SYSC3600

Assignment #3 - Part 3

Carleton University Fall 2025

Eric McFetridge #101310942

Justin Winaford #101199245

3.1

Part 3.1 – Student 1

The LRU algorithm is a page swapping algorithm that allows us to swap a certain page that is currently allocated based on its usage recency. This means that when a page fault occurs due to full memory, the page swapping algorithm will always swap the least recently used page. To do this, we maintain a list of the last time each page was accessed by the CPU. Whichever page has the greatest time since its last access is assumed to be the least important and is swapped. This algorithm ensures our most recently used data should essentially never get swapped, as it will always have a low last accessed time compared to other pages. Programs that do not frequently access certain pages will often get swapped when idle. If we had 20 pages available, arranged in chronological order by last access time, page 20 would always be swapped since it has the greatest time since last accessed.

3.1 – Student 2

The LFU algorithm swaps pages based on their frequency of use rather than recency. We use LRU to swap pages based on the amount of time it has been since they were last accessed. Least frequently used tracks the overall amount of accesses to a page, and swaps them based on their total count, not just the time they were last accessed. LRU and LFU are similar in concept, but one focuses on time based usage while the other focuses on on count based usage. We can use this to ensure we never experience page faults for data that gets accessed many times throghout a programs execution.

i) We can implement the paging algorithms by implementing a basic python simulator to observe how they will perform with our given reference string. The algorithms below simulate each of the desired paging strategies and logs their output. This data can then be used to calculate the efficiency of each algorithm. We simply make the frame number a variable to change it for both parts.

```
CONFIG

reference_string = [415, 305, 502, 417, 305, 415, 502, 518, 417, 305, 415, 502, 520, 518, 417, 305,
502, 415, 520, 518]

frames = 3

    for step, page in enumerate(reference_string, start=1): #FIFO ALGORITHM

        if page in frame_set:

            hits += 1

            status = "hit"

        else:

            faults += 1

            status = "fault"

            if len(frames) < frame_count:

                frames.append(page)

                frame_set.add(page)

            else:

                victim = frames.popleft()  # remove oldest page

                frame_set.remove(victim)

                frames.append(page)

                frame_set.add(page)

        trace.append((step, page, list(frames), status))

    return faults, hits, trace


    for step, page in enumerate(reference_string, start=1): #LRU ALGORITHM

        if page in frames:

            hits += 1

            status = "hit"

            # move page to most recently used

            frames.remove(page)

            frames.append(page)

        else:

            faults += 1

            status = "fault"

            if len(frames) < frame_count:

                #space available

                frames.append(page)

            else:

                #evict least recently used
```

```
            frames.pop(0)
            frames.append(page)
    trace.append((step, page, list(frames), status))
return faults, hits, trace



for step, page in enumerate(reference_string, start=1): #OPTIMAL ALGORITHM
    if page in frames:
        hits += 1
        status = "hit"
    else:
        faults += 1
        status = "fault"
        if len(frames) < frame_count:
            frames.add(page)
        else:
            # Find the page to evict
            farthest_use = -1
            victim = None
            for f in frames:
                try:
                    next_use = reference_string[step:].index(f)  # relative to current step
                except ValueError:
                    # page not used again evict immediately
                    victim = f
                    break
                if next_use > farthest_use:
                    farthest_use = next_use
                    victim = f
            frames.remove(victim)
            frames.add(page)
    trace.append((step, page, list(frames), status))
return faults, hits, trace
```

FIFO Results (3 Frames):

