Operating Systems Project: Topic 5

Implement a Custom Page Replacement Policy

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.20

Outline

Outline

The Mission: Topic 5 Requirements

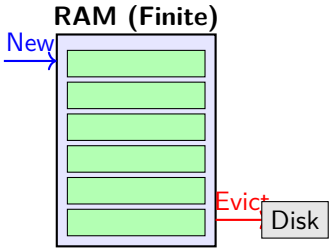
Objective: Modify the Linux kernel's page replacement algorithm.

Requirement 1: Core Implementation

- **Task:** Replace the default eviction logic.
- **Target:** Implement **Working Set** or **Adaptive LRU**.

Requirement 2: Monitoring & UI

- **Tools:** Use `vmstat` / `perf` to monitor faults.
- **Visualization:** GUI displaying real-time page status.



Advanced Options

Option A: Frequency-Awareness (LFU-like)

- **Concept:** Standard LRU tracks *Recency*. You should track **Frequency**.
- **Why?** Prevents "Scan Pollution" (One-time read of a large file shouldn't flush hot pages).

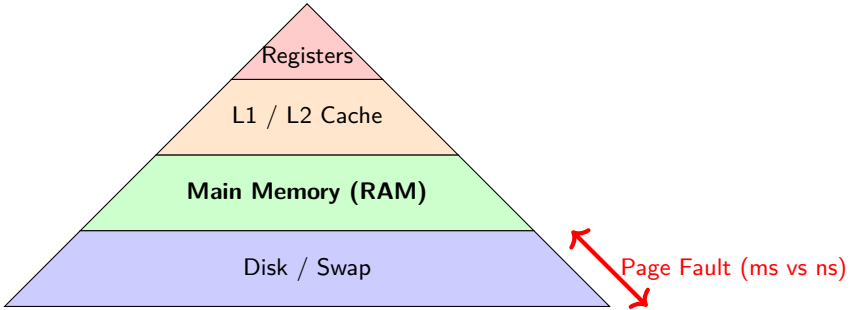
Option B: Comparative Analysis

- **Goal:** Scientific Comparison.
- **Metrics:** Hit Rate, Latency, Thrashing onset point.

Outline

Theory 1: The Memory Hierarchy

Why replace pages? Because faster memory is scarcer.

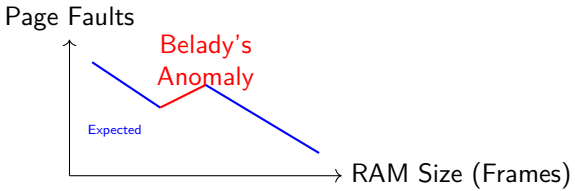


Theory 2: The Extremes - OPT and FIFO

Before designing your algorithm, understand the boundaries.

1. **The Optimal (OPT / MIN)**
   - **Logic:** Evict the page that will not be used for the longest time in the future.
   - **Status:** Impossible to implement (requires predicting the future).
   - **Use:** Benchmark only.

2. **FIFO (First-In First-Out)**
   - **Logic:** Evict the oldest page.
   - **Fatal Flaw: Belady's Anomaly**.
   - *Adding more RAM can actually INCREASE page faults!*

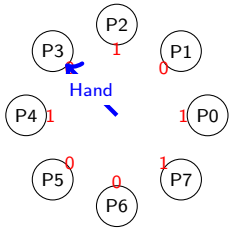


Theory 3: Hardware Reality - The Clock Algorithm

True LRU is too expensive (moving pointers on every memory access). OS uses an approximation called **Second Chance (Clock)**.

The Mechanism:

- Pages are arranged in a circular buffer.
- Each page has a hardware **Reference Bit (R)**.
- **On Access:** Hardware sets R = 1.
- **On Eviction (The Hand scans):**
  - If R==1: Give "Second Chance", set R=0, move next.
  - If R==0: Evict this page.



Theory 4: Addressing Frequency (LFU)

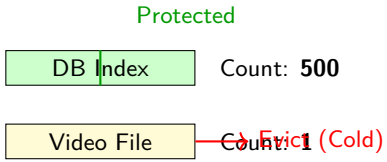
LRU fails when "History doesn't predict Future" (e.g., Database Scans).

The Problem: Scan Pollution

Reading a 10GB file once fills all 8GB RAM with useless pages, evicting frequently used hot pages.

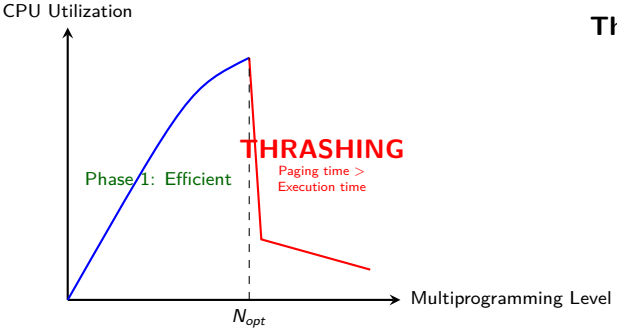
LFU (Least Frequently Used)

- **Logic:** Track `access_count`. Evict the page with the lowest count.
- **Benefit:** Resistant to scans.
- **Implementation:** Add a counter to `struct page` (This is Advanced Option A).



Theory 5: Thrashing

What happens if we run too many processes at once?



