

ASSIGNMENT 2 REPORT

Operating System

Page algorithm implementation and simulation

*Group (Individual):*

*Hoang, Doan Luong – a1960929*

*Course Coordinator:*

*Dr. Bernard Evans*

Contents

[Abstract 3](#_Toc209388998)

[Introduction 3](#_Toc209388999)

[Algorithm 3](#_Toc209389000)

[Methods 4](#_Toc209389001)

[Results 4](#_Toc209389002)

[Discussion 8](#_Toc209389003)

[Conclusion 9](#_Toc209389004)

Figures

[Figure 1: Page Hit Rate vs. Number of Frames for the bzip Trace 5](#_Toc209389005)

[Figure 2: Page Hit Rate vs. Number of Frames for the gcc Trace 6](#_Toc209389006)

[Figure 3: Page Hit Rate vs. Number of Frames for the sixpack Trace 7](#_Toc209389007)

[Figure 4: Page Hit Rate vs. Number of Frames for the swim Trace 8](#_Toc209389008)

# Appendices

[Appendix 1: Modified memsim.py 10](#_Toc209390351)

[Appendix 2: Code to plot the data 10](#_Toc209390352)

# Abstract

This report evaluates the effectiveness of three virtual memory page replacement algorithms: Random, Least Recently Used, and Enhanced Second Chance. The systems tested four real application traces - swim, bzip, gcc, and sixpack - to study algorithm response to memory constraints. The performance parameter tested was primarily page hit rates and the total number of disk I/O operations. By observing the point at which a rapid improvement in performance overtakes the increase in available memory, the analysis finds the approximate working set size of each application. We therefore conclude that though LRU serves as a yardstick for actual performance, Clock delivers practically the same efficiency and is thus best suited to modern operating systems.

# Introduction

For application programs that are larger than the available physical memory or RAM, common-day computer architecture simply assumes the work to be done by the Linux kernel through virtual memory. Through paging, the memory is divided into blocks of fixed sizes called pages in virtual memory and frames in physical memory. The operating system maintains the mapping between pages and frames; pages are loaded on demand into the memory from the disk.

A critical challenge appears when a running application requests a page that is not residing in any frame in the physical memory. This moment is called a page fault; the OS gets the halt signal for the process to locate the needed page on the disk and load it in an available frame in memory. Where there are no free frames, a page replacement algorithm must select the contents of an arbitrary frame to evict.

The efficiency of the page replacement algorithm is crucial to system performance. An optimal algorithm minimises the frequency of page faults, thereby reducing the time-consuming disk I/O operations that can severely degrade application speed. An inefficient algorithm, on the other hand, can lead to thrashing, where the system spends more time swapping pages than executing instructions.

# Algorithm

This report studies the performances of three different page-replacement algorithms (Arpaci-Dusseau & Arpaci-Dusseau 2015):

1. Random: This simple algorithm evicts a page that is randomly selected from the resident set of pages. The primary advantage of such an algorithm is less overhead, but it does not take into account any information about patterns of memory access.
2. Least Recently Used (LRU): LRU evicts a page that has gone the longest without being used. LRU is very good, but it is complicated and computationally expensive to implement.
3. Enhanced Second Chance (Clock): This algorithm gives an efficient approximation of LRU. It uses a reference bit and a dirty bit to give pages a "second chance" before being replaced, which prioritise the eviction of clean, unreferenced pages.

The objective of this study is to experimentally analyse and compare these algorithms using a custom simulator. By processing four distinct application traces under a range of memory availability, we aim to determine the memory requirements of each application and evaluate which algorithm provides the most effective performance under different conditions.

