IT308 Operating Systems
Project: Simulating Virtual Memory Management.

Aditya Joglekar (ID:201401086)* and Rajdeep Pinge (ID:201401103)†

Virtual Memory is an essential part of memory management system. Hence it is important to understand the complex internal working of the virtual memory. We have used Basic code given by University of Notre-Dame [1], [2] for simulating virtual memory in the user environment. This has allowed us to work with high level programming language rather than working at the kernel level. We have implemented page fault handling in an incremental manner. Further we have implemented three page replacement algorithms. Finally, we have compared the performance of the page replacement algorithms to get a better understanding of the working of system.

INTRODUCTION

A computer programmer assumes that their program will have access to the full physical memory (this of course will not happen). Thus, the programs which have to run on the system will almost certainly end up being much larger than what the RAM can hold at a time. Virtual memory allows us to run programmes much larger than the size of physical memory.

Thus, in a Virtual memory system only parts of a code are available in RAM (Main Memory) at any point of time with the rest stored in the non-volatile secondary storage or disk. The locality of references ensures that this is a reasonable thing to do. Most of the time, the things which are required are present in the physical memory. The programs are thus divided into chunks called 'pages'. Once in a while however, a page which is requested is not available and thus has to be fetched from the secondary memory. This is called a 'page fault'.

So, Operating Systems effectively give the programs an expanded address space to work in with the OS having to take the responsibility of handling the added complexity involved. This is accomplished using data structures called page tables which map the virtual addresses to their actual physical storages (page frames).

Background, importance and relevance

Computer memory is one of the most important building blocks of any computer system. It is however a limited resource. Though nowadays a lot more memory is available to programmers, ideally one would want an unlimited amount of memory for one's programs.

Virtual Memory is a brilliant idea which, even if is not able to give us an unlimited amount of memory, enables us to run programs which are far larger in size than main memory.

The RAM which is where things actually run is called the 'physical' memory. But since RAM is volatile, computer systems need to have a secondary non-volatile storage too along with a RAM. Even though way slower than the main memory, the secondary memory or disk has a far greater storage capacity. The Virtual Memory concept exploits this difference.

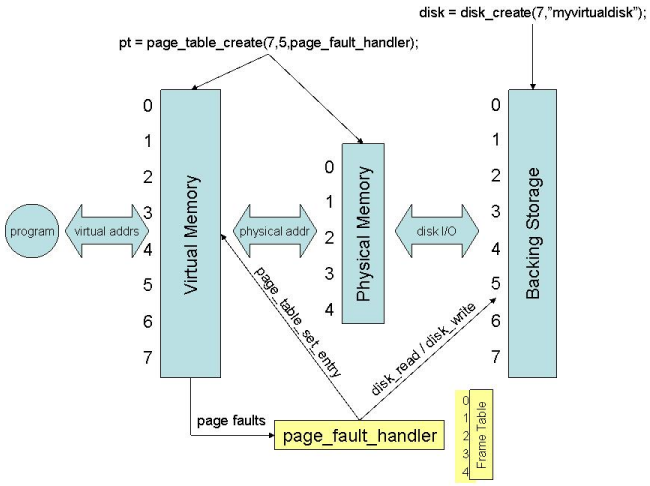

FIG. 1. Virtual Memory System

Problem Description

The aim of this project is to dig underneath the OS's hood and take over the responsibility of handling the paging ourselves.

Fig.1 shows a typical VM system. [1]

However we will only be simulating what the OS does. This is because actually when we run our simulation experiments, they themselves will be making use of the OS's actual paging system. Also, simulation can be done in the user environment rather than kernel mode which allows us to work with high level programmes and understand the concept easily rather than getting into the complexities of kernel level programming.

We will be simulating the physical and virtual memories as files stored on the main memory. This means that what we will assume to be our secondary memory will actually be a file in main memory. This will allow us to

separate ourselves from the OS's underlying paging system which is always running in the background and focus our attention to the blocks of memory we have defined. We give the details of this indirect approach in the next section.

Our main aim is to catch the page faults generated by our programme find out the reason for the page faults and then accordingly take the necessary action to solve the page fault.

PROCEDURE, METHODS USED

As mentioned above we first of all allocate files which we call as our simulated 'virtual' and 'physical' memories. As far as we are concerned, these will be the memories available to us.

When we want to run a program we will assign its code to the simulated virtual memory. Then we will create a mapping between the simulated virtual and physical memory.

