Implementation of a Frequency-Awareness Page Replacement Policy

Lei Li
The Chinese University of Hong Kong
Shenzhen, China
225040154@link.cuhk.edu.cn

***Abstract*—Page replacement algorithms are a key mechanism in operating system memory management. This paper uses the default page replacement mechanism of the Linux 5.15 kernel as a baseline to design and implement a Frequency-Awareness page replacement algorithm. By introducing access frequency counting and combining it with a aging mechanism, this method preserves long-term hot pages while suppressing the negative accumulation of historical information, thereby improving the stability of memory management.**

Under various memory pressure conditions, this paper conducts comparative experiments between the proposed algorithm and baseline methods based on metrics such as CPU utilization, major page fault rate, and the thrashing point. The results show that the page replacement policy can reduce the major-fault rate during thrashing and delay the thrashing point to some extent, but the increased overhead also leads to a waste of CPU resources.

***Index Terms*—page replacement, LRU, LFU, Linux Kernel, Memory Management**

I. Introduction

As computer system scales continue to expand and application workloads grow increasingly complex, memory management has become increasingly critical to operating system performance optimization. As one of the core mechanisms of virtual memory management, page replacement algorithms directly determine a system's performance under memory-constrained conditions. An efficient page replacement strategy not only improves memory hit rates but also significantly reduces disk I/O overhead, thereby enhancing overall system response speed.

In modern Linux kernels, the default Page Replacement Policy is based on an approximation of the Least Recently Used (LRU) algorithm, which uses linked lists of active and inactive pages to approximate page access locality. However, when faced with complex access patterns, this mechanism may fail to accurately reflect the importance of pages, leading to thrashing and severely impacting system performance.

Therefore, researchers and industry practitioners have proposed various improvement strategies, such as the Working Set model and the LFU (Least Frequently Used) strategy. However, implementing these algorithms in actual operating systems and verifying their effectiveness still poses significant engineering challenges.

The primary objective of this project is to design and implement a page replacement algorithm that tracks page access frequency. In addition, a GUI was developed to visualize page replacement behavior, and comparisons were made between the custom strategy and the Linux default strategy in terms of page faults and thrashing points under varying workloads.

II. Background

Modern operating systems commonly employ virtual memory mechanisms to separate physical memory from the logical address space, thereby supporting concurrent execution of multiple processes and a larger address space. Under this mechanism, the virtual addresses accessed by processes must be mapped to physical page frames via page tables. When the system's physical memory is limited, the operating system must select certain pages to remove from memory. This process is called Page Replacement.

The core objective of page replacement algorithms is to minimize the page faults—particularly costly major faults. According to the principle of locality, programs tend to access a relatively concentrated set of pages within a short period of time; therefore, effectively leveraging temporal and spatial locality is key to designing efficient replacement algorithms.

A. Classic Page Replacement Policy

Early research proposed several classic page replacement strategies, including the following representative examples:

- **FIFO** (First-In First-Out): Replaces pages based on the order in which they entered memory. It is simple to implement but may suffer from the Belady anomaly (Adding more resources actually led to higher error rates).
- **OPT** (Optimal): A theoretically optimal strategy that selects the page least likely to be accessed in the near future for replacement. However, since it requires foreknowledge of future access sequences, it is used only as a performance upper bound.
- **LRU** (Least Recently Used): Based on temporal locality, this algorithm prioritizes evicting the least recently used page and is one of the most commonly used strategies in practical systems [1].

B. The Linux kernel's default Page Replacement Policy

Implementing LRU exactly requires maintaining a complete access history, which is quite costly. In practice, the Linux kernel employs a policy based on an approximate LRU algorithm.

As shown in Fig. 1, the system divides pages into an Active List and an Inactive List. Pages dynamically migrate between the Active and Inactive linked lists based on access patterns,

and the kernel selects appropriate pages for eviction through scanning and reclamation processes. Additionally, Linux uses the referenced bit and page flags to assist in determining page activity.

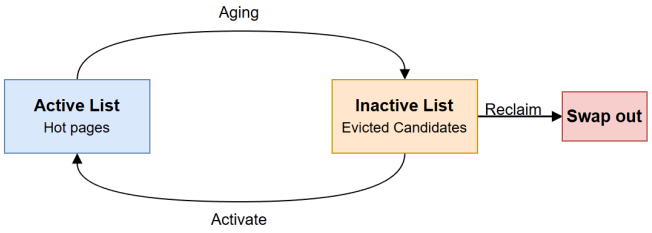

Fig. 1: Illustration of Multi-List Approximation for LRU.