# Methods

* ***Simulator and Traces***

The study used a Python program as a simulator that processes traces of memory accesses and simulates virtual memory-system behavior. The four application traces on which the simulator ran were swim, bzip, gcc, and sixpack. Each trace file contained 1000000 lines of memory read and write operations that represented the real-world memory access patterns of that application. The description of each trace is as follows:

* Swim:
* Bzip:
* Gcc:
* Sixpack:
* ***Experimental Procedure***

The simulator was executed for three replacement algorithms (rand, lru, clock) on all four traces. To understand the impact of available memory, the number of available page frames was varied systematically. Since the data is small, the chosen range of frames was between 4 and 128, in steps of 4-memory size ranging from 8 KB to 512 KB. All simulations ran under quiet mode and statistics were obtained at the end of each run. To do this, we had to modify the memsim.py file in Appendix 1.

* Add fifo
* ***Performance Metrics***

The primary metric for evaluating algorithm performance was the Page Hit Rate, which provides a clear measure of efficiency. It is calculated as:

A higher page hit rate signifies better performance. The results of all the runs on one single trace file were combined and plotted to visualise the relationship between memory availability and algorithm performance for each trace.

# Results

#### bzip Trace Analysis

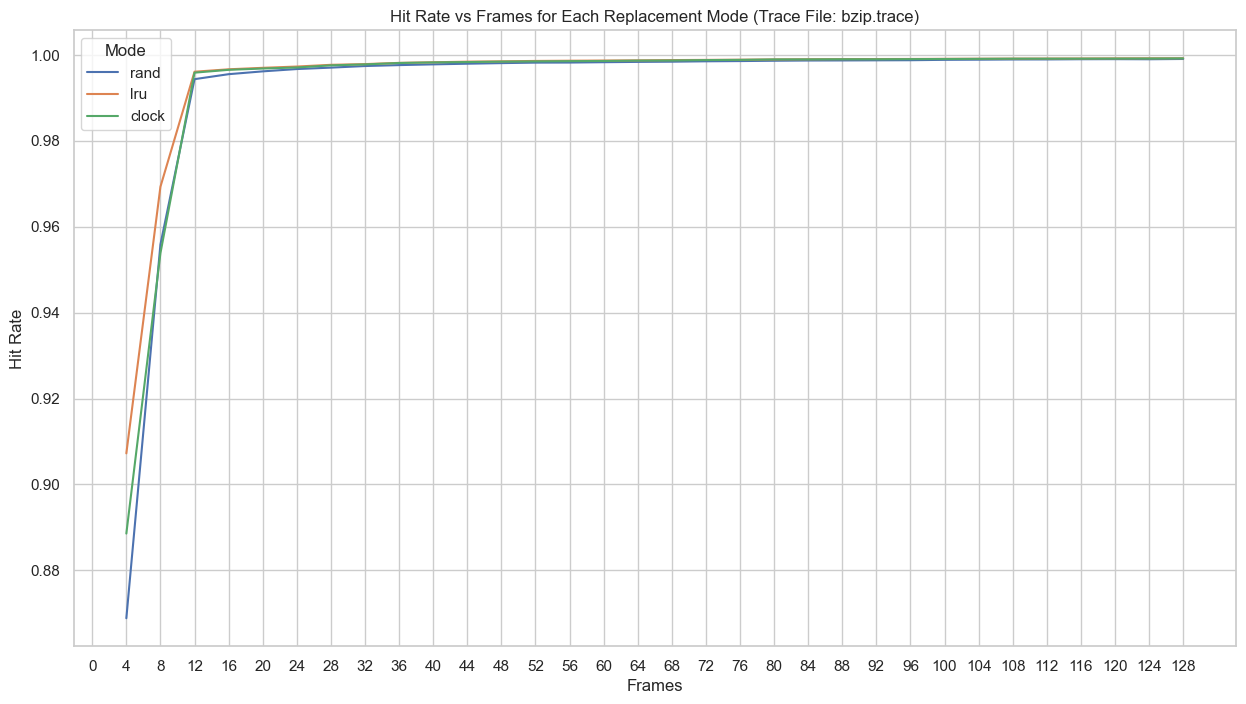


Figure 1: Page Hit Rate vs. Number of Frames for the bzip Trace

The bzip compression trace shows an overall increasing trend, as seen in Figure 1. Both the LRU (Least Recently Used algorithm) and Clock algorithms perform much better than Random, especially at the beginning. The performance curve for LRU and Clock is quite steep, showing that small increases in memory result in significant gains for the application. The main working point seems to peak at 12 frames, as hit rates for both LRU and Clock almost reach 100% at this point. This time, the Random replacement algorithm behaves so similarly to the other two that comparing their performances shows very few differences.

