**Operating system lab practicals**

**SASIKUMAR T**

**192110567**

**1.Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.**

**Aim:-** Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.

**Algorithm:-**

1. Start
2. Declare two variables of type pid\_t: current\_pid and parent\_pid.
3. Call the getpid() function to retrieve the process identifier of the current process and store it in current\_pid.
4. Call the getppid() function to retrieve the parent process's identifier and store it in parent\_pid.
5. Display "Current Process ID: " followed by the value of current\_pid.
6. Display "Parent Process ID: " followed by the value of parent\_pid.
7. End

**Program:-**

#include<stdio.h>

#include<unistd.h>

int main()

{

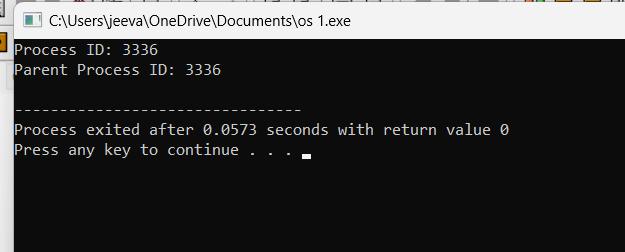
printf("Process ID: %d\n", getpid() );

printf("Parent Process ID: %d\n", getpid() );

return 0;

}

**Output:-**



**2.Identify the system calls to copy the content of one file to another and illustrate the same using a C program.**

**Aim:-** Identify the system calls to copy the content of one file to another and illustrate the same using a C program**.**

**Algorithm:-**

1.Open the source file for reading using open().

2.Open the destination file for writing using open().

3.Use a loop to read data from the source file using read() and write it to the destination file using write() until the end of the source file is reached.

4.Close both the source and destination files using close().

**Program:-**

#include <stdio.h>

#include <stdlib.h>

int main()

{

FILE \*fptr1, \*fptr2;

char filename[100], c;

printf("Enter the filename to open for reading \n");

scanf("%s", filename);

fptr1 = fopen(filename, "r");

if (fptr1 == NULL)

{

printf("Cannot open file %s \n", filename);

exit(0);

}

printf("Enter the filename to open for writing \n");

scanf("%s", filename);

fptr2 = fopen(filename, "w");

if (fptr2 == NULL)

{

printf("Cannot open file %s \n", filename);

exit(0);

}

c = fgetc(fptr1);

while (c != EOF)

{

fputc(c, fptr2);

c = fgetc(fptr1);

}

printf("\nContents copied to %s", filename);

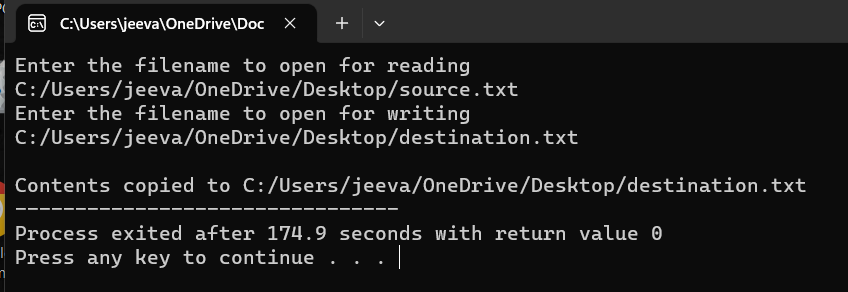
fclose(fptr1);

fclose(fptr2);

return 0;

}

**Output:-**

****

**3.Design a CPU scheduling program with C using First Come First Served technique with the following considerations.**

**a. All processes are activated at time 0.**

**b. Assume that no process waits on I/O devices.**

**Aim:-** Design a CPU scheduling program with C using First Come First Served technique with the following considerations.

**Algorithm:-**

1. Initialize the necessary data structures to store process information.
2. Read the number of processes (N) from the user.
3. For each process, read the following information:
4. Process ID (PID)
5. Arrival Time
6. Burst Time (time required for execution)
7. Sort the processes based on their arrival time in ascending order. This step ensures that the processes are executed in the order they arrive.
8. Initialize a variable current\_time to 0 (representing the current time in the simulation).
9. Initialize a variable total\_waiting\_time to 0.
10. For each process in the sorted list of processes:
    1. If the process has not arrived yet (arrival time > current\_time), update current\_time to the process's arrival time.
    2. Calculate the waiting time for the process as current\_time - arrival time.
    3. Add the waiting time to total\_waiting\_time.
    4. Update current\_time by adding the process's burst time.
    5. Print the process ID, arrival time, burst time, waiting time, and turnaround time.
11. Calculate the average waiting time as total\_waiting\_time / N.
12. Print the average waiting time.

**Program:-**

#include <stdio.h>

int main()

{

int A[100][4];

int i, j, n, total = 0, index, temp;

float avg\_wt, avg\_tat;

printf("Enter number of process: ");

scanf("%d", &n);

printf("Enter Burst Time:\n");

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

printf("P%d: ", i + 1);

scanf("%d", &A[i][1]);

A[i][0] = i + 1;

}

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

index = i;

for (j = i + 1; j < n; j++)

if (A[j][1] < A[index][1])

index = j;

temp = A[i][1];

A[i][1] = A[index][1];

A[index][1] = temp;

temp = A[i][0];

A[i][0] = A[index][0];

A[index][0] = temp;

}

A[0][2] = 0;

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

A[i][2] = 0;

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

A[i][2] += A[j][1];

total += A[i][2];

}

avg\_wt = (float)total / n;

total = 0;

printf("P BT WT TAT\n");

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

A[i][3] = A[i][1] + A[i][2];

total += A[i][3];

printf("P%d %d %d %d\n", A[i][0],A[i][1], A[i][2], A[i][3]);

}

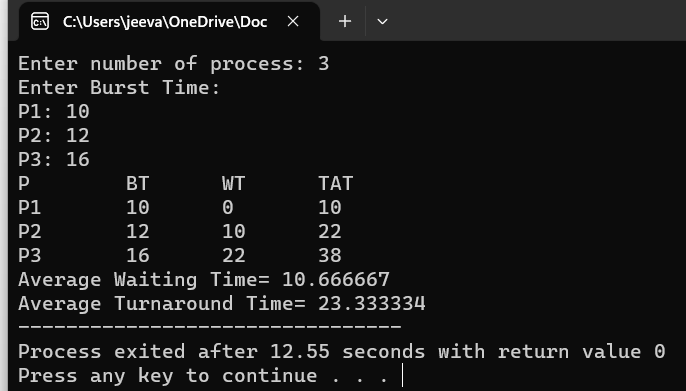
avg\_tat = (float)total / n;

printf("Average Waiting Time= %f", avg\_wt);

printf("\nAverage Turnaround Time= %f", avg\_tat);

}

**Output:-**

****

**4.Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.**

**AIM:** Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.

**ALGORITHM:**

1.Include necessary headers:

Include the necessary header files like <stdio.h> for input/output operations.

2. Define the process structure:

Define a structure to hold the process information, such as process ID, arrival time, burst time, waiting time, and turnaround time.

3.Input the number of processes and their details:

Input the number of processes and their arrival time and burst time.

4.Sort processes by burst time:

Sort the processes based on burst time in ascending order.

5.Calculate waiting time and turnaround time:

Calculate the waiting time and turnaround time for each process based on the SJF scheduling algorithm.

6.Display the scheduling information:

Display the process details including process ID, arrival time, burst time, waiting time, and turnaround time.

