Page Replacement Algorithms: A Survey of Performance and Case Studies

Burhanuddin Mal
Elctronics and Telecommunication
Vishwakarma Institute Information of Technology
Pune, India
burhanuddin.22210596@viit.ac.in

Avanti Savji
Elctronics and Telecommunication
Vishwakarma Institute Information of Technology
Pune, India
avanti.22210023@viit.ac.in

Ved Thorat
Elctronics and Telecommunication
Vishwakarma Institute Information of Technology
Pune, India
ved.22211044@viit.ac.in

Vedashri Patil
Elctronics and Telecommunication
Vishwakarma Institute Information of Technology
Pune, India
vedashri.22210642@viit.ac.in

Minal Deshmukh
Elctronics and Telecommunication
Vishwakarma Institute Information of Technology
Pune, India
minal.deshmukh@viit.ac.in

***Abstract*— This system uses paging as part of memory management in a computer's operating system. Page replacement algorithms are responsible for deciding which pages should be replaced when new allocations are required. Paging happens when there is a page fault, and there are insufficient or no free pages available for allocation.**

Keywords— Operating System, Page Replacement Algorithms, Virtual Memory Management, Page Allocation

I. INTRODUCTION

In memory management systems, page replacement is a key concept. When the kernel produces a process using system calls , pages needs to reside in the main memory. These pages, therefore, need pages to be kept in the main memory This process will first identify if a required page is not available in the main memory, a condition called as page fault.

A process has to identify the missing page on the disk if page fault is present. Then page will be allocated into that free space in the main memory if there exists free space in the memory. However, if there is no available space, a page replacement must be performed using the implemented page replacement algorithm.[1] The removed page is then written back to disk. Once the page is successfully transfered into the main memory, the process can resume its operation, as it can now access the necessary page.

A. When page fault occurs?

A page fault occurs when a page contain preferred information or instruction is searched for in translation lookaside buffer (TLB) or page tables and is found absent from the main memory. [2]

B. Role of page replacement algorithm?

Since the main memory is limited in size and smaller compared to the primary storage, the role of page replacement is to select the optimal page to remove from memory when a page fault occurs. This allows the operating system to replace it with a new page from the disk containing the necessary instructions. [3]

C. Need of page replacement algorithm :

An efficient page replacement strategy can reduce the cost of page faults, improving system performance.[12] A high number of page faults can consume resources as the system spends more time paging in and out, rather than executing tasks, eventually leading to system overload. [3]

II. PAGE REPLACEMENT ALGORITHMS

A. Optimal Page Replacement

At the time memory is full, the optimal approach for page replacement is to evict the page which will not be referenced for prolonged duration ahead. This strategy ensures that the number of page faults are minimal by always retaining the pages that will be needed sooner. [1] However, this scheme is only feasible to implement during a second, identical run of the program, where page usage patterns have been recorded in the first run.[10] In practical scenarios, especially in applications that involve external inputs, the operating system cannot foresee which pages will be accessed next, as the timing and content of inputs can vary widely, altering the access patterns.

Despite this limitation, the optimal algorithm—referred to as OPT or MIN—serves as an important theoretical benchmark. It provides a standard for comparing the efficiency of real-world page replacement algorithms. Since OPT assumes perfect knowledge of future events, it is impossible to implement in practice, but its conceptual framework helps in assessing how well practical algorithms approximate this ideal performance, such as Least Recently Used (LRU) or First-In First-Out (FIFO) algorithms.[4]

$$g(S, t, r_{t+1}) = \begin{cases} (S, t), & if\ r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, t+1), & if\ r_{t+1} \notin S, \end{cases}$$

Figure 1: Optimal PGA

The page which takes maximum time until its next reference among all pages in set S is chosen for replacement. The decision is based solely on the time of the next reference, and control state is fully described by t. In this and the subsequent algorithms, the size of set S consists of m pages.[11, 17]

