VIDHI ROHIRA

S.Y B.TECH

SEM III

COMPUTER ENGINEERING

OS LAB 8

231071052

BATCH – C

LABORATORY 8

AIM:- To implement all file allocation strategies sequential, indexed and linked.

THEORY:-

File allocation strategies are crucial in an operating system as they determine how files are stored and accessed on storage devices. The three main file allocation strategies are Sequential Allocation, Indexed Allocation, and Linked Allocation.

1. **Sequential Allocation** stores files in contiguous blocks, making it efficient for sequential file access but causing fragmentation issues if files are frequently modified.
2. **Indexed Allocation** uses an index block to store the addresses of file blocks, allowing efficient random access and minimizing fragmentation.
3. **Linked Allocation** stores file blocks non-contiguously, with each block pointing to the next, which eliminates fragmentation but may result in slower access times due to the need to traverse the links.

Each allocation strategy has its advantages and trade-offs depending on the type of file access and system requirements.

1. Sequential Allocation

In sequential file allocation, files are stored in contiguous blocks on the disk. This means that the blocks of a file are located one after the other. To find the next block of a file, the operating system simply needs to look at the next disk block.

Characteristics:

- **Storage:** Files are allocated contiguous blocks on the disk.
- **Advantages:**
    - **Efficiency:** This method is simple to implement, and it provides fast sequential access to the file as there is no need to search for the next block.
    - **Low Overhead:** Since the file is stored contiguously, the operating system only needs to keep track of the starting block and the length of the file.
- **Disadvantages:**
    - **External Fragmentation:** As files are allocated in contiguous blocks, the disk can become fragmented over time, causing free space to be scattered and making it harder to find large contiguous spaces.
    - **Fixed Size Issues:** It is difficult to expand files because they need contiguous free space. If no space is available, the file cannot grow unless it is moved or reallocated.

Example:

If a file requires 100 blocks, the operating system will find a contiguous area of 100 free blocks and assign them to the file. The file blocks will be located sequentially, so the operating system will just store the address of the first block and calculate the subsequent blocks based on the file's length.

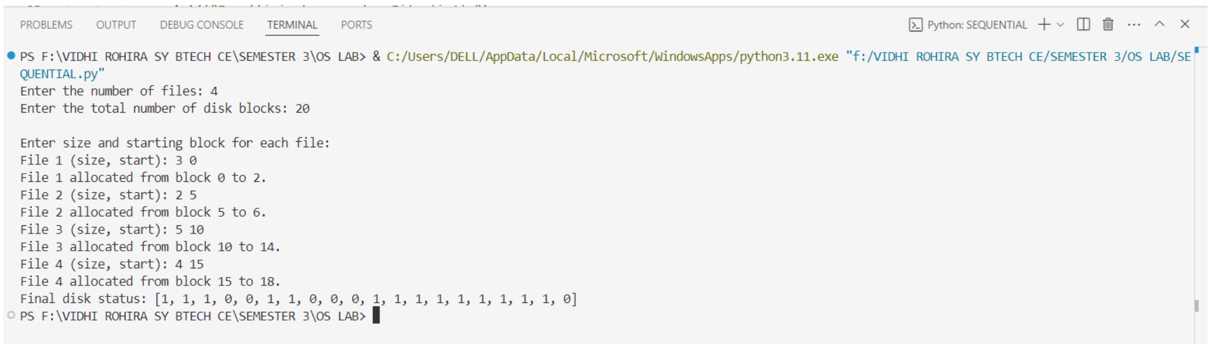
Algorithm:

1. **Input:**
    - Number of files n.
    - For each file, the size in blocks size[] and the starting block start[] are given.
2. **Allocation:**
    - Start from the first block and check if enough contiguous space is available for the file.
    - If the required space is available, mark the blocks as allocated.
    - If the space is insufficient, output an error message.
3. **Output:**
    - Updated file allocation after successfully allocating or handling insufficiency.

