MEMORY MANAGEMENT:

GARBAGE COLLECTION AND RUST'S OWNERSHIP MODEL

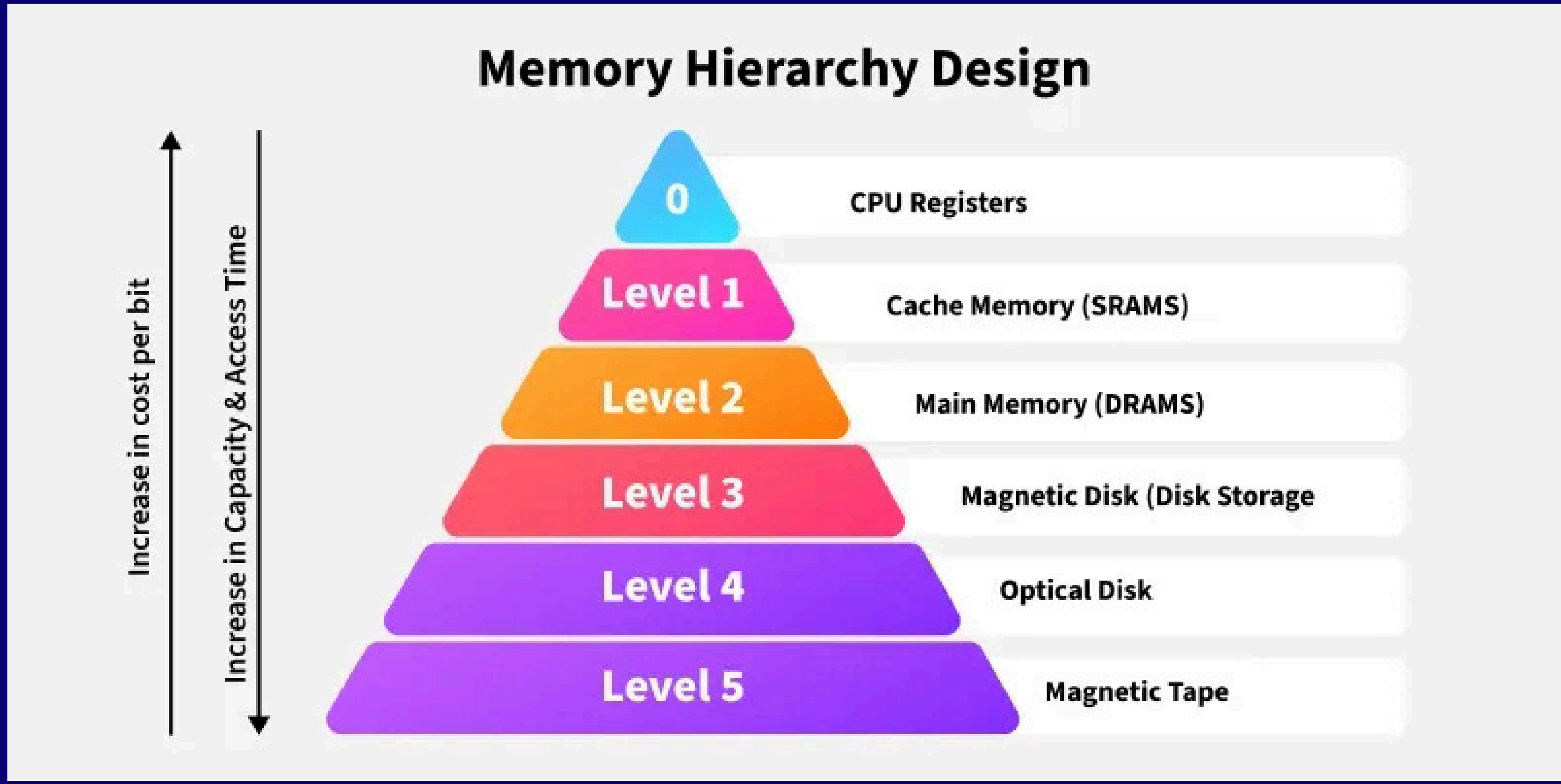
Understanding Different Approaches to Memory Safety

