Project Title:

Efficient Garbage Collection in OS

SUBMITED BY : ALEN ALEX(15), PRACHURJYA PRAN(16), TIRUMALA SRI(17) of K23AL

Introduction

Memory management is a crucial aspect of modern operating systems (OS). Efficient memory utilization ensures smooth performance and resource optimization. One of the primary mechanisms employed for this purpose is **garbage collection (GC)**. This project focuses on designing an optimized garbage collection mechanism for memory management in modern OS environments.

Objectives

The primary objectives of this project include:

- Developing a garbage collection system to manage memory efficiently.
- Implementing algorithms like **Mark and Sweep** to identify and reclaim unused memory.
- Minimizing memory fragmentation and improving memory allocation.
- Providing real-time memory usage analysis through visualization.
- Evaluating the performance of the proposed GC system through benchmarking.

Problem Statement

Traditional garbage collection mechanisms often lead to memory fragmentation and high CPU usage. Inefficient GC algorithms may also introduce latency in application performance. The proposed project aims to mitigate these issues by developing an advanced and efficient garbage collection mechanism that minimizes memory fragmentation and reduces processing time.

Expected Outcomes

- An optimized garbage collection mechanism with improved memory management.
- Reduced memory fragmentation and faster memory allocation.
- Real-time visualization of memory usage and garbage collection events.
- Detailed performance analysis reports comparing different GC algorithms.

Scope of the Project

- The project focuses on **user-space memory management** rather than kernel-level management.

- Implementation of algorithms like **Mark and Sweep** and additional memory optimization techniques.
- Development of a **monitoring module** to visualize memory usage in real-time.
- Performance benchmarking through simulated workloads.

Module-Wise Breakdown

The project is divided into three primary modules, each focusing on distinct aspects of the garbage collection mechanism:

1. Core Garbage Collection Module

- Implements the primary garbage collection algorithm (e.g., Mark and Sweep).
- Manages memory allocation and deallocation.
- Detects and collects unused memory objects.
- Prevents memory fragmentation using compaction or other optimization techniques.

2. Monitoring and Analysis Module

- Tracks real-time memory usage and garbage collection activity.
- Visualizes memory allocation, deallocation, and fragmentation.
- Provides detailed reports and insights into memory performance.
- Allows users to compare different garbage collection algorithms.

3. Simulation and Test Environment Module

- Simulates memory management scenarios using test cases.
- Evaluates the effectiveness of the garbage collection mechanism.
- Conducts stress tests with varied workloads to assess performance.
- Generates reports with performance metrics, including memory usage, execution time, and fragmentation levels.

Each module will work in coordination to ensure an efficient and effective garbage collection system.

Functionalities

1. Core Garbage Collection Module

- **Memory Allocation and Deallocation:** Efficiently manages memory allocation requests and deallocates memory when objects are no longer in use.

- **Garbage Identification:** Implements algorithms like **Mark and Sweep** to identify unused objects.
- **Fragmentation Management:** Prevents memory fragmentation by using memory compaction techniques.
- **Performance Optimization:** Ensures low overhead during GC cycles to minimize application performance impact.

2. Monitoring and Analysis Module

- **Real-Time Monitoring:** Tracks memory usage, allocation, and deallocation events.
- **Visualization:** Provides graphical representation of memory state and GC performance.
- **Performance Reporting:** Generates detailed reports on memory utilization and GC efficiency.
- **Comparative Analysis:** Allows users to compare different GC algorithms and their impact on memory management.

3. Simulation and Test Environment Module

- **Scenario Simulation:** Simulates various memory usage scenarios for testing GC algorithms.
- **Benchmarking:** Evaluates memory management efficiency based on metrics like memory fragmentation, GC time, and application latency.
- **Stress Testing:** Simulates high memory demand situations to test the robustness of the GC system.
- **Data Collection:** Gathers and logs performance data for analysis and reporting.

Technology Used

Programming Languages:

- Python

Libraries and Tools:

- Matplotlib for data visualization
- NumPy for efficient memory management simulation
- Random module for workload generation

Other Tools:

- GitHub for version control

- Jupyter Notebook for development and testing
- Visual Studio Code (VS Code) as the IDE

Flow Diagram

Below is a simplified flow diagram representing the garbage collection process using the **Mark and Sweep** algorithm:

1. **Start** → Initialization of memory management.
2. **Object Creation** → Allocate memory for new objects.
3. **Mark Phase** → Traverse through the object graph to mark reachable objects.
4. **Sweep Phase** → Identify unmarked objects (garbage) and reclaim memory.
5. **Compaction (Optional)** → Rearrange memory to reduce fragmentation.
6. **Monitoring & Visualization** → Track memory usage and display insights.
7. **Performance Evaluation** → Generate reports based on memory metrics.
8. **End**

Revision Tracking on GitHub

- **Repository Name:** OSK23AL
- **GitHub Link:** https://github.com/alensomaxx/OSK23AL

Conclusion and Future Scope

In conclusion, this project offers a robust solution to memory management challenges in modern operating systems through an optimized garbage collection mechanism. By implementing algorithms like **Mark and Sweep**, monitoring memory usage in real-time, and performing in-depth analysis, this project ensures improved memory utilization and reduced fragmentation.