We maintain a special C data structure called page table to keep track of the mappings between the two sets of pages. It also contains the permission bits [Read(R), Write(W), Execute(E)] assigned to each virtual page.

We set up a page fault handler to trap the page faults. The page fault can occur due to two reasons

1. A page requests a procedure for which it does not have the required permission. for e.g a page with read permission attempts to write. We handle these faults by making suitable changes in the permission bits.

2. The more important cause of a page fault is when the page is not present in the physical memory s(in our case the simulated memory). We then have to bring in the page from the secondary memory.

The first part of the project is to create an artificial mechanism using which the Virtual Memory system could be simulated in a simple way. We need to be able to catch the page faults and determine the cause of page faults. This requires an incremental approach of solving. We have used the following method to implementation:

1. In first version, the programme simply terminates when a page fault occurs. This is not useful at all.

2. In second version, we handle the case when number of virtual pages = number of physical pages. This will result in no page faults in the future if we map each page to its corresponding frame the first time it gives a page fault on that page. Here we must note that whenever a new page is brought into main memory, it is restricted by giving it only the Read (R) access.

3. In third version, we handle the pages faults occurring due to the restricted access. We handle the cases where a page has only read access but the programme wants to write on that page and cannot do so because of the protection bits given to that page. Here the simple solution is to modify protection bits to give Write (W) access to the programme.

   This is further useful during page replacement because initially all pages have only Read access and if a page has Write access then that page must have been modified. This means that before replacement, such pages must be written back to disk so as to avoid loss of data. Hence, Write bit (W) also acts as Dirty bit indicating which pages have been modified.

4. In fourth and final version of the page fault handler, we implement the mechanism whereby if a page is not in the page table then we first need to check if there is an unoccupied frame in the page table.

   We keep track of frame occupancy for quickly detecting an empty frame. If a frame is empty, the page is read in from the secondary memory and the page map table is modified.

   If no frame is empty then some page has to be evicted for bringing in the new page. The question of which page to remove is important and several algorithms known as page replacement algorithms are available which try to minimize the possibility that an evicted page is needed again soon. A good page replacement algorithm will lead to lesser page faults in future, less overhead and thus better performance. We call the necessary page replacement algorithm to replace appropriate page.

The second part of the project is where we implement three page replacement algorithms(PRA) and compare their performances. The performance is measured in terms of number of page fault that a PRA causes and the number of disk reads and disk writes it does to service such page faults.

Page replacement algorithms

We implemented the following page replacement algorithms:

- **Random page replacement-** This is one of the simplest algorithms available. This follows the principle, "if you don't have much idea about a process just do things randomly and hope that the problem[here page fault] will occur rarely" i.e. if a page fault occurs and a page is to be replaced, then randomly select a page (frame) from the page

table and replace it. Frankly speaking this is not a very sophisticated algorithm. It might throw out the most heavily used page too. But we hope that if the number of pages is pages large enough, then it will perform okay on an average. Though it will certainly not be optimal.

- **First in first out (FIFO)**[3]**-** this is the most simple minded algorithm. It assumes that what is the oldest page must be the one which is not used recently. Hence it kicks out the page which came first i.e. the oldest page from the page table. However, this approach may not be correct at all because an old page may also be the one which is used frequently. Therefore, if we replace that page, then it will make the page management inefficient. Thus, we expect this algorithm to perform poorly.

- **Least Recently Used (LRU)**[3]**-** This algorithm also considers the time of page usage. That is, whenever a page is accessed, it is marked as most recently used. This is based on the locality of reference in time - if a page is referenced now then there is more probability of it being referenced again in the near future rather that a page which was referenced in the past.

  We made some major changes in the basic code to implement LRU. This is because the code was mainly for allowing us to implement basic PRAs which would work during the page faults. But LRU requires updating the page list whenever the page is accessed, not just when there is a page fault.

  1. We have used the linked list approach to implement LRU wherein each page is a node in the page list. Whenever a page fault occurs, if the page table has an empty frame then a new node for that page is added in the linked list.
  2. If we are required to replace a page, then the page at the head of the list is the one which is least recently used and is replaced. The new page is stored at the tail of the list indicating that it is the most recently used.
  3. Finally if no fault occurs, i.e if the page is already in the linked list, then it is found out and shifted to the tail of the list again indicating that it is the most recently used.

RESULTS

We have tried to compare the page replacement algorithms using some standard programmes which access the memory in a particular pattern. There are many parameters which play a role in the results like the number of virtual pages and physical page frames available, page replacement algorithm used and the programme used for simulation.

