Chapter 22: Beyond Physical Memory: Policies

This chapter focuses on a critical aspect of virtual memory management: the **page-replacement policy**. When physical memory is full and a page fault occurs, the OS must decide which page to evict (or "page out") to make room for the new page being brought in. This decision is fundamental to system performance.

> **The Crux of the Problem: How to Decide Which Page to Evict?**
>
> How can the OS decide which page (or pages) to evict from memory? This decision is made by the replacement policy of the system, which usually follows some general principles but also includes certain tweaks to avoid corner-case behaviors.

1. Cache Management and Performance

Main memory can be viewed as a **cache** for the much larger virtual address space stored on disk. The goal of a replacement policy is to **minimize cache misses** (page faults) and **maximize cache hits**.

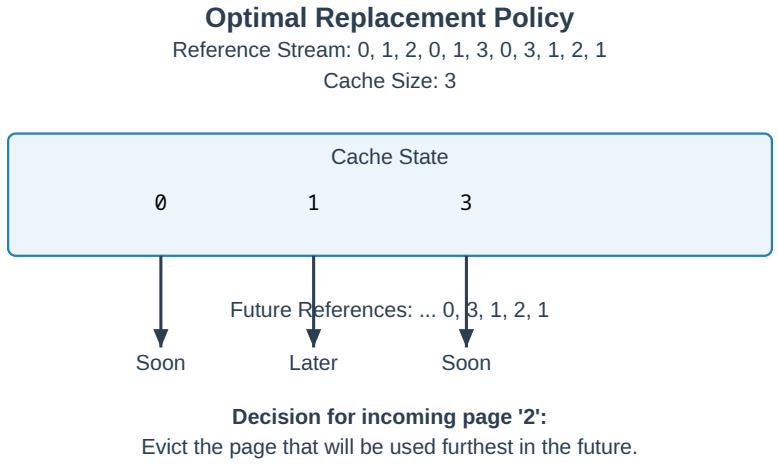
- **Average Memory Access Time (AMAT):** This metric quantifies the performance of the memory system.
  - **Formula:** $AMAT = T_{memory} + (P_{miss} \cdot T_{disk})$
  - T_{memory}: Cost of accessing memory (fast, e.g., 100 ns).
  - T_{disk}: Cost of accessing disk (very slow, e.g., 10 ms).
  - P_{miss}: The probability of a page fault (a cache miss).
- **Impact:** Because T_{disk} is orders of magnitude larger than T_{memory}, even a tiny miss rate can dominate the AMAT and make the system feel extremely slow. Therefore, a smart replacement policy is essential.

---

2. The Optimal Replacement Policy

To evaluate other policies, we first need a baseline for perfection. The **Optimal (OPT)** replacement policy, also known as MIN, provides this.

- **Policy:** When a page must be evicted, replace the page that will be accessed **furthest in the future**.
- **Nature:** This policy is unrealizable in practice because it requires knowledge of the future.
- **Utility:** It is extremely useful as a benchmark in simulations. By comparing a real-world policy to OPT, we can understand how close to perfect it is.



3. Simple Policies: FIFO and Random

Early systems used simple policies that are easy to implement.

- **First-In, First-Out (FIFO):**
  - **Policy:** The page that was brought into memory first is the first one to be evicted, like a simple queue.
  - **Problem:** FIFO is oblivious to a page's importance. It might evict a frequently used page simply because it was loaded a long time ago.
  - **Belady's Anomaly:** FIFO can exhibit the strange behavior where increasing the cache size *decreases* the hit rate for certain access patterns.
- **Random:**
  - **Policy:** Evicts a random page from memory.
  - **Properties:** Simple and avoids the worst-case behaviors that can plague more structured policies like FIFO and LRU. Its performance, however, is unpredictable and depends on luck.

---

4. Using History: LRU and LFU

To make smarter decisions, schedulers can use past behavior to predict future behavior, a concept known as the **principle of locality**.

- **Temporal Locality:** Pages accessed recently are likely to be accessed again soon.
- **Spatial Locality:** If a page is accessed, nearby pages are likely to be accessed soon.

This leads to history-based algorithms:

- **Least Recently Used (LRU):**
  - **Policy:** Evicts the page that has not been accessed for the longest period of time.
  - **Rationale:** If a page hasn't been used recently, it is likely not going to be used in the near future (based on temporal locality).
  - **Performance:** Generally performs much better than FIFO or Random, often approaching Optimal for workloads with good locality.
  - **Implementation Challenge:** Perfect LRU is expensive. It requires updating a data structure on *every single memory reference* to track access times, which would be prohibitively slow.