Future Scope:

- Enhance the garbage collection mechanism by incorporating additional algorithms like **Generational GC** or **Reference Counting**.
- Implement predictive memory management using machine learning to forecast memory requirements.
- Expand the visualization module with interactive and detailed insights.
- Conduct further stress testing under large-scale environments to refine the algorithm's efficiency.

- Explore kernel-level integration for deeper memory management optimizations.

References

- AI-Generated Project Elaboration/Breakdown Report
- **Problem Statement:** ***Efficient Garbage Collection in OS*** **-** Design an optimized garbage collection mechanism for memory management in modern OS environments
- **Solution/ Code:**

```
import tkinter as tk
from tkinter import ttk, filedialog
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import numpy as np
import random
import time
import csv

# Constants
MEMORY_SIZE = 1000
OBJECT_RANGE = (20, 100)
ALLOCATION_CYCLES = 30
DELAY = 500  # milliseconds

class MemoryBlock:
    def __init__(self, size, id):
        self.size = size
        self.id = id
        self.marked = False
        self.start_index = None

class RefCountedObject:
    def __init__(self, size, id):
        self.size = size
        self.id = id
        self.ref_count = random.randint(0, 2)
        self.start_index = None

class MemoryManager:
    def __init__(self, total_memory):
        self.total_memory = total_memory
        self.memory = []
        self.allocated_memory = []
        self.memory_usage = []
        self.memory_layout = [-1] * self.total_memory
        self.benchmark_times = []

    def allocate(self):
```

```
            size = random.randint(*OBJECT_RANGE)
            start = self.find_space(size)
            if start is not None:
                obj = MemoryBlock(size, len(self.allocated_memory))
                obj.start_index = start
                self.memory.append(obj)
                self.allocated_memory.append(obj)
                for i in range(start, start + size):
                    self.memory_layout[i] = obj.id
                return obj
            return None

        def find_space(self, size):
            count = 0
            start = 0
            for i in range(self.total_memory):
                if self.memory_layout[i] == -1:
                    if count == 0:
                        start = i
                    count += 1
                    if count == size:
                        return start
                else:
                    count = 0
            return None

        def get_used_memory(self):
            return sum(obj.size for obj in self.memory)

        def mark(self):
            for obj in self.memory:
                obj.marked = random.choice([True, False])

        def sweep(self):
            before = len(self.memory)
            new_memory = []
            self.memory_layout = [-1] * self.total_memory
            for obj in self.memory:
                if obj.marked:
                    new_memory.append(obj)
            self.memory = new_memory
            for obj in self.memory:
                start = self.find_space(obj.size)
                if start is not None:
                    obj.start_index = start
                    for i in range(start, start + obj.size):
                        self.memory_layout[i] = obj.id
            return before - len(self.memory)
```

```
    def compact(self):
        self.memory.sort(key=lambda x: x.id)
        self.memory_layout = [-1] * self.total_memory
        current = 0
        for obj in self.memory:
            obj.start_index = current
            for i in range(current, current + obj.size):
                self.memory_layout[i] = obj.id
            current += obj.size

    def simulate_cycle(self):
        start_time = time.time()
        self.allocate()
        self.mark()
        collected = self.sweep()
        self.compact()
        used = self.get_used_memory()
        self.memory_usage.append(used)
        end_time = time.time()
        self.benchmark_times.append(end_time - start_time)
        return collected, used, self.memory_layout.copy()

class GCManagerWithReferenceCounting(MemoryManager):
    def allocate(self):
        size = random.randint(*OBJECT_RANGE)
        start = self.find_space(size)
        if start is not None:
            obj = RefCountedObject(size, len(self.allocated_memory))
            obj.start_index = start
            self.memory.append(obj)
            self.allocated_memory.append(obj)
            for i in range(start, start + size):
                self.memory_layout[i] = obj.id
            return obj
        return None

    def simulate_cycle(self):
        start_time = time.time()
        self.allocate()
        collected = self.collect_garbage()
        self.compact()
        used = self.get_used_memory()
        self.memory_usage.append(used)
        end_time = time.time()
        self.benchmark_times.append(end_time - start_time)
        return collected, used, self.memory_layout.copy()