**PROGRAM:**

#include<stdio.h>

int main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;

float avg\_wt,avg\_tat;

printf("Enter number of process:");

scanf("%d",&n);

printf("nEnter Burst Time:\n");

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

{

printf("p%d:",i+1);

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

p[i]=i+1;

}

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

{

pos=i;

for(j=i+1;j<n;j++)

{

if(bt[j]<bt[pos])

pos=j;

}

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0;

for(i=1;i<n;i++)

{

wt[i]=0;

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

wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=(float)total/n;

total=0;

printf("nProcesst Burst Time tWaiting TimetTurnaround Time\n");

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

{

tat[i]=bt[i]+wt[i];

total+=tat[i];

printf("np%dtt %dtt %dttt%d",p[i],bt[i],wt[i],tat[i]);

}

avg\_tat=(float)total/n;

printf("nnAverage Waiting Time=%f",avg\_wt);

printf("nAverage Turnaround Time=%fn",avg\_tat);

}



**5. Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.**

**Aim:-** Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

**Algorithm:-**

1. Initialize the necessary data structures to store process information, including process ID, arrival time, burst time, and priority.
2. Read the number of processes (N) from the user.
3. For each process, read the following information:
4. Process ID (PID)
5. Arrival Time
6. Burst Time (time required for execution)
7. Priority (lower values indicate higher priority)
8. Sort the processes based on their arrival time in ascending order.
9. Initialize a variable current\_time to 0 (representing the current time in the simulation).
10. Initialize a variable completed to 0 (to keep track of the number of completed processes).
11. Create a priority queue or data structure to store processes based on their priority.
12. Initialize a variable total\_waiting\_time to 0.
13. While there are still processes to execute (i.e., completed < N), repeat the following:
    1. For each process that has arrived but has not been completed, add it to the priority queue.
    2. Pop the process with the highest priority from the queue.
    3. Calculate the waiting time for the process as current\_time - arrival time.
    4. Add the waiting time to total\_waiting\_time.
    5. Update current\_time by adding the process's burst time.
    6. Print the process ID, arrival time, burst time, waiting time, and turnaround time.
    7. Mark the process as completed.
14. Calculate the average waiting time as total\_waiting\_time / N.
15. Print the average waiting time.

**Program:-**

#include<stdio.h>

struct priority\_scheduling {

char process\_name;

int burst\_time;

int waiting\_time;

int turn\_around\_time;

int priority;

};

int main() {

int number\_of\_process;

int total = 0;

struct priority\_scheduling temp\_process;

int ASCII\_number = 65;

int position;

float average\_waiting\_time;

float average\_turnaround\_time;

printf("Enter the total number of Processes: ");

scanf("%d", & number\_of\_process);

struct priority\_scheduling process[number\_of\_process];

printf("\nPlease Enter the Burst Time and Priority of each process:\n");

for (int i = 0; i < number\_of\_process; i++) {

process[i].process\_name = (char) ASCII\_number;

printf("\nEnter the details of the process %c \n", process[i].process\_name);

printf("Enter the burst time: ");

scanf("%d", & process[i].burst\_time);

printf("Enter the priority: ");

scanf("%d", & process[i].priority);

ASCII\_number++;

}

for (int i = 0; i < number\_of\_process; i++) {

position = i;

for (int j = i + 1; j < number\_of\_process; j++) {

if (process[j].priority > process[position].priority)

position = j;

}

temp\_process = process[i];

process[i] = process[position];

process[position] = temp\_process;

}

process[0].waiting\_time = 0;

for (int i = 1; i < number\_of\_process; i++) {

process[i].waiting\_time = 0;

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

process[i].waiting\_time += process[j].burst\_time;

}

total += process[i].waiting\_time;

}

average\_waiting\_time = (float) total / (float) number\_of\_process;

total = 0;

printf("\n\nProcess\_name \t Burst Time \t Waiting Time \t Turnaround Time\n");

printf("------------------------------------------------------------\n");

for (int i = 0; i < number\_of\_process; i++) {

process[i].turn\_around\_time = process[i].burst\_time + process[i].waiting\_time;

total += process[i].turn\_around\_time;

printf("\t %c \t\t %d \t\t %d \t\t %d", process[i].process\_name, process[i].burst\_time, process[i].waiting\_time, process[i].turn\_around\_time);

printf("\n-----------------------------------------------------------\n");

}

average\_turnaround\_time = (float) total / (float) number\_of\_process;

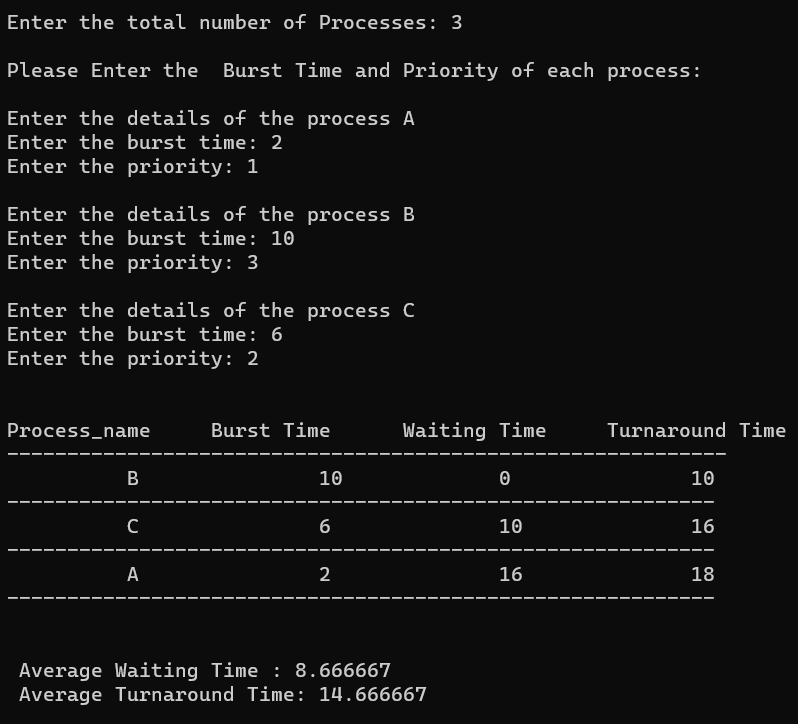
printf("\n\n Average Waiting Time : %f", average\_waiting\_time);

printf("\n Average Turnaround Time: %f\n", average\_turnaround\_time);

return 0;

}

**Output:-**

****

**6. Construct a C program to simulate Round Robin scheduling algorithm with C.**

**Aim:-** Construct a C program to simulate Round Robin scheduling algorithm with C.

**Algorithm:-**

1. Initialize the necessary data structures to store process information, including process ID, burst time, and remaining time.
2. Read the number of processes (N) from the user.
3. Read the time quantum (slice time) from the user.
4. For each process, read the following information:
5. Process ID (PID)
6. Burst Time (time required for execution)
7. Create a queue data structure to store the processes.
8. Enqueue all processes into the queue.
9. Initialize a variable current\_time to 0 (representing the current time in the simulation).
10. Initialize a variable total\_waiting\_time to 0.
11. While the queue is not empty, repeat the following:
    1. Dequeue a process from the front of the queue.
    2. Calculate the execution time for the process, which is the minimum of the time quantum and the remaining time for the process.
    3. Update the process's remaining time.
    4. Update current\_time by adding the execution time.
    5. If the process still has remaining time, enqueue it back into the queue.
    6. Calculate the waiting time for the process as current\_time - arrival time, where arrival time is the time when the process was first enqueued.
    7. Add the waiting time to total\_waiting\_time.
12. Calculate the average waiting time as total\_waiting\_time / N.
13. Print the average waiting time.

