Lab 10

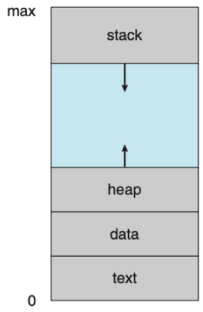
Virtual Memory Management

(Chapter 10)

Section 1: Summary – Self-study before entering the lab [Partially AI generated based on the textbook.]

Fundamental Concepts

Virtual Address Space: The logical view of how a process is stored. It typically starts at address 0 and contains the text (code), data, heap (grows up), and stack (grows down).



Demand Paging: Pages are only loaded into memory when they are "demanded" during execution. This is managed by a pager (lazy swapper).

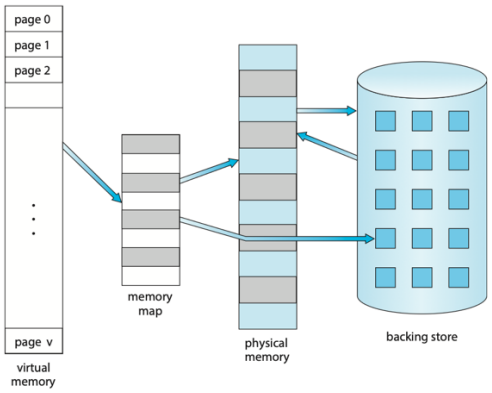
Valid-Invalid Bit: A bit in the page table where 'v' means the page is in memory and 'i' means it is not.

Background

- Programs must be in physical memory to run, but this limits program size to available memory. In reality, much of a program is often unused for example:
  1. Error-handling code rarely runs because errors seldom occur.
  2. Data structures like arrays are often allocated more space than needed.
  3. Some features (like U.S. budget-balancing routines) may go unused for years.

- Even when a whole program is needed, it's rarely all required at once. Running a program that's only partly in memory offers key benefits:
  1. Programs aren't limited by physical memory size—they can use a large virtual address space, making programming easier.
  2. Smaller memory use per program allows more programs to run simultaneously, boosting CPU use and throughput without slowing response or turnaround time.
  3. Less I/O is needed to load or swap code, so programs run faster.

- **Virtual memory** separates the memory that programmers see (**logical memory**) from the actual **physical memory**. This lets programmers work with a much larger virtual memory space, even if the computer has only a small amount of physical memory. As a result, they don't have to worry about physical memory limits and can focus on solving the problem at hand.
- A process's virtual address space is its logical view of memory, usually starting at address 0 and appearing as one continuous block.
- Physical memory is divided into non-contiguous page frames. The memory management unit (MMU) handles mapping the process's logical pages to these physical page frames.



Demand Paging

When you run a program, the computer has two main ways to bring that program from the hard drive (secondary storage) into its active memory (RAM):

1. **Full Loading:** In this approach, the computer copies the **entire** program into memory the moment you start it.

   The Problem: It is wasteful. Imagine a program with 100 different features. If you only use 2 of those features, the computer has wasted space by loading the code for the other 98 features you never touched.

2. **Demand Paging:** Instead of loading everything at once, the system only loads specific "pages" (pieces) of the program **only when they are actually needed.**
   - **How it works:** The program stays on the hard drive. When you click a specific button or run a specific command, the computer "demands" that piece of code and moves it into RAM.
   - **The Benefit:** If you never use a certain part of a program, it never takes up space in your RAM. This makes the system more efficient and allows you to run more programs at the same time because each one is only using a small "slice" of memory.

Demand paging only loads pieces of a program (pages) into RAM when the CPU actually asks for them.

How the System Keeps Track

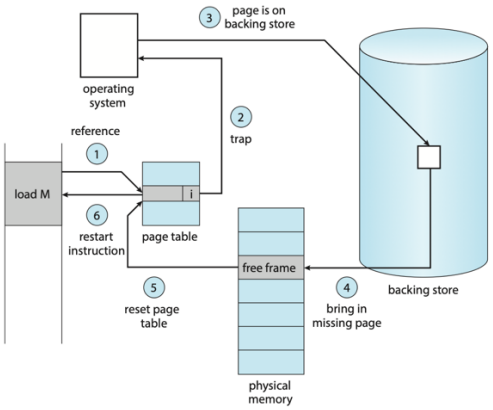
The computer uses a **"Valid-Invalid Bit"** in the page table to know where a page is:

- **Valid (v):** The page is already in RAM.
- **Invalid (i):** The page is either not part of the program or it is still sitting on the hard drive.