#### gcc Trace Analysis



Figure 2: Page Hit Rate vs. Number of Frames for the gcc Trace

The trace from the GCC compiler exhibits a similar memory behaviour, as shown in Figure 3. The performance gap between Random and the other two algorithms is significant here, where LRU and Clock both achieve 88% compared to Random’s 85% at frame 16. Both LRU and Clock perform very well, with the page hit rate for both climbing sharply after an increase in the number of frames, more especially in the region of 8 to 16 frames. In contrast, Random performs much worse, particularly when memory is constrained (4- to 36-frame region). The increase curve for LRU and Clock is less sharp compared to bzip trace and there is still improvement later, with performance gains continuing up to 100 frames and beyond.

#### sixpack Trace Analysis

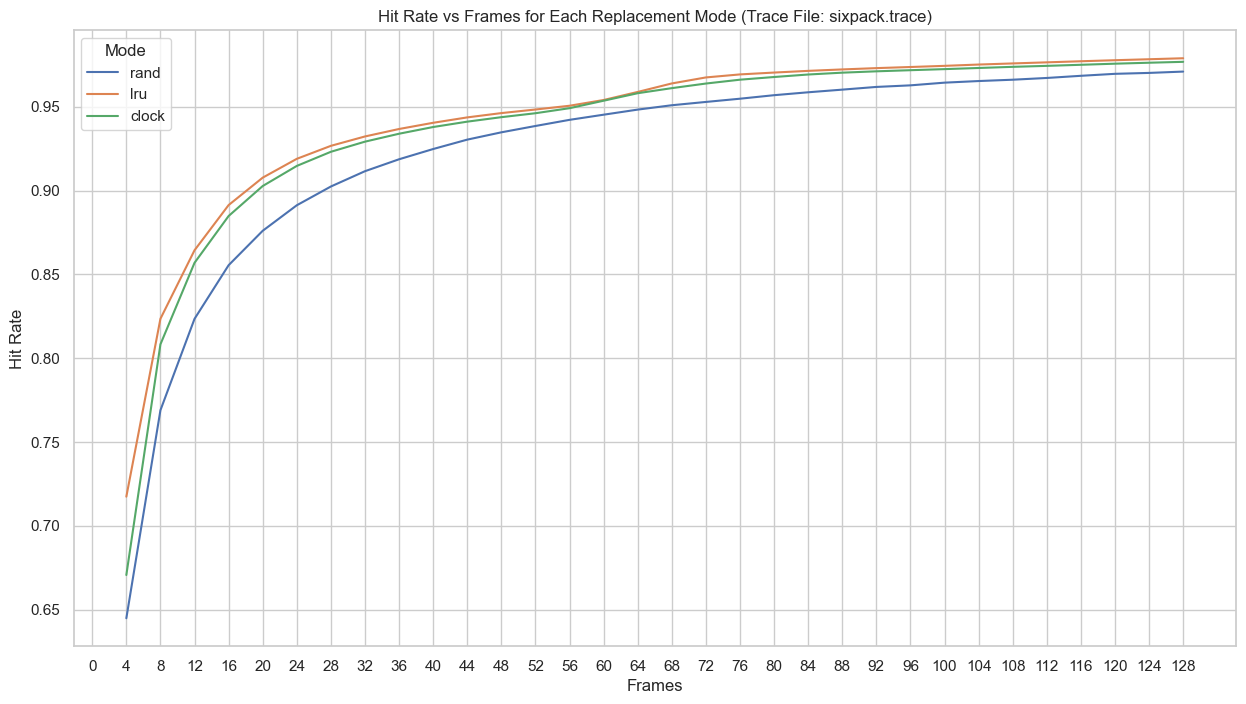


Figure 3: Page Hit Rate vs. Number of Frames for the sixpack Trace

Figure 3 shows the memory trends when applying the three algorithms to the sixpack trace. The page hit rates for both the Least Recently Used (LRU) and Clock algorithms increase significantly once the number of frames reaches 8. Similar to the gcc