Spring 5-22-2017

Switching Between Page Replacement Algorithms Based on Work Load During Runtime in Linux Kernel

Praveen Subramaniyam
San Jose State University

Recommended Citation

Subramaniyam, Praveen, "Switching Between Page Replacement Algorithms Based on Work Load During Runtime in Linux Kernel" (2017). *Master's Projects*. 515.
DOI: https://doi.org/10.31979/etd.xj6w-cucm
https://scholarworks.sjsu.edu/etd_projects/515

Switching Between Page Replacement Algorithms Based on Work Load During Runtime in Linux Kernel

A Project Report

Presented to

Dr. Robert Chun

Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Class

CS 298

By

Praveen Subramaniyam

May 2017

This Project is Approved by The Master's Committee

Dr. Professor Robert Chun

Dr. Professor Thomas Austin

Mr. Dinesh Rathinasamy Thangavel

ABSTRACT

Today’s computers are equipped with multiple processor cores to execute multiple programs effectively at a single point of time. This increase in the number of cores needs to be equipped with a huge amount of physical memory to keep multiple applications in memory at a time and to effectively switch between them, without getting affected by the low speed disk memory. The physical memory of today’s world has become so cheap such that all the computer systems are always equipped with sufficient amount of physical memory required effectively to run most of the applications. Along with the memory, the sizes of applications have also became huge. This again arises the problem of memory contention in most of the heavily loaded servers. So the physical memory has to be handled very effectively to achieve high performance. Many page replacement algorithms were developed by researchers and everything has its own advantages and disadvantages. All the algorithms have one goal of minimizing the page faults with less background work done [1]. Effectiveness of an algorithm depends on the work load as well. This paper discusses the various page replacement algorithms, their benefits, their failures and analyses them in certain workloads. Finally it presents an idea of changing the page replacement algorithms depending on the performance of each algorithm in that particular workload during runtime using system variables.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my advisor, Prof. Dr. Robert Chun, for his expertise, patience, encouragement, and continuous guidance throughout my graduate studies and my final project.

I am very grateful to the committee member, Professor Thomas Austin for being extremely helpful by giving his suggestions and for monitoring the progress of the project.

I would also like to thank Mr. Dinesh Rathinasamy Thangavel for his valuable time and support through out the project.

Finally, a big thank you to the Department of Computer Science at San Jose State University for giving me this opportunity. I couldn’t have done it without you.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Problem statement

Increase in the amount of physical memory is always accompanied by the increase in the size of the programs that are executed in the computer systems. Concurrent system models always require multiple programs to be executed in parallel. To execute programs in parallel all the programs should be in the main memory to increase the performance and to decrease the waiting time of the users. It’s impossible to keep all the running programs as a whole in the main memory at all times. Only some portions of the running process can reside in the main memory and the program is divided into multiple units called pages [16]. By this way important pages of multiple programs can be kept in the memory. Switching between processes will be faster giving equal priority to all user programs. But whenever a program is executing if it needs some pages which are not currently in the main memory, then the pages needs to be copied into the main memory replacing some pages already in the main memory. This is called demand paging. The pages will be copied to the main memory only if the process needs to access those pages. This is the main task of the page replacement algorithm. Page fault is said to happen when the process which is currently executing, requests for a page which is not already loaded into the main memory. Choosing the replacement page from the main memory is a critical decision taken by the page replacement algorithm. The algorithm should choose a page which is not going to be used in the near future as the one to be replaced. But predicting the future is not always correct, so predicting is not a correct solution for the problem. The algorithm should designed in such a way that the run time of the algorithm should not exceed the time saved between swap in and swap out of the pages from the main memory to the disk. Algorithms developed so far use many approaches to handle this page replacement problem. This paper discusses those algorithms, approaches followed by them and a comparison among all those.

1.2 Solution

The paper finally provides an approach which may be one of the solution to the page replacement problem. The solution goes as follows: instead of implementing one paging replacement algorithm in the operating system for use, the solution implements multiple algorithms and a kernel parameter is used to switch between the algorithm depending on the workload. Kernel parameter can be modified at run time to choose the algorithm that needs to be executed for the current work load. The main aim of the whole process is to reduce the execution time of a workload by decreasing the number of page faults without increasing the complexity of the whole process. Decreasing the page faults involves effectively judging the pages that might be referenced in the near future.

1.3 Scope

The scope of the project is to provide a support to change of the page replacement algorithms depending on the current load on the machine. The system assumes that the work load will be the same during the execution of the machine and a switch to the algorithm which outputs less page faults can be made manually using the system parameters. The system will adapt to a different work load after certain period of time since the algorithm to use can be changed manually according to the page faults in each algorithm. In worst case if the work load changes every time a switch needs to be made every time.

2. BACKGROUND

The project has two section. The first section introduces the various page replacement algorithms and their working. The second section provides an approach which may be one of the solution to the page replacement problem.

2.1 Belady's Min

Belady's Min algorithm is the ideal page replacement algorithm which is suited for all the workloads. It states that the algorithm can replace the page which will be used farthest in the future or which is not used at all [1]. But it's impossible to predict the future in any case as all the applications are accessed randomly in any computer system, so it's always impractical to implement this algorithm. This algorithm is used as a benchmark for the other algorithms.

2.2 First In First Out (FIFO)

As the name suggests the algorithm chooses the page which is first brought into the main memory among all the other pages as the replacement one. The pages are added to the tail of the queue which is a first in first out data structure [2]. The page which is at the FIFO queue's head will be replaced in case of page faults. This algorithm is not efficient since the page at the head of the queue might be used very often. Removing it will lead to an extra page fault [3].

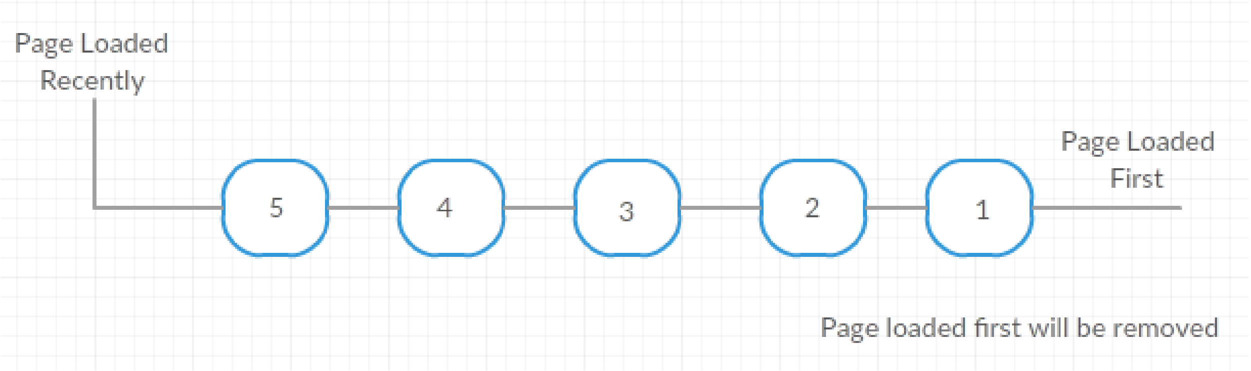


Figure 1 First In First Out Queue Representation

2.3 Second Chance Algorithm

This algorithm is an advancement over the FIFO algorithm. Similar to FIFO, this algorithm replaces those pages which are brought first into the main memory among all the other ones. A reference bit ‘r’ is added to the page structure and will be set as the page is referenced. In case of page fault, the reference bit of the oldest page in the queue is examined, if it is set, then it is made zero and the page is added back to the queue [4]. The first page which is having ‘r’ bit as zero will be chosen as the replacement page. Second chance algorithm is better than FIFO though it’s not the ideal one. In the worst case if all the bits are set, then all the bits are unset, and the queue head comes back to the original position where it was before and removed from the main memory. This algorithm requires hardware support for the reference bit [4].

