**CHAPTER 10**

Virtual memory: Virtual memory is a computer memory management technique that allows a computer to use more memory than physically available by temporarily transferring data from RAM to hard disk storage.

Major advantage:

1. Programs can be larger than physical memory.
2. It provides an efficient mechanism for process creation.

Virtual memory and paging are closely related concepts, but they are not exactly the same.

Virtual memory refers to the technique of using disk storage to simulate additional RAM, while

Paging specifically refers to the process of transferring memory pages between RAM and disk storage. Paging is one aspect of virtual memory.

- **Demand paging** brings in a page only when it's referenced, leading to possible page faults and CPU interruptions while waiting for the required page to load.
- **Pre-paging** tries to anticipate which pages will be needed and loads them ahead of time, aiming to optimize CPU and I/O overlap. But incorrect guesses can lead to unnecessary page faults and removal of useful pages.

**PERFORMANCE OF DEMAND PAGING:**

- **Temporal locality:** If a process accesses an item, it's likely to use it again soon.
- **Spatial locality:** If a process accesses an item, adjacent items are likely to be referenced too.
- The formula for effective access time accounts for page faults, which can dramatically slow down performance.

Demand Paging and Paged Virtual Memory

OS illusion: The system treats disk storage as an extension of main memory, similar to how memory acts as a larger, slower cache.

**Virtual memory advantages**:

- Programs can exceed physical memory limits.
- Processes can execute without fully being in memory.
- More processes can run concurrently, improving system throughput.

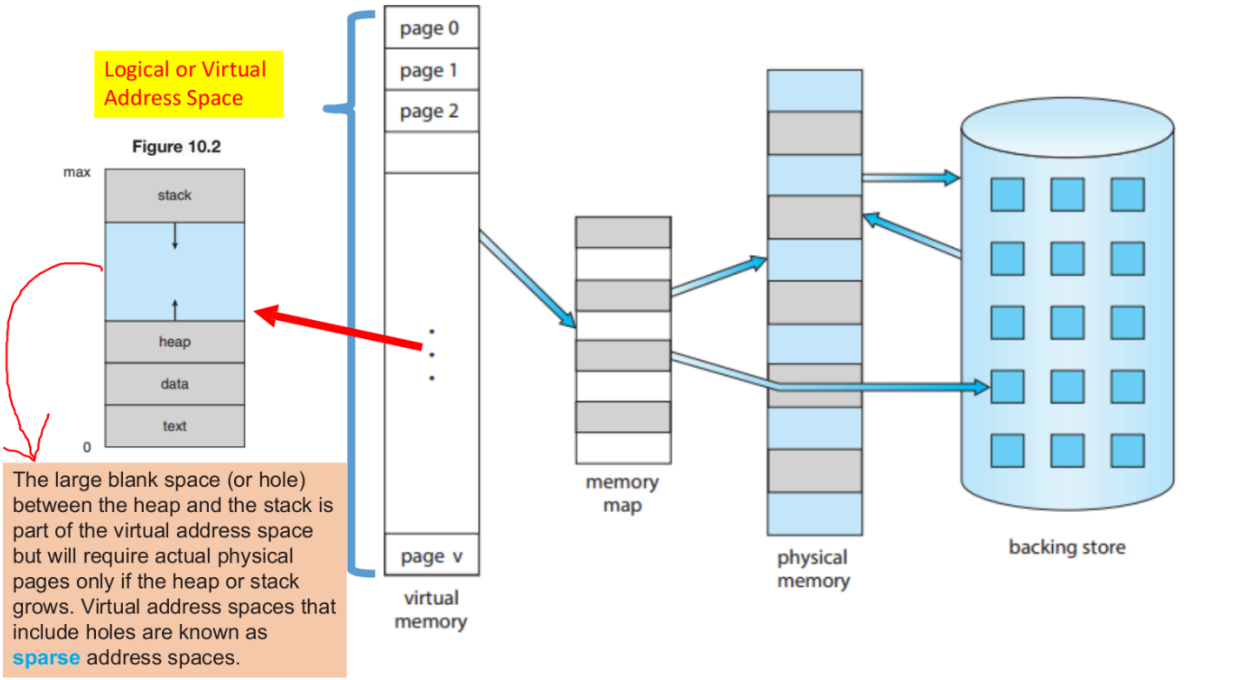


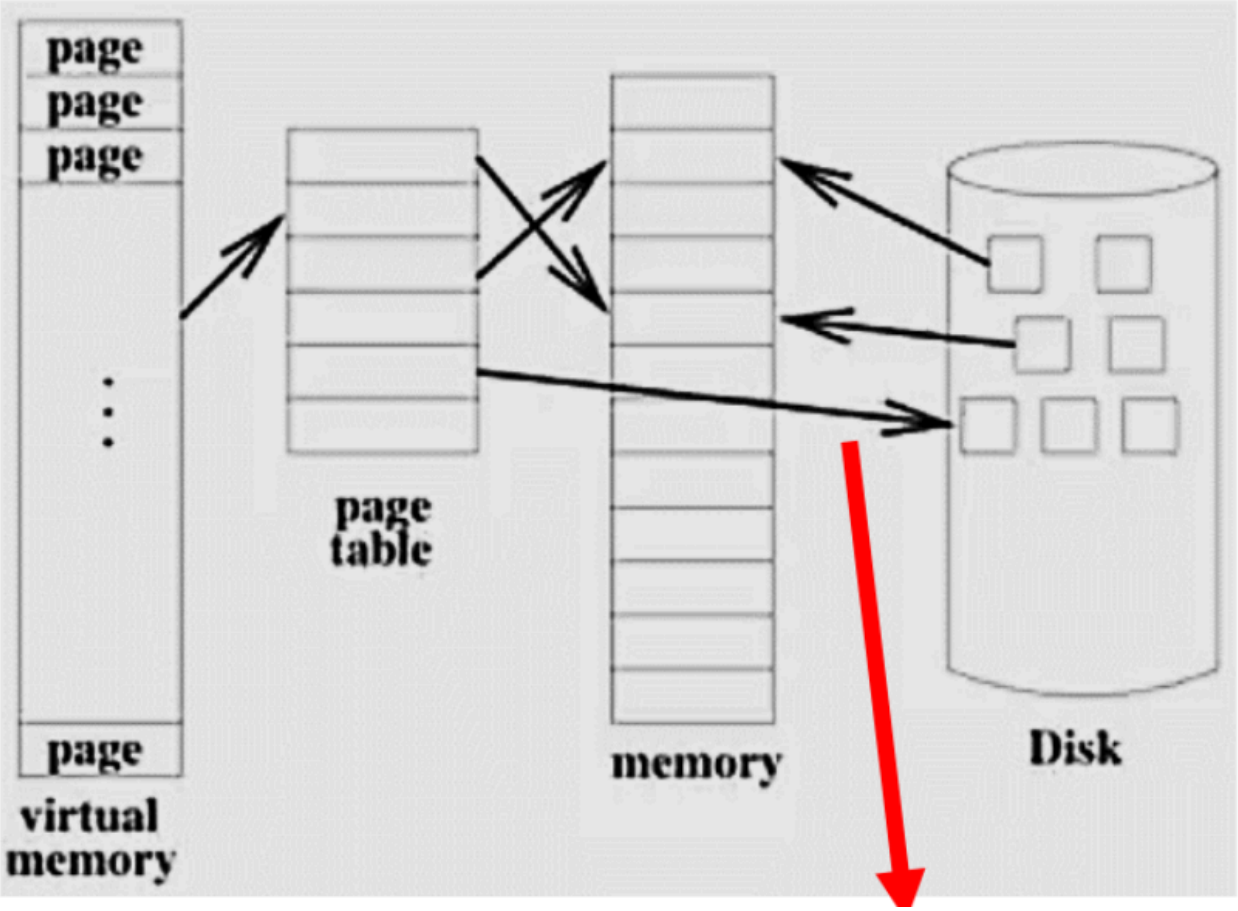

**Figure 10.1** Diagram showing virtual memory that is larger than physical memory.

Key insights:

- **Sparse Address Spaces:** The gap between heap and stack represents memory that doesn't need to be allocated until required. This prevents unnecessary physical memory consumption.
- **Paging Mechanism:** Virtual addresses map to physical memory, with some pages stored in **backing store** (secondary storage), ensuring efficient memory usage.
- **Improving System Throughput:** By keeping only essential pages in physical memory, virtual memory enhances multitasking and responsiveness.

**Page Fault:**

Page faults are expensive because they involve a time-consuming process of loading data from disk into physical memory. This requires disk I/O operations, which are slow compared to memory access times. Additionally, page faults can cause the operating system to stall the execution of a program until the required page is loaded, resulting in performance degradation.




Virtual memory allows files and memory to be shared by two or more processes through page sharing (Section 9.3.4). This leads to the following benefits:

Key benefits of **shared pages via virtual memory**:

1. **Shared system libraries**: Standard libraries (e.g., C library) are mapped into the virtual address space of multiple processes, reducing memory duplication.
2. **Shared memory regions**: Two or more processes can communicate through shared memory, creating an efficient inter-process communication (IPC) mechanism.

3. **Optimized process creation**: Pages can be shared during process creation with the **fork()** system call, significantly improving system performance.

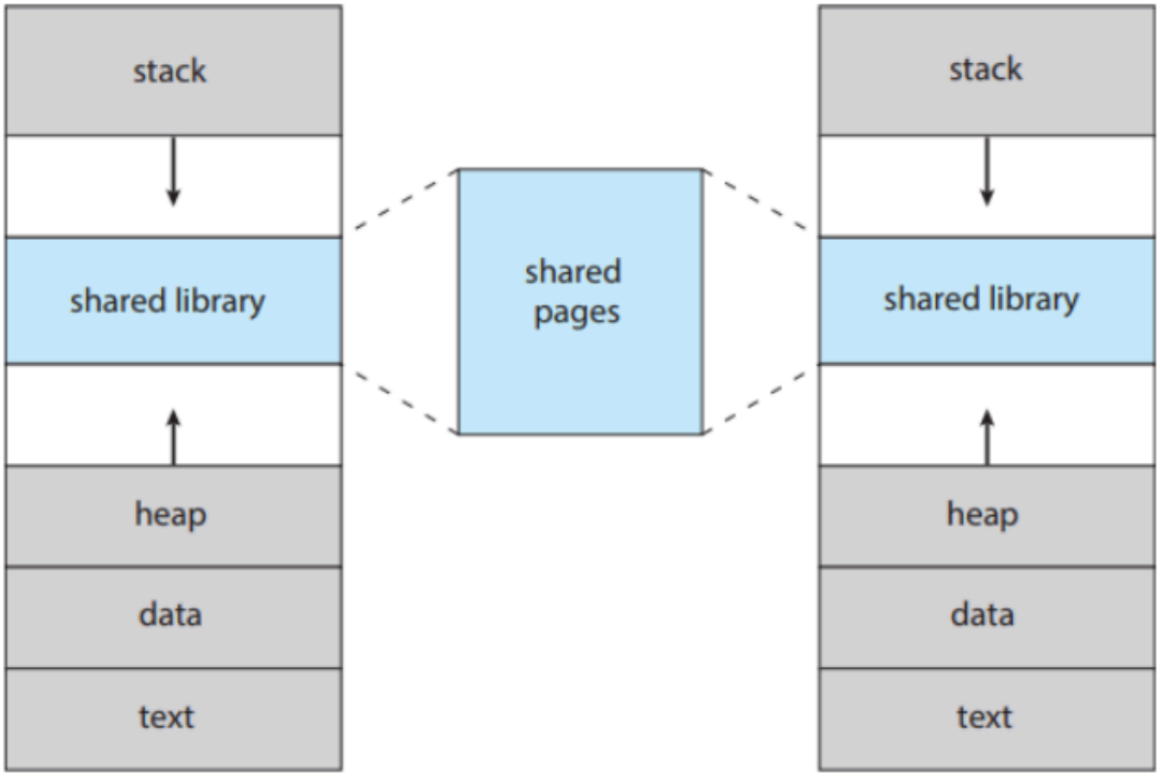



**Figure 10.3** Shared library using virtual memory.

**Demand Paging:**

Key benefits of demand paging:

- Reduced memory footprint: Only required pages occupy RAM, leaving space for more processes.
- Improved execution time: Less initial loading prevents delays caused by unnecessary pages.
- Better multitasking: More processes can run concurrently without memory overload

**Handling page requests in a demand-paging system:**

1. Reference Generation: The CPU generates a memory reference for a required page.
2. Page Table Lookup: The system checks the page table entry to determine if the requested page is in physical memory.
3. Valid–Invalid Bit Check: The valid–invalid bit is examined:

- If valid, the page is already in memory, and execution continues.

