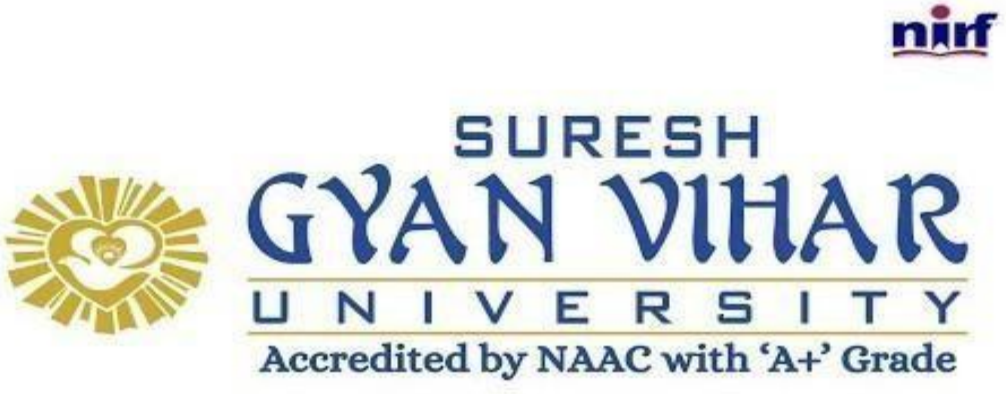


OS Project Report

File Allocation Simulation

In partial fulfillment

For the award of degree of

"Bachelor of Technology"

Department of Computer Engineering & Information technology

Submitted To:

Ms. Priyanka Gupta

(Assistant Professor)

Submitted By:

Sanjeet Kumar

SID – 2247349

Department of Computer Science and Engineering

Suresh Gyan Vihar University, Jaipur

2025-26

STUDENT DECLARATION

I declare that my 7th semester report entitled **'File Allocation Simulation'** is my own work conducted under supervision of Priyanka Gupta mam.

I further declare that to the best of our knowledge the report for B.tech 7th semester does not contain part of the work which has submitted for the award of B.tech degree either in this or any other university without proper citation.

Student's sign

Ms. Prinyanka Gupta

(Assistant professor)

CERTIFICATE

This is to certify that the project report entitled **'File Allocation Simulation,**

Is a bonafied report of the work carried by Sanjeet kumar under guidance and supervision for the partial fulfilment of degree of the B.tech CSE at Suresh Gyan Vihar University, Jaipur.

To the best of our knowledge and belief, this work embodies the work of candidates themselves, has fully been completed, fulfils the requirement of the ordinance relating to the bachelor degree of the university and is up to the standard in respect of content, presentation and language for being referred to the examiner.

Ms. Priyanka Gupta **Mr. Sohit Agarwal**

(Assistant Professor) **HOD(CEIT)**

ACKNOWLEDGEMENT

Working in a good environment and motivation enhance the quality of the work and I get it from my college through our OS project. I have been permitted to take this golden opportunity under the expert guidance of Priyanka Gupta maam from SGVU , Jaipur. I am heartily thankful to her to make complete my project successfully. She has given us her full experience and extra knowledge in practical field. I am also thankful to my head of department Mr. Sohit Agarwal and all CEIT staff to guide us. Finally, we think all the people who had directly or indirectly help as to complete our project.

Student Name:

Sanjeet Kumar

SID-2247349

Table of Contents

1.Abstract

The project **File Allocation Simulation** focuses on demonstrating how operating systems manage the storage of files on secondary memory using different allocation strategies. Efficient file allocation plays a vital role in optimizing disk utilization, minimizing access time, and reducing fragmentation. In this project, three primary allocation methods—**Contiguous Allocation, Linked Allocation, and Indexed Allocation**—are simulated to analyze their working, advantages, and limitations.

Through the simulation, disk blocks are represented in an array-based structure, and files are allocated dynamically according to the chosen strategy. The project highlights how contiguous allocation offers faster access but suffers from fragmentation, how linked allocation provides flexibility at the cost of access speed, and how indexed allocation ensures direct access with minimal fragmentation but requires additional index overhead.

The results of this project provide valuable insights into the **trade-offs between speed, efficiency, and flexibility** in file system design. This practical implementation not only deepens understanding of operating system concepts but also lays the foundation for exploring advanced file system architectures used in modern computing.

