**Implementing a Dynamic Memory Allocation Simulation**

**Introduction:**

This project aims to demonstrate how operating systems manage memory dynamically using different memory allocation strategies. We implemented a simulator in Python using **CustomTkinter**, with a focus on the **First Fit** algorithm. The application supports memory allocation, deallocation (based on process burst time), and fragmentation reporting.

**Overview of Memory Allocation Algorithms:**

| **Algorithm** | **Description** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **First Fit** | Allocates the first memory block that is large enough | Fast, simple implementation | Can lead to fragmentation over time |
| **Best Fit** | Allocates the smallest block that fits the process | Reduces memory waste initially | Slower, increases external fragmentation |
| **Worst Fit** | Allocates the largest available block | May leave large free blocks later | High fragmentation risk, inefficient |
| **Next Fit** | Like First Fit, but continues search from last allocated block | Slightly better performance than First Fit in some cases | Still prone to fragmentation |

**Justification for Choosing First Fit**:

* Offers **efficient performance** with minimal search time.
* Suitable for a basic simulation and easy to debug.
* Helps visualize fragmentation handling effectively with merging.

**Initial Stage Memory Blocks:**

| **Block Size** | **Status:** |
| --- | --- |
| 2KB | Free |
| 120KB | Occupied (P1) |
| 20KB | Free |
| 150KB | Occupied (P2) |
| 160KB | Free |
| 1KB | Occupied (P3) |
| 4KB | Free |
| 554KB | Occupied (P4) |
| 124KB | Free |

**Design And Implementation:**

The memory allocation simulator is designed using an object-oriented approach with two core classes: MemoryBlock and Process. It implements the **First Fit allocation algorithm**, which allocates the first suitable free block for each process. If the block is larger than needed, it is split; if exact, it is directly assigned.

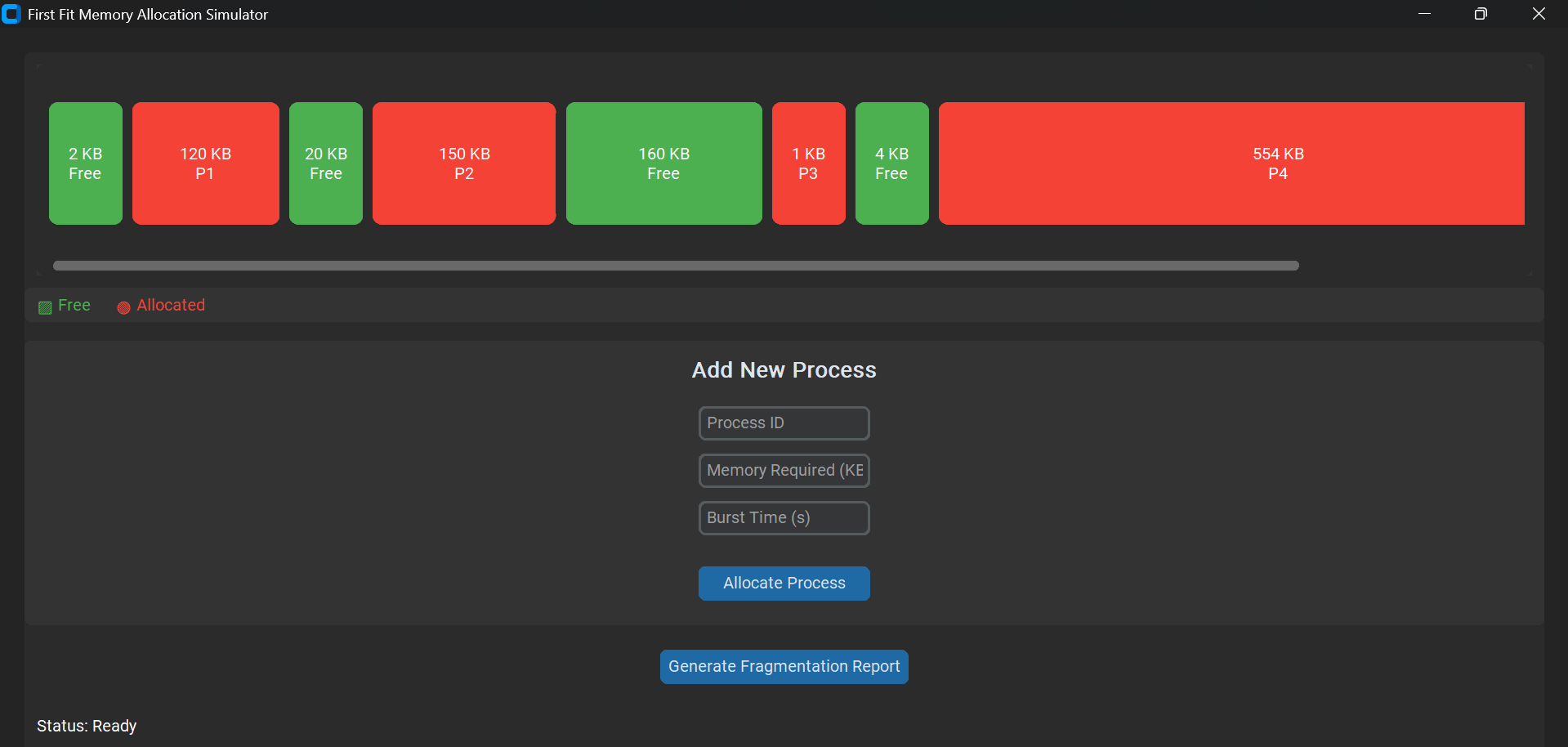
Memory is automatically deallocated after the process’s burst time using **Python threading**, simulating real-time execution. Adjacent free blocks are **merged** post-deallocation to minimize fragmentation.