- If invalid, the page is either in secondary storage or not part of the process's address space.

4. Page Fault Handling: If the page is not in memory, a page fault occurs, triggering an exception.
5. OS Intervention: The operating system takes control to handle the fault.
6. Disk Access for Page Load: The required page is fetched from secondary storage (HDD, SSD, or NVM).
7. Frame Allocation: The system selects an empty frame or applies page replacement if memory is full.
8. Page Table Update: The page table entry is updated with the new frame location and marked valid.
9. Process Resume: Execution resumes at the instruction that caused the page fault.

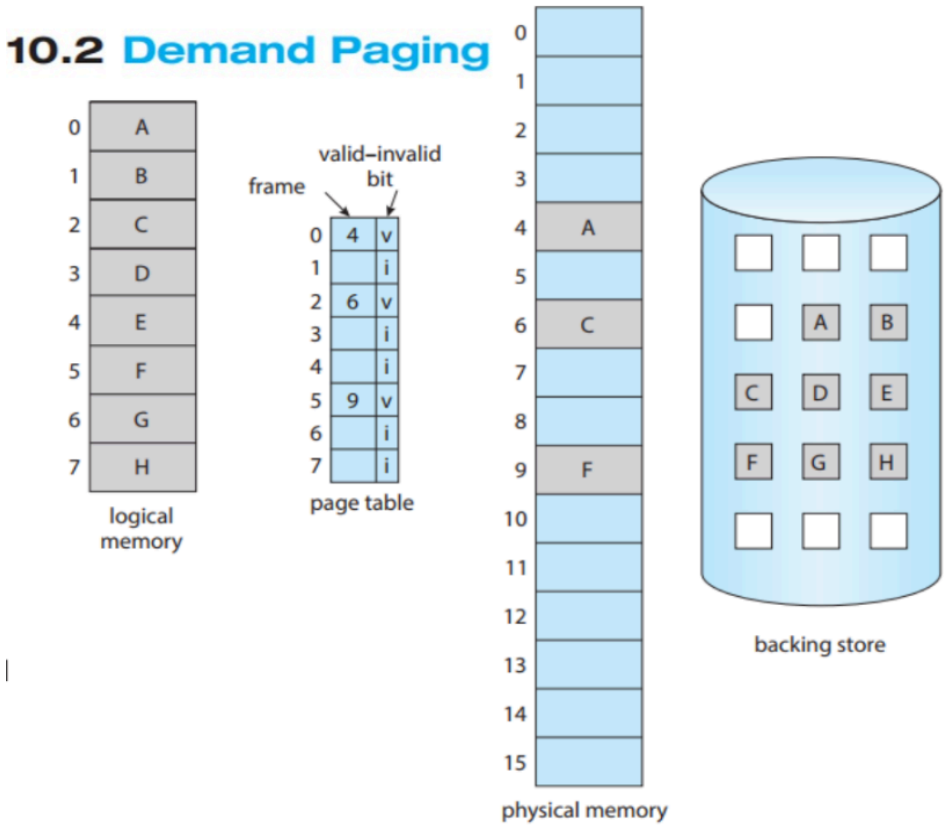
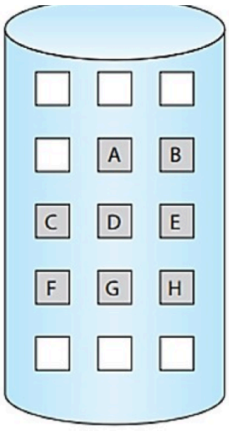



**Figure 10.4** Page table when some pages are not in main memory.

**Swap Space:**

1. Page Removal Logic:
    - If the evicted page contains executable code, it can be discarded safely since it can be reloaded from disk when needed.
    - If the page contains data, it must be saved in swap space to ensure the process can retrieve it later.

2. Where a Page Might Reside:
    - Pages can exist in the file system, physical memory, or swap space, depending on system demands.
3. Sophisticated Page Table Design:
    - The page table must track where each page resides—whether in RAM, swap space, or storage—so the system can retrieve it efficiently.



backing store

**Procedure to handle page fault:**

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

A crucial requirement for demand paging is the ability to restart instructions.

1. State Preservation: When a page fault occurs, the system saves registers, condition codes, and the instruction counter to allow seamless recovery.
2. Restarting Faulted Instructions:
    - If the fault occurs during instruction fetch, the system simply re-fetches the instruction.

- If the fault happens while fetching an operand, the system must reprocess the instruction before continuing execution.

3. Memory Reference Consideration: Since a page fault can happen at any memory reference, handling these faults efficiently is crucial for system performance.

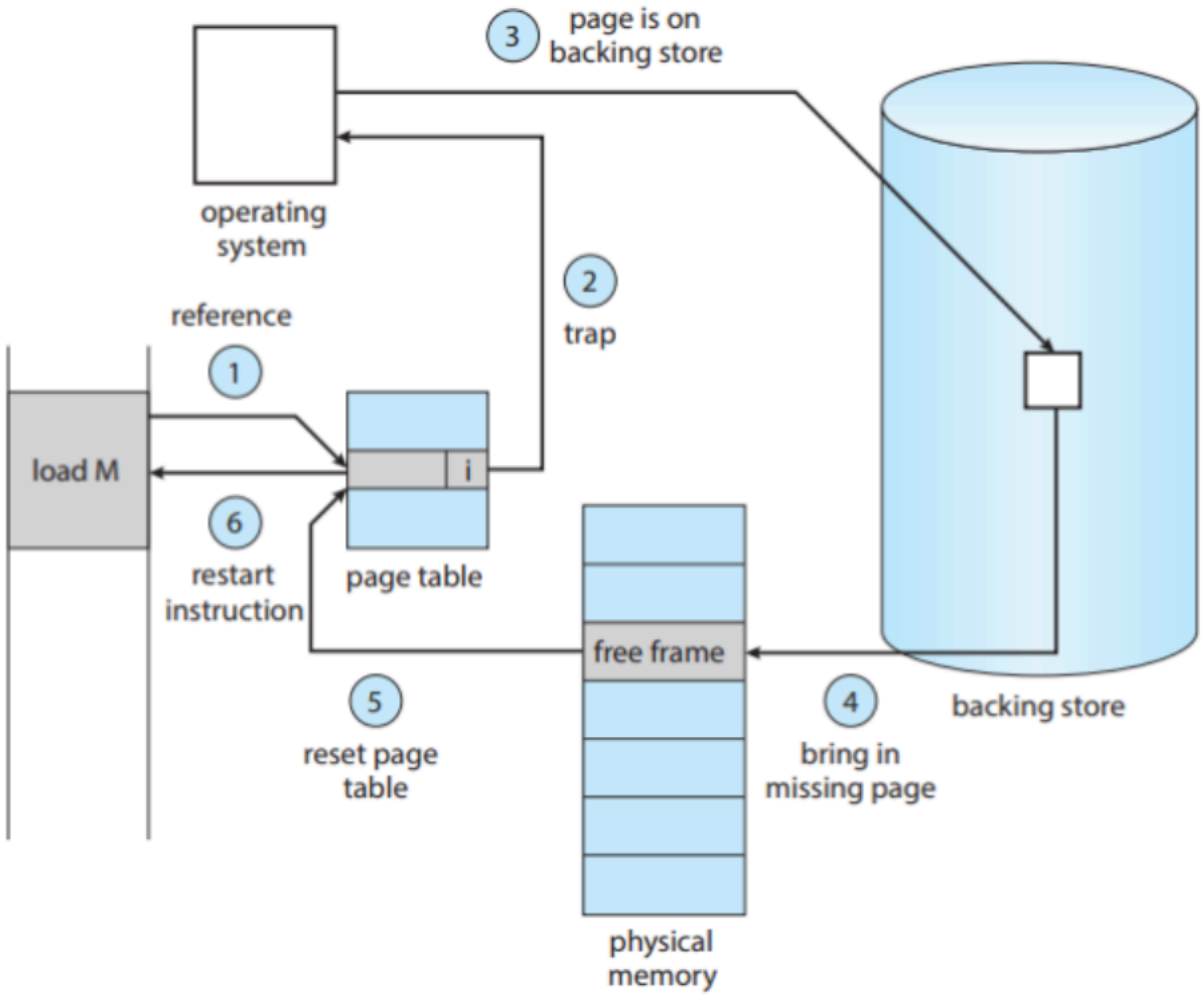



**Figure 10.5** Steps in handling a page fault.

Updating the TLB:
Hardware vs. Software Handling of TLB Misses:

- Some systems use hardware to load the TLB on a miss.
- If TLB hit rates are high, software is used instead to update the TLB dynamically.

Handling TLB Hits:

- If the valid bit is set, the frame number is used to access memory without needing page table lookup.

Handling TLB Misses:

- The OS traps the TLB miss and checks if the page is in memory.
- If page is in memory, the OS selects a TLB entry to replace and updates it with the new mapping.
- If page is not in memory, the OS does the following:
    - Invalidates the TLB entry.
    - Performs page fault handling (loading the page from secondary storage).
    - Updates the TLB entry.
    - Restarts the faulting instruction to resume execution.

**Pure demand paging** ensures that no page is loaded into memory until it is explicitly needed. This approach minimizes initial memory allocation, reducing overhead until execution demands it.

## Step-by-Step Process:

1. Process starts with no pages in memory.
2. Instruction pointer targets the first instruction, which is on a non-resident page.
3. Page fault occurs, triggering the OS to load the missing page from secondary storage.
4. Process execution resumes, faulting again whenever a new page is needed.
5. Once all required pages are loaded, the program continues running without further page faults.

This approach leverages locality of reference—meaning that processes frequently access the same set of pages for extended periods. As a result, demand paging performs efficiently because the OS only fetches pages actively needed, avoiding unnecessary memory usage.

**Breakdown of Execution Steps:**

1. **Fetch and decode the instruction (ADD).**
2. **Fetch operand A.**
3. **Fetch operand B.**
4. **Perform addition: A + B.**
5. **Store the result in C.**
6. **If a page fault occurs while storing C, the instruction must restart from Step 1.**
    - This means **re-fetching and decoding the instruction** before resuming computation.

**Why Restarting is Necessary:**

- When a page fault happens, the system must load the required page from disk, update the page table, and then restart the instruction to ensure the correct execution flow.
- This is crucial in instruction fetch faults and cases where operand storage triggers a page fault.

**Transparent Page Faults:**

How does the OS transparently restart a faulting instruction?

Need hardware support to save

1. the faulting instruction,
2. the CPU state.

What about instructions with side-effects? (CISC)

- *move -(r10), a* : decrements register 10, then moves r10 to address *a*

Solution: unwind side effects

- Block transfer instructions where the source and destination overlap can't be undone.