2.Introduction

In modern operating systems, **efficient file management** is crucial because files are the primary way of storing and accessing data. **File allocation** refers to the method by which the operating system stores files on secondary storage devices like hard disks, SSDs, or flash drives. The organization of files directly impacts **disk utilization, access time, and fragmentation**.

This project simulates file allocation strategies to demonstrate how files occupy disk blocks and how the OS handles storage dynamically. By performing this simulation, we can better understand **how operating systems optimize storage, manage free space, and locate files efficiently**.

File allocation is important because improper allocation can lead to **wasted space**, slower access times, and even **file corruption** in extreme cases. This project aims to provide a hands-on understanding of the following:

- How files are stored in contiguous or scattered memory blocks
- How pointers or index tables assist in file retrieval
- The trade-offs between speed, memory usage, and flexibility
-

3.Objectives

The main objectives of this project are:

1. **Understanding File Allocation Techniques:** To learn the three primary methods
2. of storing files: contiguous, linked, and indexed.
3. **Simulation of Allocation:** To implement these techniques in a program that
4. simulates disk allocation.
5. **Analysis of Advantages and Disadvantages:** To observe the effects of each
6. method on disk performance, fragmentation, and access speed.

7. **Practical Visualization:** To represent disk blocks, free spaces, and file allocation graphically or in the console.
8. **Problem Solving:** To practice solving challenges such as **fragmentation, file**
9. **extension, and block search efficiency**.

By completing this project, students gain a deeper understanding of file system design and the complexities of **disk storage management**.

4.Working Principle

File Allocation Methods

File allocation methods determine **how the operating system stores and retrieves files** from secondary storage. Each method has its unique characteristics, advantages, and limitations.

Contiguous Allocation

In **contiguous allocation**, each file is stored in **a set of consecutive blocks** on the disk.

- **Working:** If a file requires n blocks, the OS finds n free contiguous blocks and allocates them to the file.
- **Example:**

```
Disk Blocks: [0 1 2 3 4 5 6 7 8 9]
File1 needs 3 blocks → allocated at 0,1,2
File2 needs 2 blocks → allocated at 3,4
```

- **Advantages:**
    1. Fast Access: Sequential blocks improve read/write speed.
    2. Simple Implementation: Easy to calculate block locations.
- **Disadvantages:**
    1. External Fragmentation: Over time, free spaces may be scattered, making it difficult to allocate large files.

2. File Extension Problem: Expanding a file may require moving it to a larger contiguous space.

Indexed Allocation

In **indexed allocation**, the OS uses an **index block** to store all the addresses of the file blocks.

- **Working:**

```
Index Block = [5, 8, 3, 6]
File blocks are located by accessing the index block
```

- **Advantages:**
    1. Direct Access: You can directly access any block without traversing the file.
    2. No External Fragmentation: Blocks can be scattered anywhere on disk.
    3. Supports Large Files: Can use multi-level indexing for very large files.
- **Disadvantages:**
    1. Index Block Overhead: Requires additional storage for the index block.
    2. Limited by Index Size: File size depends on the capacity of the index block.

5.Tools and Environment

To simulate file allocation effectively, the following tools and environment were used:

- Programming Language: C++ (alternatively Python/Java, depending on implementation)
- IDE: Code::Blocks / VS Code / Eclipse
- Operating System: Windows 10 / Linux Ubuntu
- Libraries: Standard input/output libraries for C++ (iostream), arrays, and vectors
- Simulation Features:
    - Disk represented as an array of fixed-size blocks
    - Allocation and deallocation of files
    - Visualization of disk usage

