Bilal Malik

104435995

Hasan Malik

104610462

Omar Sabbagh

104052125

COMP 3300

Operating Systems

March 20, 2020

Final Project

Project

I confirm that I will keep the content of this project confidential. I confirm that I have not received any unauthorized assistance in preparing for or writing this project. I acknowledge that a mark of 0 may be assigned for copied/plagiarized work. Bilal Malik 104435995 Hasan Malik 104610462 Omar Sabbagh 104052125.

Implementation

We begin by declaring all necessary variables. The Page table size is defined by pgTableSize with the value 256 given as instructed. The same is the case for the buffer size and the physical memory size. A structure is utilized, to allow for a more object-oriented approach to the TLB declaration. The tlb acts as a fast look up to pages before going to the second hand and larger page table. By defining it in a struct, we can initialize more than one, as well this allows for a dynamic approach to changing the values in the structure.

We will start in main and begin to trace the code. Again, we introduce the necessary local variables in main, including page faults and the hit rate. The memset() function is called twice and passed the necessary parameters, including whether it's a page index or tlb frame, the value to be set which is passed as int and then converted to char, and the size in bytes. Once this function is called twice, an 2D array of characters representing the physical memory is made. This array is 256 by 256 to represent the total size of 65536. The arguments list is checked to see if an argument was even passed on execution, then checked to see if this argument is an actual file in the working directory. Then the process of reading in the information from addresses.txt is begun. The while loop checks that there is something to be read, if this returns a value of 0 then we have reached the null terminating character in the file; else it will store the read in value into a variable holder and pass it to the findPage function defined earlier. Once the find page function has been called and terminated by integer value zero; we proceed to increment the input count. This value will be used alongside page faults and hits variable holders to find the page fault rate and tlb hit rate respectively. Page fault rate is simply computed by dividing the total number of page faults to occur when finding a page number, by the total attempts at finding a page number. The tlb hit rate occurs the same way, by dividing the total number of hits in searching for page numbers, by the total attempts to find page numbers. Both values are finally printed out before exiting the program.

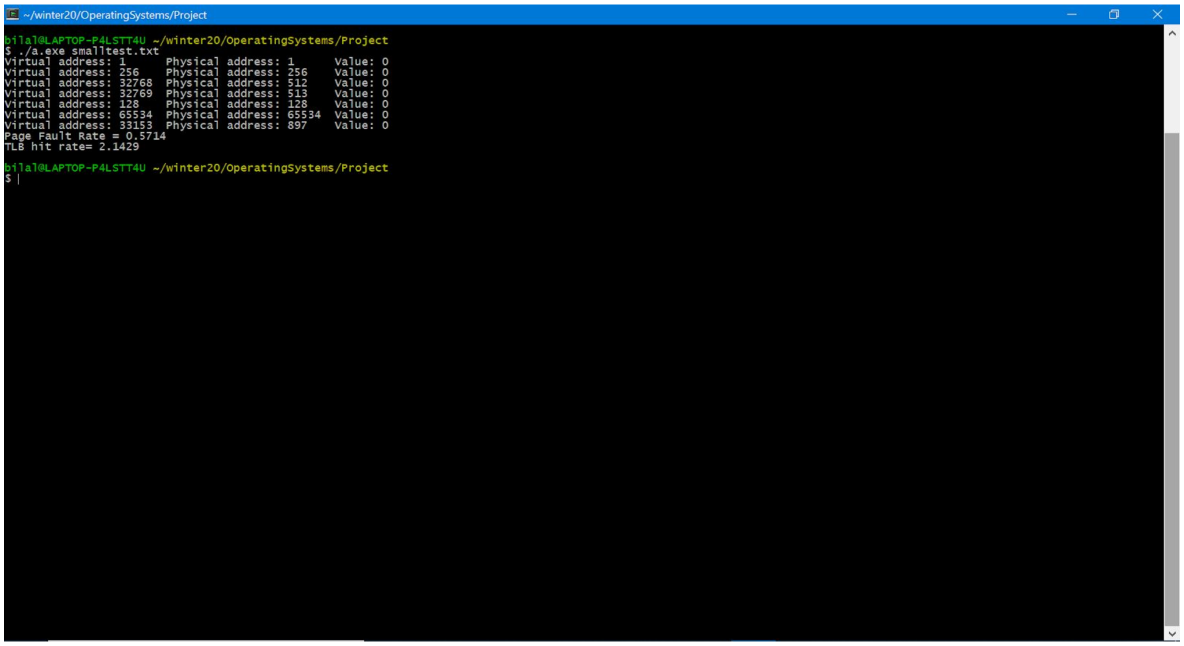
The findPage() function returns an int, and is defined with 7 parameters; the logical address, which is the value passed in from the while scan. A pointer to the value of the page table array, the structure that was instantiated in the main. A pointer to the physical memory array, the offset, the number of page faults, and finally the number of hits. The function again begins, by implementing its necessary variables. The virtual address is then printed, by taking the logical address, passed to the page find function. The page number is computed by utilizing arithmetical, specifically by shifting the logical address to the right 8 times. This is the same as dividing the logical address, by 256 (2^8) – the size of the logical address. Finally, the and bitwise operator is used together with this new shifted value and mask value 0xff (255), to gain the page number. The offset is computed by anding the two operands, without including the previous step of the shifting. Once we have the logical address from main memory, we must check to find these values in the tlb, a faster lookup key, normally placed with more frequent indexes. This translation lookaside buffer is a fast memory cache that needs to be used as much as possible to speed up the process, then the slower page table and significantly slower disk memory. We walk through the tlb looking to see if a match (tlb hit) is made to a physical address, between the page number and an index in the tlb. If this is the case, then too get the frame location from the index and to increment the hit count, for our later step of determining the hit rate. If the table hit never occurs, then we add the for through the larger page table looking for a match, if found we add the frame to our tlb for faster access when that input returns, if the tlb is full then we remove another frame to make room. If this last check fails, finally a page fault counter is incremented, for the later step of determining page fault rate. If a table hit does not occur, values of offset, physical memory and page number are passed to the last function readFromDisk(). Thus, we emulate a read from the lower level of memory, a large and heavy yet very slow, disk tape in our search for memory space; by calling this function. The value returned by readFromDisk is stored into the new frame variable holder in the find page function, and then saved into the page table at its pertaining index. After the page number has been found, whether in the tlb, page table or by making space through the backlog, we can now compute the location of the page number as an index. This is done though arithmetic; the physical memory size is multiplied by the frame before being added to the page offset. The value is the addition of both the physical memory and the index. These are then printed out to screen before, the program finally returns 0 to the earlier main function.

