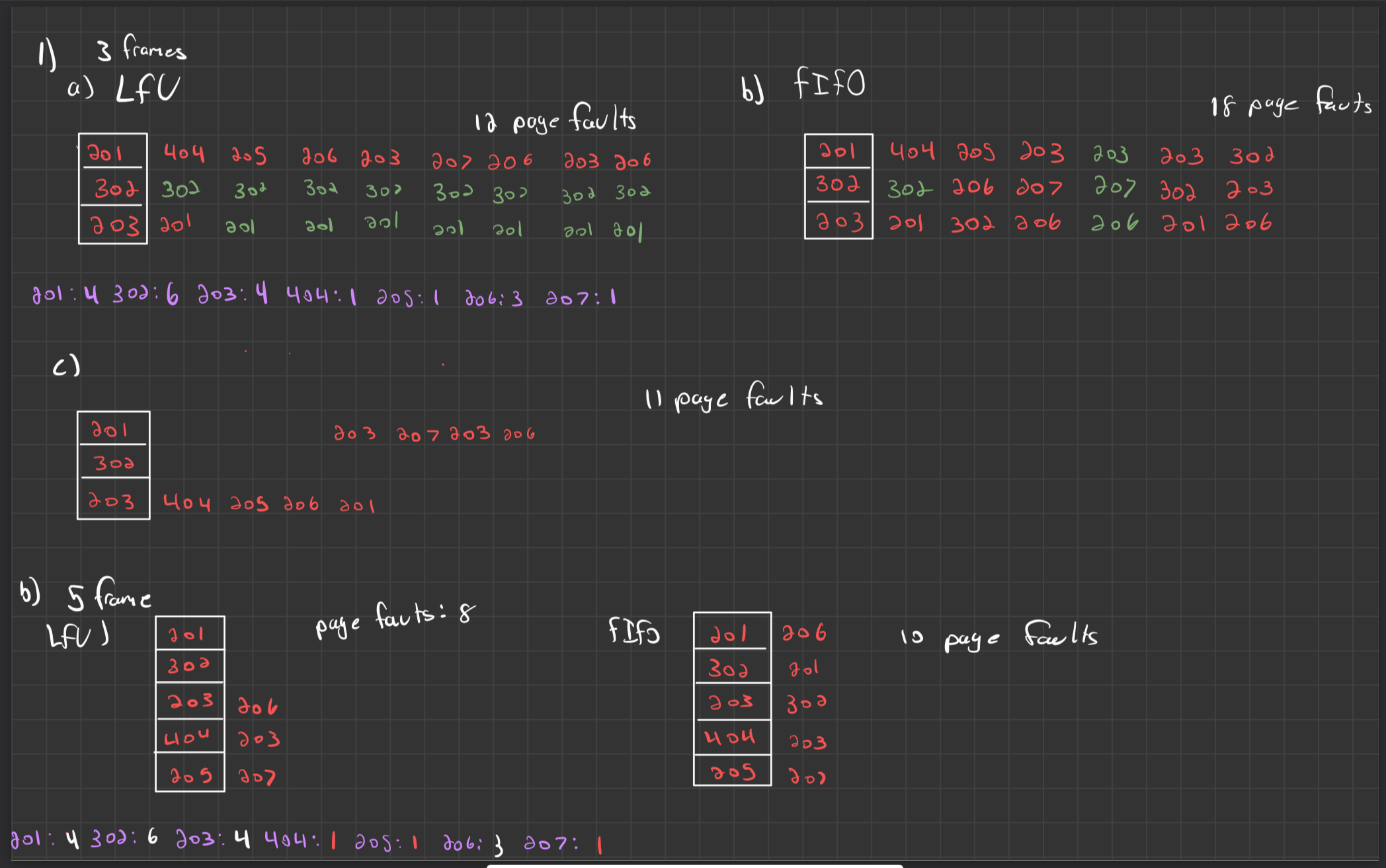
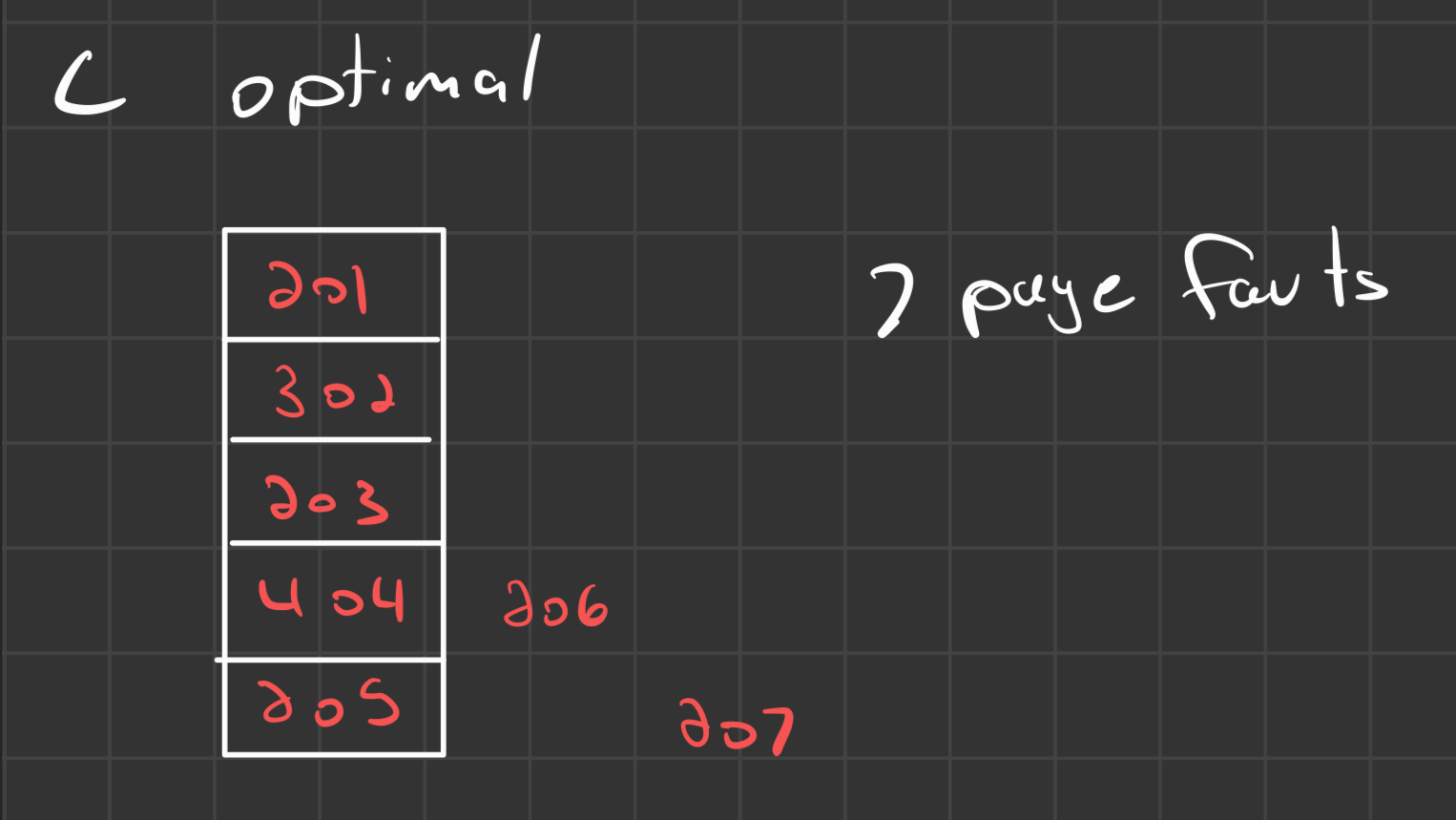
Question 1:





