**Objectives:**

The primary objective of the program is to simulate basic directory and file management in a simplified operating system environment. It allows users to create a directory, add file names, and prevent duplicate file entries within that directory. The program also checks if a file name already exists before adding a new one, ensuring the integrity of file names. Additionally, it restricts the number of files to a set limit, preventing overflow or excessive file entries. This simulation serves as an educational tool to understand file allocation concepts.

1. **WAP to simulate the following contiguous memory allocation technique:**
2. **Worst-fit**

**Code:**

#include<stdio.h>

#define max 25

int main() {

int frag[max], b[max], f[max], i, j, nb, nf, temp, highest = 0;

static int bf[max], ff[max];

printf("\n\tMemory Management Scheme - Worst Fit");

printf("\nEnter the number of blocks:");

scanf("%d", &nb);

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

scanf("%d", &nf);

printf("\nEnter the size of the blocks:-\n");

for(i = 1; i <= nb; i++) {

printf("Block %d:", i);

scanf("%d", &b[i]);

}

printf("Enter the size of the files :-\n");

for(i = 1; i <= nf; i++) {

printf("File %d:", i);

scanf("%d", &f[i]);

}

// Worst Fit Allocation Logic

for(i = 1; i <= nf; i++) {

highest = -1;

for(j = 1; j <= nb; j++) {

if(bf[j] != 1) { // Block is free

temp = b[j] - f[i];

if(temp >= 0) { // Block can accommodate the file

if(highest == -1 || b[highest] < b[j]) {

highest = j; // Track the block with the largest size

}

}

}

}

if(highest != -1) { // Block found

ff[i] = highest;

frag[i] = b[highest] - f[i];

bf[highest] = 1; // Mark block as filled

} else {

ff[i] = -1; // No suitable block found

frag[i] = -1;

}

}

// Output result

printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragment");

for(i = 1; i <= nf; i++) {

if(ff[i] != -1)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);

else

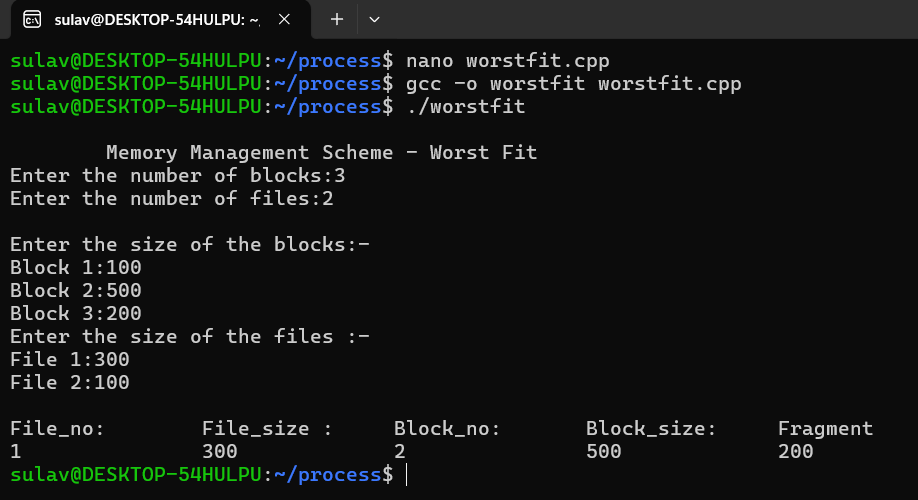
printf("\n%d\t\t%d\t\tNot Allocated", i, f[i]);

}

return 0;

}

**Output:**

****

1. **Best-fit**

**Code:**

#include<stdio.h>

#define max 25

int main() {

int frag[max], b[max], f[max], i, j, nb, nf, temp, lowest;

static int bf[max], ff[max]; // Initialize to 0

printf("\n\tMemory Management Scheme - Best Fit");

// Input number of blocks and files

printf("\nEnter the number of blocks: ");

scanf("%d", &nb);

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

scanf("%d", &nf);

// Input block sizes

printf("\nEnter the size of the blocks:\n");

for(i = 1; i <= nb; i++) {

printf("Block %d: ", i);

scanf("%d", &b[i]);

}

// Input file sizes

printf("Enter the size of the files:\n");

for(i = 1; i <= nf; i++) {

printf("File %d: ", i);

scanf("%d", &f[i]);

}

// Best Fit Allocation Logic

for(i = 1; i <= nf; i++) {

lowest = 10000; // Reset lowest for each file

for(j = 1; j <= nb; j++) {

if(bf[j] != 1) { // Block is free

temp = b[j] - f[i];

if(temp >= 0 && temp < lowest) { // Block can accommodate the file

ff[i] = j; // Track the block with the least remaining space

lowest = temp;