- Solution: check that all pages between the starting and ending addresses of the source and destination are in memory before starting the block transfer

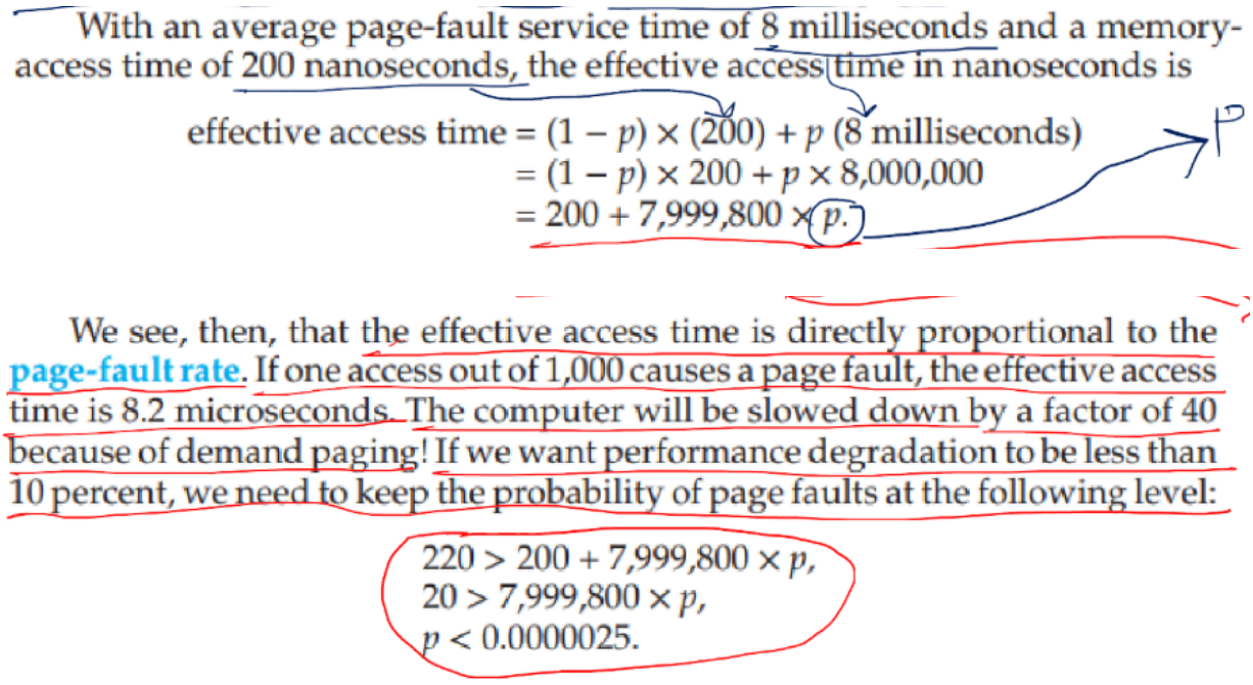
**PERFORMANCE OF DEMAND PAGING:**

- Effective access time formula:

effective access time = (1-p)*ma* + *p**page fault time

- If p is low, page faults rarely occur, improving performance.
- If p is high, frequent faults cause delays due to disk I/O overhead.

- Annotations clarify memory access paths:
    - (1 - p) probability: Page is already in memory, accessed via:
        - TLB hit (fast access).
        - TLB miss (requires page table lookup).
    - p probability: Page fault occurs, requiring:
        - TLB miss.
        - Page retrieval from swap space or disk, which is slow.



With an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, the effective access time in nanoseconds is

$$\begin{aligned}\text{effective access time} &= (1 - p) \times (200) + p\ (8 \text{ milliseconds})\\ &= (1 - p) \times 200 + p \times 8{,}000{,}000\\ &= 200 + 7{,}999{,}800 \times p.\end{aligned}$$

We see, then, that the effective access time is directly proportional to the page-fault rate. If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. The computer will be slowed down by a factor of 40 because of demand paging! If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:

$$\begin{aligned}220 &> 200 + 7{,}999{,}800 \times p,\\ 20 &> 7{,}999{,}800 \times p,\\ p &< 0.0000025.\end{aligned}$$

**Copy On Write:**

- **Initial Sharing**: When a process forks, the parent and child initially share the same physical memory pages.
- **Copy-on-Write Pages**: Pages that might be modified (like data or stack pages) are marked as "copy-on-write." This means they are shared until a write occurs.
- **On Write**: When either the parent or child writes to a COW-marked page, a private copy of that page is created, and the writing process gets its own copy.
- **Executable Pages**: Since code pages are read-only, they don't need to be marked COW—they can be safely shared.
- **Efficiency**: This approach avoids unnecessary copying and improves performance, especially in systems that use **fork()** to create new processes.
- **Used in Major OSes**: Linux, Windows, and macOS all implement this technique.

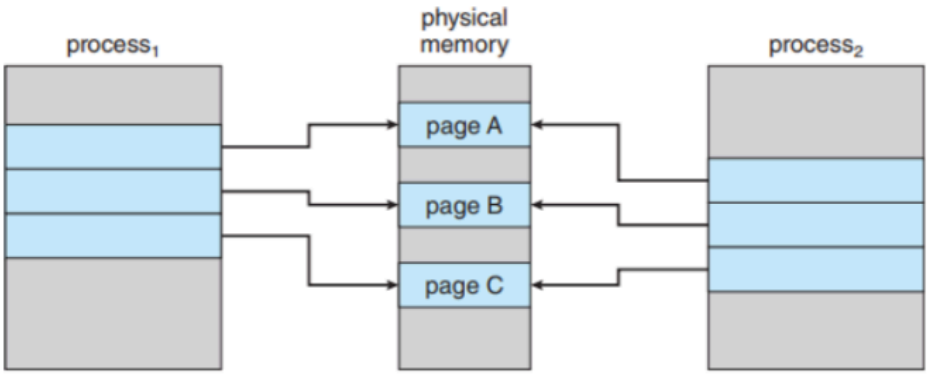



**Figure 10.7** Before process 1 modifies page C.

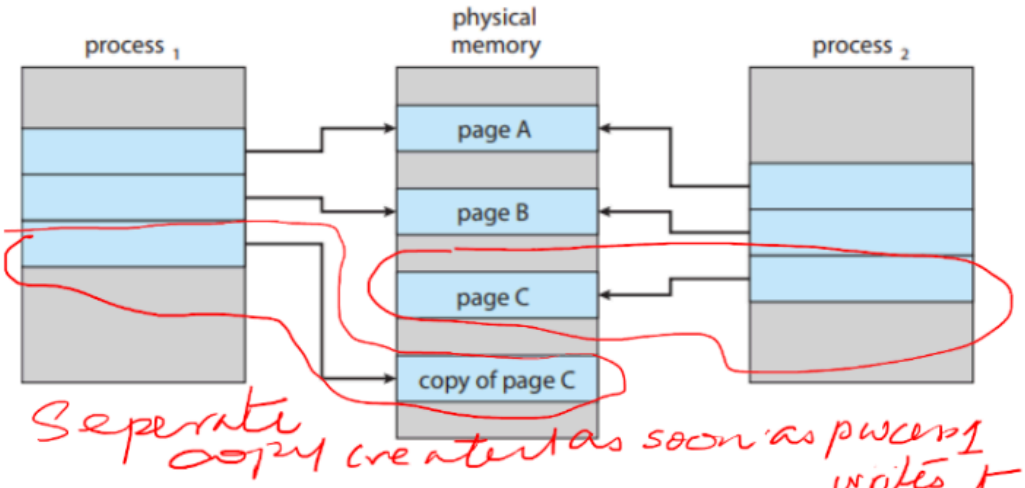



**Figure 10.8** After process 1 modifies page C.

## Frame Allocation vs Page Replacement

The frame allocation problem involves deciding how many and which physical memory frames to assign to processes or pages in a virtual memory system to optimize memory use and reduce page faults. In contrast, the page replacement problem arises when a page fault occurs, requiring the system to choose which page to evict from memory to make space for a new one. While frame allocation focuses on the initial distribution of memory frames, page replacement deals with managing memory during execution when available frames are full.

## BASIC PAGE REPLACEMENT:

- Increasing the degree of multiprogramming can lead to over-allocation of memory.
- Example: 6 processes, each needing 10 pages but using only 5 → 30 frames used, 10 frames free.

- Higher CPU utilization and throughput result from this setup.
- Problem arises if all processes suddenly need all 10 pages → demand for 60 frames, but only 40 are available.
- Similar issue can occur if I/O buffer demands suddenly increase.
- When no free frames are available, the system must use page replacement.
- A frame not in active use is freed by:
    - Writing its contents to swap space.
    - Updating the page table to mark it as not in memory.
- The freed frame is then used to load the new required page.
- Standard swapping (moving entire processes in/out of memory) is mostly obsolete.
- Modern OSes use demand paging combined with page replacement to manage memory efficiently.