INTRODUCTION TO MEMORY MANGAMENT

DEFINITION: COORDINATION MEMORY ALLOCATION/DE-ALLOCATION BETWEEN PROGRAMS

Key Objectives:

- Process Isolation(Security)
- Efficient resource utilization
- Performance Optimization



MEMORY MANAGEMENT TECHNIQUES

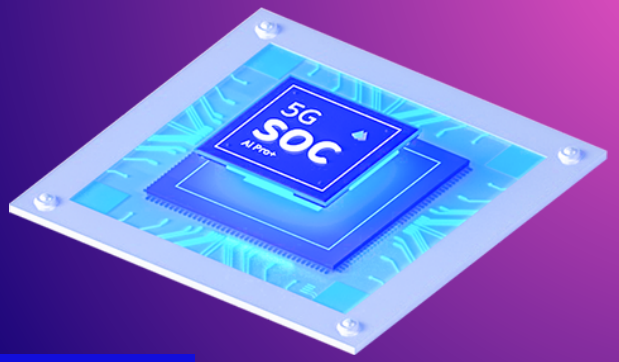
Technique	Pros	Cons	Use Case
Manual	Full Control	Error-Prone	Embedded-Systems
Garbage Collection	Automatic	Overhead	Java/Python
Rust Ownership	Safe+perfomant	Steep Learning Curve	Systems Programming

MANUAL MEMORY MANAGEMENT (CODE)


```
// C example

int* arr = malloc(100 * sizeof(int));

if(arr == NULL) exit(1);  // Critical check

free(arr); // Must explicitly free
arr = NULL; // Prevents accidental use of a dangling pointer
```

Key Points:

- Risk: Memory leaks if free() omitted
- Danger: Dangling pointers if freed memory accessed



GARBAGE COLLECTION DEEP DIVE

1) Mark-Sweep (Basic Algorithm)

- Mark Phase:Traverses object graphs starting from roots(globals,stack variables) and mark unreachable object
- Sweep Phase: Clears unmarked

```
//JavaScript example (mark phase
simulated)

let root={data:"root"}
let orphan={data:"unreachable"}
//Only 'root' is marked; 'orphan' is swept
```

2)Generational Collection

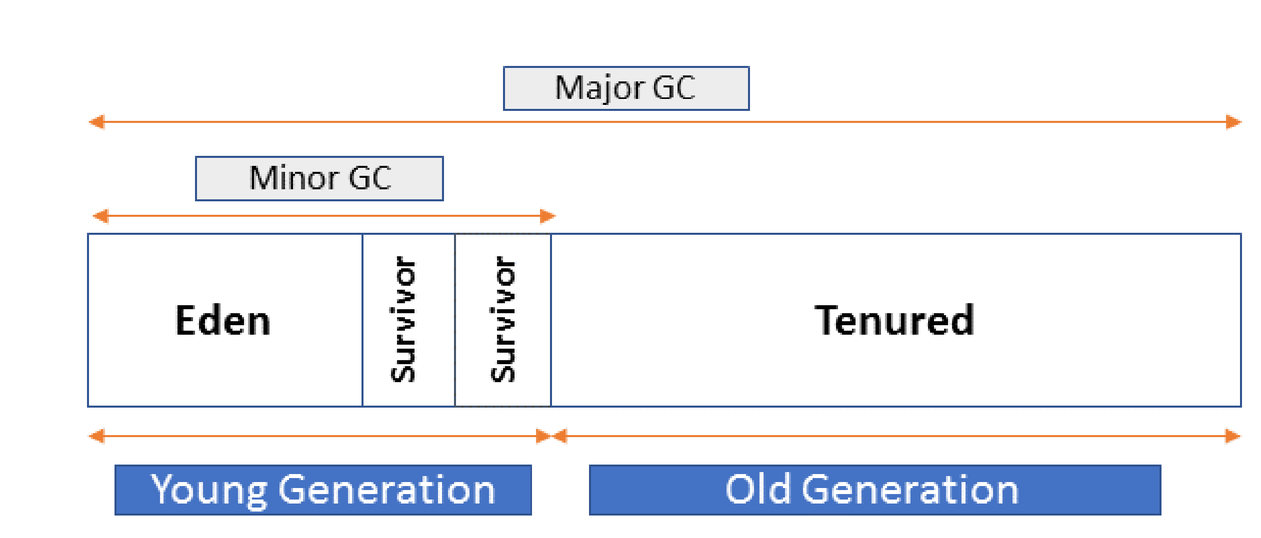
Heap Structure:

- Young generation(Eden+Survivors): Short lived

objects(loop variables)

- Old generation: Long lived objects(e.g. Caches)

Optimization:Frequent G.C. in young gen,rare in old gen



GARBAGE COLLECTION DEEP DIVE

3)Reference Counting(Python Style)

- Each object has a reference count, tracking the number of references pointing to it
- When the reference count drops to zero, the object is considered garbage and can be collected
- Languages: Python (though it also uses other techniques to handle circular references

```
#Python reference example
import sys
x=[]
print(sys getrefcount(x)) #output 2
```

ADVANCED GC STRATEGIES & OPTIMIZATION

Advanced Garbage Collectors

a. G1 (Garbage First) GC (Java)
- Splits heap into regions rather than generations.
- Performs incremental compaction.
- Reduces long pause times compared to CMS.

b. ZGC (Java 11+)
- Ultra-low pause times (<10ms), scales with heap size.
- Uses colored pointers to track object states.

c. Shenandoah GC (Java 12+)
- Fully concurrent GC, reducing pause times.
- Region-based like G1, but performs concurrent compaction.

d. Immix (Rust, Java)
- Uses line-marking and bump allocation.
- Hybrid approach between copying and mark-sweep.

PERFORMANCE COMPARISON

Aspect	Garbage Collection (GC)	Manual Memory Management
Memory Allocation	Automatic memory allocation and cleanup.	Requires explicit allocation (malloc) and deallocation (free).
Performance Overhead	GC introduces runtime overhead due to collection cycles (e.g., GC pauses).	Faster execution as there is no runtime overhead for memory cleanup.
Safety	Prevents memory leaks and dangling pointers automatically.	Prone to errors like memory leaks and dangling pointers if not managed correctly.
Real-Time Suitability	Not ideal for real-time systems due to unpredictable GC pauses.	Suitable for real-time systems where precise control over memory is required.
Ease of Use	Simplifies development by automating memory management.	Requires careful programming and debugging to avoid errors.

RUST OWNERSHIP MODEL

Rust's ownership model ensures memory safety without needing a garbage collector. It helps prevent data races, dangling pointers, and memory leaks by enforcing strict rules at compile time.

Key Rules of Ownership

- Each value in Rust has a single owner (a variable that controls the value).
- When the owner goes out of scope, Rust automatically deallocates the value (no manual memory management needed).
- Ownership can be transferred (moved) or temporarily borrowed to avoid unnecessary copies.

rust

```rust
fn main() {
 let v = vec![1,2,3]; // Heap allocation
 let v2 = v; // Ownership transfer
 // println!("{:?}", v); // Compile error!
}
```

BORROWING & LIFETIMES

1. Borrowing

- Immutable Borrowing (&T) – Multiple read-only references allowed.
- Mutable Borrowing (&mut T) – Only one mutable reference at a time.
- Prevents data races and ensures safe memory access.

```
let mut s = String::from("Hello");
let r1 = &s; //  Immutable borrow
let r2 = &mut s; //  Error! Cannot have mutable + immutable at the same time.
```

2. Lifetimes

- Ensures references remain valid and prevents dangling references.
- 'a (lifetime annotation) specifies how long a reference should live.

```
fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() { s1 } else { s2 }
}
```

- Ensures s1 and s2 outlive the returned reference.

GARBAGE COLLECTION (GC) VS RUST'S OWNERSHIP MODEL

Aspect	Garbage Collection (GC)	Rust Ownership Model
Safety Mechanism	Runtime checks to identify unreachable objects and clean them up.	Compile-time checks enforce ownership, borrowing, and lifetimes.
Performance Impact	Runtime overhead due to GC cycles, causing pauses.	No runtime overhead; memory is managed at compile time.
Ease of Use	Easier for developers as memory management is automated.	Steeper learning curve due to strict rules for ownership and borrowing.
Memory Leaks Prevention	Prevents leaks automatically unless circular references occur.	Guarantees no leaks or dangling pointers through strict compile-time checks.
Use Cases	Ideal for high-level applications (e.g., Java/ Python).	Suitable for systems programming and performance-critical applications.

