ASSIGNMENT-1

GROUP-F

NAME	PRATHAM SETH
ROLL.NO	2022A1R037
BRANCH	C.S.E
SECTION	A2
SEMESTER	3rd
SUBJECT	Operating System

SUBMITTED BY:
PRATHAM SETH

SUBMITTED TO:
Ms.MEKHLA SHARMA

Index

Task 1:

Create a program that simulates different file allocation methods, such as contiguous allocation, linked allocation, and indexed allocation. Students should implement these methods and compare their advantages and disadvantages in terms of storage utilization, fragmentation, and file access performance.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// File structure
struct File {
    char name[20];
    int size;
    // Additional attributes as needed
};
// Contiguous Allocation
void contiguousAllocation(struct File files[], int numFiles) {
 int totalSize = 100; // Total size of the disk
 int allocatedSize = 0; // Total size allocated
 int blockSize;
 printf("Enter block size for contiguous allocation: ");
 scanf("%d", &blockSize);
for (int i = 0; i < numFiles; ++i) {
 if (allocatedSize + files[i].size <= totalSize) {
printf("Allocating %s in contiguous blocks: ", files[i].name);
            for (int j = 0; j < files[i].size / blockSize; ++j) {
                printf("[%d]", allocatedSize + j * blockSize);
```

```
            }
printf("\n");
            allocatedSize += files[i].size;
        } else {
            printf("Not enough space to allocate %s. Disk full.\n", files[i].name);
        }
    }
}
// Linked Allocation
void linkedAllocation(struct File files[], int numFiles) {
    // Simplified linked allocation using a linked list
    struct Node {
        struct File data;
        struct Node* next;
  };
struct Node* head = NULL;
 for (int i = 0; i < numFiles; ++i) {
 struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
 newNode->data = files[i];
 newNode->next = NULL;
  if (head == NULL) {
  head = newNode;
        }
else
{
```

```
    struct Node* current = head;
   while (current->next != NULL) {
    current = current->next;
            }
            current->next = newNode;
        }
    }

    // Display linked allocation
    struct Node* current = head;
    while (current != NULL) {
        printf("File: %s, Size: %d, Linked to next\n", current->data.name, current->data.size);
        current = current->next;
    }

    // Clean up memory
    current = head;
    while (current != NULL) {
        struct Node* temp = current;
        current = current->next;
        free(temp);
    }
}

// Indexed Allocation
```

```
void indexedAllocation(struct File files[], int numFiles) {
    // Simplified indexed allocation using an array
    const int maxSize = 100;
    int indexTable[maxSize];

    memset(indexTable, -1, sizeof(indexTable)); // Initialize index table

    int currentIndex = 0;
 for (int i = 0; i < numFiles; ++i) {
        if (currentIndex + files[i].size <= maxSize) {
            printf("Allocating %s with index: [%d - %d]\n", files[i].name, currentIndex, currentIndex
+ files[i].size - 1);
            for (int j = 0; j < files[i].size; ++j) {
                indexTable[currentIndex + j] = files[i].size;
            }
            currentIndex += files[i].size;
        } else {
            printf("Not enough space to allocate %s. Disk full.\n", files[i].name);
        }
    }
}

int main() {
    struct File files[] = {{"File1", 20}, {"File2", 15}, {"File3", 30}};
    int numFiles = sizeof(files) / sizeof(files[0]);
```

```
    int choice;

    do {
        printf("File Allocation Methods Simulation\n");
        printf("1. Contiguous Allocation\n");
        printf("2. Linked Allocation\n");
        printf("3. Indexed Allocation\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
switch (choice) {
            case 1:
                contiguousAllocation(files, numFiles);
                break;
            case 2:
                linkedAllocation(files, numFiles);
                break;
            case 3:
                indexedAllocation(files, numFiles);
                break;
            case 4:
                printf("Exiting program.\n");
                break;
            default:
```

```
            printf("Invalid choice. Please enter a valid option.\n");
        }
} while (choice != 4);
 return 0;
}
```

OUTPUT:

```
File Allocation Methods Simulation
1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation
4. Exit
Enter your choice: _
```

```
File Allocation Methods Simulation
1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation
4. Exit
Enter your choice: 1
Enter block size for contiguous allocation: 3
Allocating File1 in contiguous blocks: [0][3][6][9][12][15]
Allocating File2 in contiguous blocks: [20][23][26][29][32]
Allocating File3 in contiguous blocks: [35][38][41][44][47][50][53][56][59][62]
```

