Virtual Memory Management Simulator

Pratham Nayak - 191IT241
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
pratham.191it241@nitk.edu.in

Aprameya Dash - 191IT209
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
aprameyadash.191it209@nitk.edu.in

Suyash Chintawar - 191IT109
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
suyash.191it109@nitk.edu.in

***Abstract*—Virtual memory serves as an essential part of computer architectures in the modern world. As computer memory, in reality, is expensive, the need for virtual memory arises, which helps to create an illusion of the presence of a very large main memory. The project here aims to implement a virtual memory manager that will facilitate the simulation of virtual memory in a way that is feasible and is easily understandable to the consumer. The objective is to experiment and analyze the different combinations of paging schemes and replacement algorithms and to get the best possible combination, and plot the obtained results.**

Keywords— Virtual Memory, paging schemes, fetch policies, replacement algorithms, Tkinter

I. Introduction

When a process that is to be executed is very large in terms of space, it is not possible to run the process normally as there is a possibility that the amount of space necessary for the process to execute is greater than the available physical memory. To overcome this difficulty, the concept of virtual memory is used. The virtual memory space is divided into smaller blocks, and the memory blocks that are required by the process that is currently under execution are stored in the main memory. All the other memory blocks are stored in the secondary memory of the computer. Hence, this creates the perception of a large memory even though the size of the physical memory is fixed.

This proposed system used here deals with the creation of a Virtual Memory Management Simulator by utilizing the various concepts of virtual memory such as different fetch policies, page replacement algorithms, and page size. This simulation will help to identify the most efficient combination of replacement algorithms, fetch policy, and page size for any given list of processes.

The paper is broken down into various sections, as explained here. Section 2 deals with the studies carried out related to the use of virtual memory. Section 3 states the exact problem statement and the objective of our work. Section 4 is the proposed work and the methodology of our virtual memory management simulator. Section 5 contains the detailed analysis of the results that were obtained during experimentation with different input parameters.

II. Literature Survey

[1] In this paper, the virtual memory management system is simulated, and its properties were verified with MSVL, which is a parallel programming language. MSVL stands for Modeling, Simulation, and Verification Language, and it is used as a formalism for verification and specification of concurrent systems. They tried to simulate virtual memory management of the operating system when two different programs are running simultaneously in the processor. This system manages the virtual memory by handling the pages and uses the aging algorithm to find a page to swap out when a page fault occurs, and there is no idle page.

[2] In this study, the MOSS simulator has been modified, and a virtual memory simulator has been developed that allows the user to switch between different page replacement algorithms and analyze and compare the number of page faults that occur in each of the algorithms. The objective of this research was to compare the goodness of the page replacement algorithms. The results in the above model showed that the optimal algorithm being the best of all the three replacement algorithms gives the least amount of page faults, and in most cases, LRU is better than FIFO. When the memory access patterns are completely random, then, in that case, LRU and FIFO work almost similarly was stated in the paper.

III. Problem Statement

Building a virtual memory management simulator to find out the most efficient combination of fetch policy, replacement algorithm, and page size for a given list of processes.

IV. Proposed Work

A. *Backend*

The backend for Virtual Memory Management Simulator involves accepting all input parameters and using the input

parameters for running the simulation and passing all the generated data and statistics to the frontend, where results will be displayed to the end-user. The Backend for the virtual memory management simulator has been implemented using the C++ programming language.

1) ***Input for running simulation****:* All parameters necessary for running a simulation have to be accepted from the front end. The input parameters contain the path to the process list file, the path to the process trace file, the page fetch policy, the page replacement policy, and the page size.

- Process list and Process trace file: The process list file is a text file describing each process involved in the simulation. The process trace file contains a series of memory requests to simulate memory usage of an actual system. More description about these files is provided in Section V.
- Page fetch policy: The virtual memory management simulator supports both demand paging and prepaging.
- Page replacement policy: The virtual memory management simulator supports FIFO, LRU, and clock replacement policies.
- Page size: The virtual memory management simulator allows page sizes of 1, 2, 4, 8, 16, and 32.

2) ***Creation of classes****:* Various classes have been created to imitate the actual processes of paging and replacement.