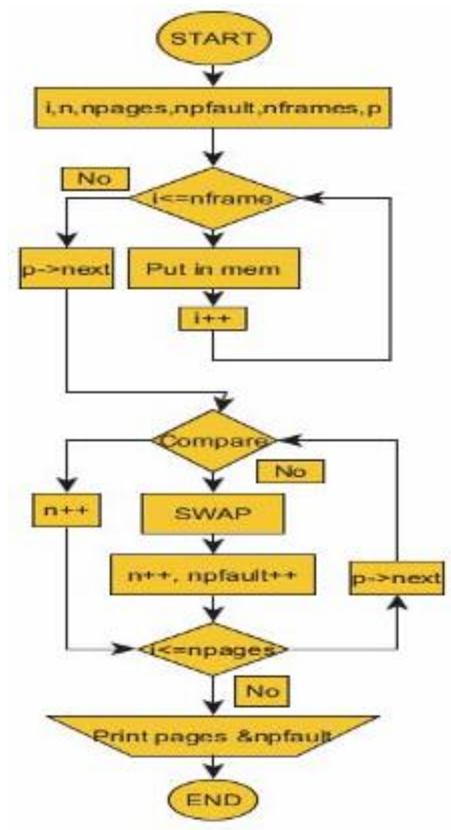

Figure 2: Optimal Page Replacement Algorithm

B. Not Recently Used (NRU)

In the Not Recently Used (NRU) is a page replacement algorithm, pages are categorized into four classes based on their reference and modification status.[4, 25]

1) Class 0 includes pages which neither been referenced nor modified.[13]

2) Class 1 consists of pages which not been referenced but have been modified.[13]

3) Class 2 includes pages which have been referenced but not modified.[13]

4) Class 3 includes pages which have both been referenced and modified[13].

Which page needs to be evicted, NRU selects a random page from the lowest-numbered class that contains pages, prioritizing pages that are less likely to be needed soon or expensive to write back to disk. This approach aims to minimize page faults by favoring unreferenced pages for replacement and by reducing the overhead of writing modified pages back to disk when unnecessary.[26]

Although NRU is simple and efficient to implement, it is an approximation of the more sophisticated Least Recently Used (LRU) algorithm. Unlike LRU, which tracks exact usage, NRU operates by periodically resetting the reference and modification bits, making it less precise but still effective in balancing performance and overhead.[15]

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & if\ r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & if\ r_{t+1} \notin S, \end{cases}$$

Figure 3: Not Recently Used (NRU) PGA

Let y be a random page selected from the lowest class that contains pages. The control state is defined as a collection of classes [16]

$$q_t = \{C_{0t}, C_{1t}, C_{2t}, C_{3t}\}. \qquad (1)$$

If $C_{0t} = \emptyset$ and $C_{1t} \neq \emptyset$, then $y \in C_{1t}$, and the next state is

$$q_{t+1} = \{C_{0t+1}, C_{1t+1}, C_{2t}, C_{3t}\}, \qquad (2)$$

where $C_{0t+1} = \{r_{t+1}\}$ and $C_{1t+1} = C_{1t} \cup \{r_{t+1} \setminus y\}$

C. First-In, First-Out (FIFO)

One of the simplest yet widely used page replacement methods is the First-In, First-Out (FIFO) algorithm. In this method, pages in memory are organized in a list where the most recently added page is placed at the beginning, and the oldest one stays at the end. [5, 20] If the requirement arises for eliminating a page, the oldest page which is at the tail is removed and the new page inserted at the head of the list. It is simply this mechanism that gives rise to the property of the page which has been in memory the longest being replaced. [18]

An alternative implementation of FIFO is the Clock algorithm, which arranges pages in a circular list, resembling a ring. A pointer moves around the ring to track page replacements. When a page must be evicted, the page currently pointed to by the pointer is replaced, and the new page is inserted in its position.[21] After replacement, the pointer moves ahead to the next page in the cycle. The Clock algorithm is superior to simple FIFO, because it permits to use page usage information of pages with reference bits, that makes it able to avoid eviction of recently referenced pages by checking and resetting them. That makes it a more practical variation of FIFO for utilization, since the number of unnecessary swaps of pages is decreased.[22]

