CS3701 - Operating Systems

Chapter 8 - Virtual Memory

Department of Computer Science
College of Computer Engineering and Science
Prince Sattam bin Abdulaziz University

AY - 2025/2026

Prepared by Dr. Mahdi Khemakhem, Reviewed by Dr. Essra Aldessouky
Updated on January 22, 2026

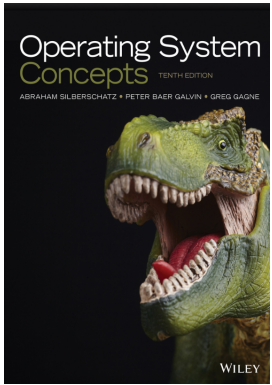
Outline

Textbook Reference

Required Reading

Refer to Chapter 10 of the textbook:

- **Title:** *Operating System Concepts*
- **Authors:** Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne
- **Publisher:** John Wiley & Sons
- **Edition:** 10th Edition (2018)



Outline

Virtual Memory

Virtual Memory Definition

Virtual memory separates the logical (virtual) memory seen by processes from the physical memory available in the system

Key Benefits:

- **Only part of a program** needs to be in memory for execution
- **Logical address space** can be *much larger* than physical address space
- **Multiple processes** can share physical memory efficiently
- More **efficient process creation** and management
- More programs running **concurrently**
- *Increases CPU utilization* and system *throughput*

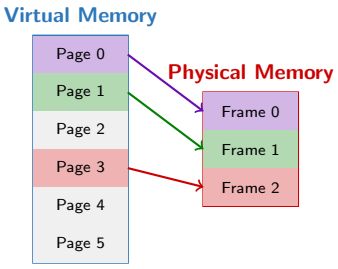


Figure 1: Virtual Memory Larger than Physical

Virtual Memory - How It Works

Operating Mechanism:

- **Only needed pages** reside in physical memory
- **When a page is referenced:**
  - **If in memory:** Continue execution normally
  - **If not in memory:** *Page fault* occurs
    - Bring page from **backing store** to a free frame
    - **If no free frame:** Use *page replacement* algorithm to select a victim, swap it out, then load needed page

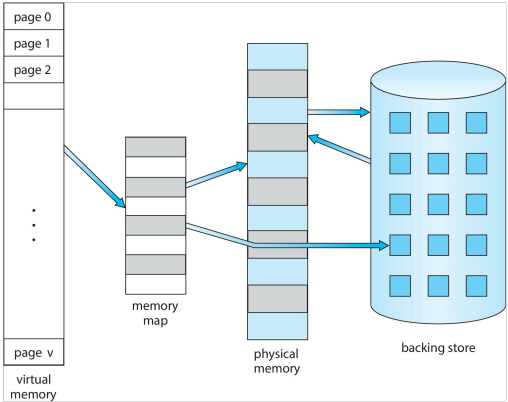


Figure 2: Diagram showing virtual memory that is larger than physical memory

Outline

Demand Paging - Introduction

Demand Paging Concept

With demand paging, pages are loaded into memory only when needed during execution, not in advance

Comparison with Traditional Approach:

- **Traditional (Eager Loading):** Load entire process into memory at start
- **Demand Paging (Lazy Loading):** Load pages **only when referenced**

Advantages:

- **Less I/O:** Only needed pages are loaded
- **Less memory:** Unused pages never loaded
- **Faster response:** Process starts immediately
- **More processes:** Better memory utilization

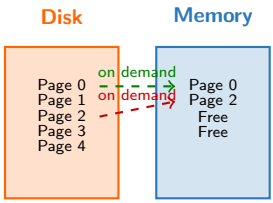


Figure 3: Demand Paging

⇒**A pager (lazy swapper) only loads pages when they are actually referenced, implementing demand paging strategy**

Demand Paging - Handling Page References

When a process references a page:

1. **Check page table:**
   - **Valid bit (v):** Page is in memory → access it
   - **Invalid bit (i):** Page not in memory → *page fault*
2. **On page fault:**
   - Check if reference is **valid** (abort if illegal)
   - Find a **free frame** (or select **victim**)
   - Load page from **backing store**
   - Update **page table** (set valid bit)
   - *Restart* the instruction

