11. Multithreading using a C program

Aim: To illustrate the concept of multithreading using a C program

Algorithm:

1. Define the number of threads to be created (NUM_THREADS).
2. Create a function that each thread will execute (thread_function).
3. Inside the thread function, perform the task that you want each thread to do concurrently.
4. Create an array of threads and spawn the threads using pthread_create.
5. Join the threads using pthread_join to ensure that the main program waits for all threads to finish executing.
6. Compile and run the program.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
void *myThreadFun(void *vargp)
{
sleep(1);
printf("Printing GeeksQuiz from Thread \n");
return NULL;
}
int main()
{
pthread_t thread_id;
printf("Before Thread\n");
pthread_create(&thread_id, NULL, myThreadFun, NULL);
pthread_join(thread_id, NULL);
printf("After Thread\n");
exit(0);
}
```

Result: Thus, the c program for multithreading is executed successfully.

Sample Input: No input is required for this program.

Sample Output:

Thread 0 is executing task A
Thread 1 is executing task B
Thread 2 is executing task C
Thread 3 is executing task D
All threads have finished execution.

12. Design a C program to simulate the concept of Dining-Philosophers problem.

Aim: To design a C program to simulate the concept of Dining-Philosophers problem.
Algorithm:

1. Create a mutex lock for each fork to ensure only one philosopher can pick up a fork at a time.
2. Create a thread for each philosopher.
3. Each philosopher will try to pick up the left fork and then the right fork to start eating.
4. If both forks are available, the philosopher will eat for some time and then release the forks.
5. To avoid deadlock, ensure that each philosopher picks up the forks in the same order (either left-right or right-left).

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
sem_t room;
sem_t chopstick[5];
void * philosopher(void *);
void eat(int);
int main()
{
int i,a[5];
pthread_t tid[5];
sem_init(&room,0,4);
for(i=0;i<5;i++)
sem_init(&chopstick[i],0,1);
for(i=0;i<5;i++){
a[i]=i;
pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
}
for(i=0;i<5;i++)
pthread_join(tid[i],NULL);
}
```

```
void * philosopher(void * num)
{
int phil=*(int *)num;
sem_wait(&room);
printf("\nPhilosopher %d has entered room",phil);
sem_wait(&chopstick[phil]);
sem_wait(&chopstick[(phil+1)%5]);
eat(phil);
sleep(2);
printf("\nPhilosopher %d has finished eating",phil);
sem_post(&chopstick[(phil+1)%5]);
sem_post(&chopstick[phil]);
sem_post(&room);
}
void eat(int phil)
{
printf("\nPhilosopher %d is eating",phil);
}
```

Sample input: Number of philosophers: 5
Duration of eating/thinking: 2 seconds
Duration of simulation: 10 seconds

Sample Output:

Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 5 is thinking.
Philosopher 1 picked up the left fork.
Philosopher 1 picked up the right fork.
Philosopher 1 is eating.
Philosopher 2 picked up the left fork.
Philosopher 2 picked up the right fork.
Philosopher 2 is eating.
Philosopher 1 finished eating and put down forks.
Philosopher 2 finished eating and put down forks.
Philosopher 3 picked up the left fork.

Result: Thus the C program to simulate the concept of Dining-Philosophers problem executed successfully.

13. Construct a C program for implementation of the various memory allocation strategies.

Aim: The aim of this C program is to implement various memory allocation strategies, including First Fit, Best Fit, and Worst Fit

Algorithm:

1. Define a data structure to represent memory blocks.
2. Implement functions for memory allocation using different strategies: First Fit, Best Fit, and Worst Fit.
3. Implement functions for deallocating memory blocks.
4. Create a menu-driven interface to allow the user to select the memory allocation strategy and perform allocation and deallocation operations accordingly.