**Program:-**

#include<stdio.h>

#include<conio.h>

int main()

{

int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];

float avg\_wt, avg\_tat;

printf(" Total number of process in the system: ");

scanf("%d", &NOP);

y = NOP;

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

{

printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);

printf(" Arrival time is: \t");

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

printf(" \nBurst time is: \t");

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

temp[i] = bt[i];

}

printf("Enter the Time Quantum for the process: \t");

scanf("%d", &quant);

printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0)

{

sum = sum + temp[i];

temp[i] = 0;

count=1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - quant;

sum = sum + quant;

}

if(temp[i]==0 && count==1)

{

y--;

printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

wt = wt+sum-at[i]-bt[i];

tat = tat+sum-at[i];

count =0;

}

if(i==NOP-1)

{

i=0;

}

else if(at[i+1]<=sum)

{

i++;

}

else

{

i=0;

}

}

avg\_wt = wt \* 1.0/NOP;

avg\_tat = tat \* 1.0/NOP;

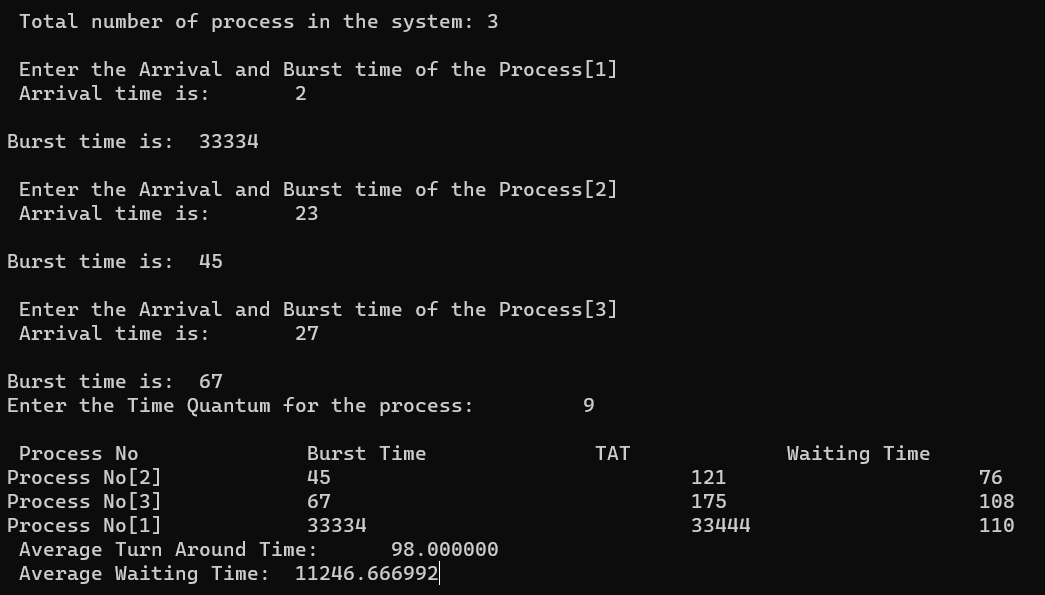
printf("\n Average Turn Around Time: \t%f", avg\_wt);

printf("\n Average Waiting Time: \t%f", avg\_tat);

getch();

}

**Output:-**

****

**7.** **Construct a C program to implement non-preemptive SJF algorithm**

**AIM:** Construct a C program to implement non-preemptive SJF algorithm

**ALGORITHM:**

1.Include necessary headers:

Include the necessary header files like <stdio.h> for input/output operations.

2. Define the process structure:

Define a structure to hold the process information, such as process ID, arrival time, burst time, waiting time, and turnaround time.

3. Input the number of processes and their details:

Input the number of processes and their arrival time and burst time.

4.Sort processes by burst time:

Sort the processes based on burst time in ascending order.

5. Calculate waiting time and turnaround time:

Calculate the waiting time and turnaround time for each process based on the non-preemptive SJF scheduling algorithm.

6. Display the scheduling information:

Display the process details including process ID, arrival time, burst time, waiting time, and turnaround time.

**PROGRAM:**

#include<stdio.h>

int main()

{

int at[10],bt[10],pr[10];

int n,i,j,temp,time=0,count,over=0,sum\_wait=0,sum\_turnaround=0,start;

float avgwait,avgturn;

printf("Enter the number of processes\n");

scanf("%d",&n);

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

{

printf("Enter the arrival time and execution time for process %d\n",i+1);

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

pr[i]=i+1;

}

for(i=0;i<n-1;i++)

{

for(j=i+1;j<n;j++)

{

if(at[i]>at[j])

{

temp=at[i];

at[i]=at[j];

at[j]=temp;

temp=bt[i];

bt[i]=bt[j];

bt[j]=temp;

temp=pr[i];

pr[i]=pr[j];

pr[j]=temp;

}

}

}

printf("\n\nProcess\t|Arrival time\t|Execution time\t|Start time\t|End time\t|waiting time\t|Turnaround time\n\n");

while(over<n)

{

count=0;

for(i=over;i<n;i++)

{

if(at[i]<=time)

count++;

else

break;

}

if(count>1)

{

for(i=over;i<over+count-1;i++)

{

for(j=i+1;j<over+count;j++)

{

if(bt[i]>bt[j])

{

temp=at[i];

at[i]=at[j];

at[j]=temp;

temp=bt[i];

bt[i]=bt[j];

bt[j]=temp;

temp=pr[i];

pr[i]=pr[j];

pr[j]=temp;

}

}

}

}

start=time;

time+=bt[over];

printf("p[%d]\t|\t%d\t|\t%d\t|\t%d\t|\t%d\t|\t%d\t|\t%d\n",pr[over],

at[over],bt[over],start,time,time-at[over]-bt[over],time-at[over]);

sum\_wait+=time-at[over]-bt[over];

sum\_turnaround+=time-at[over];

over++;

}

avgwait=(float)sum\_wait/(float)n;

avgturn=(float)sum\_turnaround/(float)n;

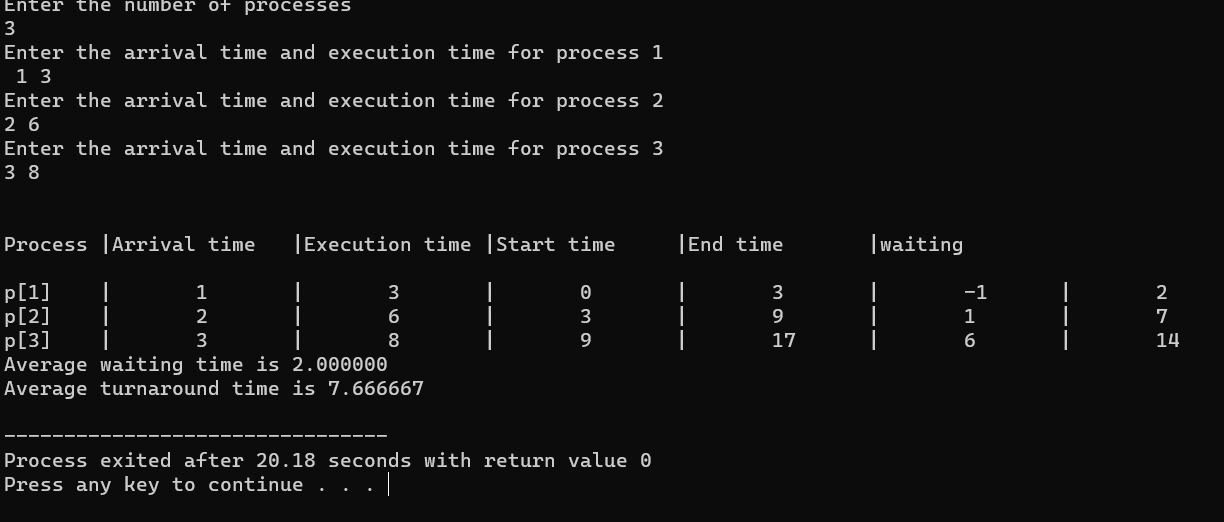
printf("Average waiting time is %f\n",avgwait);

printf("Average turnaround time is %f\n",avgturn);

return 0;

}

**Output:-**

****

**8. Construct a C program to simulate Round Robin scheduling algorithm with C.**

**AIM:** Construct a C program to simulate Round Robin scheduling algorithm with C.

**ALGORITHM:**

1.Include necessary headers:

Include the necessary header files like <stdio.h> for input/output operations.

2.Define the process structure:

Define a structure to hold the process information, such as process ID, arrival time, burst time, waiting time, and turnaround time.

3.Input the number of processes and their details:

Input the number of processes and their arrival time and burst time.

4.Input the time quantum for Round Robin:

Input the time quantum to be used in the Round Robin scheduling.

5.Simulate Round Robin scheduling:

Implement the Round Robin scheduling algorithm, including a queue to keep track of the processes.

6.Display the scheduling information:

Display the process details including process ID, arrival time, burst time, waiting time, and turnaround time.

