



Name:S.karnesh

Reg.No:192421200

Course code:CSA0429

Course Name: Operating System

1.

The screenshot shows a C IDE interface with a code editor and an output window. The code in the editor is for managing employees using a binary file ('employee.dat'). It includes functions for adding employees and displaying them. The output window shows the execution of the program, which asks for a choice (Display Employees) and then lists the available options (Add Employee, Display Employees, Update Employee, Delete Employee, Exit). It also handles errors like 'File not found'.

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define FILE_NAME "employee.dat"
6
7 typedef struct {
8     int id;
9     char name[50];
10    float salary;
11 } Employee;
12
13 // Function to add a new employee
14 void addEmployee() {
15     FILE *fp = fopen(FILE_NAME, "ab"); // Append in binary mode
16     if (!fp) {
17         printf("Unable to open file!\n");
18         return;
19     }
20
21     Employee emp;
22     printf("Enter Employee ID: ");
23     scanf("%d", &emp.id);
24     printf("Enter Name: ");
25     scanf(" %[^\n]", emp.name); // Reads string with spaces
26     printf("Enter Salary: ");
27     scanf("%f", &emp.salary);
28
29     fwrite(&emp, sizeof(Employee), 1, fp);
30     fclose(fp);
31     printf("Employee added successfully!\n");
32 }
33
34 // Function to display all employees
35 void displayEmployees() {
36     FILE *fp = fopen(FILE_NAME, "rb");
37     if (!fp) {
38         printf("File not found!\n");
39         return;
40     }
41 }
```

Output

```
... Employee Management using Random Access File ...
1. Add Employee
2. Display Employees
3. Update Employee
4. Delete Employee
5. Exit
Enter your choice: 2
File not found!

... Employee Management using Random Access File ...
1. Add Employee
2. Display Employees
3. Update Employee
4. Delete Employee
5. Exit
Enter your choice: 1
Unable to open file!

... Employee Management using Random Access File ...
1. Add Employee
2. Display Employees
3. Update Employee
4. Delete Employee
5. Exit
Enter your choice:
```

2.

The screenshot shows a C IDE interface with a code editor and an output window. The code implements a message queue using the System V Message Queue interface. It creates a queue, forks into a receiver process (which reads messages), and a sender process (which sends a message and then deletes the queue). The output window shows the exchange between the parent and child processes.

```
main.c
1 #include <stdio.h>
2 #include <sys/ipc.h>
3 #include <sys/msg.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6 #include <string.h>
7
8 #define MAX 100
9
10 // Message structure
11 struct msg_buffer {
12     long msg_type;
13     char msg_text[MAX];
14 };
15
16 int main()
17 {
18     key_t key;
19     int msgid;
20     struct msg_buffer message;
21
22     // Generate unique key
23     key = ftok("progfile", 65);
24
25     // Create message queue
26     msgid = msgget(key, 0666 | IPC_CREAT);
27
28     if (fork() == 0)
29     {
30         // Child process: Receiver
31         msgrcv(msgid, &message, sizeof(message.msg_text), 1, 0);
32         printf("\nChild Process: Message Received = %s\n", message.msg_text);
33     }
34     else
35     {
36         // Parent process: Sender
37         message.msg_type = 1;
38         strcpy(message.msg_text, "Hello from Parent using Message Queue IPC");
39
40         msgsnd(msgid, &message, sizeof(message.msg_text), 0);
41         printf("Parent Process: Message Sent\n");
42     }
43 }
```

Output

```
Parent Process: Message Sent
Child Process: Message Received = Hello from Parent using Message Queue IPC
Message Queue Deleted

*** Code Execution Successful ***
```

3.

```
main.c | 1 #include <stdio.h>
      2 #include <stdlib.h>
      3 #include <pthread.h>
      4
      5 #define NUM_THREADS 3 // Number of threads to create
      6
      7 // Function that will be executed by each thread
      8 void* threadFunction(void* arg) {
      9     int thread_id = *((int*)arg);
     10    printf("Hello from thread %d!\n", thread_id);
     11    pthread_exit(NULL);
     12 }
     13
     14 int main() {
     15     pthread_t threads[NUM_THREADS];
     16     int thread_ids[NUM_THREADS];
     17     int rc;
     18
     19     // Create threads
     20     for (int i = 0; i < NUM_THREADS; i++) {
     21         thread_ids[i] = i + 1;
     22         rc = pthread_create(&threads[i], NULL, threadFunction, (void*)&thread_ids[i]);
     23         if (rc) {
     24             printf("Error: Unable to create thread %d, return code: %d\n", i+1, rc);
     25             exit(-1);
     26         }
     27     }
     28
     29     // Wait for all threads to finish
     30     for (int i = 0; i < NUM_THREADS; i++) {
     31         pthread_join(threads[i], NULL);
     32     }
     33
     34     printf("All threads completed.\n");
     35     return 0;
     36 }
     37
```

Output

```
Hello from thread 1!
Hello from thread 2!
Hello from thread 3!
All threads completed.