Question 2:

1. Each memory access requires 2 operations:
   1. Accessing the page table to translate the virtual address to a physical one
   2. Accessing the actual physical memory

This means that the Memory reference time is equal to 2 times the memory access time

(Memory reference time = 2 X Memory access Time). Knowing each memory access time is equal to 250 ns, we can conclude that the total memory reference time wil be 500ns.

1. Firstly the TLB reduces the need to access the page table on every memory reference. With the TLB, if there is a hit on the page then the virtual to physical memory translation is found within the TLB. This saves time as the page table lookup will not be necessary anymore (250ns + 30ns = 280ns hit time). If there is a miss in the TLB, it will be longer than the normal memory reference time since the TLB look up also takes time. The memory reference time will be the TLB lookup time plus the 2X memory access time (530 ns miss time). Now employing the effective memory access time (EMAT = (Hit ratio \* Hit Time) + (Miss ratio + Miss time).
   1. EMAT = (0.8\*280ns) + (0.2\*530ns) = 330 ns
2. There are a couple reasons why adding a TLB could improve performance and also degrade it. Firstly, the positives of adding another TLB would come from the faster address translation. The table caches the frequently used pages table entries which reduces the overhead of accessing the page table for most references. Furthermore, the TLB will capitalize on locality by storing translations for frequently used accessed pages. But there are also situations where the TLB may degrade performance. For example, if there is a low hit rate and the programming running has a large working set, the overall hit rate of the TLB will decrease. The more the hit rate decreases the more the overhead of lookups increases. Overall, the situations where the performance might be worse are edge cases and dont represent common programs. The majority of the time the TLB will be beneficial for reducing page look up time