- To simulate individual pages in virtual memory, a class has been created which stores the process id of the process to which the page is associated, the page number, the time at which the page was added to main memory, and the last accessed time of the page.
- Similarly, a class has been created to store details of each process. This class contains the process id of the concerned process, the total memory allocated to that process, a list of all pages, and a page table that can be used to know if a particular page is present in the main memory or not.
- Further, another class has been created to represent each entry in the page table of a process. Each entry of a page table, associated with a particular process, contains the page number of a page associated with the concerned process, the frame number in which the page resides, and a valid bit that indicated whether the page is present in the main memory or not.
- Furthermore, each frame is represented by a class which contains the page number of the page which resides in the frame, the process id of the page, an 'occupied' flag bit which indicates whether the frame is occupied or not, and a 'used' bit which will be utilized during the implementation of Clock replacement policy.
- Finally, a class has been created to store the process id and the requested memory location of each memory request. The data read from the process trace file will be stored in this format.

3) ***Initializing simulation****:* The size of the main memory has been fixed at 512 memory locations for running any simulation. During the initialization of simulation, various structures have to be created to properly imitate the virtualization process. A list of processes is created by using the class, which represents the structure of each process. All data read from the process list file is stored in this created list. Further, a queue is created to represent all memory requests using the corresponding class. This queue stores all the data read from the process trace file. To imitate the main memory, a list of frames is created based on the class created for representing each frame. Initially, each frame of the main memory is assumed to be unoccupied, and data for all the structures are initialized using this assumption. Next up, the actual simulation is started. During the simulation, the memory request at the front of the queue containing all the memory requests is processed, and this is repeated till the entire request queue becomes empty. For each memory request, the selected paging and replacement policies are applied to check whether the page is already present in the main memory or not. If the page is not available in the main memory, then a page fault occurs, and the page is brought into the main memory using the selected replacement policy.

4) ***Implementing paging policy****:* Paging is a memory management strategy that helps us to discard the necessity of contiguous memory allocation of main memory. In Operating Systems, the concept of virtual memory is used to create an illusion of a large physical memory while executing numerous processes at once. It helps in extending the use of the main memory and also offers security as there is a mapping between the physical address and virtual address of a memory location. Paging mechanisms are used for the division of the main memory as well as the virtual memory in fixed-size blocks. These fixed-sized blocks are known as frames in physical memory, i.e., main memory and pages in virtual memory. Hence the size of a page and a frame are equal.

In the execution of a program, alternatively called a process, it requests a memory location required by it to complete its routine. Fetch Policies do the work of loading these requested pages into the virtual memory from the physical memory. Two types of fetch policies have been used in the project:

- DEMAND paging - It is a virtual memory management strategy that copies a page into the physical memory only when an attempt is made to access that particular page or memory location of a process. It is a classic example of lazy loading technique as only those pages are loaded in the main memory from the secondary memory that is demanded by the program. In case of demand paging, during the simulation, the program first checks if the page containing the requested memory location is currently present in the main memory or not. If the page is not present in the main memory, then only that particular page is brought to the main memory using a replacement policy.

- PRE paging - In this paging scheme, the pages which are not requested by the process are also brought into the main memory. This is an extension of demand paging wherein the operating system guesses which page the process may require after loading the demanded page and pre-loads them into the main memory. This is implemented by choosing the page of a process that is not present in the main memory other than the demanded page. In the case of prepaging, every time a memory location is requested, first, the program ensures that the page containing the requested memory location is present inside the main memory, and after that, the next contiguous page is also loaded into the main memory.

5) ***Implementation of replacement policy****:* When a process attempts to access a page that is not loaded in the main memory, a page fault occurs. It happens when either the main memory is empty or full. If the main memory is full, some existing pages have to be replaced from the main memory to access the demanded page, not in physical memory. This is when replacement algorithms come into the picture. When a new page is to be loaded into the memory, the algorithm that is used to find the page which is replaced by the new page is called the page replacement algorithm. There are several types of page replacement algorithms such as FIFO, LRU, Clock, Optimal, etc. In this simulation, we implement 3-page replacement algorithms, namely FIFO, Clock, and LRU.

- FIFO - In this page replacement algorithm, the page which is the oldest of all pages is replaced by the new page. This is implemented by keeping track of the time at which each page was loaded into the memory.
- LRU - In this page replacement algorithm, the page which is used least recently is replaced by the new page. This is implemented by keeping track of the time at which a specific page was accessed most recently.
- Clock - In this page replacement algorithm, all the pages are considered one by one in a round-robin format. All the pages have their 'used' bit set to 1 initially, and a page with the bit as 1 is given a second chance and is not replaced when it is considered as a candidate for replacement, rather its bit is set to 0, whereas pages with bits already set to 0 are replaced with the new page.

