Lovely Professional University, Phagwara, Punjab

CS316: Operating Systems

REPORT:

EFFICIENT GARBAGE COLLECTION IN OPERATING SYSTEM

Submitted By: Chattanya Kumari, Priyanshu Raghuvanshi

Registration Numbers: 12326190, 12322741

Roll Numbers: 20,19

Section: K23HP

Submitted to: Amandeep Kaur

Abstract

Garbage collection (GC) is a critical component of modern programming languages, responsible for automatic memory management. Efficient garbage collection ensures optimal system performance, reduces memory leaks, and prevents application crashes.

This report explores various garbage collection techniques, highlighting their advantages and limitations. We discuss traditional methods such as Reference Counting and Mark-Sweep, as well as advanced approaches like Generational and Concurrent GC. The challenges of garbage collection, including performance overhead and memory fragmentation, are also examined. Finally, we review emerging trends and optimizations in GC to enhance software efficiency and reliability.

TABLE OF CONTENTS

INTRODUCTION

Garbage collection (GC) is an automatic memory management technique used in programming languages to reclaim memory that is no longer in use. It eliminates the need for manual memory deallocation, reducing the risk of memory leaks and segmentation faults.

Memory management is a fundamental aspect of operating system design. Without proper garbage collection mechanisms, applications

may suffer from memory leaks, leading to performance degradation, crashes, or system instability. Modern operating systems employ sophisticated garbage collection strategies to optimize resource usage and enhance overall efficiency. This report provides a comprehensive overview of garbage collection techniques, their impact on performance, and strategies for efficient implementation.

When a program runs, it dynamically allocates memory to store objects, variables, and data structures. However, once these objects are no longer required (i.e., there are no references pointing to them), they continue occupying memory unless explicitly freed. In languages without garbage collection (such as C and C++), developers must manually free up memory using functions like free() or delete. Failure to do so can result in **memory leaks**, where memory is allocated but never released, leading to excessive memory consumption and possible program crashes.

Overall, Garbage Collection plays a crucial role in modern software development by **simplifying memory management, improving program stability, and reducing the likelihood of memory-related errors**, allowing developers to focus on core application logic rather than worrying about manual memory handling.

WHY GARBAGE COLLECTION NEEDED?

Prevents Memory Leaks – Ensures that unused memory is freed, preventing excessive memory consumption and system slowdowns.

Avoids Manual Errors – Eliminates common memory management issues like **dangling pointers, double free errors, and unintentional memory leaks**.

Reduces Fragmentation – Frees scattered memory blocks, making memory allocation more efficient.

Improves System Performance – Enhances application speed by reducing memory allocation overhead.

Ensures Safe and Automatic Memory Management – Frees developers from manual memory handling, reducing errors.

Supports Multi-threading – Works efficiently in modern multi-core and multi-threaded environments.

How Garbage Collection Works in Programming Languages

Different programming languages implement garbage collection in various ways, but the core principle remains the same:

1.Memory Allocation – When a program creates objects or variables, memory is allocated dynamically from the **heap**.

2.Reference Tracking – The garbage collector continuously monitors objects in memory, keeping track of which ones are still in use.

3.Memory Reclamation – Once an object is no longer referenced, the garbage collector identifies it as **unused** and reclaims its memory, making it available for new allocations.

Languages like **Java, Python, and C#** use advanced garbage collection techniques to ensure **efficient memory management without affecting program performance**.