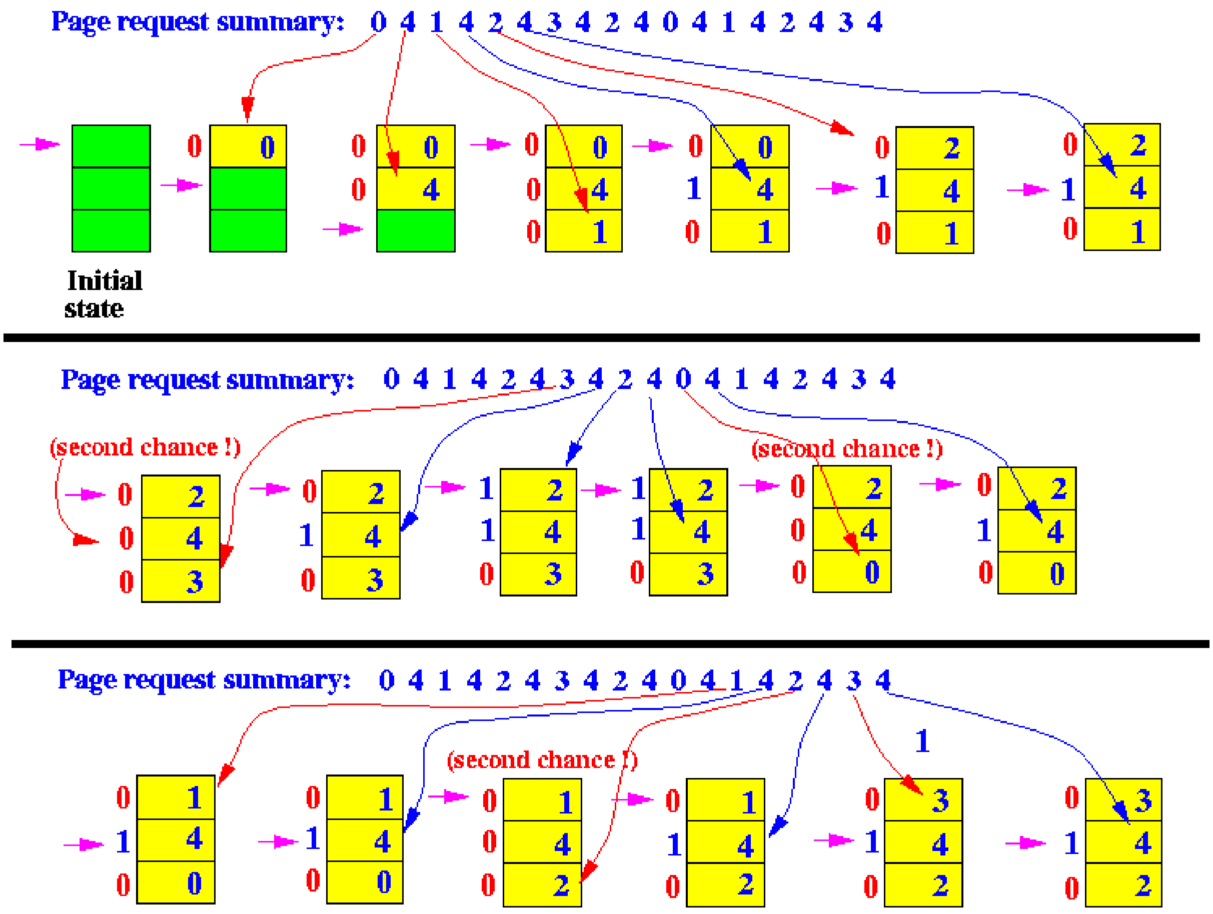

Page request summary: 0 4 1 4 2 4 3 4 2 4 0 4 1 4 2 4 3 4
Initial state
Page request summary: 0 4 1 4 2 4 3 4 2 4 0 4 1 4 2 4 3 4
(second chance !)
(second chance !)
Page request summary: 0 4 1 4 2 4 3 4 2 4 0 4 1 4 2 4 3 4
(second chance !)
1

Figure 2 Second Chance Page Replacement Algorithm [4]

2.4 Enhanced Second Chance Algorithm

This algorithm is an enhanced version of second chance algorithm. Along with the reference bit, this algorithm also introduces a modify bit. This algorithm also considers the modification done to the content of the pages. Modification done to the pages needs to be written back to the disk which takes some time, so the page without modification is chosen as the replacement page when compared to the page with modification. Whenever a page is accessed and its content is changed the modify bit is set for that page. Now the replacement policy goes like below.

(Reference Bit, Modify Bit)	Replacement Policy
(0, 0)	Not recently referenced and modified. First page to replace
(0, 1)	Modified recently. Can replace but need to write out
(1, 0)	Not modified, but accessed, so rather not replace
(1, 1)	Modified as well as accessed. Don't replace

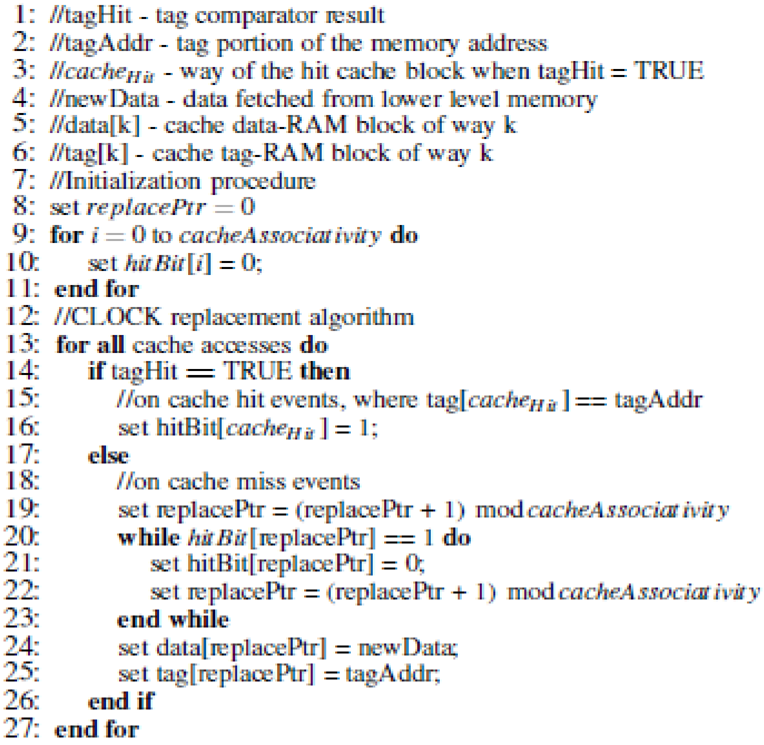
2.5 Least Recently Used (LRU)

This is a good approximation to the ideal page replacement algorithm. The algorithm considers the principle of locality. The pages which are used heavily in the last few instructions are assumed to be used heavily again or the pages which are not used for ages are assumed to be probably not used at all [2]. In case of page fault it is good to replace the page which is not used recently among all the other pages. The algorithm can be implemented using the linked list with the page which is most recently used at the head and the one which is least recently used at the tail of the linked list. The pages are replaced from the tail of the linked list. Since the algorithm always moves the recently used pages to the head of the list, the list needs to be modified after every page reference. Though implementing the LRU is achievable, it's very difficult to implement even in hardware [2].

One way of implementing in the hardware as described in [2] is to have a 64-Bit counter in the page table entry for all the pages in the memory. This counter value is incremented whenever the page is accessed. The page table entry is scanned every time and a page with the least counter value is replaced. The main problem with the LRU is that it behaves worse in case of sequential access of large files. The infrequent sequential access of large files always evicts the most commonly used files from the cache [7]. LRU doesn't perform well for cyclic pattern of access to a file if the file is little bit greater than the size of the cache. It always evicts the file which will be used sooner [8].

2.6 Clock

The algorithm is an approximation for LRU. The pages are added to a circular linked list and a pointer points to the page which is referenced last. A reference bit is added to the page table entry of a page and its set whenever the page is accessed. In case of page faults, the reference bit of those pages with a value of '1' is made 0 and the first page with the reference bit 0 is replaced [1]. The algorithm is much more like an approximation for LRU with less overhead.



```
//tagHit - tag comparator result
//tagAddr - tag portion of the memory address
//$cache_{Hit}$ - way of the hit cache block when tagHit = TRUE
//newData - data fetched from lower level memory
//data[k] - cache data-RAM block of way k
//tag[k] - cache tag-RAM block of way k
//Initialization procedure
set $replacePtr = 0$
for $i = 0$ to $cacheAssociativity$ do
    set $hitBit[i] = 0$;
end for
//CLOCK replacement algorithm
for all cache accesses do
    if tagHit == TRUE then
        //on cache hit events, where tag[$cache_{Hit}$] == tagAddr
        set hitBit[$cache_{Hit}$] = 1;
    else
        //on cache miss events
        set replacePtr = (replacePtr + 1) mod $cacheAssociativity$
        while $hitBit$[replacePtr] == 1 do
            set hitBit[replacePtr] = 0;
            set replacePtr = (replacePtr + 1) mod $cacheAssociativity$
        end while
        set data[replacePtr] = newData;
        set tag[replacePtr] = tagAddr;
    end if
end for
```

Figure 3 Clock Algorithm[1]

2.7 Frequency Based Replacement

Frequency based replacement is similar to LRU but it maintains three sections for storing pages: new, middle and old section [5]. Pages are moved between these sections depending on the time they are in the main memory. FBR maintains a reference frequency count for all of the pages. If a page is referenced its stored in the new section. On a page fault the page from the old section with the lowest frequency count is replaced. FBR's failure includes the fact that it needs to increase the reference count every time and switch a page between the three sections every time on a page reference.

2.8 Least Frequently Used (LFU)

This page replacement algorithm Least Frequently Used works by calculating the frequency of the access of each page. Each page will be assigned a counter in the page table and the counter is incremented every time a page is referenced. In case of page faults, the page with the lowest frequency is replaced. Though the algorithm seems to be very effective since it calculates the frequency of access instead of time of access which had problems in LRU, this also has its own disadvantages. Consider a situation where a page is accessed repeatedly many number of times before a long time period and it's not going to be used anytime in the future and another page which is accessed only once and will be accessed in the future for sure. Whenever a page fault occurs at this time, the page which has high frequency and is not going to be accessed in the future will reside in the memory and the page with less frequency and which is going to be accessed in the future for sure will be removed from the memory. This is a failure case for the LFU algorithm.

The algorithm gives low preference to pages that just entered the main memory since they will be having the low counter even though they might be used very frequently later. Due to these drawbacks many hybrid varieties of LFU are always used as the page replacement algorithms.

3. ADVANCE RESEARCHES IN PAGE REPLACEMENT ALGORITHMS

3.1 LRU-K

The problem with the LRU algorithm is that it considers only the pages that are recently used as worthy although the probability of accessing them in the future might be low. The page which is accessed recently but accessed only once gets the advantage over the page which is not accessed recently but accessed many times in the past. The page which is referenced many times in the past is most likely to be referenced again in the future [5]. LRU fails to consider this scenario. LRU-K which is an advanced version of LRU tries to solve this problem. LRU-K saves the last K references of each page. A reference probability is calculated using the time difference between the Kth last reference of the page and the current time is used as a page replacement policy. The page specific access rate also known as the page's heat which is used to determine the access probability is calculated as

$$\text{heat (p)} = \text{K/ (current time – Kth last reference time)}$$

Then the access probability of each page in the next 'T' time units is calculated as

$$\text{access probability} = 1 - e^{-\text{heat(p)T}}$$

The page with the minimum probability value is then used as the replacement page [2]. To gain the advantage of low overhead implementation it is sufficient to maintain the heat(p) mentioned above or the last k references of each page. The page with the lowest heat or the oldest k-th last reference page is considered for replacement. The algorithm gave good results for K = 2. In

LRU-2 the algorithm maintains two most recent references of a page and chooses a page with the least second-most reference time as the replacement one[5].

3.2 Dueling Clock (DC)

Dueling clock is an adaptive version of CLOCK algorithm and has a low overhead when compared to LRU. Clock algorithm is not scan resistant i.e.; it is prone to failure in case of sequential access of single-use pages which are greater than the memory available. In CLOCK algorithm the clock pointer always points to the recently added page and the scanning always starts from the page which is one greater than the recently added page. This might replace a page which was added before and has been accessed many times compared to the one which is recently added and accessed only once. So the CLOCK algorithm doesn't consider frequency. DC avoids this by starting the scan from the recently added page instead of one page greater than the recently added page [9]. Line 19 is removed from the figure CLOCK algorithm to implement the DC algorithm [9].