VIRTUAL MEMORY SYSYTEM

Components of Virtual Memory System

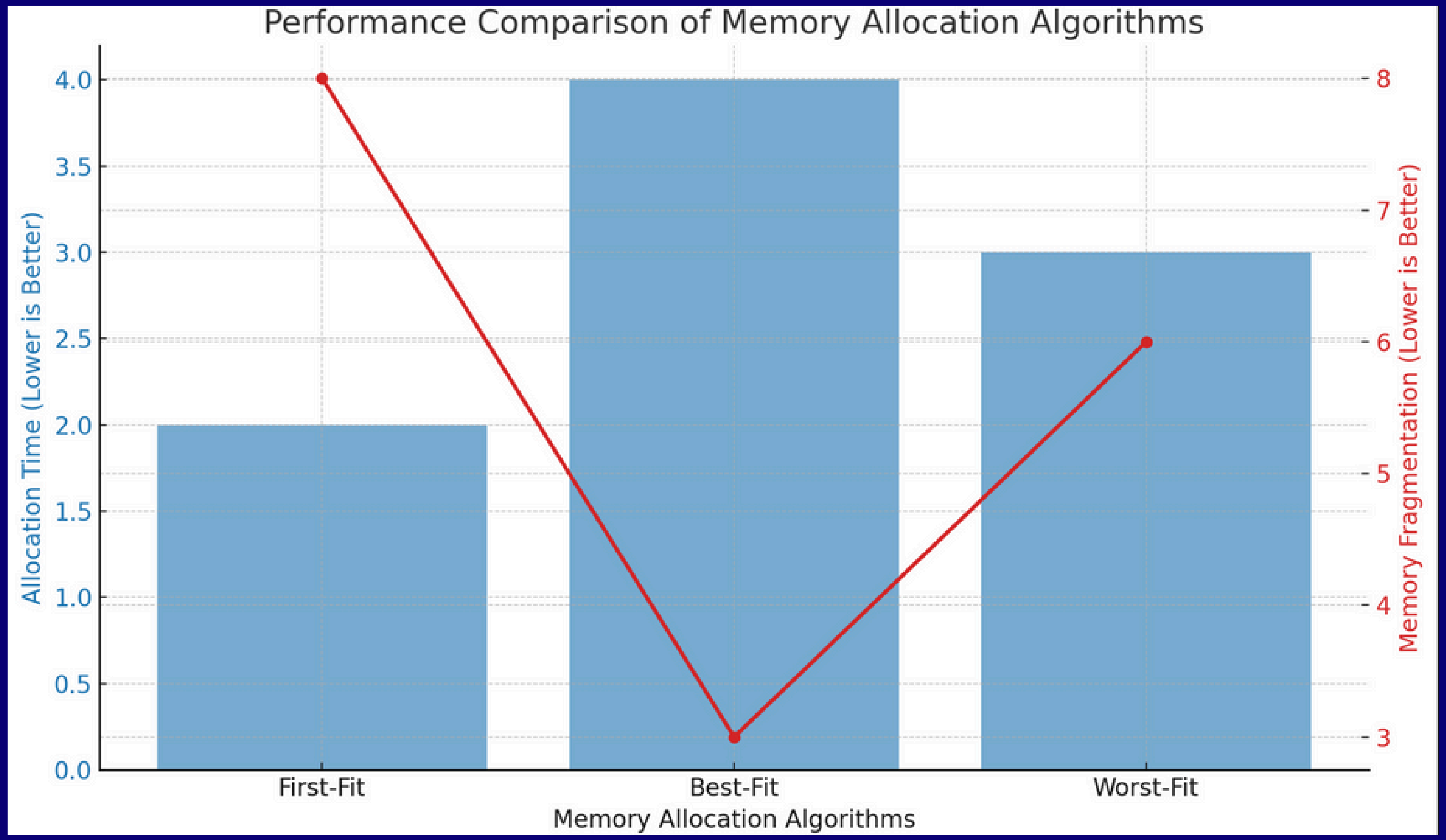
1. Page Tables (Address Translation)
   - Maps virtual addresses to physical addresses by storing the page number → frame number mapping.
   - Used for memory isolation between processes, ensuring each process gets its own logical address space.