Types of Garbage Collection:

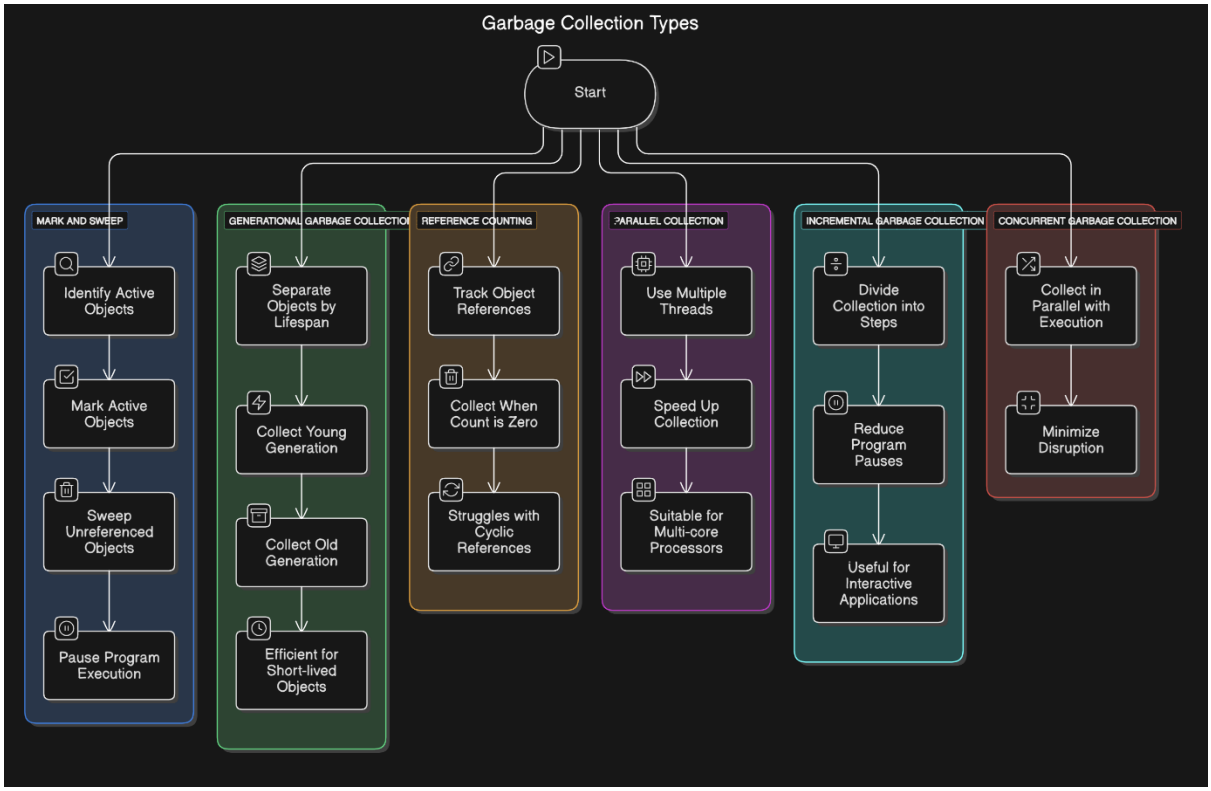


Figure 1 :- Garbage collection types

1.Reference Counting Garbage Collection

Overview: Reference counting GC tracks the number of references to an object. When an object's reference count drops to zero, it is deallocated.

Advantages:

- Simple and easy to implement.
- Immediate deallocation of unused objects.

Disadvantages:

- Cannot handle cyclic references (objects referring to each other).
- Increased performance overhead due to reference counting operations.

2.Mark and Sweep Garbage Collection

Overview:

Mark and Sweep consists of two phases: the "mark" phase, which identifies reachable objects, and the "sweep" phase, which deallocates unreferenced memory.

Advantages:

- Effectively handles cyclic references.
- No reference counting overhead.

Disadvantages:

- Causes program pauses during garbage collection.
- Can lead to memory fragmentation.

3. Mark-Compact Garbage Collection

Overview:
Mark-Compact GC is an extension of Mark and Sweep, where instead of leaving fragmented memory, it compacts objects to minimize fragmentation.

Advantages:
- Reduces memory fragmentation.
- Improves memory locality, enhancing performance.

Disadvantages:
- Requires additional CPU processing to compact objects.
- Can introduce performance overhead.

