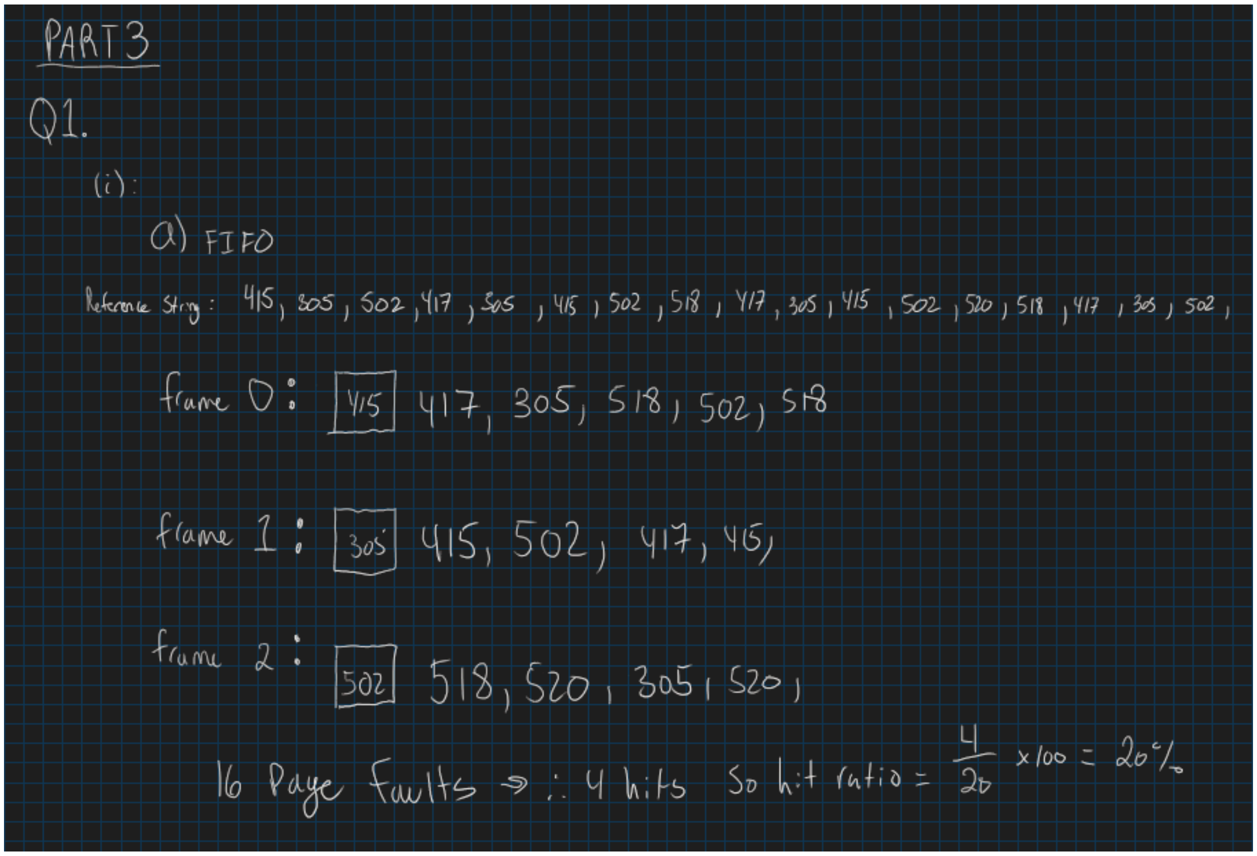
SYSC4001 - Theory Questions : Esli Emmanuel Konate 101322259

1. (Explanations for algorithms are at the end of Question 1.)



PART 3

Q1.

(i):

