**Q1.SHARED MEMORY**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <sys/types.h>**

**#include <sys/ipc.h>**

**#include <sys/shm.h>**

**#define SHM\_SIZE 1024**

**int main() {**

**key\_t key = ftok("shared\_memory\_example", 'R'); // Generate a unique key**

**int shmid;**

**// Create a shared memory segment**

**if ((shmid = shmget(key, SHM\_SIZE, IPC\_CREAT | 0666)) < 0) {**

**perror("shmget");**

**exit(1);**

**}**

**// Attach the shared memory segment to the process's address space**

**char \*shm = shmat(shmid, NULL, 0);**

**if (shm == (char \*) -1) {**

**perror("shmat");**

**exit(1);**

**}**

**// Writing data to shared memory**

**sprintf(shm, "Hello, shared memory!");**

**// Fork a new process**

**pid\_t pid = fork();**

**if (pid == -1) {**

**perror("fork");**

**exit(1);**

**} else if (pid == 0) { // Child process**

**// Child reads from shared memory**

**printf("Child Process: Received message from parent: %s\n", shm);**

**// Detach from shared memory**

**if (shmdt(shm) == -1) {**

**perror("shmdt");**

**exit(1);**

**}**

**} else { // Parent process**

**// Parent waits for the child to finish**

**wait(NULL);**

**// Parent reads from shared memory**

**printf("Parent Process: Received message from child: %s\n", shm);**

**// Detach from shared memory**

**if (shmdt(shm) == -1) {**

**perror("shmdt");**

**exit(1);**

**}**

**// Remove the shared memory segment**

**shmctl(shmid, IPC\_RMID, NULL);**

**}**

**return 0;**

**}**

**OUTPUT: Child Process: Received message from parent: Hello, shared memory!**

**Parent Process: Received message from child: Hello, shared memory!**

**Q2.MESSAGE QUENE**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <sys/types.h>**

**#include <sys/ipc.h>**

**#include <sys/msg.h>**

**#define MESSAGE\_SIZE 128**

**// Define the message structure**

**struct msg\_buffer {**

**long msg\_type;**

**char msg\_text[MESSAGE\_SIZE];**

**};**

**int main() {**

**key\_t key = ftok("message\_queue\_example", 'R'); // Generate a unique key**

**int msgid;**

**// Create a message queue**

**if ((msgid = msgget(key, IPC\_CREAT | 0666)) == -1) {**

**perror("msgget");**

**exit(1);**

**}**

**struct msg\_buffer message;**

**// Fork a new process**

**pid\_t pid = fork();**

**if (pid == -1) {**

**perror("fork");**

**exit(1);**

**} else if (pid == 0) { // Child process**

**// Child receives message from the queue**

**if (msgrcv(msgid, &message, sizeof(message.msg\_text), 1, 0) == -1) {**

**perror("msgrcv");**

**exit(1);**

**}**

**printf("Child Process: Received message from parent: %s\n", message.msg\_text);**

**} else { // Parent process**

**// Parent sends a message to the queue**

**message.msg\_type = 1;**

**strcpy(message.msg\_text, "Hello, message queue!");**

**if (msgsnd(msgid, &message, sizeof(message.msg\_text), 0) == -1) {**

**perror("msgsnd");**

**exit(1);**

**}**

**// Wait for the child to finish**

**wait(NULL);**

**// Remove the message queue**

**if (msgctl(msgid, IPC\_RMID, NULL) == -1) {**

**perror("msgctl");**

**exit(1);**

**}**

**}**

**return 0;**

**}**

**OUTPUT: Child Process: Received message from parent: Hello, message queue!