4. Copying Garbage Collection

Overview:
Copying GC divides memory into two halves: active and inactive. It copies all active objects to the inactive space and then clears the old memory.

Advantages:
- Eliminates memory fragmentation.
- Faster allocation due to contiguous memory usage.

Disadvantages:
- Wastes half of the memory space.
- Can be inefficient for large object graphs.

5. Generational Garbage Collection

Overview:
Generational GC categorizes objects into different generations: young, old, and permanent. The idea is that most objects die young, so the system optimizes garbage collection for short-lived objects.

Advantages:
- Optimized for real-world applications.
- Reduces pause times by focusing on young objects.

Disadvantages:
- More complex to implement.
- Still requires full GC cycles for long-lived objects.

6.Concurrent and Parallel Garbage Collection

Overview:
- **Concurrent GC** runs alongside application threads, reducing stop-the-world pauses.
- **Parallel GC** runs multiple GC threads simultaneously for better performance on multi-core systems.

Advantages:
- Reduces latency by allowing application execution during GC.
- Optimized for multi-core processors.

Disadvantages:
- Can increase CPU overhead due to concurrent execution.
- More complex to implement and tune.

7.Region-Based Garbage Collection

Overview:
Region-based GC divides memory into regions and collects garbage in specific regions instead of the whole heap.

Advantages:

- Reduces stop-the-world pauses by focusing on specific regions.
- More efficient than traditional GC techniques.

Disadvantages:

- Complex implementation.
- Requires tuning for optimal performance.

Efficient Garbage Collection Strategies

To improve garbage collection efficiency, modern operating systems implement various strategies:

1. **Adaptive GC Algorithms**: Dynamic tuning of garbage collection parameters based on workload and memory usage optimizes performance.

2. **Heap Compaction**: Reduces fragmentation by moving live objects closer, improving memory locality and allocation efficiency.

3. **Parallel Garbage Collection**: Uses multiple threads to perform garbage collection simultaneously, leveraging multi-core processors.

4. **Real-Time GC**: Ensures predictable response times by imposing strict time constraints on garbage collection cycles, essential for real-time systems.

5. **Lazy Collection**: Defers garbage collection until memory pressure reaches a threshold, reducing frequent interruptions.

6. **Garbage First (G1) GC**: Splits the heap into regions and prioritizes collecting regions with the most garbage, improving efficiency.

7. **Region-Based Memory Management**: Allocates memory in predefined regions, making garbage collection more predictable and efficient.

8. **Cycle Detection for Reference Counting**: Enhances reference counting by identifying and breaking cyclic references.

9. **Write Barriers and Remembered Sets**: Helps track modifications in objects to improve efficiency in generational and concurrent GC.

10. **Hybrid Approaches**: Combining multiple GC techniques to optimize performance based on application needs.

Garbage Collection in Modern Operating Systems

- **Linux**: Uses slab allocation and buddy memory allocation techniques to manage kernel memory efficiently. The Linux kernel employs various memory management techniques, including the Out-of-Memory (OOM) killer, to handle excessive memory usage effectively.

- **Windows**: Implements automatic garbage collection in the .NET runtime with generational and concurrent GC methods. The Windows operating system also uses paging and virtual memory techniques to manage memory effectively.

- **macOS**: Uses Automatic Reference Counting (ARC) for memory management in Objective-C and Swift. This approach balances

performance and memory efficiency, reducing the need for manual memory management.

Challenges in Garbage Collection

Efficient garbage collection poses several challenges:

- **Performance Overhead**: Frequent garbage collection cycles can introduce latency.
- **Memory Fragmentation**: Inefficient collection can lead to fragmented memory, reducing allocation efficiency.
- **Real-Time Constraints**: Hard real-time systems require predictable memory management without unpredictable delays.
- **Concurrency Issues**: Multi-threaded environments require careful synchronization to avoid race conditions.
- **High Memory Usage**: Some GC algorithms require additional memory overhead for tracking objects and managing heap space.