A modern **GUI built with CustomTkinter** visualizes memory blocks and allocation status in real-time. The simulator also includes a **fragmentation report generator**, giving insights into memory usage and external fragmentation. The overall design emphasizes modularity, interactivity, and educational clarity.

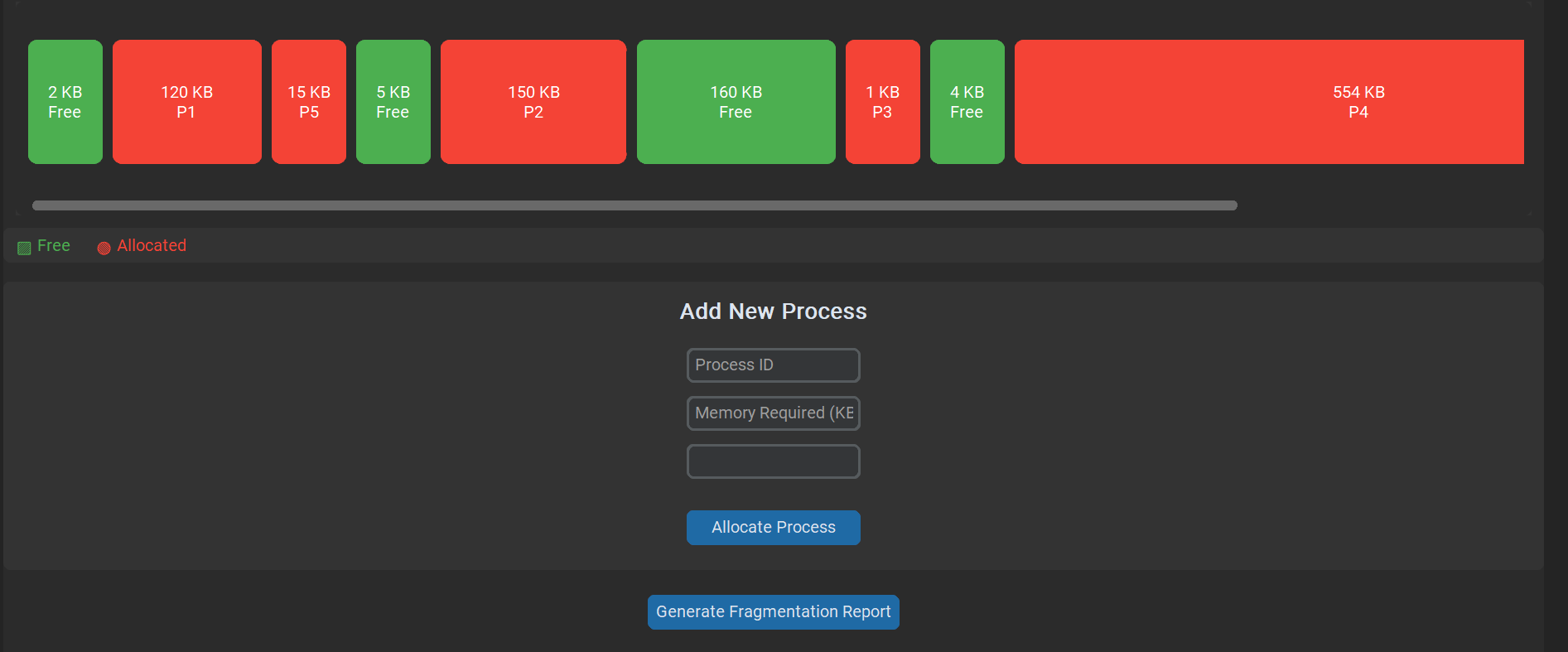
# **Source Code:**

|  |
| --- |
| import customtkinter as ctk  import threading  import time  from tkinter import messagebox  # Define memory block structure  class MemoryBlock:  def \_\_init\_\_(self, size, is\_free=True, process\_id=None):  self.size = size  self.is\_free = is\_free  self.process\_id = process\_id  # Define process structure  class Process:  def \_\_init\_\_(self, id, memory\_required, burst\_time):  self.id = id  self.memory\_required = memory\_required  self.burst\_time = burst\_time  # Simulator class  class MemorySimulatorApp:  def \_\_init\_\_(self, root):  self.root = root  self.root.title("First Fit Memory Allocation Simulator")  self.root.geometry("1100x700")  self.root.resizable(True, True)  ctk.set\_appearance\_mode("System")  ctk.set\_default\_color\_theme("blue")  self.memory\_blocks = [  MemoryBlock(2),  MemoryBlock(120, is\_free=False, process\_id="P1"),  MemoryBlock(20),  MemoryBlock(150, is\_free=False, process\_id="P2"),  MemoryBlock(160),  MemoryBlock(1, is\_free=False, process\_id="P3"),  MemoryBlock(4),  MemoryBlock(554, is\_free=False, process\_id="P4"),  MemoryBlock(124),  ]  self.main\_frame = ctk.CTkFrame(self.root)  self.main\_frame.pack(expand=True, fill="both", padx=20, pady=20)  self.memory\_frame\_container = ctk.CTkFrame(self.main\_frame)  self.memory\_frame\_container.pack(side="top", fill="x", padx=10, pady=(10, 0))  self.scrollable\_frame = ctk.CTkScrollableFrame(  self.memory\_frame\_container,  orientation="horizontal",  height=150  )  self.scrollable\_frame.pack(fill="x", expand=True)  self.legend\_frame = ctk.CTkFrame(self.main\_frame)  self.legend\_frame.pack(fill="x", pady=(10, 5))  ctk.CTkLabel(self.legend\_frame, text="🟩 Free", text\_color="#4CAF50").pack(side="left", padx=10)  ctk.CTkLabel(self.legend\_frame, text="🔴 Allocated", text\_color="#F44336").pack(side="left", padx=10)  input\_frame = ctk.CTkFrame(self.main\_frame, width=300)  input\_frame.pack(side="top", fill="x", pady=10)  ctk.CTkLabel(input\_frame, text="Add New Process", font=ctk.CTkFont(size=18, weight="bold")).pack(pady=(10, 10))  self.id\_entry = ctk.CTkEntry(input\_frame, placeholder\_text="Process ID")  self.id\_entry.pack(pady=5, padx=20)  self.size\_entry = ctk.CTkEntry(input\_frame, placeholder\_text="Memory Required (KB)")  self.size\_entry.pack(pady=5, padx=20)  self.burst\_entry = ctk.CTkEntry(input\_frame, placeholder\_text="Burst Time (s)")  self.burst\_entry.pack(pady=5, padx=20)  ctk.CTkButton(input\_frame, text="Allocate Process", command=self.handle\_submit).pack(pady=20)  