```

```
    def collect_garbage(self):
        before = len(self.memory)
        new_memory = []
        self.memory_layout = [-1] * self.total_memory
        for obj in self.memory:
            if obj.ref_count > 0:
                new_memory.append(obj)
        self.memory = new_memory
        for obj in self.memory:
            start = self.find_space(obj.size)
            if start is not None:
                obj.start_index = start
                for i in range(start, start + obj.size):
                    self.memory_layout[i] = obj.id
        return before - len(self.memory)

class GCVisualizer(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Efficient Garbage Collector Simulator")
        self.geometry("1000x800")
        self.cycle = 0
        self.manager = None
        self.selected_algo = tk.StringVar(value="Mark and Sweep")
        self.start_time = None
        self.setup_ui()
        self.after_id = None

    def setup_ui(self):
        control_frame = ttk.Frame(self)
        control_frame.pack(pady=10)

        ttk.Label(control_frame, text="GC Algorithm:").grid(row=0,
column=0, padx=5)
        algo_dropdown = ttk.Combobox(control_frame,
textvariable=self.selected_algo, values=["Mark and Sweep", "Reference
Counting"], state="readonly")
        algo_dropdown.grid(row=0, column=1, padx=5)

        self.start_button = ttk.Button(control_frame, text="Start
Simulation", command=self.start_simulation)
        self.start_button.grid(row=0, column=2, padx=10)

        self.export_button = ttk.Button(control_frame, text="Export
Logs to CSV", command=self.export_logs)
        self.export_button.grid(row=0, column=3, padx=10)

```

```
        self.status_label = ttk.Label(control_frame, text="Status: Ready")
        self.status_label.grid(row=1, column=0, columnspan=4, pady=5)

        self.benchmark_label = ttk.Label(control_frame, text="Benchmark: -")
        self.benchmark_label.grid(row=2, column=0, columnspan=4)

        self.fig, (self.ax1, self.ax2) = plt.subplots(2, 1, figsize=(10, 6))
        self.fig.tight_layout(pad=3.0)

        self.line, = self.ax1.plot([], [], marker='o', label='Used Memory')
        self.ax1.set_title('Memory Usage Over Time')
        self.ax1.set_xlabel('Cycle')
        self.ax1.set_ylabel('Used Memory')
        self.ax1.set_ylim(0, MEMORY_SIZE)
        self.ax1.legend()
        self.ax1.grid(True)

        self.fragment_display = self.ax2.imshow(np.zeros((1, MEMORY_SIZE)), aspect='auto', cmap='tab20', interpolation='nearest', vmin=-1, vmax=20)
        self.ax2.set_title('Memory Fragmentation')
        self.ax2.set_yticks([])

        self.canvas = FigureCanvasTkAgg(self.fig, master=self)
        self.canvas.get_tk_widget().pack()

    def start_simulation(self):
        self.cycle = 0
        algo = self.selected_algo.get()
        if algo == "Reference Counting":
            self.manager = GCManagerWithReferenceCounting(MEMORY_SIZE)
        else:
            self.manager = MemoryManager(MEMORY_SIZE)

        self.start_time = time.time()
        self.run_cycle()

    def run_cycle(self):
        if self.cycle < ALLOCATION_CYCLES:
            collected, used, layout = self.manager.simulate_cycle()
            self.line.set_xdata(range(len(self.manager.memory_usage)))
            self.line.set_ydata(self.manager.memory_usage)
            self.ax1.set_xlim(0, ALLOCATION_CYCLES)

```

```
            visual_array = np.array(layout).reshape((1, -1))
            self.fragment_display.set_data(visual_array)

            avg_time = np.mean(self.manager.benchmark_times)
            self.status_label.config(text=f"Cycle {self.cycle+1}:
Used={used}, Collected={collected}")
            self.benchmark_label.config(text=f"Avg GC Time:
{avg_time:.4f} sec")

            self.canvas.draw()
            self.cycle += 1
            self.after_id = self.after(DELAY, self.run_cycle)
        else:
            total_time = time.time() - self.start_time
            self.status_label.config(text="Simulation Completed")
            self.benchmark_label.config(text=f"Total Time:
{total_time:.2f} sec | Avg GC Time:
{np.mean(self.manager.benchmark_times):.4f} sec")

    def export_logs(self):
        if not self.manager or not self.manager.memory_usage:
            self.status_label.config(text="No data to export yet!")
            return

        file_path =
filedialog.asksaveasfilename(defaultextension=".csv", filetypes=[("CSV
files", "*.csv")])
        if not file_path:
            return

        with open(file_path, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(["Cycle", "Used Memory", "GC Time (sec)"])
            for i, (mem, t) in enumerate(zip(self.manager.memory_usage,
self.manager.benchmark_times), start=1):
                writer.writerow([i, mem, round(t, 6)])

        self.status_label.config(text=f"Logs exported to: {file_path}")

# Run the final GUI
if __name__ == "__main__":
    app = GCVisualizer()
    app.mainloop()
```