```
Enter your choice: 2
File: File1, Size: 20, Linked to next
File: File2, Size: 15, Linked to next
File: File3, Size: 30, Linked to next
```

```
Enter your choice: 3
Allocating File1 with index: [0 - 19]
Allocating File2 with index: [20 - 34]
Allocating File3 with index: [35 - 64]
```

Advantages of Contiguous Allocation:

1. **Sequential Access**:Contiguous allocation allows for efficient sequential access to files. Since the entire file is stored in a contiguous block of storage, reading or writing the file sequentially is straightforward and doesn't require complex indexing.

2. **Simple Implementation**: Contiguous allocation is relatively simple to implement. The file system just needs to keep track of the starting block and size of each file. This simplicity can lead to better performance in certain scenarios.

3. **Low Overhead**:There is minimal overhead associated with managing file allocation in a contiguous manner. There is no need for complex data structures like linked lists or index tables.

Disadvantages of Contiguous Allocation:

1. **Fragmentation**:One of the significant drawbacks of contiguous allocation is internal fragmentation. As files are allocated in contiguous blocks, there may be wasted space within these blocks if the file size is not an exact multiple of the block size. Over time, this can lead to inefficient use of storage space.

2. **Limited Flexibility**: Contiguous allocation can be inflexible when it comes to inserting or removing files. It may be challenging to find a contiguous block large enough for a new file, especially in a system with varying file sizes.

3. **File Deletion Issues**: When a file is deleted, the space it occupied becomes available. However, if this space is not immediately reused for a new file, it can contribute to external fragmentation, where free space is scattered throughout the storage.

4. **Difficult to Allocate Large Files**: Allocating large files in a contiguous manner can be challenging, especially if the storage space is fragmented. Finding a single contiguous block large enough for the file may be difficult.

Storage Utilization, Fragmentation, and File Access Performance Summary:

1.**Storage Utilization**:Contiguous allocation can provide good storage utilization for smaller systems or systems with consistent file sizes. However, as the system grows and file sizes vary, internal fragmentation can become a significant concern.

2. **Fragmentation**:Contiguous allocation is susceptible to both internal and external fragmentation, impacting the overall efficiency of storage usage.

3. **File Access Performance**: File access performance for contiguous allocation is generally good for sequential access patterns. However, random access or frequent insertions and deletions of files can lead to performance challenges due to fragmentation.

Advantages of Linked Allocation:

1**. Dynamic File Sizes**: Linked allocation supports dynamic file sizes, as each block of a file can be individually allocated. This makes it more flexible than contiguous allocation when dealing with files of varying sizes.

2. **Ease of File Insertion and Deletion**: Linked allocation makes it relatively easy to insert and delete files. When a file is deleted, its blocks can be easily marked as free, and the space can be efficiently reused.

3. **No Internal Fragmentation**: Unlike contiguous allocation, linked allocation does not suffer from internal fragmentation. Each block is allocated individually, eliminating wasted space within blocks.

Disadvantages of Linked Allocation:

1. **Storage Overhead**: Linked allocation introduces storage overhead due to the need to store pointers or addresses for each block. This can result in a higher overhead compared to contiguous allocation, especially for small files.

2**. Random Access Inefficiency**:Accessing files randomly can be less efficient with linked allocation. To access a specific block, the system must traverse the linked list from the beginning, which can result in slower access times for large files.

3. **Fragmentation and Scattered Free Space**: Linked allocation can suffer from external fragmentation. Free blocks may be scattered throughout the storage, making it challenging to find contiguous space for a new file, especially if the free blocks are not co-located.

4. **Limited Sequential Access Performance**:While sequential access can be efficient, linked allocation may not perform as well as contiguous allocation for large sequential reads or writes. This is because the data blocks are not physically contiguous.

Storage Utilization, Fragmentation, and File Access Performance Summary:

1**. Storage Utilization**: Linked allocation can provide good storage utilization for systems with varying file sizes. However, the storage overhead of maintaining pointers can impact overall utilization.

2. **Fragmentation**: Linked allocation is less prone to internal fragmentation but can suffer from external fragmentation due to scattered free blocks.

3.**File Access Performance**:Linked allocation performs well for sequential access but may be less efficient for random access. Traversing linked lists for each block access can introduce additional latency.

Advantages of Indexed Allocation:

1**. Efficient Random Access**: Indexed allocation excels in providing efficient random access to files. An index table stores the addresses of all the blocks of a file, allowing direct access to any block without the need to traverse linked lists or search for contiguous blocks.