**

**Q3.multithtreading**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <pthread.h>**

**#define NUM\_THREADS 3**

**// Function that each thread will execute**

**void \*print\_hello(void \*thread\_id) {**

**long tid = (long)thread\_id;**

**printf("Hello from Thread %ld\n", tid);**

**pthread\_exit(NULL);**

**}**

**int main() {**

**pthread\_t threads[NUM\_THREADS];**

**int rc;**

**long t;**

**// Create multiple threads**

**for (t = 0; t < NUM\_THREADS; t++) {**

**printf("Creating thread %ld\n", t);**

**rc = pthread\_create(&threads[t], NULL, print\_hello, (void \*)t);**

**if (rc) {**

**printf("Error creating thread %ld\n", t);**

**exit(-1);**

**}**

**}**

**// Wait for all threads to finish**

**for (t = 0; t < NUM\_THREADS; t++) {**

**rc = pthread\_join(threads[t], NULL);**

**if (rc) {**

**printf("Error joining thread %ld\n", t);**

**exit(-1);**

**}**

**}**

**printf("All threads have completed.\n");**

**pthread\_exit(NULL);**

**return 0;**

**}**

**Output: rbin/ld: /tmp/ccUA2NV4.o: in function `main':**

**ASALo8wCkL.c:(.text+0x7e): undefined reference to `pthread\_create'**

**/rbin/ld: ASALo8wCkL.c:(.text+0xd3): undefined reference to `pthread\_join'**

**collect2: error: ld returned 1 exit status**

**Q4. Dining-Philosophers problem**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <pthread.h>**

**#include <unistd.h>**

**#define NUM\_PHILOSOPHERS 5**

**pthread\_mutex\_t forks[NUM\_PHILOSOPHERS];**

**pthread\_t philosophers[NUM\_PHILOSOPHERS];**

**// Function prototypes**

**void \*philosopher(void \*arg);**

**void pickup\_forks(int philosopher\_id);**

**void putdown\_forks(int philosopher\_id);**

**int main() {**

**// Initialize mutexes for each fork**

**for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {**

**pthread\_mutex\_init(&forks[i], NULL);**

**}**

**// Create threads for each philosopher**

**for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {**

**int \*philosopher\_id = malloc(sizeof(int));**

**\*philosopher\_id = i;**

**pthread\_create(&philosophers[i], NULL, philosopher, (void \*)philosopher\_id);**

**}**

**// Join threads**

**for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {**

**pthread\_join(philosophers[i], NULL);**

**}**

**// Destroy mutexes**

**for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {**

**pthread\_mutex\_destroy(&forks[i]);**

**}**

**return 0;**

**}**

**void \*philosopher(void \*arg) {**

**int philosopher\_id = \*((int \*)arg);**

**free(arg);**

**while (1) {**

**printf("Philosopher %d is thinking.\n", philosopher\_id);**

**sleep(rand() % 5); // Simulate thinking**

**pickup\_forks(philosopher\_id);**

**printf("Philosopher %d is eating.\n", philosopher\_id);**

**sleep(rand() % 5); // Simulate eating**

**putdown\_forks(philosopher\_id);**

**}**

**pthread\_exit(NULL);**

**}**

**void pickup\_forks(int philosopher\_id) {**

**int right\_fork = (philosopher\_id + 1) % NUM\_PHILOSOPHERS;**

**int left\_fork = philosopher\_id;**

**// Lock the smaller index first to prevent deadlock**

**if (left\_fork < right\_fork) {**

**pthread\_mutex\_lock(&forks[left\_fork]);**

**pthread\_mutex\_lock(&forks[right\_fork]);**

**} else {**

**pthread\_mutex\_lock(&forks[right\_fork]);**

**pthread\_mutex\_lock(&forks[left\_fork]);**

**}**

**}**

**void putdown\_forks(int philosopher\_id) {**

**int right\_fork = (philosopher\_id + 1) % NUM\_PHILOSOPHERS;**

**int left\_fork = philosopher\_id;**

**pthread\_mutex\_unlock(&forks[left\_fork]);**

**pthread\_mutex\_unlock(&forks[right\_fork]);**

**}**

**OUTPUT:**

**gcc /tmp/ASALo8wCkL.c -lm**

**/rbin/ld: /tmp/cc7d869B.o: in function `main':**

**ASALo8wCkL.c:(.text+0x84): undefined reference to `pthread\_create'**

**/rbin/ld: ASALo8wCkL.c:(.text+0xb1): undefined reference to `pthread\_join'**

**collect2: error: ld returned 1 exit