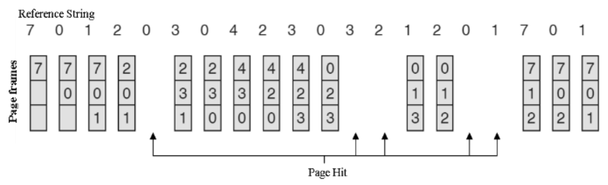

Figure 3: First-In, First-Out (FIFO)

D. *Least Recently Used (LRU)*

A page that was accessed recently is more likely to be needed again soon, whereas a page that has not been used for a long time is less likely to be needed in the immediate future. This is the basis of the (LRU) algorithm and can be implemented by maintaining a sorted list of all pages in memory. [8, 23]

1. The list is ordered by the last time each page was referenced; more frequently it is called the LRU stack.

2) That is to say, it means that this position of each and every page in the list must be updated with every access to the page about its recent access.[24]

But still the implementation of LRU directly is costly. In fact overheads occur due to the constant updating of the position of pages after every access or clock tick involving sort and reordering of the list. Another limitation is that the algorithm is only an approximation: it cannot distinguish between two pages unless they happened to be referenced in the exact same tick of clocks. With this amount of complexity and the cost of frequent updates, pure LRU is rarely used in practice, especially when more efficient approximations such as the Clock algorithm are available and frequently used in real systems. [7, 21, 22]

$$g(S, q_t, r_{t+1}) = \begin{cases} (S,\ q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y_m\},\ q_t + 1), & \text{if } r_{t+1} \notin S, \end{cases}$$

Figure 4: Least Recently Used (LRU)

Let $\mathbf{y}_m$ be the page that is used the least recently in set **S**. The control state is defined as

$$\mathbf{q}_t = (\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \ldots, \mathbf{y}_m), \qquad (3)$$

where the resident pages are ordered by their most recent reference, with $\mathbf{y}_1$ being the most recently referenced page and $\mathbf{y}_m$ being the least recently referenced.[24]

E. *Adaptive Replacement Cache (ARC)*

ARC is a page replacement algorithm. It is designed by the researchers at IBM's Almaden Research Center. This algorithm tracks pages which recently and frequently used while providing information about how long it has been since a page was evicted.[27] The cache is broken down into two LRU lists: L1 contains all the pages accessed only once recently, and L2 contains pages that have been accessed at least twice. L1 captures short-term utility (recency), while L2 reflects long-term utility (frequency). [5, 28]

These groups are then classified into two parts: top cache entries and bottom ghost entries. L1 is divided into T1, which includes the most recent cache entries, and B1 which includes the entries that have just been retrieved from T1. Similarly, L2 consists of T2, the category of more frequent entries, and B2, whose contents consisted of more recent entries that had been taken out from T2. Active cache is a union of T1 and T2, B1 and B2 are inaccessible caches, or ghost lists, which help the algorithm in understanding recently removed entries and modifying its action accordingly. These ghost lists store only metadata, not the actual pages. The entire cache directory is composed of four LRU lists:[29, 30]

1) *T1: recently accessed cache entries*
2) *T2: frequently accessed entries in the cache, which have been at least accessed once*
3) *B1: recently evicted entries from T1 but still tracked*
4) *B2: recently evicted entries from T2[29]*

Assuming that c denotes the size of the cache then

$$|T1+T2| = c, \qquad (4)$$

and if $|T1|=p$,

$$\text{then } |T2|=c-p \qquad (5)$$

ARC continuously adjusts the value of p based on whether recency or frequency is going to dominate in the workload. When recency is important, then p is incremented to give more space to T1. If frequency is important then p is decremented thus giving more space to T2. The total size of the cache directory $|L1+L2| = 2c$.