2. TLB (Translation Look aside Buffer)
   - High-speed cache that stores recently used virtual-to-physical address translations to speed up access.
   - Reduces page table lookups, improving CPU performance by minimizing memory access delays.

3. Demand Paging
   - Loads pages into memory only when needed, reducing initial memory allocation.
   - Uses page faults to bring missing pages from disk into RAM, optimizing memory usage but potentially slowing execution if too many page faults occur.

MEMORY ALLOCATION ALGORITHMS

- **First-Fit:** Scans memory from the beginning and allocates the first available block that fits.
- **Best-Fit:** Finds the smallest available block that fits the request.
- **Worst-Fit:** Finds the largest block to leave the biggest remaining free space.



MEMORY FRAGMENTATION

Memory Fragmentation:
Fragmentation occurs when memory is inefficiently utilized, leading to wasted space. It can be categorized into internal and external fragmentation.

1. Internal Fragmentation

- Occurs within allocated memory blocks when a process is given more memory than it actually needs.
- Example :If a 1025B process is allocated a 2048B block, 1023B remains unused

2. External Fragmentation

- Occurs when free memory is split into small non-contiguous blocks, preventing larger allocations
- Example: If memory has free blocks of 100B, 500B, 200B, but a request for 600B fails, even though 800B is free in total.

OS MEMORY MANAGEMENT (LINUX EXAMPLE)

Content:

1) malloc() and System Calls (brk() / mmap())

- malloc() dynamically allocates memory in user space.

It internally calls:

- brk() → Expands/shrinks the heap for small allocations.
- mmap() → Allocates large memory regions (e.g., for big arrays or shared memory)

2)Kernel allocators:

- Buddy system (power-of-2 blocks):Splits memory into power-of-2 sized blocks (e.g., 4KB → 2KB → 1KB).
- Slab allocator (object caching):Pre-allocates memory for frequently used objects (e.g., process control blocks).

PERFORMANCE OPTIMIZATION

1. Cache Locality Principles

- Spatial Locality → Data close together in memory is accessed together (e.g., arrays).
- Temporal Locality → Recently accessed data is likely to be accessed again (e.g., loop variables).

2. False Sharing Prevention

- False sharing happens when multiple threads modify variables in the same cache line, causing unnecessary cache invalidations.
- Solution → Align frequently updated variables to separate cache lines to prevent conflicts.

3. Example: Padding in C Structs

```
#include <stdio.h>

struct AlignedStruct {
   int x;
   char padding[60];  // Ensures x and y are in different cache lines
   int y;
};

int main() {
   printf("Size of struct: %lu bytes\n", sizeof(struct AlignedStruct));
   return 0;
}
```

DEBUGGING MEMORY ISSUES

1. Valgrind (C/C++)

- Detects memory leaks, invalid accesses, and uninitialized memory.
- Commonly used tool: memcheck.

2. Rust's Borrow Checker

- Prevents memory leaks and unsafe memory access at compile time.
- Enforces ownership, borrowing, and lifetimes.

3. GC Log Analyzers (Java)

- Helps analyze Java Garbage Collection (GC) performance.
- Detects memory leaks & excessive GC pauses.

CONCLUSION & Q&A

Summary Points:

Tradeoffs Between Control & Safety:

Manual Memory Management (C/C++) → More control but risk of memory leaks, dangling pointers.
Garbage Collection (Java, Python) → Safer but GC pauses can impact performance.
Rust's Ownership Model → Balances performance & safety with no GC.

Right Tool for the Job:

C/C++ → High-performance systems (OS, embedded, real-time).
Rust → Safe systems programming (memory safety without GC).
Java/Python → Application-level development (web, enterprise).