Program:

```
#include
<stdio.h>
#include
<stdlib.h>#inclu
de <fcntl.h>
#include
<unistd.h>

#define BUFFER_SIZE
4096 void copy(){

const char *sourcefile= "C:/Users/itssk/OneDrive/Desktop/sasi.txt";

    const char *destination_file="C:/Users/itssk/OneDrive/Desktop/sk.txt"; int
    source_fd = open(sourcefile,O_RDONLY);
int dest_fd = open(destination_file, O_WRONLY | O_CREAT |
O_TRUNC, 0666);

    char buffer[BUFFER_SIZE];
    ssize_t bytesRead,bytesWritten;
        while ((bytesRead = read(source_fd, buffer, BUFFER_SIZE)) > 0) { bytesWritten
        = write(dest_fd, buffer,bytesRead);
    }

    close(source_
    fd);
```

```
close(dest_fd)
;

printf("File copied successfully.\n");

}

void create()

{

char path[100];

FILE *fp;
fp=fopen("C:/Users/itssk/OneDrive/Desktop/sasi.txt","w");
printf("file createdsuccessfully");

}
int main(){

int n;

printf("1. Create \t2. Copy \t3. Delete\nEnter your choice: " );
scanf("%d",&n);
switch(n){

case 1:
create();
break;

case 2:
copy();
break;
case 3:
remove("C:/Users/itssk/OneDrive/Desktop/sasi.txt");
Printf ("Deleted successfully");
}}
```

Sample input and Output:

Memory Allocation Strategies:
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit

Enter your choice: 1

Enter total memory size: 100

First Fit Memory Allocation:
1. Allocate
2. Deallocate
3. Exit

Enter your choice: 1
Enter process size: 20

Memory was allocated successfully at position 0.

First Fit Memory Allocation:
1. Allocate
2. Deallocate
3. Exit

Enter your choice: 2
Enter position to deallocate: 0
Memory deallocated successfully.
First Fit Memory Allocation:
1. Allocate
2. Deallocate
3. Exit
Enter your choice: 4
Exiting program…

Result: Thus the c program is to implement various memory allocation strategies, including First Fit, Best Fit, and Worst Fit is executed successfully.

14.Construct a C program to organize the file using a single level directory.

Aim: To organize files into a single-level directory based on their file extensions.

Algorithm:

1. Create a structure to represent a file. This structure may contain attributes like file name, file size, file type, etc.
2. Create a structure to represent a directory. This structure may contain attributes like directory name and an array or linked list to store files in the directory.
3. Implement functions to perform operations like creating a file, deleting a file, listing files in a directory, etc.
4. Write a main function to interact with the user and perform operations based on user input.

Program:

```
#include
<stdio.h>
#include
<stdlib.h>
#include
<string.h> int
main() {

    char mainDirectory[] = "C:/Users/itssk/OneDrive/Desktop";
    char subDirectory[] = "os";
    char fileName[] =
    "example.txt"; char
    filePath[200];
    char mainDirPath[200];
snprintf(mainDirPath, sizeof(mainDirPath), "%s/%s/", mainDirectory, subDirectory);
    snprintf(filePath, sizeof(filePath), "%s%s", mainDirPath, fileName);
    FILE *file = fopen(filePath,"w");
        if (file == NULL) {
        printf("Error creating
        file.\n");return 1;
        }
    fprintf(file, "This is an example file content.");
    printf("File createdsuccessfully: %s\n");
```

}

Sample input:

1. Create file: filename.txt
2. Delete file: filename.txt
3. List files in directory
4. Exit

Enter your choice: 1
Enter file name: filename.txt

Enter your choice: 3

Sample Output:

Files in directory:
1. filename1.txt
2. filename2.txt
3. Filename3.txt

Result: Thus the c program to organize the file using a single-level directory is executed successfully.

15. Design a C program to organize the file using a two-level directory structure.

Aim: to organize files by creating a two-level directory structure based on certain criteria, such as file type or file name prefix.