Question 3:

1. To find the processor's logical address in a paged memory system, we need two pieces of information. One being the page number and the other being the page offset. Given the number of pages in our address space being 32 with a page size of 2 KBytes we first have to calculate the number of bits for the page offset. 2 Kbytes is 211 meaning we need 11 bits. Secondly to find the number of btis for the page numbers, we find that 25 = 32 meaning we need 5 bits for page numbers. This concludes that we need 16 bits for processoirs logical address format.
2. To find the length and width, we first start by calculating the number of page table entries we need. This will correspond to the length of the table and this is given as 32 entries or equivalently known as the number of pages. Secondly, to find the width, it is known that each page in the table entry contains a frame number of the corresponding page in memory. We need to calculate the number of bits needed for the frame number. This is done by dividing the physical memory size (2^20) over the page size (2^11) thus giving 2^9. This concludes that we need 9 bits for the frame number in each of the pages in the page table. And this is not counting any control bits needed.
3. In the case where physical memory is cut by half meaning instead of 1MB we have 524288 KB, we can reduce the bit count in our previous calculation from 2^20 to 2^19. With this, the new amount of bits needed for the frame number is reduced by 1 to 8 bits. Thus concluding the width of the page table is now 8 bits.

Question 4:

There are several steps while executing the write operation. Firstly, this execution starts with locating the file. The file system uses the file path provided during the open call to locate the file in the directory. Secondly there is the file permissions check. This process consists of the file system verifying for write permission for the user/process attempting the operation. If there is insufficient permission, the executing is aborted and an error is thrown. Thirdly the size of the file is determined. It retrieves the meta data to determine this information which will also lead to the end of the file. Because this is in APPEND more, the writing is performed at the end of the file which is why the metadata needs to be retrieved to get the pointer pointing to the end of the file. Next, if necessary disk blocks are allocated. This is done if the files current storage space isnt not big enough for appending new data. If so, the process of allocating said new disk blocks entails finding free blocks and updating the files metadata to reflect the newly added block. Continuing, the data appended is to be written to the new or formally existing disk block. The file pointer is updated to reflect the new end of file as it is being appended. With the data being written, the metadata of the file is also updated to show the changes done by the append. In the case where multiple processes are writing to the same file, concurrency issues need to be solved by using mechanisms like semaphores for example. This will insure the atomicity of the file system.

Question 5:

1. To determine the maximum size of a file that can be stored in this system, each of the following disk blocks can be calculated separately.
   1. Data stored in direct block: Each of the 12 direct pointers refer to a 8KB block. Simply multiply the number of blocks times the size will give the maximum size. Data from direct blocks = 12 \* 8 KB = 96 KB
   2. Data stored via a single indirect block: a single indirect block contains pointers to data blocks. To find the max size we calculate dividing the block size (2^13) by the pointer size (2^2) giving 2^11 pointers (2048 pointers). Then we simply multiply the number of pointers by each 8KB block giving 16 MB.
   3. Data stored via a double indirect block: A double indirect block contains pointers to single indirect blocks and each of those single indirect blocks contains a pointer to data blocks. Knowing this we can use our previously found value for the pointers per block (2048) And multiply it by the data from one single indirect block. 2048 \* 16MB = 32GB
   4. Data stored via triple indirect block. This is very similar to the double indirect block but with another layer. Meaning we can use the same process of multiplying the pointers per block value by data from one double indirect block. 2048 \* 32GB = 64 TB

We can conclude that the total maximum file size is around 64TB by summing the total contributions from each of the disk blocks

1. If a file size exceeds the maximum size of the file system there are a couple of ways to conquer this issue. One of those approaches would be to split the file into parts. This would be dividing the large file into smaller segments, each of those segments being stored as a separate file in the file system. The smaller segmented files could be logically linked or grouped to represent the big file all together. For example, if we needed to store a giant 256 TB file. We could divide the file into 4 smaller segments, 64 TB each. Then we separately store each of those segments. While storing these segments we keep some link between all of them for the final representation. Finally a control file will be needed which will contain the links / pointers to all 4 of the files.