Advanced Engineering Technology and Application
An International Journal

http://dx.doi.org/10.18576/aeta/050203

Design and Implementation of Page Replacement Algorithms based on an Inverted Page Table

*Yeskendir Sultanov**

Kazakh, British Technical University, Almaty, Kazakhstan

Received: 15 Mar. 2016, Revised: 20 Apr. 2016, Accepted: 22 Apr. 2016
Published online: 1 May 2016

Abstract: Main memory is one of the most important parts in a computer. Although amount of main memory has increased in recent decades, it is very limited resource. This fact says us that it has demand, to optimize swapping of pages between main memory and auxiliary memory. As is known, a performance of paged systems significantly depends on an efficiency of the transformation of virtual address into a physical address. There are some well-known very fast and flexible algorithms in which paged systems are constructed. The choice of data structure that supports these page replacement algorithms plays important role for paged systems. In this paper, as such data structure is taken the Inverted Page Table (IPT). This data structure allows system to optimize memory occupied by the page table and to ultimately reduce time to transform the virtual address into the physical address. In this work, we will design and implement page replacement algorithms based on the inverted page table.

Keywords: Inverted page table (IPT), virtual memory, page replacement algorithms

1 Introduction

All memory of the computer as a virtual as well as a physical divided into successive pages of same size[1]. Each element of the program receives a virtual address, when the program is running. There's certainly to be a correspondence between the physical and virtual address and this process is done automatically by the operating system. The auxiliary and main memory exchanging is realized by whole pages, and when this process is running, CPU switches to execution of commands of other programs (see Figure 1). If in the main memory a link to page missing takes place during the execution of the program, then the page faults occur (failure)[2]. The program is interrupted for the time necessary to swap the page. In this case, one or some of the pages of the program is deleted from the main memory, that is, the memory occupied by them is considered to be free. If the content of the page to be deleted, distorted during her stay in the main memory, then the system provides an overwriting of modified pages in the auxiliary memory while preserving the original content of the pages of the original. Otherwise, the necessity to rewrite the page at the auxiliary memory is not necessary. Select a page (or group of pages) to be deleted from the main memory; the system is carried out in accordance with a particular algorithm called page replacement strategy. As processor get faster rapidly, the impact of swapping between main memory and auxiliary memory increases. It says us that there is necessity to optimize transforming process of virtual address into physical address, which significantly effects on the performance of paged systems. There are some well-known fast and flexible page replacement algorithms, in which paged systems are constructed. In this paper, we will first introduce general concepts of virtual memory and paging, followed by proposed formal

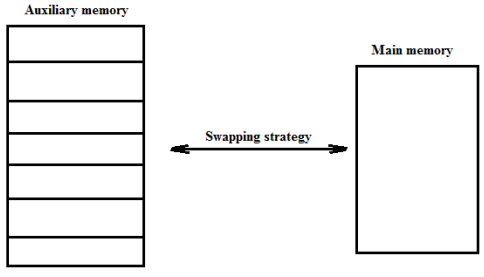

Fig. 1: Figure 1. The exchange between auxiliary and main memory

* Corresponding author e-mail: yeskendir.sultanov@gmail.com

models for page replacement algorithms and program behaviour. Then we will design and implement these page replacement algorithms based on the inverted page table, which allows system to optimize memory occupied by page table and to ultimately reduce time to transforming process. Also, we will compare their performance on generated memory traces. The performance comparison is made with help of page fault table, which show us the page faults count.

