**Q1. Write a program in “C” Language to demonstrate the working of fork() system call.**

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid;

pid=fork();

if(pid==0)

{

printf("\nInside If\n");

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

}

else if(pid>0)

{

printf("Inside Else If\n");

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

}

else{

printf("\nFork Failed i.e,(pid<0)\n");

}

}

**OUTPUT:**

Inside Else If

Parent Process PID: 47008

Inside If

Child Process PID: 47009

**Q2. Write a program in “C” Language in which Parent Process Computes the SUM OF ODD NUMBERS and Child Process Computes the sum of EVEN NUMBERS stored in array using fork().**

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t x;

x=fork();

int arr[]={2,3,5,7,8,9,12};

int size=sizeof(arr)/sizeof(arr[0]);

int esum=0,osum=0;

if(x==0) //child

{

for(int i=0;i<size;i++)

{

if(arr[i]%2==0)

{

esum+=arr[i];

}

}

printf("Even Sum: %d\n",esum);

}

else if(x>0) //parent

{

for(int i=0;i<size;i++)

{

if(arr[i]%2!=0)

{

osum+=arr[i];

}

}

printf("Odd Sum: %d\n",osum);

}

else{

printf("Fork Failed.");

}

}

**OUTPUT:**

Odd Sum: 24

Even Sum: 22

**Q3. Write a program in “C” Language to implement the Zombie Process.**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

pid\_t pid = fork();

if (pid == 0)

{

printf("Child process (PID %d) is running.\n", getpid());

sleep(3);

printf("Child process (PID %d) is exiting.\n", getpid());

exit(0);

}

else

{

printf("Parent process (PID %d) created a child process (PID %d).\n", getpid(),

pid);

printf("Parent process is sleeping for a while to allow the child to become a

zombie...\n");

sleep(20); // Sleep to allow the child to become a zombie

printf("Parent process (PID %d) is exiting.\n", getpid());

}

}

**OUTPUT:**

Parent process (PID 47480) created a child process (PID 47481).

Parent process is sleeping for a while to allow the child to become a

zombie...

Child process (PID 47481) is running.

Child process (PID 47481) is exiting.

Parent process (PID 47480) is exiting.

**Q4. Write a program in “C” Language to implement the Orphan Process.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

pid\_t pid = fork();

if (pid == 0)

{

printf("Child process (PID %d) is running.\n", getpid());

sleep(5); //initiated some work

printf("Child process (PID %d) is exiting.\n", getpid());

exit(0);

}

else

{

printf("Parent process (PID %d) created a child process (PID %d).\n", getpid(),

pid);

printf("Parent process is sleeping for a while...\n");

sleep(1); // Sleep briefly and then exit

printf("Parent process (PID %d) is exiting.\n", getpid());

}

}

**OUTPUT:**

Parent process (PID 47635) created a child process (PID 47636).

Parent process is sleeping for a while...

Child process (PID 47636) is running.

Parent process (PID 47635) is exiting.

**Q5. Write a program in “C” Language to Implement FCFS CPU Scheduling Algorithm.**

#include<stdio.h>

typedef struct process{

int id;

int arrival;

int burst;

int completion;

int turn\_around\_time;

int waiting;

int response\_time;

}process;

void sort(process\* processes, int n){

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

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

if(processes[j].arrival > processes[j+1].arrival){

process temp = processes[j];

processes[j] = processes[j+1];

processes[j+1] = temp;

}

}

}

}

void compute(process\* processes, int n){

sort(processes,n);

int currTime = 0;

int idleTime = 0;

float averageTurnAroundTime = 0;

float averageWaitingTime = 0;

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

if(currTime < processes[i].arrival){

idleTime += (processes[i].arrival - currTime);

currTime = processes[i].arrival;

}

processes[i].completion = currTime + processes[i].burst;

currTime = processes[i].completion;

processes[i].turn\_around\_time = processes[i].completion - processes[i].arrival;

processes[i].waiting = processes[i].turn\_around\_time - processes[i].burst;

processes[i].response\_time = processes[i].waiting;

averageTurnAroundTime += processes[i].turn\_around\_time;

averageWaitingTime += processes[i].waiting;

}

float cpu\_util = ((currTime - idleTime)/(1.0\*currTime))\*100;

averageTurnAroundTime /= (1.0\*n);

averageWaitingTime /= (1.0\*n);

float throughput = ((1.0)\*n)/currTime;

printf("\nCPU Utilization: %f\n",cpu\_util);

printf("Idle Time: %d\n",idleTime);

printf("Average Turn Around Time: %f\n",averageTurnAroundTime);

printf("Average Waiting Time: %f\n",averageWaitingTime);

printf("Throughput: %f: \n",throughput);