a) FIFO

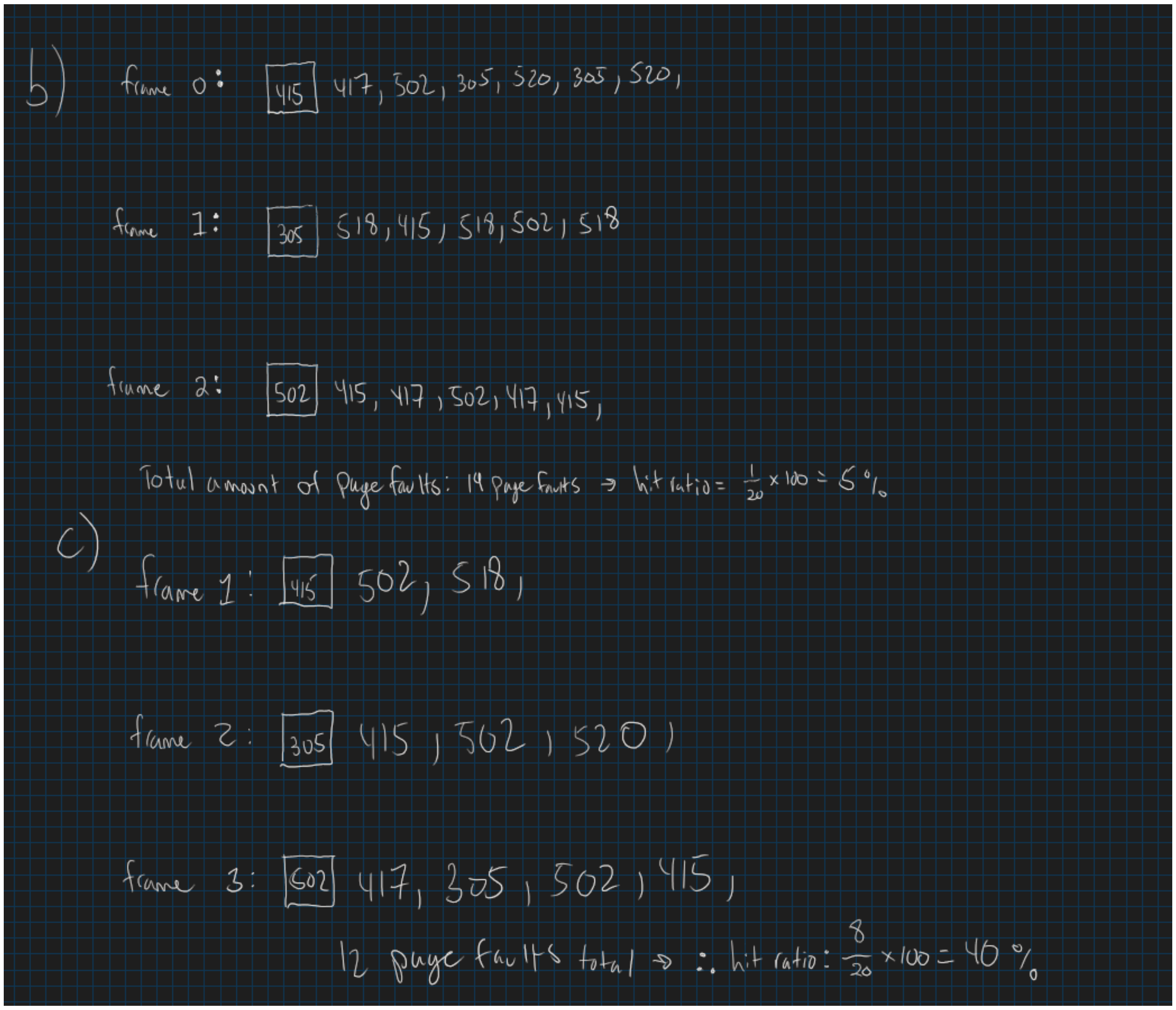
Reference String: 415, 305, 502, 417, 305, 415, 502, 518, 417, 305, 415, 502, 520, 518, 417, 305, 502,

frame 0: $\boxed{415}$ 417, 305, 518, 502, 518

frame 1: $\boxed{305}$ 415, 502, 417, 415,

frame 2: $\boxed{502}$ 518, 520, 305, 520,

16 Page faults $\Rightarrow$ $\therefore$ 4 hits So hit ratio = $\frac{4}{20} \times 100 = 20\%$



b) frame 0: $\boxed{415}$ 417, 302, 305, 520, 305, 520,

frame 1: $\boxed{305}$ 518, 415, 518, 502, 518

frame 2: $\boxed{502}$ 415, 417, 502, 417, 415,

Total amount of page faults: 19 page faults $\Rightarrow$ hit ratio = $\frac{1}{20} \times 100 = 5\%$

c) frame 1: $\boxed{415}$ 502, 518,

frame 2: $\boxed{305}$ 415, 502, 520,

frame 3: $\boxed{502}$ 417, 305, 502, 415,

12 page faults total $\Rightarrow$ $\therefore$ hit ratio: $\frac{8}{20} \times 100 = 40\%$

a)