Algorithm:

1. Open the current directory.
2. Read each file in the directory.
3. Determine the criteria based on which the files should be organized (e.g., file type, file name prefix).
4. Create a two-level directory structure based on the criteria.
5. Move files to their respective directories.
6. Repeat steps 2-5 until all files are processed.
7. Close the directory

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#define MAX_FILES_PER_DIR 100
#define MAX_DIRS 100

struct File {
    char name[50];
    int size;
};

struct Directory {
    char name[50];
    struct File files[MAX_FILES_PER_DIR];
    int file_count;
};

struct TwoLevelDir {
    struct Directory dirs[MAX_DIRS];
    int dir_count;
};

void addFile(struct TwoLevelDir *dir, char *dir_name, char *file_name, int size) {
    int i, found = 0;
    for (i = 0; i < dir->dir_count; i++) {
        if (strcmp(dir->dirs[i].name, dir_name) == 0) {
            if (dir->dirs[i].file_count < MAX_FILES_PER_DIR) {
                struct File new_file;
                strcpy(new_file.name, file_name);
                new_file.size = size;
                dir->dirs[i].files[dir->dirs[i].file_count++] = new_file;
                printf("File '%s' added to directory '%s'\n", file_name, dir_name);
            } else {
                printf("Directory '%s' is full, cannot add more files\n", dir_name);
            }
            found = 1;
            break;
        }
    }
    if (!found) {
        printf("Directory '%s' not found\n", dir_name);
    }
```

```
}

void displayContents(struct TwoLevelDir *dir) {
    int i, j;
    for (i = 0; i < dir->dir_count; i++) {
        printf("Directory: %s\n", dir->dirs[i].name);
        printf("Files:\n");
        for (j = 0; j < dir->dirs[i].file_count; j++) {
            printf("    %s - Size: %d\n", dir->dirs[i].files[j].name, dir->dirs[i].files[j].size);
        }
    }
}

int main() {
    struct TwoLevelDir root_dir;
    root_dir.dir_count = 0;

    addFile(&root_dir, "Documents", "Report.docx", 2048);
    addFile(&root_dir, "Documents", "Presentation.pptx", 4096);
    addFile(&root_dir, "Images", "Photo1.jpg", 1024);

    displayContents(&root_dir);

    return 0;
}
```

Sample input:
Enter source directory: sourceDir
Enter criteria (e.g., file type, file name prefix): to

Sample output:

Files organized successfully!

Result: Thus the C program to organize the file using a two level directory structure is executed successfully.

16. Develop a C program for implementing random access files for processing the employee details.

Aim: To write a C program for implementing random access files for processing the employee details.

Algorithm:

1. Define a structure for representing an employee record, including fields such as employee ID, name, department, salary, etc.
2. Create functions for performing various operations on the employee records,
3. Implement a function to open the random-access file in read/write mode.
4. Implement functions to read and write employee records to the file at specific positions using fseek() and fread()/fwrite().
5. For adding a new record, seek to the end of the file and append the record.
6. For updating or deleting a record, seek to the position of the record within the file using fseek() and then perform the necessary operation.
7. For searching for a record, iterate through the file, reading each record and comparing it with the search criteria.
8. Displaying all records involves reading each record from the file sequentially and printing its details.
9. Close the file when all operations are completed.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct Employee {
        int empId;

        char
        empName[50];
        float
        empSalary;};

        int main()
        { FILE
        *filePtr;

        struct Employee emp;

        filePtr = fopen("employee.dat", "rb+");
        if (filePtr == NULL) {
```

```
        filePtr = fopen("employee.dat", "wb+");
        if (filePtr == NULL) {
            printf("Error creating the
            file.\n"); return 1; }

    }

    Int
    choie;
    do {
        printf("\nEmployee Database Menu:\n");
        printf("1. AddEmployee\n");
        printf("2. Display Employee Details\n");
        printf("3. Update Employee Details\n");
        printf("4. Exit\n");
        printf("Enter your choice:
        ");
        scanf("%d",&choice);swith
        (choice) {

            case 1:

                printf("Enter Employee ID: ");
                scanf("%d",&emp.empId);printf("Ente
                r EmployeeName: ");
                scanf("%s", emp.empName);
                printf("Enter Employee Salary:
                "); scanf("%f",
                &emp.empSalary);

fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee), SEEK_SET);

                fwrite(&emp, sizeof(struct Employee), 1, filePtr);
                printf("Employee details added successfully.\n");
                break;
            case 2:

                printf("Enter Employee ID to display: ");
                scanf("%d",&emp.empId);

fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee), SEEK_SET);
```

```
                    fread(&emp, sizeof(struct Employee), 1, filePtr);
                    printf("Employee ID: %d\n", emp.empId);
                    printf("Employee Name: %s\n",
                    emp.empName);printf("EmployeeSalary:%.2f\n",emp.
                    empSalary); break;

                case 3:

                    printf("Enter Employee ID to update: ");
                    scanf("%d",&emp.empId);

        fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee), SEEK_SET);

                    fread(&emp, sizeof(struct Employee), 1, filePtr);
                    printf("Enter EmployeeName: ");

                    scanf("%s", emp.empName);
                    printf("Enter Employee Salary:");
                    scanf("%f",&emp.empSalary);
        fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee), SEEK_SET);

                    fwrite(&emp, sizeof(struct Employee), 1, filePtr);
                    printf("Employee detailsupdated successfully.\n");
                    break;

                case 4:
                    break;
                    default()
                    printf("Invalid choice. Please try again.\n");

            }

        }
        while(choice!=4);
        fclose(filePtr);
        return 0;
}
```

