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**

**Program:**

#include<stdio.h>

#include<unistd.h>

int main() {

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

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

return 0;

}

**Output:**

Process ID: 6899

Parent Process ID: 6892

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

**Program:**

#include <stdio.h>

#include <stdlib.h>

int main() {

FILE \*fptr1, \*fptr2;

char source\_filename[100], dest\_filename[100], c;

// Prompt user to enter the source filename

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

scanf("%s", source\_filename);

// Open the source file for reading

fptr1 = fopen(source\_filename, "r");

if (fptr1 == NULL) {

printf("Cannot open file %s for reading.\n", source\_filename);

exit(1); // Exit with non-zero status if file cannot be opened

}

// Prompt user to enter the destination filename

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

scanf("%s", dest\_filename);

// Open the destination file for writing

fptr2 = fopen(dest\_filename, "w");

if (fptr2 == NULL) {

printf("Cannot open file %s for writing.\n", dest\_filename);

fclose(fptr1); // Close the first file if the second cannot be opened

exit(1); // Exit with non-zero status if file cannot be opened

}

// Copy contents from source file to destination file

c = fgetc(fptr1);

while (c != EOF) {

fputc(c, fptr2);

c = fgetc(fptr1);

}

// Display success message

printf("\nContents copied to %s\n", dest\_filename);

// Close the files

fclose(fptr1);

fclose(fptr2);

return 0;

}

**Output:**

Enter the filename to open for reading:

vamsi

Cannot open file vamsi for reading.

1. **Design a CPU scheduling program with C using First Come First Served**

**Program:**

#include <stdio.h>

int main() {

int A[100][4]; // Array to store process ID, Burst Time, Waiting Time, and Turnaround Time

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

float avg\_wt, avg\_tat;

// Taking input for the number of processes

printf("Enter number of processes: ");

scanf("%d", &n);

// Taking input for Burst Time of each process

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; // Store process ID

}

// Sorting processes based on Burst Time (SJF)

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

index = i;

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

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

index = j;

}

}

// Swap the burst times

temp = A[i][1];

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

A[index][1] = temp;

// Swap the process IDs

temp = A[i][0];

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

A[index][0] = temp;

}

A[0][2] = 0; // Waiting time for the first process is 0

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

A[i][2] = 0; // Initializing the waiting time for each process

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

A[i][2] += A[j][1]; // Calculate the waiting time for each process

}

total += A[i][2]; // Sum up the waiting times for average calculation

}

avg\_wt = (float)total / n; // Calculate average waiting time

total = 0; // Reset total for turnaround time calculation

// Print the process table and calculate turnaround time

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

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

A[i][3] = A[i][1] + A[i][2]; // Calculate turnaround time (TAT = BT + WT)

total += A[i][3]; // Sum up the turnaround times for average calculation

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

}

avg\_tat = (float)total / n; // Calculate average turnaround time

// Print the averages

printf("Average Waiting Time = %.2f\n", avg\_wt);

printf("Average Turnaround Time = %.2f\n", avg\_tat);

return 0;

}

**Output:**

Enter number of processes: 3

Enter Burst Time:

P1: 4

P2: 5

P3: 6

P BT WT TAT

P1 4 0 4

P2 5 4 9

P3 6 9 15

Average Waiting Time = 4.33

Average Turnaround Time = 9.33

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

**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;

// Input the number of processes

printf("Enter number of processes: ");

scanf("%d", &n);

// Input the burst time for each process

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

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

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

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

p[i] = i + 1; // Assign process number

}

// Sorting burst time and processes using selection sort (Shortest Job First)

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

pos = i;

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

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

pos = j;

}

}

// Swap burst time

temp = bt[i];

bt[i] = bt[pos];

bt[pos] = temp;

// Swap process number

temp = p[i];

p[i] = p[pos];

p[pos] = temp;

}

// Initialize waiting time for the first process

wt[0] = 0;

// Calculate waiting time for each process

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

wt[i] = 0;

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

wt[i] += bt[j];

}

total += wt[i];

}

// Calculate average waiting time

avg\_wt = (float)total / n;

total = 0; // Reset total for turnaround time calculation

// Print process details

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

tat[i] = bt[i] + wt[i]; // Turnaround time = Burst time + Waiting time

total += tat[i]; // Add turnaround time to total

printf("P%d\t%d\t\t%d\t\t%d\n", p[i], bt[i], wt[i], tat[i]);

}

// Calculate average turnaround time

avg\_tat = (float)total / n;

// Print the average waiting and turnaround times

printf("\nAverage Waiting Time = %.2f", avg\_wt);

printf("\nAverage Turnaround Time = %.2f\n", avg\_tat);

return 0;

}

**Output:**

Enter number of processes: 3

Enter Burst Time:

P1: 9

P2: 8

P3: 7

Process Burst Time Waiting Time Turnaround Time

P3 7 0 7

P2 8 7 15

P1 9 15 24

Average Waiting Time = 7.33

Average Turnaround Time = 15.33

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

**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; // ASCII value for 'A'

int position;

float average\_waiting\_time;

float average\_turnaround\_time;

// Input the total number of processes

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

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

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

// Input the burst time and priority for each 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++;

}

// Sort processes based on priority (higher priority comes first)

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;

}

// Swap the processes

temp\_process = process[i];

process[i] = process[position];

process[position] = temp\_process;

}

// Calculate waiting time for each 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;

}

// Calculate average waiting time

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

total = 0; // Reset total for turnaround time calculation

// Output process details and calculate turnaround time

printf("\n\nProcess\_name \t Burst Time \t Waiting Time \t Turnaround Time\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\n", process[i].process\_name, process[i].burst\_time,

process[i].waiting\_time, process[i].turn\_around\_time);