(ii) f0: $\boxed{415}$ 518, 502,

f1 $\boxed{305}$ 415, 518

f2 $\boxed{502}$ 520,

f3 $\boxed{417}$ 305,

10 Page faults in total ⇒ hit ratio = $\frac{10}{20} \times 100 = 50\%$

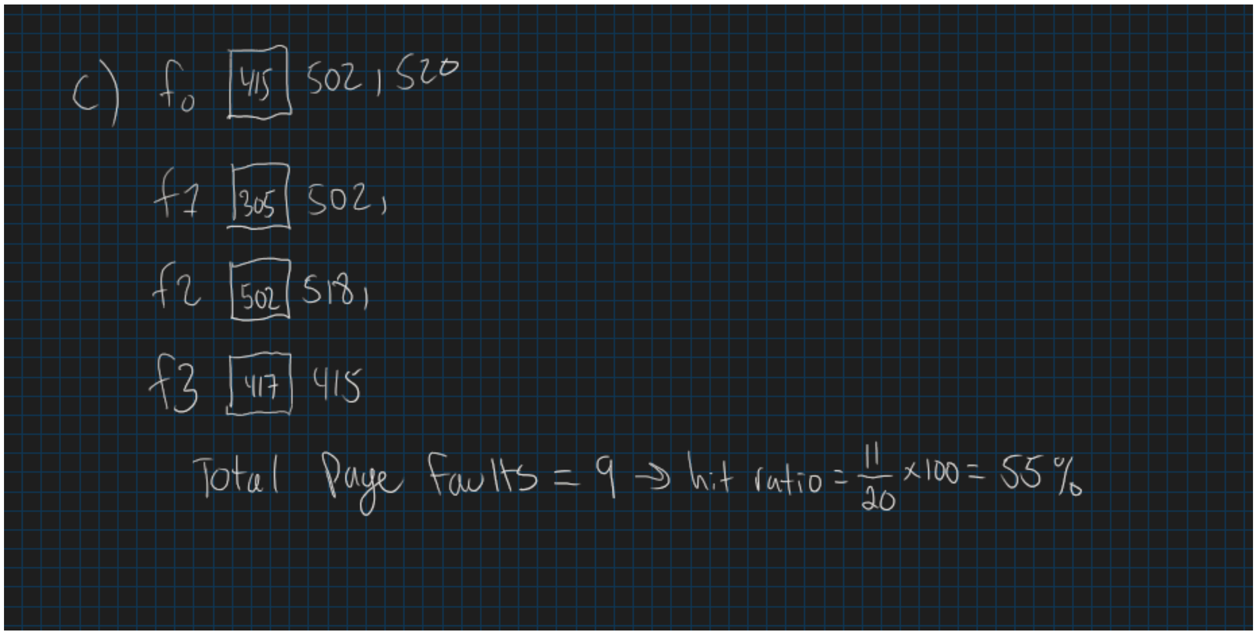
b) f0 $\boxed{415}$ 305, 518, 415,

f_1 $\boxed{305}$ 417, 520, 502

f_2 $\boxed{502}$ 415, 417, 520

f_3 $\boxed{417}$ 518, 502, 305, 518

Page faults = 17 ⇒ $\frac{3}{20} \times 100 = 15\%$ ⇒ hit ratio



Total Page faults = 9 → hit ratio = $\frac{11}{20} \times 100 = 55\%$

iii)

Which algorithm performs best with 3 frames and why?

Optimal performs the best with 3 frames with only 12 page faults and 40% hit ratio. Compared to FIFO's 16 faults and 20% hit ratio and LRU's 19 faults with 5% hit ratio. Optimal is the best possible replacement choice because it replaces the page that's not going to be used for the largest period of time. The textbook highlights that Optimal "guarantees the lowest possible page-fault rate for a fixed number of frames" and that "no replacement algorithm can process this reference string in three frames with fewer than [the optimal number of] faults". (Silbershatz Chapter 10).

Which algorithm, performs best with 4 frames and why?

Again, it is Optimal with only 9 page faults with 55% hit ratio, compared to FIFO's 10 faults with 50 % hit ratio and LRU's 17 faults with 15% hit ratio. Optimal has knowledge of the future of the reference string and always makes the best replacement choices. It will always outperform or be equal to any other algorithm because it has perfect information about which page will not be needed for the longest time.

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

When going from 3 to 4 frames,

- FIFO goes from 16 to 10 page faults which are 6 page faults less.
- LRU goes from 19 to 17 page faults which are 2 page faults less.
- Optimal goes from 12 to 9 page faults which are 3 page faults less.

The link is that normally, as the number of frames increases, the number of page faults decreases. The textbook highlights: “as the number of frames available increases, the number of page faults decreases” (Silbershatz Chapter 10). It is also shown in the textbook with figure 10.11 of the textbook showing the curve where page faults drop as frames increase.

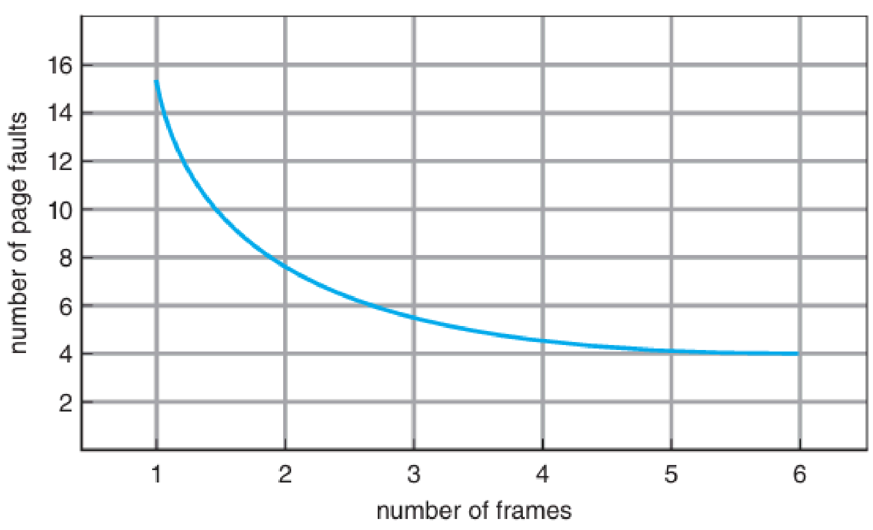


Figure 10.11 Graph of page faults versus number of frames.