Performance Consideration

The critical performance factor is the page-fault service time

Page Replacement Necessity

Page replacement is needed when physical memory is full and a page fault occurs

Demand Paging - Steps

1. If there is a reference to a page, first reference to that page will trap to operating system
   - *Page fault*
2. Operating system looks at another table to decide:
   - **Invalid reference** ⇒ abort
   - **Just not in memory**
3. **Find free frame**
4. **Swap page into frame** via scheduled disk operation
5. **Reset tables** to indicate page now in memory
   - Set validation bit = **v**
6. *Restart the instruction* that caused the page fault

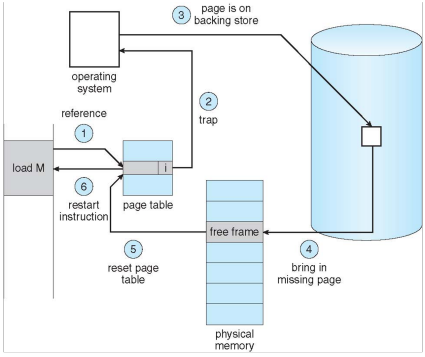


Figure 4: Demand paging process

What Happens if There is No Free Frame?

If there is no free frame, the operating system must select a victim frame to evict using a page replacement algorithm

1. If victim page is **modified** (**dirty**[a]), write it back to disk
2. **Bring in the desired page** into the newly freed frame
3. **Update the page and frame tables**
4. *Restart the instruction* that caused the trap

Note: If no free frame, **two page transfers** are required (victim out, desired in), effectively *doubling the page-fault service time*

[a] A dirty page is a page whose contents have been modified in memory but not yet written to disk

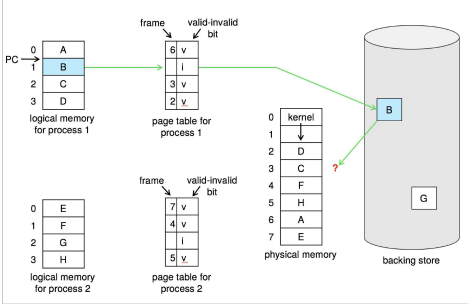


Figure 5: Handling no free frame situation

Page Replacement

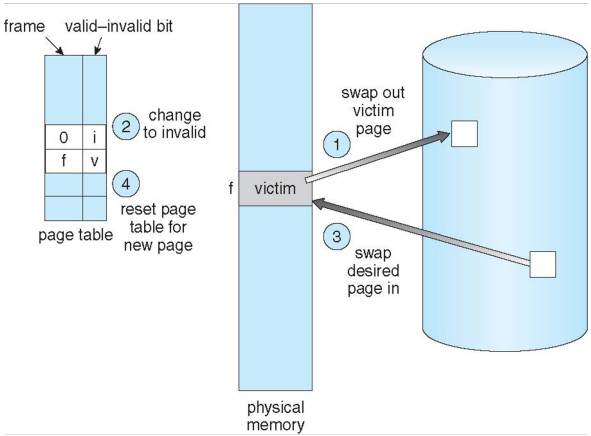


Figure 6: Page replacement process when no free frame is available

Key Takeaways

Key Takeaways

- **Virtual memory** allows processes to use more memory than physically available
- **Demand paging** loads pages only when needed, improving memory utilization
- **Page faults** occur when a referenced page is not in memory
- **Page replacement algorithms** select which pages to evict when memory is full
- **Effective page replacement strategies are crucial for system performance $\Rightarrow$ minimize page-fault rate**

Outline

Page Replacement - Algorithm Evaluation

Goal of Page Replacement

Want lowest page-fault rate on both first access and re-access

- **Frame-allocation algorithm** determines:
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm:** Selects which frame to replace
- **Evaluate algorithm** by running it on a particular string of memory references (*reference string*) and computing the number of page faults
  - String is just **page numbers**, not full addresses
  - Repeated access to the same page **does not cause** a page fault
  - Results depend on **number of frames available**
- **Reference String Example - In all our examples, the reference string of referenced page numbers is: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1**

Page Faults vs. Number of Frames

- **General trend:** More frames → fewer page faults
- **Exception:** Some algorithms may exhibit *Belady's Anomaly*
- **Asymptotic behavior:** Eventually, adding more frames yields diminishing returns