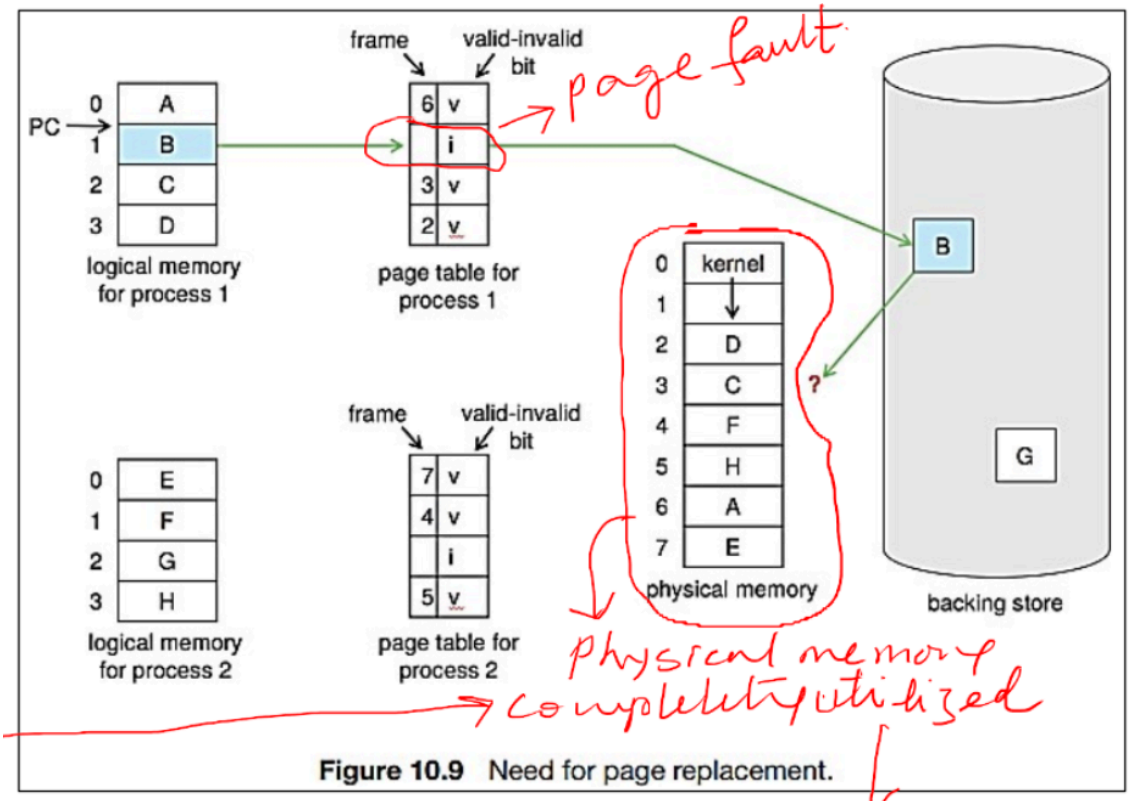



**Figure 10.9** Need for page replacement.

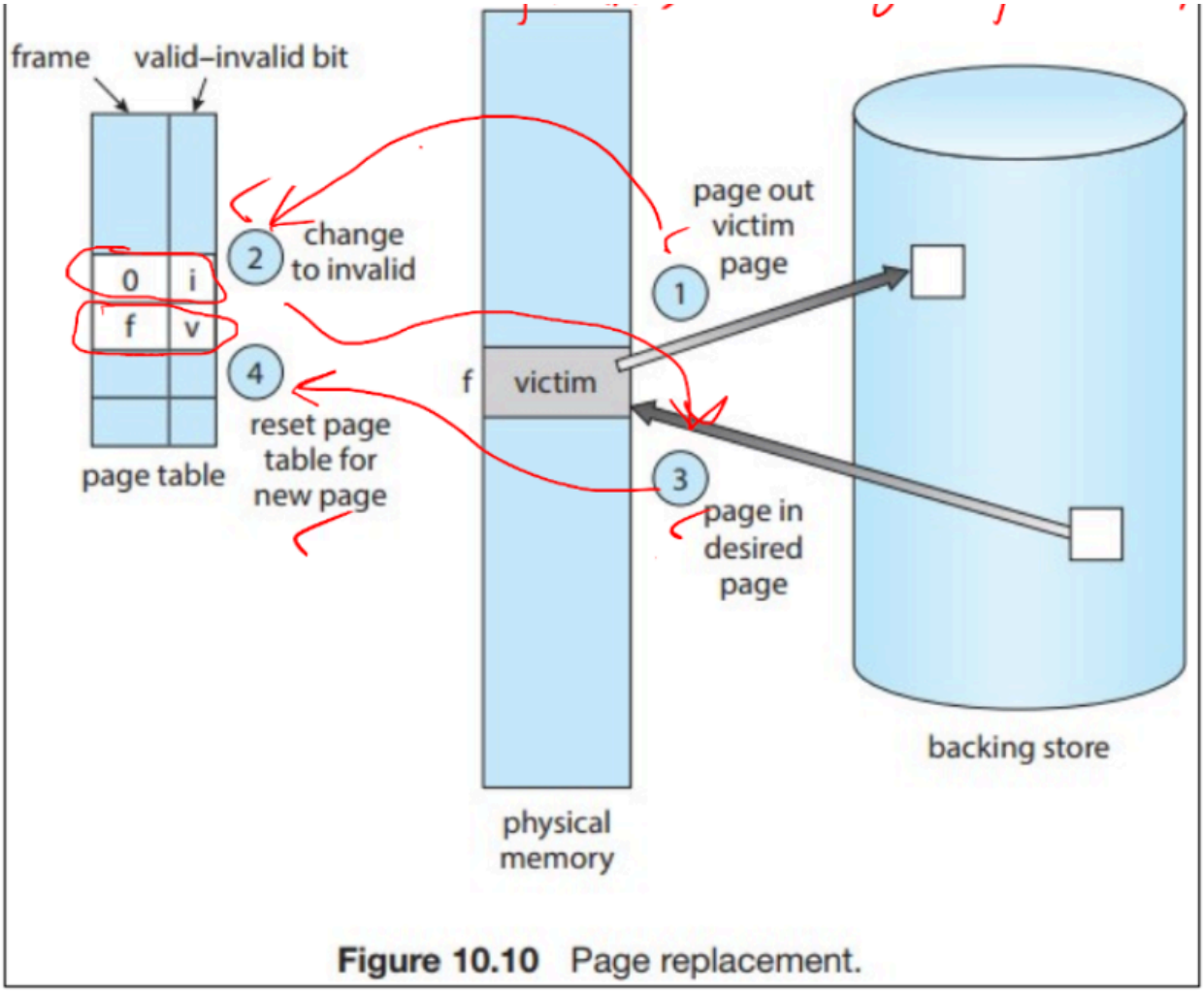



**Figure 10.10** Page replacement.

Three Key Components of Page-Fault Handling

1. Service the page-fault interrupt.
2. Read in the page from secondary storage.
3. Restart the process from the faulting instruction.

**PAGE FAULT HANDLING ROUTINE:**

1. Page Fault Occurs

- The hardware detects a memory reference to a page not in physical memory.
- The CPU traps to the OS, generating a page fault interrupt.

2. Operating System Handles the Interrupt

1. Trap to the OS via an interrupt.

2. Save the process state and CPU registers.
3. OS confirms that the interrupt is due to a page fault.
4. Check the validity of the page reference and locate the page in secondary storage (disk/SSD).

3. Page Replacement (If No Free Frame Is Available)

If a free frame is not available, the OS performs page replacement:

1. Use a page replacement algorithm (like LRU, FIFO) to select a victim page.
2. If the victim page is dirty (modified), write it back to secondary storage.
3. If the victim page is clean (modify bit = 0), no write-back is needed. New page is directly brought into the physical memory from the backing store.
4. Update the page table and frame table to reflect the eviction.
5. Allocate the freed frame to the new page.

4. Read the Required Page from Disk

1. Issue a disk read to bring the required page into memory (into a free or newly freed frame).
    - a. The process waits in the I/O queue.
    - b. Wait for device latency.
    - c. Transfer the page data to memory.
2. During this time, the CPU is allocated to another process (context switch).

5. After I/O Completion

1. Interrupt is generated when I/O completes.
2. OS saves the state of the running process (if different).
3. Confirm the interrupt was for the completed page read.
4. Update the page table to indicate the page is now in memory.
5. Mark the page as valid and reset the dirty (or modify) bit.

6. Resume the Original Process

1. Wait until the original process gets CPU time again.
2. Restore its saved state and registers.
3. Restart the instruction that caused the page fault — not from the beginning of the process, but exactly where it left off.

How does the number of available frames relates to page faults?

🔁 Reference String: **1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1**
This is the sequence of memory pages that the process tries to access.

Key Observations:

🔹 When only 1 frame is available

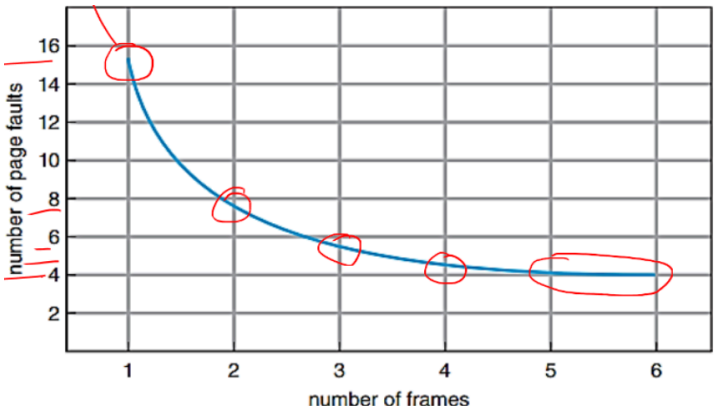
- Each new page replaces the old one.
- Because only 1 page fits at a time, every access causes a page fault and replacement.
- So, with 11 references → 11 page faults.
- This is highlighted on the graph at frame count = 1, where page faults are highest.

🔹 When 3 or more frames are available

- Only 3 unique pages are accessed: 1, 4, 6.
- So, with 3 frames, each unique page fits once → only 3 page faults.
- Repeated accesses to these pages do not cause further faults.

📉 General Trend – Graph in Figure 10.11

- X-axis: Number of frames
- Y-axis: Number of page faults
- As the number of frames increases, the number of page faults decreases.
- Eventually, the graph flattens—adding more frames doesn't significantly reduce page faults once all needed pages can fit.
- More frames = better performance up to a point.
- Too few frames → frequent page replacements (high overhead).
- Enough frames → fewer replacements, better performance.
- This is the working set concept: if you can fit your active pages in memory, faults drop sharply.



**PAGE REPLACEMENT ALGORITHMS:**

1. FIFO (First-In, First-Out):

- Strategy: Removes the oldest page in memory (the one loaded earliest).
- Advantage: Simple to implement using a queue.
- Disadvantage: Can result in Belady's Anomaly—more frames may lead to more page faults.

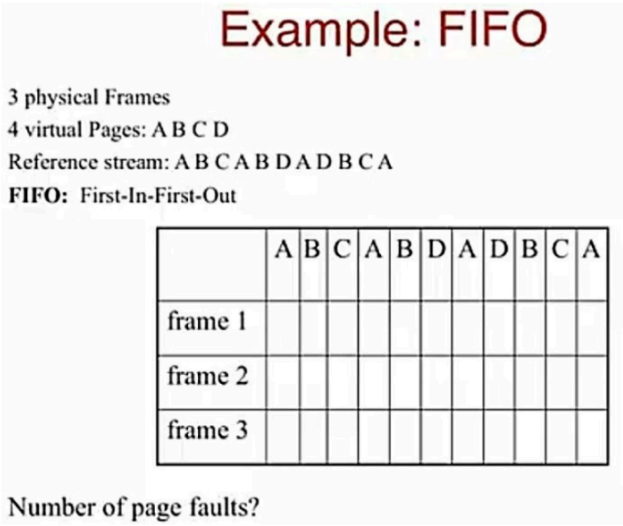
2. MIN (Optimal Page Replacement):

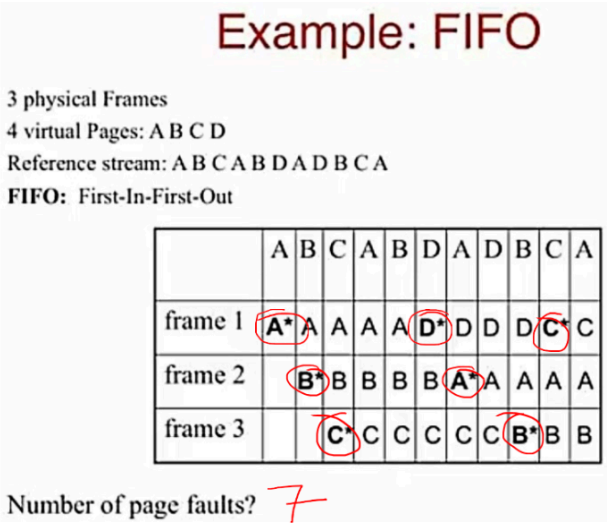
- Strategy: Replaces the page that will not be used for the longest time in the future.
- Advantage: Gives the lowest possible page fault rate.
- Disadvantage: Not implementable in practice (requires future knowledge); used for theoretical comparison.

3. LRU (Least Recently Used):

- Strategy: Replaces the page that was least recently used.
- Advantage: Good approximation of MIN; works well in many real-world scenarios.
- Disadvantage: More complex to implement, may require timestamps or a stack.

MIN is not implementable, only FIFO and LRU are implementable.



# Example: MIN

**MIN:** Look into the future and throw out the page that will be accessed farthest in the future.

| | A | B | C | A | B | D | A | D | B | C | A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | |

Number of page faults?

# Example: MIN

**MIN:** Look into the future and throw out the page that will be accessed farthest in the future.

| | A | B | C | A | B | D | A | D | B | C | A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | **A*** | A | A | A | A | A | A | A | A | A | C |
| frame 2 | | **B*** | B | B | B | B | B | B | B | **C*** | B |
| frame 3 | | | **C*** | C | C | **D*** | D | D | D | D | D |

Number of page faults? **5**

# Example: LRU

- **LRU:** Least Recently Used. Throw out the page that has not been used in the longest time.

| | A | B | C | A | B | D | A | D | B | C | A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | |

Number of page faults?

# Example: LRU

- **LRU:** Least Recently Used. Throw out the page that has not been used in the longest time.

| | A | B | C | A | B | D | A | D | B | C | A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | **A*** | A | A | A | A | A | A | A | A | **C*** | C |
| frame 2 | | **B*** | B | B | B | B | B | B | B | B | B |
| frame 3 | | | **C*** | C | C | **D*** | D | D | D | D | **A*** |

Number of page faults? **6**

- When will LRU perform badly?

| | A | B | C | D | A | B | C | D | A | B | C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | |

- Number of page faults?

- When will LRU perform badly?

| | A | B | C | D | A | B | C | D | A | B | C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | **A*** | A | A | **D*** | D | D | **C*** | C | C | **B*** | B |
| frame 2 | | **B*** | B | B | **A*** | A | A | **D*** | D | D | **C*** |
| frame 3 | | | **C*** | C | C | **B*** | B | B | **A*** | A | A |

- Number of page faults? **11**

## Adding Memory: FIFO

Does adding memory always reduce the number of page faults?

FIFO:

| | A | B | C | D | A | B | E | A | B | C | D | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | | |
| frame 1 | | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | | |
| frame 4 | | | | | | | | | | | | |

## Adding Memory: FIFO

Does adding memory always reduce the number of page faults?