status**

**Q5.MEMORY ALLOCATION**

**#include <stdio.h>**

**#include <stdlib.h>**

**// Structure to represent a memory block**

**typedef struct {**

**int size;**

**int allocated; // 1 if allocated, 0 if free**

**} MemoryBlock;**

**#define MEMORY\_SIZE 100**

**MemoryBlock memory[MEMORY\_SIZE];**

**// Function prototypes**

**void initializeMemory();**

**void displayMemory();**

**int firstFit(int requestSize);**

**int bestFit(int requestSize);**

**int worstFit(int requestSize);**

**int main() {**

**initializeMemory();**

**// Example usage**

**int allocatedBlock1 = firstFit(20);**

**int allocatedBlock2 = bestFit(30);**

**int allocatedBlock3 = worstFit(15);**

**displayMemory();**

**// Deallocate blocks**

**free(memory[allocatedBlock1].allocated);**

**free(memory[allocatedBlock2].allocated);**

**free(memory[allocatedBlock3].allocated);**

**displayMemory();**

**return 0;**

**}**

**void initializeMemory() {**

**// Initialize memory blocks with random sizes**

**for (int i = 0; i < MEMORY\_SIZE; i++) {**

**memory[i].size = rand() % 20 + 1; // Random size between 1 and 20**

**memory[i].allocated = 0; // 0 for free, 1 for allocated**

**}**

**}**

**void displayMemory() {**

**printf("Memory Status:\n");**

**for (int i = 0; i < MEMORY\_SIZE; i++) {**

**printf("Block %2d - Size: %2d - %s\n", i, memory[i].size,**

**memory[i].allocated ? "Allocated" : "Free");**

**}**

**printf("\n");**

**}**

**int firstFit(int requestSize) {**

**for (int i = 0; i < MEMORY\_SIZE; i++) {**

**if (!memory[i].allocated && memory[i].size >= requestSize) {**

**memory[i].allocated = 1;**

**return i;**

**}**

**}**

**return -1; // No suitable block found**

**}**

**int bestFit(int requestSize) {**

**int bestFitIndex = -1;**

**int bestFitSize = MEMORY\_SIZE + 1;**

**for (int i = 0; i < MEMORY\_SIZE; i++) {**

**if (!memory[i].allocated && memory[i].size >= requestSize && memory[i].size < bestFitSize) {**

**bestFitIndex = i;**

**bestFitSize = memory[i].size;**

**}**

**}**

**if (bestFitIndex != -1) {**

**memory[bestFitIndex].allocated = 1;**

**}**

**return bestFitIndex;**

**}**

**int worstFit(int requestSize) {**

**int worstFitIndex = -1;**

**int worstFitSize = -1;**

**for (int i = 0; i < MEMORY\_SIZE; i++) {**

**if (!memory[i].allocated && memory[i].size >= requestSize && memory[i].size > worstFitSize) {**

**worstFitIndex = i;**

**worstFitSize = memory[i].size;**

**}**

**}**

**if (worstFitIndex != -1) {**

**memory[worstFitIndex].allocated = 1;**

**}**

**return worstFitIndex;**

**}**

**OUTPUT: emory Status:**

**Block 0 - Size: 4 - Free**

**Block 1 - Size: 7 - Free**

**Block 2 - Size: 18 - Free**

**Block 3 - Size: 16 - Free**

**Block 4 - Size: 14 - Free**

**Block 5 - Size: 16 - Free**

**Block 6 - Size: 7 - Free**

**Block 7 - Size: 13 - Free**

**Block 8 - Size: 10 - Free**

**Block 9 - Size: 2 - Free**

**Block 10 - Size: 3 - Free**

**Block 11 - Size: 8 - Free**

**Block 12 - Size: 11 - Free**

**Block 13 - Size: 20 - Allocated**

**Block 14 - Size: 4 - Free**

**Block 15 - Size: 7 - Free**

**Block 16 - Size: 1 - Free**

**Block 17 - Size: 7 - Free**

**Block 18 - Size: 13 - Free**

**Block 19 - Size: 17 - Free**

**Block 20 - Size: 12 - Free**

**Block 21 - Size: 9 - Free**

**Block 22 - Size: 8 - Free**

**Block 23 - Size: 10 - Free**

**Block 24 - Size: 3 - Free**

**Block 25 - Size: 11 - Free**

**Block 26 - Size: 3 - Free**

**Block 27 - Size: 4 - Free**

**Block 28 - Size: 8 - Free**

**Block 29 - Size: 16 - Free**

**Block 