But, it is also important to take into account an exception, FIFO can suffer from something called Belady’s anomaly, where the page-fault rate increase as the number of allocated frames increases (Silbershatz Chapter 10). In the current example of the assignment, FIFO improved as normal, but the anomaly can occur with specific reference strings. The textbook also highlights that “optimal replacement does not suffer from Belady’s anomaly” and both Optimal and LRU “belong to a class of page-replacement algorithms, called stack algorithms, that can never exhibit Belady’s anomaly”. (Silbershatz Chapter 10).
Why is the Optimal algorithm impractical in real-world OS?

Because it requires knowledge of what happens in the future. We have to know what will happen in the future in terms of reference string. The OS cannot predict which pages a process will reference in the future. The textbook says: “(We encountered a similar situation with the SJF CPU-scheduling algorithm in Section 5.3.2)”. So , just like we can’t predict future CPU burst times, we cannot predict the future page references. (Silbershatz Chapter 10). The optimal algorithm is actually used for comparison studies. For example it is used to know that even though a new algorithm is not optimal, it is within a specific percentage of optimal at worst and within another percentage on average. (Silbershatz Chapter 10).

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

The case for this assignment is special because FIFO performed better than LRU.

With 3 frames:

- FIFO had 16 page faults with 20% hit ratio
- LRU had 19 page faults with 5% hit ratio
- FIFO performed better

With 4 frames:

- FIFO had 10 page faults with 50% hit ratio
- LRU had 17 page faults with 15% hit ratio
- Again, FIFO performed better

FIFO might be worse than LRU when a program accesses a page in sequential or random pattern with no possible “temporal locality”, because recent usage cannot be used to predict future usage. Which is the case of this assignment question because it has a pattern where FIFO’s replacement method had better choices than LRU’s way. Another case where FIFO would be better than LRU is when a program cycles through more pages than available frames in a regular pattern. FIFO, which uses an approach using queues, can often work better with access patterns than LRU. Finally, FIFO is simpler to implement, because it is just a queue, so it has less overhead when trying to track and update page info.

Cases where FIFO might be worse than LRU are when the program makes use of temporal locality, so pages used recently will again be used in the future. That’s when LRU shines the most. Another case is that FIFO has a weakness which is Belady’s Anomaly, which are cases where the page fault rate increases as the number of frames increases. In the case of String which could lead to Belady’s Anomaly when using FIFO, LRU is the best choice because it does not suffer from Belady’s Anomaly. Also, FIFO can get rid of pages, which are frequently used because they have been in memory for the longest, which could be bad in the cases where that specific page is accessed constantly. LRU is best because it will keep that page until it is less frequently used while FIFO could get rid of it even though it is accessed multiple times frequently.

Student 1: Explain the LRU algorithm

LRU, also known as Least Recently Used algorithm, associates with each page the time when that page was last used. When a page has to be replaced, LRU will select the page that hasn’t been used for the largest amount of time. (Silbershatz Chapter 10). It resembles the Optimal algorithm but instead of looking into the future, LRU looks into the past. To implement it, it uses counters, which will be added to the CPU. The counter/clock will be incremented for every memory reference. When a reference to a page is made, the content of the clock register is then copied to the field associated with the time of use in the page-table entry for that specific page. This way, we have the time of the last reference to each page. We replace the page containing the smallest time value. Another way to implement LRU is to keep a stack of page numbers.

When a page is then referenced, it will be removed from the stack and put on the top. This way, the least recently used page is always at the bottom of the stack.

Advantages of LRU is that it is not affected by Belady’s Anomaly. It belongs to a class of page-replacement algorithms named stack algorithms, that can never suffer from Belady’s Anomaly. (Silbershatz Chapter 10).