3.3 Least Recently/Frequently Used (LRFU)

The least recently/frequently used (LRFU) algorithm is a combination of LFU and LRU algorithm. Every page is assigned an initial value $F(x) = 0$. $F(x)$ is updated for every page access, depending on the parameter $y > 0$ as

$$F(x) = yF(x) \quad \text{for all the pages other than the page which is referenced}$$

$$F(x) = 1 + yF(x) \text{ for the page which is referenced}$$

In case of page faults the page with the minimum $F(x)$ value is chosen as replacement one. As y tends to 0 then $F(x)$ will be equal to the number of times the page is referenced, hence the algorithm is similar to LFU. As 'y' tends to 1, the page which is recently accessed will have the greater value for F(x), so the algorithm behaves similar to LRU. So it’s very critical to choose the ‘y’ value. The performance of the algorithm depends on the ‘y’ value [14].

3.4 Adaptive Replacement Cache (ARC)

The Adaptive Replacement Cache (ARC) algorithm is an advancement to LRU and LFU algorithm. The algorithm dynamically adapts to different workloads. The algorithm LRU-k and LRFU uses a parameter which needs to be tuned by the user for different work loads. ARC doesn’t use any parameter and the implementation is simpler than the other two. Time complexity is constant per request of a page.

The implementation consists of two LRU page lists instead of one, L1 and L2 [5]. The pages which are referenced only once are added to the list L1 and the pages which are referenced more than once are added to the list L2. The pages which are accessed more than once are likely to be accessed again, so it can be said that L2 captures frequency and L1 captures recency [5]. L1 is further separated into B1 and T1, similarly L2 into B2 and T2. T1 is for recently referred pages and T2 is for frequently referred pages. T1 is extended by B1 which is called a ghost list, since it has the metadata of those pages which are evicted from T1. It’s to be noted that B1 has only metadata of all the pages; not the original pages. Similarly, T2 has those pages that are accessed more than once, and B2 is the ghost list of the pages evicted from T2 [6].

The diagram below gives a good view of the ARC algorithm.

$$T1 + T2 = c \quad B1 + B2 = c$$

$$L1 + L2 = 2c$$ [6]

where c denotes the total cache size in the number of pages. Top of L1 is the T1 which contains the most recently used (MRU) pages; bottom of L1 is B1 which contains the least recently used pages. Similarly, top of L2 is T2 and bottom of L2 is B2.

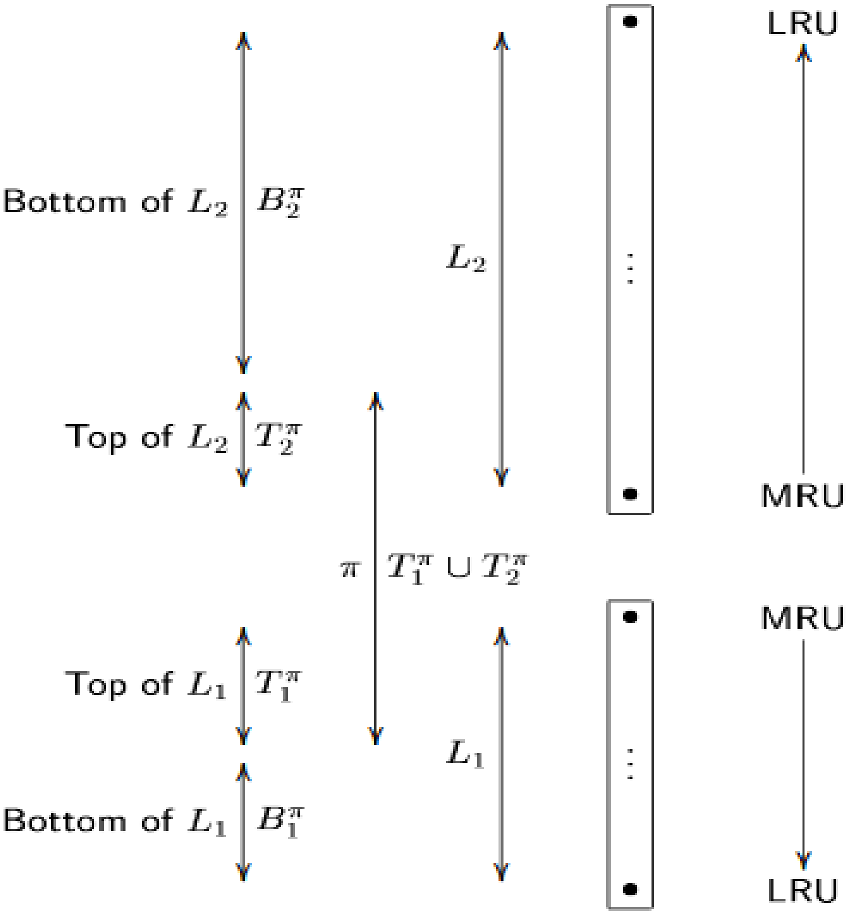
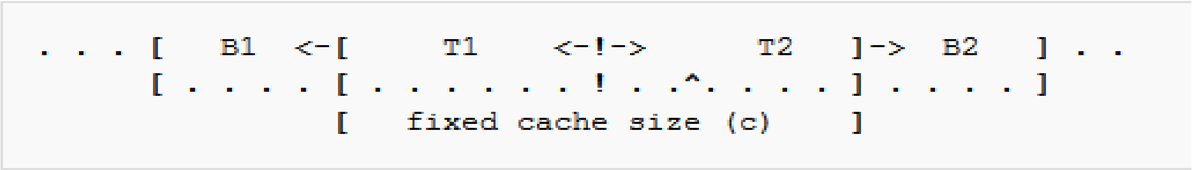


Figure 4 Working of ARC algorithm [6]



```
. . . [   B1  <-[     T1    <-!->      T2   ]-> B2   ] . .
      [ . . . . [ . . . . . . ! . .^. . . . ] . . . . ]
                [   fixed cache size (c)    ]
```

Figure 5 Representation Model of ARC algorithm [6]

In the above figure L1 list is interpreted from right to left, starting from the **!** Marker. ^ marker indicates the size limit of the list T1. The pages that just enter into the cache is placed to the left of ! and is gradually moved to the left. Eventually it is moved into the B1 and finally it is dropped. In case if a page in T1 is referenced again, it is moved into T2 just to the right of ! and is moved to the further right gradually, finally its moved into B2. If any of the pages in the T2 is referenced one more time, then its again added just right of ! and it follows the same process again.

In case of page fault, if the referenced page is in the list B1, the page will be moved to the list T1 from B1, moving the indicator ^ towards the right end and the last page in the list T2 will be moved to B2. If the page to be referenced is in B2, then ^ is moved towards the left end and the last page in T1 will be moved into B1. If the page is not in both B1and B2, then the page is added to T1 without affecting the ^ indicator [6].

4. IMPLEMENTATION

4.1 Current Implementation

The previous section discussed all the page replacement algorithms suggested as of today. Each one has its own advantage and disadvantage. The algorithm which works in most of the case is often implemented in many of the current operating systems available these days. Among those, Least Recently Used (LRU) algorithm is implemented currently in the Linux Kernel.

The following sections discuss about the implementation details of the Least Recently Algorithm and implements Least Frequently Used algorithm and modified version of Least Recently Used algorithm which is Least Recently Used – K (LRU-K).

4.2 Factors to be considered for a page replacement algorithm

Before discussing about the implementation, its important to consider some of the factors that influences a page replacement algorithm. The page replacement is mainly done on three occasions:

Low on Memory Reclaiming

This condition occurs whenever the kernel detects a low free space in the memory for the process to start executing. The kernel should replace some pages in the memory before it can start executing the process or while the process is executing [15].

Hibernation Reclaiming

This condition occurs whenever hibernation happens and the kernel needs to remove all the pages from the memory [15].

Periodic Reclaiming:

In some linux kernel a thread is invoked periodically to free some space in the memory if necessary [15].

The algorithm should also consider the different kind of pages that exists in the system. Some of them are described below.

- Unreclaimable Pages

  The pages that belong to the kernel. These pages should not be replaced at all and are identified by PG_reserved flag set, which informs the operating system that the page should not be replaced at all.

- Swappable Pages

These are the pages that belongs to a user process and can be swapped whenever necessary. These pages can be replaced and can be stored in the swap area whenever needed.

- Syncable Pages

  These are pages that are mapped from the secondary disk and belongs to a user process. These can be replaced and needs to be synchronized with its image on the disk before replacing.

- Discardable Pages

  These are pages of some disk caches and some unused pages included in the memory caches. These pages can be replaced and can be discarded while replacing.

4.3 Design of the Page Replacement Algorithm

Selecting the appropriate page for replacement is the most sensitive task in the page replacement algorithm. Compared to kernel pages, disk or memory cache or pages of user mode process should have more chance for replacement. Effective page replacement algorithm should take care of all these conditions below

Harmless pages can be freed first

- Pages belonging to the caches in the disk or memory and which are not used by the currently executing processes should be first replaced than the ones which belongs to the currently executing user process. In this case, the page tables of the user process need not be modified as well [15].

User Mode Process Pages should be made reclaimable

- The next candidate for effective page replacement policy are the pages that belong to the user mode process and those which are not locked. In this way the pages that belong to a long sleeping process can be reclaimed [15].

Replacement of a shared page

- The next candidate for effective page replacement policy is the shared pages. Before replacing a shared page all the page table entries that belong to multiple processes need to be removed [15].

Replace “unused” pages only

- According to principle of locality the pages that are not referenced for a long time period will not be mostly referenced again. The algorithm should consider this principle and replace only the pages which are not referenced for a long time period.

4.4 Current Implementation in Linux Kernel

Least Recently Used (LRU) is the algorithm which is currently implemented in the Linux kernel [19]. LRU replaces those pages which are not used recently or the oldest pages. The algorithm maintains two lists namely active list and inactive list to facilitate the page replacement [19]. The algorithm maintains those pages which are recently accessed in the active list and those pages which are not accessed for a certain period of time in the inactive list. The page to be replaced is always taken from the inactive list.