Future Trends in Garbage Collection

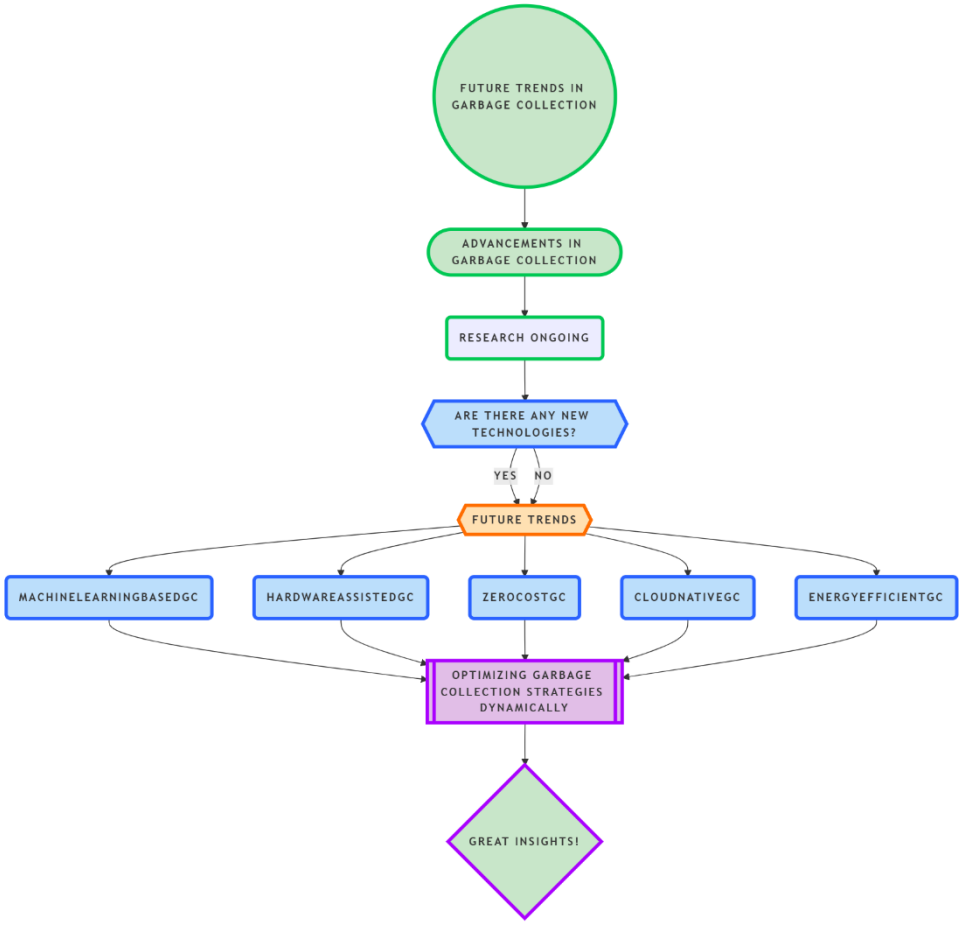


Figure 2: future trends in garbage collection

Advancements in garbage collection continue to improve efficiency. Future trends include:

- **Machine Learning-Based GC**: Using AI to optimize garbage collection strategies dynamically.

- **Hardware-Assisted GC**: Leveraging hardware features to accelerate memory management.

- **Zero-Cost Garbage Collection**: Research into minimizing GC overhead for high-performance applications.

- **Cloud-Native GC**: Optimizing memory management for distributed cloud computing environments.

- **Energy-Efficient GC**: Reducing power consumption in mobile and embedded devices.

Case Studies and Real-World Implementation

Several companies and technologies have implemented optimized garbage collection mechanism to improve system performance.