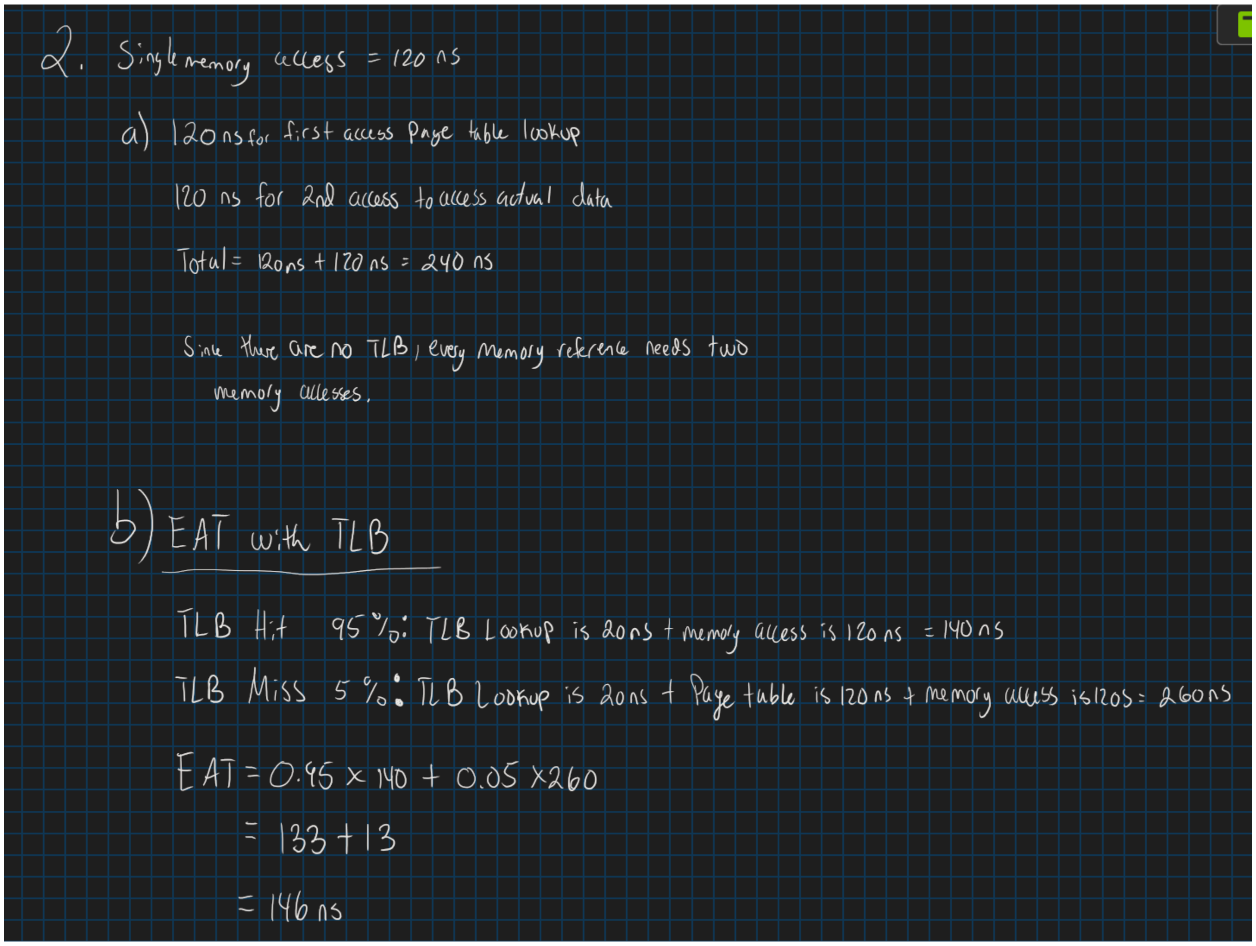
Student 2: Explain the LFU algorithm

LFU also known as Least Frequently Used algorithm, requires that the page with the smallest count is replaced. The page with the smallest count is replaced because the ones, which are frequently accessed will have a large reference count. (Silbershatz Chapter 10). So, it keeps a counter of the number of references that have been made to each page, when a page fault happens and we need to replace a page, the page with the smallest reference count will be evicted.

There is a problem with LFU though, is that when a page is used frequently at first by a process but then it's never used again, it will have a large count and will still be in memory even if it is not needed anymore. A solution to that problem would be to shift the counts right by 1 at normal intervals, which will form an average usage count which decays exponentially (Silbershatz Chapter 10). This will allow recent references to weigh more than past ones.

Finally, LFU is not commonly used because its implementation is expensive and they don’t approximate Optimal replacement correctly (Silbershatz Chapter 10).

2.



2. Single memory access = 120 ns

a) 120 ns for first access Page table lookup

120 ns for 2nd access to access actual data

Total = 120ns + 120 ns = 240 ns

Since there are no TLB, every memory reference needs two memory accesses.