method, the Random algorithm performs poorly in comparison, achieving only 87% page hit rates, while LRU and Clock reach 91%. The performance of the Clock algorithm closely follows that of LRU. Moreover, the rate of increase for the LRU and Clock algorithms is steeper than that of gcc. However, there is a slight fluctuation in the performance at frame 56. After this point, both algorithms resume their upward trend, and subsequently, the page hit rates stabilize at 98%.

#### swim Trace Analysis

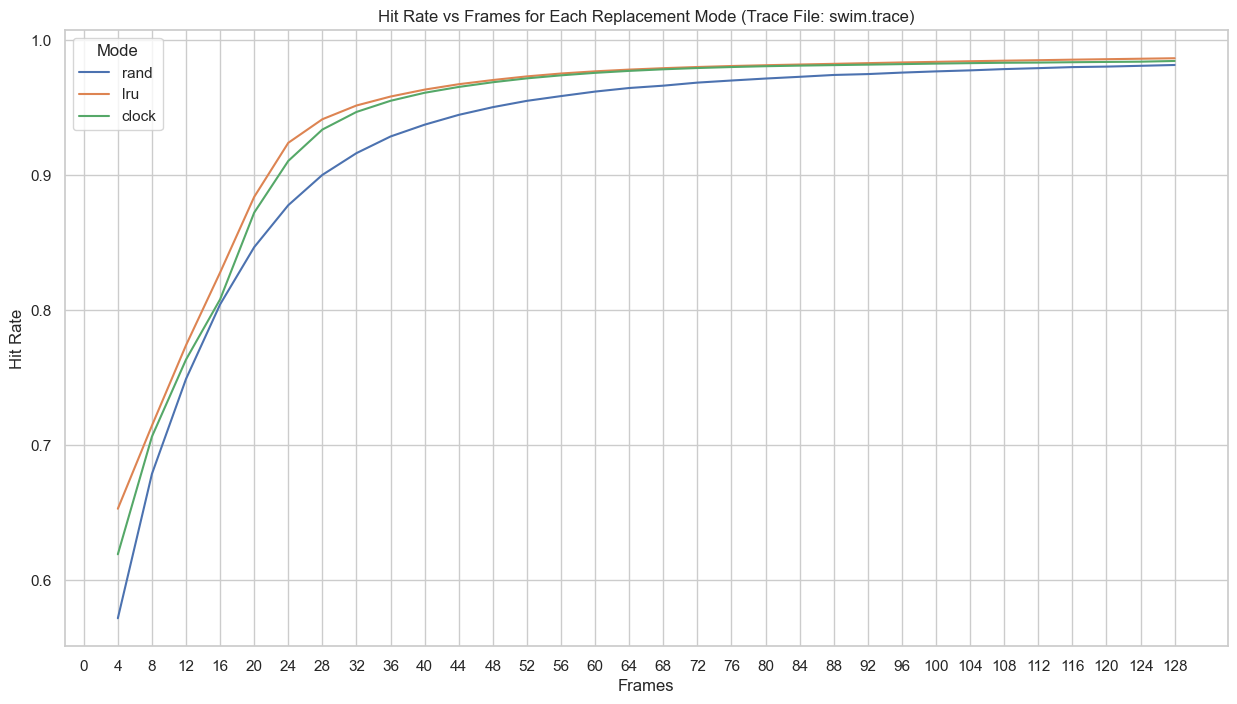


Figure 4: Page Hit Rate vs. Number of Frames for the swim Trace

Compared with other traces, we see a comparatively much lower page hit rate in the earlier stages. LRU and Clock really shine and with the number of frames, the page hit rates shoot up substantially, as shown in Figure 1. The Random algorithm fares a bit worse, especially with tight-memory constraints of 4 to 32 frames. There is the biggest leap in performance for both LRU and Clock between 20 and 28 frames, with more page hits being recorded once the number of frames available passes that threshold. The Clock algorithm also puts up a performance on par with that of LRU.