Sample Input:
Add Record: Employee ID = 101, Name = John Doe, Department = HR, Salary = 50000
Update Record: Employee ID = 101, New Salary = 55000
Search Record: Employee ID = 101
Delete Record: Employee ID = 101

Sample Output:
After adding the record:
Employee ID: 101
Name: John Doe
Department: HR
Salary: 50000

Result: Thus the C program for implementing random access files for processing the employee details is executed successfully.

17. Illustrate the deadlock avoidance concept by simulating Banker’s algorithm with C.

Aim: To write the c program for simulating Banker’s algorithm.
Algorithm:

1. Accept user inputs for the number of processes, resource instances, maximum matrix, allocation matrix, and available resources.
2. Show the process allocation, maximum, and available resources.
3. Calculate need matrix: Determine the need matrix using the formula: need[i][j] = max[i][j] - alloc[i][j].
4. Implement the Banker's safety algorithm to ensure the system's safety by checking for safe sequences of processes.
5. Display the sequence of processes and declare whether the system is in a safe state or not.
6. Verify if all processes are finished to determine the system's state.
7. Identify any processes that are not finished, indicating a deadlock and an unsafe state.
8. End the program execution after completing the safety check and displaying the results.

Program:

```
#include<stdio.h>
#include<conio.h>
int max[100][100];
int alloc[100][100];
```

```
int need[100][100];
int avail[100];
int n,r;
void input();
void show();
void cal();
int main()
{
int i,j;
printf("********** Banker's Algo ************\n");
input();
show();
cal();
getch();
return 0;
}
void input()
{
int i,j;
printf("Enter the no of Processes\t");
scanf("%d",&n);
printf("Enter the no of resources instances\t");
scanf("%d",&r);
printf("Enter the Max Matrix\n");
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
scanf("%d",&max[i][j]);
}
}
printf("Enter the Allocation Matrix\n");
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
scanf("%d",&alloc[i][j]);
}
```

```
}
printf("Enter the available Resources\n");
for(j=0;j<r;j++)
{
scanf("%d",&avail[j]);
}
}
void show()
{
int i,j;
printf("Process\t Allocation\t Max\t Available\t");
for(i=0;i<n;i++)
{
printf("\nP%d\t ",i+1);
for(j=0;j<r;j++)
{
printf("%d ",alloc[i][j]);
}
printf("\t");
for(j=0;j<r;j++)
{
printf("%d ",max[i][j]);
}
printf("\t");
if(i==0)
{
for(j=0;j<r;j++)
printf("%d ",avail[j]);
}
}
}
void cal()
{
int finish[100],temp,need[100][100],flag=1,k,c1=0;
int safe[100];
int i,j;
for(i=0;i<n;i++)
{
```

```
finish[i]=0;
}
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
need[i][j]=max[i][j]-alloc[i][j];
}
}
printf("\n");
while(flag)
{
flag=0;
for(i=0;i<n;i++)
{
int c=0;
for(j=0;j<r;j++)
{
if((finish[i]==0)&&(need[i][j]<=avail[j]))
{
c++;
if(c==r)
{
for(k=0;k<r;k++)
{
avail[k]+=alloc[i][j];
finish[i]=1;
flag=1;
}
printf("P%d->",i);
if(finish[i]==1)
{
i=n;
}
}
}
}
}
```

```
}
for(i=0;i<n;i++)
{
if(finish[i]==1)
{
c1++;
}
else
{
printf("P%d->",i);
}
}
if(c1==n)
{
printf("\n The system is in safe state");
}
else
{
printf("\n Process are in dead lock");
printf("\n System is in unsafe state");
}
}
```

