**ASSIGNMENT**

*BY*

Aryan Manhas

2022A1R033

3rd Semester

Computer Science Engineering (CSE)

GROUP F (2022A1R033-2022A1R038)

A logo with colorful triangles

Description automatically generated

**Model Institute of Engineering and Technology** (Autonomous)

(Permanently Affiliated to the University of Jammu, Accredited by NAAC with “A” Grade)

Jammu, India

2023

**A close-up of a logo

Description automatically generated**

# ASSIGNMENT

**Subject Code:** COM 302

**Subject Name:** Operating Systems

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Question Number** | **Course Outcomes** | **Blooms’ Level** | **Maximum Marks** | **Marks Obtain** |
| Q1 | CO1, CO2 & CO5 | 3,4 | 10 |  |
| Q2 | CO3, CO4 | 3,4 | 10 |  |
| **Total Marks** | | | 20 |  |
| **Faculty Name:** Assistant Professor Ms. Mekhla Sharma | | | | |

**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.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// Constants

#define MAX\_BLOCKS 100

// Structure for a file

typedef struct {

int startBlock;

int size;

} File;

// Array to represent disk blocks

int disk[MAX\_BLOCKS];

// Function prototypes

void contiguousAllocation();

void linkedAllocation();

void indexedAllocation();

int main() {

int choice;

printf("File Allocation Methods:\n");

printf("1. Contiguous Allocation\n");

printf("2. Linked Allocation\n");

printf("3. Indexed Allocation\n");

printf("Enter your choice (1-3): ");

scanf("%d", &choice);

switch (choice) {

case 1:

contiguousAllocation();

break;

case 2:

linkedAllocation();

break;

case 3:

indexedAllocation();

break;

default:

printf("Invalid choice. Exiting...\n");

break;

}

return 0;

}

void contiguousAllocation() {

int totalBlocks, numFiles;

printf("Enter the total number of blocks on the disk: ");

scanf("%d", &totalBlocks);

printf("Enter the number of files: ");

scanf("%d", &numFiles);

if (numFiles > totalBlocks) {

printf("Error: Number of files exceeds the total number of blocks. Exiting...\n");

return;

}

File files[numFiles];

// Initialize disk blocks

for (int i = 0; i < totalBlocks; ++i) {

disk[i] = 0; // 0 indicates the block is free

}

// Allocate contiguous blocks for each file

for (int i = 0; i < numFiles; ++i) {

printf("Enter the size of File %d: ", i + 1);

scanf("%d", &files[i].size);

// Check if the file size exceeds available space

if (files[i].size > totalBlocks) {

printf("Error: File %d size exceeds the total number of blocks. Skipping allocation for this file.\n", i + 1);

continue;

}

// Find contiguous blocks for the file

int start = -1;

int count = 0;

for (int j = 0; j < totalBlocks; ++j) {

if (disk[j] == 0) {

if (start == -1) {

start = j;

}

count++;

if (count == files[i].size) {

break;

}

} else {

start = -1;

count = 0;

}

}

// Update disk and file information

if (start != -1) {

files[i].startBlock = start;

for (int j = start; j < start + files[i].size; ++j) {

disk[j] = 1; // 1 indicates the block is allocated

}

printf("File %d allocated from block %d to %d.\n", i + 1, start, start + files[i].size - 1);

} else {

printf("File %d cannot be allocated contiguous blocks.\n", i + 1);

}

}

}

void linkedAllocation() {

int totalBlocks, numFiles;

printf("Enter the total number of blocks on the disk: ");

scanf("%d", &totalBlocks);

printf("Enter the number of files: ");

scanf("%d", &numFiles);

if (numFiles > totalBlocks) {

printf("Error: Number of files exceeds the total number of blocks. Exiting...\n");

return;

}

File files[numFiles];

// Initialize disk blocks

for (int i = 0; i < totalBlocks; ++i) {

disk[i] = -1; // -1 indicates the block is free

}

// Allocate linked blocks for each file

for (int i = 0; i < numFiles; ++i) {

printf("Enter the size of File %d: ", i + 1);

scanf("%d", &files[i].size);

// Check if the file size exceeds available space

if (files[i].size > totalBlocks) {

printf("Error: File %d size exceeds the total number of blocks. Skipping allocation for this file.\n", i + 1);

continue;

}

int start = -1;

int prev = -1;