What Happens During a Page Fault?

If a program tries to use a page marked "invalid," it triggers a **Page Fault**. The Operating System handles it in three steps:

1. **Check:** The OS looks at internal records to see if the request was a mistake (illegal access) or just a page that hasn't been loaded yet.
2. **Verify:** If it was a mistake, the program crashes. If it was a legal request, the OS prepares to bring the page in from the disk.
3. **Find Space:** The OS finds an empty slot (free frame) in the RAM to hold the new page.
4. **Fetch:** The OS schedules a read from the hard drive (secondary storage) to pull the missing page into the newly found empty frame in RAM.
5. **Update:** Once the data is finished loading, the OS updates its internal tables and the page table. It changes the bit from Invalid (i) to Valid (v) to show the page is now ready in memory.
6. **Restart:** The OS restarts the specific instruction that caused the "trap" in the first place. The program continues running as if the page had been there all along.



Pure Demand Paging

- A memory management strategy where a process begins execution with **zero** pages in physical memory. The operating system only brings a page into RAM at the exact moment it is required by the CPU.
- To support demand paging, the system uses two main hardware components:
  - **Page Table:** A directory that uses a **valid-invalid bit** to track whether a page is currently in RAM or out on the disk.
  - **Secondary Memory (Swap Space):** A high-speed storage area (like an SSD or NVM) that holds the pages not currently in use. This area is often called the **swap device**.
- To implement demand paging, the system must be able to **restart any instruction** from scratch after a page fault.

- When a page fault happens, the OS saves the process's current state (registers and instruction counter). Once the missing page is loaded, the process restarts at the exact same spot, as if nothing happened.
- **The "Worst-Case" Example**
    - Consider a single instruction that performs multiple memory steps, like ADD A + B = C:
        - **Fetch/Decode** the instruction.
        - **Fetch A** (Possible Page Fault here!)
        - **Fetch B** (Possible Page Fault here!)
        - **Add** the values.
        - **Store in C** (Possible Page Fault here!)
- The **Solution:** If a fault occurs at *any* of these steps (e.g., while fetching "B"), the entire instruction is simply restarted from Step 1. Because the state was saved, the CPU just tries again. This time, since "A" and "B" are now in memory, the instruction completes successfully.
- The OS keeps a "buffer" of empty RAM slots called the **Free-Frame List**.
    - **Zero-Fill:** Before giving a free frame to a program, the OS wipes it with zeros so the new program can't see "leftover" data from the previous user (a vital security step).
    - **Refilling:** When the list gets too short, the OS kicks out old pages to make room for new ones.

Performance of Demand Paging

In demand paging, the system's performance is measured by how much the paging process slows down the computer. This is expressed as the **Effective Access Time (EAT)**.

The Key Formula

The performance depends heavily on the **page-fault rate (p)**, which is the probability that a memory access will result in a fault ($0 <= p <= 1$).

$$EAT = (1 - p) \cdot \text{memory access time} + p \cdot (\text{page fault time})$$

Calculation Example: standard numbers found in the Abraham Silberschatz book.

- **Memory Access Time:** 200 nanoseconds.
- **Average Page-Fault Service Time:** 8 milliseconds (8,000,000 nanoseconds).

Scenario A: 1 fault every 1,000 accesses (p = 0.001)

$$EAT = (1 - 0.001) \times 200 + 0.001 \times 8{,}000{,}000$$
$$EAT = 199.8 + 8{,}000 = 8{,}199.8 \text{ ns}$$

Result: The computer is roughly **40 times slower** than normal just because of a 0.1% fault rate.

Scenario B: To keep slowdown under 10%

If we want the performance to stay within 10% of the 200 ns speed (meaning we want EAT to be less than 220 ns):

$$220 > 200 + 7{,}999{,}800 \times p$$
$$p < 0.0000025$$

Result: To maintain high performance, you can only afford **one page fault for every 400,000 memory accesses.**

As shown in the example, the "cost" of a page fault is massive compared to a standard memory check. The page fault time includes:

1. **Service the trap:** The OS takes over.
2. **Read the page:** The slow process of moving data from the disk to RAM.
3. **Restart the process:** Returning control to the user.