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

int main()

{

int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];

float avg\_wt, avg\_tat;

printf(" Total number of process in the system: ");

scanf("%d", &NOP);

y = NOP;

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

{

printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);

printf(" Arrival time is: \t");

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

printf(" \nBurst time is: \t");

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

temp[i] = bt[i];

}

printf("Enter the Time Quantum for the process: \t");

scanf("%d", &quant);

printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0)

{

sum = sum + temp[i];

temp[i] = 0;

count=1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - quant;

sum = sum + quant;

}

if(temp[i]==0 && count==1)

{

y--;

printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

wt = wt+sum-at[i]-bt[i];

tat = tat+sum-at[i];

count =0;

}

if(i==NOP-1)

{

i=0;

}

else if(at[i+1]<=sum)

{

i++;

}

else

{

i=0;

}

}

avg\_wt = wt \* 1.0/NOP;

avg\_tat = tat \* 1.0/NOP;

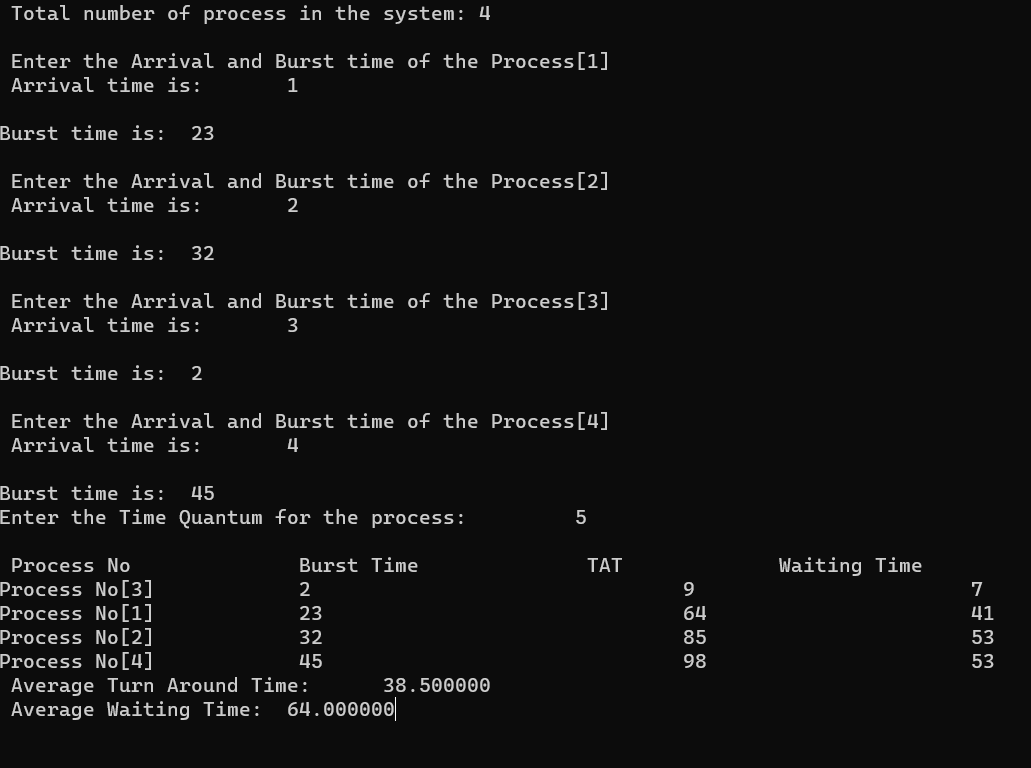
printf("\n Average Turn Around Time: \t%f", avg\_wt);

printf("\n Average Waiting Time: \t%f", avg\_tat);

getch();

}

**Output:-**



**9 Illustrate the concept of inter-process communication using shared memory with a C program**

**AIM:**

To implement the concept of inter-process communication using shared memory using C programming.

**ALGORITHM:**

1. **Create a shared memory segment:**
   * Use **shmget()** function to create a new shared memory segment or get the identifier of an existing one.
   * Ensure to handle errors if the shared memory creation fails.
2. **Attach shared memory to processes:**
   * Use **shmat()** function to attach the shared memory segment to the process address space.
   * This allows processes to read and write data to the shared memory.
3. **Read/Write data in shared memory:**
   * Processes can read and write data directly to the shared memory location.
   * Ensure proper synchronization mechanisms (like semaphores) are used to avoid race conditions and maintain data consistency.
4. **Detach shared memory and clean up:**
   * Use **shmdt()** function to detach the shared memory segment from the process when done.
   * Optionally, remove the shared memory segment using **shmctl()** with the **IPC\_RMID** command.

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 1024 // Size of the shared memory segment

int main() {

key\_t key = ftok("shmfile", 65); // Generate a unique key for the shared memory segment

// Create a new shared memory segment (or get the identifier of an existing one)

int shmid = shmget(key, SHM\_SIZE, IPC\_CREAT | 0666);

if (shmid == -1) {

perror("shmget");

exit(EXIT\_FAILURE);

}

// Attach the shared memory segment to the process address space

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

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

perror("shmat");

exit(EXIT\_FAILURE);

}

// Write data to the shared memory

strcpy(shm\_ptr, "Hello, shared memory!");

// Detach the shared memory segment from the process

if (shmdt(shm\_ptr) == -1) {

perror("shmdt");

exit(EXIT\_FAILURE);

}

printf("Data written to shared memory: %s\n", shm\_ptr);

// Optional: Remove the shared memory segment

if (shmctl(shmid, IPC\_RMID, NULL) == -1) {

perror("shmctl");

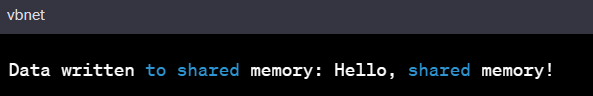
exit(EXIT\_FAILURE);

}

return 0;

}

**OUTPUT:**

****

**10. Illustrate the concept of inter-process communication using message queue with a c program**

**AIM :**

To implement the concept of inter-process communication using message queue with a c program

**ALGORITHM :**

1. **Create a message queue:**
   * Use **msgget()** function to create a new message queue or get the identifier of an existing one.
   * Ensure to handle errors if the message queue creation fails.
2. **Send a message to the queue (Producer process):**
   * Define a structure for the message containing necessary data fields.
   * Populate the message structure with appropriate data.
   * Use **msgsnd()** function to send the message to the message queue.
   * Handle errors if message sending fails.
3. **Receive a message from the queue (Consumer process):**
   * Define a structure for the message to receive data.
   * Use **msgrcv()** function to receive a message from the message queue.
   * Process the received message as needed.
   * Handle errors if message receiving fails.
4. **Remove the message queue (Optional):**
   * Use **msgctl()** function with **IPC\_RMID** command to remove the message queue when it's no longer needed.

**PROGRAM :**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct message {

long msg\_type;

char msg\_text[100];

};

int main() {

key\_t key = ftok("msgqfile", 65); // Generate a unique key for the message queue

// Create a new message queue (or get the identifier of an existing one)

int msgid = msgget(key, IPC\_CREAT | 0666);

if (msgid == -1) {

perror("msgget");

exit(EXIT\_FAILURE);

}

struct message msg;

msg.msg\_type = 1; // Message type (can be any positive number)

// Producer: Send a message to the message queue

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

if (msgsnd(msgid, (void\*)&msg, sizeof(msg.msg\_text), IPC\_NOWAIT) == -1) {

perror("msgsnd");

exit(EXIT\_FAILURE);

}

printf("Producer: Data sent to message queue: %s\n", msg.msg\_text);

// Consumer: Receive a message from the message queue

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

perror("msgrcv");

exit(EXIT\_FAILURE);

}

printf("Consumer: Data received from message queue: %s\n", msg.msg\_text);

// Remove the message queue

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

perror("msgctl");

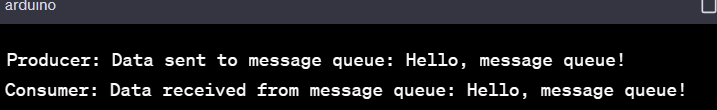
exit(EXIT\_FAILURE);

}

return 0;

}

**OUTPUT :**

****

**11. Illustrate the concept of multithreading using a C program**

**AIM :**

To implement the concept of multithreading using C program

**ALGORITHM :**

1. **Include Necessary Libraries:**
   * Include the necessary header files for multithreading. For POSIX threads in C, include **pthread.h**.
2. **Define Thread Function:**
   * Define a function that will be executed by the threads. This function should have a **void\*** return type and a **void\*** parameter, which can be used to pass data to the thread.
3. **Create Threads:**
   * In the main function or any other function, create thread objects using **pthread\_t**.
   * Use **pthread\_create()** to create threads and specify the thread function as well as any parameters to pass.
4. **Thread Execution:**
   * Threads execute concurrently and perform the tasks defined in the thread function.
5. **Synchronization and Coordination (Optional):**
   * Use synchronization mechanisms such as mutexes, semaphores, or condition variables to coordinate the execution of threads and ensure data consistency.
6. **Wait for Threads to Complete (Optional):**
   * Use **pthread\_join()** to wait for threads to finish their execution if the main thread needs to synchronize with the created threads.