2 Overview of Memory Management

Almost all modern operating systems those for general purpose usually use virtual memory to solve overlay problem[3]. In the virtual memory there may be lack of the amount of main memory available in the system for the combined size of program code, data and stack. The operating system uses secondary memory, in addition to main memory to keep active pages in main memory and inactive pages in auxiliary memory. Pages located in auxiliary memory can retrieved back to main memory, when it is necessary[4]. The process of storing data from main memory to secondary memory is called swapping out, and retrieving data back to main memory is called swapping in. These will be referred as swapping except when distinction between the two is necessary. The part of the secondary memory, that is reserved for virtual memory, is called swap space, and is often implemented as a swap partition or a swap file[5]. There are two granularities in which swapping is commonly done in multitasking operating systems. The simplest one is to swap out a whole program when memory is needed. This simple method can be used as a load balancing technique [6]. Virtual memory provides processes a virtual address space. Programs use virtual addresses to refer to their own virtual address space. When virtual address space is used, each program sees a flat continuous memory dedicated for it alone. All memory, however, is not available for a running program. The kernel usually maps its own address to constant area of each programs address space. In Linux the kernels space is normally mapped at the end of the processes address space. As an example, on x86 architecture the last 1 GB of the 4 GB address space is reserved for the kernel. This leaves 3 GB for the user process[7]. Virtual address space simplifies compilers and applications as the memory used by the operating system, and other running programs, are not directly visible to a running program.

2.1 Paging and Page fault handling

The operating system divides virtual address space into units called pages. Main memory is also divided to fixed size units called page frames[8]. Each used page can be either in secondary memory or in a page frame in main memory. A paging algorithm is needed to manage paging. A paging algorithm consists of three algorithms: placement algorithm, fetch algorithm and replacement algorithm. The placement algorithm is used to decide on which free page frame a page is placed. The fetch algorithm decides on which page or pages are to be put in main memory. Finally, the page replacement algorithm decides on which page is swapped out. Further, paging algorithms can be demand paging or prepaging. A demand paging algorithm places a page to main memory only when it is needed, while a prepaging algorithm attempts to guess which pages are needed next by placing them to main memory before they are needed. In general cases, it is very difficult to make accurate guesses of page usage and demand paging is generally accepted as a better choice. It can also be proved, that for certain constraints, optimal paging algorithm is a demand paging algorithm. Exact constraints and proof is given in[9]. A virtual address must be translated to corresponding physical address before the memory can be used. As this address translation is done with every memory reference, it is important that it is very fast. Usually special hardware, called Memory Management Unit (MMU), is used to make this translation. MMU uses virtual-to-physical address mapping information, located in operating systems page table, to make the translation. Each process has its own virtual address space and therefore page tables are per process. If the given virtual address is not mapped to main memory, the MMU traps the operating system. This trap, called page fault, gives the operating system an opportunity to bring the desired page from secondary memory to main memory, and update to page table accordingly[6].

2.2 Page replacement algorithm theory

Page replacement algorithms have been studied and some formal models are proposed to be used as basis of theoretical analysis. The following conventions are used. Set of pages of a n-page program is defined as

$$N = \{p_1, ..., p_n\}; \tag{1}$$

and

$$M = \{pf_1, ..., pf_m\}; \tag{2}$$

is a set of page frames of main memory with space for m pages. Function

$$f : N \to M; \tag{3}$$

gives current page map and can be defined as

$$f(p_i) = pf_j \tag{4}$$

if page $p_i \in pf_j$.
Otherwise it is undefined and page fault must occur[5].

3 Inverted Page Table

The presence of regular table scheme pages make such a scheme is not sufficiently effective. Storing in the main memory of the computer of excessive use information, so information on inactive pages, which makes up most of the page table, will slow down the entire storage management system and negatively affect the performance of the system as a whole. This leads to the idea that it would be more economical, at any point in the process of activity, stored in the main memory only information regarding only those pages that are currently in main memory. In this connection, instead of a regular page tables, we will use the inverted page table (see Figure 2).

