COMP 304 - Project 3

Kerem Aksoy - 64243
Demet Tümkaya - 72210

“Virtual Memory Manager”

PART 1 **2**
Page Masking 2
Reaching The Page and Loading A Needed Page 2
Implementation of TLB 3
PART 2 **3**
FIFO Implementation 3
LRU Implementation 4
Screenshots from Different Runs **4**
No page replacement 4
FIFO Implementation 4
LRU Implementation 4

PART 1

In this part, we are implementing the virtual memory manager without any replacement policy since the virtual address space and the physical address space of the program have the same size which is 1024 pages (named “page” in virtual space, “frame” in physical space).

Page Masking

First, we needed to do masking to have offset and logical page number from a logical address. Since the last ten bits are used for offset, we can simply use a mask,which has ones on the last ten bits and zeros on other bits, and do a “logical and” with the logical address. This will produce the offset. To have the logical page number, we first need to shift to right by 10 to get rid of the offset bits. Then, we simply do an “logical and” operation between the logical address and the page mask which have 1s on the last 10 bits.

Reaching The Page and Loading A Needed Page

After having the logical page number and offset, it is time to translate the logical address to its corresponding physical address. First we check the tlb table whether the address is there or not, then if not we need to have it from the page table. If it is also not in the page table, we need to take it from the backing store. Since we have the page size and logical page number of the needed page, we simply copy it to a corresponding physical page.To have the physical page, since the virtual space and the physical space have the same size, we can implement this part by just holding a variable (called free_page) which indicates where the next page will be added in the physical memory.

Implementation of TLB

In memory management, the reason behind implementing a TLB is similar to the idea called “cashing”. When we try to reach a part of a memory, we first need to go to the page table,where logical pages and their corresponding physical page’s addresses lie, and search for the page in the table to have the physical address. This is an overhead and it slows down the performance of the system. So, we can get rid of this overhead by implementing a TLB.

In this part, we are required to use fifo policy in the implementation tlb. In fifo policy (first in first out), we need to replace the one which is added earliest into the table when our table is full. To implement this approach, we can use a circular queue. To implement this queue, we need a global variable which will hold the information about where to add (which one to be replaced) the new tlb entry. This whole work is done inside add_to_tlb function.

More detailed explanation has been done in the code as comments.

PART 2

In this part, we need to consider two replacement policies, since the virtual address space and the physical address space of the program haven't the same size, as LRU and FIFO. We have 256 frames and 1024 pages. Therefore, we created a count table to be able to perform page replacements.

We used a replacement boolean to see if we are at the end of the memory, when the nextPlace is equal to 255, we need to use respective replacement policies. For FIFO, we will replace the new page with the first page in the current condition and for LRU, the replaced page will be the recently used one according to the count table.

FIFO Implementation

The boolean replacement will be initialized as false and the same procedures with the first part are applied until we reach the end of the page table. As it is explained above, when nextPlace equals to 255, replacement becomes true and updateTables() function will be used for next coming pages.

updateTables() gets the address to be replaced and updates that logical address in the page table and physical address in the TLB as -1 as a part of page replacement.

LRU Implementation

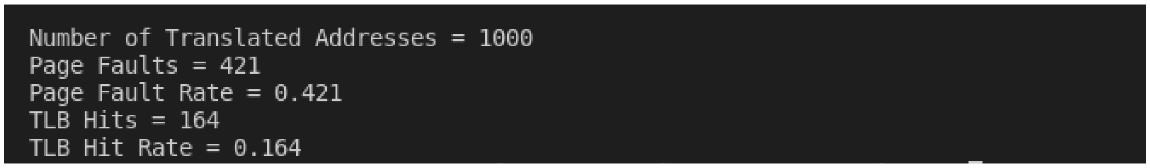
For LRU policy, when replacement boolean becomes true, it calls the findLeastRecent() function to find the page to be replaced and it returns an index as least_logical. Then, the updateTablesLRU() replaces the logical pages.

findLeastRecent() uses the count table to find the page that is the most previously reached one by finding the minimum count from that table.

updateTablesLRU() changes the information about replaced pages by making their physical and logical addresses -1 as it is done in FIFO, also the count of that index becomes -1 since it is not there anymore.

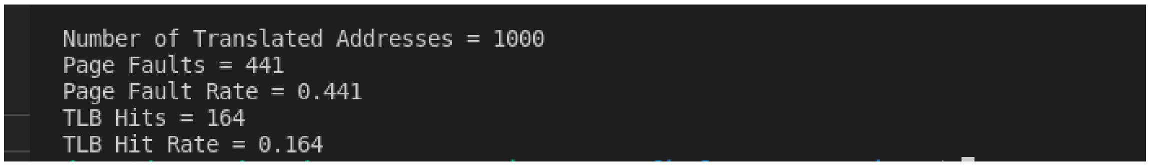
Screenshots from Different Runs

No page replacement



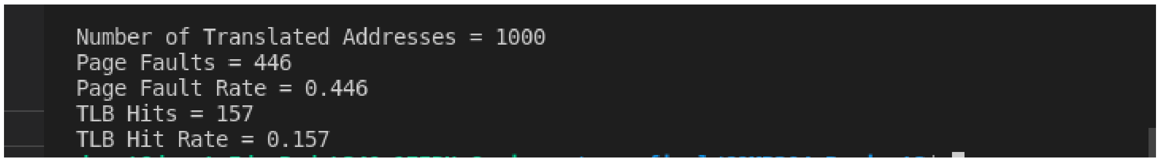
```
Number of Translated Addresses = 1000
Page Faults = 421
Page Fault Rate = 0.421
TLB Hits = 164
TLB Hit Rate = 0.164
```

FIFO Implementation



```
Number of Translated Addresses = 1000
Page Faults = 441
Page Fault Rate = 0.441
TLB Hits = 164
TLB Hit Rate = 0.164
```

LRU Implementation



```
Number of Translated Addresses = 1000
Page Faults = 446
Page Fault Rate = 0.446
TLB Hits = 157
TLB Hit Rate = 0.157
```