}

// Calculate average turnaround time

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

// Output average waiting time and turnaround time

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

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

return 0;

}

**Output:**

Enter the total number of Processes: 3

Please Enter the Burst Time and Priority of each process:

Enter the details of the process A

Enter the burst time: 9

Enter the priority: 1

Enter the details of the process B

Enter the burst time: 8

Enter the priority: 2

Enter the details of the process C

Enter the burst time: 7

Enter the priority: 3

Process\_name Burst Time Waiting Time Turnaround Time

C 7 0 7

B 8 7 15

A 9 15 24

Average Waiting Time : 7.333333

Average Turnaround Time: 15.333333

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

**Program:**

**#include<stdio.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;

// Input the number of processes

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

scanf("%d", &NOP);

y = NOP;

// Input arrival and burst time for each process

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

printf("\nEnter 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];

}

// Input time quantum

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

scanf("%d", &quant);

// Printing the table header

printf("\nProcess No \t\t Burst Time \t\t Turnaround Time \t Waiting Time\n");

// Main round-robin scheduling loop

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;

}

// Process completed

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]; // Waiting time = Turnaround Time - Burst Time

tat = tat + sum - at[i]; // Turnaround Time = Completion Time - Arrival Time

count = 0;

}

// Move to the next process

if(i == NOP - 1) {

i = 0;

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

i++;

} else {

i = 0;

}

}

// Calculating average waiting time and turnaround time

avg\_wt = wt \* 1.0 / NOP;

avg\_tat = tat \* 1.0 / NOP;

// Printing the average times

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

printf("\nAverage Waiting Time: \t%f", avg\_wt);

return 0;

}

**Output:**

Total number of processes in the system: 3

Enter the Arrival and Burst time of the Process[1]

Arrival time is: 0

Burst time is: 10

Enter the Arrival and Burst time of the Process[2]

Arrival time is: 2

Burst time is: 5

Enter the Arrival and Burst time of the Process[3]

Arrival time is: 4

Burst time is: 8

Enter the Time Quantum for the process: 4

Process No Burst Time Turnaround Time Waiting Time

Process No[1] 10 10 0

Process No[2] 5 10 5

Process No[3] 8 12 4

Average Turnaround Time: 10.666667

Average Waiting Time: 3.000000

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

**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;

// Input the number of processes

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

scanf("%d", &n);

// Input arrival time and burst time for each process

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

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

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

pr[i] = i + 1;

}

// Sorting processes based on arrival time

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;

}

}

}

// Printing the table header

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

// Main scheduling loop

while(over < n) {

count = 0;

// Find processes that have arrived by the current time

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

if(at[i] <= time) {

count++;

} else {

break;

}

}

// If more than one process is available, sort by burst time (Shortest Job First)

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 of the current process

start = time;

time += bt[over]; // Update current time by adding burst time of the current process

// Printing process details

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]);

// Calculating waiting time and turnaround time

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

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

over++;

}

// Calculating average waiting time and turnaround time

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

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

// Printing average times

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

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

return 0;

}

**Output:**

Enter the number of processes

3

Enter the arrival time and execution time for process 1: 0 5

Enter the arrival time and execution time for process 2: 2 3

Enter the arrival time and execution time for process 3: 4 2

Process | Arrival time | Execution time | Start time | End time | Waiting time | Turnaround time

p[1] | 0 | 5 | 0 | 5 | 0 | 5

p[3] | 4 | 2 | 5 | 7 | 1 | 3

p[2] | 2 | 3 | 7 | 10 | 4 | 8

Average waiting time is: 1.666667

Average turnaround time is: 5.333333

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

**Program:**

#include<stdio.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;

// Input the total number of processes

printf("Enter the total number of processes in the system: ");

scanf("%d", &NOP);

y = NOP;

// Input the arrival and burst time for each process

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

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

printf("Arrival time: ");

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

printf("Burst time: ");

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

temp[i] = bt[i]; // Copy burst time to temporary array

}

// Input the time quantum for the Round Robin algorithm

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

scanf("%d", &quant);

// Printing the table header

printf("\nProcess No\tBurst Time\tWaiting Time\tTurnaround Time\n");

// Round Robin Scheduling

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 the process is completed

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

y--;

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

wt = wt + sum - at[i] - bt[i]; // Waiting Time calculation

tat = tat + sum - at[i]; // Turnaround Time calculation

count = 0;

}

// Check if we need to move to the next process

if(i == NOP - 1) {

i = 0;

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

i++;

} else {

i = 0;

}

}

// Calculate average waiting time and turnaround time

avg\_wt = (float)wt / NOP;

avg\_tat = (float)tat / NOP;

// Print the average times

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

printf("\nAverage Waiting Time: %f", avg\_wt);

return 0;

}

**Output**:

Enter the total number of processes in the system: 3

Enter the Arrival and Burst time of Process[1]:

Arrival time: 0

Burst time: 5

Enter the Arrival and Burst time of Process[2]:

Arrival time: 2

Burst time: 3

Enter the Arrival and Burst time of Process[3]:

Arrival time: 4

Burst time: 2

Enter the Time Quantum for the process: 3

Process No Burst Time Waiting Time Turnaround Time

Process No[1] 5 0 5

Process No[2] 3 3 6

Process No[3] 2 4 6

Average Turnaround Time: 5.666667

Average Waiting Time: 2.333333

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

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

if (key == -1) {

perror("ftok");

exit(EXIT\_FAILURE);

}

// 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:**

Data written to shared memory: Hello, shared memory!

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

**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() {

// Generate a unique key for the message queue

key\_t key = ftok("msgqfile", 65);

// 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:

Producer: Data sent to message queue: Hello, message queue!

Consumer: Data received from message queue: Hello, message queue!