OS LAB- 8

AIM: Implement all file allocation strategies ie Sequential, Indexed, Linked.

THEORY:

The file allocation strategies are important in an operating system as they determine
The three main file allocation strategies are Sequential Allocation, Indexed Allocation, and Linked Allocation.

1. Sequential Allocation

Theory:
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.

2. Indexed Allocation

Theory:
In indexed allocation, a special index block is created for each file, which contains the addresses (pointers) of all the blocks that are part of the file. The index block serves as a map to locate the scattered blocks of a file.

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.

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**:
    - Indexed allocation table and any errors.

3. Linked Allocation

Theory:
In linked allocation, each block of a file contains a pointer to the next block of the file, creating a linked list of file blocks. There is no need for an index block as in the indexed allocation scheme, but each data block contains the address of the next block.

Characteristics:

- **Storage**: Blocks are not contiguous. Each data block has a pointer to the next block in the chain.
- **Advantages**:
    - **No External Fragmentation**: Like indexed allocation, linked allocation avoids fragmentation.
    - **Dynamic File Growth**: New blocks can be added to the file without any need for contiguous space.
    - **Efficient for Sequential Access**: This allocation is well-suited for files that require sequential access, as reading the next block is just a matter of following the pointer.
- **Disadvantages**:

- **No Direct Access**: Linked allocation is inefficient for random access because to access a particular block, the system must follow the chain from the beginning of the file until it reaches the desired block.
- **Overhead**: Each block requires extra space for storing a pointer to the next block, reducing the space available for actual data.

Example: In linked allocation, if a file has blocks located at 5, 10, 20, and 25, each block (except the last one) contains a pointer to the next block. Block 5 will point to block 10, block 10 will point to block 20, and block 20 will point to block 25. When reading the file, the system starts at block 5 and follows the pointers until it reaches the end.

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.
   - Each block of the file points to the next block using a pointer.
   - If there is not enough space for the entire file, output an error message.
3. **Output**:
   - Linked allocation table and any errors.

CODE:

SEQUENTIAL:

```
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)
sequential_allocation()
```

```
Enter the number of files: 3
Enter the total number of disk blocks: 15

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): 5 5
File 2 allocated from block 5 to 9.
File 3 (size, start): 2 10
File 3 allocated from block 10 to 11.
Final disk status: [1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0]
```

INDEXED:

```
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 size
        allocated_blocks = []  # List to store the blocks allocated for this file

        #Allocating
        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:
            print(f"File {i+1} allocated with index block pointing to blocks: {allocated_blocks}.")
        else:
            print(f"Insufficient space for File {i+1}.")

    print("Final disk status:", disk)
indexed_allocation()
```

```
Enter the number of files: 3
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: 5
File 2 allocated with index block pointing to blocks: [3, 4, 5, 6, 7].
File 3 size: 2
File 3 allocated with index block pointing to blocks: [8, 9].
Final disk status: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
```

LINKED:

```
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
        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)
linked_allocation()
```

```
Enter the number of files: 3
Enter the total number of disk blocks: 15

Enter the size of each file:
File 1 size: 3
File 1 allocated with linked blocks: [(0, 1), (1, 2), (2, None)].
File 2 size: 5
File 2 allocated with linked blocks: [(3, 4), (4, 5), (5, 6), (6, 7), (7, None)].
File 3 size: 2
File 3 allocated with linked blocks: [(8, 9), (9, None)].
Final disk status: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
PS C:\Anagha\college\OS\OS_lab>
```

CONCLUSION:

We implemented and analyzed three file allocation strategies: Sequential Allocation, Indexed Allocation, and Linked Allocation.

If the workload involves primarily sequential access with fixed-size files, Sequential Allocation is the most efficient choice due to its simplicity and high performance.

For workloads with frequent random access and dynamic file sizes, Indexed Allocation is the optimal strategy because it supports non-contiguous storage and fast lookups.

In cases where disk space utilization is critical and files grow unpredictably, Linked Allocation is appropriate, as it avoids fragmentation and allows for flexible file growth.