For a fixed p, the replacement process works as follows:

5) *If* $|T1| > p$, *replace the LRU page in T1.*
6) *if* $|T1| < p$ *then, replace LRU page in T2*
7)*if* $|T1| = p$ *and missed page is in B1, replace the LRU page in T2*
8)*if* $|T1| = p$ *and missed page is in B2, replace the LRU page in T1.[28, 29, 30]*

The adaptation of p depends on the following principle: a hit in B1 indicates the importance of recency, so *p* should increase to allocate more space to T1. Conversely, a hit in B2 highlights the relevance of frequency, prompting *p* to decrease, giving more space to T2. The size adjustment of *p* is proportional to the relative sizes of B1 and B2.[28]

F. *CLOCK with Adaptive Replacement (CAR)*

CAR is an algorithm that combines the adaptive strategy of ARC and the effectiveness of CLOCK. It manages four doubly linked lists, T1, T2, B1, and B2. T1 and T2 are implemented as structures of CLOCK while B1 and B2 as simple LRU lists. The structure overall is a synonym for ARC. In T1 and T2-the cache-pages there is also reference bit which can be set or cleared. [5, 31]

The lists are defined as follows:

*1) **T1 and B1**: T1 and B1 hold pages that have been referenced only once since their last removal from any of the lists (T1, T2, B1, B2) or pages that have not been referenced at all.*

*2) **T2 and B2**: T2 and B2 are caches which store pages that have been referenced a multiple of times since their last eviction from any of the lists (T1, T2, B1, B2).[21, 32]*

Two important constraints on the sizes of T1, T2, B1, and B2 are:

1) ∣ T1 ∣ + ∣ B1 ∣ ⩽c, where c is the cache size. T1 and B1 represent recency. The size of these lists changes as Recently accessed or highly accessed pages keep changing. This reduces the chances of a page which is accessed only once from filling up the entire cache directory (which is limited to size 2c). In case the size of T1 and B1 becomes huge then it is a sign that the recently accessed pages are no longer referred to and hence, the recency information stored becomes ineffective. This also, therefore implies either that frequently accessed pages are being reused or novel pages are being accessed. [6, 32]

2) ∣T2∣+∣B2∣≤2c. When accessing a small set of pages frequently and there are no references being formed, then its directory in the cache will carry mainly the frequency information of those pages.

G. Belady's MIN

Belady's MIN algorithm, also referred to as the clairvoyant algorithm, is theoretically the most optimal page replacement strategy. Its goal is to remove the page that won't be needed for the longest period, ensuring the lowest page fault rate possible.[7, 33] Nonetheless, the fundamental disadvantage of this algorithm is that it assumes the availability of future page requests, which cannot be done in real life. Therefore, it is not applicable in practice.[7]

Although it is not applicable in practice, Belady's MIN algorithm has still a functional importance in theory. For instance, it is used as standard to compare the performance of other replacement policies and in particular those such as LRU, LFU, or CLOCK. This way, when using MIN as a comparison, researchers are able to approximate to what extent the performance of the given algorithm reaches its optimal performance. [7, 34] In simulation studies, it also allows obtaining the worst case of page fault rates and thus enables the system designers to study the behavior of different algorithms layers used in the system under different workloads and operating conditions.[31]

Moreover, Belady's MIN algorithm has revealed page replacement anomalies, such as Belady's anomaly, which occurs when an increase in the number of page frames results in more page faults for some algorithms A situation which was not seen with MIN. While it remains a theoretical ideal, MIN's insights are foundational in the development of efficient memory management techniques.