ctk.CTkButton(self.main\_frame, text="Generate Fragmentation Report", command=self.generate\_fragmentation\_report).pack(pady=10)  self.status\_label = ctk.CTkLabel(self.main\_frame, text="Status: Ready", text\_color="white", anchor="w")  self.status\_label.pack(fill="x", pady=(10, 5), padx=10)  self.render\_memory()  def render\_memory(self):  for widget in self.scrollable\_frame.winfo\_children():  widget.destroy()  for block in self.memory\_blocks:  color = "#4CAF50" if block.is\_free else "#F44336"  label\_text = f"{block.size} KB\n{'Free' if block.is\_free else block.process\_id}"  lbl = ctk.CTkLabel(  self.scrollable\_frame,  text=label\_text,  fg\_color=color,  text\_color="white",  corner\_radius=8,  height=100,  width=max(60, block.size),  anchor="center"  )  lbl.pack(side="left", padx=4, pady=10)  def handle\_submit(self):  try:  pid = self.id\_entry.get().strip()  memory\_required = int(self.size\_entry.get().strip())  burst\_time = int(self.burst\_entry.get().strip())  if pid and memory\_required > 0 and burst\_time > 0:  process = Process(pid, memory\_required, burst\_time)  self.allocate\_memory(process)  self.id\_entry.delete(0, 'end')  self.size\_entry.delete(0, 'end')  self.burst\_entry.delete(0, 'end')  except ValueError:  messagebox.showerror("Error", "Invalid input. Enter valid numbers.")  def allocate\_memory(self, process):  for i, block in enumerate(self.memory\_blocks):  if block.is\_free and block.size >= process.memory\_required:  if block.size > process.memory\_required:  remaining = block.size - process.memory\_required  self.memory\_blocks[i] = MemoryBlock(process.memory\_required, is\_free=False, process\_id=process.id)  self.memory\_blocks.insert(i + 1, MemoryBlock(remaining))  else:  block.is\_free = False  block.process\_id = process.id  self.render\_memory()  self.update\_status(f"✅ Allocated Process {process.id} ({process.memory\_required}KB)")  threading.Thread(target=self.\_deallocate\_after\_burst, args=(process,), daemon=True).start()  return  self.update\_status(f"❌ Allocation failed for Process {process.id}")  messagebox.showwarning("Allocation Failed", f"⚠ Process {process.id} could not be allocated.")  def \_deallocate\_after\_burst(self, process):  time.sleep(process.burst\_time)  self.root.after(0, lambda: self.\_free\_and\_render(process))  def \_free\_and\_render(self, process):  for block in self.memory\_blocks:  if block.process\_id == process.id:  block.is\_free = True  block.process\_id = None  break  self.merge\_free\_blocks()  self.render\_memory()  self.update\_status(f"ℹ Process {process.id} completed and memory deallocated.")  