*** Code Execution Successful ***
```

4.

The screenshot shows a code editor interface with a toolbar at the top. The file being edited is `main.c`. The code implements the philosopher's problem using threads and mutexes. The output window shows the execution of the program, where each philosopher alternates between thinking and eating, and the forks are released after each philosopher has finished eating.

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 #define NUM_PHILOSOPHERS 5
7
8 pthread_mutex_t forks[NUM_PHILOSOPHERS]; // Forks as mutexes
9 pthread_t philosophers[NUM_PHILOSOPHERS];
10
11 void* philosopher(void* num) {
12     int id = *((int*)num);
13     int left = id; // Left fork
14     int right = (id + 1) % NUM_PHILOSOPHERS; // Right fork
15
16     while (!) {
17         printf("Philosopher %d is thinking.\n", id + 1);
18         sleep(1); // Thinking
19
20         // To prevent deadlock, pick the lower-numbered fork first
21         if (id % 2 == 0) {
22             pthread_mutex_lock(&forks[left]);
23             pthread_mutex_lock(&forks[right]);
24         } else {
25             pthread_mutex_lock(&forks[right]);
26             pthread_mutex_lock(&forks[left]);
27         }
28
29         printf("Philosopher %d is eating.\n", id + 1);
30         sleep(2); // Eating
31
32         pthread_mutex_unlock(&forks[left]);
33         pthread_mutex_unlock(&forks[right]);
34
35         printf("Philosopher %d finished eating and is back to thinking.\n", id + 1);
36     }
37
38     pthread_exit(NULL);
39 }
40
41 int main() {
```

Output

```
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 5 is thinking.
Philosopher 1 is eating.
Philosopher 1 finished eating and is back to thinking.
Philosopher 1 is thinking.
Philosopher 2 is eating.
Philosopher 5 is eating.
Philosopher 2 finished eating and is back to thinking.
Philosopher 2 is thinking.
Philosopher 5 finished eating and is back to thinking.
Philosopher 5 is thinking.
Philosopher 1 is eating.
Philosopher 4 is eating.
Philosopher 1 finished eating and is back to thinking.
Philosopher 1 is thinking.
Philosopher 4 finished eating and is back to thinking.
Philosopher 4 is thinking.
Philosopher 3 is eating.
Philosopher 5 is eating.
Philosopher 3 finished eating and is back to thinking.
Philosopher 3 is thinking.
Philosopher 4 is eating.
Philosopher 1 is eating.
Philosopher 5 finished eating and is back to thinking.
Philosopher 5 is thinking.
Philosopher 4 finished eating and is back to thinking.
Philosopher 4 is thinking.
Philosopher 2 is eating.
Philosopher 5 finished eating and is back to thinking.
Philosopher 5 is thinking.
Philosopher 4 is eating.
Philosopher 1 is eating.
Philosopher 2 finished eating and is back to thinking.
Philosopher 2 is thinking.
Philosopher 1 finished eating and is back to thinking.
```

5.

C Online Compiler

The screenshot shows a C Online Compiler interface. On the left, there is a file browser with icons for various file types (C, C++, JS, TS, GO, R, ERL, PHP, SWC) and a search bar. The main area contains the code for 'main.c'.

```
main.c
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #define MAX_FILES 100
6 #define MAX_FILENAME 50
7
8 typedef struct {
9     char filename[MAX_FILENAME];
10 } File;
11
12 // Single-level directory
13 typedef struct {
14     File files[MAX_FILES];
15     int fileCount;
16 } Directory;
17
18 // Function to create a file
19 void createFile(Directory *dir, char *name) {
20     if (dir->fileCount >= MAX_FILES) {
21         printf("Directory full! Cannot create more files.\n");
22         return;
23     }
24
25     // Check if file already exists
26     for (int i = 0; i < dir->fileCount; i++) {
27         if (strcmp(dir->files[i].filename, name) == 0) {
28             printf("File '%s' already exists.\n", name);
29             return;
30         }
31     }
32
33     strcpy(dir->files[dir->fileCount].filename, name);
34     dir->fileCount++;
35     printf("File '%s' created successfully.\n", name);
36 }
37
38 // Function to display all files
39 void displayFiles(Directory *dir) {
40     if (dir->fileCount == 0) {
41         printf("Directory is empty.\n");
42     }
43 }
```

The 'Run' button is highlighted in blue. To the right, the 'Output' tab is selected, showing the program's execution:

```
Single Level Directory Operations:
1. Create File
2. Display Files
3. Search File
4. Delete File
5. Exit
Enter your choice: 1
Enter filename to create: FILE.TXT
File 'FILE.TXT' created successfully.

Single Level Directory Operations:
1. Create File
2. Display Files
3. Search File
4. Delete File
5. Exit
Enter your choice:
```