6) ***Returning the derived data****:* After completion of the simulation, all the derived data is passed to the front end. The number of processes involved in the simulation, the number of memory requests read from the process trace file (stored in the memory request queue), and the total number of page faults encountered are all passed to the frontend.

B. ***Frontend***

The front end of the virtual memory management simulator deals with taking all input parameters from the user through a GUI (graphical user interface), creating plots to present the data received after the simulation and then, displaying all the final results to the user through a GUI. The frontend part has been implemented using the python programming language. Tkinter library has been used to create the GUI, and Matplotlib has been used to create all the plots for presentation.

1) ***Taking input from user through GUI****:* For easier access to the entire simulation of the project, a Graphical User Interface is created using the Tkinter module in python. The paths of the process list and process trace files are taken as input from the user, and the file paths are validated. The page size, fetch policy, and the replacement algorithm are accepted as input from the user. Then a sub-process call to the backend is made to get the number of page faults for the inputs given by the user.

2) ***Creation of plots****:* The data received after completion of the simulation is used to create presentable plots. This is done using the matplotlib plotting library. A bar graph displaying the number of page faults for various page sizes and another bar graph displaying the number of page faults for various combinations of page fetch policy and page replacement policy is created and stored.

3) ***Displaying the results through a GUI****:* Finally, all the collected data is represented using a GUI. Various details like the number of processes involved, the total number of memory requests involved, the page fetch and replacement policies selected by the user, and the total number of page faults are displayed through the GUI. Next, all the created plots are displayed through the GUI. This way, all generated data along with created plots are displayed to the user.

V. Dataset Description

This section describes the dataset that is used for running the virtual memory management simulator to generate the final results, which are displayed later in this paper. The following two text files are required for initializing a simulation:

A. Process List file

It is a text file that contains the list of processes involved in the simulation, along with the size of memory required by each of the processes. Each line contains two space-separated integers, first, denoting the process ID, which is a zero-based indexed, and second is the number of memory locations required to store the complete program. Hence, the project implements virtual memory management for multiple processes. The results displayed in this paper have been generated using a process list file containing a total of 10 processes. For each process, the process ID and number of memory locations are mentioned.

B. Process trace file

It is a text file that contains an ordered list of memory locations that are to be executed by the processes. This series of memory location requests simulates the memory usage of real systems. Each line contains two space-separated integers, the first denotes the process ID, and the second integer denotes the memory location (one-based) of the instruction which is to be executed by the process. The process trace file used for generating the results in this paper contains a total of 11,00,010 memory requests.

VI. Experimentation and Results

This section describes all the trends and patterns observed after running the proposed virtual memory management simulator using different combinations of page fetch policies, replacement policies, and page sizes. These trends and patterns were collected by carefully analyzing the statistics and plots, which were generated whenever the simulator is initialized using a certain combination of page fetch policy, replacement policy, and page size.

The below figures represent the number of page faults recorded for various combinations of fetch policies and replacement algorithms with varying page sizes. Fig. 1, 2 & 3 show how the number of page faults vary when demand paging is used as the fetch policy. Similarly, Fig. 4, 5 & 6, represent how the number of page faults vary when pre-paging fetch policy is used. Fig. 1 & 4 uses FIFO page replacement algorithm, Fig. 2 & 5 uses LRU replacement algorithm, while Fig. 3 & 6 uses Clock replacement algorithm.