**Thread Termination (Optional):**

* Threads can exit by returning from the thread function or by calling **pthread\_exit()**. The main thread can also call **pthread\_exit()** to terminate the entire process.

**PROGRAM :**

#include <stdio.h>

#include <pthread.h>

void\* threadFunction(void\* arg) {

char\* message = (char\*)arg;

printf("%s\n", message);

return NULL;

}

int main() {

pthread\_t thread1, thread2;

char\* message1 = "Hello from Thread 1!";

char\* message2 = "Hello from Thread 2!";

// Create threads

pthread\_create(&thread1, NULL, threadFunction, (void\*)message1);

pthread\_create(&thread2, NULL, threadFunction, (void\*)message2);

// Wait for threads to complete

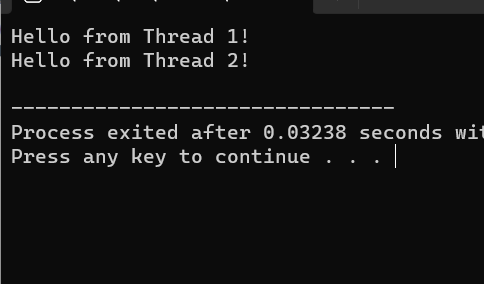
pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

return 0;

}

**OUTPUT :**

****

**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. **Initialize Semaphores:**
   * Create an array of semaphores, one for each philosopher, to control access to the chopsticks.
   * Initialize each semaphore to 1, indicating that the chopstick is available.
2. **Define Philosopher Structure:**
   * Define a structure to represent a philosopher, which includes their ID and the semaphores representing the left and right chopsticks.
3. **Philosopher Lifecycle:**
   * Each philosopher runs as a separate thread. In the thread function:
     + Think: Philosopher thinks for a random amount of time.
     + Pick up Chopsticks: Philosopher tries to pick up the left and right chopsticks (acquire semaphores). If both are available, the philosopher proceeds to eat. If not, they release the acquired chopstick(s) and return to thinking.
     + Eat: Philosopher eats for a random amount of time.
     + Put Down Chopsticks: Philosopher releases the left and right chopsticks (release semaphores) after eating.
     + Repeat the above steps indefinitely to simulate the philosopher's life cycle.
4. **Implement Deadlock Avoidance:**
   * To avoid deadlock, impose a constraint such that a philosopher can only pick up both chopsticks if both are available.
   * One way to achieve this is by introducing a global semaphore that limits the number of philosophers allowed to pick up chopsticks simultaneously. For example, if there are 5 philosophers, allow at most 4 philosophers to pick up chopsticks simultaneously. This prevents the circular wait condition and avoids deadlock.
5. **Compile and Run:**
   * Compile the program with appropriate multithreading flags (e.g., **-pthread** for gcc on Unix-based systems).
   * Run the program to observe the dining philosophers problem simulation.

**PROGRAM :**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

pthread\_mutex\_t chopsticks[NUM\_PHILOSOPHERS];

void\* philosopherLifeCycle(void\* arg) {

int id = \*((int\*)arg);

int left\_chopstick = id;

int right\_chopstick = (id + 1) % NUM\_PHILOSOPHERS;

while (1) {

// Think

printf("Philosopher %d is thinking...\n", id);

// Pick up chopsticks

pthread\_mutex\_lock(&chopsticks[left\_chopstick]);

pthread\_mutex\_lock(&chopsticks[right\_chopstick]);

// Eat

printf("Philosopher %d is eating...\n", id);

sleep(rand() % 3 + 1); // Eating time

// Put down chopsticks

pthread\_mutex\_unlock(&chopsticks[left\_chopstick]);

pthread\_mutex\_unlock(&chopsticks[right\_chopstick]);

// Repeat the cycle

}

}

int main() {

pthread\_t philosophers[NUM\_PHILOSOPHERS];

int philosopher\_ids[NUM\_PHILOSOPHERS];

// Initialize mutex locks

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

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

}

// Create philosopher threads

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

philosopher\_ids[i] = i;

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

}

// Wait for threads to finish (although they run indefinitely)

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

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

}

// Destroy mutex locks

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

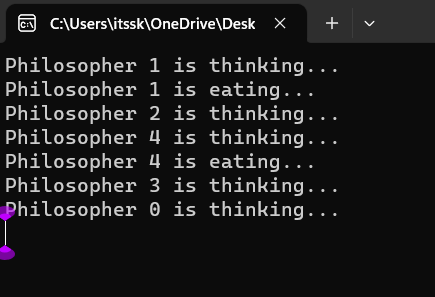
pthread\_mutex\_destroy(&chopsticks[i]);

}

return 0;

}

**OUTPUT :**

****

**13. Construct a C program to implement various memory allocation strategies.**

**AIM :**

To construct a C program to implement various memory allocation strategies.

**ALGORITHM :**

1. **Include Necessary Libraries:**
   * Include the necessary header files such as **stdio.h**, **stdlib.h**, etc.
2. **Define Process Control Block (PCB) Structure:**
   * Define a structure to represent a Process Control Block (PCB) that contains information about each process, including process ID, memory size, and allocation status.
3. **Implement Memory Allocation Functions:**
   * Implement functions for memory allocation strategies like First Fit, Best Fit, and Worst Fit.
   * Each function should search for a suitable block of memory in the memory pool based on the specific strategy (first fit, best fit, or worst fit).
   * Allocate memory to the process by updating the allocation status in the PCB and updating the memory pool accordingly.
4. **Implement Memory Deallocation Function:**
   * Implement a function to deallocate memory occupied by a process.
   * Update the allocation status in the PCB and release the memory block, merging it with adjacent free blocks if necessary.
5. **Main Function:**
   * In the **main** function, initialize the memory pool (an array representing the available memory).
   * Create PCBs for processes with specific memory requirements.
   * Call the appropriate memory allocation functions based on the desired strategy for each process.
   * Deallocate memory for completed processes using the memory deallocation function.
6. **Print Memory Allocation Status:**
   * Implement a function to print the memory allocation status after each allocation and deallocation operation.
7. **Compile and Run:**
   * Compile the program and run it to observe how different memory allocation strategies work.