def merge\_free\_blocks(self):  i = 0  while i < len(self.memory\_blocks) - 1:  if self.memory\_blocks[i].is\_free and self.memory\_blocks[i+1].is\_free:  self.memory\_blocks[i].size += self.memory\_blocks[i+1].size  del self.memory\_blocks[i+1]  else:  i += 1  def update\_status(self, message):  self.status\_label.configure(text=f"Status: {message}")  def generate\_fragmentation\_report(self):  free\_blocks = [block for block in self.memory\_blocks if block.is\_free]  total\_free = sum(block.size for block in free\_blocks)  total\_used = sum(block.size for block in self.memory\_blocks if not block.is\_free)  total\_size = sum(block.size for block in self.memory\_blocks)  largest\_free\_block = max((block.size for block in free\_blocks), default=0)  external\_fragmentation = total\_free - largest\_free\_block if len(free\_blocks) > 1 else 0  external\_fragmentation\_percent = (external\_fragmentation / total\_size) \* 100 if total\_size > 0 else 0  internal\_fragmentation = 0  internal\_fragmentation\_percent = 0  report\_window = ctk.CTkToplevel(self.root)  report\_window.title("Fragmentation Report")  report\_window.geometry("400x300")  report\_window.resizable(False, False)  report\_window.grab\_set()  report\_window.focus()  report\_window.configure(fg\_color="white")  report\_frame = ctk.CTkFrame(report\_window, fg\_color="white")  report\_frame.pack(fill="both", expand=True, padx=20, pady=20)  ctk.CTkLabel(  report\_frame,  text="FRAGMENTATION REPORT",  font=ctk.CTkFont(size=14, weight="bold"),  text\_color="black",  anchor="w",  justify="left"  ).pack(fill="x", pady=(0, 10))  ctk.CTkLabel(report\_frame, text=f"Total memory: {total\_size} KB", font=ctk.CTkFont(size=12), text\_color="black").pack(fill="x", pady=2)  ctk.CTkLabel(report\_frame, text=f"Memory used: {total\_used} KB", font=ctk.CTkFont(size=12), text\_color="black").pack(fill="x", pady=2)  ctk.CTkLabel(report\_frame, text=f"Memory free: {total\_free} KB", font=ctk.CTkFont(size=12), text\_color="black").pack(fill="x", pady=2)  ctk.CTkLabel(report\_frame, text=f"Number of free blocks: {len(free\_blocks)}", font=ctk.CTkFont(size=12), text\_color="black").pack(fill="x", pady=2)  ctk.CTkLabel(report\_frame, text=f"Largest free block: {largest\_free\_block} KB", font=ctk.CTkFont(size=12), text\_color="black").pack(fill="x", pady=2)  ctk.CTkLabel(report\_frame, text=f"External fragmentation: {external\_fragmentation} KB ({external\_fragmentation\_percent:.2f}%)", font=ctk.CTkFont(size=12), text\_color="black").pack(fill="x", pady=2)  ctk.CTkLabel(report\_frame, text=f"Internal fragmentation: {internal\_fragmentation} KB ({internal\_fragmentation\_percent:.2f}%)", font=ctk.CTkFont(size=12), text\_color="black").pack(fill="x", pady=2)  ctk.CTkLabel(report\_frame, text="Note: Internal fragmentation is 0 because blocks are split to fit processes exactly.", font=ctk.CTkFont(size=10), text\_color="black", wraplength=350).pack(fill="x", pady=2)  ctk.CTkButton(report\_frame, text="Close", command=report\_window.destroy).pack(pady=10)  if \_\_name\_\_ == "\_\_main\_\_":  root = ctk.CTk()  app = MemorySimulatorApp(root)  root.mainloop() |