This design performs well in most scenarios, but it has limited capabilities in capturing access frequency and long-term behavior.

Therefore, this project aims to propose a replacement algorithm that better combines temporal locality with frequency information, and to evaluate its performance compared to the default algorithm.

III. Experimental Environment Setup

This experiment is conducted in a virtualized environment using the **Ubuntu** operating system, on which a custom Linux kernel is compiled and run. The experimental environment is configured as follows:

- Operating System: **Ubuntu 22.04**
- Kernel Version: **Linux 5.15** (*compiled from source, with a custom page replacement policy*)
- Virtualization platform: VMware Workstation
- CPU: x86_64 architecture (2–4 cores)
- Memory: 4 GB
- Storage: $\geq$40 GB

To ensure the reproducibility of the experiment, all tests are conducted under the same hardware resource constraints, and unnecessary background services are disabled to minimize interference.

IV. Design and Implementation

This paper extends and modifies the critical path of the Linux kernel's default Page Replacement Policy. The overall design follows these principles: modifying the logic while minimizing disruption to the existing LRU framework; and avoiding excessive time and space overhead in the page scanning path.

Therefore, the modifications include:

1) Track page visit frequency;
2) Update the aging mechanism: Add frequency decay;
3) Remove the page with the lowest frequency within the retention window.

A. Access Frequency Statistics

In the `mm/swap.c` file of the original Linux kernel, the `mark_page_accessed()` is used for marking a page as "recently accessed" and moving it to the active list. To track access frequency, add a counter to the page [2].

```
if (!PageReferenced(page)) {
    SetPageReferenced(page);
    /* Initialize the counter */
    if (page->private == 0)
        page->private = 1;
}
else {
    /* the counter increments */
    if (page->private < 1024)
        page->private++;
     ...
}
```

In addition, in order to take into account both short-term page visit records and long-term visit trends, the criteria for inclusion in the active list have been modified to require that both the reference bit be set to 1 and the frequency exceed a preset threshold. Here the referenced bit indicates a recent visit, while the frequency reflects the long-term visit trends of the page.

B. Updated Page Aging

The original aging mechanism is designed to deactivate pages that haven't been accessed recently. The modified aging mechanism will shift the counter value 1 bit to the right during each scan of the inactive list.

```
if (page->private > 0)
    page->private >>= 1;
```

The advantage of the shift operation is that once a page is no longer visited, its access frequency will decline rapidly, even if it was previously highly frequented. This mechanism can gradually deactivate and remove pages that have been used frequently before, but are inactive now.

C. Elimination Strategy

Instead of removing the page at the top of the Inactive list, the eviction strategy has been modified to remove the page with the lowest frequency within a scan window in the Inactive queue. Specifically, to reduce overhead, rather than sorting the entire list, it only scan a window of a specified size.

```
list_for_each_entry(page, page_list, lru) {
    unsigned long freq = page->private;

    if (freq < min_freq) {
        min_freq = freq;
        min_page = page;
    }

    if (++count >= window_size)
```

```
            break;
}
```

Compared to sorting the entire list, scanning only within a page window effectively reduces overhead while more accurately identifying "pages that are unlikely to be used."

V. Experimental Design and Methodology

A. Workload Construction

To trigger varying degrees of page-filling behavior, this article uses the following commands:

```
stress-ng --vm 1 --vm-bytes 3G --timeout 60s
```

This will place a specific amount of memory pressure on the system. The size is gradually increased from 1 GB to near the upper limit of physical memory to observe the transition of the system from normal operation to a thrashing state.

B. Evaluation Criteria

The key data metrics monitored during the experiment include:

- **major fault** / second: The number of pages the system needs to load from disk every second (because these pages are not in memory);
- **CPU utilization**: The percentage of time the CPU spends executing tasks. In practice it was computed as:

$$\text{CPU Utilization} = \frac{\text{time on task-clock}}{\text{total time}} \tag{1}$$

- **Thrashing Point**: Timing when the system's page fault rate rises significantly and CPU utilization drops.

The above metrics can be displayed on the system command line using the `pref`. As shown in Fig. 2, The system generated several statistical metrics within 60 seconds after the load was applied.

```
cuhksz@cuhksz-virtual-machine:~/Desktop$ sudo perf stat -e task-clock,page-fault
s,major-faults stress-ng --vm 1 --vm-bytes 2.9G --timeout 60s
stress-ng: info:  [3523] setting to a 60 second run per stressor
stress-ng: info:  [3523] dispatching hogs: 1 vm
stress-ng: info:  [3523] successful run completed in 60.06s (1 min, 0.06 secs)

 Performance counter stats for 'stress-ng --vm 1 --vm-bytes 2.9G --timeout 60s':

         56,581.60 msec task-clock                #    0.942 CPUs utilized

         6,882,309      page-faults               #  121.635 K/sec

               540      major-faults              #    9.544 /sec


      60.062731358 seconds time elapsed

      38.588021000 seconds user
      18.079932000 seconds sys
```

Fig. 2: Example of using the 'pref' command to display system metrics under workload.