Algorithm	Page Faults	Hits	Hit Ratio
FIFO	16	4	0.200

Step	Page	Frames after access	Status
1	415	[415]	fault
2	305	[415, 305]	fault
3	502	[415, 305, 502]	fault
4	417	[305, 502, 417]	fault
5	305	[305, 502, 417]	hit
6	415	[502, 417, 415]	fault
7	502	[502, 417, 415]	hit
8	518	[417, 415, 518]	fault
9	417	[417, 415, 518]	hit
10	305	[415, 518, 305]	fault
11	415	[415, 518, 305]	hit
12	502	[518, 305, 502]	fault
13	520	[305, 502, 520]	fault
14	518	[502, 520, 518]	fault
15	417	[520, 518, 417]	fault
16	305	[518, 417, 305]	fault
17	502	[417, 305, 502]	fault
18	415	[305, 502, 415]	fault
19	520	[502, 415, 520]	fault
20	518	[415, 520, 518]	fault

LRU Results (3 Frames):

Algorithm	Page Faults	Hits	Hit Ratio
LRU	19	1	0.050

Step	Page	Frames after access	Status
1	415	[415]	fault
2	305	[415, 305]	fault
3	502	[415, 305, 502]	fault
4	417	[305, 502, 417]	fault
5	305	[502, 417, 305]	hit
6	415	[417, 305, 415]	fault
7	502	[305, 415, 502]	fault
8	518	[415, 502, 518]	fault
9	417	[502, 518, 417]	fault
10	305	[518, 417, 305]	fault
11	415	[417, 305, 415]	fault
12	502	[305, 415, 502]	fault
13	520	[415, 502, 520]	fault
14	518	[502, 520, 518]	fault
15	417	[520, 518, 417]	fault
16	305	[518, 417, 305]	fault
17	502	[417, 305, 502]	fault
18	415	[305, 502, 415]	fault
19	520	[502, 415, 520]	fault
20	518	[415, 520, 518]	fault

Optimal Results (3 Frames):

Algorithm	Page Faults	Hits	Hit Ratio
Optimal	12	8	0.400

Step	Page	Frames after access	Status
1	415	[415]	fault
2	305	[305, 415]	fault
3	502	[305, 502, 415]	fault
4	417	[305, 417, 415]	fault
5	305	[305, 417, 415]	hit
6	415	[305, 417, 415]	hit
7	502	[305, 417, 502]	fault
8	518	[305, 417, 518]	fault
9	417	[305, 417, 518]	hit
10	305	[305, 417, 518]	hit
11	415	[417, 518, 415]	fault
12	502	[417, 518, 502]	fault
13	520	[417, 518, 520]	fault
14	518	[417, 518, 520]	hit
15	417	[417, 518, 520]	hit
16	305	[305, 518, 520]	fault
17	502	[518, 502, 520]	fault
18	415	[518, 520, 415]	fault
19	520	[518, 520, 415]	hit
20	518	[518, 520, 415]	hit

ii) **FIFO Results (4 Frames):**

Algorithm	Page Faults	Hits	Hit Ratio
FIFO	10	10	0.500

Step	Page	Frames after access	Status
1	415	[415]	fault
2	305	[415, 305]	fault
3	502	[415, 305, 502]	fault
4	417	[415, 305, 502, 417]	fault
5	305	[415, 305, 502, 417]	hit
6	415	[415, 305, 502, 417]	hit
7	502	[415, 305, 502, 417]	hit
8	518	[305, 502, 417, 518]	fault
9	417	[305, 502, 417, 518]	hit
10	305	[305, 502, 417, 518]	hit
11	415	[502, 417, 518, 415]	fault
12	502	[502, 417, 518, 415]	hit
13	520	[417, 518, 415, 520]	fault
14	518	[417, 518, 415, 520]	hit
15	417	[417, 518, 415, 520]	hit
16	305	[518, 415, 520, 305]	fault
17	502	[415, 520, 305, 502]	fault
18	415	[415, 520, 305, 502]	hit
19	520	[415, 520, 305, 502]	hit
20	518	[520, 305, 502, 518]	fault

LRU Results (4 Frames):