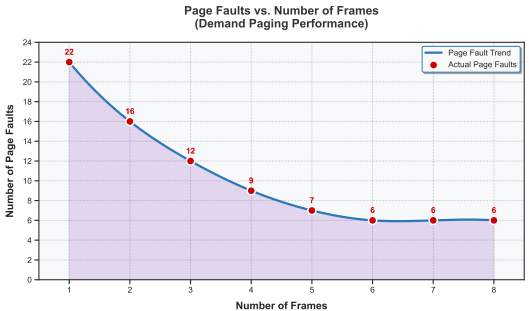


Figure 7: Graph of page faults versus the number of frames

First-In-First-Out (FIFO) Algorithm

FIFO Principle

Replace the oldest page in memory (first page loaded in the memory is first to be replaced)

Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Number of Frames: 3 Total Page Faults: 15

Ref.	Init	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
Frame 1	–	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	0	0	7	7	7
Frame 2	–	–	0	0	0	0	3	3	3	2	2	2	2	2	2	2	1	1	1	1	1	0	0
Frame 3	–	–	–	1	1	1	1	0	0	0	3	3	3	3	3	3	3	2	2	2	2	2	1
Fault	–	✗	✗	✗	✗	–	✗	✗	✗	✗	✗	✗	–	–	–	–	✗	✗	–	–	✗	✗	✗

- ✗: Initial page faults (filling empty frames)
- ✗ Page faults requiring replacement (victim selected by FIFO order)
- –: Page hit (no fault, page already in memory)

FIFO Algorithm - Characteristics

- **Simple to implement:** Maintain a queue of pages in memory
- **Easy to understand:** Oldest page is replaced first
-
- **Not optimal:** Can lead to suboptimal performance compared to other algorithms
- **May suffer from Belady's Anomaly:** Increasing number of frames can increase page faults

Belady's Anomaly - Illustration

- **Reference String:** 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- **3 frames** $\rightarrow$ **9 page faults**
- **4 frames** $\rightarrow$ **10 page faults**
- More frames = MORE faults!

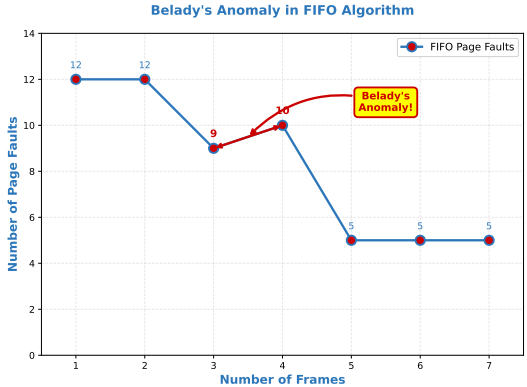


Figure 8: FIFO illustrating Belady's Anomaly

Temporal Locality and Belady's Anomaly

Temporal Locality

Temporal Locality: If a page is referenced, it is **likely to be referenced again soon**

Comparison of Reference Strings in terms of Temporal Locality:

Good Temporal Locality

String A: 1, 2, 1, 2, 1, 2, 3, 3, 3

- Pages **1, 2, 3** are reused frequently
- **Short intervals** between reuses
- **Result:** FIFO performs well, **stable with more frames**

Poor Temporal Locality

String B: 1, 2, 3, 4, 5, 1, 2, 3, 4, 5

- Pages referenced **far apart**
- **Long intervals** between reuses
- **Result:** FIFO struggles, **risk of Belady's Anomaly**

Why Poor Locality Causes Belady's Anomaly:

- With **few frames**, pages get replaced before being reused → **many faults**
- With **more frames**, FIFO's rigid order conflicts with access pattern → **even more faults!**
- The **oldest page** might be needed soon, but FIFO replaces it anyway

Belady's Anomaly: Detailed Example with FIFO (3 Frames)

Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 **Frames:** 3 **Page Faults:** 9

Reference	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	4	4	4	5	5	5	3	3	3
Frame 2	–	2	2	2	2	2	2	2	2	2	4	4
Frame 3	–	–	3	3	1	1	1	1	1	1	1	5
Fault	✗	✗	✗	✗	✗	–	✗	–	–	✗	✗	✗

Observation with 3 Frames:

- **Steps 1-3:** Initial fills (3 faults)
- **Step 4:** Replace page 1 with 4 (FIFO: 1 is oldest) → page 1 needed soon!
- **Step 5:** Replace page 2 with 1 (FIFO: 2 is oldest) → page 2 needed soon!
- **Step 6:** Hit on page 2 (still in memory)
- **Step 7:** Replace page 3 with 5 (FIFO: 3 is oldest)
- **Steps 8-9:** Hits on pages 1, 2 (fortunate timing!)
- **Steps 10-12:** More replacements needed (3 faults)
- **Total: 9 page faults**

Belady's Anomaly: Detailed Example with FIFO (4 Frames)

Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 Frames: 4 Page Faults: 10 (WORSE!)