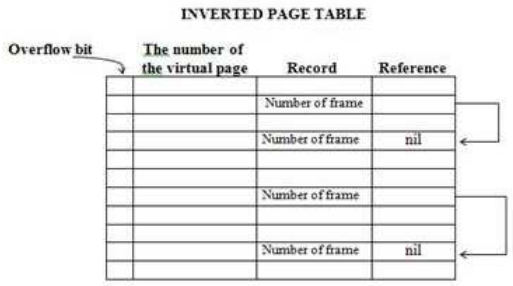

Fig. 2: Inverted Page Table

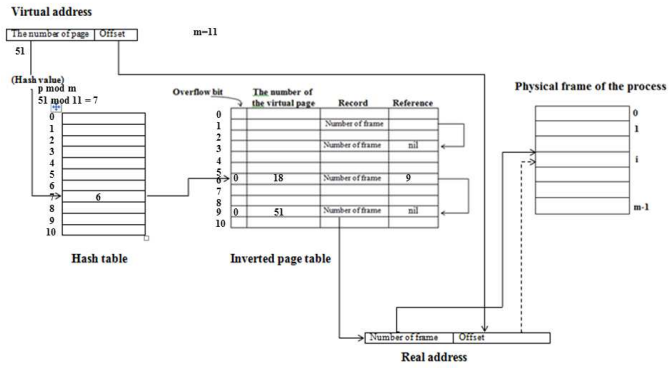

Fig. 3: The design of the page replacement algorithm based on the Inverted Page Table

An inverted page table, at any time of the program, stores information about a virtual page part of the process, namely those in which the copy currently in the main memory, therefore it reduces the size of the memory occupied by page table, in spite of fact that we search through linked list elements[10]. Finally, if we get page faults, it means that page does not exist in the main memory and system should load it from the auxiliary memory[11]. The Figure 3 shows the design of the page replacement algorithm based on the inverted page table.

4 Implementation of page replacement algorithms based on the Inverted Page Table

In this paragraph, we will introduce page replacement algorithms theory and we will show the implementation of those algorithms based on the inverted page table. One of those algorithms is optimal page replacement algorithm, which is easy to describe. When memory is full, you always evict a page that will be unreferenced for the longest time. This scheme, of course, is possible to implement only in the second identical run, by recording page usage on the first run. But generally the operating system does not know which pages will be used, especially in applications receiving external input. The content and the exact time of the input may greatly change the order and timing in which the pages are accessed. But nevertheless it gives us a reference point for comparing practical page replacement algorithms. This algorithm is often called OPT or MIN[5]. Next algorithm is First-In, First-Out (FIFO) algorithm is also applicable to page replacement. All pages in main memory are kept in a list where the newest page is in head and the oldest in tail. When a page needs to be evicted, the oldest page is selected, and the new page is inserted to head of the list[5]. The third algorithm that we are going to implement base on the inverted page table is random page replacement algorithm. If a frequently used page is evicted, the performance may suffer. For example, some page, that contains program initialization code which may never be needed again during the program execution, could be evicted instead. So there are performance benefits available with choosing the right page[5]. We implement those algorithms based on the inverted page table on programming language C++ and we have gotten good results. Below we have given pseudocode of that implementation with explanations.

4.1 Pseudocode of the page replacement algorithms implementation based on the inverted page table

The implementation has two tables: *HASH TABLE* and *IPT*; functions: *getHash(pageNum)*, *find(pageNum)*, *search(pageNum)*, *algoDesign(pageNum)*. Below we have provided realization of those functions in form of pseudocode.

```
struct ipt_row
begin
   boolean isListOverflown;
```

```
   integer pageNumber;
   integer pageFrameNumber;
   integer reference;
   boolean isRowFree;
end;

DECLARATION:
integer HASH_TABLE[LENGTH];
ipt_row IPT[LENGTH];

FUNCTIONS:
integer getHash(pageNumber)
begin
   return pageNumber mod LENGTH;
end

boolean find(pageNumber)
begin
   integer hash = getHash(pageNumber);
   if (hash NOT_EXIST in HASH_TABLE)
      return false;
   else
      return true;
end

boolean search(pageNumber)
begin
   integer hash = integer hash =
       getHash(pageNumber);
   integer startRow = HASH_TABLE[hash];

   integer curRow = startRow;

   while (curRow != NULL)
   begin
      if (IPT[curRow].pageNumber ==
          pageNumber)
         return true;
      curRow = IPT[curRow].reference;
   end;
   return false;
end

procedure algoDesign(pageNumber)
begin
   if (find(pageNumber))
   begin
      pageFault occurs
      make pageLoading
   end;
   else
   begin
      boolean isPageFault =
          search(pageNumber);

      if (isPageFault)
      begin
         pageFault occurs;
         make pageLoading;
      end
      else
      begin
         page was found in the inverted
             page table;
      end;
   end
end;
```

We have used *p mod m* function as a hash function, where *p* - is page number and *m* - is inverted page table length. It is implemented in the *getHash(pageNumber)* function. Next function is *find(pageNumber)*. It looks for page number's hash in the *HASH TABLE*. If this function returns true value, then next function *search(pageNumber)* is called, otherwise, page fault occurs and it must happen page loading. The *search(pageNumber)* function looks for page with necessary page number in a linked list with same hash value in the inverted page table. If we get true result after calling this function then we have found page frame of our page; else page fault occurs and it must happen page loading.