// Find blocks for the file

for (int j = 0; j < totalBlocks && files[i].size > 0; ++j) {

if (disk[j] == -1) {

disk[j] = 1; // 1 indicates the block is allocated

files[i].size--;

if (start == -1) {

start = j;

}

if (prev != -1) {

disk[prev] = j;

}

prev = j;

}

}

// Update file information

if (start != -1) {

files[i].startBlock = start;

disk[prev] = -1; // Set the last block's link to -1

printf("File %d allocated with linked blocks starting from block %d.\n", i + 1, start);

} else {

printf("File %d cannot be allocated linked blocks.\n", i + 1);

}

}

}

void indexedAllocation() {

int totalBlocks, numFiles, indexBlock;

printf("Enter the total number of blocks on the disk: ");

scanf("%d", &totalBlocks);

printf("Enter the number of files: ");

scanf("%d", &numFiles);

printf("Enter the index block number: ");

scanf("%d", &indexBlock);

if (numFiles > totalBlocks) {

printf("Error: Number of files exceeds the total number of blocks. Exiting...\n");

return;

}

File files[numFiles];

// Initialize disk blocks

for (int i = 0; i < totalBlocks; ++i) {

disk[i] = 0; // 0 indicates the block is free

}

// Initialize the index block

int index[MAX\_BLOCKS];

for (int i = 0; i < MAX\_BLOCKS; ++i) {

index[i] = -1;

}

// Allocate indexed blocks for each file

for (int i = 0; i < numFiles; ++i) {

printf("Enter the size of File %d: ", i + 1);

scanf("%d", &files[i].size);

// Check if the file size exceeds available space

if (files[i].size > totalBlocks) {

printf("Error: File %d size exceeds the total number of blocks. Skipping allocation for this file.\n", i + 1);

continue;

}

int start = -1;

// Find blocks for the file

for (int j = 0; j < totalBlocks && files[i].size > 0; ++j) {

if (disk[j] == 0) {

disk[j] = 1; // 1 indicates the block is allocated

files[i].size--;

if (start == -1) {

start = j;

index[i] = j;

}

}

}

// Update file information

if (start != -1) {

files[i].startBlock = start;

printf("File %d allocated with indexed blocks starting from block %d.\n", i + 1, start);

} else {

printf("File %d cannot be allocated indexed blocks.\n", i + 1);

}

}

// Display the index block

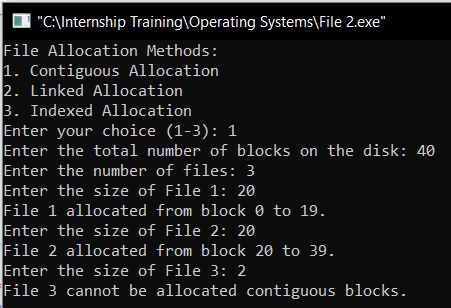
printf("\nIndex Block Contents:\n");

for (int i = 0; i < numFiles; ++i) {

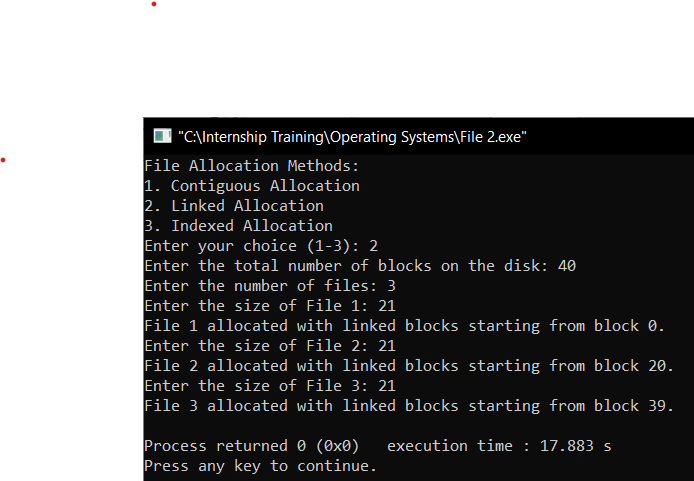
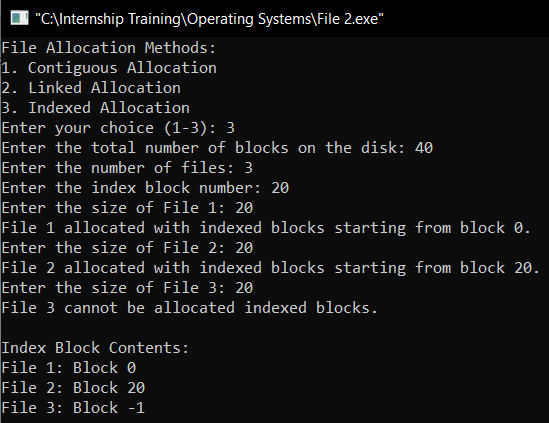
printf("File %d: Block %d\n", i + 1, index[i]);

}

}

**Outputs of The Code**

Contiguous Allocation

****

Indexed Allocation

Linked Allocation

**Sequence of Execution of Code**

***Header and Function Prototypes:***

The code starts with including necessary header files (stdio.h and stdlib.h).