}

}

}

// If a suitable block is found

if(lowest != 10000) {

frag[i] = lowest; // Store the fragmentation

bf[ff[i]] = 1; // Mark block as filled

} else {

ff[i] = -1; // No suitable block found

frag[i] = -1;

}

}

// Output the result

printf("\nFile No\tFile Size\tBlock No\tBlock Size\tFragment");

for(i = 1; i <= nf; i++) {

if(ff[i] != -1)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);

else

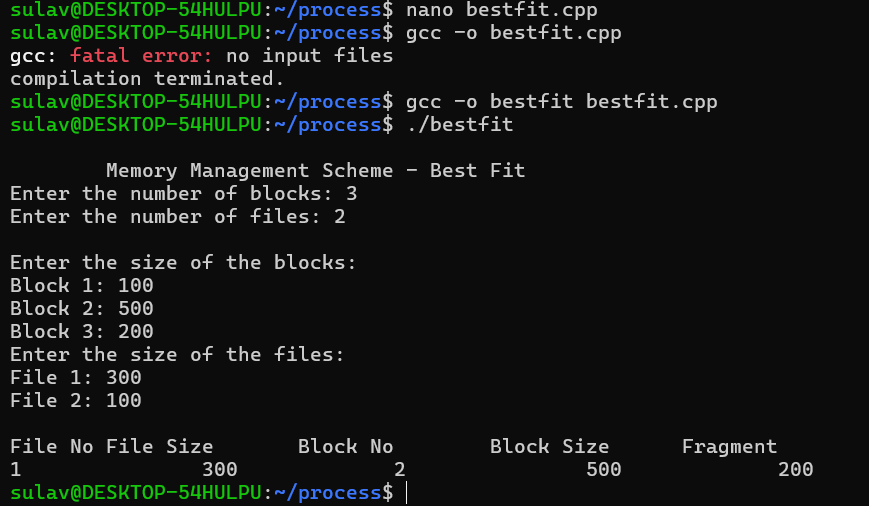
printf("\n%d\t\t%d\t\tNot Allocated", i, f[i]);

}

return 0;

}

**Output:**

****

1. **First-fit**

**Code:**

#include<stdio.h>

int main() {

int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

// Initialize the flags and allocation arrays

for(i = 0; i < 10; i++) {

flags[i] = 0; // Block is initially unallocated

allocation[i] = -1; // No process is allocated to this block

}

// Input for number of blocks and their sizes

printf("Enter number of blocks: ");

scanf("%d", &bno);

printf("Enter size of each block: ");

for(i = 0; i < bno; i++)

scanf("%d", &bsize[i]);

// Input for number of processes and their sizes

printf("\nEnter number of processes: ");

scanf("%d", &pno);

printf("Enter size of each process: ");

for(i = 0; i < pno; i++)

scanf("%d", &psize[i]);

// First Fit Allocation Logic

for(i = 0; i < pno; i++) { // For each process

for(j = 0; j < bno; j++) { // Find the first available block

if(flags[j] == 0 && bsize[j] >= psize[i]) { // Block is free and can fit the process

allocation[j] = i; // Allocate process i to block j

flags[j] = 1; // Mark block as allocated

break; // Move to the next process

}

}

}

// Display allocation details

printf("\nBlock No.\tBlock Size\tProcess No.\tProcess Size");

for(i = 0; i < bno; i++) {

printf("\n%d\t\t%d\t\t", i+1, bsize[i]);

if(flags[i] == 1) // Block is allocated

printf("%d\t\t\t%d", allocation[i] + 1, psize[allocation[i]]);

else // Block is not allocated

printf("Not allocated");

}

// Wait for user input before exiting

printf("\nPress Enter to exit...");

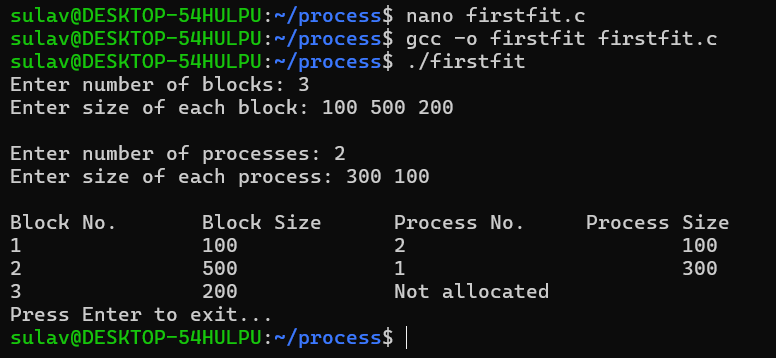
getchar(); // To consume the newline from previous input

getchar(); // Wait for Enter key

return 0;