Because the difference between RAM speed (nanoseconds) and Disk speed (milliseconds) is so vast, even a tiny increase in the page-fault rate can make a computer feel significantly slower.

Copy-On-Write

Copy-on-Write (COW) is a resource-management technique used during process creation (like the fork() system call) to make the process faster and more memory-efficient.

How it Works

- **Shared Beginnings:** When a parent process creates a child process, they initially **share the exact same pages** in physical memory.
- **Read-Only Access:** Both processes can read these pages. As long as they are only reading, no extra memory is used.
- **The Trigger (The Write):** If either the parent or the child tries to **modify** (write to) a shared page, the Operating System intervenes.
- **The Copy:** The OS creates a physical copy of that specific page for the process that wants to change it. Now, the modification only affects that process's private copy.

Why We Use It

- **Efficiency:** Most child processes immediately call exec() to start a new program, meaning they would have wasted time copying the parent's memory only to throw it away.
- **Speed:** Creating a process becomes near-instant because only a few small tables are copied initially, not the entire memory.

Simple Example

Imagine two students (Parent and Child) sharing one **textbook**.

- They both read from the same book to save money (**Shared RAM**).
- If the Child wants to highlight a sentence, the teacher stops them, photocopies that one page, and gives it to the Child (**Copy-on-Write**).
- The Child highlights their own page, while the Parent's original book remains clean.

Page Replacement

Page replacement occurs when a process demands a page, but there are **no free frames** left in memory. To make room, the OS must swap an existing page out of RAM to the disk.

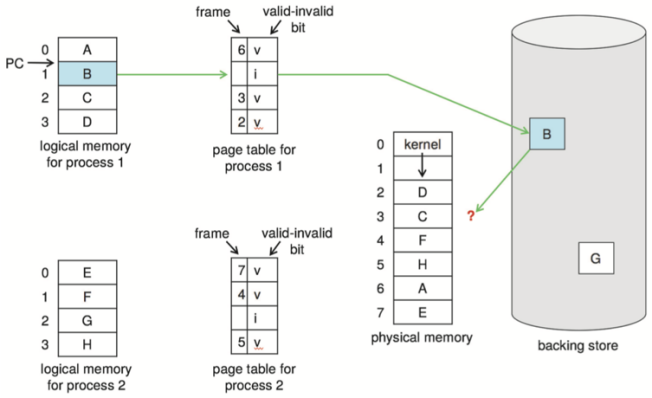
1. The Basic Mechanism

If no frame is free, the OS follows these steps:

- **Find** the desired page on the disk.
- **Select a victim frame** currently in memory.
- **Write the victim** to the disk and change its page table bit to "invalid."
- **Read the new page** into the freed frame and set its bit to "valid."
- **Restart** the user process.

The Modify Bit (**Dirty Bit**): To speed this up, the OS uses a modify bit.

- If the bit is **0**, the page hasn't changed since it was loaded. The OS can simply overwrite it (skipping Step 3).
- If the bit is **1**, the page was modified, so the OS *must* write it back to the disk to save changes.



2. FIFO (First-In, First-Out)

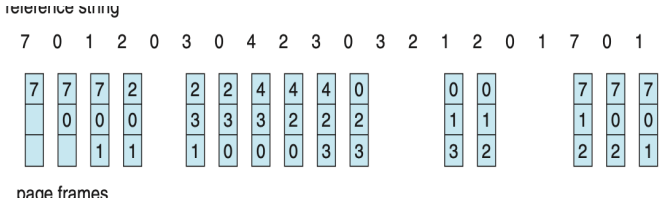
The simplest algorithm: the "oldest" page in memory is the first to be replaced.

- **The Problem:** It can suffer from **Belady's Anomaly** a strange situation where giving the system *more* RAM actually results in *more* page faults.
- **Example Calculation:**

Reference string: 7, 0, 1, 2, 0, 3 with 3 frames.

