Operating Systems Assignment 2 Report

Gian-Piero Carpinelli, Yihong Su. Equal contributions.

January 14, 2026

1 Introduction

At the heart of modern operating systems (OS) is the concept of virtualisation. At the CPU level, this technique turns a small finite number of processors into a seemingly infinite number, providing the illusion of exclusive access to a process and concurrency to the user.

Similarly, memory is virtualised such that each process believes it has a large, contiguous, and private address space. In reality the amount of RAM in the system is limited, and the OS (with support from hardware) does a lot of work to provide this illusion and enable more processes to be concurrently active than could fit simultaneously in physical memory.

The way in which memory is virtualised is critical, as the OS needs to ensure temporal and spatial efficiency. Different techniques can be used to virtualise memory, such as the simple base and bounds registers. Unfortunately, this technique can have large amounts of internal fragmentation which can limit the number of active processes due to insufficient contiguous memory, even if enough memory is available.

Segmentation aimed to reduce the internal fragmentation that base and bounds produced, by dividing memory into variable sized logical segments. Unfortunately, this introduces external fragmentation. As segments are variable sized, allocating memory to a process typically results in free memory of odd sizes; making subsequent memory allocation requests difficult.

Paging was developed to combat the inadequacies of segmentation. Instead of splitting a process's address space into some number of variable sized segments we divide it into fixed-sized units, each of which we call a page. Correspondingly, physical memory is viewed as an array of fixed sized slots, called page frames, where each frame can hold a single page from a processes address space. Paging supports simple free-space management, as the size of a page frame within memory is the same as a virtual page of a process.

A key decision in the implementation of paging is which page to evict when memory becomes full, i.e. what should the page replacement policy be? When a process attempts to access a page that is not currently in memory, a page fault occurs and the OS must load the page from disk. Selecting an optimal policy will reduce the number of page faults and thus reduce the number of disk I/O operations to swap space. This is critical as I/O operations are very slow compared to memory access.

This report evaluates the performance of three page replacement policies, namely Least Recently Used (LRU), Clock, and Random. These policies are implemented in a simulated environment and tested against memory traces from real programs; swim, bzip, gcc, and sixpack. As the number of physical frames that the simulation is modelling is configurable, the performance of each replacement policy in regards to varying memory constraints can be analysed.

2 Methods

Firstly, the simulator (memsim.c) was developed and tested against the given sample traces to ensure it accurately modeled the LRU, Clock, and Random policies. A Python script was then developed to assist in

understanding the memory constrains of the four applications (bzip, gcc, sixpack, and swim). This script automated running memsim across various numbers of frames and produced a .csv file with the data.

A lower and upper bounds of 4 and 1024 frames was selected so that the behaviour of the applications when experiencing memory shortage, memory close to process needs, and excess memory could be analysed. The lower bound was based on manual testing, which showed that 4 frames created sufficient memory pressure while still allowing the application to run. The upper bound was selected to exceed the working sets of the applications, as determined by manual testing, enabling analysis with an abundance of memory. Notably, gcc continued to show improvements even as the frame count approached 1024 (Figure 4).

R was used to plot the data obtained via the Python script. A hit rate plot was generated directly from the data, and derivative plots that depict changes in hit rate with respect to changes in number of frames were also generated. These derivative plots assist in understanding when additional frames become redundant. The Python and R scripts were submitted alongside memsim to GradeScope.

3 Results

For Figures 1 to 4 both panels display the same data at different scales. Left panel focuses on initial behaviour (under 64 frames) whereas the right panel focuses on frame counts beyond 64 to highlight the asymptotic behaviour of the hit rate.

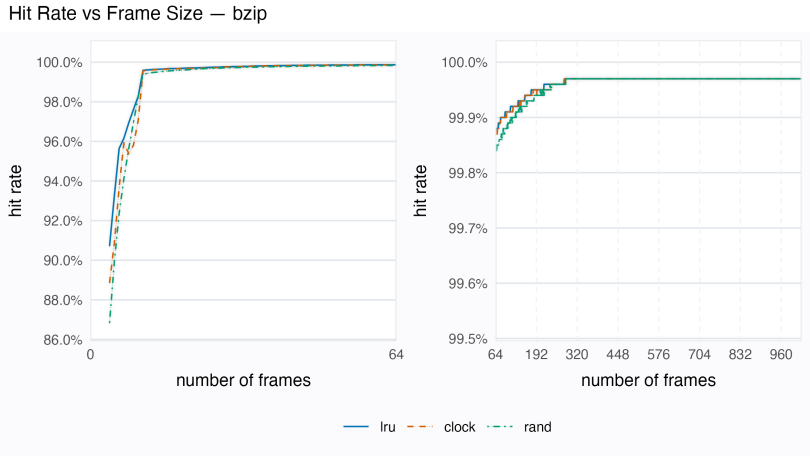

Figure 1: bzip number of frames vs hit rate.

