PRACTICAL NO:4

AIM :Write a c program to create a child process.

```c
#include <unistd.h>
#include <stdlib.h>

int main() {
    pid_t p = fork();
    if (p < 0) {
        perror("fork failed");
        exit(1);
    } else if (p == 0) {

        printf("Hello from Child! Process ID (PID): %d\n", getpid());
    } else {

        printf("Hello from Parent! Process ID (PID): %d\n", getpid());
    }
    return 0;
}

```

Output

```
/tmp/USn5o314sN.o
Hello from Parent! Process ID (PID): 7773
Hello from Child! Process ID (PID): 7774
```

PRACTICAL NO:7

AIM: a) Write a Shell script to determine whether a given ile exists or not.
b) Write a Shell script that prints the even number till the number given by the user.

A file is a collection of related data that represents a complete unit of information about something (e.g., a document, image, program).
Files can be stored physically (e.g., paper documents) or electronically (e.g., computer files).
Types of Files:

Logical Files: Focus on the data content and operations performed on it, independent of storage details.
Physical Files: Focus on how data is stored on a device and accessed by the computer.
Checking File Existence with Shell Scripts:

Use shell scripts with parameters to check if a file exists.
Common parameters:
-f: Checks for regular files.
-d: Checks for directories.
-e: Checks for any type of file.
Others like -r (readable), -w (writable), -x (executable) for specific file properties.
Remember:

Logical vs. Physical files provide different perspectives on file storage.
Shell scripts offer flexibility for checking file existence based on various criteria.

(A) Write a Shell script to determine whether the ile exits or not. Code:

```
# Prompt the user to enter the file name
echo "Enter the file name: "
read filename

# Check if the file exists
if [ -e "$filename" ]; then
    echo "File $filename exists."
else
    echo "File $filename does not exist."
fi
Enter the file name:
CRM
File CRM exists.
```

OUTPUT:

```
Enter filename: abc.txt
abc.txt does not exist.

Enter filename: pyramid.sh
pyramid.sh exists.
```

B) Write a Shell script that prints the even number till the number given by the user.

```
# Prompt the user to enter a number
echo "Enter a number: "
read num

# Function to check if a number is even
is_even() {
    if [ $(( $1 % 2 )) -eq 0 ]; then
        return 0  # Even
    else
        return 1  # Odd
    fi
}

# Loop to print even numbers up to the given number
echo "Even numbers up to $num are:"
for ((i = 0; i <= $num; i++)); do
    is_even $i
    if [ $? -eq 0 ]; then
        echo $i
    fi
done
```

OUTPUT:

```
Enter a number:
25
Even numbers up to 25 are:
0
2
4
6
8
10
12
14
16
18
20
22
24
```

PRACTICAL NO:8

AIM: Write a program for process creation using c (use gcc compiler)

THEORY:

GCC (GNU Compiler Collection) is a free and open-source compiler system.

It can compile code written in various languages like C, C++, and FORTRAN.

Key Features:

Supports multiple languages: Compile code in different languages with one tool.

Optimization: Improves code performance for speed or size.

Portable: Works on various platforms (Linux, Windows, etc.) with minimal changes.

Open source: Freely available for anyone to use, modify, and distribute.

Standards compliant: Ensures code works across different systems.

Extensible: Add custom functionalities through plugins.

How it Works:

Preprocesses: Handles header files and removes comments.

Compiles: Generates assembly code from the preprocessed code.

Assembles: Converts assembly code to machine code.

Links: Combines compiled code with libraries to create an executable.

Significance:

Industry standard compiler for various languages.

Large ecosystem of libraries, tools, and documentation.

Benefits from a global developer community for improvements.

Empowers developers to create efficient and portable software.

Basic Syntax:

gcc [-c|-S|-E] [-std=standard] source_file.c -o output_file

PROGRAM TO CREATE A CHILD PROCESS:

Code:

```c
#include <stdio.h>
#include <unistd.h> // for fork() function

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        // Error handling for fork failure
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Name: Krish Amit Maheshbhai Kadam, ERP No: 2203031241412\n");
        printf("I am a Child Process.\n");
        printf("ID: %d\n", getpid()); // Print child process ID
    } else {
        // Parent process
        wait(NULL); // Wait for child process to finish (optional)
    }

    return 0;
}
```

OUTPUT

```
Name: Krish Amit Maheshbhai Kadam, ERP No: 2203031241412
I am a Child Process.
ID: 995
```

PRACTICAL NO:9

AIM: Write a program to show the implementation of FCFS and Round Robin Algorithm.

THEORY:

FCFS (First Come First Served) is a CPU scheduling algorithm that works on a first-in, first-out (FIFO) basis. Processes are queued and executed in the order they arrive.

Advantages:

Simple and easy to implement.

Fair - no process starvation (preemptive version).