}

**Output:**

****

1. **WAP to implement sequential file allocation technique.**

**Code:**

#include<stdio.h>

int main() {

int f[50], i, st, len, j, c, k, count = 0;

// Initialize all blocks to 0 (free)

for(i = 0; i < 50; i++)

f[i] = 0;

printf("Files Allocated are:\n");

// Start allocation

do {

count = 0;

// Get user input for starting block and length

printf("\nEnter the starting block and length of file: ");

scanf("%d%d", &st, &len);

// Check if the blocks from st to st+len are free

for(k = st; k < (st + len); k++) {

if(f[k] == 0)

count++;

}

// If all required blocks are free, allocate them

if(len == count) {

for(j = st; j < (st + len); j++) {

f[j] = 1; // Mark block as allocated

printf("Block %d is allocated.\n", j);

}

printf("The file is successfully allocated to the disk.\n");

} else {

printf("The file cannot be allocated. Not enough continuous free blocks.\n");

}

// Ask if the user wants to allocate more files

printf("Do you want to allocate another file? (Yes - 1 / No - 0): ");

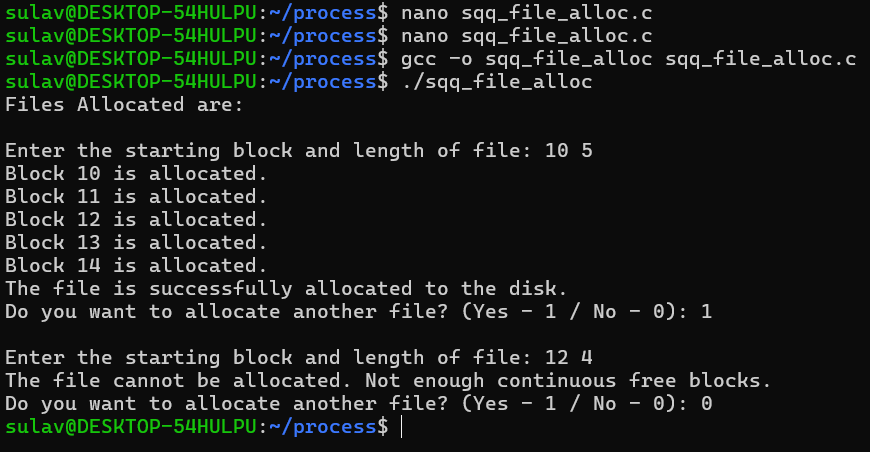
scanf("%d", &c);

} while(c == 1);

return 0;

}

**Output:**

****

1. **WAP to implement linked list file allocation technique.**

**Code:**

#include<stdio.h>

#include<stdlib.h>

int main() {

int f[50], p, i, st, len, j, c, k, a;

// Initialize the block allocation array (all free initially)

for(i = 0; i < 50; i++)

f[i] = 0;

// Input the number of already allocated blocks

printf("Enter how many blocks are already allocated: ");

scanf("%d", &p);

// Mark the already allocated blocks

printf("Enter the block numbers already allocated: ");

for(i = 0; i < p; i++) {

scanf("%d", &a);

if(a >= 0 && a < 50) // Ensure the block number is within valid range

f[a] = 1;

else

printf("Invalid block number %d. Must be between 0 and 49.\n", a);

}

// Main allocation loop

do {

// Input the starting block and the length of the file

printf("Enter starting block and length of file: ");

scanf("%d%d", &st, &len);

// Check if the starting block is free

if(st >= 0 && st + len <= 50) { // Ensure allocation stays within bounds

if(f[st] == 0) {

int allocated = 1;

// Try allocating the requested length

for(j = st; j < (st + len); j++) {

if(f[j] == 0) {

f[j] = 1; // Mark the block as allocated

printf("Block %d is allocated.\n", j);

} else {

printf("Block %d is already allocated.\n", j);

allocated = 0; // Mark the file as partially allocated

}

}

if(allocated) {

printf("The file has been successfully allocated.\n");

} else {

printf("The file could not be fully allocated due to some blocks already being taken.\n");

}

} else {

printf("Starting block %d is already allocated.\n", st);

}

} else {

printf("Invalid allocation range. Please ensure the file fits within the available blocks (0 to 49).\n");

}

// Ask the user if they want to allocate another file

printf("Do you want to allocate another file? (Yes - 1 / No - 0): ");