Reference	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	5	5	5	5	4	4
Frame 2	–	2	2	2	2	2	2	1	1	1	1	5
Frame 3	–	–	3	3	3	3	3	3	2	2	2	2
Frame 4	–	–	–	4	4	4	4	4	4	3	3	3
Fault	✗	✗	✗	✗	–	–	✗	✗	✗	✗	✗	✗

Observation with 4 Frames:

- **Steps 1-4:** Initial fills (4 faults) - all pages fit!
- **Steps 5-6:** Hits on pages 1, 2 (all in memory)
- **Step 7:** Replace page 1 with 5 (FIFO: Frame 1 is oldest) → *page 1 needed next!*
- **Step 8:** Page 1 not in memory → fault, replace page 2 with 1 (FIFO: Frame 2 is oldest)
- **Step 9:** Page 2 not in memory → fault, replace page 3 with 2 (FIFO: Frame 3 is oldest)
- **Step 10:** Page 3 not in memory → fault, replace page 4 with 3 (FIFO: Frame 4 is oldest)
- **Step 11:** Page 4 not in memory → fault, replace page 5 with 4 (FIFO: Frame 1 is oldest)
- **Step 12:** Page 5 not in memory → fault, replace page 1 with 5 (FIFO: Frame 2 is oldest)
- **Total: 10 page faults (1 more than with 3 frames!)**

Belady's Anomaly: Why More Frames = More Faults?

Comparison Summary:

Configuration	Frames	Page Faults	Result
3 Frames	3	9	Better
4 Frames	4	10	Worse (Anomaly!)

Root Cause Analysis:

With 3 Frames

- **Faster cycling** through pages
- Pages replaced **before** long gaps
- Some **lucky hits** at steps 6, 8, 9
- FIFO order happens to align better with reuse pattern

With 4 Frames

- **Slower cycling** keeps old pages longer
- Page 1 held until step 7, then *needed at step 8!*
- Creates **chain reaction** of faults (steps 8-12)
- FIFO order conflicts with reuse pattern

Belady's Anomaly: Key Insight

Belady's Anomaly Explanation

Key Insight: With poor temporal locality, FIFO's rigid replacement order can create **worse timing** when more frames are available. The oldest page (per FIFO) may be the *next one needed*, causing a cascade of unnecessary faults!

Why This Matters:

- **Challenges intuition:** More resources → worse performance!
- **Algorithm-specific:** Only affects FIFO and some FIFO-based algorithms
- **Stack algorithms[1] immune:** LRU, Optimal never exhibit this anomaly (**see next**)
- **Design implication:** Prefer stack algorithms for predictable behavior

[1] Stack algorithms maintain an inclusion property: the set of pages in memory with m frames is always a **subset** of the set with $m+1$ frames at any point in time, i.e., $M(m) \subseteq M(m+1)$. This prevents anomalies like Belady's.

Least Recently Used (LRU) Algorithm

LRU Algorithm Principle

Use past knowledge about page usage to make better replacement decisions. Replace page that has not been used (referenced) for the longest time

Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Number of Frames: 3 **Total Page Faults: 12**

Ref.	Init	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
Frame 1	–	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	1	1	1	1	1	1	1
Frame 2	–	–	0	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3	0	0	0	0	0
Frame 3	–	–	–	1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	2	2	7	7	7
Fault	–	✗	✗	✗	✗	–	✗	–	✗	✗	✗	✗	–	–	–	–	✗	–	✗	–	✗	–	–

LRU Algorithm - Characteristics

- **Generally good algorithm:** 12 page faults (better than FIFO, worse than OPT in this example) **(see next algorithm)** - **The best practical algorithm**
- **Uses past knowledge:** Tracks when each page was last accessed
- **Does not suffer** from Belady's Anomaly (it's a stack algorithm)
- **Implementation challenge:** Requires tracking access time for every page

Implementing LRU

LRU is practical and implementable, using historical data (the past) to make replacement decisions. It performs well in most scenarios.

