# Operating Systems
# Tutorial 11: Page Replacement Algorithms

Yonghao Lee

June 9, 2025

# 1 Reminder: The Page Replacement Problem

## 1.1 Virtual Memory and the Replacement Challenge

Modern operating systems use virtual memory to give each process its own virtual address space (VAS) that maps to physical memory. Since physical memory (RAM) is typically much smaller than the combined virtual address spaces of all running processes, we face a fundamental challenge: when RAM runs out of available frames and we need to load a new page, **which existing page should we remove to make space?**

This decision is critical because we want to avoid removing pages that we'll need again soon. However, since we cannot predict future memory accesses, we must rely on heuristics based on past behavior. The page we choose to remove is called the **victim page**.

## 1.2 Why Simple Approaches Fall Short

A naive approach might be to select victim pages randomly, but this ignores usage patterns entirely. Consider what happens if we randomly evict a page that the program accesses frequently - we'll need to reload it almost immediately, creating unnecessary overhead.

Some pages are particularly costly to evict:

- **Frequently accessed data pages** that the program uses repeatedly will cause immediate page faults when accessed again
- **Page table pages** are especially critical - evicting a page table page containing 15 valid entries can multiply the page fault rate by approximately 15, since accessing any of those virtual pages now requires loading the page table page first

***Note:*** *Page replacement algorithms work independently of address translation. The process of converting virtual addresses to physical addresses is separate from the process of selecting which pages to evict.*

## 1.3 Types of Replacement Algorithms

Several page replacement strategies have been developed, each with different performance characteristics:

- **Optimal** - Provides the best theoretical performance but cannot be implemented in practice
- **FIFO (First-In-First-Out)** - Simple to implement but can suffer from Belady's anomaly
- **Second Chance FIFO** - Improves on basic FIFO by considering recent usage
- **NRU (Not Recently Used)** - Uses reference and modify bits for practical approximation
- **LRU (Least Recently Used)** - Excellent performance but requires significant overhead
- **Pseudo-LRU** - Approximates LRU behavior with reduced implementation cost

- **LFU (Least Frequently Used)** - Makes decisions based on access frequency
- **Random** - Simple but unpredictable performance

## 1.4 The Optimal Algorithm: A Theoretical Benchmark

The Optimal algorithm (also known as Belady's optimal page replacement policy) serves as our theoretical gold standard. It works by examining all future memory accesses and always replacing the page that won't be used for the longest time ahead.

While this algorithm minimizes page faults for any reference sequence, it's impossible to implement in real systems since it requires perfect knowledge of future memory accesses. Instead, it serves as a benchmark for comparing practical algorithms.

### 1.4.1 Optimal Algorithm Example

Let's trace through the Optimal algorithm using this reference string with 4 available frames:

$$0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1$$

**Initial accesses (0, 2, 1, 6):** Since RAM starts empty, all four accesses result in page faults. RAM now contains: {0, 2, 1, 6}.

**Access 4:** RAM is full, so we need a victim. Looking at future accesses:

- Page 0: appears next in the sequence
- Page 2: appears later in the sequence
- Page 1: appears soon after page 0
- Page 6: never appears again

Since page 6 will never be used again, it's the clear choice for replacement.

**Subsequent accesses (0, 1, 0):** All hit since these pages are in RAM.

**Access 3:** Need another victim. Looking ahead:

- Page 0: not used again in the remaining sequence
- Page 4: not used again in the remaining sequence

Either page 0 or 4 could be victims since both are never referenced again. We choose page 0.

**Final result:** 6 page faults out of 12 accesses gives a fault rate of $\frac{6}{12} = 0.50$, which represents the absolute best performance possible for this reference string.

## 1.5 FIFO Algorithm

FIFO replaces the page that has been in memory the longest, regardless of how recently or frequently it has been used.

Using the same reference string $[0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1]$ with 4 frames:

- **Initial loads (0, 2, 1, 6):** Four page faults, RAM: {0, 2, 1, 6}
- **Access 4:** Evict oldest page (0), RAM: {4, 2, 1, 6}
- **Access 0:** Evict oldest page (2), RAM: {4, 0, 1, 6}
- **Accesses 1, 0:** Both hit
- **Access 3:** Evict oldest page (1), RAM: {4, 0, 3, 6}
- **Access 1:** Evict oldest page (6), RAM: {4, 0, 3, 1}
- **Access 2:** Evict oldest page (4), RAM: {2, 0, 3, 1}
- **Access 1:** Hit

This results in 9 page faults for a fault rate of $\frac{9}{12} = 0.75$ - significantly worse than optimal.

FIFO's main weaknesses are that it completely ignores usage patterns and systematically targets long-resident pages, which are often the most important ones to keep in memory.

## 1.6 Second Chance FIFO

Second Chance FIFO improves on basic FIFO by incorporating usage information through reference bits, giving recently accessed pages a "second chance" before eviction.

**How it works:**

1. Each page has a reference bit (R) set to 1 when accessed
2. Pages are organized in a FIFO queue
3. When selecting a victim:
   - Check the oldest page's reference bit
   - If R = 0: replace this page (hasn't been used recently)
   - If R = 1: clear the bit (R = 0), move page to end of queue, and continue searching

This approach combines FIFO's simplicity with consideration for recent usage, often performing much better than pure FIFO.

## 1.7 Least Recently Used (LRU)

LRU replaces the page that hasn't been accessed for the longest time, leveraging the principle of temporal locality - the idea that recently accessed pages are likely to be accessed again soon.