b) EAT with TLB

TLB Hit 95%: TLB Lookup is 20ns + memory access is 120 ns = 140 ns

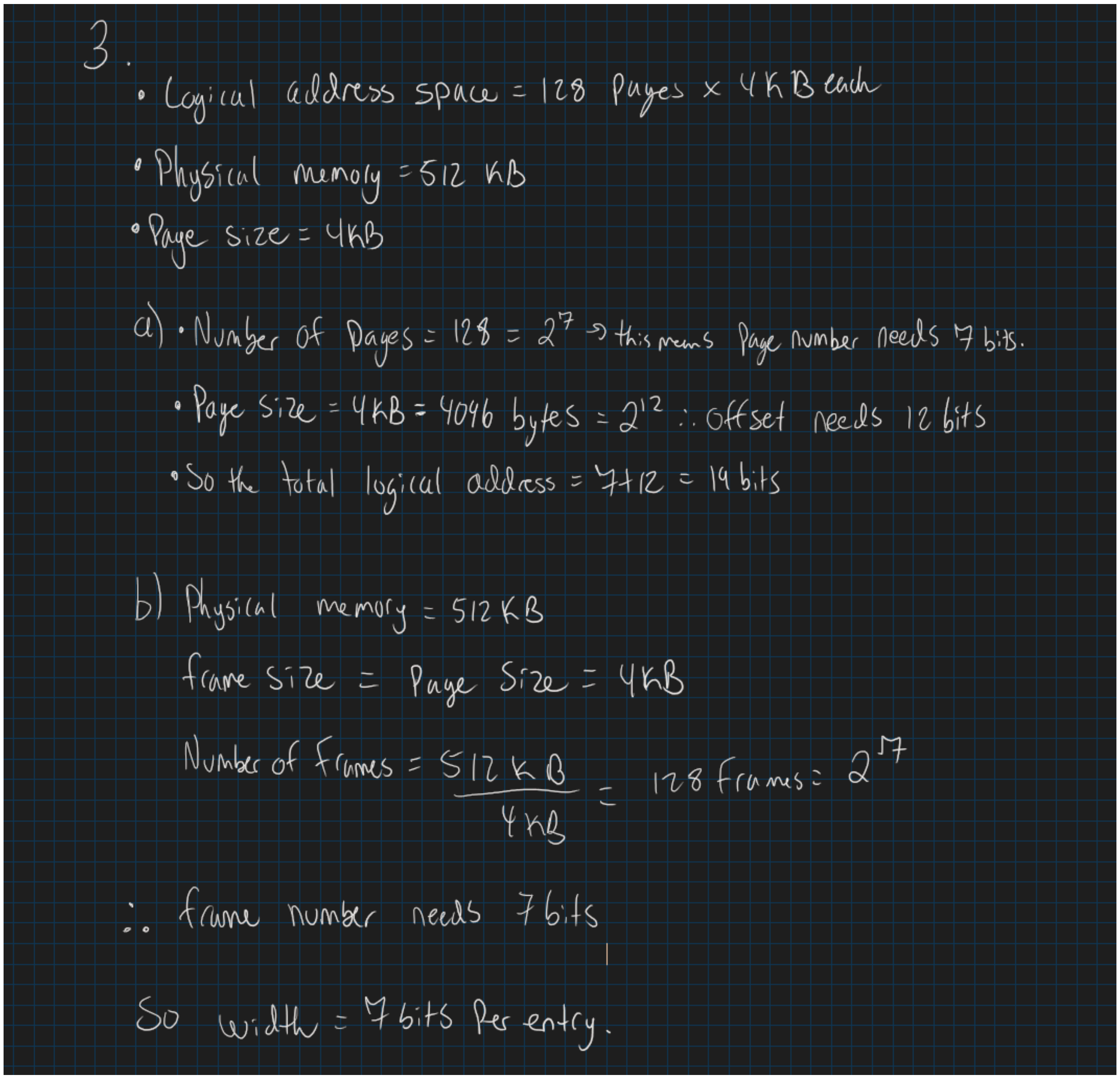
TLB Miss 5%: TLB Lookup is 20ns + Page table is 120 ns + memory access is120s = 260ns

$$EAT = 0.95 \times 140 + 0.05 \times 260$$
$$= 133 + 13$$
$$= 146 \text{ ns}$$

c)

The TLB improves performance by eliminating one memory access for most references. With a 95% hit ratio, we get 146 ns instead of a 240ns, which is a large improvement. The TLB may worsen performance in cases where there are very low hit ratios, the TLB overhead on every access makes performance worse than no TLB. In addition, in cases where the program has bad locality, where random access patterns that constantly miss in the TLB pay the 20ns overhead with no benefit. (Silbershatz p(366-368)).

3.



3.

- Logical address space = 128 Pages × 4 KB each
- Physical memory = 512 KB
- Page size = 4KB

a) • Number of Pages = 128 = 2^7 → this means Page number needs 7 bits.

- Page size = 4KB = 4096 bytes = 2^{12} ∴ Offset needs 12 bits
- So the total logical address = 7+12 = 19 bits