1. 7 (Fault) → [7, _, _]
2. 0 (Fault) → [7, 0, _]
3. 1 (Fault) → [7, 0, 1]
4. 2 (Fault, replaces 7) → [2, 0, 1]

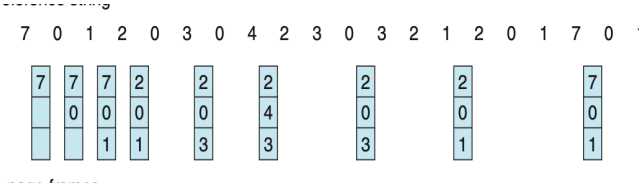
5. 0 (Hit) → [2, 0, 1]
6. 3 (Fault, replaces 0) → [2, 3, 1]
7. Total: 5 Page Faults



3. Optimal Algorithm (OPT)

This algorithm replaces the page that will **not be used for the longest period of time**.

- **The Catch:** It is impossible to implement because the OS cannot predict the future. It is used only as a benchmark to measure how well other algorithms perform.

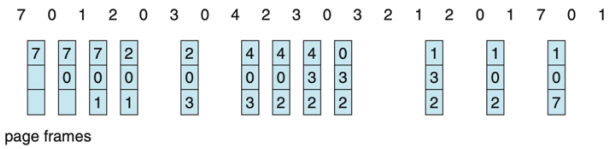


4. LRU (Least Recently Used)

This replaces the page that has not been used for the longest time. It is a "backward-looking" version of the Optimal algorithm.

Example Calculation (Same string): 7, 0, 1, 2, 0, 3 with **3 frames**.

1. 7, 0, 1 (3 Faults) → [7, 0, 1]
2. 2 (Fault, replaces 7 because it's oldest) → [2, 0, 1]
3. 0 (Hit, 0 is now the "most recently used") → [2, 0, 1]
4. 3 (Fault, replaces 1 because 2 and 0 were used more recently) → [2, 0, 3]
5. Total: 5 Page Faults (In longer strings, LRU is usually much better than FIFO).



LRU is a great algorithm, but it is hard to implement because the computer must track exactly when every page was last used. This requires special hardware to avoid slowing down the system.

There are two main ways to implement it:

1. **Counters (Time-Stamps):** Every page entry has a "time-of-use" field.
   - **How it works:** Every time a page is used, the current time (from a CPU clock) is copied into that page's counter.

- **To Replace:** The OS searches the entire table for the page with the **smallest** (oldest) time value.
- **Drawback:** It requires a time-consuming search every time a page needs to be swapped out.

2. Stack (Linked List): The system maintains a stack of page numbers.
   - **How it works:** Whenever a page is used, it is pulled out of the stack and moved to the top.
   - **The Result:** The top is always the Most Recently Used (MRU), and the bottom is always the Least Recently Used (LRU).
   - **Benefit:** No searching is required; the OS just looks at the bottom of the stack to find the victim.

Why LRU is "Safe" (No Belady's Anomaly)

LRU is a **stack algorithm**. This means that if you give the computer more RAM (more frames), the pages that were in memory with less RAM are **guaranteed** to still be there. Because of this, adding more memory will never accidentally cause *more* page faults.

The Bottom Line

Both methods require fast hardware. If the software had to manage these counters or stacks manually for every single memory access, the computer would run **10 times slower**.

5. LRU-Approximation (Clock Algorithm)

True Least Recently Used (LRU) page replacement needs special hardware to track exactly how often and when pages are used but most systems don't have that. Instead, they use simpler methods that **approximate LRU**.

Many systems include a **reference bit** for each page. The hardware sets this bit to 1 whenever the page is accessed (read or written). The OS clears all reference bits to 0 when a program starts. Later, by checking which bits are set, the OS can tell which pages have been used—but not the exact order. This basic info helps build better replacement strategies.

A. Additional-Reference-Bits Algorithm

To get more detail about **how recently** pages were used, the OS can record reference bits over time. For each page, it keeps an 8-bit history register. Every fixed interval (e.g., every 100 ms), the OS shifts the current reference bit into the leftmost position of this register and shifts the rest right.

- A register like 00000000 means the page hasn't been used in the last 8 intervals.
- 11111111 means it was used every time.
- 11000100 shows more recent use than 01110111.

The OS treats these 8-bit values as numbers: the **smallest number** indicates the least recently used page and is the best candidate for replacement. If multiple pages tie for the lowest value, the OS can replace them all or use FIFO to pick one.