Figure 3:java virtual machine(jvm)

Figure 4: .NET (CLR)

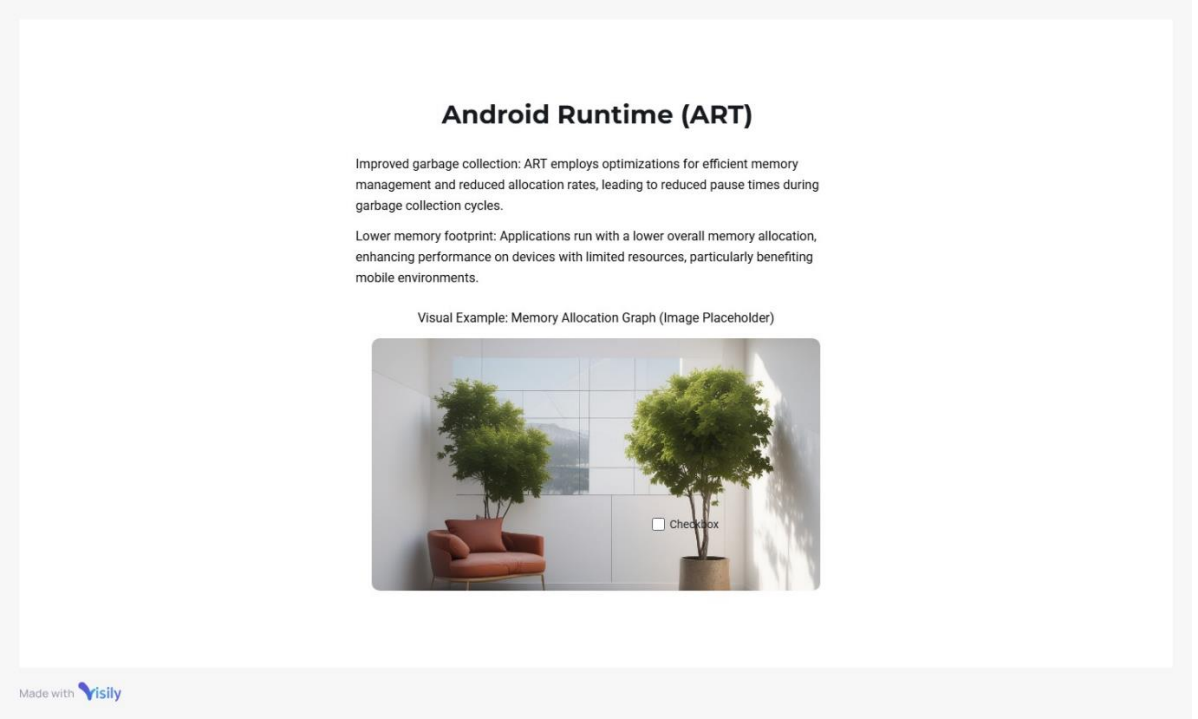


Figure 5:Android Runtime (Art)