- Handling dynamic file size and block search

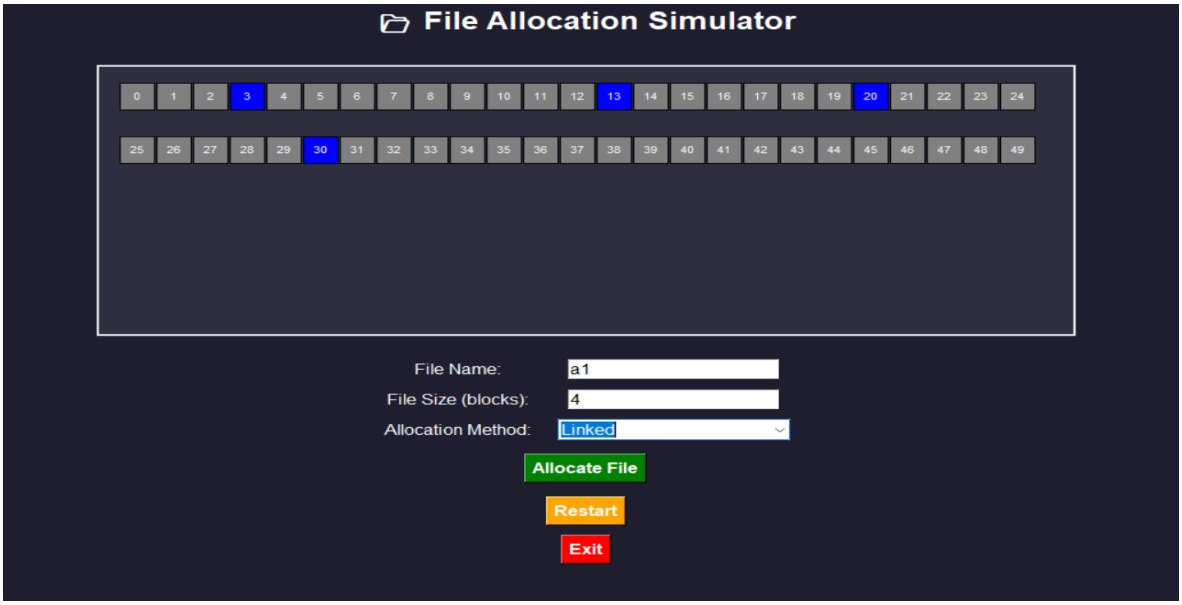
This setup allows a console-based simulation, which makes it easier to understand the core concepts of file allocation without the complexity of a real file system.

6.Code Implementation

The implementation simulates all three file allocation techniques with the following steps:

1. **Disk Initialization:** Create an array to represent disk blocks (e.g., disk[50]). Each element represents a block and its status (Free or Allocated).
2. **File Creation:** Input file name and size (in blocks).
3. **File Allocation:**
    - **Contiguous:** Search for consecutive free blocks and allocate.
    - **Linked:** Allocate scattered blocks and store pointers.
    - **Indexed:** Create an index block and map all allocated blocks.
4. **Display Disk Status:** Show disk blocks, free blocks, and which file occupies which block.
5. **Search File:** Locate a file and list its allocated blocks.
6. **Delete File:** Free allocated blocks and update disk status.

Example Simulation (Linked Allocation):



7.Source Code:

main.py

```
import tkinter as tk
from tkinter import messagebox, ttk
import random

# Disk size (number of blocks)
DISK_SIZE = 50
disk_blocks = [None] * DISK_SIZE

# Colors for methods
METHOD_COLORS = {
    "Contiguous": "red",
    "Linked": "blue",
    "Indexed": "green"
}

# GUI Window
root = tk.Tk()
root.title("File Allocation Methods Simulator")
root.geometry("900x600")
root.config(bg="#1e1e2f")

# Title
title = tk.Label(root, text=" File Allocation Simulator",
                 font=("Arial", 20, "bold"), bg="#1e1e2f", fg="white")
title.pack(pady=10)

# Canvas for disk blocks
canvas = tk.Canvas(root, width=850, height=300, bg="#2e2e3e")
canvas.pack(pady=20)

block_rects = []
for i in range(DISK_SIZE):
    x = 20 + (i % 25) * 32
    y = 20 + (i // 25) * 60
    rect = canvas.create_rectangle(x, y, x + 30, y + 30, fill="gray")
    text = canvas.create_text(x + 15, y + 15, text=str(i), fill="white", font=("Arial", 8))
    block_rects.append((rect, text))
```

```
    block_rects.append((rect, text))

# Form
frame = tk.Frame(root, bg="#1e1e2f")
frame.pack()

tk.Label(frame, text="File Name:", fg="white", bg="#1e1e2f", font=("Arial", 12)).grid(row=0, column=0, padx=10, pady=5)
file_name_entry = tk.Entry(frame, font=("Arial", 12))
file_name_entry.grid(row=0, column=1, padx=10, pady=5)

tk.Label(frame, text="File Size (blocks):", fg="white", bg="#1e1e2f", font=("Arial", 12)).grid(row=1, column=0, padx=10, pady=5)
file_size_entry = tk.Entry(frame, font=("Arial", 12))
file_size_entry.grid(row=1, column=1, padx=10, pady=5)

tk.Label(frame, text="Allocation Method:", fg="white", bg="#1e1e2f", font=("Arial", 12)).grid(row=2, column=0, padx=10, pady=5)
method_combo = ttk.Combobox(frame, values=["Contiguous", "Linked", "Indexed"], font=("Arial", 12))
method_combo.grid(row=2, column=1, padx=10, pady=5)

# Functions
def allocate_file():
    file_name = file_name_entry.get().strip()
    try:
        file_size = int(file_size_entry.get().strip())
    except:
        messagebox.showerror("Error", "Enter valid file size!")
        return

    method = method_combo.get()
    if not file_name or file_size <= 0 or not method:
        messagebox.showerror("Error", "Enter all details correctly!")
        return

    if file_size > DISK_SIZE:
        messagebox.showerror("Error", "File too large for disk!")
        return
```

```
    if method == "Contiguous":
        start = random.randint(0, DISK_SIZE - file_size)
        if all(disk_blocks[i] is None for i in range(start, start + file_size)):
            for i in range(start, start + file_size):
                disk_blocks[i] = file_name
                canvas.itemconfig(block_rects[i][0], fill=METHOD_COLORS[method])
        else:
            messagebox.showerror("Error", "No contiguous space available!")

    # Linked Allocation
    elif method == "Linked":
        free_blocks = [i for i in range(DISK_SIZE) if disk_blocks[i] is None]
        if len(free_blocks) < file_size:
            messagebox.showerror("Error", "Not enough free blocks!")
            return
        chosen = random.sample(free_blocks, file_size)
        for i in chosen:
            disk_blocks[i] = file_name
            canvas.itemconfig(block_rects[i][0], fill=METHOD_COLORS[method])

    # Indexed Allocation
    elif method == "Indexed":
        free_blocks = [i for i in range(DISK_SIZE) if disk_blocks[i] is None]
        if len(free_blocks) < file_size + 1:
            messagebox.showerror("Error", "Not enough free blocks for index + file!")
            return
        index_block = random.choice(free_blocks)
        disk_blocks[index_block] = file_name + "_index"
        canvas.itemconfig(block_rects[index_block][0], fill="yellow")

        free_blocks.remove(index_block)
        chosen = random.sample(free_blocks, file_size)
        for i in chosen:
            disk_blocks[i] = file_name
            canvas.itemconfig(block_rects[i][0], fill=METHOD_COLORS[method])
```

```
    messagebox.showinfo("Success", f"File '{file_name}' allocated using {method} method!")

# Restart Function
def restart_simulation():
    global disk_blocks
    disk_blocks = [None] * DISK_SIZE
    for i in range(DISK_SIZE):
        canvas.itemconfig(block_rects[i][0], fill="gray")  # Reset color
    file_name_entry.delete(0, tk.END)
    file_size_entry.delete(0, tk.END)
    method_combo.set("")
    messagebox.showinfo("Restart", "Simulation has been reset!")

# Buttons
tk.Button(root, text="Allocate File", font=("Arial", 12, "bold"), bg="green", fg="white",
          command=allocate_file).pack(pady=10)

tk.Button(root, text="Restart", font=("Arial", 12, "bold"), bg="orange", fg="white",
          command=restart_simulation).pack(pady=5)

tk.Button(root, text="Exit", font=("Arial", 12, "bold"), bg="red", fg="white",
          command=root.destroy).pack(pady=5)

root.mainloop()
```