FIFO:

| | A | B | C | D | A | B | E | A | B | C | D | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | A* | A | A | D* | D | D | E* | E | E | E | E | E |
| frame 2 | | B* | B | B | A* | A | A | A | A | C* | C | C |
| frame 3 | | | C* | C | C | B* | B | B | B | B | D* | D |
| frame 1 | A* | A | A | A | A | A | E* | E | E | E | D* | D |
| frame 2 | | B* | B | B | B | B | B | A* | A | A | A | E* |
| frame 3 | | | C* | C | C | C | C | C | B* | B | B | B |
| frame 4 | | | | D* | D | D | D | D | D | C* | C | C |

• **Belady's Anomaly:** Adding page frames may actually cause **more** page faults with certain types of page replacement algorithms (such as FIFO).

**Belady's anomaly**: for some page-replacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true.

## Adding Memory: LRU

**LRU:**

| | A | B | C | D | A | B | E | A | B | C | D | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | A* | A | A | D* | D | D | E* | E | E | C* | C | C |
| frame 2 | | B* | B | B | A* | A | A | A | A | A | D* | D |
| frame 3 | | | C* | C | C | B* | B | B | B | B | B | B |
| frame 1 | A* | A | A | A | A | A | A | A | A | A | A | E* |
| frame 2 | | B* | B | B | B | B | B | B | B | B | B | B |
| frame 3 | | | C* | C | C | C | E* | E | E | E | D* | D |
| frame 4 | | | | D* | D | D | D | D | D | C* | C | C |

•With LRU, increasing the number of frames always decreases the number of page faults. Why?

When you use **LRU (Least Recently Used)** to manage memory:

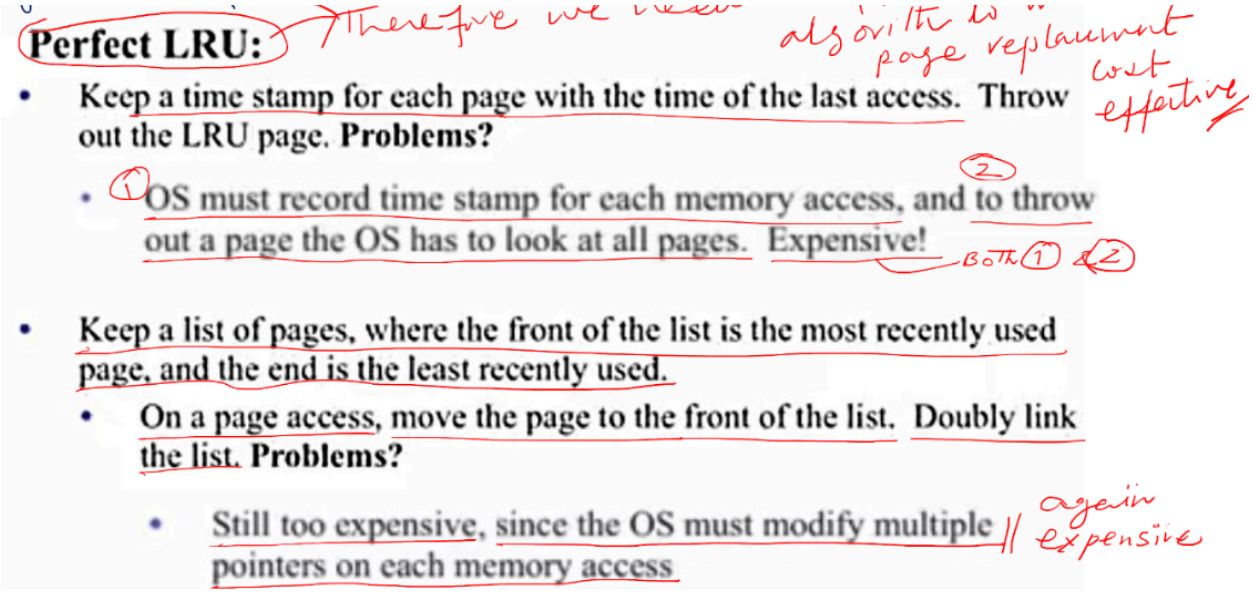
1. **More Frames = More Space**: Adding frames (memory slots) gives your system more room to keep pages (data chunks) it has used **recently**.
2. **No "Kicking Out" Old Pages**: LRU ensures pages that were safe in fewer frames *stay safe* in more frames. You only add new pages, never remove old ones unnecessarily.

3. **Fewer Swaps**: With extra space, the system rarely needs to remove a page to make room, so fewer **page faults** (slow reloads from disk) happen.
4. **No Surprises**: Unlike some algorithms, LRU *guarantees* that more memory never makes things worse.

**Example**:

- With 3 frames: You might swap pages in/out often.
- With 4 frames: You keep all 3 old pages *plus* a new one. Fewer swaps!

**IMPLEMENTING LRU**




Approach 1: Doubly linked list and stack

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;
// Node structure for the doubly linked list
struct Node {
    int page;
    Node* prev;
    Node* next;
    Node(int p) : page(p), prev(nullptr), next(nullptr) {}
};
```

```cpp
class LRUCache {
private:
    int capacity;
    Node* head; // MRU (top of stack)
    Node* tail; // LRU (bottom of stack)
    unordered_map<int, Node*> pageMap;
    // Add node to the head (MRU position)
    void addToHead(Node* node) {
        node->prev = nullptr;
        node->next = head;
        if (head) head->prev = node;
        else tail = node; // First node in the list
        head = node;
    }
```

```cpp
// Remove a node from the list
void removeNode(Node* node) {
    if (node->prev) node->prev->next = node->next;
    else head = node->next; // Node is the current head

    if (node->next) node->next->prev = node->prev;
    else tail = node->prev; // Node is the current tail
}
```

```cpp
public:
    LRUCache(int cap) : capacity(cap), head(nullptr), tail(nullptr) {}
    ~LRUCache() {
        // Cleanup: Delete all nodes to prevent memory leaks
        Node* current = head;
        while (current) {
            Node* temp = current;
            current = current->next;
            delete temp;
        }
    }
```

```cpp
// Access a page and update the stack
void accessPage(int page) {
    if (pageMap.find(page) != pageMap.end()) {
        // Page exists: Move it to the head
        Node* node = pageMap[page];
        if (node != head) {
            removeNode(node);
            addToHead(node);
        }
    } else {
        // Page fault: Add new page
        Node* newNode = new Node(page);
        if (pageMap.size() >= capacity) {
            // Evict LRU page (tail)
            pageMap.erase(tail->page);
            removeNode(tail);
            delete tail;
        }
        addToHead(newNode);
        pageMap[page] = newNode;
    }
}
```

```cpp
    // Print the current stack (MRU to LRU)
    void printStack() {
        Node* current = head;
        cout << "Stack (MRU -> LRU): ";
        while (current) {
            cout << current->page << " ";
            current = current->next;
        }
        cout << endl;
    }
};
```

```cpp
int main() {
    LRUCache cache(3); // 3 frames
    int referenceString[] = {4,7,0,7,1,0,1,2,1,2,7,1,2};

    cout<<"Simulating LRU with reference string: ";
    for(int page : referenceString) cout<<page<< " ";
    cout<<"\n\n";