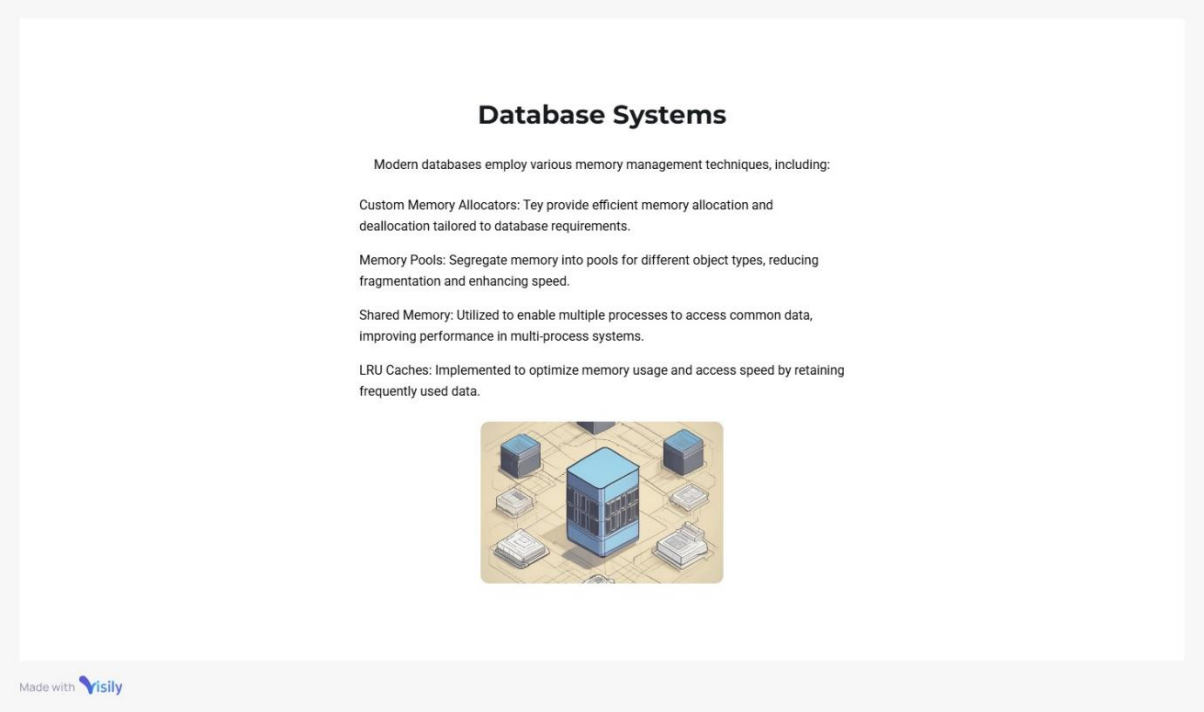


Figure 6: Database management

Best Practices for Efficient Garbage Collection

To enhance garbage collection performance, developers and system administrators can adopt best practices such as:

1. **Tuning GC Parameters**: Adjusting heap size, generation thresholds, and GC frequency based on application requirements.

2. **Optimizing Memory Allocation**: Reducing unnecessary object creation and reusing memory whenever possible.

3. **Monitoring GC Performance**: Using profiling tools to analyze GC behavior and optimize application performance.

4. **Avoiding Memory Leaks**: Ensuring proper deallocation of objects and avoiding circular references.

5. **Balancing Performance and Latency**: Choosing the appropriate GC algorithm based on application needs

Key Actors & Their Roles

1. Operating System (OS Kernel)

The **OS Kernel** is the core component of the operating system that manages system resources, including memory. It **triggers the garbage collection process** when necessary, ensuring efficient memory usage.

- **Use Cases for OS Kernel**

Monitor Memory Usage

- Tracks system memory usage and detects when garbage collection is needed.

Trigger Garbage Collection

- Initiates the garbage collection process when memory usage reaches a critical threshold or after a specific time interval.

Coordinate with Memory Manager

- Informs the memory manager when to allocate or free memory efficiently.

Ensure System Performance

- Prevents memory leaks and ensures that memory-intensive applications run smoothly.

2 .Garbage Collector (GC)

The **Garbage Collector (GC)** is responsible for **automatically managing memory** by reclaiming unused memory and optimizing memory allocation. It runs in the background without user intervention.

◇ Use Cases for Garbage Collector

Mark Phase (Identify Reachable Memory)

- Scans memory and marks all objects that are still in use (reachable objects).

Sweep Phase (Remove Unreachable Memory)

- Deletes objects that are no longer referenced by the program.

Compact Phase (Optimize Memory Layout)

- After deletion, it reorganizes the remaining objects to **reduce memory fragmentation** and improve performance.

Work with OS Kernel

- Listens for garbage collection triggers from the OS and executes the cleanup process.

Detect Memory Leaks

- Identifies objects that are no longer needed and removes them before they consume excess memory.