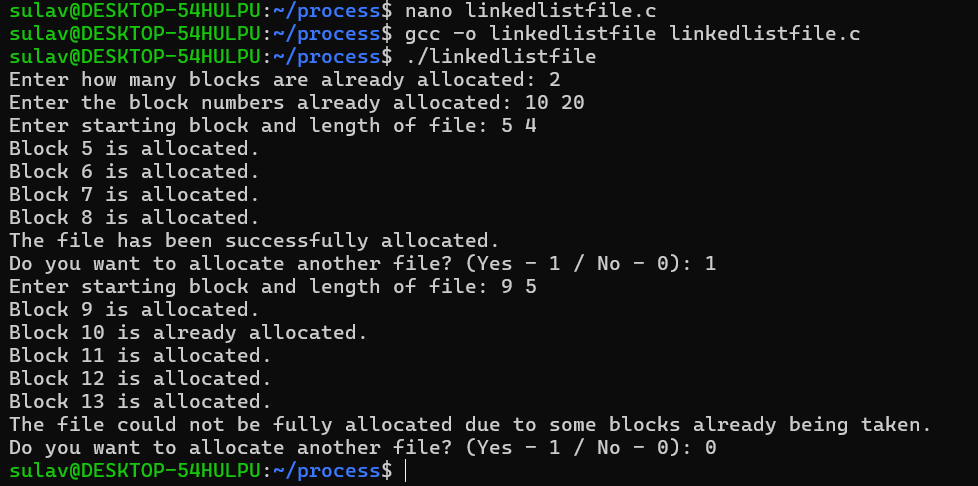
scanf("%d", &c);

} while(c == 1);

return 0;

}

**Output:**

****

1. **WAP to implement single level directory.**

**Code:**

#include<stdio.h>

#include<string.h>

int main() {

int nf = 0, i = 0, j = 0, ch;

char mdname[10], fname[10][10], name[10];

// Input the directory name

printf("Enter the directory name: ");

scanf("%s", mdname);

// User decides how many files they want to create

printf("Enter the number of files to create initially: ");

scanf("%d", &nf);

// File creation loop

do {

// Input the name of the file to be created

printf("Enter file name to be created: ");

scanf("%s", name);

// Check if the file already exists

for(i = 0; i < j; i++) {

if(!strcmp(name, fname[i])) {

printf("There is already a file named %s\n", name);

break;

}

}

// If the file doesn't already exist, add it

if(i == j && j < 10) { // Check if there is space for new files

strcpy(fname[j++], name);

} else if(j >= 10) {

printf("Directory is full, cannot create more files.\n");

}

// Ask user if they want to add another file

printf("Do you want to enter another file (yes - 1 or no - 0): ");

scanf("%d", &ch);

} while(ch == 1);

// Output the directory and files

printf("\nDirectory name is: %s\n", mdname);

printf("Files in the directory:");

for(i = 0; i < j; i++) {

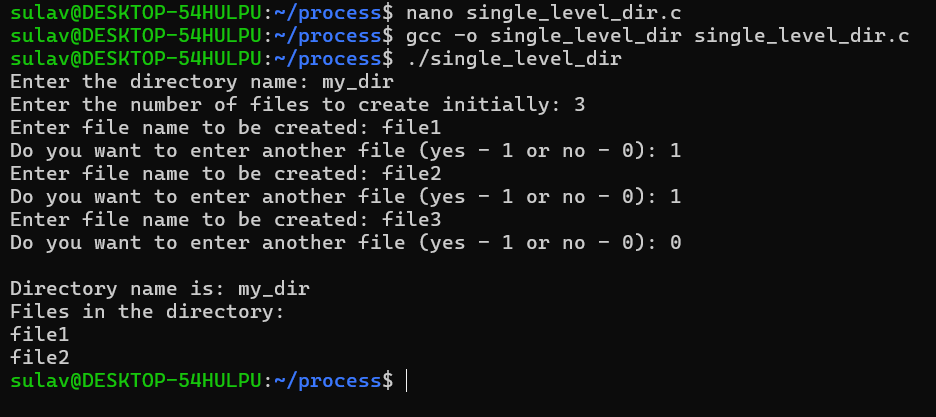
printf("\n%s", fname[i]);

}

return 0;

}

**Output:**

****

**Conclusion:**

In conclusion, the program successfully demonstrates the fundamental workings of directory management, such as file creation and duplicate prevention. By allowing users to input and manage files within a directory, it illustrates how file systems handle file allocation and error checking. The use of conditions to check for duplicate files and the limit on file entries ensures smooth operation, mimicking real-world constraints. This basic file management system serves as an effective learning model for understanding essential concepts in operating systems. It highlights the importance of efficient file handling in maintaining a reliable directory structure.