}

int main(){

int n;

scanf("%d",&n);

process processes[n];

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

processes[i].id = i + 1;

printf("Enter Arrival and Burst Time: ");

scanf("%d%d",&processes[i].arrival, &processes[i].burst);

}

compute(processes,n);

printf("\nPID\tAT\tBT\tCT\tTAT\tWT\tRT\n");

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

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].id, processes[i].arrival, processes[i].burst, processes[i].completion, processes[i].turn\_around\_time, processes[i].waiting, processes[i].response\_time);

}

}

**OUTPUT:**

A screenshot of a computer

Description automatically generated

**Q6. Write a program in “C” Language to implement SJF CPU Scheduling Algorithm.**

#include<stdio.h>

typedef struct process{

int id;

int arrival;

int burst;

int completion;

int turn\_around\_time;

int waiting;

int response\_time;

}process;

void sort(process\* processes, int n){

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

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

if(processes[j].burst > processes[j+1].burst){

process temp = processes[j];

processes[j] = processes[j+1];

processes[j+1] = temp;

}

}

}

}

void processSort(process\* processes, int n){

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

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

if(processes[j].id > processes[j+1].id){

process temp = processes[j];

processes[j] = processes[j+1];

processes[j+1] = temp;

}

}

}

}

void compute(process\* processes, int n){

sort(processes,n);

int currTime = 0;

int idleTime = 0;

float averageTurnAroundTime = 0;

float averageWaitingTime = 0;

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

if(currTime < processes[i].arrival){

idleTime += (processes[i].arrival - currTime);

currTime = processes[i].arrival;

}

processes[i].completion = currTime + processes[i].burst;

currTime = processes[i].completion;

processes[i].turn\_around\_time = processes[i].completion - processes[i].arrival;

processes[i].waiting = processes[i].turn\_around\_time - processes[i].burst;

processes[i].response\_time = processes[i].waiting;

averageTurnAroundTime += processes[i].turn\_around\_time;

averageWaitingTime += processes[i].waiting;

}

processSort(processes,n);

float cpu\_util = ((currTime - idleTime)/(1.0\*currTime))\*100;

averageTurnAroundTime /= (1.0\*n);

averageWaitingTime /= (1.0\*n);

float throughput = ((1.0)\*n)/currTime;

printf("\nCPU Utilization: %f\n",cpu\_util);

printf("Idle Time: %d\n",idleTime);

printf("Average Turn Around Time: %f\n",averageTurnAroundTime);

printf("Average Waiting Time: %f\n",averageWaitingTime);

printf("Throughput: %f: \n",throughput);

}

int main(){

int n;

scanf("%d",&n);

process processes[n];

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

processes[i].id = i + 1;

printf("Enter Arrival and Burst Time: ");

scanf("%d%d",&processes[i].arrival, &processes[i].burst);

}

compute(processes,n);

printf("\nPID\tAT\tBT\tCT\tTAT\tWT\tRT\n");

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

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].id, processes[i].arrival, processes[i].burst, processes[i].completion, processes[i].turn\_around\_time, processes[i].waiting, processes[i].response\_time);

}

}

**OUTPUT:**

**A screenshot of a computer program

Description automatically generated**

**Q6. Write a program in “C” Language to implement SRTF CPU Scheduling Algorithm.**

#include<stdio.h>

#include<stdbool.h>

#include<limits.h>

typedef struct process{

int id;

int arrival;

int burst;

int remaining;

int completion;

int turn\_around\_time;

bool completed;

int waiting;

int response\_time;

}process;

int findMinInd(process\* processes, int n, int CurrTime){

int ind = -1;

int shortestTime = INT\_MAX;

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

if(processes[i].remaining < shortestTime && processes[i].completed == false && processes[i].arrival <= CurrTime){

ind = i;

shortestTime = processes[i].remaining;

}

}

return ind;

}

int main(){

int n;

scanf("%d",&n);

process processes[n];

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

printf("Enter Arrival and Burst Time: ");

scanf("%d%d",&processes[i].arrival, &processes[i].burst);

processes[i].id = i + 1;

processes[i].completed = false;

processes[i].response\_time = -1;

processes[i].remaining = processes[i].burst;

}

int totalProcesses = 0;

int currTime = 0;

int idleTime = 0;

float averageTurnAroundTime = 0;

float averageWaitingTime = 0;

while(totalProcesses < n){

int minInd = findMinInd(processes, n, currTime);

if(minInd == -1){

currTime++;

idleTime++;

continue;

}

if(processes[minInd].response\_time == -1){

processes[minInd].response\_time = currTime;