Fewer bits can be used to speed things up. With just **one bit** (the basic reference bit), this becomes the **Second-Chance algorithm**.

B. **Second-Chance (Clock) Algorithm**

This is a smarter version of FIFO. The OS uses a circular list (like a clock face) and a pointer to track pages.

- When a page needs to be replaced, the OS checks the page the pointer is on.
- If its reference bit is **0**, it's replaced.
- If the bit is **1**, the page gets a "second chance": the bit is cleared, and the pointer moves to the next page.
- This continues until a page with a 0 bit is found.

In the worst case (all bits are 1), the pointer loops once around, clears all bits, and then replaces the next page. If all pages always have their bits set, this method falls back to plain FIFO.

C. Enhanced Second-Chance Algorithm

This version uses **two bits** per page:

- **Reference bit** (was it used recently?)
- **Modify bit** (was it changed?)

This creates four categories:

1. (0, 0) – Not used, not modified → best to replace
2. (0, 1) – Not used, but modified → okay to replace, but must write it back to disk first
3. (1, 0) – Recently used, not modified → likely needed soon
4. (1, 1) – Recently used and modified → likely needed soon, and writing it back is costly

When replacing a page, the OS scans the circular queue and picks the first page in the lowest nonempty category (starting with category 1). This reduces unnecessary disk writes by favouring clean (unmodified) pages.

This approach balances simplicity, speed, and smart memory management—without needing expensive hardware.

Thrashing

Allocation of Frames discusses how an operating system decides how many physical memory frames to give each process and how to distribute the available frames among competing processes.

Key points include:

- Minimum number of frames: A process needs a minimum number of frames to hold the pages involved in a single instruction's execution (e.g., for instructions that span two pages or access multiple operands). This minimum depends on the hardware's instruction set.
- Allocation algorithms:
  - Equal allocation: Each process gets the same number of frames. Simple but inefficient if processes have different memory needs.
  - Proportional allocation: Frame allocation is based on process size (e.g., a process using 10% of memory gets 10% of frames).
  - Priority-based allocation: Higher-priority processes receive more frames.
- Global vs. Local Replacement:
  - In global replacement, a process can replace any page in memory (including pages from other processes). This allows flexible frame sharing but can harm other processes' performance.
  - In local replacement, a process can only replace its own pages. This provides more predictable performance but less flexibility.
- Thrashing: If a process doesn't get enough frames, it may constantly page in and out, spending more time on paging than useful work. Thrashing occurs when total memory demand exceeds available memory, causing low CPU utilization. The OS can prevent this by using a working-set model or page-fault frequency (PFF) scheme to adjust frame allocation dynamically.

Section 2: Discussion – selected questions only (1 Hour)

1. Consider the page table for a system with 12-bit virtual and physical addresses with 256-byte pages. The list of free page frames is D, E, F (D is at the head of the list, E is second, and F is last)

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal. A dash for frame indicates that the page is not in memory. If the frame is not in memory, it will be brought to one of the free page frames in order.

9EF
111
700
0FF

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

2. Consider the following page table in a demand paging system. Assume the page size of 2000 bytes. Assume the usage of decimal values.

Check whether the virtual addresses given below would generate a page fault? If it do not generate page fault, identify the physical address.

- 10230
- 5213
- 100
- 8125

Frame	Valid-Invalid bit
5	v
22	i
300	v
150	v
30	i
50	i
120	v
101	v

3. On a system using demand-paged memory, it takes 200 ns to satisfy a memory request if the page is in memory. If the page is not in memory, the request takes 7 ms if a free frame is available or the page to be swapped out has not been modified. It takes 15 ms if the page to be swapped out has been modified. What is the effective access time if the page fault rate is 5%, and 60% of the time the page to be replaced has been modified/free frame is not available?

4. Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus each memory reference through the page table takes two accesses.

   To improve this time, an associative memory is added that reduces access time to one memory reference if the page-table entry is in the associative memory.

   Assume that 80 percent of the accesses are in the associative memory and 10 percent cause page faults. What is the effective memory access time?

5. 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

   Assume the availability of FOUR frames and al the frames are initially empty. How many page faults would occur for First in First out (FIFO) page replacement algorithm? Identify the final pages in the frames.