The read from disk function takes three parameters as input and returns an integer value. The three parameters are an integer for page number, a char pointer to physical memory, and a pointer to an integer representing offset. A variable block initializes the necessary variables, and the BACKING_STORE.bin file is open. The backing store is a non-volatile form of memory similar to the ROM memory. We use the .bin file as a symbolic form of the actual backing store. This backing store, sitting in the disk, is now accessed and a 256-byte page is taken from the file, and stored in an available place in physical memory. The backing file is checked to see if its successfully opened, and then seeked across until it gains enough page space, that is large enough to hold the physical memory size multiplied by the page number (256 * 256). It reads this page sized memory, and then copies the buffer into the new physical memory space. The offset is computed and returned to the find page function.

Two values you may need to know before we run the execution are the Translation Lookaside Buffer (TLB) and Page Fault. The Translation Lookaside Buffer is a high speed cache of the page table. It holds recently accessed virtual to physical translations. A perfect scenario would see every single hit occur in the quick accessing tlb. The TLB hit rate value is determined by the number of TLB hits divided by the total number of queries whether it's found in the tlb or page table or even faulted.

<u>Execution</u>

We begin the first execution, with a small sample input offered in the how to begin section of the project pdf. The values; 1, 256, 32768, 32769, 128, 65534, 33153 are put into a text file called 'smalltest.txt.' We run this by simply executing the executable, with an argument that is the name of the text file; in this case smalltest.txt. The output, is as follows:

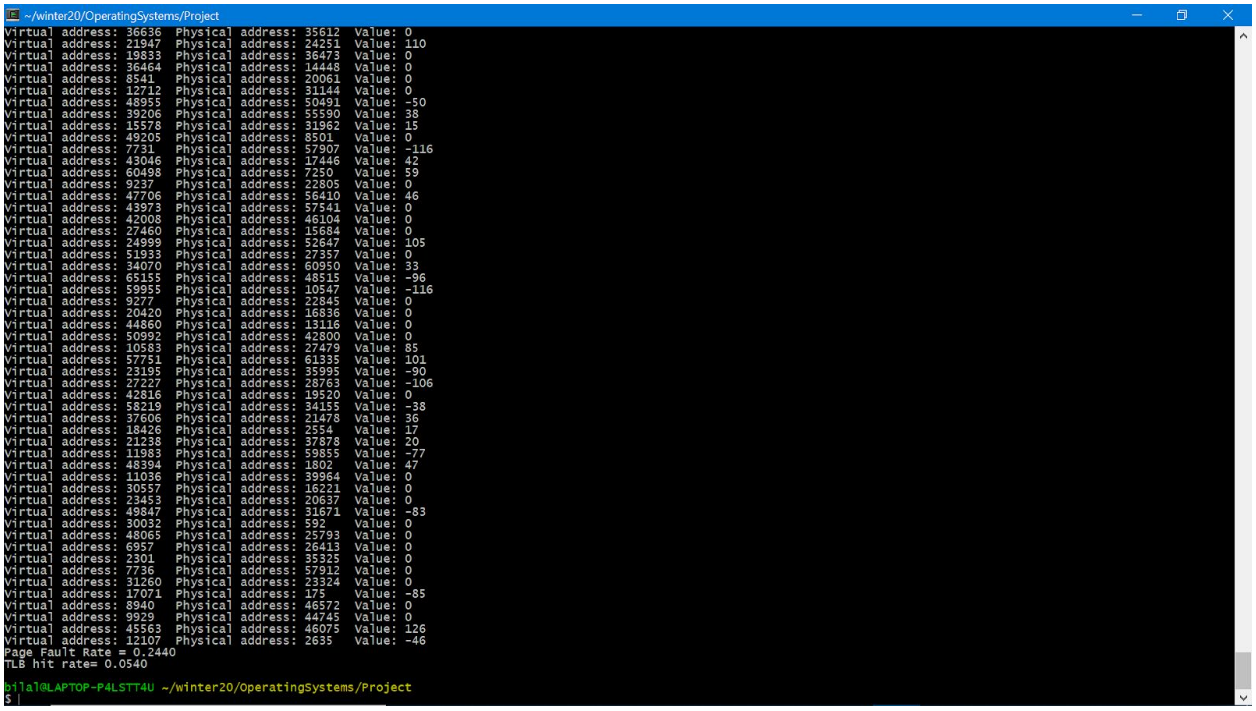


```
bilal@LAPTOP-P4LSTT4U ~/winter20/OperatingSystems/Project
$ ./a.exe smalltest.txt
Virtual address: 1      Physical address: 1      Value: 0
Virtual address: 256    Physical address: 256    Value: 0
Virtual address: 32768  Physical address: 512    Value: 0
Virtual address: 32769  Physical address: 513    Value: 0
Virtual address: 128    Physical address: 128    Value: 0
Virtual address: 65534  Physical address: 65534  Value: 0
Virtual address: 33153  Physical address: 897    Value: 0
Page Fault Rate = 0.5714
TLB hit rate= 2.1429

bilal@LAPTOP-P4LSTT4U ~/winter20/OperatingSystems/Project
$ |
```

The execution led to a 0.5714 page fault rate and the TLB hit rate was at a solid 2.1429. The TLB hit rate value is determined by the number of TLB hits divided by the total number of TLB queries.

Running the larger text file as argument leads to a comparatively different answer:



```
Virtual address: 36636  Physical address: 35612  Value: 0
Virtual address: 21947  Physical address: 24251  Value: 110
Virtual address: 19833  Physical address: 36473  Value: 0
Virtual address: 36464  Physical address: 14448  Value: 0
Virtual address: 8541   Physical address: 20061  Value: 0
Virtual address: 12712  Physical address: 31144  Value: 0
Virtual address: 48955  Physical address: 50491  Value: -50
Virtual address: 39206  Physical address: 55590  Value: 38
Virtual address: 15578  Physical address: 31962  Value: 15
Virtual address: 49205  Physical address: 8501   Value: 0
Virtual address: 7731   Physical address: 57907  Value: -116
Virtual address: 43046  Physical address: 17446  Value: 42
Virtual address: 60498  Physical address: 7250   Value: 59
Virtual address: 9237   Physical address: 22805  Value: 0
Virtual address: 47706  Physical address: 56410  Value: 46
Virtual address: 43973  Physical address: 57541  Value: 0
Virtual address: 42008  Physical address: 46104  Value: 0
Virtual address: 27460  Physical address: 15684  Value: 0
Virtual address: 24999  Physical address: 52647  Value: 105
Virtual address: 51933  Physical address: 27357  Value: 0
Virtual address: 34070  Physical address: 60950  Value: 33
Virtual address: 65155  Physical address: 48515  Value: -96
Virtual address: 59955  Physical address: 10547  Value: -116
Virtual address: 9277   Physical address: 22845  Value: 0
Virtual address: 20420  Physical address: 16836  Value: 0
Virtual address: 44860  Physical address: 13116  Value: 0
Virtual address: 50992  Physical address: 42800  Value: 0
Virtual address: 10583  Physical address: 27479  Value: 85
Virtual address: 57751  Physical address: 61335  Value: 101
Virtual address: 23195  Physical address: 35995  Value: -90
Virtual address: 27227  Physical address: 28763  Value: -106
Virtual address: 42816  Physical address: 19520  Value: 0
Virtual address: 58219  Physical address: 34155  Value: -38
Virtual address: 37606  Physical address: 21478  Value: 36
Virtual address: 18426  Physical address: 2554   Value: 17
Virtual address: 21238  Physical address: 37878  Value: 20
Virtual address: 11983  Physical address: 59855  Value: -77
Virtual address: 48394  Physical address: 1802   Value: 47
Virtual address: 11036  Physical address: 39964  Value: 0
Virtual address: 30557  Physical address: 16221  Value: 0
Virtual address: 23453  Physical address: 20637  Value: 0
Virtual address: 49847  Physical address: 31671  Value: -83
Virtual address: 30032  Physical address: 592    Value: 0
Virtual address: 48065  Physical address: 25793  Value: 0
Virtual address: 6957   Physical address: 26413  Value: 0
Virtual address: 2301   Physical address: 35325  Value: 0
Virtual address: 7736   Physical address: 57912  Value: 0
Virtual address: 31260  Physical address: 23324  Value: 0
Virtual address: 17071  Physical address: 175    Value: -85
Virtual address: 8940   Physical address: 46572  Value: 0
Virtual address: 9929   Physical address: 44745  Value: 0
Virtual address: 45563  Physical address: 46075  Value: 126
Virtual address: 12107  Physical address: 2635   Value: -46
Page Fault Rate = 0.2440
TLB hit rate= 0.0540

bilal@LAPTOP-P4LSTT4U ~/winter20/OperatingSystems/Project
$ |
```