CODE:-

```python
def sequential_allocation():
    # User Input
    n = int(input("Enter the number of files: "))
    total_blocks = int(input("Enter the total number of disk blocks: "))
    disk = [0] * total_blocks  # Initialize the disk array: 0 = unallocated, 1
= allocated

    print("\nEnter size and starting block for each file:")

    for i in range(n):
        # Get the size and starting block for the current file
        size, start = map(int, input(f"File {i+1} (size, start): ").split())

        # Check if there is enough contiguous space starting at 'start'
        # Conditions:
        # 1. Blocks from start to start + size - 1 must fit within the disk.
        # 2. All these blocks must be free (disk[j] == 0).
        if start + size - 1 < total_blocks and all(disk[start + j] == 0 for j
in range(size)):
            # Allocate
            for j in range(size):
                disk[start + j] = 1  # Mark the block as allocated
            print(f"File {i+1} allocated from block {start} to {start + size -
1}.")
        else:
            print(f"Insufficient space for File {i+1}.")

    print("Final disk status:", disk)

# Call the function to run
sequential_allocation()
```

OUTPUT:-



```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/OS LAB/SE
QUENTIAL.py"
Enter the number of files: 4
Enter the total number of disk blocks: 20

Enter size and starting block for each file:
File 1 (size, start): 3 0
File 1 allocated from block 0 to 2.
File 2 (size, start): 2 5
File 2 allocated from block 5 to 6.
File 3 (size, start): 5 10
File 3 allocated from block 10 to 14.
File 4 (size, start): 4 15
File 4 allocated from block 15 to 18.
Final disk status: [1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB>
```

2. Indexed Allocation

In indexed allocation, a special index block is created for each file, which contains the addresses (or pointers) of all the blocks that are part of the file. The index block acts as a map that helps in locating the scattered blocks of a file across the disk.

Characteristics:

- **Storage:** Instead of storing files contiguously, the file's data blocks can be scattered across the disk. The index block holds pointers to each of the file's data blocks.
- **Advantages:**
  - **No External Fragmentation:** Since blocks can be scattered, the disk does not suffer from fragmentation.
  - **File Expansion:** Files can be easily expanded because additional blocks can be allocated anywhere on the disk, as long as there is free space.
  - **Efficient Random Access:** Indexed allocation allows for direct access to any block in the file using the index, providing better performance for random access.
- **Disadvantages:**
  - **Index Block Overhead:** There is an additional overhead for maintaining an index block, especially if the file is large and requires many pointers.
  - **Internal Fragmentation:** If the index block is not large enough, it might have to be split into multiple index blocks, causing overhead.

Example: For a file, an index block is created that stores pointers to the data blocks of the file. If a file has data in blocks 10, 20, and 30, the index block will contain the addresses (or pointers) for these blocks. The operating system needs to read the index block first and then access the data blocks based on the pointers in the index block.

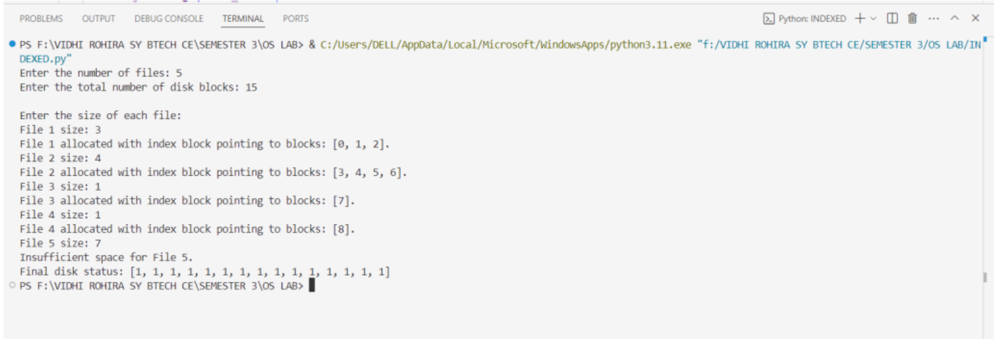
ALGORITHM:

1. **Input:**
   - Number of files n.
   - For each file, the size in blocks size[] and the starting block start[] are given.
2. **Allocation:**
   - Allocate each block independently using an index block.
   - Check if enough space exists for each block (i.e., there should be sufficient unallocated blocks).
   - Use an index block to store the address of each allocated block.
   - If insufficient space is available, output an error message.
3. **Output:**
   - Display the indexed allocation table and any errors encountered during the allocation process.