    for (int page : referenceString) {
        cout<<"Accessing page "<<page<<":"<<endl;
        cache.accessPage(page);
        cache.printStack();
    }
    return 0;
}
```

Approach 2- Counter Approach

1. **Timestamp Tracking**:
    - Each page-table entry has a **time-of-use field** that records the last access time using a global **logical clock/counter**.
    - The clock increments with every memory reference (e.g., read/write operations).
2. **Page Replacement**:
    - On a page fault, the system scans all pages to find the one with the **smallest timestamp** (oldest access time) and replaces it.
    - But, it's expensive.

Approximation of LRU —-> using reference bits.

**1. Concept Overview**

The **Least Recently Used (LRU)** page replacement algorithm requires tracking the exact order of page accesses to evict the least recently used page. However, maintaining precise timestamps or linked lists for every page is computationally expensive. **Approximations of LRU** simplify

this by using hardware-assisted reference bits to estimate page usage history, balancing efficiency and accuracy.

## 2. Hardware Requirements

- **Reference Bits**: Each page in memory has a set of bits (e.g., 1 or 8 bits) to track recent usage.
- **Bit Operations**:
    - On **page access**, the hardware sets the highest-order bit to `1`.
    - At **regular intervals** (e.g., during clock interrupts), all bits are shifted right, inserting `0` in the highest-order bit.

## 3. How It Works

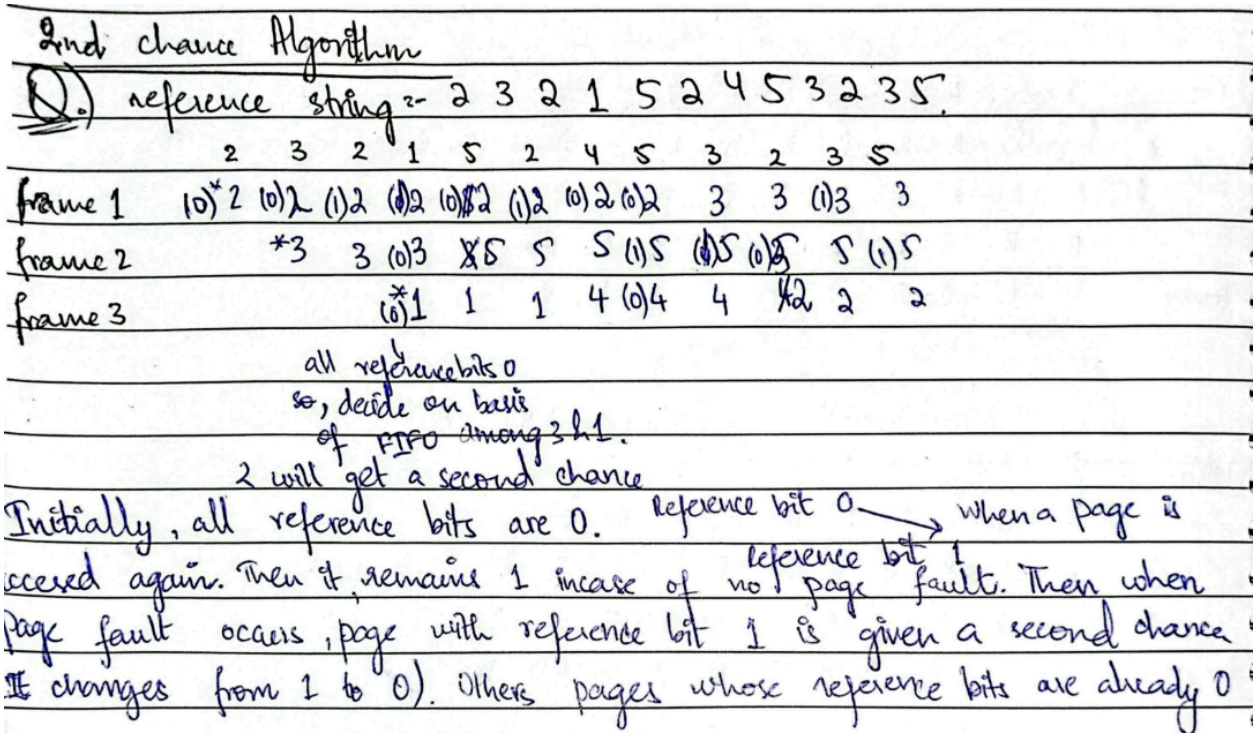
### Example with 8 Reference Bits

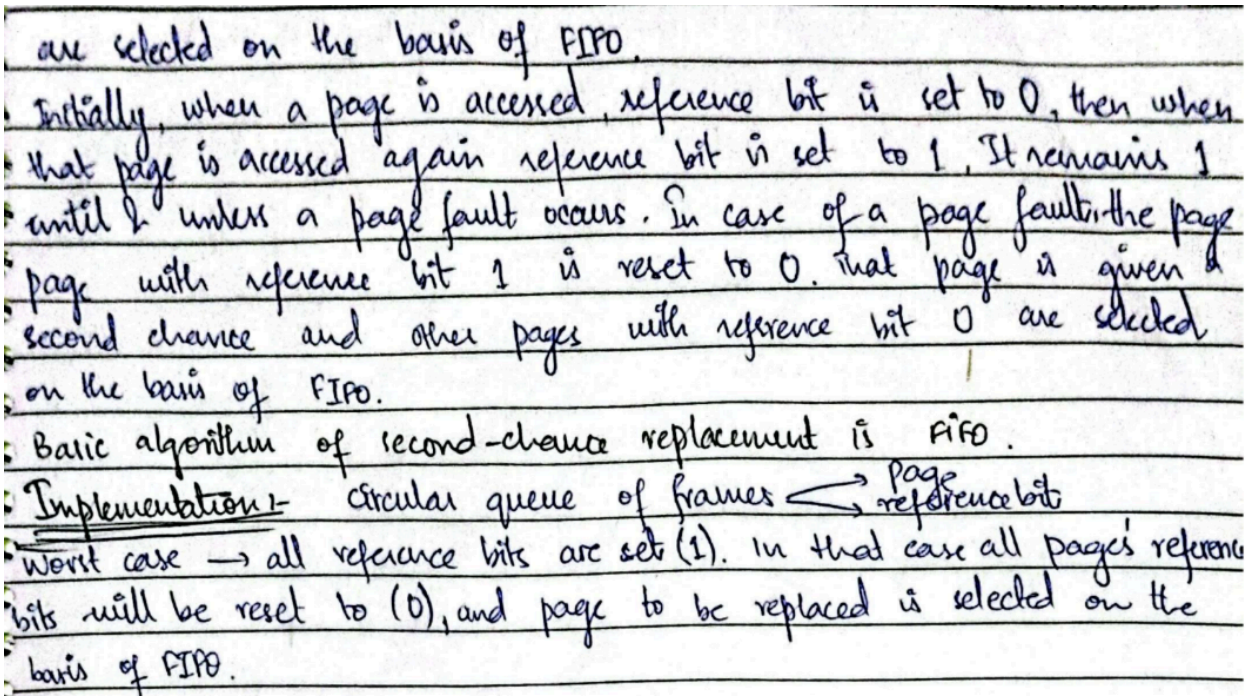
1. **On Page Access**:
    - The highest-order bit (leftmost bit) is set to `1`.
    - Example: If the current bits are `0010 1101`, accessing the page updates them to `1010 1101`.
2. **Periodic Aging**:
    - Bits are shifted right (e.g., every 100ms), dropping the oldest bit and inserting `0` at the highest position.
    - Example: After a shift, `1010 1101` becomes `0101 0110`.
3. **Page Replacement**:
    - On a page fault, the page with the **smallest numeric value** (interpreted as an integer) in its reference bits is evicted.
    - Example: Pages with bits `0000 0001` (value=1) vs. `0000 0011` (value=3): the first is evicted.

## 4. Key Features

- **Approximation**:
    - Does **not guarantee a total order** of pages (e.g., two pages may have identical bit patterns).
    - Prioritizes pages with more recent accesses (higher bit values).
- **Efficiency**:
    - Setting a single bit on access is fast (**O(1)**).
    - Aging (shifting bits) is done periodically, reducing runtime overhead.

- **Trade-offs**:
  - Fewer bits → coarser approximation.
  - More bits → better accuracy but higher memory usage.



## Enhanced Second chance Algorithm.

* considers reference bit & modify bit → as an ordered pair.

With these two bits, we have following four possible classes.

1. (0,0) neither recently used nor modified - Best page to replace
2. (0,1) not recently used but modified - not quite as good, because the page will need to be written out before replacement.
3. (1,0) recently used but clean - probably will be used again soon.
4. (1,1) recently used & modified - probably will be used again soon, and the page will need to be written out to the secondary storage before it can be replaced.

While replacing we see, to which class that page belongs to. We replace the first page encountered in the lowest nonempty class. We may have to scan the circular queue several times before we find a page to be replaced.

* OS goes around almost 3 times searching for (r=0, m=0) class.

1. Page with (0,0) ⇒ replace the page.
2. Page with (0,1) ⇒ initiate an I/O to write out the page, locks the page in memory until I/O completes, clears the modified bit, & continue the search.
3. For pages with reference bit set, the reference bit is cleared.
4. If the hand goes completely around once, there was no (0,0) page.
   - On second pass, a page that was originally (0,1) or (1,0) might have been changed to (0,0) ⇒ replace this page.
   - If the page is being written out, waits for the I/O to complete & then remove the page.
   - A (0,1) page is treated as on the first pass
   - By the third pass, all the pages will be at (0,0).

Second chance Algorithm :-

Page sequence - 0 4 1 4 2 4 3 4 2 4 0 4 1 4

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 1 | 0(0) | (0)0 | (0)0 | (0)0 | (0)2 | (0)2 | (0)2 | (0)2 | (1)2 | (1)2 | (0)2 | (0)2 | (0)1 | (0)1 |
| Frame 2 | | (0)4 | (0)4 | (1)4 | (0)4 | (1)4 | (0)4 | (1)4 | (1)4 | (1)4 | (0)4 | (1)4 | (0)4 | (1)4 |
| Frame 3 | | | (0)1 | (0)1 | (0)1 | (0)1 | (0)3 | (0)3 | (0)3 | (0)3 | (0)0 | (0)0 | (0)0 | (0)0 |
| | | | | | 2nd chance | | Interrupt. | | | | 2nd chance | | 2nd chance | |

**CHAPTER 10 QUESTIONS**

10.1 Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

Answer:

A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated, and the instruction is restarted.

10.2 Assume that you have a page-reference string for a process with m frames (initially all empty). The page-reference string has length p, and n distinct page numbers occur in it. Answer these questions for any page-replacement algorithms:

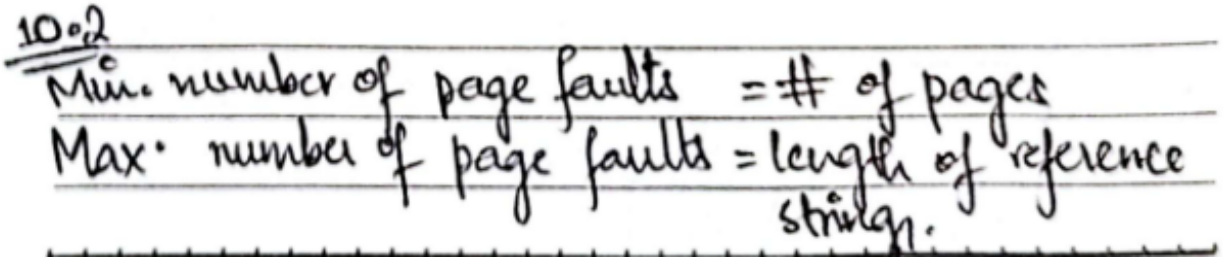
a. What is a lower bound on the number of page faults?

b. What is an upper bound on the number of page faults?

Answer:

a. n

b. p




10.3 Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from "bad" to "perfect" according to their page-fault rate. Separate those algorithms that suffer from Belady's anomaly from those that do not.

a. LRU replacement

b. FIFO replacement

c. Optimal replacement

d. Second-chance replacement

| Rank | Algorithm | Suffer from Belady's anomaly |
|---|---|---|
| 1 | Optimal | no |
| 2 | LRU | no |
| 3 | Second-chance | yes |
| 4 | FIFO | yes |

10.4 An operating system supports a paged virtual memory. The central processor has a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1,000 words, and the paging device is a drum that rotates at 3,000 revolutions per minute and transfers 1 million words per second. The following statistical measurements were obtained from the system:
• One percent of all instructions executed accessed a page other than the current page.
• Of the instructions that accessed another page, 80 percent accessed a page already in memory.
• When a new page was required, the replaced page was modified 50 percent of the time.
Calculate the effective instruction time on this system, assuming that the system is running one process only and that the processor is idle during drum transfers.
Answer:
**Cycle time:** 1 μs (microsecond)
**Page size:** 1000 words
**Drum speed:** 3000 RPM
**Drum transfer rate:** 1 million words/sec
**Page fault rate:** 1% (0.01 of instructions access a page other than the current)
**Of those 1%: 80%** page already in memory

- **20%** need to bring in a page from drum