**PROGRAM :**

#include<stdio.h>

void bestfit(int mp[],int p[],int m,int n){

int j=0;

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

if(mp[i]>p[j]){

printf("\n%d fits in %d",p[j],mp[i]);

mp[i]=mp[i]-p[j++];

i=i-1;

}

}

for(int i=j;i<m;i++)

{

printf("\n%d must wait for its process",p[i]);

}

}

void rsort(int a[],int n){

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

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

if(a[i]>a[j]){

int t=a[i];

a[i]=a[j];

a[j]=t;

}

}

}

}

void sort(int a[],int n){

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

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

if(a[i]<a[j]){

int t=a[i];

a[i]=a[j];

a[j]=t;

}

}

}

}

void firstfit(int mp[],int p[],int m,int n){

sort(mp,n);

sort(p,m);

bestfit(mp,p,m,n);

}

void worstfit(int mp[],int p[],int m,int n){

rsort(mp,n);

sort(p,m);

bestfit(mp,p,m,n);

}

int main(){

int m,n,mp[20],p[20],ch;

printf("Number of memory partition : ");

scanf("%d",&n);

printf("Number of process : ");

scanf("%d",&m);

printf("Enter the memory partitions : \n");

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

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

}

printf("ENter process size : \n");

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

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

}

printf("1. Firstfit\t2. Bestfit\t3. worstfit\nEnter your choice : ");

scanf("%d",&ch);

switch(ch){

case 1:

bestfit(mp,p,m,n);

break;

case 2:

firstfit(mp,p,m,n);

break;

case 3:

worstfit(mp,p,m,n);

break;

default:

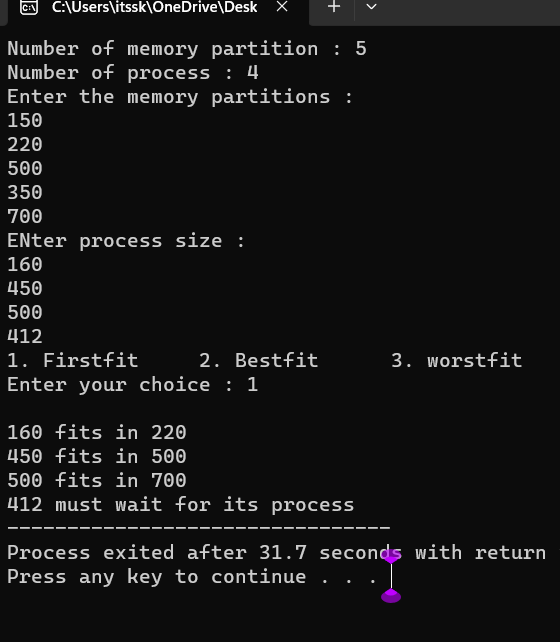
printf("invalid");

break;

}

}

**OUTPUT :**

****

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

**AIM:**

To construct a c program to organize the file using single level directory

**ALGORITHM :**

**Step 1: Define Structures**

Define structures to represent files and the directory.

**Step 2: Initialize Directory**

Create a function or code segment to initialize the directory structure. Set the initial file count to 0.

**Step 3: Add Files**

Implement a function or code segment to add files to the directory. This function should handle adding files, updating the file count, and handling errors if the directory is full.

**Step 4: List Files**

Create a function or code segment to list all the files in the directory. This function should iterate through the file list and print the file names.

**Step 5: Delete Files (Optional)**

Implement a function or code segment to delete files from the directory. This function should handle removing files, updating the file count, and handling errors if the file is not found.

**Step 6: Implement User Interface**

Create a user interface for interacting with the program. This could be a menu-driven interface where users can choose to add files, list files, delete files, or exit the program.

**Step 7: Test the Program**

Compile the program using a C compiler and test it by adding files, listing files, and deleting files. Make sure the program handles different scenarios and errors gracefully.

**Step 8: Refine and Expand (Optional)**

Refine your program based on testing results. You can also expand the functionality by adding more features, error handling, or optimizing the code.

**Step 9: Document Your Code (Optional)**

Document your code by adding comments to explain the functionality of different sections of your program. This will make it easier for others (and yourself) to understand the code in the future.

**Step 10: Compile and Distribute**

Once your program is complete and thoroughly tested, compile it into an executable file. If you want to distribute the program, you can create an installer or provide the executable along with necessary instructions.

**PROGRAM :**

#include <stdio.h>

#include <stdlib.h>

#include <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 created successfully");

}

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;

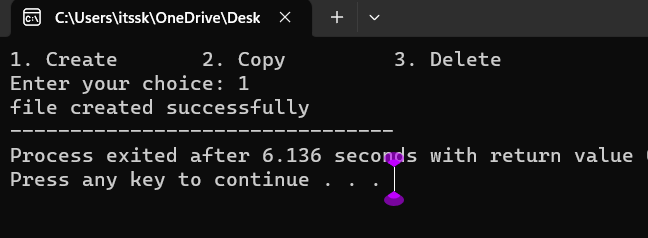
case 3:

remove("C:/Users/itssk/OneDrive/Desktop/sasi.txt");

printf("Deleted successfully");

}}

**OUTPUT :**

****

1. **Design a C program to organize the file using two level directory structure.**

**AIM :**

To design a C program to organize the file using two level directory structure

**Algorithm :**

1. **Define Structures:** Define structures for files and directories. Each directory structure should contain an array for files and an array for subdirectories.
2. **Initialize Root Directory:** Create a root directory structure. This serves as the starting point for the two-level directory structure.
3. **Add Files to Directories:** Implement a function to add files to a specific directory. Handle adding files, updating the file count, and handling errors if the directory is full.
4. **Add Subdirectories:** Implement a function to add subdirectories to a specific directory. Manage adding directories, updating the directory count, and handling errors if the parent directory is full.
5. **List Files and Subdirectories:** Create functions to list all the files and subdirectories in a directory. These functions should iterate through the file and subdirectory arrays and print their names.
6. **Delete Files and Subdirectories (Optional):** Implement functions to delete files and subdirectories from a directory. Handle removing files or directories, updating the counts, and handling errors if the file or directory is not found.
7. **Implement User Interface:** Design a user interface for interacting with the program. This could be a menu-driven interface where users can add files, add subdirectories, list files, list subdirectories, delete files, delete subdirectories, or exit the program.
8. **Test the Program:** Compile the program and test it thoroughly. Add files, add subdirectories, list files, list subdirectories, delete files, and delete subdirectories. Ensure the program handles different scenarios and errors gracefully.
9. **Refine and Expand (Optional):** Refine the program based on testing results. Expand the functionality by adding more features, error handling, or optimizing the code.
10. **Document Your Code (Optional):** Document your code by adding comments to explain the functionality of different sections. This will make it easier for others to understand the code in the future.

**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;

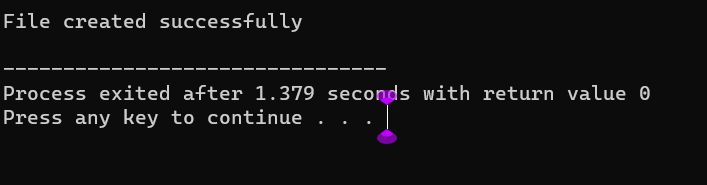
}

fprintf(file, "This is an example file content.");

printf("File created successfully: %s\n");

}

**OUTPUT :**

****

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

**AIM :**

To develop a C program for implementing random access file for processing the employee details

**ALGORITHM :**

1. **Define Structure:** Define a structure to represent employee details. Include attributes like employee ID, name, salary, and any other relevant information.
2. **Open File in Binary Mode:** Open a file in binary mode using the **fopen** function. Specify the file path and mode ("rb+" for reading and writing binary files).
3. **Menu-Driven Interface:** Create a menu-driven interface for the user to perform operations. Options could include adding a new employee, updating existing employee details, searching for an employee, deleting an employee, listing all employees, and exiting the program.
4. **Implement Functions:** Implement functions corresponding to each menu option. For example, implement functions to add a new employee, update employee details, search for an employee by ID, delete an employee, and list all employees. These functions should perform file operations like reading and writing records.
5. **Random Access File Operations:** Utilize fseek and ftell functions to perform random access file operations. Use fseek to move the file pointer to the desired record based on the employee ID and ftell to determine the current position of the file pointer.
6. **File Read and Write:** Implement functions to read and write employee records to the file. Use fread and fwrite functions to read and write structures to the file.
7. **Error Handling:** Implement error handling to deal with situations where the file cannot be opened or when operations like adding, updating, or deleting employees fail. Display appropriate error messages to the user.
8. **Close the File:** Close the file using the **fclose** function when the program is exiting or when the file operations are completed.
9. **Testing:** Test the program thoroughly by adding, updating, searching, and deleting employee records. Ensure that the program handles edge cases and errors gracefully.
10. **Documentation (Optional):** Add comments and documentation to your code to explain the functionality of different sections, making it easier for others (and yourself) to understand the code in the future.