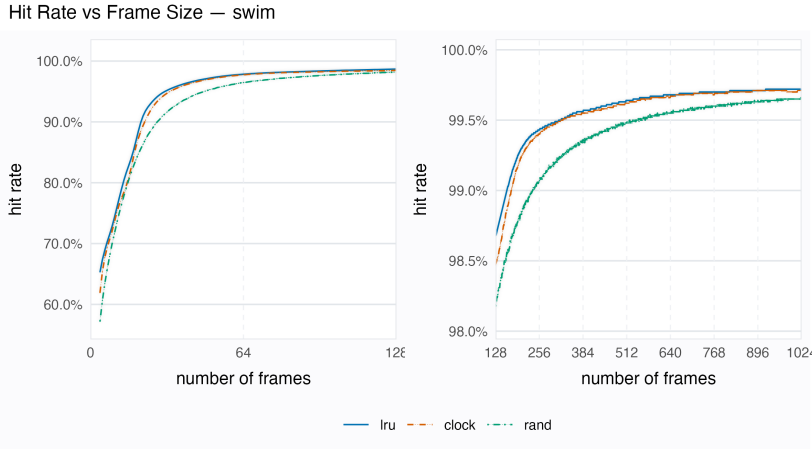

Figure 2: swim number of frames vs hit rate.

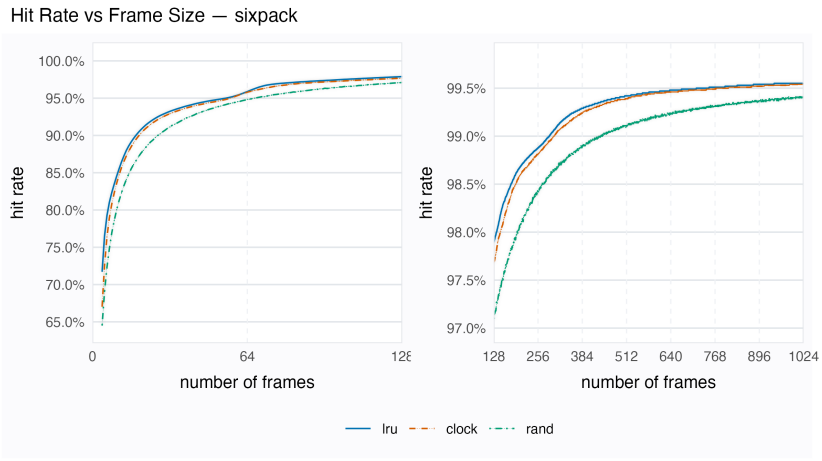

Figure 3: sixpack number of frames vs hit rate.

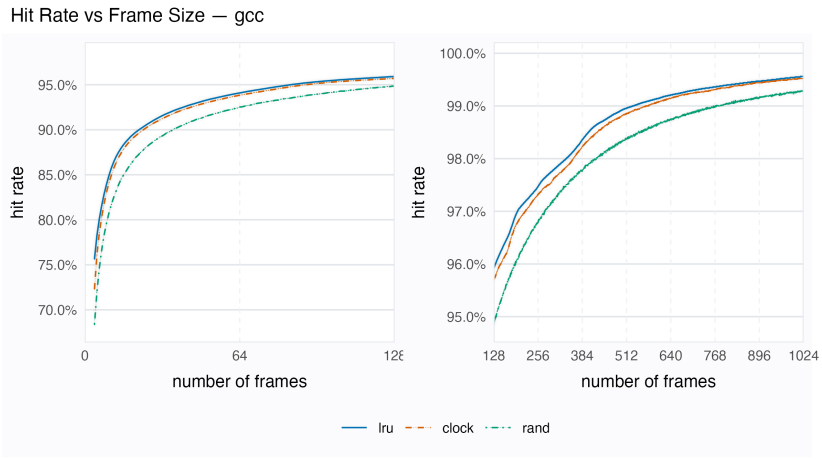

Figure 4: gcc number of frames vs hit rate.