CODE:-

```python
def indexed_allocation():
    # User Input
    n = int(input("Enter the number of files: "))
    total_blocks = int(input("Enter the total number of disk blocks: "))
    disk = [0] * total_blocks  # Initialize the disk array: 0 = unallocated, 1 = allocated

    print("\nEnter the size of each file:")
    for i in range(n):
        size = int(input(f"File {i+1} size: "))  # User input for file size
        allocated_blocks = []  # List to store the blocks allocated for this file

        # Allocating blocks for the file
        for j in range(total_blocks):
            if disk[j] == 0 and len(allocated_blocks) < size:
                disk[j] = 1  # Mark block as allocated
                allocated_blocks.append(j)  # Add block index to the allocation list

            if len(allocated_blocks) == size:  # If all required blocks are allocated
                break

        # Check if enough blocks were allocated
        if len(allocated_blocks) == size:
            print(f"File {i+1} allocated with index block pointing to blocks: {allocated_blocks}.")
        else:
            print(f"Insufficient space for File {i+1}.")

    # Display final disk status
    print("Final disk status:", disk)

# Call the function
indexed_allocation()
```

OUTPUT:-



```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/OS LAB/INDEXED.py"
Enter the number of files: 5
Enter the total number of disk blocks: 15

Enter the size of each file:
File 1 size: 3
File 1 allocated with index block pointing to blocks: [0, 1, 2].
File 2 size: 4
File 2 allocated with index block pointing to blocks: [3, 4, 5, 6].
File 3 size: 1
File 3 allocated with index block pointing to blocks: [7].
File 4 size: 1
File 4 allocated with index block pointing to blocks: [8].
File 5 size: 7
Insufficient space for File 5.
Final disk status: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB>
```

3] Linked Allocation

In linked allocation, each block of a file contains a pointer to the next block of the file, creating a chain of file blocks. Unlike indexed allocation, there is no need for a separate index block, as each data block stores the address of the next block in the file. This method provides a way to manage files that are stored in non-contiguous blocks on the disk.

Characteristics:

- **Storage:** File blocks are not stored contiguously. Each block contains a pointer to the next block in the sequence.
- **Advantages:**
  - **No External Fragmentation:** Since blocks are not contiguous, there is no external fragmentation, and space can be utilized more efficiently.
  - **Dynamic File Growth:** Files can easily grow as new blocks can be added anywhere on the disk without requiring contiguous free space.
  - **Efficient for Sequential Access:** This allocation is very efficient for files requiring sequential access, as accessing the next block is simply a matter of following the pointer.
- **Disadvantages:**
  - **No Direct Access:** It is inefficient for random access, as accessing a particular block requires traversing the chain of blocks starting from the first one.
  - **Overhead:** Each block requires extra space for storing a pointer to the next block, reducing the space available for actual file data.

Example: In linked allocation, if a file requires blocks at locations 5, 10, 20, and 25, each block (except the last one) contains a pointer to the next block. For example:

- Block 5 points to Block 10.
- Block 10 points to Block 20.
- Block 20 points to Block 25. To read the file, the system starts at Block 5 and follows the chain of pointers to reach the subsequent blocks.