III. SUMARRY

Table 1: Page Replacement Algorithms

Algorithm	***Description***	***Features***
Optimal Page Replacement (OPT)	Evicts the page that will not be referenced for the longest time in the future.	Used as a theoretical benchmark for comparing other algorithms.
Not Recently Used (NRU)	Pages are classified into 4 categories based on their reference and modification status. Evicts a random page from the lowest-numbered class.	Periodically resets reference and modification bits.
First-In , First-Out (FIFO)	Evicts the oldest page in memory.	FIFO variant (Clock algorithm) avoids evicting recently accessed pages.
Least Recently Used (LRU)	Evicts the least recently used page.	Evicts the least recently used page.
Adaptive Replacement Cache (ARC)	Tracks both recency and frequency, maintaining two LRU lists: one for recent pages (T1) and one for frequent pages (T2).	Uses ghost lists (B1 and B2) to track evicted pages for adaptive behavior.
Clock with Adaptive Replacement (CAR)	Combines the efficiency of CLOCK with ARC's adaptive strategy.	Uses CLOCK structures with reference bits and adaptive size for recency/frequency.
Belady's MIN Algorithm	Evicts the page that will not be needed for the longest time in the future.	Serves as a theoretical benchmark; identifies anomalies like Belady's anomaly.

IV. CASE STUDY: LEAST RECENTLY USED (LRU) ALGORITHM

A. Objective:

This case study evaluates the practical performance of the Least Recently Used (LRU) page replacement algorithm by comparing it against Optimal Page Replacement (OPT) and First-In, First-Out (FIFO) algorithms. The primary focus is on minimizing page faults in a simulated environment using a fixed sequence of page references.

B. Simulation Setup:

- **Workload**: A predefined sequence of page references: [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0].
- **Page Frames**: A total of 3-page frames were allocated for the simulation.
- **Metric**: The number of page faults recorded during the execution of the workload.

C. Results:

1) **Optimal Page Replacement (OPT):**
The OPT algorithm recorded 9-page faults during the simulation, demonstrating the theoretically best performance by replacing the page that would not be used for the longest duration in the future. However, its reliance on future knowledge of page references renders it impractical for real-world implementation.[18]

2) **Least Recently Used (LRU):**
The LRU algorithm achieved 12-page faults, closely approximating the performance of the OPT algorithm. By replacing the least recently used page upon a page fault, LRU showcased its adaptability to real-time memory access patterns, making it a feasible alternative for practical applications. [21, 22]

3) **First-In, First-Out (FIFO):**
The FIFO algorithm experienced 15-page faults, significantly more than both OPT and LRU. Its "oldest page first" replacement policy introduced inefficiencies and exhibited Belady's Anomaly, where an increase in the number of allocated frames paradoxically resulted in a higher number of page faults in certain scenarios.[20]

D. Analysis:

The results demonstrated that LRU effectively minimizes page faults by leveraging recent access patterns, making it significantly better than FIFO and comparable to OPT in performance. The LRU algorithm's trade-off lies in its computational complexity, as it requires tracking the order of recent page accesses, which adds overhead compared to simpler algorithms like FIFO.

V. CASE STUDY: ADAPTIVE REPLACEMENT CACHE (ARC) ALGORITHM

A. Objective:

This case study evaluates the performance of the Adaptive Replacement Cache (ARC) algorithm compared to Least Recently Used (LRU) and First-In, First-Out (FIFO) algorithms. The focus is on its ability to dynamically adapt to varying workloads by balancing recency and frequency of page references.

B. Simulation Setup:

- **Workload**: A mixed sequence of page references: [1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5].
- **Page Frames**: A total of 3-page frames were used for the simulation.
- **Metric**: The primary metric was the number of page faults encountered. Additionally, the algorithm's adaptability to recency and frequency was observed.

C. Results:

1) **Adaptive Replacement Cache (ARC):**
The ARC algorithm achieved 9-page faults, outperforming both LRU and FIFO. By dynamically adjusting its cache allocation between recently accessed pages (T1) and frequently accessed pages (T2), ARC effectively balanced recency and frequency. This adaptability enabled it to maintain optimal performance across varied workload patterns. [27, 28]

2) **Least Recently Used (LRU):**
The LRU algorithm recorded 10-page faults, performing well in scenarios where recency was the dominant access pattern. However, it struggled to account for the reuse of frequently accessed pages, leading to additional page faults compared to ARC.[21, 22[

3) **First-In, First-Out (FIFO):**
The FIFO algorithm experienced 12-page faults, the highest among the three algorithms. Its rigid "oldest page first" replacement policy led to inefficiencies, including the unnecessary eviction of pages still in active use, which negatively impacted its performance.[20]

D. Analysis:

The ARC algorithm outperformed LRU and FIFO due to its adaptive nature. By dynamically dividing cache space between recently and frequently used pages, ARC managed to reduce unnecessary evictions and page faults. In contrast, LRU struggled with recognizing frequently used pages, while FIFO's fixed eviction policy led to poor performance.