6. 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

   Assume the availability of FOUR frames and al the frames are initially empty. How many page faults would occur for Least Recently Used (LRU) page replacement algorithm? Identify the final pages in the frames.

7. 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

   Assume the availability of FOUR frames and al the frames are initially empty. How many page faults would occur for optimal page replacement algorithm? Identify the final pages in the frames.

8. Given a system with four-page frames, the following table indicates page number, load time, last reference time, dirty bit, and reference bit.

Page number	Load Time	Last Reference Time	Dirty Bit	Reference bit
0	167	374	1	1
1	321	321	0	0
2	254	306	1	0
3	154	331	0	1

   Identify the victim (page to be replaced) for the following algorithms.

   - First-In-First-Out (FIFO)
   - Least-Recently-Used (LRU)
   - Reference bit – Dirty bit combination
   - Second Chance

9. Assume that a computer system has four frames and uses six reference bits to implement additional-reference bits (aging) page-replacement algorithm. At the first clock tick, the reference bits for pages 0 through 3 are 1,0,1, and 1 respectively. Values at subsequent ticks are 0011,1001,0110,1100, and 0001.

    - After the last tick, what is the value of the reference bits for all four pages?
    - Identify the victim page for page replacement.

Section 3: Practical exercises

Exercise 1: [The Virtual Address Layout]

Every Linux process has its own virtual memory map. You can view this using the `/proc` file system.

1. Open a terminal and run `sleep 1000 &`.
2. Find the Process ID (PID) using `ps`.
3. Run: `cat /proc/[PID]/maps`.

You will see memory ranges for the **executable code**, the **heap**, and the **stack**. These are *virtual* addresses. Two different programs might both use virtual address `0x400000`, but they map to different physical locations in your RAM.

Exercise 2: [C Code - Demand Paging & Page Faults]

The following program allocates a large amount of memory but doesn't use it immediately. Because of **Demand Paging**, the OS won't actually give the program physical RAM until the program writes to a specific address.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/resource.h>

void print_page_faults() {
    struct rusage usage;
    getrusage(RUSAGE_SELF, &usage);
    printf("Minor Page Faults: %ld\n", usage.ru_minflt);
}

int main() {
    int page_size = getpagesize();
    printf("System Page Size: %d bytes\n", page_size);

    printf("\n--- Initial State ---\n");
    print_page_faults();
    // 1. Allocate 100MB of virtual memory
    // This is 'Lazy Allocation' - no physical frames are assigned yet.
    size_t size = 100 * 1024 * 1024;
    char *buffer = malloc(size);

    printf("\n--- After malloc (100MB) ---\n");
    print_page_faults(); // Faults should remain low

    // 2. Access the memory
    // Writing to the buffer triggers 'Demand Paging'
    printf("\n--- Accessing Memory (Writing to pages) ---\n");
    for (int i = 0; i < size; i += page_size) {
        buffer[i] = 'A'; // Trigger a page fault for every page
    }

    print_page_faults(); // Faults will jump significantly

    free(buffer);
    return 0;
}
```

Exercise 3: [C Code - Copy-on-Write (COW)]

This code demonstrates the efficiency of `fork()`. The parent and child share the same physical memory until one of them writes to it.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int global_var = 10; // This sits in a data page

int main() {
    printf("Parent PID: %d, Global Var Address: %p\n", getpid(), &global_var);

    pid_t pid = fork();

    if (pid == 0) { // Child Process
        printf("Child: Reading Global Var: %d\n", global_var);
        // Initially, Child and Parent share the SAME physical frame for global_var.

        printf("Child: Writing to Global Var...\n");
        global_var = 20; // This triggers COPY-ON-WRITE

        printf("Child: New Global Var Value: %d at address %p\n", global_var,
&global_var);
    } else { // Parent Process
        wait(NULL);
        printf("Parent: Global Var is still: %d\n", global_var);
    }

    return 0;
}
```

References:

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, “Operating System Concepts”, 9/E, John Wiley & Sons, 2013.
2. William Stallings, "Operating Systems: Internals and Design Principles", 7th Edition, Prentice Hall, 2011
3. J. Archer Harris, “Operating Systems”, McGraw-Hill,2002.