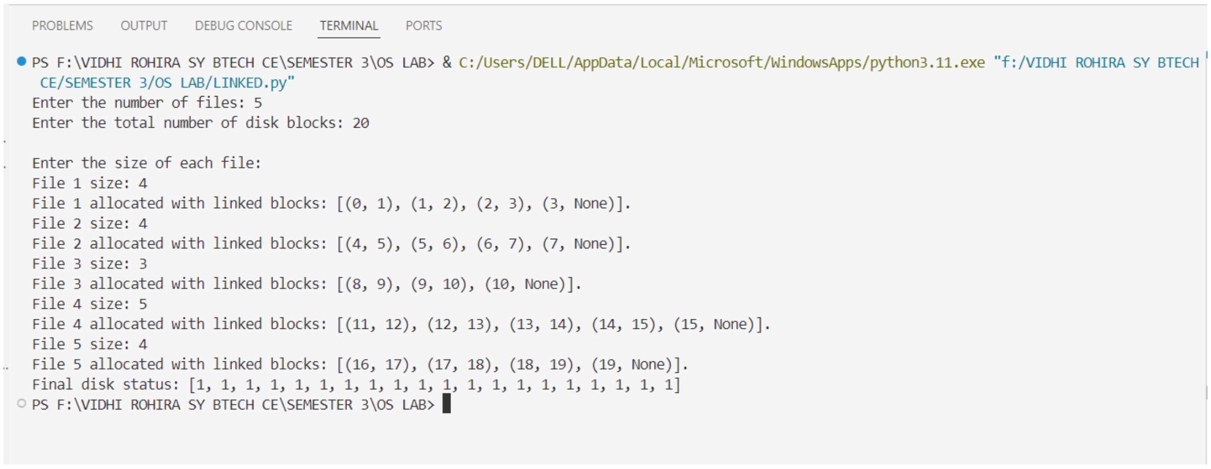
ALGORITHM:

1. **Input:**
   - Number of files, n.
   - For each file, the size (in blocks), size[], and the starting block start[] are given.
2. **Allocation:**
   - Allocate blocks for the file non-contiguously, ensuring that each block points to the next block using a pointer.
   - If there is insufficient space to allocate the file, an error message is displayed.
3. **Output:**
   - Linked allocation table, showing the allocated blocks and their respective pointers.
   - Any errors encountered during allocation.

CODE:-

```
def linked_allocation():
    # User Input
    n = int(input("Enter the number of files: "))
    total_blocks = int(input("Enter the total number of disk blocks: "))
    disk = [0] * total_blocks  # Initialize the disk array: 0 = unallocated, 1
= allocated

    print("\nEnter the size of each file:")

    for i in range(n):
        # Get the size of the current file
        size = int(input(f"File {i+1} size: "))
        allocated_blocks = []

        # Allocating blocks
        for j in range(total_blocks):
            if disk[j] == 0 and len(allocated_blocks) < size:
                disk[j] = 1  # Mark the block as allocated
                allocated_blocks.append(j)  # Add block index to the list

        # Check if the required number of blocks was successfully allocated
        if len(allocated_blocks) == size:
            # Create a linked block list (each block points to the next)
            linked_list = [
                (allocated_blocks[j], allocated_blocks[j + 1] if j + 1 <
len(allocated_blocks) else None)
                for j in range(len(allocated_blocks))
            ]
            print(f"File {i+1} allocated with linked blocks: {linked_list}.")
        else:
            print(f"Insufficient space for File {i+1}.")

    print("Final disk status:", disk)

# Call the linked_allocation function
linked_allocation()
```

OUTPUT:-



```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "f:/VIDHI ROHIRA SY BTECH
 CE/SEMESTER 3/OS LAB/LINKED.py"
Enter the number of files: 5
Enter the total number of disk blocks: 20

Enter the size of each file:
File 1 size: 4
File 1 allocated with linked blocks: [(0, 1), (1, 2), (2, 3), (3, None)].
File 2 size: 4
File 2 allocated with linked blocks: [(4, 5), (5, 6), (6, 7), (7, None)].
File 3 size: 3
File 3 allocated with linked blocks: [(8, 9), (9, 10), (10, None)].
File 4 size: 5
File 4 allocated with linked blocks: [(11, 12), (12, 13), (13, 14), (14, 15), (15, None)].
File 5 size: 4
File 5 allocated with linked blocks: [(16, 17), (17, 18), (18, 19), (19, None)].
Final disk status: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\OS LAB>
```

CONCLUSION:-

Hence in this lab session we implemented and analyzed three file allocation strategies: Sequential, Indexed, and Linked Allocation. Sequential Allocation is most efficient for workloads with sequential access and fixed-size files, offering simplicity and high performance but suffering from fragmentation. Indexed Allocation is best for workloads with frequent random access and dynamic file sizes, as it allows non-contiguous storage and fast block lookups. Linked Allocation is ideal for scenarios with critical disk space utilization and unpredictable file growth, as it avoids fragmentation and supports flexible growth, although it is less efficient for random access. Each strategy's effectiveness depends on the specific workload and system requirements.