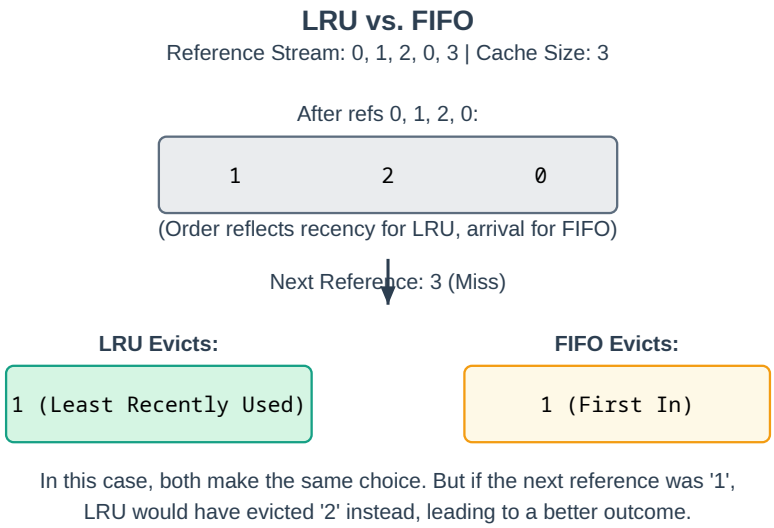


Figure 1: LRU vs. FIFO Policy Trace

- **Least Frequently Used (LFU):**
  - **Policy:** Evicts the page that has been accessed the fewest number of times.
  - **Problem:** A page might be used heavily at the beginning of a program and then never again, but LFU would keep it in memory.

---

5. Approximating LRU: The Clock Algorithm

Since perfect LRU is too costly, modern systems use an approximation that requires minimal hardware support.

- **Hardware Support: The Use Bit:** Each page in memory has a **use**

bit (or reference bit) associated with it.
  - When a page is referenced (read or written), the **hardware automatically sets the use bit to 1**.
  - The hardware *never* clears this bit; that is the OS's job.
- **The Clock Algorithm:**
  1. The OS arranges all physical pages in a circular list, with a "clock hand" pointing to one of them.
  2. When a page needs to be evicted, the OS checks the page pointed to by the clock hand.
  3. If its use bit is **1**, it means the page was recently used. The OS clears the bit (sets it to **0**) and advances the clock hand to the next page.
  4. If its use bit is **0**, it means the page has not been used recently. This page is chosen as the victim to be replaced.
  5. This process continues until a page with a use bit of 0 is found.

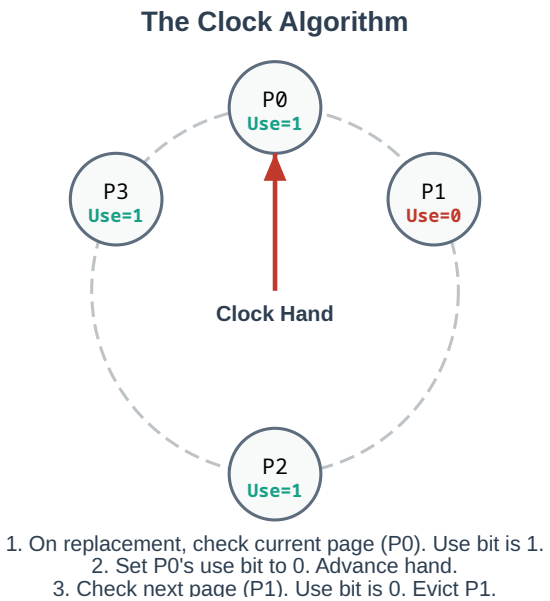


Figure 2: The Clock Algorithm

- **Considering Dirty Pages:** The algorithm can be improved by also considering the **dirty bit** (or modified bit), which the hardware sets when a page is written to.
  - Evicting a **clean** page (not modified) is fast because it can just be overwritten.
  - Evicting a **dirty** page is slow because its contents must first be written back to disk.
  - An enhanced clock algorithm would prefer to evict clean, unused pages over dirty, unused pages.

---

6. Other Virtual Memory Policies

- **Page Selection Policy:** Decides *when* to bring a page into memory.
  - **Demand Paging:** The standard approach. A page is loaded only when it is accessed (on demand).
  - **Prefetching:** The OS guesses that a page will be needed soon and loads it proactively (e.g., loading page `P+1` when page `P` is accessed).
- **Write Policy:** Decides *when* to write dirty pages to disk.
  - **Clustering (or Grouping):** The OS collects multiple dirty pages in memory and writes them to disk in a single, efficient I/O operation.

---

7. Thrashing

- **Definition:** A state where the system is constantly paging to and from disk because the memory demands of the running processes far exceed the available physical memory.
- **Effect:** System performance grinds to a halt, as the CPU spends most of its time waiting for the disk.
- **Solutions:**
  - **Admission Control:** An OS can detect thrashing and decide to suspend (or "swap out") one or more processes to reduce memory pressure.
  - **Out-Of-Memory (OOM) Killer:** A more drastic approach used by systems like Linux. When memory is critically low, a daemon chooses a memory-intensive process and kills it.

---

8. Summary

Page replacement policies are crucial for the performance of a virtual memory system.

- **Optimal (MIN)** is the ideal but unrealizable benchmark.
- Simple policies like **FIFO** and **Random** are easy to implement but often perform poorly.
- History-based policies like **LRU** perform well by leveraging locality but are expensive to implement perfectly.
- The **Clock algorithm** provides an efficient approximation of LRU using a simple **use bit** provided by hardware.
- Considering the **dirty bit** can further optimize eviction decisions by preferring to replace clean pages.
- When a system is **thrashing**, the only real solution is to reduce the number of active processes or add more memory.