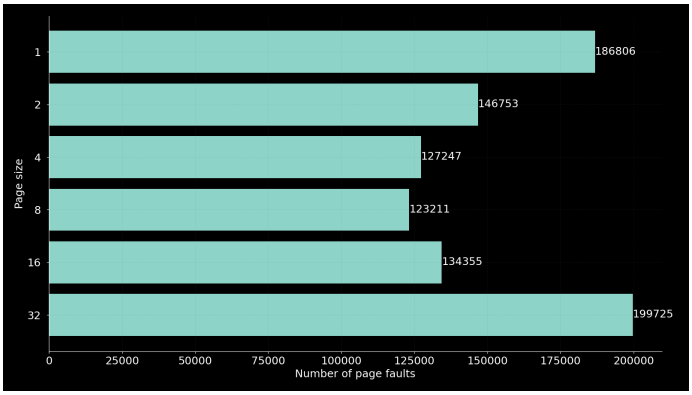

Fig. 1: Page faults vs Page size - Demand + Fifo

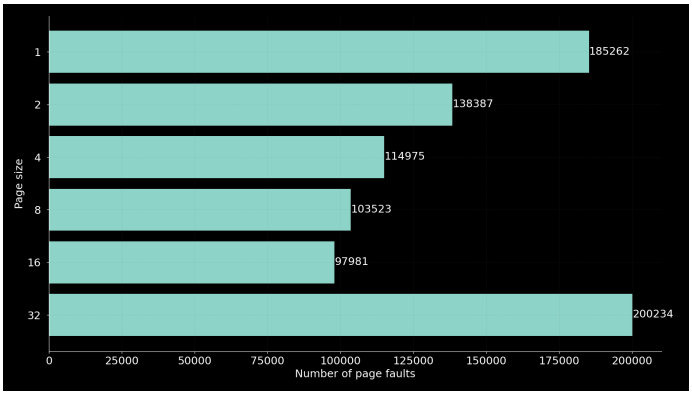

Fig. 2: Page faults vs Page size - Demand + Lru

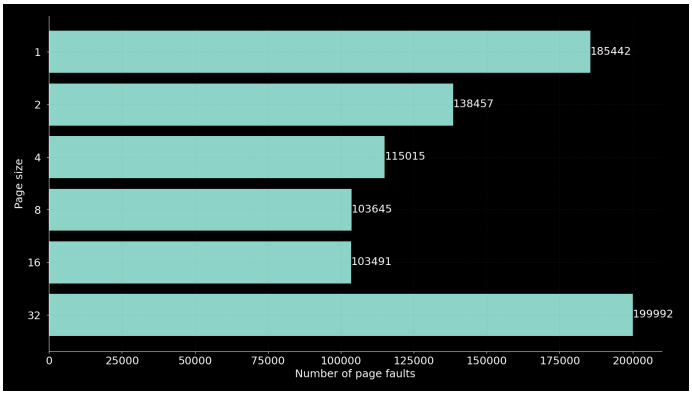

Fig. 3: Page faults vs Page size - Demand + Clock

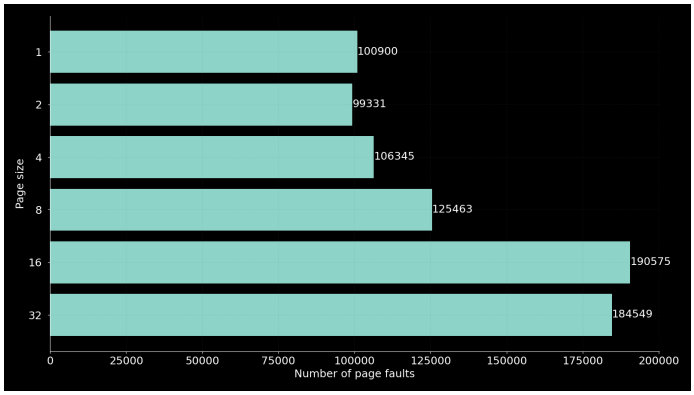

Fig. 4: Page faults vs Page size - Pre + Fifo

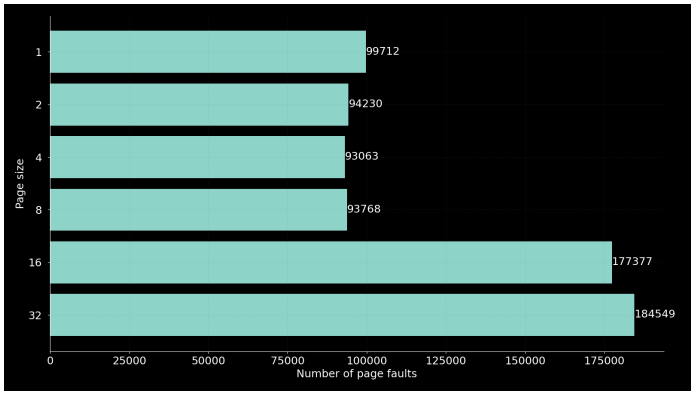

Fig. 5: Page faults vs Page size - Pre + Lru

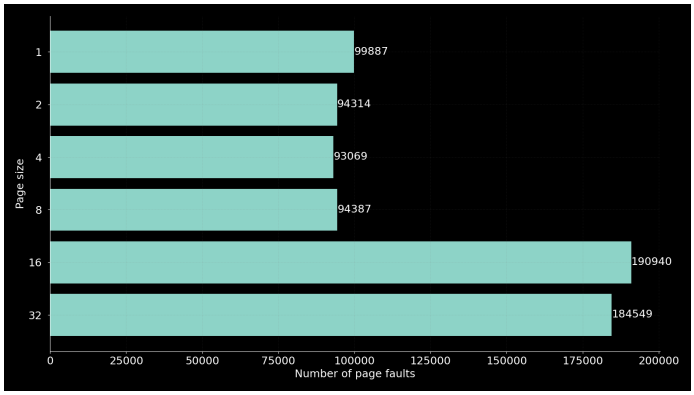

Fig. 6: Page faults vs Page size - Pre + Clock

TABLE I: Number of Page Faults in case of Demand Paging

Page Size	FIFO	LRU	Clock
1	186806	185262	185442
2	146753	138387	138457
4	127247	114975	115015
8	123211	103523	103645
16	134355	97981	103491
32	199725	200234	199992

TABLE II: Number of Page Faults in case of Pre-paging

Page Size	FIFO	LRU	Clock
1	100900	99712	99887
2	99331	94230	94314
4	106345	93063	93069
8	125463	93768	94387
16	190575	177377	190940
32	184549	184549	184549

Table I shows the number of page faults recorded for different combinations of page sizes and page replacement policies when demand paging is used, whereas Table II shows the number of page faults in case of pre-paging. The following trends were observed about the performance of each combination of page fetch policy, replacement algorithm, and page size:

- When the page size is extreme, i.e., too large or too low, then, usually all the replacement policies show similar behavior for a given page fetch policy. Observing the collected statistics for page sizes 1 and 32, it is evident that the number of page faults is almost close for each of the page replacement policies for a given page fetch policy.
- Further, when the page size is small, it can be observed that, generally, pre-paging produces much better performance as compared to demand paging.
- Also, when the page size becomes too large, the performance of pre-paging reduces, and in some cases, it is out-performed by demand paging.