Function prototypes are declared to inform the compiler about the functions that will be defined later in the code.

***Constants and File Structure:***

A constant MAX\_BLOCKS is defined to represent the maximum number of blocks on the disk.

A structure File is defined to represent a file, with attributes startBlock (the starting block on the disk) and size (the size of the file in blocks).

***Main Function:***

The main function is the entry point of the program.

The user is prompted to choose a file allocation method (1 for contiguous, 2 for linked, 3 for indexed).

Based on the user's choice, one of the three functions (contiguousAllocation, linkedAllocation, or indexedAllocation) is called.

***Contiguous Allocation (contiguous Allocation):***

The user is prompted to enter the total number of blocks on the disk and the number of files.

An array disk is initialized to represent the disk blocks, where 0 indicates a free block.

For each file, the user enters the file size, and the program attempts to find a contiguous block of the required size. If found, the file is allocated to those blocks.

***Linked Allocation (linked Allocation):***

Like contiguous allocation, the user inputs the total number of blocks and the number of files.

An array disk is initialized to represent the disk blocks, where -1 indicates a free block, and a linked list structure is used.

For each file, the program allocates linked blocks to the file until the required size is met.

***Indexed Allocation (indexed Allocation):***

The user inputs the total number of blocks, the number of files, and the index block number.

Arrays disk and index are initialized to represent the disk blocks and the index block, respectively.

For each file, the program allocates blocks for the file and updates the index block with the starting block of each file.

***Display Results:***

After each allocation method, the program displays the allocated blocks or linked list information to inform the user about the storage allocation.

End of Program:

The program ends, and the user is returned to the operating system.

**Algorithms Used: -**

Below is an overview of the algorithms used in each file allocation method:

***Contiguous Allocation:***

Algorithm: First Fit

Explanation: The program uses the "First Fit" algorithm to find a contiguous block of free space on the disk that is large enough to accommodate the file. It iterates through the disk blocks and allocates the file to the first available contiguous block that meets the size requirement.

***Linked Allocation:***

Algorithm: First Fit (for linked list allocation)

Explanation: The linked allocation method involves using a linked list to allocate non-contiguous blocks to a file. The program uses the "First Fit" algorithm to find free blocks on the disk and links them together to form a linked list for each file.

***Indexed Allocation:***

Algorithm: First Fit (for data block allocation), Sequential Search (for updating the index block)

Explanation: In indexed allocation, a separate index block is used to store the addresses of the data blocks that make up the file. The program uses the "First Fit" algorithm to find free data blocks for the file. It then uses a sequential search to update the index block with the starting block of each file.

**Advantages and Disadvantages of The Different File Allocation Methods**

Below is a summary of the pros and cons of three common file allocation methods: Contiguous Allocation, Linked Allocation, and Indexed Allocation.

***Contiguous Allocation:***

***Advantages:***

Sequential Access: Contiguous allocation is well-suited for applications that require sequential access to data because the blocks are stored consecutively on the disk.

Simple Implementation: It is relatively simple to implement and understand, making it efficient for small file systems.

No Overhead: There is no need for extra data structures to manage file locations.

***Disadvantages:***

Fragmentation: Contiguous allocation can suffer from both external and internal fragmentation. External fragmentation occurs when free space is scattered throughout the disk, and internal fragmentation arises when the allocated block size is larger than the file size.

Dynamic Storage Management: It is challenging to efficiently manage dynamic storage allocation and deal with file deletions or modifications.

***Linked Allocation:***

***Advantages:***

Dynamic Storage: Linked allocation allows for dynamic storage allocation, making it more flexible than contiguous allocation.

No Fragmentation: Linked allocation avoids external fragmentation because files are not required to be stored in contiguous blocks.