The states active and inactive are not just enough to handle all the workloads. For example in case of a logger which will be accessed every hour will be in the inactive list most of the time but when it is accessed after an hour its immediately moved to the active list, thereby denying the replacement of the logger pages even though it will not be accessed for the next one hour. To avoid this situation an extra flag PG_referenced is added to the page structure. A flag PG_active is used to indicate whether the page is in the active list (PG_active = 1) or its in the inactive list (PG_active = 0). Initially a page entering the LRU will be in the inactive list with flag PG_active = 0 and PG_referenced = 0. Whenever a page is accessed after this PG_referenced flag is set to 1, but the page still remains in the inactive list. If further after this the page is accessed before certain period of time, its moved to the active list with PG_active = 1 and PG_referenced = 0. If its not accessed for a certain period of time when PG_active = 0 and PG_referenced = 1, then again the PG_referenced will be set to 0. If a page is accessed within a certain period of time after it reached the active list (PG_active = 0), then its PG_referenced is set to 1. Now clearly there are 4 accesses needed to move the page to the top of the list.

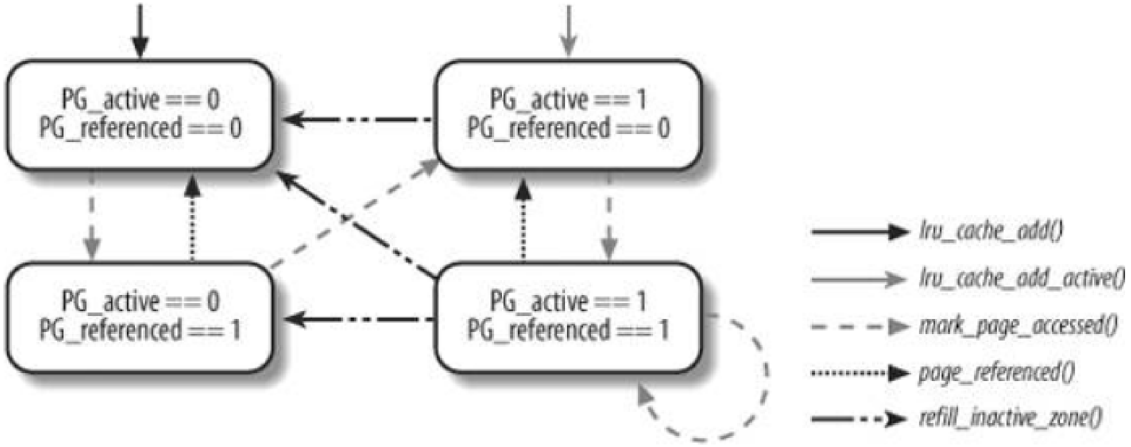
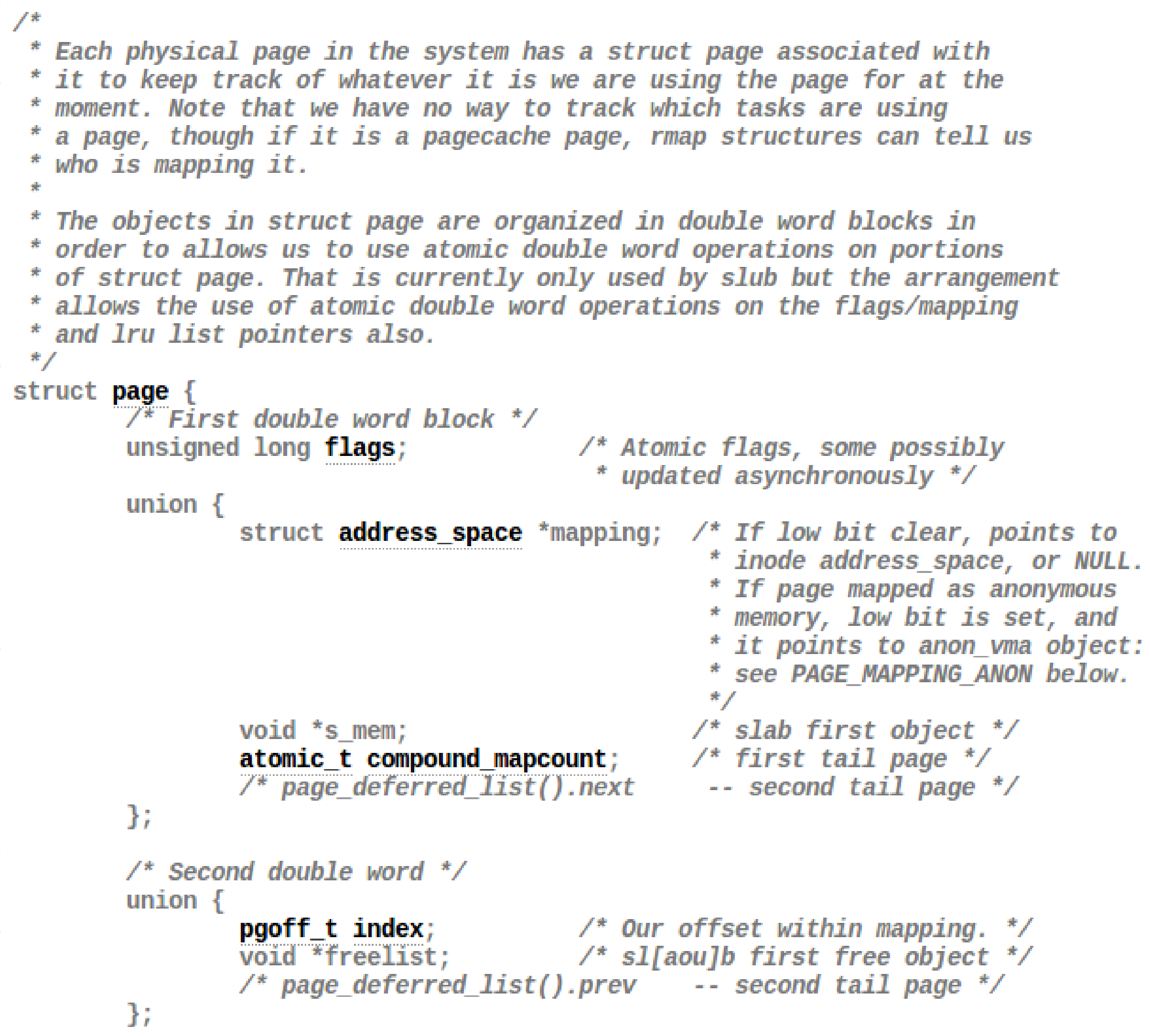


Figure 6 Describing the page state in LRU[15]

4.5 Functions Involved in the Implementation

Default page size used in the Linux kernel is 4kb of memory. Each page in the Linux kernel is identified by a structure which contains all the information necessary to represent a single page in the memory.



```
/*
 * Each physical page in the system has a struct page associated with
 * it to keep track of whatever it is we are using the page for at the
 * moment. Note that we have no way to track which tasks are using
 * a page, though if it is a pagecache page, rmap structures can tell us
 * who is mapping it.
 *
 * The objects in struct page are organized in double word blocks in
 * order to allows us to use atomic double word operations on portions
 * of struct page. That is currently only used by slub but the arrangement
 * allows the use of atomic double word operations on the flags/mapping
 * and lru list pointers also.
 */
struct page {
        /* First double word block */
        unsigned long flags;            /* Atomic flags, some possibly
                                         * updated asynchronously */
        union {
                struct address_space *mapping;  /* If low bit clear, points to
                                                 * inode address_space, or NULL.
                                                 * If page mapped as anonymous
                                                 * memory, low bit is set, and
                                                 * it points to anon_vma object:
                                                 * see PAGE_MAPPING_ANON below.
                                                 */
                void *s_mem;                    /* slab first object */
                atomic_t compound_mapcount;     /* first tail page */
                /* page_deferred_list().next     -- second tail page */
        };

        /* Second double word */
        union {
                pgoff_t index;          /* Our offset within mapping. */
                void *freelist;         /* sl[aou]b first free object */
                /* page_deferred_list().prev    -- second tail page */
        };
```

Figure 7 Page Structure in Linux Kernel[15]

The following section discusses the major functions that are involved in the implementation of the least recently used algorithm. It is necessary to understand how the functions are implemented currently before proceeding to the next implementation.

add_page_to_active_list( )

This function is to add a page to the LRU’s active list. The pages are added to the head of the list and the nr_active flag which denotes the number of pages in the active list is incremented every time after adding a new page [15].

add_page_to_inactive_list( )

This function is to add a page to the LRU’s inactive list. The pages are added to the head of the list and the nr_inactive flag which denotes the number of pages in the inactive list is incremented every time after adding a new page [15].

del_page_from_active_list( )

This function is to remove a page from the active list of the LRU. The flag nr_active is decremented by one, denoting the removal of a page from the active list [15].

del_page_from_inactive_list( )

This function is to remove a page from the inactive list of the LRU. The flag nr_inactive is decremented by one denoting the removal of a page from the inactive list [15].

del_page_from_lru( )

This function is to delete the page from the LRU lists. It first checks the PG_active flag, and depending on the value, removes the page from either the active list or the inactive list and decrements the nr_active or nr_inactive flag accordingly in the zone descriptor [15]. Then it finally clears the PG_active flag

activate_page( )

This function is to move the page in to the active list if the PG_active flag is set and then executes del_page_from_inactive_list( ) to delete the page from the inactive list, then executes add_page_to_active_list( ) to add the page to the active list and finally sets the PG_active flag [15]. These tasks are executed after acquiring the zone’s lru_lock spin lock.

lru_cache_add( )

This function is invoked to add a page to the LRU list if it is not already in the list. The PG_lru flag will be set and the page is added to the inactive list of the particular zone by calling the function add_page_to_inactive_list().

lru_cache_add_active( )

This function is invoked to add a page to the LRU list if it is not already in the list. This function sets the PG_active, PG_lru flag and moves the page to the active list by calling the function add_page_to_active_list() [15].

mark_page_accessed()

This is the important function in the whole implementation of the algorithm. This is called for every access of a page and depending on the flags in page structure, the page can be moved to the active list [15]

4.6 LRU-K Implementation

This project implements the LRU-K algorithm in the Linux kernel 4.4.1. In the implementation of LRU-K choosing a value for K is one of the critical parameter. If the value of K is 1 then its

LRU algorithm. If K = 2 the algorithm remembers the last two references of a page. If K is chosen to be greater than 2, then the algorithm provides only somewhat performance over K = 2 when the access pattern is stable [13] and remembering more references of a page would affect the performance of the algorithm. The algorithm developed in the project takes the value of K to be equal to 2. The algorithm finds the difference in time interval between the last access of a page and the current time. It only considers those pages which has time interval less than a threshold value and K = 2 denotes that the page should have been accessed at-least twice. Whenever the page is accessed for the first time, the page's first reference time is copied to a variable. This page can be chosen as a replacement page since its accessed only once. Whenever the page is accessed for the second time, second reference time is copied to a variable. Now whenever a decision needed to be taken whether to replace this time, the difference between the current time and the second reference time is calculated and if it is greater than a threshold value then its considered for replacement. When the page is accessed for the third time, the second reference time is copied to the first reference time and second reference time is copied with the current time value. All the time values noted here are the time since the computer is booted.