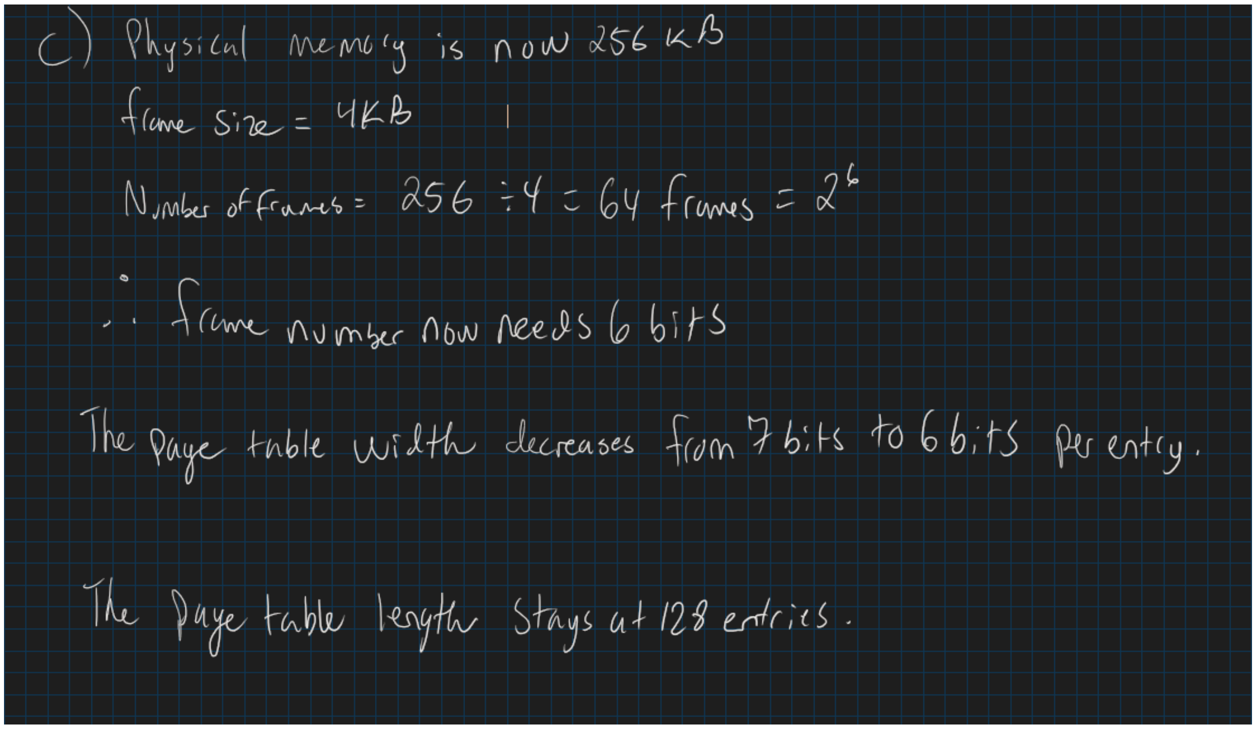
b) Physical memory = 512 KB

frame size = Page Size = 4KB

Number of Frames = $\frac{512\text{ KB}}{4\text{ KB}}$ = 128 frames = 2^7

∴ frame number needs 7 bits

So width = 7 bits Per entry.



c) Physical memory is now 256 KB

frame size = 4KB

Number of frames = 256 ÷ 4 = 64 frames = 2^6

∴ frame number now needs 6 bits

The page table width decreases from 7 bits to 6 bits per entry.

The page table length stays at 128 entries.

Student 1 - Physical Memory Space:

Physical memory space is another name for RAM, which is physical hardware memory available in the system. It has multiple storage cells, with a unique physical address that the memory hardware can access directly. Physical memory is organized into blocks of fixed size called frames. When the CPU needs data, it has to retrieve that data from a physical address in RAM. The physical address has a limit, which is the actual amount of RAM hardware present in the system. So if you have 4GB of RAM installed, the physical address space is 4GB. The OS manages physical memory by tracking which of the frames are free and which are allocated to processes.

Student 2 - Logical Memory Space:

Logical memory space is also known as virtual address space. It is the programmer's somewhat abstract view of memory as seen by currently running processes. From the program's perspective, it will be seen as having its own specific contiguous block of memory which starts at address 0. The program does not know that its memory is actually spaced throughout the physical RAM or that other processes are also sharing the same physical memory. The logical address space can be bigger than physical memory. For example, a 32-bit system gives 4GB logical address space even if only 512 MB of physical RAM exists. The separation allows programmers to write code without having to care about memory constraints. The MMU hardware will translate the logical addresses created by the CPU into physical addresses.

4.

1. The process will execute lseek(fd, offset, SEEK_END) which activates a system call for a transition from user mode to kernel mode.
2. The kernel will then use the file descriptor to index a table of open files for the currently running process. It will validate that the fd is within a valid range. It will also check that the entry exists, so that the file is actually open and finally it will return an error if fd is invalid.
3. Each entry in the process table has a pointer to a file structure in the system wide open file table. The kernel will then follow the pointer to access the file structure. The file structure has a current file offset, which is the position pointer that will be changed. It also has a pointer to the inode, access mode/permissions and a reference count.
4. The file structure points to the inode, which is an in-core copy of the inode on the disk. The kernel accesses the inode to take the file size, which will be used for SEEK_END calculations. The inode has a file size, file type, owner and permissions, timestamps, and pointers to data blocks.
5. Now there is computation of new_position = file_size + offset. Using the file size from the inode and the offset given by the user, the kernel calculates the new file position. SEEK_END means that the offset is relative to the end of the file. If offset is 0, the position is set to the end of the file. If the offset is positive, the position is set beyond the end. If the offset is negative, the position is set back from the end.
6. The kernel updates the file offset stored in the file structure with the calculated position. The offset will be kept in the file structure because multiple processes may have the same file open and they each need their own independent position pointer.
7. The system call returns the new file position to the calling process. If it was successful, it will return the new offset, else it will return -1.

Data Structures accessed:

1. Per process open file table to validate the fd.
2. System wide open file table (the file structure) which has the offset and updates it.
3. In-core inode which gives the file size for calculations.

5.

a)

