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
```

2. **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.
```

3. 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
```

4. 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
```

5. **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
```

6. 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!
```