file_allocation.py

```
import random

DISK_SIZE = 30

class Disk:
    def __init__(self, size=DISK_SIZE):
        self.size = size
        self.blocks = [None] * size
        self.next_ptr = [-1] * size
        self.directory = {}

    def show_map(self):
        print("\nDisk Map (index:owner):")
        for i, b in enumerate(self.blocks):
            owner = b if b is not None else "."
            print(f"{i:02}:{owner}", end="  ")
            if (i+1) % 8 == 0:
                print()
        print()

    def show_directory(self):
        print("\nDirectory:")
        if not self.directory:
            print(" <empty>")
            return
        for name, meta in self.directory.items():
            print(f" - {name}: {meta}")
        print()

    # ---------- CONTIGUOUS ----------
    def allocate_contiguous(self, name, length, fit='first'):
        # build free segments list (start, len)
        free_segments = []
        i = 0
        while i < self.size:
            if self.blocks[i] is None:
                j = i
```

```
            j = i
            while j < self.size and self.blocks[j] is None:
                j += 1
            free_segments.append((i, j - i))
            i = j
        else:
            i += 1
    # choose segment
    candidate = None
    if fit == 'first':
        for s, l in free_segments:
            if l >= length:
                candidate = (s, length)
                break
    elif fit == 'best':
        best = None
        for s, l in free_segments:
            if l >= length and (best is None or l < best[1]):
                best = (s, l)
        if best:
            candidate = (best[0], length)
    elif fit == 'worst':
        worst = None
        for s, l in free_segments:
            if l >= length and (worst is None or l > worst[1]):
                worst = (s, l)
        if worst:
            candidate = (worst[0], length)

    if not candidate:
        print("No contiguous segment large enough. Allocation failed.")
        return False

    start, _ = candidate
    for i in range(start, start + length):
```

```
            self.blocks[i] = name
        self.directory[name] = {"method":"contiguous", "start":start, "length":length, "blocks":list(range(start, start+length))}
        print(f"Allocated {name} contiguously at {start}..{start+length-1}")
        return True

    def free_contiguous(self, name):
        meta = self.directory.get(name)
        if not meta or meta["method"] != "contiguous":
            return False
        for i in meta["blocks"]:
            self.blocks[i] = None
        del self.directory[name]
        return True

    # ---------- LINKED ----------
    def allocate_linked(self, name, length):
        free_indices = [i for i, b in enumerate(self.blocks) if b is None]
        if len(free_indices) < length:
            print("Not enough free blocks for linked allocation.")
            return False
        chosen = free_indices[:length]
        # link them
        for i in range(length-1):
            self.next_ptr[chosen[i]] = chosen[i+1]
            self.blocks[chosen[i]] = name
        self.next_ptr[chosen[-1]] = -1
        self.blocks[chosen[-1]] = name
        self.directory[name] = {"method":"linked", "head":chosen[0], "blocks":chosen.copy()}
        print(f"Allocated {name} linked blocks: {chosen}")
        return True

    def free_linked(self, name):
        meta = self.directory.get(name)
        if not meta or meta["method"] != "linked":
            return False
```

```
            return False
        for i in meta["blocks"]:
            self.blocks[i] = None
            self.next_ptr[i] = -1
        del self.directory[name]
        return True

    # ---------- INDEXED ----------
    def allocate_indexed(self, name, length):
        # need 1 index block + 'length' data blocks
        free_indices = [i for i, b in enumerate(self.blocks) if b is None]
        if len(free_indices) < length + 1:
            print("Not enough free blocks for indexed allocation.")
            return False
        index_block = free_indices[0]
        data_blocks = free_indices[1:length+1]
        # mark them
        self.blocks[index_block] = f"{name}_idx"
        for db in data_blocks:
            self.blocks[db] = name
        self.directory[name] = {"method":"indexed", "index_block":index_block, "blocks":data_blocks}
        print(f"Allocated {name} indexed: index={index_block}, data={data_blocks}")
        return True

    def free_indexed(self, name):
        meta = self.directory.get(name)
        if not meta or meta["method"] != "indexed":
            return False
        self.blocks[meta["index_block"]] = None
        for i in meta["blocks"]:
            self.blocks[i] = None
        del self.directory[name]
        return True
```

```python
def create_file(self, name, length, method="contiguous", fit='first'):
    if name in self.directory:
        print("File already exists.")
        return False
    if method == "contiguous":
        return self.allocate_contiguous(name, length, fit)
    elif method == "linked":
        return self.allocate_linked(name, length)
    elif method == "indexed":
        return self.allocate_indexed(name, length)
    else:
        print("Unknown method.")
        return False

def delete_file(self, name):
    meta = self.directory.get(name)
    if not meta:
        print("No such file.")
        return False
    m = meta["method"]
    if m == "contiguous":
        return self.free_contiguous(name)
    if m == "linked":
        return self.free_linked(name)
    if m == "indexed":
        return self.free_indexed(name)
    return False

def stats(self):
    free = sum(1 for b in self.blocks if b is None)
    holes = 0
    i = 0
    holesizes = []
    while i < self.size:
        if self.blocks[i] is None:
```

```
                j = i
                while j < self.size and self.blocks[j] is None:
                    j += 1
                holes += 1
                holesizes.append(j - i)
                i = j
            else:
                i += 1
        print(f"\nDisk Stats: free_blocks={free}, holes={holes}, biggest_hole={max(holesizes) if holesizes else 0}")

def demo():
    d = Disk(30)
    while True:
        print("\nMenu: 1.show 2.dir 3.create 4.delete 5.stats 6.exit")
        cmd = input("Choice: ").strip()
        if cmd == "1":
            d.show_map()
        elif cmd == "2":
            d.show_directory()
        elif cmd == "3":
            name = input("Filename: ").strip()
            size = int(input("Size (blocks): ").strip())
            print("Methods: contiguous / linked / indexed")
            method = input("Method: ").strip()
            fit = 'first'
            if method == 'contiguous':
                fit = input("Fit (first/best/worst) [first]: ").strip() or 'first'
            d.create_file(name, size, method, fit)
        elif cmd == "4":
            name = input("Filename to delete: ").strip()
            d.delete_file(name)
        elif cmd == "5":
            d.stats()
        elif cmd == "6":
            print("Bye")
            break
        else:
            print("Invalid")

if __name__ == "__main__":
    demo()
```