30 - Size: 10 - Free**

**Block 31 - Size: 3 - Free**

**Block 32 - Size: 3 - Free**

**Block 33 - Size: 19 - Free**

**Block 34 - Size: 10 - Free**

**Block 35 - Size: 8 - Free**

**Block 36 - Size: 14 - Free**

**Block 37 - Size: 17 - Free**

**Block 38 - Size: 12 - Free**

**Block 39 - Size: 3 - Free**

**Block 40 - Size: 10 - Free**

**Block 41 - Size: 14 - Free**

**Block 42 - Size: 2 - Free**

**Block 43 - Size: 20 - Allocated**

**Block 44 - Size: 5 - Free**

**Block 45 - Size: 18 - Free**

**Block 46 - Size: 19 - Free**

**Block 47 - Size: 5 - Free**

**Block 48 - Size: 16 - Free**

**Block 49 - Size: 11 - Free**

**Block 50 - Size: 14 - Free**

**Block 51 - Size: 7 - Free**

**Block 52 - Size: 12 - Free**

**Block 53 - Size: 1 - Free**

**Block 54 - Size: 17 - Free**

**Block 55 - Size: 14 - Free**

**Block 56 - Size: 3 - Free**

**Block 57 - Size: 11 - Free**

**Block 58 - Size: 17 - Free**

**Block 59 - Size: 2 - Free**

**Block 60 - Size: 6 - Free**

**Block 61 - Size: 6 - Free**

**Block 62 - Size: 5 - Free**

**Block 63 - Size: 8 - Free**

**Block 64 - Size: 17 - Free**

**Block 65 - Size: 6 - Free**

**Block 66 - Size: 7 - Free**

**Block 67 - Size: 10 - Free**

**Block 68 - Size: 14 - Free**

**Block 69 - Size: 18 - Free**

**Block 70 - Size: 5 - Free**

**Block 71 - Size: 16 - Free**

**Block 72 - Size: 3 - Free**

**Block 73 - Size: 6 - Free**

**Block 74 - Size: 15 - Free**

**Block 75 - Size: 8 - Free**

**Block 76 - Size: 15 - Free**

**Block 77 - Size: 5 - Free**

**Block 78 - Size: 4 - Free**

**Block 79 - Size: 11 - Free**

**Block 80 - Size: 8 - Free**

**Block 81 - Size: 9 - Free**

**Block 82 - Size: 17 - Free**

**Block 83 - Size: 19 - Free**

**Block 84 - Size: 9 - Free**

**Block 85 - Size: 5 - Free**

**Block 86 - Size: 4 - Free**

**Block 87 - Size: 12 - Free**

**Block 88 - Size: 15 - Free**

**Block 89 - Size: 20 - Free**

**Block 90 - Size: 13 - Free**

**Block 91 - Size: 1 - Free**

**Block 92 - Size: 17 - Free**

**Block 93 - Size: 9 - Free**

**Block 94 - Size: 20 - Free**

**Block 95 - Size: 13 - Free**

**Block 96 - Size: 7 - Free**

**Block 97 - Size: 7 - Free**

**Block 98 - Size: 15 - Free**

**Block 99 - Size: 20 - Free**

**Segmentation fault**

**Q6.SINGLE LEVEL DIRECTORY**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#define MAX\_FILES 10**

**#define MAX\_FILE\_NAME 20**

**// Structure to represent a file**

**typedef struct {**

**char name[MAX\_FILE\_NAME];**

**} File;**

**File files[MAX\_FILES]; // Array to store files**

**int fileCount = 0; // Number of files in the directory**

**// Function prototypes**

**void createFile(const char \*fileName);**

**void deleteFile(const char \*fileName);**

**void listFiles();**

**int main() {**

**int choice;**

**char fileName[MAX\_FILE\_NAME];**

**while (1) {**

**printf("\nFile Organization Menu:\n");**

**printf("1. Create File\n");**

**printf("2. Delete File\n");**

**printf("3. List Files\n");**

**printf("4. Exit\n");**

**printf("Enter your choice: ");**

**scanf("%d", &choice);**

**switch (choice) {**

**case 1:**

**printf("Enter the name of the file to create: ");**

**scanf("%s", fileName);**

**createFile(fileName);**

**break;**

**case 2:**

**printf("Enter the name of the file to delete: ");**

**scanf("%s", fileName);**

**deleteFile(fileName);**

**break;**

**case 3:**

**listFiles();**

**break;**

**case 4:**

**printf("Exiting the program.\n");**

**exit(0);**

**default:**

**printf("Invalid choice. Please try again.