2. **No External Fragmentation**: Unlike contiguous and linked allocation, indexed allocation does not suffer from external fragmentation. Each file's blocks are referenced in the index table, and there is no need to search for free contiguous blocks in the storage.

3. **Flexibility in File Sizes**: Indexed allocation supports dynamic file sizes, similar to linked allocation. Files can grow or shrink without the need for contiguous blocks, making it suitable for systems with varying file sizes.

Disadvantages of Indexed Allocation:

1. **Storage Overhead**: Indexed allocation introduces storage overhead due to the need for an index table. Each file requires additional space in the index table to store the addresses of its blocks. This overhead can be significant for small files.

2. **Limited Sequential Access Performance**: While random access is efficient, sequential access can be less efficient compared to contiguous allocation. This is because the blocks of a file may not be physically contiguous in the storage.

3. **Complex Implementation**: Indexed allocation is more complex to implement than contiguous or linked allocation. Managing and updating the index table adds complexity to the file system design.

Storage Utilization, Fragmentation, and File Access Performance Summary:

1.**Storage Utilization**: Indexed allocation can provide good storage utilization, especially for systems with dynamic file sizes. However, the storage overhead of the index table can impact overall utilization, particularly for small files.

2.**Fragmentation**: Indexed allocation does not suffer from external fragmentation, making it suitable for systems where avoiding fragmentation is a priority. However, internal fragmentation may still occur if the block size is not aligned with the file size.

3.**File Access Performance**: Indexed allocation excels in random access scenarios, providing efficient access to any block of a file. However, sequential access performance may not be as efficient as contiguous allocation.

Task 2:

Differentiate between deadlock and starvation in the context of operating systems. Define each concept, describe their effects on system performance, and provide examples of scenarios where they might occur. Discuss potential strategies to address and prevent both.

Deadlock vs. Starvation in Operating Systems

Deadlock and starvation are two undesirable conditions that can arise in operating systems, both of which can hinder system performance and hinder the progress of processes. While they share some similarities, they are distinct phenomena with different causes and consequences.

Deadlock

Deadlock occurs when two or more processes are waiting for each other to release resources they each hold, creating a circular dependency that prevents any process from making progress. This situation is often referred to as a "circular wait" or a "deadlock trap."

Effects of Deadlock on System Performance:

1.Deadlocked processes cannot execute, leading to wasted CPU cycles and decreased throughput.
2.System resources remain unavailable to other processes, causing delays and inefficiencies.
3.In severe cases, deadlock can bring the entire system to a halt.

Example of Deadlock:

Process 1 holds Resource A and waits for Resource B, while Process 2 holds Resource B and waits for Resource A. Both processes are stuck, unable to proceed, and the system is deadlocked.

Starvation

Starvation occurs when a process is indefinitely denied access to a resource it needs, causing it to wait indefinitely without ever getting to execute. This situation typically arises when high-priority processes continuously consume resources, leaving low-priority processes perpetually waiting.

Effects of Starvation on System Performance:

1.Low-priority processes are unable to complete their tasks, leading to unfair resource allocation.
2.Overall system throughput is reduced as low-priority processes remain blocked.
3.Starved processes may consume excessive CPU time trying to acquire resources, further degrading system performance.

Example of Starvation:

A low-priority process is constantly waiting for a printer while high-priority processes continuously send print jobs. The low-priority process remains starved, unable to access the printer, while the high-priority processes monopolize the resource.

Strategies to Address and Prevent Deadlock:

1.**Resource Preemption**: Allow the operating system to forcibly take away resources from deadlocked processes to break the circular dependency.

2.**Resource Ordering**: Assign resources in a specific order to prevent processes from holding resources required by others.

3.**Deadlock Detection and Avoidance**: Use algorithms to detect deadlocks and take preventive measures, such as delaying resource allocation or temporarily locking resources.

Strategies to Address and Prevent Starvation:

1.Priority-Based Scheduling: Implement scheduling algorithms that prioritize low-priority processes to ensure they get a fair share of resources.

2.Aging: Gradually increase the priority of starved processes to give them a chance to acquire resources and make progress.

3.Resource Quotas: Enforce limits on the amount of resources high-priority processes can consume to prevent them from monopolizing resources and starving others.