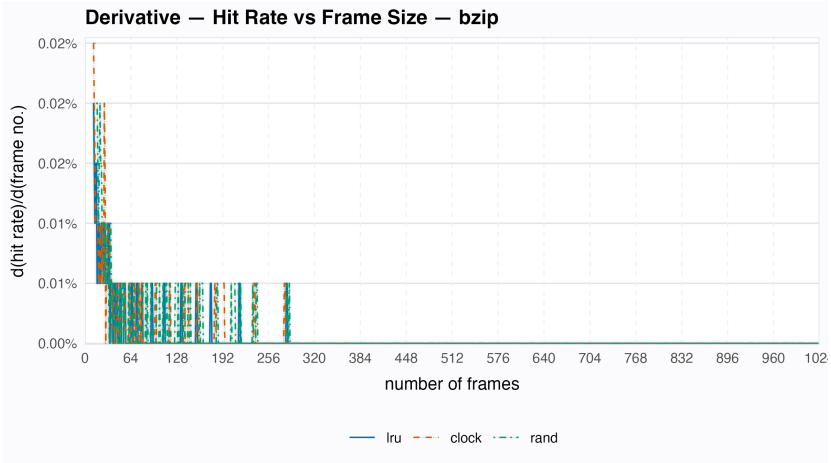

Figure 5: bzip rate of change of hit rate with respect to number of frames.

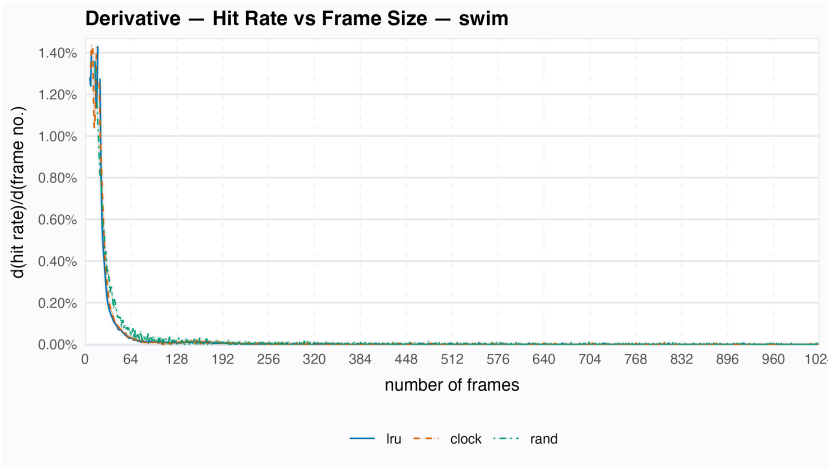

Figure 6: swim rate of change of hit rate with respect to number of frames.

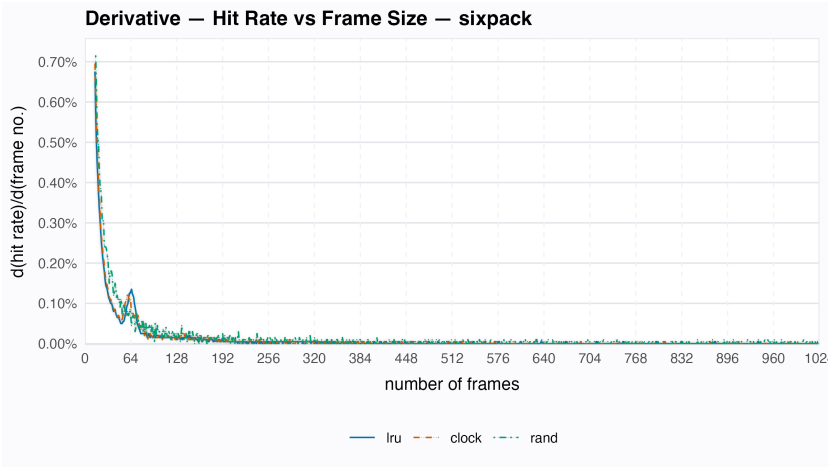

Figure 7: sixpack rate of change of hit rate with respect to number of frames.