- When the page size is intermediate, i.e., neither too high nor too low, then it can be observed that FIFO replacement policy produces the maximum number of page faults whereas Clock and LRU generally produce a lesser number of page faults. Although the number of page faults produced by both Clock and LRU replacement policies is both close, generally, LRU produces a lesser number of page faults as compared to Clock. This shows that in most cases, LRU performs the best, closely followed by Clock, whereas FIFO generally gives the worst performance.
- Also, in case of intermediate page size, again, pre-paging usually produces better results as compared to demand paging as the number of page faults produced for pre-paging for a given replacement algorithm is generally lesser than the number of page faults produced by demand paging.
- Further, for a given combination of page fetch policy and page replacement policy, it is observed that as the page size increases, the number of page faults initially decreases and then increases. When page size is very small, then the number of page faults is quite high. The number of page faults decreases as the page size is increased till a certain point, after which the number of page faults again increases.

VII. Conclusion

In this paper, we successfully developed an end-to-end model for virtual memory management to experiment and analyze the various combinations of fetch policies and replacement algorithms based on varying page sizes as given by the user, and we concluded on the fact that when the page size is too small or too high all replacement polices show similar results for a given fetch policy. For intermediate page sizes, LRU and Clock replacement policies are generally better than FIFO. On the other hand, pre-paging usually gives better results than demand as the operating system guesses the future page requirements for the process.

VIII. References

[1] Meng Wang, Zhenhua Duan, Cong Tian, *Simulation and Verification of the Virtual Memory Management System with MSVL*, IEEE, 2014

[2] Fadi N. Sibai, Maria Ma and David A. Lill, *Development of a Virtual Memory Simulator to Analyze the Goodness of Page Replacement Algorithms*, IEEE, 2007

[3] A. Silberchatz, P. Galvin, and P. Gagne., J. Wiley., *Operating Systems Concepts, 8th Edition*, 2012.

[4] William Stallings., *Operating Systems: Internals and Design Principles, 7th Edition*, 2012.

[5] Page replacement algorithms, *http://www2.cs.uregina.ca*