Optimal (OPT) Page Replacement Algorithm

Optimal Algorithm Principle

Replace page that **will not be used** for the **longest period of time**. Use **future knowledge** of page references (rather than past knowledge) to make replacement decisions.

Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Number of Frames: 3 **Total Page Faults:** 9

Ref.	Init	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
Frame 1	–	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	1	1	1	1	1	1	1
Frame 2	–	–	0	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3	0	0	0	0	0
Frame 3	–	–	–	1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	2	2	7	7	7
Fault	–	✗	✗	✗	✗	–	✗	–	✗	✗	✗	✗	–	–	–	–	✗	–	✗	–	✗	–	–

Bridging Past and Future

Key Contrast: LRU looks at the **past** (what was used), OPT looks at the **future** (what will be used). Modern AI/ML bridges this gap by **predicting the future** from past patterns!

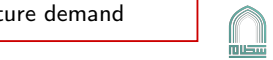
Optimal Algorithm - Characteristics & Modern Perspective

- **Lowest possible page-fault rate:** Only 9 faults for the reference string (best among all algorithms)
- **Not implementable traditionally:** theoretically optimal but **not easy to implement** in practice since it requires future knowledge of page references.
- **Used as benchmark:** Provides theoretical lower bound to compare other algorithms
- **Does not suffer** from Belady's Anomaly

Modern OS: ML-Based Predictive Systems

Approximating the future: Modern OS research uses **Machine Learning** to predict future page access patterns:

- **Workload analysis:** Learn application behavior to predict next page references
- **Approaching OPT:** ML models can approach optimal performance by forecasting *what pages will be needed*
- **Adaptive systems:** Neural networks trained on historical patterns to predict future demand



Least Frequently Used (LFU) Algorithm

LFU Algorithm Principle

Keep a counter of references for each page. Replace page with smallest count. Use other algorithm (e.g., FIFO) to break ties.

Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Number of Frames: 3 Total Page Faults: 13

Ref.	Init	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
Frame 1	–	7	7	7	2	2	2	2	4	4	3	3	3	3	3	3	3	3	3	1	7	7	1
Frame 2	–	–	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Frame 3	–	–	–	1	1	1	3	3	3	2	2	2	2	2	2	2	1	2	2	2	2	2	2
Fault	–	✗	✗	✗	✗	–	✗	–	✗	✗	✗	–	–	–	–	–	✗	✗	–	✗	✗	–	✗

- **Rationale:** Actively used page should have **large reference count**
- **Problem:** Page used heavily initially keeps high count even if not used anymore

Most Frequently Used (MFU) Algorithm

MFU Algorithm Principle

Keep a counter of references for each page. Replace page with largest count. Use other algorithm (e.g., FIFO) to break ties.

Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Number of Frames: 3 Total Page Faults: 14

Ref.	Init	7	0	1	2	0	3	0	4	2	3	0	3	0	3	2	1	2	0	1	7	0	1
Frame 1	–	7	7	7	2	2	2	2	2	2	2	0	0	0	0	2	2	2	0	0	7	7	7
Frame 2	–	–	0	0	0	0	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1	0	1
Frame 3	–	–	–	1	1	1	1	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Fault	–	✗	✗	✗	✗	–	✗	✗	✗	–	–	✗	–	–	–	✗	✗	–	✗	–	✗	✗	✗

- **Rationale:** Page with **smallest count** was probably **just brought in** and yet to be used
- **Problem:** Not very intuitive; doesn't perform well in practice

Counting Algorithms - Summary

Takeaways on Counting Algorithms

Both LFU and MFU are not common in practice. They are expensive to implement and do not approximate OPT replacement well

Comparison of Page Faults:

Algorithm	Page Faults
FIFO	15
LRU	12
Optimal (OPT)	**9 (Best)**
LFU	13
MFU	14

Bounds on Page Faults

Mathematical Analysis:

Let R = *number of references*, F = *number of frames*, P = *number of distinct pages*

Minimum Page Faults

$$\text{Min Page Faults} = P \quad \text{when } F \geq P$$

- With enough frames for all distinct pages, only **first access** to each page causes a fault
- **Best case:** One fault per distinct page, all subsequent accesses are hits
- **Example:** $P = 5$ distinct pages, $F = 5$ frames $\rightarrow$ **5 faults minimum**