Disadvantages:

Long waiting times for short processes behind long ones (convoy effect).

Doesn't consider process priority.

Not very efficient overall.

Implementation:

Processes are added to a queue based on arrival time.

The CPU processes them one by one until completion.

FCFS CODE:

```c
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
  wt[0] = 0;

  for (int i = 1; i < n; i++) {
    wt[i] = bt[i - 1] + wt[i - 1];
  }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
  for (int i = 0; i < n; i++) {
    tat[i] = bt[i] + wt[i];
  }
}

void findAverageTime(int processes[], int n, int bt[]) {
  int wt[n], tat[n], total_wt = 0, total_tat = 0;

  findWaitingTime(processes, n, bt, wt);
  findTurnAroundTime(processes, n, bt, wt, tat);

  printf("Process | Burst Time | Waiting Time | Turn Around Time\n");
  for (int i = 0; i < n; i++) {
    total_wt = total_wt + wt[i];
    total_tat = total_tat + tat[i];
    printf(" %d ", processes[i]);
    printf("|   %d      ", bt[i]);
    printf("|    %d     ", wt[i]);
    printf("|    %d \n", tat[i]);
  }

  float avg_wt = (float)total_wt / n;
  float avg_tat = (float)total_tat / n;
  printf("Average Waiting Time = %f \n", avg_wt);
  printf("Average Turnaround Time = %f \n", avg_tat);
}

int main() {
  int processes[] = {1, 2, 3};  // Process IDs
  int n = sizeof(processes) / sizeof(processes[0]);
  int burst_time[] = {10, 5, 8};  // Burst times for each process

  findAverageTime(processes, n, burst_time);

  return 0;
}
```

OUTPUT:

```
/tmp/rs9ehNzkf2.o
Process | Burst Time | Waiting Time | Turn Around Time
 1 |   10      |    0     |    10
 2 |   5       |    10    |    15
 3 |   8       |    15    |    23
Average Waiting Time = 8.333333
Average Turnaround Time = 16.000000


=== Code Execution Successful ===
```

Round Robin (RR) is a preemptive CPU scheduling algorithm that simulates time-sharing. Processes are allocated the CPU for a fixed time slice called a time quantum.

Characteristics:

Simple and easy to implement.

Starvation-free - all processes get a chance to run eventually.

Commonly used technique for fair CPU allocation.

Advantages:

Provides fairness - all processes get a share of the CPU.

Good for interactive systems with short response time requirements.

Disadvantages:

Increased context switching overhead due to frequent preemption.

May lead to longer waiting times for longer processes.

Throughput (number of processes completed) may be lower than FCFS for some workloads.

Implementation:

Processes are placed in a ready queue.

The CPU is assigned to the first process in the queue for the time quantum.

If the process finishes within the time quantum, it exits the CPU.

If the process doesn't finish, it's placed at the back of the queue and the next process gets the CPU.

This cycle repeats until all processes finish.

```c
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum) {
  int rem_bt[n]; // Remaining burst time for each process
  for (int i = 0; i < n; i++) {
    rem_bt[i] = bt[i];
  }
  int t = 0; // Current time

  while (1) {
    int done = 1;
    for (int i = 0; i < n; i++) {
      if (rem_bt[i] > 0) {
        done = 0; // There is a process in ready queue

        if (rem_bt[i] > quantum) {
          // Reduce remaining burst time by quantum
          rem_bt[i] -= quantum;
          t += quantum;
        } else {
          t += rem_bt[i];
          wt[i] = t - bt[i]; // Waiting time for this process is current time minus its bu
          rem_bt[i] = 0; // Mark this process as complete
        }
      }
    }

    if (done == 1) {
      break;
    }
  }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
  // Turn around time for a process is the sum of burst time and waiting time
  for (int i = 0; i < n; i++) {
    tat[i] = bt[i] + wt[i];
  }
}

void findAverageTime(int processes[], int n, int bt[], int quantum) {
```

```
  for (int i = 0; i < n; i++) {
    tat[i] = bt[i] + wt[i];
  }
}

void findAverageTime(int processes[], int n, int bt[], int quantum) {
  int wt[n], tat[n];

  // Function calls to calculate waiting time and turnaround time
  findWaitingTime(processes, n, bt, wt, quantum);
  findTurnAroundTime(processes, n, bt, wt, tat);

  // Display processes, burst time, waiting time, and turnaround time
  printf("Process | Burst Time | Waiting Time | Turn Around Time\n");
  for (int i = 0; i < n; i++) {
    printf(" %d ", processes[i]);
    printf("|    %d       ", bt[i]);
    printf("|     %d      ", wt[i]);
    printf("|     %d \n", tat[i]);
  }

  // Calculate average waiting time and turnaround time
  float avg_wt = 0, avg_tat = 0;
  for (int i = 0; i < n; i++) {
    avg_wt += wt[i];
    avg_tat += tat[i];
  }
  avg_wt = (float)avg_wt / n;
  avg_tat = (float)avg_tat / n;
  printf("Average Waiting Time = %f \n", avg_wt);
  printf("Average Turnaround Time = %f \n", avg_tat);
}

int main() {
  int processes[] = {1, 2, 3};  // Process IDs
  int n = sizeof(processes) / sizeof(processes[0]);
  int burst_time[] = {10, 5, 8};  // Burst times for each process
  int quantum = 2;                // Time quantum

  findAverageTime(processes, n, burst_time, quantum);

  return 0;
}
```