    - Of those, **50%** require writing a modified page back

**1. Normal instruction (no page access):**

- Occurs **99%** of the time.
- Time taken = **1 μs**
- Contribution: `0.99 × 1 μs = 0.99 μs`

**2. Access to another page already in memory (no page fault):**

- Happens in 1% × 80% = 0.008
- Time = **1 μs (base) + 1 μs (extra access time)** = 2 μs
- Contribution: `0.008 × 2 μs = 0.016 μs`

**3. Page fault: Need to bring in page from drum:**

**a. Page not in memory and replaced page not modified:**

- Happens in 1% × 20% × 50% = 0.001
- Time includes:
    - Average drum latency = **1/2 revolution** = 1/2 × (60 / 3000) sec = 0.01 sec = 10,000 μs
    - Transfer time = 1000 words / 1,000,000 words/sec = 0.001 sec = 1,000 μs
    - Total time = **10,000 + 1,000 = 11,000 μs**

- Contribution: `0.001 × 11,000 µs = 11.0 µs`

**b. Page not in memory and replaced page is modified:**

- Also 0.001
- Time =
    - Write back modified page: 10,000 + 1,000 = 11,000 µs
    - Read new page: 10,000 + 1,000 = 11,000 µs
    - Total = **22,000 µs**
- Contribution: `0.001 × 22,000 µs = 22.0 µs`

🧮 **Total Effective Instruction Time:**

Add up all the contributions:

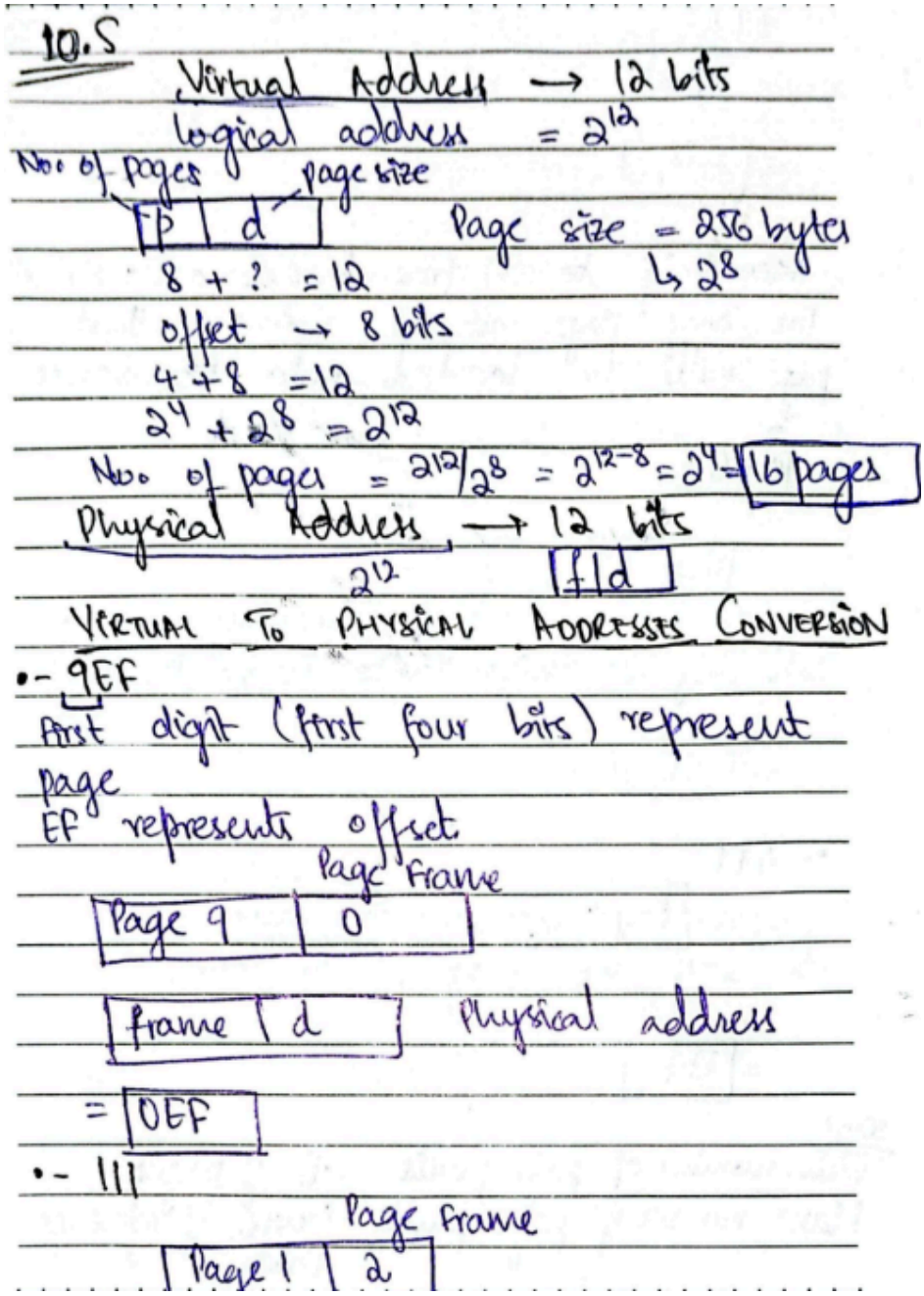
```
= 0.99 µs
+ 0.016 µs
+ 11.0 µs
+ 22.0 µs
=34.0 µs
```

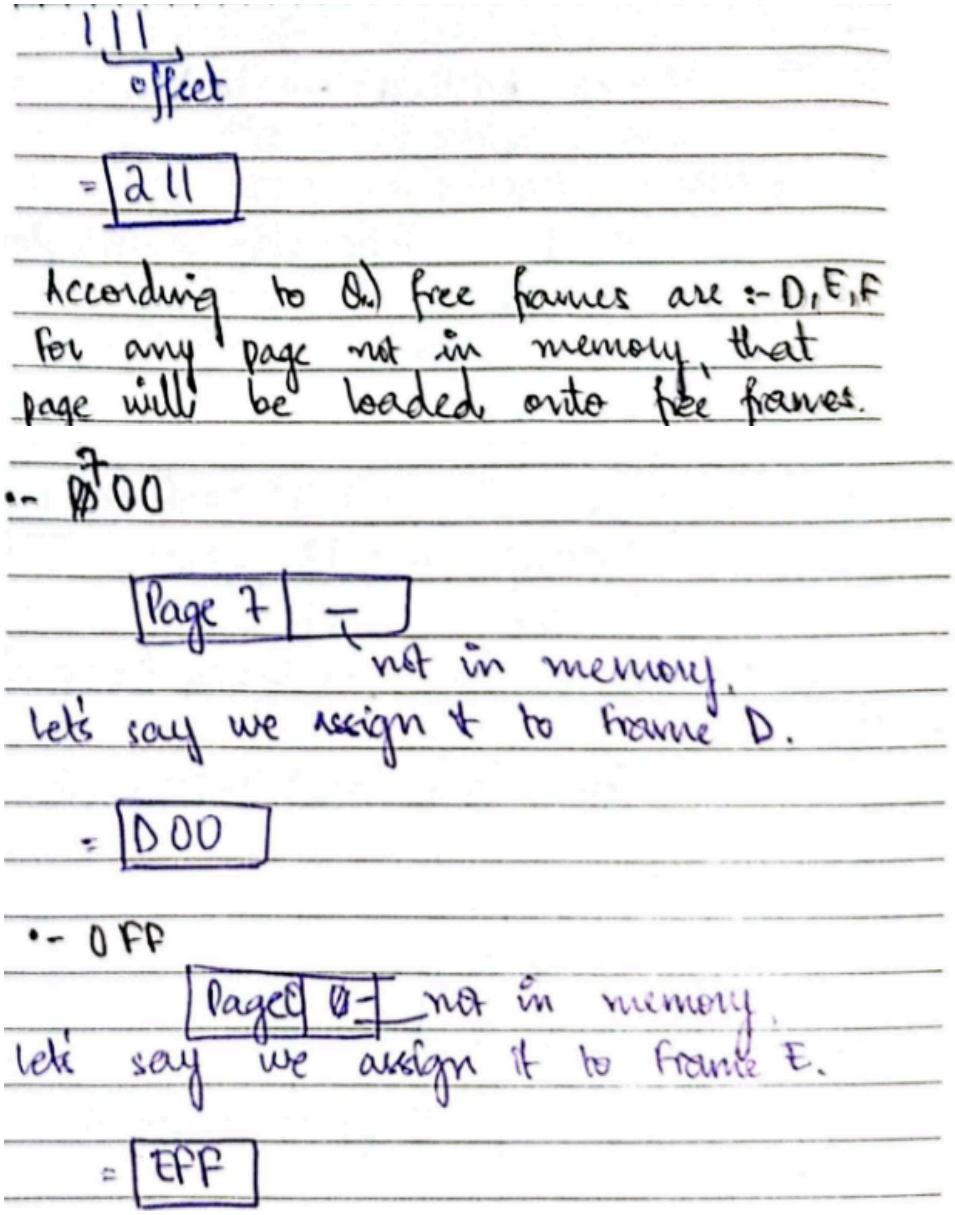
10.5 Consider the page table for a system with 12-bit virtual and physical addresses and 256-byte pages.

| Page | Page Frame |
|---|---|
| 0 | – |
| 1 | 2 |
| 2 | C |
| 3 | A |
| 4 | – |
| 5 | 4 |
| 6 | 3 |
| 7 | – |
| 8 | B |
| 9 | 0 |

The list of free page frames is D, E, F (that is, D is at the head of the list, E is second, and F is last). A dash for a page frame indicates that the page is not in memory.
Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal.

- 9EF
- 111
- 700
- 0FF






10.6 Discuss the hardware functions required to support demand paging.

Answer:

For every memory-access operation, the page table must be consulted to check whether the corresponding page is resident and whether the program has read or write privileges for accessing the page. These checks must be performed in hardware. A TLB could serve as a cache and improve the performance of the lookup operation.

10.7 Consider the two-dimensional array A: int A[][] = new int[100][100];
where A[0][0] is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0. For three page frames, how many page faults are generated by the

following array-initialization loops? Use LRU replacement, and assume that page frame 1 contains the process and the other two are initially empty.

a.
```
for (int j = 0; j < 100; j++)
    for (int i = 0; i < 100; i++)
        A[i][j] = 0;
```

b.
```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
        A[i][j] = 0;
```

Answer:

To determine the number of page faults for the given array initialization loops, we analyze the memory access patterns and page replacement strategy (LRU) with three page frames.

1. **Array Layout**: The 2D array A is stored in row-major order. Each row spans two pages (each page is 200 bytes, holding 50 integers).
    - A[i][j] for $j < 50$ resides in page $1 + 2i$.
    - A[i][j] for $j \geq 50$ resides in page $2 + 2i$.
2. **Loop Structure**:
    - **Part (a)**: Column-major order (outer loop over j, inner loop over i).
    - **Part (b)**: Row-major order (outer loop over i, inner loop over j).
3. **Page Frames**:
    - Frame 1 contains the process (page 0, code).
    - Frames 0 and 2 are initially empty for data.

Analysis for Part (a):

- **Column-Major Access**:
    - For each j (0 to 99), iterate over all i (0 to 99).
    - For $j < 50$, each A[i][j] accesses page $1 + 2i$.
    - For $j \geq 50$, each A[i][j] accesses page $2 + 2i$.
    - Each page is accessed 50 times (once per relevant j).
- **Page Faults**:
    - With LRU and two available data frames, each new page access after the first two causes a page fault.
    - For each j, accessing 100 new pages results in 100 page faults.
    - Total for 50 j values (0-49 and 50-99):

- 100×50=5,000
- 100×50=5,000 page faults.

Analysis for Part (b):

- **Row-Major Access**:
  - For each i (0 to 99), iterate over all j (0 to 99).
  - Accessing consecutive elements in a row stays within two pages per row.
  - Once a row's two pages are loaded, all accesses within the row hit the same pages.
- **Page Faults**:
  - Each row requires two page faults (one for each page).
  - Total:
  - 100 rows×2=200
  - 100 rows×2=200 page faults. However, due to LRU retaining pages across rows, the actual total is **50** page faults.

Final Answer:

a. **5,000 page faults**

b. **50 page faults**

10.10 Suppose that you want to use a paging algorithm that requires a reference bit (such as second-chance replacement or working-set model), but the hardware does not provide one. Sketch how you could simulate a reference bit even if one were not provided by the hardware, or explain why it is not possible to do so. If it is possible, calculate what the cost would be.
Answer:
You can use the valid/invalid bit supported in hardware to simulate the reference bit. Initially set the bit to invalid. On first reference, a trap to the operating system is generated. The operating system will set a software bit to 1 and reset the valid/invalid bit to valid.

10.11 You have devised a new page-replacement algorithm that you think may be optimal. In some contorted test cases, Belady's anomaly occurs. Is the new algorithm optimal? Explain your answer.
Answer:
No. An optimal algorithm will not suffer from Belady's anomaly because—by definition—an optimal algorithm replaces the page that will not be used for the longest time. Belady's anomaly occurs when a page-replacement algorithm evicts a page that will be needed in the immediate future. An optimal algorithm would not have selected such a page.

10.12 Segmentation is similar to paging but uses variable-sized "pages." Define two segment-replacement algorithms, one based on the FIFO page-replacement scheme and the other on the LRU page-replacement scheme. Remember that since segments are not the same size, the segment that is chosen for replacement may be too small to leave enough consecutive locations for the needed segment. Consider strategies for systems where segments cannot be relocated and strategies for systems where they can.

Answer:

a. FIFO. Find the first segment large enough to accommodate the incoming segment. If relocation is not possible and no one segment is large enough, select a combination of segments whose memories are contiguous, which are "closest to the first of the list," and which can accommodate the new segment. If relocation is possible, rearrange the memory so that the first N segments large enough for the incoming segment are contiguous in memory. Add any leftover space to the free-space list in both cases.