***Disadvantages:***

Random Access: Random access to data is less efficient compared to contiguous allocation due to the non-contiguous storage of blocks.

Overhead: Additional space is required for storing pointers in each block, leading to overhead in terms of storage space.

Complexity: Managing the linked list structure can be more complex, and traversing the list can be time-consuming for large files.

***Indexed Allocation:***

***Advantages:***

Random Access: Indexed allocation provides efficient random access to data as the index block contains direct pointers to each block of the file.

No External Fragmentation: Like linked allocation, indexed allocation avoids external fragmentation.

***Disadvantages:***

Index Block Overhead: The index block introduces additional overhead in terms of storage space.

Limited Block Size: The number of pointers in the index block limits the file size, and if the file grows beyond this limit, additional levels of indirection may be needed, leading to increased complexity.

Wasted Space: There can be wasted space in the last block of a file if it is not fully utilized.

General Considerations:

File Size and Access Patterns: The choice of allocation method often depends on the characteristics of the files, such as their sizes and access patterns.

Efficiency and Performance: The efficiency of each method can vary based on the specific requirements and workload of the system.

Dynamic Changes: The ability to handle dynamic changes, such as file deletions or modifications, is an important factor in the choice of allocation method.

**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**

***Definition:***

Deadlock in the context of operating systems refers to a situation where two or more processes are unable to proceed because each is waiting for the other to release a resource. Essentially, each process is stuck, and the system as a whole cannot make progress.

***Effects on System Performance:***

*Resource Utilization*: Deadlocks lead to inefficient resource utilization, as resources are held by processes that are unable to make progress.

*System Halting*: The entire system may come to a halt, as the processes involved in the deadlock cannot be terminated without releasing the resources they are holding.

***Examples:***

Consider two processes, A and B, where A holds a resource X and is waiting for resource Y, and B holds resource Y and is waiting for resource X. If both processes reach this state simultaneously, a deadlock occurs.

***Strategies to Address and Prevent Deadlocks:***

***Deadlock Avoidance:*** Use a resource allocation strategy that ensures that the system will not enter an unsafe state where deadlock can occur. This may involve dynamic checks and allocation decisions.

***Resource Allocation Graph:*** Employ techniques like the Resource Allocation Graph to detect and prevent circular wait conditions.

***Timeouts and Pre-emption:*** Introduce timeouts for resource requests and pre-empt resources from processes if needed, breaking the potential deadlock.

***Resource Allocation Policies:*** Define policies for resource allocation that minimize the possibility of circular waits.

***Deadlock Detection and Recovery:*** Implement algorithms for detecting deadlocks and recovering from them by releasing resources.

***Starvation:***

***Definition:***

Starvation occurs when a process is perpetually denied necessary resources and is unable to make progress. It is not a complete system halting, but specific processes are prevented from executing.

***Effects on System Performance:***

Decreased Throughput: Starvation reduces the overall throughput of the system as certain processes are not able to execute and complete their tasks.

Unfair Resource Distribution: Some processes may receive resources consistently, while others are denied, leading to an unfair distribution of resources.

***Examples:***

Consider a scenario where a high-priority process constantly acquires resources, preventing lower-priority processes from obtaining the resources they need. The lower-priority processes may starve, as they are consistently deprioritized.

***Strategies to Address and Prevent Starvation:***

***Priority Scheduling:*** Implement priority-based scheduling algorithms to ensure that all processes have an opportunity to access resources, with higher-priority processes getting preference.

***Aging:*** Introduce aging mechanisms where the priority of a process increases the longer it has been waiting for a resource. This prevents indefinite postponement.

***Fair Share Scheduling:*** Allocate resources based on the fair share principle, ensuring that each process gets a fair share of resources over time.

***Resource Quotas:*** Implement resource quotas to guarantee that each process gets a minimum number of resources within a specified time frame.

***Summary:***

***Deadlock:*** Processes are stuck in a circular waiting state, unable to proceed.

***Starvation:*** Processes are denied necessary resources and are unable to make progress, potentially leading to reduced system throughput.

***Prevention Strategies:*** Deadlock avoidance, resource allocation policies, timeouts, pre-emption, and detection and recovery for deadlocks. For starvation, priority scheduling, aging, fair share scheduling, and resource quotas can be employed.

In summary, while deadlocks bring the entire system to a standstill, starvation affects individual processes, reducing overall system throughput. Effective resource allocation policies and scheduling strategies are crucial for preventing and mitigating both deadlock and starvation scenarios in operating systems.