or vnode) that describes the file on disk.
    - This inode contains things like file size, permissions, and pointers to data blocks
- If the inode is not already in memory, the kernel might need to read it from disk (one disk I/O). If it's already cached, it's just a memory access
- The kernel reads the file size from the inode
- Computes the new file offset for SEEK_END: new_offset = filesize + offset(argument)
- The kernel checks that new_offset is not negative
  - Return -1 if negative
- If valid, the kernel writes new_offset into the file offset field of the open file table entry
- The kernel returns new_offset to the process as the return value of lseek
- Control goes back to user mode
  - The next read() or write() on this fd will start from this new file position

5.

a) Block size = 8Kb
   Pointer Size = 4 bytes
   Number of pointers per block = block size / pointer size = 8192 / 4 = 2048 bytes

   Each inode has: 12 direct block pointers, 1 single-indirect, 1 double-indirect, 1 triple-indirect

   Direct blocks: 12 blocks
   Single-indirect:
      1 block full of pointers -> 2048 pointers -> 2048 data blocks
   Double-indirect:

1 block of pointers to indirect blocks (2048 pointers), and each of those points to a block with 2048 pointers to data blocks.

$2048 \times 2048 = 2048^2$ data blocks

Triple-indirect: (same logic as double but with an additional layer)

$2048 \times 2048 \times 2048 = 2048^3$ data blocks

Max file size = total number of blocks x the size of each block

Max file size = $(12 + 2048 + 2048^2 + 2048^3) \times 8192$

Max file size = 70403120791552 ≈ 64 TB

b) If you need to store a file that is larger than the maximum size computed, you can:
- Increase block size
  - Ex: (8 Kb -> 16 Kb)
    Max file size = $(12 + 2048 + 2048^2 + 2048^3) \times 16384 \approx 128$ TB
- Add level of indirect
  - Ex: (quadruple indirect)
    Max file size = $(12 + 2048 + 2048^2 + 2048^3 + 2048^4) \times 8192 \approx 131136$ TB
- Increase number of direct/indirect pointers in the inode
- Use multiple inodes and an “index” file (split the logical file into several smaller physical files)

Repositories:
- https://github.com/KushalPoudel6/SYSC4001_A3_P1
- https://github.com/amit46/SYSC4001_A3_P2