4.7 LFU Implementation

In Least Frequently Used implementation a frequency variable is used to count the number of references of a particular page. The frequency variable associated with the page structure is incremented every time when the page is referenced. The page is considered for replacement whenever the page frequency count is less than a threshold value. Among those pages which has the same frequency, the page which is at the tail of the list is considered for replacement [12].

5. TEST SCENARIOS

All the tests are carried out under the following circumstances:

1) Hardware Specification:

- PC with 2 GB of RAM. A 2 GB of RAM plugged into the personal computer will not always have an exact 2 GB usage memory. It will be always less than that. So its better to use a 4 GB of RAM and then limit the memory to 2 GB in the GRand Unified Bootloader (GRUB) menu.
- Another option is to have a virtual box with 2 GB of RAM. In this case the hyper visor will exactly allocate 2 GB of RAM to the guest operating system.

2) Software Specification:

- Ubuntu 16.04
- Linux Kernel 4.4.1
- A large file of size 3 GB created using dd command
- Graphical User Interface (GUI) was disabled in the Ubuntu operating system, since GUI process consumed around 250 MB of memory while executing. Because of this, the test were not able to properly get executed, since the available free memory always fluctuated.
- Networking is disabled.

3) Before executing each test case, the pages in the current memory is dropped.

4) The readahead feature in the kernel should be disabled. Readahead is a feature which prefetches next few consecutive bytes of memory from a file into the cache, when a file is read, so that the next access for the file can be directly made from the cache instead of the disk [10].

This feature will be enabled by default. The test was initially carried out with this feature enabled, because of that the page faults occurred were found not to be the one expected.

5) A system parameter kernel.page_algorithm defined in the sysctl.h is used to determine which algorithm is currently executing [17].

kernel.page_algorithm = 0 LRU is being executed

kernel.page_algorithm = 1 LRU-K is being executed

kernel.page_algorithm = 2 LFU is being executed

A system parameter kernel.page_threshold is used to set the threshold value for LRU-K and LFU.

6) Page Faults and the execution time are noted for the current algorithm and the system parameter is modified to execute the next algorithm.

7) Finally the page faults and the execution time [11] are taken for all the algorithms and the results are compared with each other.

5.1 Test scenario 1:

This test is to showcase the performance of LRU-K algorithm over the other two algorithms (LRU & LFU). The following steps are executed using a separate Linux process in a sequential manner. The input file of size 3 GB mentioned above is accessed across all the steps in the test.

Steps Involved

1) Read 512 MB of memory from 1 GB to 1.5 GB in the 3GB input file. Since the file is read for the first time, the number of faults should be equal to (512 MB/ 4096) (Default page size) in all the page replacement algorithms.

2) Sleep for 10 seconds

3) Then read the same 512 MB which is read in the first step again. Since the page is already in the cache, all the page replacement algorithms will have zero page faults.

4) Sleep 10 seconds

5) Now read 750 MB of memory from 0 to 750 MB in the 3GB input file. Since this 750 MB is read for the first time, the number of faults should be equal to (750 MB/ 4096) (Default page size) in all the page replacement algorithms.

6) Again read the same 750M of memory for three times without any delay. In case of LRU-K page access difference value will be updated and the pages belonging to the 750 MB will be moved to active list. In case of LRU, reference bit will be set and the pages will be moved to active list as well. In case of LFU, since the threshold is 5, the page will be still in the inactive list.

7) Read 512M of memory (from 1 to 1.5 GB) again. In case of LRU, this page will be in the inactive list before and now the reference bit will be set and it will be moved to active LRU list. In case of LRU-K page access difference value will be updated but the value will be greater than threshold value (10 secs) because the difference between current reference time and last reference will be at least 60 seconds. At this stage, in LRU both 750M and 512M array will be in active list and pages of 512M array will be in head of list since it is recently referred. In LRU-K 750M will be in active list but 512M will be in inactive list. The value of threshold determines whether the 512M will be in the active list or inactive list. Any value less then 60 secs will make the 512M to be in the inactive list, for simplicity the value is chosen to be 10 secs. In case of LFU, both the pages will be in the inactive list. The threshold value (reference count) for LFU is

chosen in such a way that both the pages will be in the inactive list. In this test scenario, threshold value for LFU is 5 counts.

8) Now read first 2GB array for two times(./temp). Since the RAM won't have space for entire 2GB some pages from page cache will be removed to accommodate for new pages. In case of LRU, the pages of 750M array will be in the first place to be removed compared to pages of 512M array. In case of LRU-K it is opposite; pages of 512M will be in first place to be removed since it is in inactive list. In case of LFU, both will be in the inactive list and 750M will be in the tail of list since it is not accessed last and so will be replaced.

9) Now read the first 1GB of memory (0 – 1GB). At this step, LRU and LFU technique should throw large number of faults compared to LRU-K, since the first 750MB (0-750 MB) was replaced in case of LRU and LFU, so a fetch of 0-1GB will throw more page faults. In case of LRU-K a small amount of 0-750 MB only would have been replaced, so there will be less number of page faults.

Results

The results from the previous test shows that LRU-K performs better than LRU and LFU. The test was executed for 10 iterations and in case of LRU the average page fault obtained is 912537. In case of LFU average page fault is 938372 and in case of LRU-K the average page fault is 786787. Average Execution time for LRU, LFU, LRU-K are 96 seconds, 97 seconds and 86 seconds respectively. This is 11% increase in the efficiency in case of LRU-K.