8.Flowchart/Algorithm

Contiguous Allocation

1. Input file size.
2. Find n consecutive free blocks.
3. If available, allocate and mark as used.
4. Else, display “No sufficient contiguous space.”

Linked Allocation

1. Input file size.
2. Allocate random free blocks.
3. Store pointer in each block to next block.
4. Mark blocks as used.

Indexed Allocation

1. Create an index block for the file.
2. Allocate required blocks and store their addresses in the index block.
3. Update disk allocation table**.**

Flowchart (simplified steps):

```
Start → Initialize Disk → Input File → Allocate File (Contiguous/Linked/Indexed)
→ Update Disk Status → Display Disk → End
```

8.Results

After simulating the project, the following observations were made:

Allocation Type	**Access Speed**	**Fragmentation**	**Flexibility**	**Remarks**
Contiguous	Fast	High	Low	Best for small or fixed-size files
Linked	Moderate	None	High	Sequential access only
Indexed	Fast	None	Moderate	Supports direct access, index overhead

Key Observations:

1. Contiguous allocation is fast but inefficient when files grow.
2. Linked allocation uses disk space efficiently but slows down access.
3. Indexed allocation provides balanced performance, suitable for large files and random access.

10.Conclusion

The File Allocation Simulation project helped in understanding the core concepts of file storage in operating systems. Through simulation, the advantages and limitations of contiguous, linked, and indexed allocations were clearly observed.

Key takeaways:

- File allocation strategy directly affects disk utilization, performance, and fragmentation.
- Understanding these methods is crucial for designing efficient file systems.
- Indexed allocation provides a good compromise between speed and flexibility for modern systems.

11.Future Enhancements

1. Implement a **GUI-based simulation** using Python Tkinter or Java Swing for better visualization.
2. Handle **dynamic file resizing** and file extension issues.
3. Add **defragmentation simulation** for contiguous allocation.
4. Extend the simulation for **multi-user access** and concurrency.
5. Include **multi-level indexed allocation** for very large files.

12.References

1. Silberschatz, Galvin, Gagne, **Operating System Concepts**, 10th Edition.
2. Tanenbaum, **Modern Operating Systems**, 4th Edition.
3. Achyut S. Godbole, **Operating Systems**, 3rd Edition.
4. Online tutorials and documentation for file allocation algorithms.