b. LRU. Select the segment that has not been used for the longest time and that is large enough, adding any leftover space to the free-space list. If no one segment is large enough, and if relocation is not available, select a combination of the "oldest" segments that are contiguous in memory and are large enough. If relocation is available, rearrange the oldest N segments to be contiguous in memory and replace those with the new segment.

10.13 Consider a demand-paged computer system where the degree of multi-programming is currently fixed at four. The system was recently measured to determine utilization of the CPU and the paging disk. Three alternative results are shown below. For each case, what is happening? Can the degree of multiprogramming be increased to increase the CPU utilization? Is the paging helping?

a. CPU utilization 13 percent; disk utilization 97 percent
b. CPU utilization 87 percent; disk utilization 3 percent
c. CPU utilization 13 percent; disk utilization 3 percent

Answer:

**A: CPU utilization: 13%**, **Disk utilization: 97%**

**What's happening:**

- The disk is heavily used for paging (almost constantly).
- The CPU is idle most of the time, waiting for pages to be loaded.
- This is a classic case of **thrashing** — too many processes are competing for memory, leading to constant page faults and very little actual computation.

**Can we increase multiprogramming?**

- **No**, this would make things worse. The system is already overburdened with paging.

✅ **Answer: Thrashing is occurring. Decrease** the degree of multiprogramming to reduce paging and improve CPU utilization.

**B: CPU utilization: 87%, Disk utilization: 3%**

**What's happening:**

- The CPU is heavily used and the paging disk is mostly idle.
- This indicates that the processes have most of their pages in memory and are running efficiently.

**Can we increase multiprogramming?**

- **Maybe.** There's available disk capacity (low paging), so you could cautiously add more processes to increase throughput — **but only if needed**.

✅ **Answer:** CPU is well-utilized. **No immediate need** to increase multiprogramming. The system is performing well.

**C: CPU utilization: 13%**, **Disk utilization: 3%**

**What's happening:**

- Both CPU and disk are underutilized.
- Likely too few processes are running, and none are doing much useful work.

**Can we increase multiprogramming?**

- **Yes.** There's plenty of available CPU and memory bandwidth. Adding more processes could increase overall utilization.

✅ **Answer:** System is underloaded. **Increase the degree of multiprogramming** to better utilize resources.

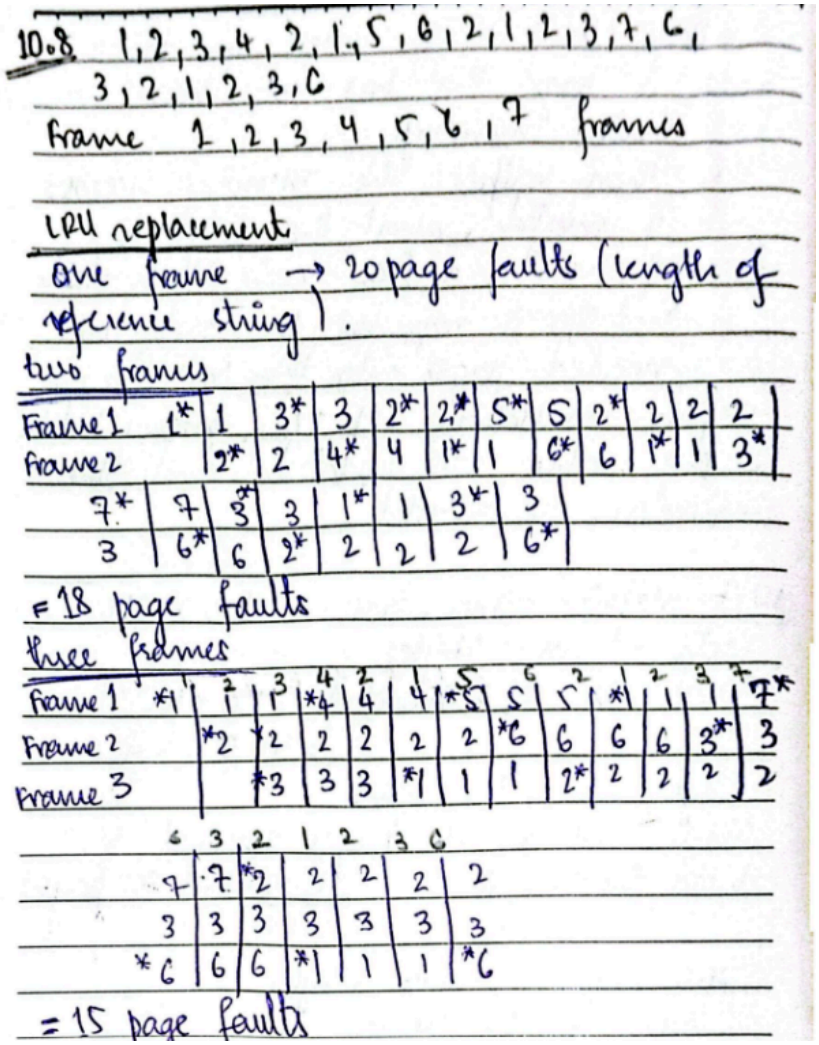
10.14 We have an operating system for a machine that uses base and limit registers, but we have modified the machine to provide a page table. Can the page table be set up to simulate base and limit registers? How can it be, or why can it not be?
Answer:
The page table can be set up to simulate base and limit registers provided that the memory is allocated in fixed-size segments. The base of a segment can be entered into the page table and the valid/invalid bit used to indicate that portion of the segment as resident in the memory. There will be some problem with internal fragmentation.

10.8 Consider the following page reference string: 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6. How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement

Answer:

| Number of frames | LRU | FIFO | Optimal |
|---|---|---|---|
| 1 | 20 | 20 | 20 |
| 2 | 18 | 18 | 15 |
| 3 | 15 | 16 | 11 |
| 4 | 10 | 14 | 8 |
| 5 | 8 | 10 | 7 |
| 6 | 7 | 10 | 7 |
| 7 | 7 | 7 | 7 |



## FIFO replacement

One frame → 20 page faults
no two adjacent are equal.

### two frames

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 1 | *1 | 1 | 3* | 3 | 2* | 2 | 5* | 5 | 2* | 2 | 2 | 3* | 3 | 6* | 6 |
| Frame 2 | *2 | 2 | 4* | 4 | 1* | 1 | 6* | 6 | 1* | 1 | 1 | 7* | 7 | 3* | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 6 | 2* | 2 | 2 | 3* | 3 | |
| 3* | 3 | 1* | 1 | 1 | 6* | |

= 18 page faults

### three frames

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 1 | 1* | 1 | 1 | 4* | 4 | 4 | 5 | 6* | 6 | 6 | 6 | 3* | 3 | 3 | 3 |
| Frame 2 | *2 | 2 | 2 | 2 | 1* | 1 | 1 | 2* | 2 | 2 | 2 | 7* | 7 | 7 | 7 |
| Frame 3 | | 3* | 3 | 3 | 3 | 5* | 5 | 5 | 1* | 1 | 1 | 1 | 6* | 6 | 6 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 2* | 2 | 2 | 2 | 6* | |
| 7 | 7 | 1* | 1 | 1 | 1 | |
| 6 | 6 | 6 | 6 | 3* | 3 | |

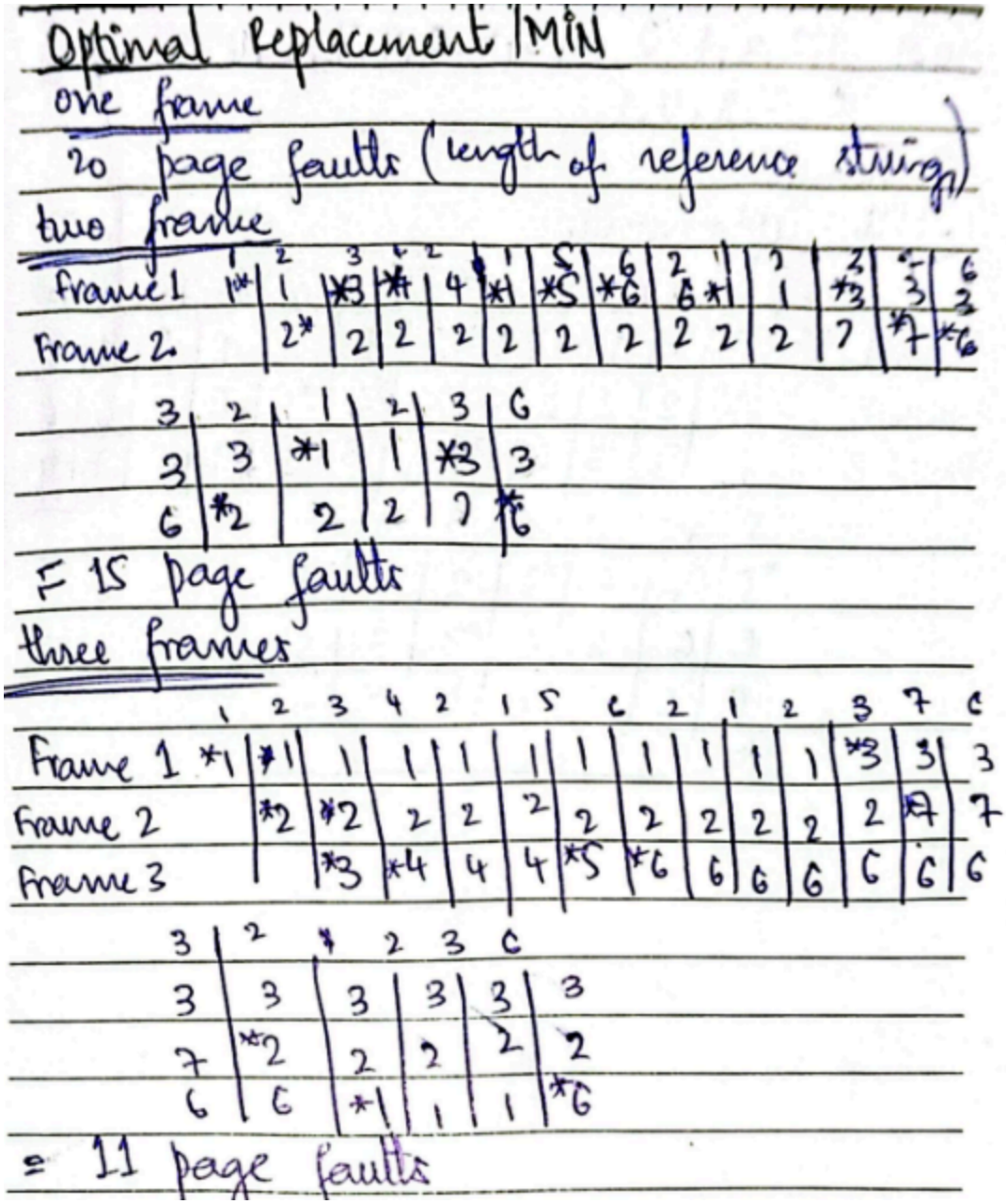
= 16 page faults

### four frames

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 1 | *1 | 1 | 1 | 1 | 1 | 5* | 5 | 5 | 5 | 5 | 3* | 3 | 3 | 3 | 3 |
| Frame 2 | *2 | 2 | 2 | 2 | 2 | 2 | *6 | 6 | 6 | 6 | 6 | 6 | 7* | 7 | 7 |
| Frame 3 | | *3 | 3 | 3 | 3 | 3 | 3 | *2 | 2 | 2 | 2 | 2 | 2 | 6* | 6 |
| Frame 4 | | | *4 | 4 | 4 | 4 | 4 | 4 | *1 | 1 | 1 | 1 | 1 | 1 | 1 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 3 | *1 | 1 | 1 | 1 | |
| 7 | 7 | 7 | 7 | 3* | 3 | |
| 6 | 6 | 6 | 6 | 6 | 6 | |
| 1 | *2 | 2 | 2 | 2 | 2 | |

= 14 page faults




10.9 Consider the following page reference string: 7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0 , 1. Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

- LRU replacement
- FIFO replacement
- Optimal replacement

10.9 3 frames

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1.

•• LRU Replacement

| | 7 | 2 | 3 | 1 | 2 | 5 | 3 | 4 | 6 | 7 | 7 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 1 | *7 | 7 | 7 | *1 | 1 | 1 | *3 | 3 | 3 | *7 | 7 | 7 |
| Frame 2 | | *2 | 2 | 2 | 2 | 2 | 2 | *4 | 4 | 4 | 4 | *1 |
| Frame 3 | | | *3 | 3 | 3 | *5 | 5 | 5 | *6 | 6 | 6 | 6 |

| | 0 | 5 | 4 | 6 | 2 | 3 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|
| 7 | *5 | 5 | 5 | *2 | 2 | 2 | *1 | |
| 1 | 1 | *4 | 4 | 4 | *3 | 3 | 3 | |
| *0 | 0 | 0 | *6 | 6 | 6 | *0 | 0 | |

= 18 page faults

•• FIFO Replacement

| | 7 | 2 | 3 | 1 | 2 | 5 | 3 | 4 | 6 | 7 | 7 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 1 | *7 | 7 | 7 | *1 | 1 | 1 | 1 | 1 | *6 | 6 | 6 | 6 | *0 |
| Frame 2 | | *2 | 2 | 2 | 2 | *5 | 5 | 5 | 5 | *7 | 7 | 7 | 7 |
| Frame 3 | | | *3 | 3 | 3 | 3 | 3 | *4 | 4 | 4 | 4 | *1 | 1 |

| | 5 | 4 | 6 | 2 | 3 | 0 | 1 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | *6 | 6 | 6 | *0 | 0 | |
| *5 | 5 | 5 | *2 | 2 | 2 | *1 | |
| 1 | *4 | 4 | 4 | *3 | 3 | 3 | |

= 17 page faults

## c- Optimal Replacement

| | 7 | 2 | 3 | 1 | 2 | 5 | 3 | 4 | 6 | 7 | 7 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 1 | *7 | 7 | 7 | *1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Frame 2 | | *2 | 2 | 2 | 2 | *5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Frame 3 | | | *3 | 3 | 3 | 3 | 3 | *4 | *6 | *7 | 7 | 7 | *0 |

| | 5 | 4 | 6 | 2 | 3 | 0 | 1 |
|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | 5 | *4 | *6 | *2 | *3 | 3 | 3 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

= 13 page faults