\n");**

**}**

**}**

**return 0;**

**}**

**void createFile(const char \*fileName) {**

**if (fileCount < MAX\_FILES) {**

**// Check if the file already exists**

**for (int i = 0; i < fileCount; i++) {**

**if (strcmp(files[i].name, fileName) == 0) {**

**printf("Error: File '%s' already exists.\n", fileName);**

**return;**

**}**

**}**

**// Create the file**

**strcpy(files[fileCount].name, fileName);**

**fileCount++;**

**printf("File '%s' created successfully.\n", fileName);**

**} else {**

**printf("Error: Maximum number of files reached.\n");**

**}**

**}**

**void deleteFile(const char \*fileName) {**

**int found = 0;**

**// Find and delete the file**

**for (int i = 0; i < fileCount; i++) {**

**if (strcmp(files[i].name, fileName) == 0) {**

**// Shift remaining files to fill the gap**

**for (int j = i; j < fileCount - 1; j++) {**

**strcpy(files[j].name, files[j + 1].name);**

**}**

**fileCount--;**

**found = 1;**

**break;**

**}**

**}**

**if (found) {**

**printf("File '%s' deleted successfully.\n", fileName);**

**} else {**

**printf("Error: File '%s' not found.\n", fileName);**

**}**

**}**

**void listFiles() {**

**if (fileCount == 0) {**

**printf("No files in the directory.\n");**

**} else {**

**printf("List of Files:\n");**

**for (int i = 0; i < fileCount; i++) {**

**printf("%d. %s\n", i + 1, files[i].name);**

**}**

**}**

**}**

**OUTPUT: File Organization Menu:**

**1. Create File**

**2. Delete File**

**3. List Files**

**4. Exit**

**Enter your choice: 4**

**Exiting the program.**

**Q7.MULTI LEVEL**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#define MAX\_DIRECTORIES 10**

**#define MAX\_FILES\_PER\_DIRECTORY 10**

**#define MAX\_DIRECTORY\_NAME 20**

**#define MAX\_FILE\_NAME 20**

**// Structure to represent a file**

**typedef struct {**

**char name[MAX\_FILE\_NAME];**

**} File;**

**// Structure to represent a directory**

**typedef struct {**

**char name[MAX\_DIRECTORY\_NAME];**

**File files[MAX\_FILES\_PER\_DIRECTORY];**

**int fileCount;**

**} Directory;**

**Directory directories[MAX\_DIRECTORIES]; // Array to store directories**

**int directoryCount = 0; // Number of directories**

**// Function prototypes**

**void createDirectory(const char \*dirName);**

**void createFile(const char \*dirName, const char \*fileName);**

**void deleteFile(const char \*dirName, const char \*fileName);**

**void listFiles(const char \*dirName);**

**int main() {**

**int choice;**

**char dirName[MAX\_DIRECTORY\_NAME];**

**char fileName[MAX\_FILE\_NAME];**

**while (1) {**

**printf("\nFile Organization Menu:\n");**

**printf("1. Create Directory\n");**

**printf("2. Create File\n");**

**printf("3. Delete File\n");**

**printf("4. List Files\n");**

**printf("5. Exit\n");**

**printf("Enter your choice: ");**

**scanf("%d", &choice);**

**switch (choice) {**

**case 1:**

**printf("Enter the name of the directory to create: ");**

**scanf("%s", dirName);**

**createDirectory(dirName);**

**break;**

**case 2:**

**printf("Enter the name of the directory: ");**

**scanf("%s", dirName);**

**printf("Enter the name of the file to create: ");**

**scanf("%s", fileName);**

**createFile(dirName, fileName);**

**break;**

**case 3:**

**printf("Enter the name of the directory: ");**

**scanf("%s", dirName);**

**printf("Enter the name of the file to delete: ");**

**scanf("%s", fileName);**

**deleteFile(dirName, fileName);**

**break;**

**case 4:**

**printf("Enter the name of the directory: ");**

**scanf("%s", dirName);**

**listFiles(dirName);**

**break;**

**case 5:**

**printf("Exiting the program.