5 Empirical analysis

We have generated some reference string. Then give it as input to the program. The program has executed and we got results as shown in the Table 1. Table 1 shows us that

Data	OPT	FIFO	RANDOM
Ref count	17566	17566	17566
Page count	8783	8783	8783
Page Faults	7150	9031	8679
Hit count	10416	8535	8887

Table 1: (Results on scan data)

optimal algorithm more fast than other two algorithms. This result was taken by generated reference string. As you see from the table in each page replacement algorithm implementation reference and page counts have the same value. Hit count of each algorithm are different and it says us that usage of the inverted page table does not change properties of page replacement algorithms.

6 Conclusion

In this paper we have explained design and implementation of page replacement algorithms based on the inverted page table. Usually, operating systems use page table as a data structure, which uses memory inefficiently. The inverted page table allows to ultimately reduce memory occupied by pages. In this work, initially, we have introduced theoretical concepts memory

management of operating systems, paging and page fault handling. Also we have introduced the design of page replacement algorithms based on the inverted page table. The Inverted Page Table in contrast to the page table stores only active pages, this leads to the idea that it would be more economical, at any point in the process of activity, stored in the main memory only information regarding only those pages that are currently in main memory. It optimizes the size of memory occupied by page table and it reduces the time of transformation virtual address into physical as system will not spend time to record information about inactive pages. Then we have implemented those page replacement algorithms and we made empirical analysis. The result is shown as Table 1 and by analyzing that result we can see that page replacement algorithm's properties have been retained. This implementation need further improvements and additional tests. However, inverted page table implementation ultimately optimizes the memory usage and it increases the performance of operating system.

References

[1] XU Chao, HE Yan-xiang, CHEN Yong, WU Wei, Zeng Xiao-ling. An Optimization Algorithm of Variable Allocation Based on Block Architecture, Volume 7, No. 2 (Mar. 2013), PP:691-699, *Natural Science Publishing: International Journal of Applied Mathematics and Information Sciences*

[2] Slo-Li Chu, Min-Jen Lo, Novel Memory Access Scheduling Algorithms for a Surveillance System, Volume 7, No. 2 (Mar. 2013), PP:801-808 *Natural Science Publishing: International Journal of Applied Mathematics and Information Sciences*

[3] S.L. Harris, D.M. Harris. Digital Design and Computer Architecture. Elsevier Inc., 2nd Ed.2012

[4] Randal E. Bryant, David R. OHallaron, Carnegie Mellon University. Computer Systems: A Programmers Perspective, third edition. 2015

[5] Heikki Paajanen, Page replacement in operating system memory management, 2007

[6] Song Jiang and Xiaodong Zhang, "Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems, Performance Evaluation" 60 529, 2005.

[7] Mel Gorman, "Understanding the Linux Virtual Memory Manager", Bruce Perens Open Source Series, Prentice Hall, 2004.

[8] Andrew S. Tanenbaum and Albert S. Woodhull, "Operating Systems: Design and Implementation", Third Edition, Prentice Hall, 2006.

[9] Alfred V. Aho, Peter J. Denning and Jeffrey D. Ullman Principals of Optimal Page Replacement Journal of the Association for Computing Machinery, Volume 18, No. 1, January 1971

[10] W. Stallings. Computer Organization and Archtecture. Pearson Ed.,2006

[11] A.Duysembaev. Computer Architecture. Almaty, 2011

Yeskendir Sultanov is master's degree student of Kazakh-British Technical University. His main research interests are: algorithms, data structures, operating systems, big data.