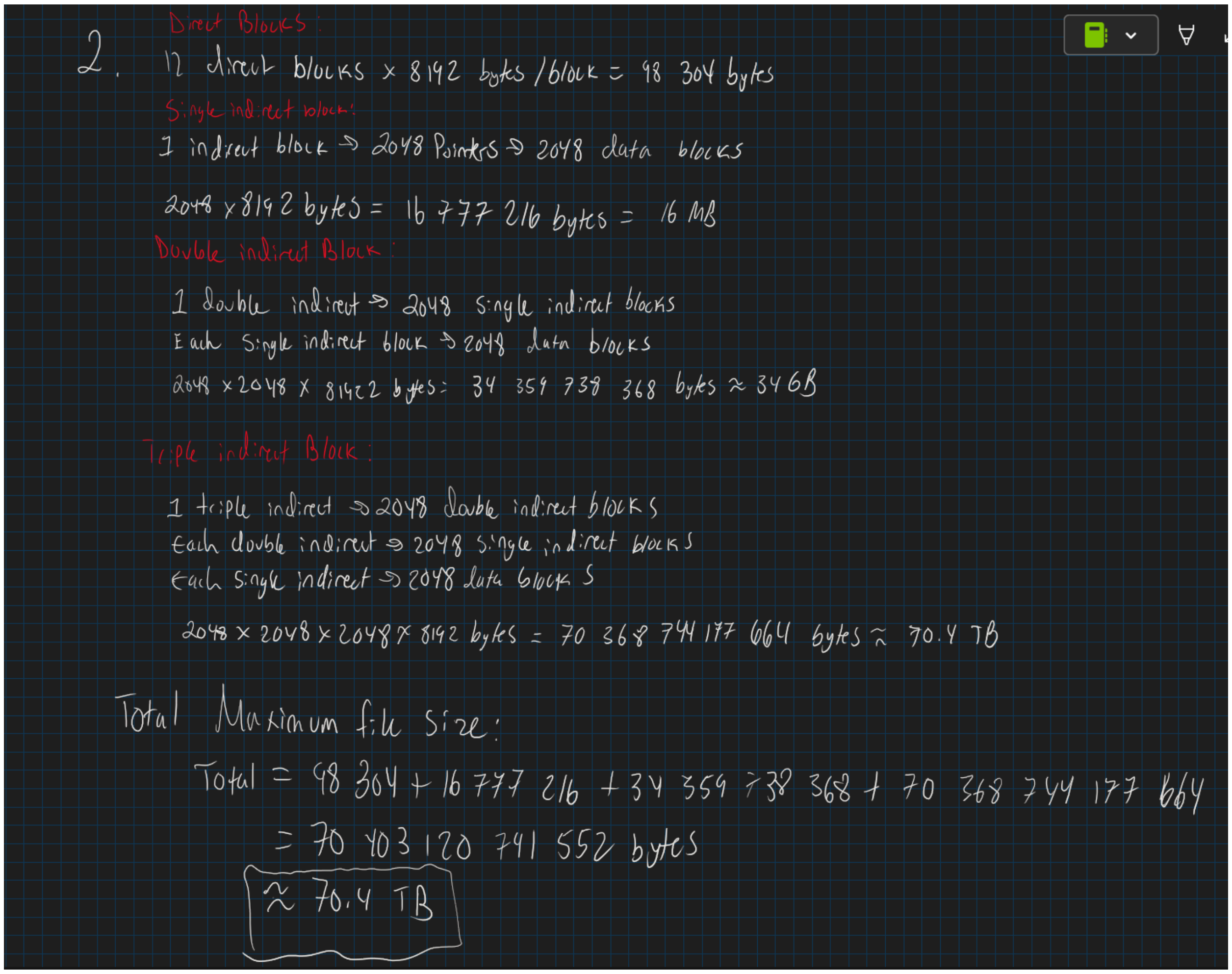
5.

a)
- Disk block size = 8KB = 8192 bytes
- Pointer size = 4 bytes
- 12 direct blocks
- 1 single indirect block
- 1 double indirect block
- 1 triple indirect block

I. Pointers Per block $= \dfrac{\text{Block Size}}{\text{Pointer size}}$

$$= \frac{8192 \text{ bytes}}{4 \text{ bytes}}$$

$=$ 2048 Pointers Per blocks



2. Direct Blocks:

12 direct blocks × 8192 bytes/block = 98 304 bytes

Single indirect block:

1 indirect block → 2048 Pointers → 2048 data blocks

2048 × 8192 bytes = 16 777 216 bytes = 16 MB

Double indirect Block:

1 double indirect → 2048 single indirect blocks

Each single indirect block → 2048 data blocks

2048 × 2048 × 8192 bytes: 34 359 738 368 bytes ≈ 34 GB

Triple indirect Block:

1 triple indirect → 2048 double indirect blocks

Each double indirect → 2048 single indirect blocks

Each single indirect → 2048 data blocks

2048 × 2048 × 2048 × 8192 bytes = 70 368 744 177 664 bytes ≈ 70.4 TB

Total Maximum file size:

Total = 98 304 + 16 777 216 + 34 359 738 368 + 70 368 744 177 664

= 70 403 120 741 552 bytes

$\boxed{\approx 70.4 \text{ TB}}$

b)

If you need to store a file bigger than the maximum inode capacity, you can split it into multiple smaller files and link them in a logical manner. But as said by the textbook files can't be split through multiple file systems for a single file but it is possible to use multiple files. So, it is possible to split the file using naming convention. To do so, we can split the large file into numbered parts that each fit within the maximum size. E.g, largefile.part1, largefile.part2, largefile.part3 which when added together are greater than the 70.4 TB but separated they are each below the threshold. It is also possible to use symbolic links by creating directory structure with symbolic links, which will point to the component files and give some sort of logical unity while also being physically split data. It could work just like unions E.g:

```
/large_file/
        Dt_001 (70TB)
        Dt_002 (70TB)
        Dt_003 (10TB)
Total = 150 TB
```

It is also possible to implement file management within the application, where the program will understand that the program is split and will in consequence handle reading and writing through multiple files.