Algorithm	Page Faults	Hits	Hit Ratio
LRU	17	3	0.150

Step	Page	Frames after access	Status
1	415	[415]	fault
2	305	[415, 305]	fault
3	502	[415, 305, 502]	fault
4	417	[415, 305, 502, 417]	fault
5	305	[415, 502, 417, 305]	hit
6	415	[502, 417, 305, 415]	hit
7	502	[417, 305, 415, 502]	hit
8	518	[305, 415, 502, 518]	fault
9	417	[415, 502, 518, 417]	fault
10	305	[502, 518, 417, 305]	fault
11	415	[518, 417, 305, 415]	fault
12	502	[417, 305, 415, 502]	fault
13	520	[305, 415, 502, 520]	fault
14	518	[415, 502, 520, 518]	fault
15	417	[502, 520, 518, 417]	fault
16	305	[520, 518, 417, 305]	fault
17	502	[518, 417, 305, 502]	fault
18	415	[417, 305, 502, 415]	fault
19	520	[305, 502, 415, 520]	fault
20	518	[502, 415, 520, 518]	fault

Optimal Results (4 Frames):

Algorithm	Page Faults	Hits	Hit Ratio
Optimal	9	11	0.550

Step	Page	Frames after access	Status
1	415	[415]	fault
2	305	[305, 415]	fault
3	502	[305, 502, 415]	fault
4	417	[305, 417, 502, 415]	fault
5	305	[305, 417, 502, 415]	hit
6	415	[305, 417, 502, 415]	hit
7	502	[305, 417, 502, 415]	hit
8	518	[305, 417, 518, 415]	fault
9	417	[305, 417, 518, 415]	hit
10	305	[305, 417, 518, 415]	hit
11	415	[305, 417, 518, 415]	hit
12	502	[305, 417, 518, 502]	fault
13	520	[417, 518, 520, 305]	fault
14	518	[417, 518, 520, 305]	hit
15	417	[417, 518, 520, 305]	hit
16	305	[417, 518, 520, 305]	hit
17	502	[518, 520, 305, 502]	fault
18	415	[518, 520, 502, 415]	fault
19	520	[518, 520, 502, 415]	hit
20	518	[518, 520, 502, 415]	hit

iii) Based on our results we can observe that for both number of frames, optimal performs best. This makes sense and is expected since optimal is a theoretical algorithm that is able to predetermine access rate. We can also see that the performance of all 3 algorithms improve when adding more frames. This is also expected since more frames means there is a higher chance of our desired page being loaded. The optimal algorithm is not practical in real life because we will not be able to look ahead in our reference string to know which frames actually should be swapped. This behavior depends entirely on having a predetermined access pattern. In real life we can optimize to get behavior similar to this using caching and other methods, but fully predetermining execution time access is unpredictable. LRU performed consistently lower than the other two algorithms this is because it would only typically outperform FIFO when the program performs a lot of repetitive accesses. Our test string used did not have enough repetition, and we did not test a large amount of traces.

3.2

a) with no TLB we can assume that every memory lookup involves two steps. First accessing the page table which is stored in main memory (120ns access time), second is accessing the actual frame of data returned by the page table (another 120ns) this gives us a total of 240ns access time.

b) if we have a TLB with a hit rate of 95% and 20ns of operation overhead, to get our effective operating time we can calculate the average access time for 100 operations. Hit time is 120ns access overhead + 20ns TLB overhead = 140ns. Miss time is 120ns access overhead + 120ns table overhead + 20ns TLB overhead = 260ns total. We assume 95 hits and 5 misses for 100 accesses. This means that our average is: 95 * 140ns + 5 * 260ns. Divided by 100 accesses. (13300 + 1300) / 100 = 146ns. This means that our effective operating time is 146ns when we assume normal operation for long periods of time since this is the average overhead per access.

c) Generally adding a TLB improves performance because we are caching frequent memory accesses. This always reduces overhead that was originally present from the memory table being stored in main memory. This means that everytime we make one of these accesses in the future we save on operating time due to having the TLB. There are rare cases in which we can actually lose time due to the TLB, but most of the time it will save time because programs perform predictable access operations. An example of a case that causes bad TLB overhead is a program with no repeated accesses. The TLB will be useless since it will never contain the address we are looking up next. We will always have to hit the main page table and then lose an additional few ns. This then makes the program run a bit slower than if no TLB was present.