Enhance Multitasking

- Ensures efficient memory management in multi-threaded environments.

3. Memory Manager

The **Memory Manager** is a subsystem within the OS responsible for **allocating and deallocating memory** dynamically.

◇ Use Cases for Memory Manager

Allocate Memory to Processes

- Assigns memory blocks to running processes based on demand.

Track Memory Usage

- Monitors how much memory is allocated to different processes and prevents fragmentation.

Free Memory After GC Runs

- Works with the garbage collector to deallocate freed memory for future use.

Defragment Memory

- Rearranges memory blocks after garbage collection to ensure efficient memory access.

Optimize Paging and Virtual Memory

- Helps in swapping pages between RAM and disk to keep the system responsive.

◇ **Summary of the Process**

◇ The **OS Kernel** monitors system memory and **triggers garbage collection** when needed.

◇ The **Garbage Collector (GC) marks, sweeps, and compacts memory** to remove unused objects.

The **Memory Manager allocates, tracks, and optimizes** memory usage, working closely with the GC.

Code for implementation of Efficient in garbage collection in python

```
import gc

import random

import psutil

import time

import matplotlib.pyplot as plt

import seaborn as sns
```

```
from collections import deque
memory_queue = deque(maxlen=5)
def allocate_memory(num_objects=10000):
    objects = []
    for _ in range(num_objects):
        objects.append([random.random() for _ in range(100)])  # Creating random lists
    return objects
def get_memory_usage():
    return psutil.Process().memory_info().rss / (1024 * 1024)  # Convert bytes to MB
def monitor_gc():
    memory_before = get_memory_usage()
    objects = allocate_memory()
    memory_allocated = get_memory_usage()
    del objects  # Remove references
    gc.collect()  # Force garbage collection
    memory_after_gc = get_memory_usage()
     memory_queue.append((memory_before, memory_allocated, memory_after_gc))
for _ in range(5):
    monitor_gc()
    time.sleep(1)
memory_before = [entry[0] for entry in memory_queue]
memory_allocated = [entry[1] for entry in memory_queue]
memory_after_gc = [entry[2] for entry in memory_queue]
iterations = list(range(1, len(memory_queue) + 1))
sns.set_style("whitegrid")
sns.set_palette("coolwarm")
plt.figure(figsize=(10, 6))
sns.lineplot(x=iterations, y=memory_before, label="Before Allocation", marker='o',
linestyle='dashed', linewidth=2)
```

sns.lineplot(x=iterations, y=memory_allocated, label="After Allocation", marker='s', linestyle='dotted', linewidth=2)

sns.lineplot(x=iterations, y=memory_after_gc, label="After GC", marker='^', linestyle='solid', linewidth=2)

plt.xlabel("Iteration", fontsize=12, fontweight='bold')

plt.ylabel("Memory Usage (MB)", fontsize=12, fontweight='bold')

plt.title("Efficient Garbage Collection in OS (FIFO-based Tracking)", fontsize=14, fontweight='bold')

plt.legend(fontsize=10, fancybox=True, shadow=True, loc="upper left")