OUTPUT:

```
/tmp/rs9eWNzKf2.o
Process | Burst Time | Waiting Time | Turn Around Time
 1 |    10       |     0      |     10
 2 |    5        |     10     |     15
 3 |    8        |     15     |     23
Average Waiting Time = 8.333333
Average Turnaround Time = 16.000000
```

PRACTICAL NO:10

AIM: Write a program to show the implementation of Banker's Algorithm.

Banker's Algorithm is a deadlock avoidance algorithm used in operating systems for resource allocation. It ensures safe allocation of resources to processes to prevent deadlocks.

Data Structures:

Available: Available resources in the system (vector).

Max: Maximum resources each process can request (matrix).

Allocation: Currently allocated resources to processes (matrix).

Need: Resources still needed by each process (matrix = Max - Allocation).

Finish: Indicates if a process has finished execution (boolean vector).

Safety Algorithm:

Checks if a system is in a safe state (can finish all processes without deadlock).

Uses Work (available resources) and Finish vectors.

Iterates through processes:

If Need[i] <= Work and Finish[i] is False, it's safe.

Allocate resources to process i and update Work and Finish.

If all processes can be finished, the system is safe.

Resource Request Algorithm:

Checks if a resource request from a process can be granted safely.

Verifies if the request is within process's Need and available resources (Available).

If safe, allocates resources and updates Available, Allocation, and Need.

If unsafe, the process waits for resources.

Advantages:

Prevents deadlocks by ensuring safe resource allocation.

Disadvantages:

Requires processes to declare maximum resource needs upfront (not always realistic).

Overhead for maintaining resource tables.

Implementation:

Complex and requires careful design.

Used in multiprogramming systems for efficient resource management.

CODE FOR BANKER THEROM:

```c
#include <stdio.h>
#include <stdbool.h>

#define NUM_RESOURCES 3
#define NUM_PROCESSES 5

void calculate_need(int need[NUM_PROCESSES][NUM_RESOURCES], int max_resources[NUM_PROCESSES][NUM_RESOURCES], int
    allocated[NUM_PROCESSES][NUM_RESOURCES]) {
    for (int i = 0; i < NUM_PROCESSES; i++) {
        for (int j = 0; j < NUM_RESOURCES; j++) {
            need[i][j] = max_resources[i][j] - allocated[i][j];
        }
    }
}

bool is_safe(int processes[NUM_PROCESSES], int available[NUM_RESOURCES], int max_resources[NUM_PROCESSES][NUM_RESOURCES], int
    allocated[NUM_PROCESSES][NUM_RESOURCES]) {
    int need[NUM_PROCESSES][NUM_RESOURCES];
    calculate_need(need, max_resources, allocated);

    bool finish[NUM_PROCESSES] = { false };
    int work[NUM_RESOURCES];
    for (int i = 0; i < NUM_RESOURCES; i++) {
        work[i] = available[i];
    }

    int count = 0;
    while (count < NUM_PROCESSES) {
        bool found = false;
        for (int i = 0; i < NUM_PROCESSES; i++) {
            if (!finish[i]) {
                bool can_allocate = true;
                for (int j = 0; j < NUM_RESOURCES; j++) {
                    if (need[i][j] > work[j]) {
                        can_allocate = false;
                        break;
                    }
                }
```

```
                if (can_allocate) {
                    for (int j = 0; j < NUM_RESOURCES; j++) {
                        work[j] += allocated[i][j];
                    }
                    finish[i] = true;
                    found = true;
                    count++;
                }
            }
        }
        if (!found) {
            break;
        }
    }

    return count == NUM_PROCESSES;
}

int main() {
    int processes[NUM_PROCESSES] = {0, 1, 2, 3, 4};
    int available[NUM_RESOURCES] = {3, 3, 2};
    int max_resources[NUM_PROCESSES][NUM_RESOURCES] = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };
    int allocated[NUM_PROCESSES][NUM_RESOURCES] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };

    if (is_safe(processes, available, max_resources, allocated)) {
        printf("System is in a safe state\n");
    } else {
        printf("System is in an unsafe state\n");
    }

    return 0;
}
```

OUTPUT:

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
System is in a safe state
```