**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 choice;

do {

printf("\nEmployee Database Menu:\n");

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

printf("2. Display Employee Details\n");

printf("3. Update Employee Details\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter Employee ID: ");

scanf("%d", &emp.empId);

printf("Enter Employee Name: ");

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("Employee Salary: %.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 Employee Name: ");

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 updated successfully.\n");

break;

case 4:

break;

default:

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

}

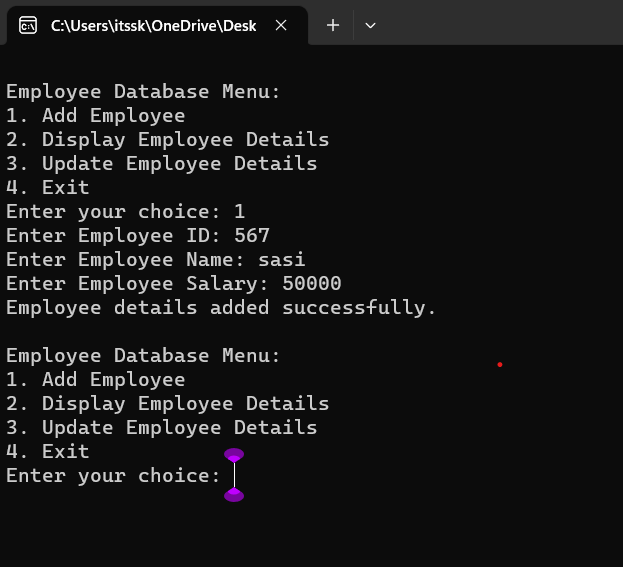
} while (choice != 4);

fclose(filePtr);

return 0;

}

**OUTPUT :**

****

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

**AIM :**

To illustrate the deadlock avoidance concept by simulating Banker’s algorithm using C.

**ALGORITHM :**

1. **Define Data Structures:** Define appropriate data structures to store the available resources, maximum resources, allocated resources, and need matrix for each process.
2. **Initialize Matrices:** Initialize the available, maximum, allocated, and need matrices based on the system's resources and the maximum demand of each process.
3. **Input Request:** Implement a function to input resource request from processes. This function should validate if the request is within the maximum limit specified by each process.
4. **Safety Algorithm:** Implement the Banker's safety algorithm to check if the system is in a safe state. Use the available, allocated, and need matrices to determine if the system can allocate resources to processes without entering into a deadlock state.
5. **Resource Allocation:** Implement resource allocation functions to handle the request from processes. Check if the request can be granted safely using the Banker's algorithm. If the request can be granted, update the allocated and available matrices accordingly.
6. **User Interface:** Create a user interface to interact with the program. Allow users to input resource requests and display the current state of the system, including available resources and resource allocation status.
7. **Deadlock Scenario:** Introduce scenarios where a deadlock can potentially occur (e.g., requesting more resources than available or requesting resources in a circular wait condition).
8. **Testing:** Test the program with different resource request scenarios. Ensure that the system handles requests properly without entering into a deadlock state. Also, test scenarios where the system should deny requests to prevent deadlock.
9. **Documentation (Optional):** Add comments and documentation to the code to explain the functionality of different sections. Document the Banker's algorithm steps and how it prevents deadlocks in the system.
10. **Error Handling:** Implement error handling mechanisms to deal with invalid input, unexpected scenarios, or any issues that might occur during resource allocation and deallocation.

**PROGRAM :**

#include <stdio.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

int is\_safe();

int available[MAX\_RESOURCES] = {3, 3, 2}; // Available instances of each resource

int maximum[MAX\_PROCESSES][MAX\_RESOURCES] = {{7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};

int allocation[MAX\_PROCESSES][MAX\_RESOURCES] = {{0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};

int request\_resources(int process\_num, int request[]) {

// Check if request can be granted

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

if (request[i] > available[i] || request[i] > maximum[process\_num][i] - allocation[process\_num][i])

return 0; // Request cannot be granted

}

// Try allocating resources temporarily

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

available[i] -= request[i];

allocation[process\_num][i] += request[i];

// Update maximum and need matrix if request is granted

maximum[process\_num][i] -= request[i];

}

// Check if system is in safe state after allocation

if (is\_safe()) {

return 1; // Request is granted

} else {

// Roll back changes if not safe

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

available[i] += request[i];

allocation[process\_num][i] -= request[i];

maximum[process\_num][i] += request[i];

}

return 0; // Request is denied

}

}

int is\_safe() {

int work[MAX\_RESOURCES];

int finish[MAX\_PROCESSES] = {0};

// Initialize work array

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

work[i] = available[i];

}

// Check if processes can finish

int count = 0;

while (count < MAX\_PROCESSES) {

int found = 0;

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

if (finish[i] == 0) {

int j;

for (j = 0; j < MAX\_RESOURCES; j++) {

if (maximum[i][j] - allocation[i][j] > work[j])

break;

}

if (j == MAX\_RESOURCES) {

// Process can finish, update work and mark as finished

for (int k = 0; k < MAX\_RESOURCES; k++) {

work[k] += allocation[i][k];

}

finish[i] = 1;

found = 1;

count++;

}

}

}

if (found == 0) {

return 0; // No process can finish, not safe state

}

}

return 1; // All processes can finish, safe state

}

int main() {

int process\_num, request[MAX\_RESOURCES];

printf("Enter process number (0 to 4): ");

scanf("%d", &process\_num);

printf("Enter resource request (e.g., 0 1 0): ");

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

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

}

if (request\_resources(process\_num, request)) {

printf("Request granted.\n");

} else {

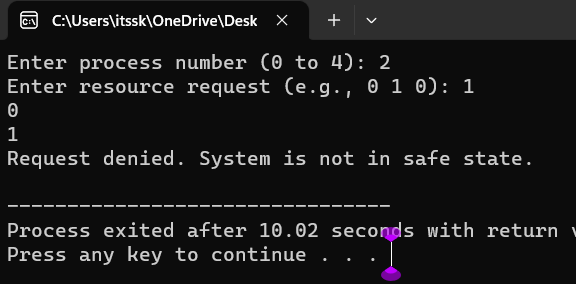
printf("Request denied. System is not in safe state.\n");

}

return 0;

}

**OUTPUT :**

****

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

**AIM :**

To construct a C program to simulate producer consumer problem using semaphores.

**ALGORITHM :**

1. **Include Libraries:** Include necessary libraries such as **stdio.h**, **stdlib.h**, **pthread.h** for threads, and **semaphore.h** for semaphores.
2. **Define Constants:** Define constants such as the size of the buffer (maximum number of items), number of producer and consumer threads, etc.
3. **Declare Global Variables:** Declare global variables including the buffer (an array to hold the items), indices to track the next position for inserting and removing items, and semaphores for synchronization.
4. **Initialize Semaphores:** Initialize semaphores for controlling access to the buffer, tracking empty spaces in the buffer, and tracking available items in the buffer.
5. **Create Producer and Consumer Threads:** Create threads for producers and consumers. Each thread should have its own unique identifier (for example, an integer value).
6. **Define Producer and Consumer Functions:** Implement functions for producers and consumers. These functions will be executed by the corresponding threads. The producer function will generate items and insert them into the buffer, while the consumer function will remove items from the buffer.
7. **Implement Buffer Operations:** Implement functions for inserting items into the buffer (enqueue operation) and removing items from the buffer (dequeue operation). Use semaphores to control access to the buffer and update the indices accordingly.
8. **Synchronize Producer and Consumer Threads:** Use semaphores to synchronize the producer and consumer threads. The producer should wait if the buffer is full, and the consumer should wait if the buffer is empty.
9. **Handle Thread Joining and Cleanup:** After creating the threads, ensure that the main program waits for all threads to finish their execution. Use **pthread\_join** for this purpose. Also, clean up any resources allocated during the program execution.
10. **Compile and Run:** Compile the C program using a C compiler (such as **gcc**) and run the executable. Observe the behavior of the producer and consumer threads, ensuring that they are properly synchronized and the buffer operations are correctly implemented.