Executing the much larger addresses.txt file, containing many more addresses. The page fault rate here is seen as 0.2440 and the transition lookaside buffers (TLB) hit rate are at abysmal 0.0540 hit rate.

Firstly, it needs to be pointed that as the number of inputs decrease - as is the case with this small text file, the fault rate will look that much larger, that is the same number of faults in a smaller page versus a larger will create a rate much larger in the smaller execution. The page fault rate here is 0.5714, more than half of the page numbers were not found either in the tlb, or the page table. This page fault rate, and its tlb hit rate counterpart will remain constant on runs of the same text file.

The **TLB hit rate** decreases on larger input. This is due to the fact that the **input entries are random**, thus placing a tlb miss in the tlb does not increase your chances of having a tlb hit on the next run. If i filled the tlb with a frame from a miss, that doesn't necessarily mean i would get a hit later, that input could never be seen or placed in a much later search. Yet the total number of tlb misses as possibilities increases. The number of inputs grows the tlb becomes full, less hits are found and more tlb misses become prevalent. The tlb needs to be continually refitted with the frame from failed runs, and this bogs down the search process. This is the case on page faults as well, the page fault service time that occurs after a page fault and hence after a TLB miss is a form of overhead - the swap in swap out process is one of the reasons, and so by every input the smaller function would do poorly in terms of time. Again the tlb hit rate formula is the number of hits divided by the total input count. Since the tlb is updated with random values, the success chance does not increase, actually decreasing with the larger input. Thus, the rate is smaller because of the larger input count.

For this reason, the page table and tlb fail, resulting in a high fault rate. As the input increases, the fault rate will decrease. More and more inputs can be found in the page table, and less are being caught in the smaller tlb then the previous run. Noticing that the tlb hit rate decreases, and the overall page fault rate decreases as well shows the page table is working well and gaining a lot of hits. The page table is effective against these large inputs contrasting in the earlier smaller execution, the values are new and not in either the tlb or the page table so in this smaller execution the **page fault rate is higher**. Misses occuring in the page table, resulting in page faults. In the larger inputs, the page table has matured more and is capable of successfully taking in new values. Only a small portion of memory from the process is stored in the page table allowing for many processes to be stored at a time in the table. This advantage allows us to cast a wider net, virtual; memory allows more processes to be in main memory then there is even space for, because of this, more hits occur in the page table.

In conclusion, we notice that with a larger input the tlb hit rate falters (the tlb fails to work), while the page faults rate decreases to a success (less page faults are occuring) which shows an increase in page table hits. The tlb is a smaller form of cache memory and only stores 15 values at a time, so as the input grows, this tlb size stays static and only catches the same amount it would on any sized input, thus as it grows, the tlb is less likely to have the same effect. In the second run, the effective memory access time (emat) is much slower per frame found, because of the use of the heavier and slower page table, and the rare use of the tlb. In the smaller run, the frames emat is much faster due to successful tlb hits. As well, when the tlb is full the continually removal of frames for missed frames bogs down the process. Now, we can see the change in terms of hits and time it takes, in both runs and understand why these numbers are the way they are.