C. Experimental Setup and Procedure

The experimental group set the default Linux 5.15.0 kernel as the baseline. Test the metrics described in the previous section by varying the size of the workload.

Each set of experiments is conducted according to the following steps:

1) Start the specified kernel version;
2) Clear the cache to isolate historical effects:
   `sudo sh -c "echo 3 > /proc/sys/vm/drop_caches"`
3) Run the test workload and collect the following metrics simultaneously.

VI. Results and Analysis

A. Results

The experiment was conducted on the control group described earlier, and the results are shown in the Fig. 3 and Fig. 4.

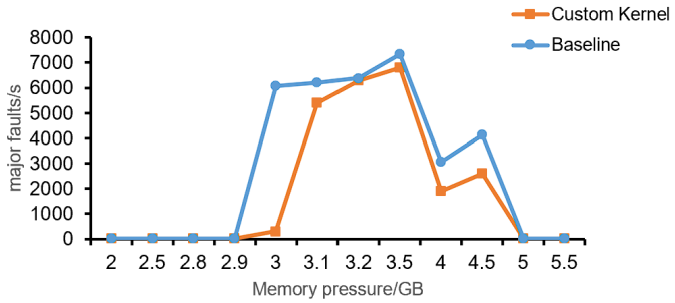

Fig. 3: Major Faults per second of the Two Kernels Under Different Workloads.

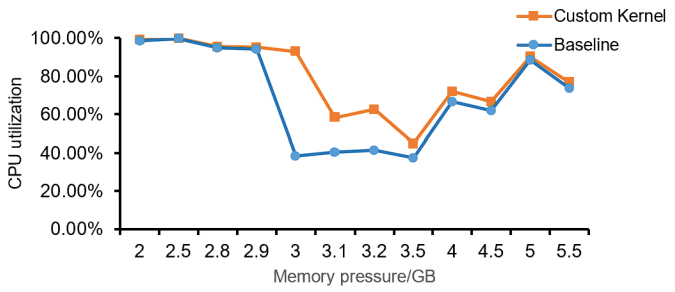

Fig. 4: CPU Utilization of the Two Kernels Under Different Workloads.

As shown in Fig. 3, it can be observed that as the load increases, the custom kernel and the baseline kernel both exhibit a significant increase in the major-fault rate at 3.1 GB and 3 GB, respectively, which indicates the occurrence of system thrashing. In comparison, the custom kernel exhibits thrashing slightly later.

On the other hand, after thrashing occurs, the major-fault rate of the former is almost always lower than that of the latter.

At the same time, as shown in 4, although CPU utilization in both groups dropped significantly after thrashing, the utilization of the custom kernel remained higher for most of the time and exhibited more pronounced fluctuations.

B. Analysis

Based on the experimental results, the modified replacement algorithm appears to delay the thrashing points and reduce the major-fault rate under high loads. This may be due to tracking page visit frequency, which helps better retain long-term popular pages. At the same time, the decay mechanism effectively prevents the pollution from historical data.

On the other hand, however, higher CPU utilization during thrashing does not mean the system is executing useful application logic; rather, it may be spending more resources on maintaining frequency counters, performing in-window sorting, and so on. This also highlights the overhead drawbacks of the custom page replacement algorithm compared to the default replacement algorithm.

VII. Conclusion

In this work, based on the kernel of linux 5.15.0, a Frequency-Aware page replacement policy was designed by modifying or adding logic such as tracking access frequency to the original kernel code.

Based on quantitative experiments, the performance of the custom replacement algorithm was evaluated. The results show that the Frequency-Awareness page replacement policy can reduce the major-fault rate during thrashing and delay the thrashing point to some extent, but the increased overhead also leads to a waste of CPU resources, which is reflected in higher CPU utilization during thrashing.

References

[1] Tanenbaum, Andrew S., and Herbert Bos. Modern operating systems. Pearson Education, Inc., 2015.

[2] The Linux Kernel Organization. 2026. Idle Page Tracking. https://docs.kernel.org/admin-guide/mm/idle_page_tracking.html