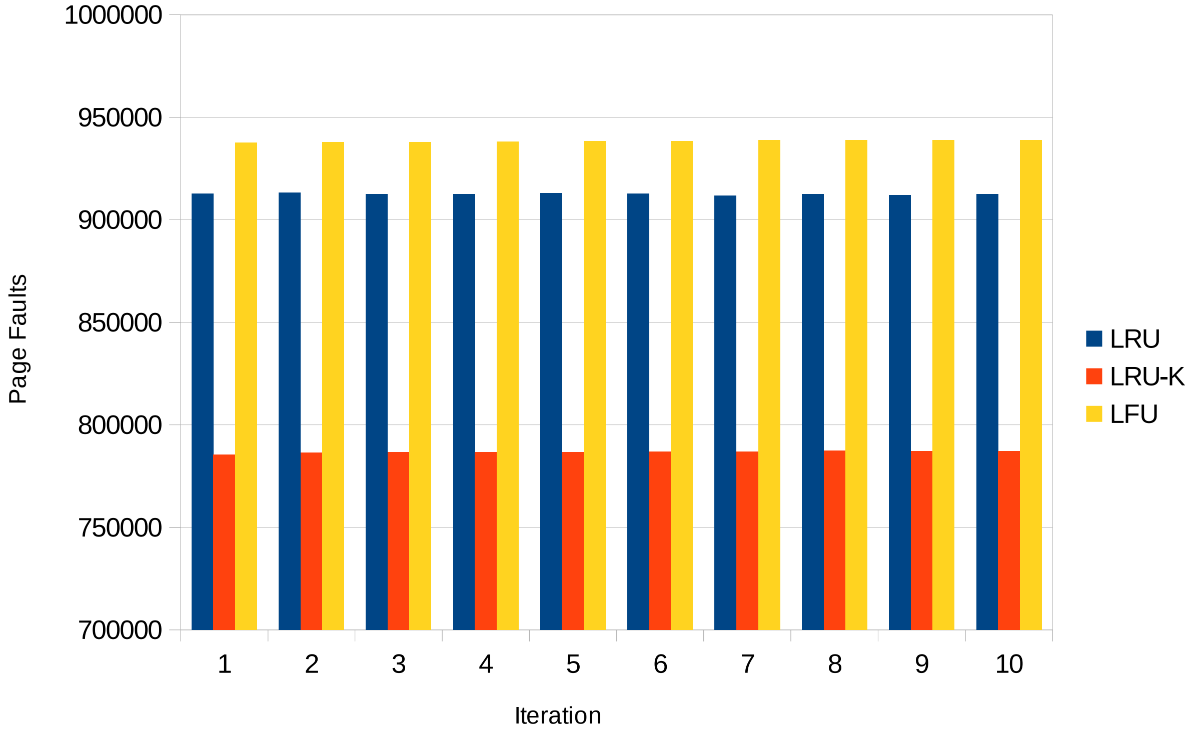


Figure 8 Chart Showing Page Faults of LRU, LRU-K and LFU

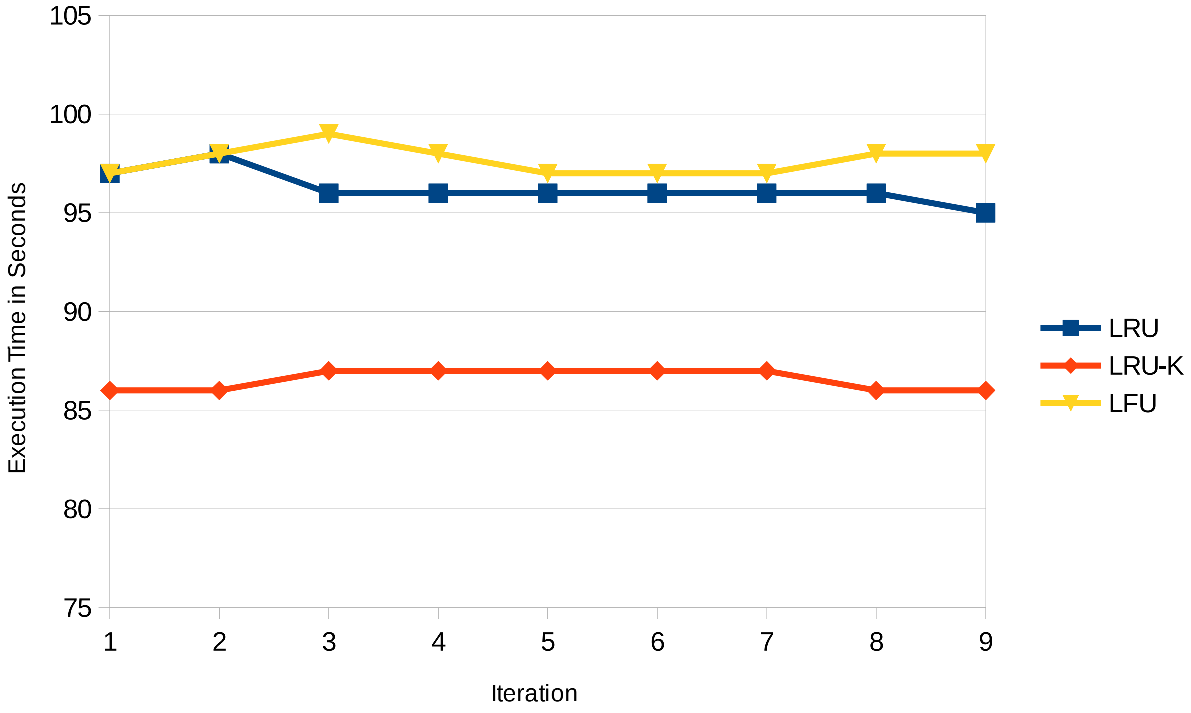


Figure 9 Chart Showing Execution time of LRU, LRU-K and LFU

5.2 Test Scenario 2

This test is to showcase the performance of LRU algorithm over the other two algorithms (LRU-K & LFU). The following steps are executed using a separate Linux process in a sequential manner. The input file of size 3 GB mentioned above is accessed across all the steps in the test.

1) Read 1GB of memory (from 0 to 1GB) from the file for first time. All three algorithms will show same number of page faults.

2) Sleep for 15 seconds

3) Read the same 1GB (from 0 to 1GB) memory again. All three algorithms will show zero page faults

4) Sleep for 15 seconds

5) Read 512MB of memory from 1G to 1.5G for the first time

6) Read 512MB of memory again for four times without a delay. In case of both LRU and LRU-K, 512MB pages will be in active list and the reference bit is set. In case of LFU the pages will be in inactive list since the access count is 2 which is less than the threshold value.

7) Read 1GB of memory again(from 0 to 1G). In case of LRU, reference bit will be set and it is moved to active list. In case of LRU-K, since these array pages are accessed after LRU-K threshold seconds, it won't be moved to active list and it remains in inactive list. The value of threshold determines whether the 1GB pages will be in the active list or inactive list. Any value less then (15 secs + time taken to access 512MB of memory) will make the 1GB to be in the inactive list, for simplicity the value is chosen to be 10 secs. In case of LFU it remains in inactive list as well since the threshold count is chosen to be 5.

8) Read 2GB array (from 0 to 2G) two times. In case of LRU, 512 MB pages will be replaced first to accommodate for new pages and in case of LRU-K and LFU, 1GB pages will be replaced first. So In case of LRU-K and LFU more number of 1GB pages will be replaced.

9) Now read 1GB array of pages again. As expected, LRU-K and LFU will throw larger number of faults compared to LRU. Execution time of LRU should be less than the LFU and LRU-K.

Results

The results from the previous test shows that LRU performs better than LFU and LRU-K. The test was executed for 10 iterations and in case of LRU the average page fault obtained is 726642. In case of LRU-K average page fault is 933426 and in case of LFU the average page fault is

950348. Average Execution time for LRU, LRU-K, LFU are 88 seconds, 102 seconds and 103 seconds respectively. This is a 10% increase in the efficiency in case of LRU.

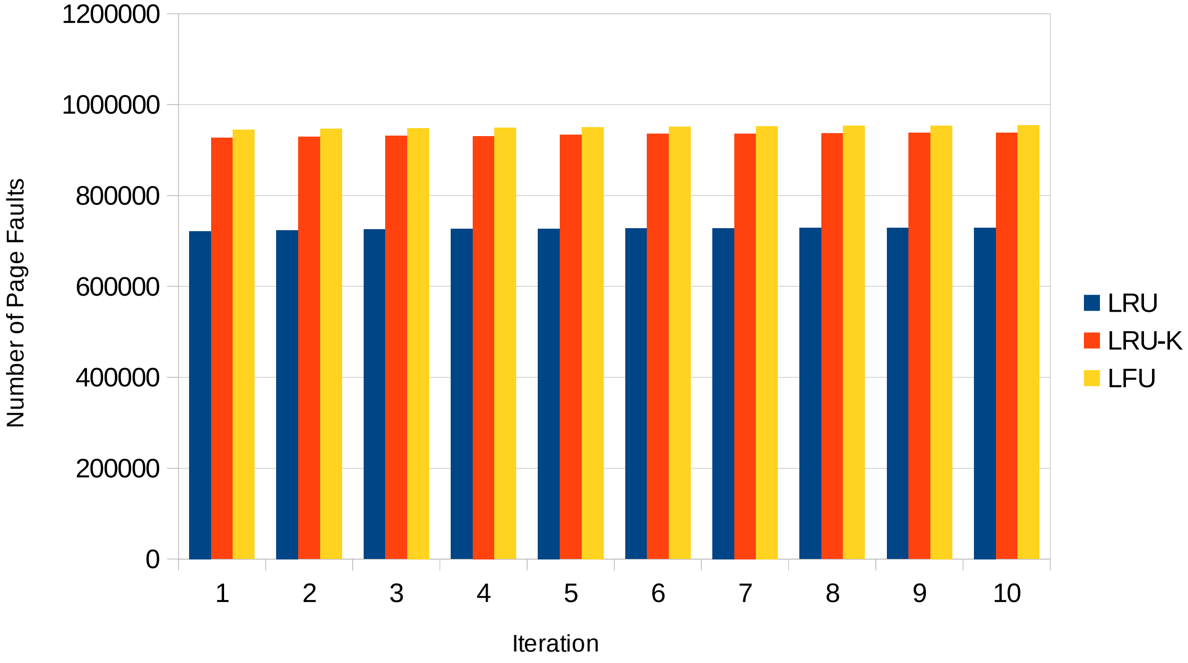


Figure 10 Chart comparing page faults of LRU with LFU and LRU-K

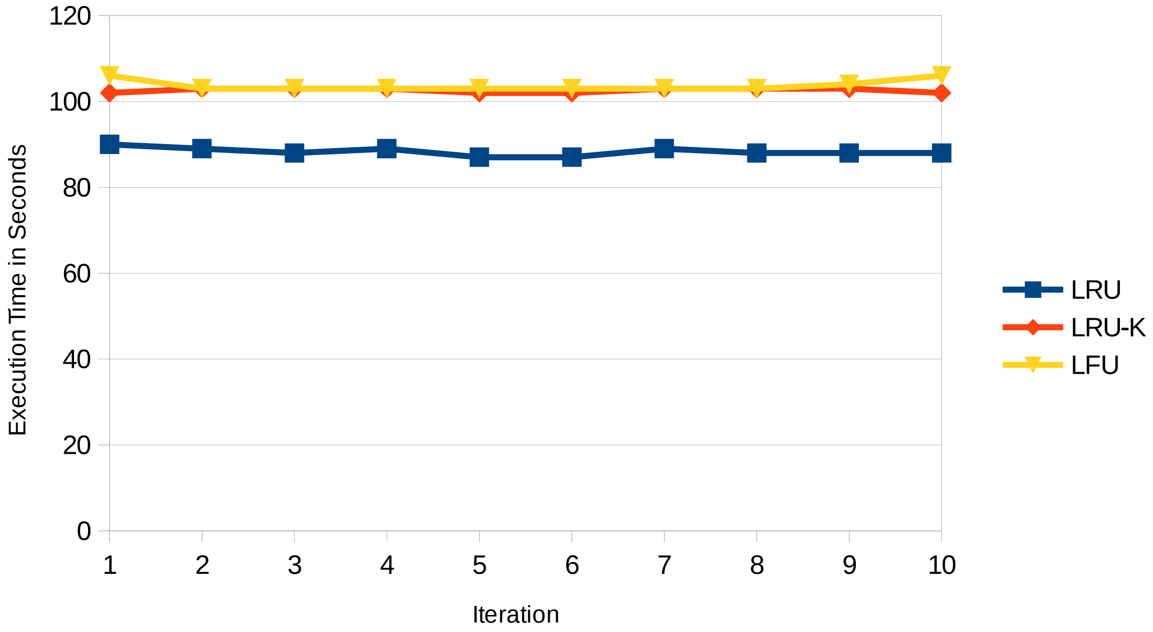


Figure 11 Chart comparing execution time of LRU with LFU and LRU-K

5.3 Test Scenario 3

This test is to showcase the performance of LFU algorithm over the other two algorithms (LRU & LRU-K). The following steps are executed using a separate Linux process in a sequential manner. The input file of size 3 GB is accessed across all the steps in the test. Threshold chosen for LRU-K is 10 secs and for LFU threshold is 3 counts.

Steps Involved

1) Read 512 MB of memory from 1 GB to 1.5 GB in the 3GB input file. Since the file is read for the first time, the number of faults should be equal to (512 MB/ 4096) (Default page size) in all the page replacement algorithms.