1. **Focus Test Programme:** In this programme. First the memory is accessed for writing in sequence. Then there is randomized read access.

Number of Pages = 12
Number of Frames = 7
Test Programme : Focus

	Random	FIFO	LRU
Page Faults	1245	1196	1186
Disk Reads	627	603	597
Disk Writes	616	593	586

FIG. 2. Focus programme test 1

Number of Pages = 112
Number of Frames = 7
Test Programme : Focus

	Random	FIFO	LRU
Page Faults	2994	2876	2869
Disk Reads	1553	1494	1490
Disk Writes	1441	1382	1378

FIG. 3. Focus programme test 2

In the above three tables FIG.2, FIG.3, FIG.4, we observe that the random Page Replacement Algorithm consistently performs poorly while LRU is the best.

For lesser number of page frames, the difference between Page Faults of LRU and FIFO is quite small but when we increase the number of available page frames the difference increases. This may be because as number of frames increase, there is more scope for LRU page replacement to be different from FIFO page replacement. Hence as per theory, LRU performs better.

Number of Pages = 112
Number of Frames = 37
Test Programme : focus

	Random	FIFO	LRU
Page Faults	2210	2166	2138
Disk Reads	1155	1135	1122
Disk Writes	1055	1031	1016

FIG. 4. Focus programme test 3

2. **Scan Test Programme:** In this programme. First the memory is accessed for writing in sequence. Then memory is accessed for reading again in sequence.

Number of Pages = 12
Number of Frames = 7
Test Programme : scan

	Random	FIFO	LRU
Page Faults	67	84	77
Disk Reads	55	72	65
Disk Writes	12	12	12

FIG. 5. Scan programme test 1

This is a simpler programme with less memory access and sequential access. Therefore overall number of page faults are much less as compared to previous programme. (See FIG.5, FIG.6)

FIFO is still worse than LRU. But LRU is not the best. This may be because since the memory access is sequential therefore there is not much advantage of LRU. And hence random works better because it does not have any specific constraints while its disadvantage is removed because of sequential access and locality.

Number of Pages = 22
Number of Frames = 17
Test Programme : scan

	Random	FIFO	LRU
Page Faults	93	154	113
Disk Reads	71	132	91
Disk Writes	21	22	22

FIG. 6. Scan programme test 2

CONCLUSION

Through this project we have been able to understand the role and working of the virtual memory in memory management system of the OS. This simulation has allowed us to understand the memory mapping between the virtual and the physical memory environments of the system. It has also allowed us to identify different reasons for page faults and how to handle them. Finally we were able to successfully implement three of the page replacement algorithms and compare their performances to some extent. In results, we conclude that LRU always performs either equally well or even better than FIFO.

CONTRIBUTION

Both of us had to understand the basic code along with the basic concepts of virtual memory[3].

Aditya Joglekar, 201401086

- Implementation of page fault handling and page map table management.
- Implementation of Random page replacement algorithm.

Rajdeep Pinge 201401103

- Implementation of FIFO page replacement algorthm
- Extending the code to detect page access. Using this ability to implement LRU algorithm.

Finally, we had to compare and analyze the performance of the page replacement algorithms. This was more difficult than we expected because we had to find

a page access pattern which will be fair to all algorithms and allow proper analysis between their performance.

POSSIBLE EXTENSIONS

The page table does not contain R(Recently Referenced) and M(Modified) bits. These can help us implement other page replacement algorithms like Not Recently Used (NRU), FIFO with second chance and working-set algorithm. The modified bit will help us distinguish between pages which have been modified and the once which have not. This will help us save the cost of writing back the page contents to secondary storage while evicting it.

We have noticed that the performance of the algorithm is heavily dependent on the ratio of pages frames to pages, the page replacement algorithm used and the page access pattern of the running program.

We have given a limited analysis for a particular set of the above parameters and for a specific random sequence of accesses. Thus, we need to perform the analysis a number of times and report the average performance just to ensure that our results are not obtained by chance.

* 201401086@daiict.ac.in
† 201401103@daiict.ac.in

[1] **Project guide** on Course Website of Notre Dame University www3.nd.edu/ dthain/courses/cse30341/spring2017/project5/
[2] **Base source code** www3.nd.edu/~dthain/courses/cse30341/spring2017/project5/
[3] Andrew S. Tanenbaum, Modern Operating Systems, Second Edition, Chapter 4