Sample input:

********** Banker's Algo ************

Enter the no of Processes 5

Enter the no of resources instances 3

Enter the Max Matrix

7 5 3

3 2 2

9 0 2

2 2 2

4 3 3

Enter the Allocation Matrix

0 1 0

2 0 0

3 0 2

2 1 1

0 0 2

Enter the available Resources
3 3 2

Sample output:

Process Allocation Max Available
P1 0 1 0 7 5 3 3 3 2
P2 2 0 0 3 2 2
P3 3 0 2 9 0 2
P4 2 1 1 2 2 2
P5 0 0 2 4 3 3
P1->P3->P4->P2->P5->
The system is in safe state

Result: Thus the c program for simulating Banker's algorithm is executed successfully.

18. Construct a C program to simulate producer-consumer problem using semaphores.

Aim: To write a C program to simulate producer-consumer problem using semaphores.

Algorithm:

1. Initialize the semaphore variables mutex, full, and empty.
2. Implement the wait and signal functions to control access to critical sections.
3. Implement the producer function to add items to the buffer, controlling access with semaphores.
4. Implement the consumer function to remove items from the buffer, controlling access with semaphores.
5. Use a menu-driven approach to allow the user to choose between producer, consumer, or exit options.
6. Ensure mutual exclusion by using the mutex semaphore.
7. Manage buffer fullness using the full semaphore.
8. Manage buffer emptiness using the empty semaphore.

Program:

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;
int main()
{
int n;
void producer();
```

```
void consumer();
int wait(int);
int signal(int);
printf("\n1.Producer\n2.Consumer\n3.Exit");
while(1)
{
printf("\nEnter your choice:");
scanf("%d",&n);
switch(n)
{
case 1: if((mutex==1)&&(empty!=0))
producer();
else
printf("Buffer is full!!");
break;
case 2: if((mutex==1)&&(full!=0))
consumer();
else
printf("Buffer is empty!!");
break;
case 3:
exit(0);
break;
}
}
return 0;
}
int wait(int s)
{
return (--s);
}
int signal(int s)
{
return(++s);
}
void producer()
{
mutex=wait(mutex);
```

```
full=signal(full);
empty=wait(empty);
x++;
printf("\nProducer produces the item %d",x);
mutex=signal(mutex);
}
void consumer()
{
mutex=wait(mutex);
full=wait(full);
empty=signal(empty);
printf("\nConsumer consumes item %d",x);
x--;
mutex=signal(mutex);
}
```

Sample input:

```
1    // Choose Producer
2    // Choose Consumer
3    // Choose Exit
```

Sample output:

```
1. Producer
2. Consumer
3. Exit
Enter your choice: 1
Producer produces the item 1
1. Producer
2. Consumer
3. Exit
Enter your choice: 2
Consumer consumes item 1
1. Producer
2. Consumer
3. Exit
Enter your choice: 3
```

Result: Thus the C program to simulate producer-consumer problem using semaphores is executed successfully.

19. Design a C program to implement process synchronization using mutex locks.

Aim: To write a C program to implement process synchronization using mutex locks.
Algorithm:

1. Initialize a mutex lock to manage access to shared resources.
2. Create two threads, each executing the threadFunction.
3. In the threadFunction, simulate some processing to emphasize the need for synchronization.
4. Ensure that each thread operates in isolation by locking and unlocking the mutex.
5. Wait for both threads to finish execution using pthread_join.
6. Destroy the mutex lock after thread execution to release resources.
7. Print the final value of the shared variable counter.

Program:

```
#include <stdio.h>
#include
<pthread.h>
variables int
counter = 0;

pthread_mutex_t mutex;

void*threadFunction(void *arg) {
    int i;

    for (i = 0; i < 1000000; ++i) { }

    return NULL;

}
int main() {
  pthread_mutex_init(&mutex,
  NULL); pthread_t thread1,thread2;
   pthread_create(&thread1, NULL, threadFunction, NULL);
   pthread_create(&thread2,NULL, threadFunction, NULL);

   pthread_join(thread1,
   NULL);
   pthread_join(thread2,
   NULL);
```

```
pthread_mutex_destroy(&mu
tex);printf("Final        counter
value: %d\n",counter);

return 0;

}
```

Sample input: No user input is required for this program.
Sample output: Final counter value: 0

Result: Thus the C program to implement process synchronization using mutex locks is executed successfully.

20. Construct a C program to simulate Reader-Writer problem using Semaphores.

Aim: To implement a C program to simulate Reader-Writer problem using Semaphores.
Algorithm:

1. Create semaphores mutex and writeBlock.
2. Initialize threads for readers and writers.
3. Increment readersCount, acquire mutex, and manage access to writeBlock.
4. Manage access to writeBlock for writing operations.
5. Initialize semaphores, create threads, and manage their execution.
6. Use semaphores to ensure only one thread (reader or writer) accesses the shared resource at a time.
7. Coordinate between reader and writer threads using semaphores to prevent race conditions and maintain data consistency.

Program:

```
#include <stdio.h>

#include <pthread.h>
#include
<semaphore.h>

sem_t mutex, writeBlock;

int data = 0, readersCount = 0;

    void
    *reader(void
    *arg) { int i=0;
```

```
        while (i<10) {
        sem_wait(&mute
        x);readersCount
        ++;

            if (readersCount == 1)
            {
            sem_wait(&writeBloc
            k);
        }

        sem_post(&mutex);

        printf("Reader reads data: %d\n", data);

        sem_wait(&mute
        x);
        readersCount--;

            if (readersCount == 0)
            {
            sem_post(&writeBloc
            k);

        }

        sem_post(&mutex);
        i++;

    }

}

          void
          *writer(void
          *arg) { int i=0;
        while (i<10) {
        sem_wait(&writeBloc
        k);

        operation
        data++;
        printf("Writer writes data: %d\n", data);
```

```
        sem_post(&writeBloc
        k); i++;
    }
}

int main() {
    pthread_t readers,
    writers;
    sem_init(&mutex, 0,1);

    sem_init(&writeBlock, 0, 1);
    pthread_create(&readers, NULL, reader, NULL);
    pthread_create(&writers, NULL, writer,
    NULL);pthread_join(readers, NULL);
    pthread_join(writers,
    NULL);sem_destroy(&mutex);
    sem_destroy(&writeBlock
    ); return 0;
}
```

Sample input: There's no direct input for this program

Sample output:

Reader reads data: 0
Writer writes data: 1
Reader reads data: 1
Reader reads data: 1
Reader reads data: 1
Reader reads data: 1
Writer writes data: 2
Writer writes data: 3
Writer writes data: 4
Reader reads data: 4
Reader reads data: 4
Reader reads data: 4

Result: Thus the C program to simulate Reader-Writer problem using Semaphores is executed successfully.