2) Sleep for 1 second

3) Then read the same 512 MB which is read in the first step again. Since the page is already in the cache, all the page replacement algorithms will have zero page faults.

4) Sleep 1 second

5) Now read 750 MB of memory from 0 to 750 MB in the 3GB input file. Since this 750 MB is read for the first time, the number of faults should be equal to (750 MB/ 4096) (Default page size) in all the page replacement algorithms.

6) Again read the same 750M of memory for three times without any delay. In case of LFU, since the threshold is 3 and the page is accessed more than thrice the page will be moved to the active list. In case of LRU-K page access difference value will be updated and the pages belonging to the 750 MB will be moved to active list. In case of LRU, reference bit will be set and the pages will be moved to active list as well.

7) Read 512M of memory (from 1 to 1.5 GB) again. In case the pages are referenced 3 times since its not greater than the threshold value which is 3, the pages are not moved to active list. In case of LRU, now the reference bit will be set and it will be moved to active LRU list. In case of LRU-K page access difference value will be updated and the value is less than the threshold value (10 secs) so the pages will be moved to active list as well. Since this test was to showcase the performance of LFU over LRU and LRU-K, the threshold value in LRU-K is chosen in such a way that the 512M of memory is moved to active list as well. Any value larger than the time taken to read 712M of memory (which was around 5 secs) will move the pages to active list. For simplicity the threshold value is chosen to be 10 secs. At this stage, In LFU 750M will be in the active and 512M will be in the inactive list. In LRU and LRU-K, both the pages will be in the active list.

8) Now read first 2GB array for two times(./temp). Since the RAM won't have space for entire 2GB some pages from page cache will be removed to accommodate for new pages. In case of LRU and LRU-K, the pages of 750M array will be in the first place to be removed compared to pages of 512M array. In case of LFU it is opposite, pages of 512M will be in first place to be removed since it is in inactive list.

9) Now read the first 1GB of memory (0 – 1GB). At this step, LRU and LRU-K technique should throw large number of faults compared to LFU, since the first 750MB (0-750 MB) was replaced in case of LRU and LRU-K, so a fetch of 0-1GB will throw more page faults. In case of LFU a small amount of 0-750 MB only would have been replaced, so there will be less number of page faults.

Results

The results from the previous test shows that LFU performs better than LRU and LRU-K. The test was executed for 10 iterations and in case of LRU the average page fault obtained is 952899. In case of LRU-K average page fault is 861541 and in case of LFU the average page fault is 729527. Average Execution time for LRU, LRU-K, LFU are 75 seconds, 70 seconds and 60 seconds respectively. This is a 20% increase in the efficiency in case of LFU.

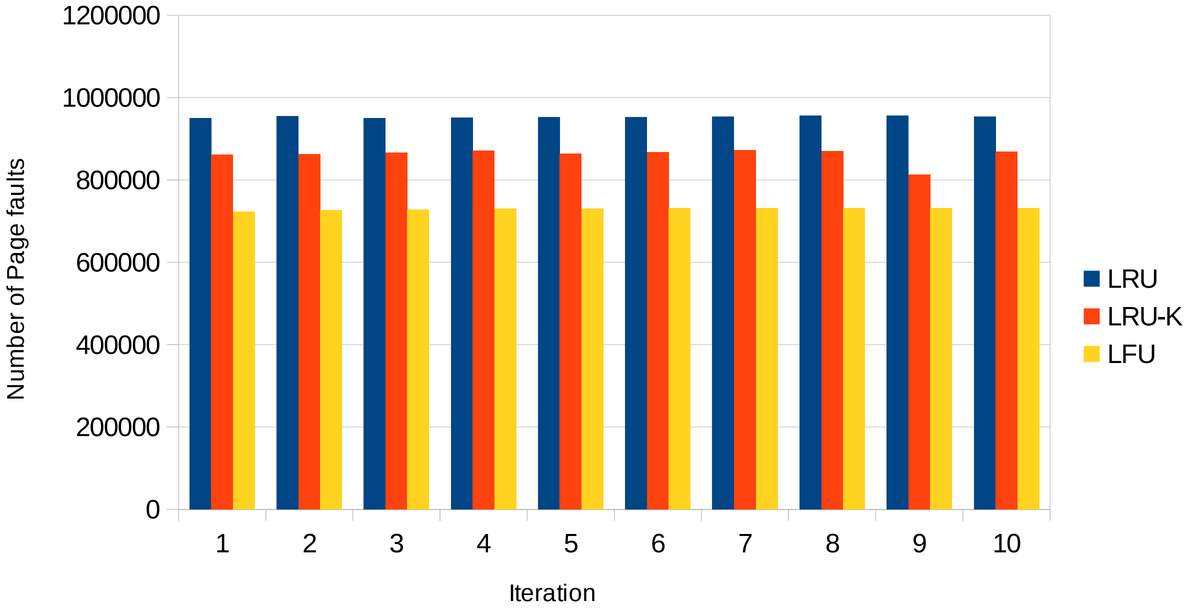


Figure 12 Page Faults of LFU over LRU and LRU-K

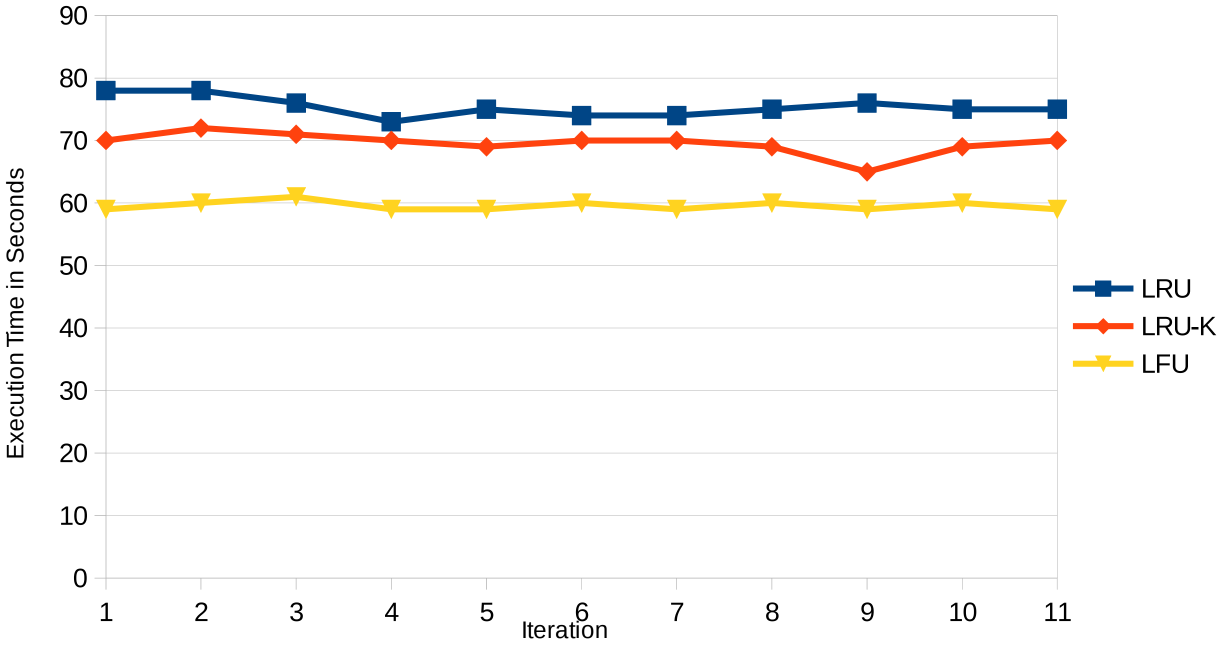


Figure 13 Execution time of LFU over LRU and LRU-K

5.4 Test Scenario 4

The following steps are executed in a sequential manner to prove that LRU-K algorithm performs better than LRU. In this test case instead of using the same input file, the input file to be read is varied for 10 different execution times. For this test case, the free memory available was 1GB before executing the test case.

Steps Involved

1. First read 512MB of memory from the input file starting from 512MB to 1GB.

2. sleep for 10 seconds

3. Then read the same 512 MB of memory again

4. sleep 10 seconds

5. Read another 512 MB of memory starting from 0 to 512MB from the input file.

6. Again read the same 512MB of memory for three times without any delay. In case of LRU-K page heat value will be updated and it will be moved to active list and in case of LRU, reference bit will be set and it will be moved to active list.

7. Now read the 512MB of memory from 512MB to 1GB again. In case of LRU, reference bit is set and it will be moved to active list. In case of LRU-K page heat value will be updated but the page heat value will be less than threshold value because the difference between current reference time and last reference will be at-least 60 seconds. LRU-K threshold value is 10.

At this stage, In LRU both 512MB (512MB-1GB) and 512M (0 – 512MB) array will be in active list and pages of 512M (512MB – 1GB) array will be in head of list since it is recently referred.

In LRU-K (0 - 512MB) will be in active list but pages of 512MB – 1GB memory will be in inactive list.

8. Now read 512MB of memory from 1GB to 1.5 GB of memory from the input file for four times. Since the RAM won't have space for the new 512MB, some pages from page cache will be removed to accommodate for new pages. In case of LRU, the pages of first 512Mb of memory will be in the first place to be removed compared to pages of second 512MB (512MB - 1GB) of memory. In case of LRU-K it is opposite; pages of second 512MB (512MB – 1GB) of memory will be in first place to be removed since it is in inactive list.

9. Now read first 512MB of memory from 0 – 512MB from the input file. At this step, LRU page replacement algorithm should throw large number of faults compared to LRU-K.

Results

This experiment was executed for 10 different input files of the same size 2 GB and found the results to be same for all the files. LRU-K throw less number of page faults when compared to LRU. On average LRU-K throws on average 393216 page faults while LRU throws around 524288 page fault. The average execution time taken by LRU was 42 seconds while LRU-K took 40 seconds, which is 5% increase in the efficiency.