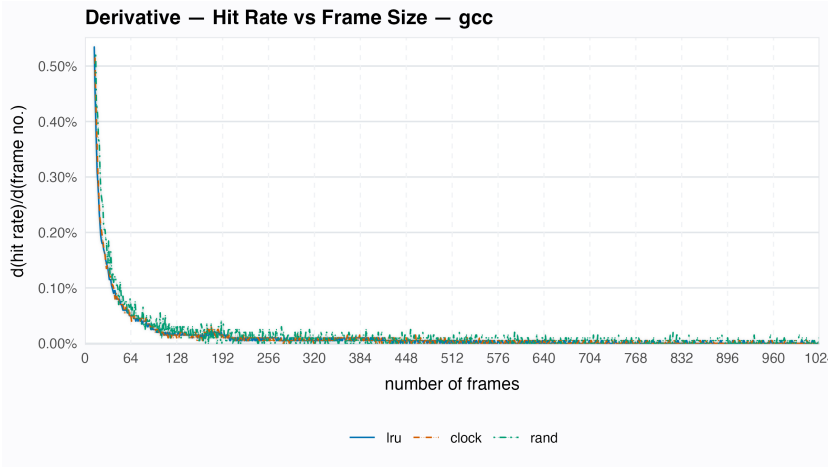

Figure 8: gcc rate of change of hit rate with respect to number of frames.

Unsurprisingly, LRU consistently outperforms Clock and Random across all programs tested in this experiment, with Clock always performing better than Random. Clock can be seen to perform almost as-well as LRU across all applications, with performance gaps being under 1% when the number of frames does not place memory pressure on the application. This shows that Clock can be used as an approximation of LRU in real systems where the overhead of LRU would impact system performance.

Although all policies' hit rate converges to within 1% once the memory size is sufficiently large; significant differences in performance is observed at smaller frame numbers. For example, a 13% difference between LRU and Random at low frame numbers for the bzip trace (Figure 1). Such behaviour is expected as LRU is exploiting temporal locality in the aims of achieving a higher hit rate.

The hit rate plots (Figures 1 to 4) portray the variance in memory access that the four programs have, highlighted by differences in hit rate at low frame quantities and the amount of frames needed to hit the asymptotic limit for the program. Bzip (Figure 1) is the quickest to reach its performance limit, doing so in 16 frames. At low frame numbers, LRU and Clock perform excpetionally well for bzip compared to other applications, indicating bzip's significant temporal locality and a small working set (around 16 pages) which these policies are able to exploit. Bzip having the smallest working load out of the programs tested is reasonable due to the sliding window technique which it uses, meaning that only a small amount of pages are need in memory at any given time.

Swim (Figure 2) can be seen to plateu next, beginning to display this plataeu at 32 frames and well and truly hitting this limit at 50 frames, indicating that swim's work load is the second smallest. Swim having the second smallest working load of the given applications is reasonable as the memory trace would likely be accessing adjacent elements within the 2D arrays that the program uses. Evidently, policies which utilise spatial and temporal locality perform well.

The program with the next largest work load is sixpack (Figure 3), which reaches a plataeu in performance at around 80 frames. This indicates that sixpack needs a larger amount of pages in memory or has more complex memory access patterns than bzip and swim. Gcc (Figure 4) takes the longest to reach its plataeu, with improvements in hit rate greater than 1% occuring with frame quantities as large as 320. For example, 98% hit rate at 320 frames compared to 99% at 512 frames for LRU. Such behaviour can be expected for a compiler that has complex memory access patterns needed to parse source code, manage symbol tables, perform optimisations, and generate code.

As all policies examined achieve a hit rate of 99% or greater for all applicationes tested when given enough frames, the choice of replacement policy seems meaningless. Note however that this experiment models only one process at a time. Real OS page replacement occurs in a multi-process environment, where the frames in memory will contain virtual pages from various processes. The results presented here are a best case scenario, when all available frames (all memory) can be allocated to a single process, which will never occur in a real system.

Let's consider running these applications in a real, multi-process, system to see if the the replacement policy can truly be considered meaningless. The number of frames in memory is limited by the size of the memory. If we keep the page size to the 4 KB used in this experiment and assume a memory size of 4 GB, 8 GB, or 16 GB then there are 1 million, 2 million, or 4 million page frames, respectively.