3.3

Part 3.3 – Student 1

The physical memory space is the actual amount of hardware memory space that you have on your computer. Such as a traditional 4, 8, or 16 GB multi-stick RAM arrangement on consumer PC's. No matter what type of actual RAM hardware you have, the physical memory space is the actual physical memory that the CPU can access, eg, the real, limited addressable memory. This excludes all the virtual memory that the OS uses to extend our logical memory size, only counting the real memory where parts of currently executing programs are located.

Part 3.3 – Student 2

The logical memory space is the memory space that the OS orchestrates alongside the devices physical memory management unit. We can expand the space well beyond the physical memory capabilities of the machine. For example many modern systems can have terabytes of addressable logical memory using 64 bit addressing. There are upper limits to this based on the operating system. This allows the OS to store additional data on things like hard drives using page swapping. This logical space maps to a more limited amount of physical memory frames through a page table, however since we typically have a much larger amount of logical memory we can't map each individual logical page to a phyiscal frame. The OS must keep track of which frames are allocated to which data.

a)

512 kB = 512 * 1024 = 524,288 bytes -> 524,288=2^19.

So the logical address size must be 19 bits.

b)

The page table must include 128 entries (512kB/4kB). Since we have 128 frames the page table width in bits must be 7 bits. This is because to address 128 physical frames we need 2^7 addresses.

c)

If we half the size of physical memory we must also half the size of the page table width. This means going from 128 entries to 64 entries. Which is the same as doing 2^7 -> 2^6. This means our new table width is 6 bits.

4.

Lseek() is a system call which means we must go through the standard process of using the system api. We take the input syscall with its parameters and then switch our execution context to kernel mode. The kernel takes the input parameters from userspace and returns safely with an error if one of the values is bad. The kernel then looks up the provided file descriptor from the PCB, and then proceeds or errors based on whether or not this is a valid seekable object. If it is, the file object points to an inode for the file, and the file flags and file position. The kernel then performs locking on the inode so that while it is being modified we don't accidentially create a race condition by having multiple processes change the same files metadata. Inode should be loaded into memory if it isn't already. We then need to get the size of the file to perform end offset seeking. The kernel computes the new file position by applying an offset to the file size. The kernel then updates the file object assuming no overflow error occurs during the offset operation. We now have the updated file position stored in our active file object. We then unlock our inode and file lock and return the updated position to the invoking process. This takes us back out of the kernel level and back into user space with the syscall return.

5.

a)

Direct blocks: 12 * 8kB blocks = 98,304 bytes

Pointers per block = Block size / Pointer Size = 8192 bytes / 4 byes = 2048 bytes.

Indirect 1 = 2048 * 8192 = 16,777,216 bytes

Double = 20482 * 8192 = 34,359,738,368 bytes

Triple = 20483 × 8192 = 70,368,744,177,664 bytes

Sum of all blocks: 98,304 + 16,777,216 + 34,359,738,368 + 70,368,744,177,664

= 70,403,120,791,552 bytes = 70.40312 TB

This is the largest file we can store with 12 direct access blocks and triple indirect.

b)

to store a file even larger than this theoretical maximum there are multiple techniques we can employ. The first is to simply modify the nature of the filesystem itself. This can be done by increasing number of blocks, block size, pointer size etc, but this fundamentally changes the architecture of our filesystem. Therefore, to store a larger file without changing any underlying implementations we will have to split the file into parts. Similar to how we split an applications memory between RAM and swap when we hit our operating limits. This means that we could store a 211.21248 TB file as 3 70.40312 TB files. This allows us to combine multiple files into a single larger one. This can be done by just maintaining a short list of the grouped files in the large file.