Maximum Page Faults

$$\text{Max Page Faults} = R \quad \text{when } F = 1$$

- With only **one frame**, almost every reference causes a page fault
- **Worst case:** Every access requires replacement (except consecutive duplicates)
- **Example:** $R = 20$ references, $F = 1$ frame $\rightarrow$ **up to 20 faults**

Key Takeaways

Key Takeaways

- **Goal:** Minimize page-fault rate using effective page-replacement algorithms
- **FIFO:** Simple but can suffer from Belady's Anomaly
  - While rare in practice, Belady's Anomaly demonstrates fundamental difference between **FIFO** (non-stack) and **LRU/Optimal** (stack algorithms)
  - **Stack property (inclusion property):** $M(m) \subseteq M(m+1)$ guarantees monotonic fault reduction with more frames, where $M(m)$ is the set of pages in memory with m frames
- **LRU:** Practical and effective, uses past access patterns
- **Optimal (OPT):** Theoretical benchmark using future knowledge, not easy to implement in practice
- **Counting Algorithms (LFU, MFU):** Less common, more complex, and generally less effective
- **Modern OS research explores ML-based predictive systems to approximate OPT performance by forecasting future page accesses.**

Outline

Exercise: Page Fault Bounds Analysis

Page Fault Bounds Analysis

Consider a system with m frames and a reference string of length n containing p unique pages ($p \leq n$).
Part A: Prove the following bounds on page faults for *any* page replacement algorithm:

- **Minimum:** At least $\min(p, n)$ page faults
- **Maximum:** At most n page faults

Part B: For the **Optimal** algorithm with m frames and p unique pages:

- Prove: Number of page faults $\geq p$ (compulsory misses)
- Explain when Optimal achieves exactly p faults
- Give an example reference string where Optimal with $m = 3$ frames and $p = 5$ unique pages achieves more than p faults

Part C (Tricky): Consider reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**

- How many page faults for Optimal with $m = 1$, $m = 3$, and $m = 4$ frames?
- **Challenge:** Can you find m where Optimal(m) $>$ Optimal($m - 1$)?
- Does this violate the stack property? Explain!

Solution Part A: General Bounds on Page Faults

Theorem 1: Any algorithm must have **at least** $\min(p, n)$ page faults.

Proof.

- **Case 1:** If $p \leq m$ (unique pages fit in memory)
  - First p unique pages cause compulsory misses $\rightarrow$ at least p faults
  - After that, all pages in memory $\rightarrow$ no more faults (if algorithm is smart)
  - Lower bound: p faults (since $p \leq n$, $\min(p, n) = p$)
- **Case 2:** If $p > m$ (more unique pages than frames)
  - Still need at least p faults to bring each unique page into memory once
  - Actually need $\geq p$ faults (likely more due to replacements)
  - Lower bound: $\min(p, n)$ (can't exceed string length)

Theorem 2: Any algorithm has **at most** n page faults. □

Proof.

- Each reference can cause *at most* one page fault
- Worst case: Every reference is a fault (e.g., $m = 1$ with all different pages)
- Upper bound: n faults

Solution Part A: Summary of General Bounds

Summary of General Bounds

Summary: For any algorithm with m frames and reference string of length n with p unique pages:

$$\boxed{\min(p, n) \leq \text{Page Faults} \leq n}$$

Solution Part B: Optimal Algorithm Bound

Theorem: For Optimal algorithm with m frames and p unique pages: Page faults $\geq p$
Stronger bound: When $p > m$, page faults $\geq p$ (cannot be less even for Optimal)

Proof.

- **Case 1:** If $p \leq m$ (all unique pages fit in memory)
  - First access to each of the p unique pages causes a *compulsory* (cold-start) fault
  - Total: exactly p faults for first accesses
  - After loading all p pages into m frames ($p \leq m$), no more faults occur
  - **Optimal achieves minimum:** p faults
- **Case 2:** If $p > m$ (more unique pages than frames)
  - Still need at least p compulsory faults (first access to each unique page)
  - **Cannot avoid:** Each of the p unique pages must be loaded at least once
  - Even Optimal cannot predict which pages to keep without causing at least p initial faults
  - In practice: Optimal achieves exactly p faults only in best-case access patterns
  - **Lower bound:** p faults (compulsory misses)
- **Conclusion:** For Optimal with m frames: Page faults $\geq p$ $\square$

Solution Part B: When Does Optimal Achieve Minimum?