**PROGRAM :**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include<Windows.h>

#define BUFFER\_SIZE 5

#define MAX\_ITEMS 10 // Maximum number of items to be produced/consumed

int buffer[BUFFER\_SIZE];

sem\_t empty, full;

int produced\_items = 0, consumed\_items = 0;

void\* producer(void\* arg) {

while (produced\_items < MAX\_ITEMS) {

sem\_wait(&empty);

// Critical section: add item to buffer

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

if (buffer[i] == 0) {

buffer[i] = produced\_items + 1;

printf("Produced: %d\n", buffer[i]);

produced\_items++;

break;

}

}

sem\_post(&full);

Sleep(1); // Sleep for a while

}

return NULL;

}

void\* consumer(void\* arg) {

while (consumed\_items < MAX\_ITEMS) {

sem\_wait(&full);

// Critical section: remove item from buffer

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

if (buffer[i] != 0) {

printf("Consumed: %d\n", buffer[i]);

buffer[i] = 0;

consumed\_items++;

break;

}

}

sem\_post(&empty);

Sleep(2); // Sleep for a while

}

return NULL;

}

int main() {

pthread\_t producer\_thread, consumer\_thread;

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

// Create producer and consumer threads

pthread\_create(&producer\_thread, NULL, producer, NULL);

pthread\_create(&consumer\_thread, NULL, consumer, NULL);

// Wait for threads to finish

pthread\_join(producer\_thread, NULL);

pthread\_join(consumer\_thread, NULL);

// Destroy semaphores

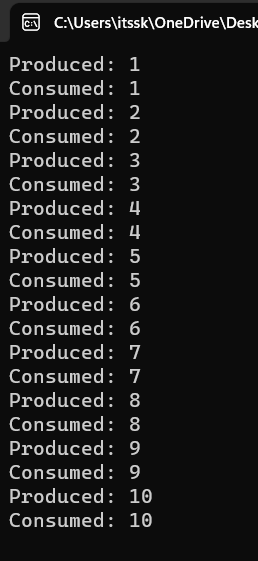
sem\_destroy(&empty);

sem\_destroy(&full);

return 0;

}

**OUTPUT :**

****

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

**AIM:**

To design a C program to implement process synchronization using mutex locks.

**ALGORITHM :**

**Step 1: Include Necessary Libraries:** Include the required header files for pthreads and mutex locks.

**Step 2: Declare Global Variables:** Declare any global variables needed for synchronization, such as mutex variables.

**Step 3: Initialize Mutex:** In the **main** function or initialization function, initialize the mutex using **pthread\_mutex\_init** function.

**Step 4: Define Functions:** Define functions that represent the actions of threads. These functions should include the critical sections where the mutex lock is acquired and released.

**Step 5: Create Threads:** In the **main** function or any other appropriate function, create threads and assign the functions to execute for each thread. Pass **NULL** or any necessary data as arguments to the functions.

**Step 6: Implement Mutex Synchronization:** Inside the functions that represent the actions of threads, use **pthread\_mutex\_lock** to acquire the mutex lock and **pthread\_mutex\_unlock** to release the lock. This ensures that only one thread can execute the critical section at a time.

**Step 7: Join Threads and Cleanup:** In the **main** function or any other appropriate function, wait for the threads to finish using **pthread\_join**. After the threads have finished their execution, destroy the mutex using **pthread\_mutex\_destroy** function.

**Step 8: Compile and Run:** Compile the program using a C compiler with the appropriate flags (for example, **-pthread** for GCC) to link the pthread library. Then, run the compiled executable to observe the synchronized behavior of threads due to mutex locks.

**PROGRAM :**

#include <stdio.h>

#include <pthread.h>

// Shared variables

int counter = 0;

pthread\_mutex\_t mutex;

// Function to be executed by threads

void \*threadFunction(void \*arg) {

int i;

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

// Lock the mutex before accessing the shared counter

pthread\_mutex\_lock(&mutex);

// Critical section: Increment the shared counter

counter++;

// Unlock the mutex after accessing the shared counter

pthread\_mutex\_unlock(&mutex);

}

return NULL;

}

int main() {

// Initialize the mutex

pthread\_mutex\_init(&mutex, NULL);

// Create two threads

pthread\_t thread1, thread2;

pthread\_create(&thread1, NULL, threadFunction, NULL);

pthread\_create(&thread2, NULL, threadFunction, NULL);

// Wait for the threads to finish

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

// Destroy the mutex

pthread\_mutex\_destroy(&mutex);

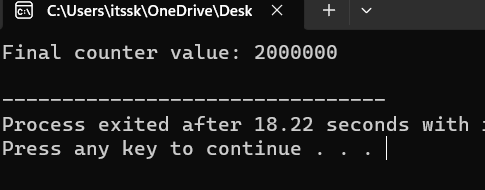
// Print the final value of the counter

printf("Final counter value: %d\n", counter);

return 0;

}

**OUTPUT :**

****

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

**AIM :**

To construct a C program to simulate Reader-Writer problem using semaphores

**ALGORITHM :**

1. **Include Libraries:** Include necessary libraries for using semaphores, threads, and other required functionalities.
2. **Initialize Semaphores:** Create semaphores to control access to the shared resources:
   * **Semaphore for Readers Count:** Initialize a semaphore to 1 (binary semaphore).
   * **Semaphore for Writers Count:** Initialize a semaphore to 1 (binary semaphore).
   * **Semaphore for Readers Waiting:** Initialize a semaphore to 1 (binary semaphore).
   * **Semaphore for Writers Waiting:** Initialize a semaphore to 1 (binary semaphore).
   * **Semaphore for Mutex:** Initialize a semaphore to 1 (binary semaphore).
3. **Reader Function:** Create a function for readers to execute. This function should handle the logic for readers accessing the shared resource.
4. **Writer Function:** Create a function for writers to execute. This function should handle the logic for writers accessing the shared resource.
5. **Implement Reader-Writer Logic:** Inside the reader and writer functions, implement the logic that ensures proper synchronization using semaphores. Readers should check and update the readers count semaphore and writers should check and update the writers count semaphore.
6. **Create Threads:** In your main function, create multiple threads for readers and writers to simulate concurrent access.
7. **Join Threads:** Use thread joining functions to wait for all threads to complete their execution.
8. **Clean Up:** Destroy the semaphores and perform any necessary clean-up operations before exiting the program.

**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(&mutex);

readersCount++;

if (readersCount == 1) {

sem\_wait(&writeBlock);

}

sem\_post(&mutex);

// Reading operation

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

sem\_wait(&mutex);

readersCount--;

if (readersCount == 0) {

sem\_post(&writeBlock);

}

sem\_post(&mutex);

i++;

}

}

void \*writer(void \*arg) {

int i=0;

while (i<10) {

sem\_wait(&writeBlock);

// Writing operation

data++;

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

sem\_post(&writeBlock);

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);

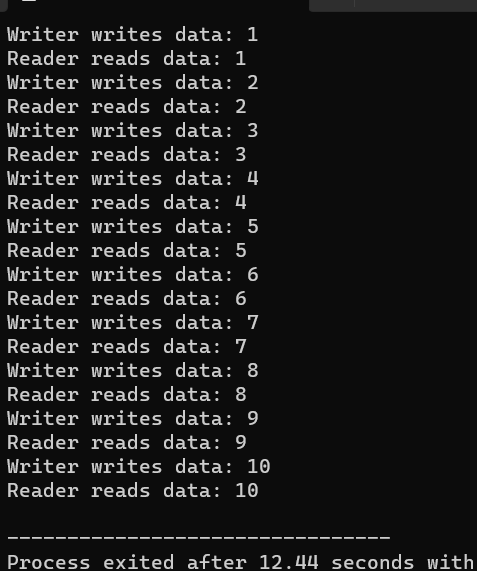
sem\_destroy(&mutex);

sem\_destroy(&writeBlock);

return 0;

}

**OUTPUT :**

****