\n");**

**exit(0);**

**default:**

**printf("Invalid choice. Please try again.\n");**

**}**

**}**

**return 0;**

**}**

**void createDirectory(const char \*dirName) {**

**if (directoryCount < MAX\_DIRECTORIES) {**

**// Check if the directory already exists**

**for (int i = 0; i < directoryCount; i++) {**

**if (strcmp(directories[i].name, dirName) == 0) {**

**printf("Error: Directory '%s' already exists.\n", dirName);**

**return;**

**}**

**}**

**// Create the directory**

**strcpy(directories[directoryCount].name, dirName);**

**directories[directoryCount].fileCount = 0;**

**directoryCount++;**

**printf("Directory '%s' created successfully.\n", dirName);**

**} else {**

**printf("Error: Maximum number of directories reached.\n");**

**}**

**}**

**void createFile(const char \*dirName, const char \*fileName) {**

**int dirIndex = -1;**

**// Find the directory**

**for (int i = 0; i < directoryCount; i++) {**

**if (strcmp(directories[i].name, dirName) == 0) {**

**dirIndex = i;**

**break;**

**}**

**}**

**if (dirIndex != -1) {**

**// Check if the file already exists in the directory**

**for (int i = 0; i < directories[dirIndex].fileCount; i++) {**

**if (strcmp(directories[dirIndex].files[i].name, fileName) == 0) {**

**printf("Error: File '%s' already exists in directory '%s'.\n", fileName, dirName);**

**return;**

**}**

**}**

**// Create the file in the directory**

**strcpy(directories[dirIndex].files[directories[dirIndex].fileCount].name, fileName);**

**directories[dirIndex].fileCount++;**

**printf("File '%s' created successfully in directory '%s'.\n", fileName, dirName);**

**} else {**

**printf("Error: Directory '%s' not found.\n", dirName);**

**}**

**}**

**void deleteFile(const char \*dirName, const char \*fileName) {**

**int dirIndex = -1;**

**// Find the directory**

**for (int i = 0; i < directoryCount; i++) {**

**if (strcmp(directories[i].name, dirName) == 0) {**

**dirIndex = i;**

**break;**

**}**

**}**

**if (dirIndex != -1) {**

**int found = 0;**

**// Find and delete the file in the directory**

**for (int i = 0; i < directories[dirIndex].fileCount; i++) {**

**if (strcmp(directories[dirIndex].files[i].name, fileName) == 0) {**

**// Shift remaining files to fill the gap**

**for (int j = i; j < directories[dirIndex].fileCount - 1; j++) {**

**strcpy(directories[dirIndex].files[j].name, directories[dirIndex].files[j + 1].name);**

**}**

**directories[dirIndex].fileCount--;**

**found = 1;**

**break;**

**}**

**}**

**if (found) {**

**printf("File '%s' deleted successfully from directory '%s'.\n", fileName, dirName);**

**} else {**

**printf("Error: File '%s' not found in directory '%s'.\n", fileName, dirName);**

**}**

**} else {**

**printf("Error: Directory '%s' not found.\n", dirName);**

**}**

**}**

**void listFiles(const char \*dirName) {**

**int dirIndex = -1;**

**// Find the directory**

**for (int i = 0; i < directoryCount; i++) {**

**if (strcmp(directories[i].name, dirName) == 0) {**

**dirIndex = i;**

**break;**

**}**

**}**

**if (dirIndex != -1) {**

**if (directories[dirIndex].fileCount == 0) {**

**printf("No files in directory '%s'.\n", dirName);**

**} else {**

**printf("List of Files in directory '%s':\n", dirName);**

**for (int i = 0; i < directories[dirIndex].fileCount; i++) {**

**printf("%d. %s\n", i + 1, directories[dirIndex].files[i].name);**

**}**

**}**

**} else {**

**printf("Error: Directory '%s' not found.