VI. CONCLUSION

The objective of this study was to determine which page replacement algorithm delivers optimal system performance. An efficient page replacement strategy helps minimize page faults during execution, reduces input/output operations, and greatly enhances overall system performance. Since time is a crucial factor for system efficiency, lowering the number of page faults contributes significantly to improved performance.

A. *Optimal Page Replacement Algorithm (OPT/MIN):*

- **Performance**: The Optimal Page Replacement algorithm makes optimistic choices of pages which will not be accessed for the longest time in the future. It is ensured that minimum page faults are generated.
- **Feasibility**: While it provides the best theoretical performance, it is impractical for real-world systems because predicting future page access patterns is impossible.
- **Usage**: Its main purpose is to serve as a benchmark for evaluating the performance of other page replacement algorithms.

B. *First-In, First-Out (FIFO):*

- **Performance**: FIFO tends to perform poorly, especially as the number of pages increases. It frequently leads to more page faults (degenerates) because it replaces the oldest page, regardless of its future use.
- **Issue**: FIFO can make inefficient decisions, such as evicting a page only to bring it back shortly after, which causes unnecessary disk I/O operations.
- **Conclusion**: Due to its tendency to make poor replacement choices, FIFO often results in higher page fault rates compared to other algorithms.

C. *Least Recently Used (LRU):*

- **Performance**: LRU is a more practical and efficient algorithm compared to FIFO, as it attempts to approximate the behavior of the Optimal algorithm by replacing the least recently used page.
- **Advantage**: It significantly reduces the number of pages faults and generally performs close to the Optimal algorithm in practical scenarios.
- **Conclusion**: LRU is the better choice for real-world implementations where frequent page replacements occur, balancing performance and simplicity.

D. *Not Recently Used (NRU):*

- **Performance**: NRU is a simpler, less precise approximation of LRU. It sorts pages into classes according to their reference and modification status, preferring pages from lower-priority classes for replacement.
- **Feasibility**: NRU is less accurate than LRU but easier to implement and still fair about page faults and overhead writing modified pages back to disk.
- **Conclusion**: NRU should be suitable for systems where simplicity outweighs the fine-tuned The accuracy of LRU.

E. *Adaptive Replacement Cache (ARC):*

- **Performance**: ARC dynamically and automatically change its behavior according to the characteristics of workload from recency versus frequency for optimizing cache performance.
- **Advantage**: It keeps track of recently as well as frequently accessed pages along with using ghost lists to place metadata of pages replaced out of the cache thereby easily adapting a wide variety of access patterns.
- **Conclusion**: ARC performs well in diverse workloads and is highly adaptive, making it a strong candidate for modern systems where workloads may vary over time.

F. *Clock with Adaptive Replacement (CAR):*

- **Performance**: It employs the adaptive characteristics of ARC while enhancing implementation practices of the CLOCK scheme. In this regard, it keeps several lists to consider the recently accessed pages and frequently accessed pages.
- **Advantage**: CAR is more efficient than typical ARC due to the implementation of policies akin to the CLOCK which help reduce the burden LRU incur.
- **Conclusion**: Not only does CAR demonstrates a extreme level of adaptation, but also can be implemented at a reasonable cost which makes it ideal for any system with varying access patterns and great expectations in performance.