The Definition:

- Thrashing occurs when the system spends more time **paging** (swapping in/out) than **executing**.
- **Symptom:** CPU utilization drops to nearly 0%, while Disk I/O goes to 100%.
- **The Trap:** The scheduler sees low CPU usage $\rightarrow$ Loads *more* processes $\rightarrow$ Situation gets worse.

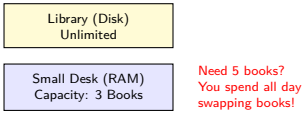
Theory 6: Preventing Thrashing (Working Set)

The Root Cause: Total size of localities $>$ Total Physical Memory.

$$\sum \text{Size of Working Set}_i > \text{Total RAM}$$

Solution 1: Working Set Model (Your Project)

- Track the set of pages a process is *actually* using (Working Set).
- **Policy:** If free RAM $<$ Working Set of new process, **Suspend** the process (Swap it out completely). Don't let it run halfway.



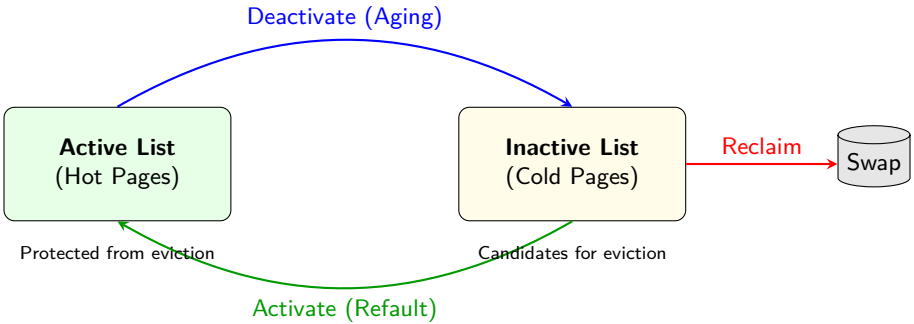
Solution 2: Page-Fault Frequency (PFF)

- If Fault Rate $>$ Upper Bound: Give more frames.
- If Fault Rate $<$ Lower Bound: Take back frames.

Outline

Kernel Mechanics 1: The "Active/Inactive" Lists

Linux uses a "2Q" approximation of LRU to handle Scan Resistance.



Kernel Mechanics 2: Key Structures

File: `include/linux/mm_types.h`

struct folio (formerly struct page)

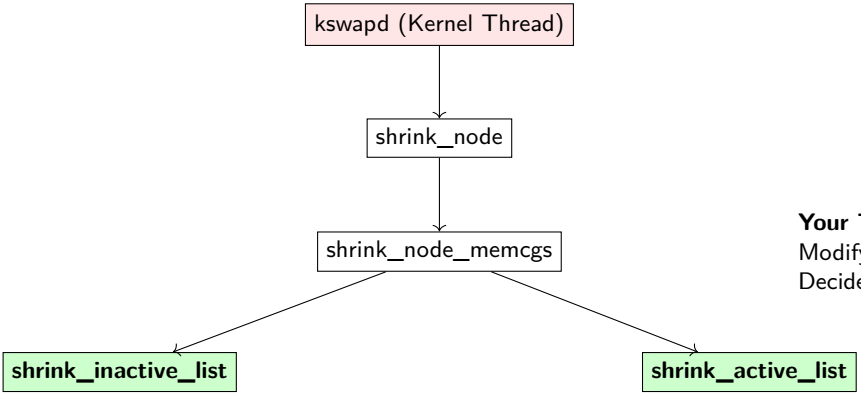
- The fundamental unit of memory.
- **Flags:**
  - `PG_active`: Is it in the Active list?
  - `PG_referenced`: Was it touched recently?
- **LRU List:** A linked list node connecting pages.

```
1  struct folio {
2      // ...
3      unsigned long flags;
4      struct list_head lru;
5      // ...
6  };
7
8  // In mm/vmscan.c
9  void shrink_active_list(...) {
10     // Logic to move pages
11     // Active -> Inactive
12 }
13
```

Outline

Step 1: The Call Graph

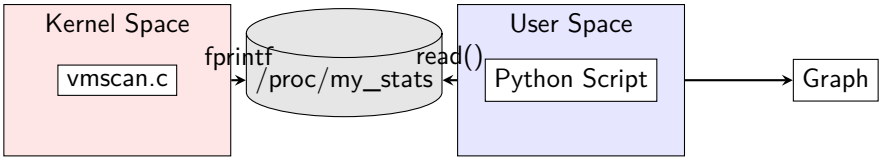
Where exactly do you inject your code? `mm/vmscan.c` is the battlefield.



Your Task:
Modify logic in these two functions.
Decide WHO moves between lists.

Step 2: The Visualization Pipeline

How to see inside the kernel? Use the Producer-Consumer model.



Resources & Next Steps

Action Plan:

1. **Setup:** Compile a vanilla Linux kernel inside QEMU/KVM.
2. **Trace:** Add `printk("Evicting page!")` in `shrink_page_list`.
3. **Observe:** Run a memory hog and watch `dmesg`.
4. **Modify:** Change the logic (e.g., protect pages with high access counts).

Resources:

- *Understanding the Linux Kernel*: Chapter on Page Frame Reclamation.
- Source code: `mm/vmscan.c`, `mm/rmap.c`.