\n", dirName);**

**}**

**}**

**OUTPUT:** **/tmp/ASALo8wCkL.o**

**File Organization Menu:**

**1. Create Directory**

**2. Create File**

**3. Delete File**

**4. List Files**

**5. Exit**

**Enter your choice: 5**

**Exiting the program.**

**Q8.RANDOM ACCESS**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define FILE\_NAME "employee\_records.dat"**

**// Structure to represent an employee**

**typedef struct {**

**int empId;**

**char empName[50];**

**float empSalary;**

**} Employee;**

**// Function prototypes**

**void addEmployee();**

**void displayAllEmployees();**

**void searchEmployee(int empId);**

**int main() {**

**int choice;**

**while (1) {**

**printf("\nEmployee Record Menu:\n");**

**printf("1. Add Employee\n");**

**printf("2. Display All Employees\n");**

**printf("3. Search Employee by ID\n");**

**printf("4. Exit\n");**

**printf("Enter your choice: ");**

**scanf("%d", &choice);**

**switch (choice) {**

**case 1:**

**addEmployee();**

**break;**

**case 2:**

**displayAllEmployees();**

**break;**

**case 3:**

**{**

**int empId;**

**printf("Enter Employee ID to search: ");**

**scanf("%d", &empId);**

**searchEmployee(empId);**

**}**

**break;**

**case 4:**

**printf("Exiting the program.\n");**

**exit(0);**

**default:**

**printf("Invalid choice. Please try again.\n");**

**}**

**}**

**return 0;**

**}**

**void addEmployee() {**

**Employee newEmployee;**

**FILE \*file = fopen(FILE\_NAME, "ab");**

**if (file == NULL) {**

**perror("Error opening file");**

**exit(EXIT\_FAILURE);**

**}**

**printf("Enter Employee ID: ");**

**scanf("%d", &newEmployee.empId);**

**printf("Enter Employee Name: ");**

**scanf("%s", newEmployee.empName);**

**printf("Enter Employee Salary: ");**

**scanf("%f", &newEmployee.empSalary);**

**// Write the new employee record to the file**

**fwrite(&newEmployee, sizeof(Employee), 1, file);**

**fclose(file);**

**printf("Employee added successfully.\n");**

**}**

**void displayAllEmployees() {**

**FILE \*file = fopen(FILE\_NAME, "rb");**

**if (file == NULL) {**

**perror("Error opening file");**

**exit(EXIT\_FAILURE);**

**}**

**Employee currentEmployee;**

**// Read and display all employee records in the file**

**printf("\nEmployee Records:\n");**

**while (fread(&currentEmployee, sizeof(Employee), 1, file) == 1) {**

**printf("Employee ID: %d\n", currentEmployee.empId);**

**printf("Employee Name: %s\n", currentEmployee.empName);**

**printf("Employee Salary: %.2f\n", currentEmployee.empSalary);**

**printf("\n");**

**}**

**fclose(file);**

**}**

**void searchEmployee(int empId) {**

**FILE \*file = fopen(FILE\_NAME, "rb");**

**if (file == NULL) {**

**perror("Error opening file");**

**exit(EXIT\_FAILURE);**

**}**

**Employee currentEmployee;**

**int found = 0;**

**// Search for the employee by ID**

**while (fread(&currentEmployee, sizeof(Employee), 1, file) == 1) {**

**if (currentEmployee.empId == empId) {**

**printf("Employee found:\n");**

**printf("Employee ID: %d\n", currentEmployee.empId);**

**printf("Employee Name: %s\n", currentEmployee.empName);**

**printf("Employee Salary: %.2f\n", currentEmployee.empSalary);**

**found = 1;**

**break;**

**}**

**}**

**if (!found) {**

**printf("Employee with ID %d not found.\n", empId);**

**}**

**fclose(file);**

**}**

**OUTPUT:** **/tmp/ASALo8wCkL.o**

**Employee Record Menu:**

**1. Add Employee**

**2. Display All Employees**

**3. Search Employee by ID**

**4. Exit**

**Enter your choice: 4**

**Exiting the program.**