The OS keeps some frames for itself, other frames are needed for the page table, and some frames may be reserved for hardware, with the remaining frames being available to user processes. The amount of memory required by the OS varies based on the system, however it is safe to assume it is not a trivial amount and it will corespond to a meaningful reduction in the memory available for user processes. Running $ps - e \mid wc - l$ in a Unix environment will display the number of active processes, i.e. running, blocked, or ready - meaning they have pages in memory or at least in swap space. Doing so on my system (2023 MacBook) across a few days consistently gave values in the 500's. So, let's assume 500 processes and assume 1000 frames per process, significantly more than needed to achieve a 99% hit rate for the applications tested here. This would require 500,000 frames which is a significant amount for the 4 GB memory, but can be considered negligible if we have a 16 GB memory.

At first, this would suggest that in real systems it is reasonable to expect a near optimal hit rate as there is an abundance of memory, however the above analysis assumption of memory required per process (1000 frames) is rather low. Although the programs tested in this experiment can perform well with as little as 16 frames for bzip or 500 frames to achieve a similar hit rate for gcc, applications such as web browsers, games, or video editors can easily require tens of thousands of frames per process. So memory pressure is more significant than what this multi-process example initially implied, and selecting an appropriate page replacement policy is critical. Furthermore, we have failed to account for the CPU scheduling algorithm. The efficacy of a page replacement policy in a multi-process system is dependent on the CPU scheduling policy.

Unlike the single-process model examined in this experiment, a real system would include a CPU scheduler which determines process execution order. Different CPU scheduling policies would impact the performance of page replacement policies. For example, the temporal locality that LRU utilises would likely be made useless by a round-robin scheduler as recently accessed pages from one process could be irrelevant by the time this process is scheduled again.

Interactions between the CPU scheduling policy and the page replacement policy highlight how system performance isn't based on individual component performance but on how these components work together. So, although the single-process simulation results were valuable in understanding the behaviour of the paging policies themselves, future work includes testing these in a multi-process environment that supports different scheduling policies.

Turning to the deriviative plots (Figures 5 to 8) it is evident that meaningful improvements in performance occured within the first 32 additional frames for swim, sickpack, and gcc or for the first additional 16 frames for bzip. After this point, although additional frames continue to increase the hit rate the relative change on a per frame basis is significantly lower. This reduction in improvements to additional frames indicates that the application is provided with a sufficinetly large memory.

4 Conclusion

This report investigated the performance of three page replacement policies (LRU, Clock, and Random) across four application traces (bzip, swim, sixpack, and gcc). LRU consistently achieved the highest hit rate with Clock trailing slightly behind, doing so with lower implementation overheads, while Random always performed worst. The plots revealed that differences in performance between the policies is most significant when the system is under memory pressure. LRU and Clock were able to perform 13% better than Random when frame quantities were below an application's working set size.

The plots of number of frames versus hit rate enabled the applications' working set to be determined. Bzip was found to be the smallest, requiring only 16 frames, followed by swim with 32 frames. Next was sickpack requiring 80 frames, with gcc requiring the most frames (320). Analysis of the plots indicated that the best return on investment in terms of additional frames occurs when increasing the number of frames from below an application's working set to a value approaching it. Once this threshold is reached, all policies were seen to converge to similar performance levels; even Random exceeded 95% hit rate. This indicates that replacement policies cannot overcome working set requirements and that a sufficiently large number of frames is essential in achieving a reasonable hit rate.

Although this single-process simulation provided valuable insights, real, multi-process systems face additional challenges not modelled in this experiment. Namely, larger applications with working sets in the tens of thousands and memory pressure as a result of being in a multi-process system where processes compete for memory. Additionally, interactions with the CPU scheduler would impact the performance of the page replacement policies, for example a round robin thread scheduler would likely impact the temporal locality that LRU relies on.

Cumulatively, this experiment found that while LRU offers the best performance in a simulated single process

environment, a real system may be served just as well by Clock which has the benefit of reduced implementation overheads. Furthermore, effective memory management in real systems was found to rely on a combination of an intelligent replacement policy, adequate memory allocation per process (which requires sufficiently large physical memory), and consideration of the thread scheduler iterating with the paging policy.

5 References

- Arpaci-Dusseau, R. H., Arpaci-Dusseau, A. C. (2020). Operating Systems: Three Easy Pieces. Available at: https://pages.cs.wisc.edu/ remzi/OSTEP/
- Wickham, H. (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. ISBN 978-3-319-24277-4. Available at: https://ggplot2.tidyverse.org