# Discussion

Regarding the questions of the simulation, we conclude the following:

* **How much memory does each traced program actually need?** From the graphs, we can see that for all 4 traces, LRU and Clock both reach 90% page hit rate at frame 24 - 28. We do not consider the Random method as it is far less effective compared to the other two. Given that each frame size is 4 KB, we can conclude that these two methods can work well given around 98 - 112 KB.
* **Which algorithm works best with a low number of frames?** In situations with constrained memory, both LRU and Clock are vastly superior to Random as they are far more effective at keeping the most critical pages in memory, thus minimising page faults. However, compared to each other, LRU seems slightly better than Clock, although we haven’t considered the complexity of the algorithm.
* **Does one algorithm work best in all situations?** While true LRU consistently provided the best or near-best performance, the Clock algorithm performed at a nearly identical level across all traces and memory sizes. Given its lower implementation overhead, Clock stands out as the most practical and effective algorithm for general-purpose operating systems. The Random algorithm, conversely, was uniformly the worst-performing and is not a suitable choice for any memory-intensive application.

# Conclusion

In conclusion, this study successfully simulated and evaluated the performance of three main-page replacement algorithms on four different application traces. The results have always shown that those algorithms that recognize memory access patterns, such as LRU and Clock, perform much better as compared to the simple Random replacement method, especially under memory constraints. LRU, although slightly better, was joined by Clock whose performance was practically indistinguishable from it, thus cementing its practical and effective approximation for use in real-world systems. In other words, selecting either LRU or Clock means good system performance with minimum disk I/O and an application running smoothly and efficiently.

# References

* Arpaci-Dusseau, R and Arpaci-Dusseau, A 2015, *Operating systems: Three easy pieces*, Madison: Arpaci-Dusseau Books.

# Appendices

from clockmmu import ClockMMU

from lrummu import LruMMU

from randmmu import RandMMU

def run\_simulation(input\_file, frames, replacement\_mode, debug\_mode):

    PAGE\_OFFSET = 12  # page is 2^12 = 4KB

    ############################

    # Setup MMU based on replacement mode

    ############################

    if replacement\_mode == "rand":

        mmu = RandMMU(frames)

    elif replacement\_mode == "lru":

        mmu = LruMMU(frames)

    elif replacement\_mode == "clock":

        mmu = ClockMMU(frames)

    else:

        print("Invalid replacement mode. Valid options are [rand, lru, clock]")

        return

    # Set debug mode

    if debug\_mode == "debug":

        mmu.set\_debug()

    elif debug\_mode == "quiet":

        mmu.reset\_debug()

    else:

        print("Invalid debug mode. Valid options are [debug, quiet]")

        return

    ############################################################

    # Read the trace file and simulate

    ############################################################

    total\_accesses = 0

    with open(input\_file, 'r') as trace\_file:

        for trace\_line in trace\_file:

            trace\_cmd = trace\_line.strip().split(" ")

            logical\_address = int(trace\_cmd[0], 16)

            page\_number = logical\_address >>  PAGE\_OFFSET

            # Process read or write

            if trace\_cmd[1] == "R":

                mmu.read\_memory(page\_number)

            elif trace\_cmd[1] == "W":

                mmu.write\_memory(page\_number)

            else:

                print(f"Badly formatted file. Error on line {no\_events + 1}")

                return

            total\_accesses += 1

    # Calculate hit rate

    hit\_rate = 1 - (mmu.get\_total\_page\_faults() / total\_accesses) if total\_accesses > 0 else 0

    return (frames, total\_accesses, mmu.get\_total\_disk\_reads(), mmu.get\_total\_disk\_writes(), mmu.get\_total\_page\_faults(), hit\_rate)

Appendix 1: Modified memsim.py

Appendix 2: Code to plot the data