G. *Belady's MIN (Clairvoyant Algorithm):*

- **Performance**: MIN is theoretically the most optimal algorithm, as it evicts the page that will not be used for the longest time, guaranteeing the lowest page fault rate.
- **Feasibility**: Due to its reliance on future knowledge, MIN is impractical for real-world use but serves as a key theoretical benchmark.
- **Conclusion**: While MIN cannot be implemented in real systems, it helps evaluate the effectiveness of other algorithms and provides a lower bound on page fault rates in simulations.

H. *Best Algorithm:*

The Optimal Page Replacement Algorithm (OPT/MIN) is the best in terms of performance because it guarantees the fewest page faults. However, it is impractical for real-world use due to the need for future knowledge.

For practical implementation, Least Recently Used (LRU) is generally the best, as it closely approximates the performance of the Optimal algorithm.

I. *Moderate Algorithm:*

The Clock Algorithm (or its variants like CLOCK with Adaptive Replacement (CAR)) is considered a moderate algorithm. It strikes a balance between performance and simplicity by approximating LRU but with lower overhead. While not as efficient as LRU, it performs better than FIFO and is more practical to implement in many systems.

J. *Worst Algorithm:*

First-In, First-Out (FIFO) is considered the worst due to its tendency to result in higher page faults, especially under increasing workloads. It can make inefficient eviction decisions, such as removing pages that are still needed soon.

REFERENCES

[1] Thakkar, Binita & Padhy, Rabi. (2024). Comparative Analysis of Page Replacement Algorithms in Operating System. Journal of Emerging Technologies and Innovative Research. 11. 7.

[2] Rexha, Genta & Elmazi, Erand & Tafa, Igli. (2015). A Comparison of Three Page Replacement Algorithms: FIFO, LRU and Optimal. Academic Journal of Interdisciplinary Studies. 4. 10.5901/ajis.2015.v4n2s2p56.

[3] Shastri, Shourab, Anand Sharma, and Vibhakar Mansotra. "Study of Page Replacement Algorithms and their analysis with C#." The International Journal Of Engineering And Science (IJES) 5.1 (2016).

[4] Tingare, Bhagyashree A., and Vaishali L. Kolhe. "Analysis of Various Page Replacement Algorithms in Operating System."

[5] Saini Mohit, and Manju Verma. "A Study of Page Replacement Algorithms." International Journal of Research Fellow for Engineering Volume 4, Issue 1.

[6] Kumari, Juhi, et al. "A Comparison of Page Replacement Algorithms: A Survey." International Journal of Scientific and Engineering Research, Volume 7, Issue 12, December-2016 57 ISSN 2229-5518.

[7] Chawan, Pramila. (2011). A Comparison of Page Replacement Algorithms. International Journal of Engineering and Technology. 3. 10.7763/IJET.2011.V3.218.

[8] Farooqui, Mohd Zeeshan, et al. "A Comprehensive Survey of Page Replacement Algorithms." International Journal of Advanced Research in Computer Engineering and Technology (IJARCET) Volume 3 Issue 1, January 2014.

[9] Saxena, Anvita. "A Study of Page Replacement Algorithms." International Journal of Engineering Research and General Science Volume 2, Issue 4, June-July, 2014 ISSN 2091-2730.

[10] Medak, Jogamohan & Gogoi, Partha. (2020). Critical Scrutiny of Page Replacement Algorithms: FIFO, Optimal and LRU. International Journal of Innovative Technology and Exploring Engineering. 9. 345-348. 10.35940/ijitee.J7553.0891020.

[11] Madisetti, Prashanth & Truong, Dan & Yallapragada, Srisailendra. (2013). Identifying replacement memory pages from three page record lists.

[12] Tsai, Hong-Bin & Lei, Chin-Laung. (2017). A page replacement algorithm based on frequency derived from reference history. 1522-1527. 10.1145/3019612.3019737.

[13] Ramachandran, Rajesh & Paulson, Hitha. (2017). Page Replacement Algorithms – Challenges and Trends. International Journal of Computer & Mathematical Sciences. 6. 112.