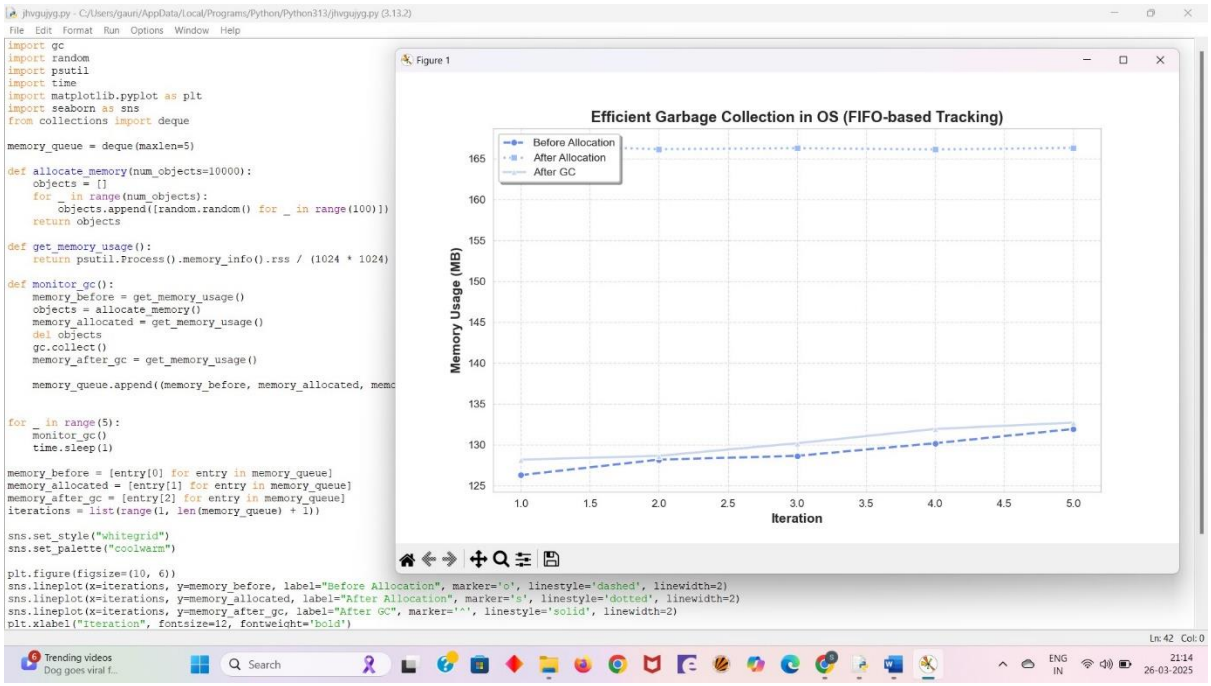
plt.grid(True, linestyle='--', alpha=0.6)

plt.xticks(fontsize=10)

plt.yticks(fontsize=10)

plt.show()

output of Implementation



Conclusion

Efficient garbage collection is vital for system stability and performance. By employing advanced algorithms, adaptive strategies, and multi-threaded execution, modern operating systems ensure that memory is managed effectively with minimal overhead. Continuous research and optimization in garbage collection techniques further enhance efficiency in real-world applications. As technology evolves, garbage collection strategies will continue to improve, providing better memory management for future computing environments.

This report has expanded on the key concepts of garbage collection, addressing its challenges, current implementations, and future directions. By implementing efficient garbage collection techniques, operating systems can ensure optimal memory utilization, reducing latency and improving application performance. The advancements in memory management are crucial for developing efficient and scalable computing systems.

REFERENCE

1.**Microsoft Docs.(n.d.). .NET Garbage Collection**-Fundamentals of garbage collection https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

2.**Study .com.(n.d.).** Garbage Collection in Operating Systems: Definition & Strategies

https://study.com/academy/lesson/garbage-collection-in-operating-systems-definition-strategies.html

3. **Oracle. (n.d.).** Factors Affecting Garbage Collection Performance https://docs.oracle.com/en/java/javase/11/gctuning/factors-affecting-garbage-collection-performance.html

4. **SpringerLink. (n.d.).** Generational Real-Time Garbage Collection

https://link.springer.com/chapter/10.1007/978-3-540-73589-2_6

5.**Unity Documentation.(n.d.).** Garbage collection best practices https://docs.unity3d.com/2019.4/Documentation/Manual/performance-garbage-collection-best-practices.html

6.**Wikipedia.(n.d.).** Garbage collection - https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

7.**ResearchGate. (n.d.).**A study of concurrent real-time garbage collectors

https://www.researchgate.net/publication/220752316_A_study_of_concurrent_real-time_garbage_collectors