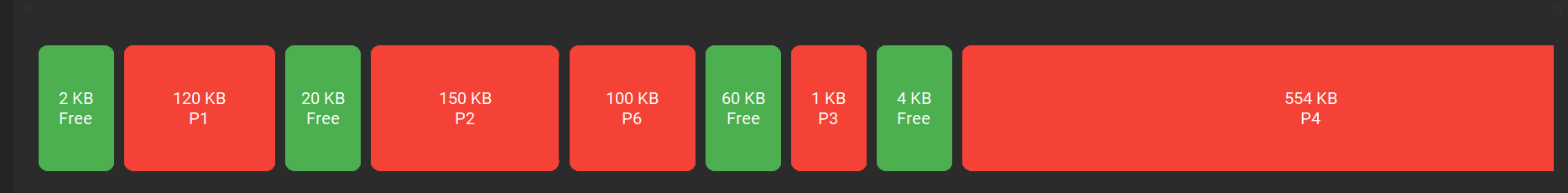
**Graphical User Interface:**

****

**P5 Allocation Request of 15KB:**

****

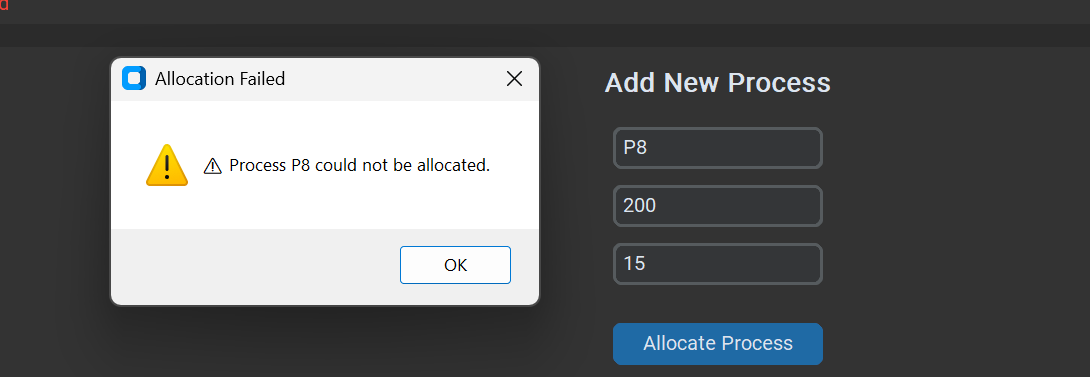
**P6 Allocation Request of 100KB:**

****

**P7 Allocation Request of 5KB:**

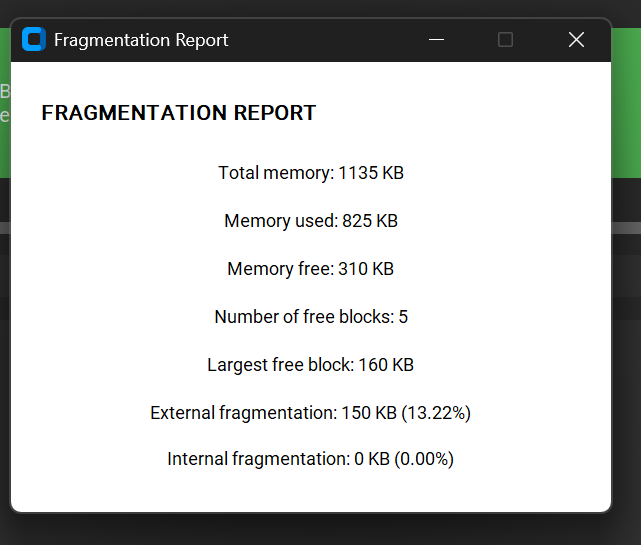
****

**P8 Allocation Request of 200KB:**

****

**P9 Allocation Request of 3KB:** ****

**Fragmentation Report:**



**Analysis And Discussion:**

**1. Internal Fragmentation**

No internal fragmentation occurs. This is because the simulator splits free blocks exactly to the size required by the process. Any remaining space after allocation becomes a new free block.

**Result:**

* internal\_fragmentation = 0 KB
* internal\_fragmentation\_percent = 0%

**2. External Fragmentation**

Free blocks are scattered throughout the memory. Even if total free memory is large, a process may fail to allocate if no **single block** is large enough.

Total free memory = 310 KB and largest free block = 160 KB, then:

external\_fragmentation = 150 KB

**3. Memory Utilization**

**In the simulator:**

* Memory used = sum of all allocated blocks
* Memory utilization % = (total\_used / total\_size) \* 100

**Good utilization occurs when:**

* Large free blocks are merged.
* Few small unusable holes exist.

**Conclusion**

The implemented **First Fit memory allocator** effectively demonstrates dynamic memory management within an operating system environment. While simple, it reflects real-world trade-offs between allocation speed and memory fragmentation. Visual and functional design helps users grasp core OS concepts intuitively.