[14] Akbari-Bengar, Davood & Ebrahimnejad, Ali & Motameni, Homayun & Golsorkhtabar, Mehdi. (2020). A page replacement algorithm based on a fuzzy approach to improve cache memory performance. Soft Computing. 24. 10.1007/s00500-019-04624-w.

[15] Lo, Chia-Tien & Srisa-an, Witawas & Chang, J.Morris. (2001). A study of page replacement performance in garbage collection heap. Journal of Systems and Software. 58. 235-245. 10.1016/S0164-1212(01)00041-3.

[16] Rathod, Vasundhara & Chim, Monali & Chawan, Pramila. (2013). A Survey Of Page Replacement Algorithms In Linux. International Journal of Engineering Research and Applications (IJERA). 3. 1397-1401.

[17] Aho, Alfred & Denning, Peter & Ullman, Jeffrey. (1971). Principles of Optimal Page Replacement. J. ACM. 18. 80-93. 10.1145/321623.321632.

[18] Research Publish Journals. "Page Replacement Algorithms." Research Publish. Accessed November 27, 2024.

[19] Chavan, Amit S., et al. "A comparison of page replacement algorithms." International Journal of Engineering and Technology 3.2 (2011): 171.

[20] Sandeep N. Bhatt ,Fan R.K. Chung, F.Thomas Leighton, Arnold L. Rosenberg

[21] Kavar, Chandu & Parmar, Shaktisinh. (2013). Improve the performance of LRU page replacement algorithm using augmentation of data structure. 1-5. 10.1109/ICCCNT.2013.6726496.

[22] Neil, Elizabeth & Neil, Patrick & Weikum, Gerhard. (1999). An Optimality Proof of the LRU-K Page Replacement Algorithm. Journal of the ACM, v.46, 92-112 (1999). 46. 10.1145/300515.300518.

[23] Gupta, Ruchin & Teotia, Narendra. (2013). Least Recently Used Page Replacement using Last Use Distance (LRUL). International Journal of Computer Applications. 84. 8-10. 10.5120/14546-2631.

[24] Wang, Hong. (2014). Research on the Realization of LRU Algorithm. Applied Mechanics and Materials. 530-531. 891-894. 10.4028/www.scientific.net/AMM.530-531.891.

[25] Sa'ed Abed, Sara Abdul Aziz AlAwadh, Wathiq Mansoor, Shadi Atalla, Ahmad Alomari & Eric Yocam

[26] Zohreh Safari; Nahid Bohlol; Erfan Fouladfar

[27] Li, Jiatong & Hu, Huaixiang. (2021). An Adaptive Double Area Page Replacement Algorithm for NAND Flash. Journal of Physics: Conference Series. 1757. 012163. 10.1088/1742-6596/1757/1/012163.

[28] Lee, Woojoong & Park, Sejin & Sung, Baegjae & Park, Chanik. Improving Adaptive Replacement Cache (ARC) by Reuse Distance.

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

[30] P. Singh, R. Kumar, S. Kannaujia and N. Sarma, "Adaptive Replacement Cache Policy in Named Data Networking," 2021 International Conference on Intelligent Technologies (CONIT), Hubli, India, 2021, pp. 1-5

[31] Bansal, Sorav, and Dharmendra S. Modha. "Car: Clock with adaptive replacement." FAST. Vol. 4. 2004.

[32] Jiang, Song, Feng Chen, and Xiaodong Zhang. "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement." USENIX Annual Technical Conference, General Track. 2005.

[33] A. Jain and C. Lin, "Rethinking Belady's Algorithm to Accommodate Prefetching," 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 2018, pp. 110-123, doi: 10.1109/ISCA.2018.00020

[34] A. Vakil-Ghahani, S. Mahdizadeh-Shahri, M. -R. Lotfi-Namin, M. Bakhshalipour, P. Lotfi-Kamran and H. Sarbazi-Azad, "Cache Replacement Policy Based on Expected Hit Count," in IEEE Computer Architecture Letters, vol. 17, no. 1, pp. 64-67, 1 Jan.-June 2018, doi: 10.1109/LCA.2017.2762660.