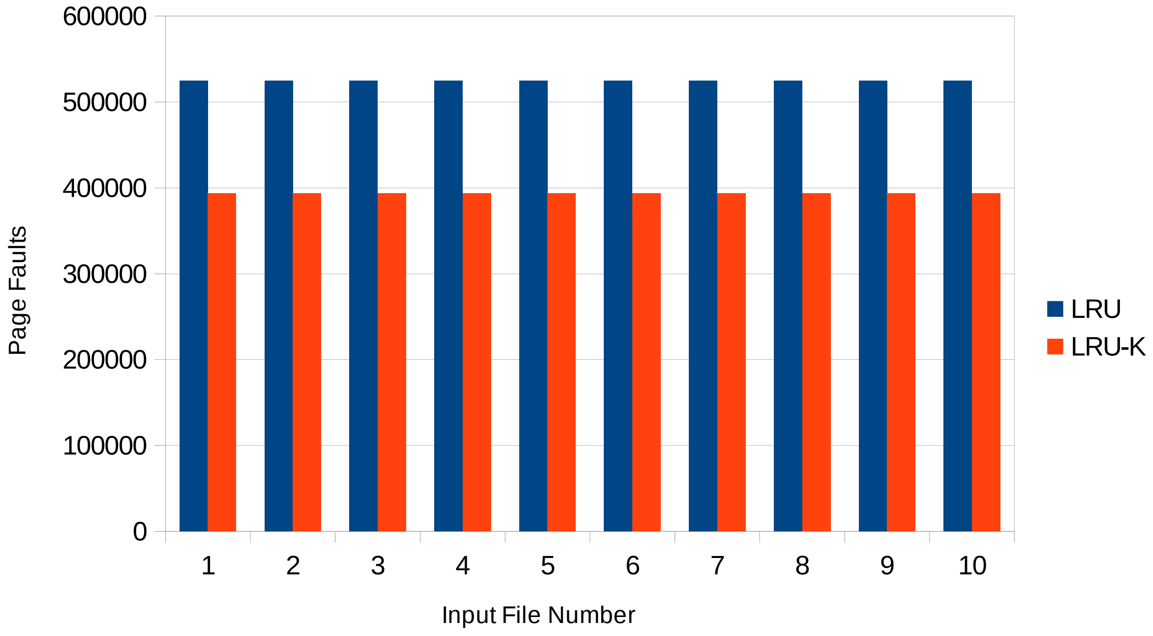


Figure 14 Page faults of LRU-K over LRU on varying input files

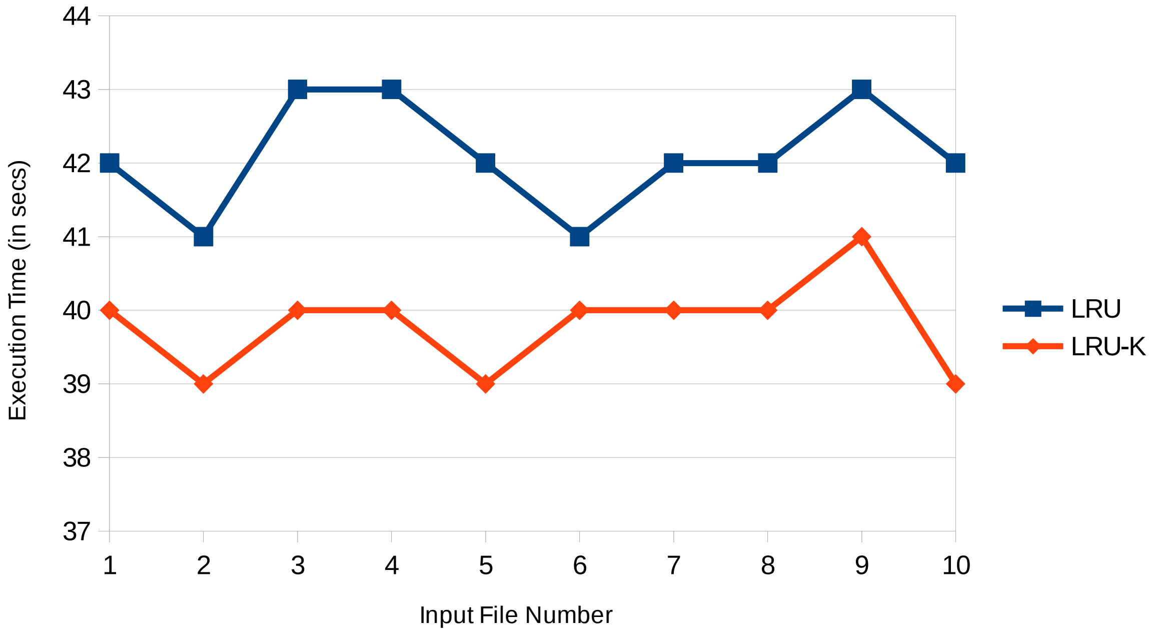


Figure 15 Execution time of LRU-K over LRU on varying input files

6. CONCLUSION

The results obtained from the previous test scenarios proves that the efficiency of a page replacement algorithm depends on the work load at that particular moment and also the parameters used in the algorithms. Many test scenarios can be developed to prove the advantage of one algorithm over another. Its not always easy to predict the work load and also the parameters before in advance in case of the real time systems. So its very difficult to employ a page replacement algorithm which gives always the best result for all the work loads. It is better to employ an approximation algorithm which gives some good results to all the work loads. Least frequently Used (LFU) always considers the number of times the page is used, but a page might be used heavily in the past, but not at all the future, which will always lead to more page faults since the page heavily used in the past will not be replaced at all. Least Recently Used (LRU) and Least Recently Used - K (LRU-K) are the best approximation to the optimal algorithm, since the page which is used recently is likely to be used again. The K value in the LRU-K makes it better suited, since the page which is accessed more than twice (K == 2) recently is more likely to be used again. So the project concludes saying the LRU and LRU-K algorithm are the best choices for all the work loads.

7. FUTURE WORK

This project work suggests to change the page replacement algorithm manually by changing the system parameters in the kernel, whenever the work load changes. There should be a findings/proof that a particular algorithm works good among others for a particular workload and the system administrator needs to change the parameter when the workload executes. This project can be extended to dynamically change the page replacement algorithm by the kernel itself. For this to get implemented the kernel should calculate all the page replacement algorithm’s page faults at all times and switch to an algorithm with the least number of page faults after a scheduled timer. Care should be taken such that the time complexity and the space complexity of the algorithm will not exceed the current algorithm.

8. REFERENCES

[1] Amit S. Chavan, Kartik R. Nayak, Keval D. Vora, Manish D. Purohit and Pramila M. Chawan, “A Comparison of Page Replacement Algorithms”, IACSIT International Journal of Engineering and Technology, Vol.3, No.2, April 2011.

[2] A. S. Tanenbaum and A. S. Woodhull, Operating Systems: Design and Implementation. Prentice - Hall, 1997.

[3] M. Z. Farooqui, M. Shoaib, M. Z. Khan, “A Comprehensive Survey of Page Replacement Algorithms”, International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume 3 Issue 1, January 2014

[4] Shun Yan Cheung, “The second chance page replacement policy”. [Online]. Available: http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/9-virtual-mem/SC-replace.html

[5] N. Meigiddo and D. S. Modha, “Outperforming LRU with an adaptive replacement cache algorithm,” in Computer, vol. 37, no. 4, pp. 58-65, April 2004.

Doi: 10.1109/MC.2004.1297303

[6] N. Meigiddo and D. S. Modha, “ARC: A Self-Tuning, Low overhead Replacement Cache”, IEEE Transactions on Computers, pp. 58-65, 2004

[7] S. Jiang, and X. Zhang, “LIRS: An Efficient Policy to improve Buffer Cache Performance”, IEEE Transactions on Computers, pp. 939- 952, 2005.

[8] Y. Smaragdakis, S. Kaplan, and P. Wilson, “EELRU: Simple and Effective Adaptive Page Replacement”, Proceedings of 1999 ACM SIGMETRICS Conference on Measuring of Computer Systems, May 1999, pp. 122-133.

[9] A. Janapsatya, A. Ignjatovic, J. Peddersen and S. Parameswaran, “Dueling CLOCK: Adaptive cache replacement policy based on the CLOCK algorithm”, Design, Automation and Test in Europe Conference and Exhibition, pp. 920-925, 2010.

[10] S. Ahn, S. Hyun and K. Koh, "Improving Demand Paging Performance of Compressed Filesystem with NAND Flash Memory," *2009 International Conference on Computational Science and Its Applications*, Yongin, 2009, pp. 84-88. doi: 10.1109/ICCSA.2009.37

[11] RTAI - the Real-Time Application Interface for Linux from DIAPM [online]. https://www.rtai.org/

[12] Z. s. Li, D. w. Liu and H. j. Bi, "CRFP: A Novel Adaptive Replacement Policy Combined the LRU and LFU Policies," 2008 IEEE 8th International Conference on Computer and Information Technology Workshops, Sydney, QLD, 2008, pp. 72-79.

[13] E. J. O'Neil P. E. O'Neil and Gerhard Weikum "The LRU-K Page Replacement Algorithm for Database Disk Buffering" Proceedings of the 1993 ACM SIGMOD Conference pp. 297-306 1993.

[14] Donghee Lee, Jongmoo Choi, Honggi Choe, Sam H. Noh, Sang Lyul Min and Yookun Cho, "Implementation and performance evaluation of the LRFU replacement policy," *Proceedings 23rd Euromicro Conference New Frontiers of Information Technology - Short Contributions*, Budapest, 1997, pp. 106-111.

[15] Daniel Bovet , Marco Cesati, Understanding The Linux Kernel, Oreilly & Associates Inc, 2005.

[16] Abraham Silberschatz , Peter Baer Galvin , Greg Gagne, Operating System Concepts, John Wiley & Sons, Inc., New York, NY, 2001

[17] Robert Love, Linux Kernel Development, Addison-Wesley Professional, 2010.

[18] A. S. Sumant, and P. M. Chawan, ―Virtual Memory Management Techniques in 2.6 Linux kernel and challenges, IASCIT International Journal of Engineering and Technology, pp. 157-160, 2010

[19] M. J. Bach, The Design of the UNIX Operating System. Engle-wood Cliffs, NJ: Prentice-Hall, 1986.