Using our example reference string with 4 frames:

- **Initial loads (0, 2, 1, 6):** Four page faults
- **Access 4:** Replace least recently used page (0)
- **Access 0:** Replace least recently used page (2)

- **Accesses 1, 0:** Both hit, update usage order
- **Access 3:** Replace least recently used page (6)
- **Access 1:** Hit, update usage order
- **Access 2:** Replace least recently used page (4)
- **Access 1:** Hit

LRU achieves 8 page faults for a fault rate of $\frac{8}{12} = 0.67$ - better than FIFO and closer to optimal performance.

### 1.8 Understanding Replacement Algorithm Performance

Since physical memory is limited, page replacement is inevitable in most systems. The ideal strategy would be to evict pages that won't be needed for the longest time in the future, but this requires perfect knowledge of future memory accesses.

Instead, practical algorithms use heuristics based on locality principles:

- **Temporal locality**: Recently accessed pages are likely to be accessed again soon
- **Spatial locality**: Pages near recently accessed ones are also likely to be needed

These heuristics work well because of typical program behavior patterns, though they cannot guarantee optimal performance since they rely on predictions rather than perfect future knowledge.

## 2 Exam Questions

### 2.1 Question A

Consider the following code snippet that initializes an array of integers:

```
for (int i = 0; i < 2^29; i++) {
  numbers[i] = 0;
}
```

Given these system parameters:

- Physical memory: $2^{32}$ bytes divided into frames of $2^{12}$ bytes each
- Integer data type: 4 bytes
- Array elements are stored contiguously starting at the beginning of the first page
- The entire code fits in one page
- The system uses demand paging with the loop counter `i` stored in a register
- Initially, only page tables are in memory (and they remain there throughout execution)

- The process is allocated $2^{12}$ frames in memory

**Question:** Using LRU page replacement, how many page faults will occur during execution?

**Solution:** *We need to count page faults from two sources: loading the program code and accessing array data.*

***Code page fault:*** *Since the system uses demand paging and initially contains only page tables, the first instruction fetch will trigger a page fault to load the single code page.*

$$\textit{Code page faults} = 1$$

***Array data page faults:*** *Let's calculate how many pages the array spans:*

- *Array size:* $2^{29}$ *elements* $\times$ 4 *bytes/element* $= 2^{31}$ *bytes*
- *Number of pages needed:* $\frac{2^{31}\ \textit{bytes}}{2^{12}\ \textit{bytes/page}} = 2^{19}$ *pages*

*Since the array is accessed sequentially, each of these* $2^{19}$ *pages will be touched exactly once, causing one page fault per page.*

$$\textit{Array data page faults} = 2^{19}$$

***Total page faults:***

$$\textit{Total} = 2^{19} + 1$$

## 2.2 Question B: FIFO with Second-Chance (Clock Algorithm)

Using the same parameters as Question A, but with the FIFO with Second-Chance (Clock) algorithm instead of LRU.

**Question:** Determine the exact number of page faults during one complete execution of the loop.

### 2.2.1 Clock Algorithm Overview

The Clock algorithm maintains a circular list of frames with a rotating "clock hand" pointer:

- Each frame has a reference bit (R) that gets set to 1 when the page is accessed
- When replacement is needed, the clock hand moves clockwise
- For each frame encountered:
  - If R = 0: evict this page (victim found)
  - If R = 1: set R = 0 (give second chance) and continue

### 2.2.2 Analysis Strategy

We'll analyze the page faults in two categories:

$$\text{Total Faults} = \text{Array Data Faults} + \text{Code Page Faults}$$

**Array data analysis:** The array requires $2^{19}$ pages and is accessed sequentially. Each page is accessed exactly once, so we get exactly $2^{19}$ page faults for array data.

**Code page analysis:** This is more complex and requires detailed examination of the Clock algorithm's behavior.

### 2.2.3 Detailed Code Page Behavior

**Phase 1 - Initial memory filling:**

1. Code page C loaded into frame 0 (1st page fault)
2. Loop executes, loading array pages 0 through 4094 into frames 1 through 4095
3. All frames now have R = 1 from recent access
4. Memory is completely full

**Phase 2 - First replacement cycle:**

1. Need to load array page 4095, but no free frames
2. Clock hand starts at frame 0 (code page C)
3. First sweep: All frames have R = 1, so clock hand sets each R = 0 and continues
4. Second sweep: Clock hand returns to frame 0, finds R = 0, evicts code page C
5. Array page 4095 loaded into frame 0 (code page evicted)
6. CPU tries to fetch next instruction $\rightarrow$ page fault! (2nd page fault)
7. Clock hand at frame 1: data page has R = 0, gets evicted
8. Code page C reloaded into frame 1 (3rd page fault)

**Phase 3 - Steady state:** From this point forward, the code page is never evicted again because:

- Code page is accessed every loop iteration (constant R bit refreshing)
- Clock hand takes $2^{12}$ steps for a full revolution
- During each revolution, code page gets accessed thousands of times
- When clock hand reaches code page: R = 1, so it gets second chance (R set to 0)
- Before hand could return to code page, it's been accessed many more times
- Clock hand always finds data pages with R = 0 first

**Final result:**

$$\begin{aligned}\text{Total Page Faults} &= \text{Array Data Faults} + \text{Code Page Faults} \quad (1)\\ &= 2^{19} + 2 \quad (2)\end{aligned}$$

The breakdown is:

- $2^{19}$ faults from sequential array access
- 1 fault from initial code page load
- 1 fault from code page eviction and immediate reload

After the initial reload, the code page's constant access pattern ensures it never gets evicted again, making this the final answer.