Question: When does Optimal achieve exactly p faults?

Answer: When access pattern allows all p unique pages to be loaded once and remain useful

Best Case - Optimal Achieves p Faults

- **Scenario:** $p \leq m$ (all pages fit)
- String: **1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, ...** with $p = 5$, $m = 5$
- Result: Exactly 5 faults (one per unique page), then all hits
- **Optimal achieves theoretical minimum!**

Realistic Case - Optimal Has $> p$ Faults

- **Scenario:** $p > m$ and poor access pattern
- String: **1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...** with $p = 5$, $m = 3$
- First cycle: Faults at positions 1,2,3,4,5 (5 faults)
- Second cycle: Need 1 (in memory), 2 (in memory), 3 (fault: replace oldest), 4 (fault), 5 (fault)
- **Pattern repeats:** 2 faults per cycle after initial load
- Total for 12 references: $5 + 2 + 2 =$ **9 faults** $> p = 5$
- **Even Optimal cannot avoid capacity misses when $p > m$**

Solution Part C: The Tricky Challenge

Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 ($n = 12$, $p = 5$ unique pages)

Optimal Algorithm Analysis:

Reference	1	2	3	4	1	2	5	1	2	3	4	5	Faults
Optimal with $m = 1$ frame:													
Frame 1	1	2	3	4	1	2	5	1	2	3	4	5	12
Optimal with $m = 3$ frames:													
Frame 1	1	1	1	1	1	1	1	1	1	3	4	4	
Frame 2	–	2	2	2	2	2	2	2	2	2	2	2	7
Frame 3	–	–	3	4	4	4	5	5	5	5	5	5	
Optimal with $m = 4$ frames:													
Frame 1	1	1	1	1	1	1	1	1	1	1	1	1	
Frame 2	–	2	2	2	2	2	2	2	2	2	2	2	5
Frame 3	–	–	3	3	3	3	3	3	3	3	3	3	
Frame 4	–	–	–	4	4	4	5	5	5	5	5	5	

Solution Part C: The Mind-Bending Result

Answer to Challenge: Can Optimal(m) > Optimal($m-1$)? **NO!**

Theorem

Stack Property for Optimal: *For all m, Optimal(m) $\leq$ Optimal($m+1$)*
(More frames $\Rightarrow$ fewer or equal page faults)

Proof.

- **Stack property:** *Let $M(m)$ = set of pages in memory with m frames*
- *For stack algorithms: $M(m) \subseteq M(m+1)$ at every time step*
- **Optimal is a stack algorithm:**
  - *With $m+1$ frames, Optimal keeps the $m+1$ pages with nearest future use*
  - *With m frames, Optimal keeps the m pages with nearest future use*
  - *The m pages with nearest future use $\subseteq$ the $m+1$ pages with nearest future use*
  - *Therefore: $M(m) \subseteq M(m+1)$ at every time step*
- **Consequence:** *Every page in $M(m)$ is also in $M(m+1)$*
- *Therefore: Every page fault with $m+1$ frames $\Rightarrow$ would also fault with m frames*
- *But m frames may have additional faults for pages that fit in $m+1$ but not m*
- *Therefore: Optimal($m+1$) $\leq$ Optimal(m)* $\square$

Solution Part C: Why This is Tricky

- **Trap:** We might think "more frames could cause more faults" (like Belady's Anomaly)
- **Truth:** Belady's Anomaly **only** happens with *non-stack* algorithms (FIFO)
- **Optimal and LRU** are stack algorithms $\rightarrow$ *immune* to anomaly
- Our example shows: $12 \geq 7 \geq 5$ (monotonic decrease as frames increase) ✓
- **Stack property ensures:** Faults$(m+1) \leq$ Faults(m) for all m

Key Insight on Optimal's Stack Property

Optimal's *clairvoyance* (future knowledge) + **inclusion property** $M(m) \subseteq M(m+1)$ $\rightarrow$ **guaranteed monotonic improvement** with more frames.
FIFO **violates inclusion property** $\rightarrow$ pages in memory with m frames may differ completely from those with $m+1$ frames $\rightarrow$ can have anomalies!

End of Chapter 8