}

processes[minInd].remaining--;

currTime++;

if(processes[minInd].remaining == 0){

processes[minInd].completed = true;

totalProcesses++;

processes[minInd].completion = currTime;

processes[minInd].turn\_around\_time = currTime - processes[minInd].arrival;

processes[minInd].waiting = processes[minInd].turn\_around\_time - processes[minInd].burst;

averageTurnAroundTime += processes[minInd].turn\_around\_time;

averageWaitingTime += processes[minInd].waiting;

}

}

float cpu\_util = ((currTime - idleTime)/(1.0\*currTime))\*100;

averageTurnAroundTime /= (1.0\*n);

averageWaitingTime /= (1.0\*n);

float throughput = ((1.0)\*n)/currTime;

printf("\nPID\tAT\tBT\tCT\tTAT\tWT\tRT\n");

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

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].id, processes[i].arrival, processes[i].burst, processes[i].completion, processes[i].turn\_around\_time, processes[i].waiting, processes[i].response\_time);

}

printf("CPU Utilization: %f\n",cpu\_util);

printf("Idle Time: %d\n",idleTime);

printf("Average Turn Around Time: %f\n",averageTurnAroundTime);

printf("Average Waiting Time: %f\n",averageWaitingTime);

printf("Throughput: %f: \n",throughput);

}

**OUTPUT:**

**A screenshot of a computer program

Description automatically generated**

**Q7. Write a program in “C” Language to Implement Pre-emptive Priority CPU Scheduling Algorithm.**

#include<stdio.h>

#include<limits.h>

#include<stdbool.h>

typedef struct process {

int id;

int priority;

int arrival;

int burst;

int completion;

int remaining;

int turn\_around\_time;

int waiting\_time;

int response\_time;

bool completed;

} process;

int maxPriorityInd(process\* processes, int n, int currTime){

int ind = -1;

int priority = INT\_MAX;

int arrival = INT\_MAX;

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

if(processes[i].priority <= priority && !processes[i].completed && processes[i].arrival <= currTime){

if(processes[i].priority == priority){

if(processes[i].arrival >= arrival){

continue;

}

}

ind = i;

priority = processes[i].priority;

arrival = processes[i].arrival;

}

}

return ind;

}

int main(){

int n;

scanf("%d", &n);

process processes[n];

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

printf("Enter Priority, Arrival And Burst Time Of Process %d: ", i + 1);

scanf("%d %d %d",&processes[i].priority,&processes[i].arrival, &processes[i].burst);

processes[i].id = i + 1;

processes[i].remaining = processes[i].burst;

processes[i].response\_time = -1;

processes[i].completed = false;

}

int currTime = 0;

int totalProcesses = 0;

int idleTime = 0;

float averageTurnAroundTime = 0;

float averageWaitingTime = 0;

currTime++;

processes[0].remaining--;

processes[0].response\_time = 0;

if(processes[0].remaining == 0){

processes[0].completed = true;

totalProcesses++;

processes[0].completion = currTime;

processes[0].turn\_around\_time = processes[0].completion - processes[0].arrival;

processes[0].waiting\_time = processes[0].turn\_around\_time - processes[0].burst;

averageTurnAroundTime += processes[0].turn\_around\_time;

averageWaitingTime += processes[0].waiting\_time;

}

while(totalProcesses < n){

int highestPriorityInd = maxPriorityInd(processes, n, currTime);

if(highestPriorityInd == -1){

currTime++;

idleTime++;

continue;

}

if(processes[highestPriorityInd].response\_time == -1){

processes[highestPriorityInd].response\_time = currTime;

}

currTime++;

processes[highestPriorityInd].remaining--;

if(processes[highestPriorityInd].remaining == 0){

processes[highestPriorityInd].completed = true;

totalProcesses++;

processes[highestPriorityInd].completion = currTime;

processes[highestPriorityInd].turn\_around\_time = processes[highestPriorityInd].completion - processes[highestPriorityInd].arrival;

processes[highestPriorityInd].waiting\_time = processes[highestPriorityInd].turn\_around\_time - processes[highestPriorityInd].burst;

averageTurnAroundTime += processes[highestPriorityInd].turn\_around\_time;

averageWaitingTime += processes[highestPriorityInd].waiting\_time;

}

}

float cpu\_util = ((currTime - idleTime)/(1.0\*currTime))\*100;

float throughput = ((1.0)\*n)/currTime;

averageTurnAroundTime /= (1.0\*n);

averageWaitingTime /= (1.0\*n);

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

processes[i].response\_time -= processes[i].arrival;

}

printf("\nPID\tPR\tAT\tBT\tCT\tTAT\tWT\tRT\n");

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

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].id, processes[i].priority,processes[i].arrival, processes[i].burst, processes[i].completion, processes[i].turn\_around\_time, processes[i].waiting\_time, processes[i].response\_time);

}

printf("CPU Utilization: %f\n",cpu\_util);

printf("Idle Time: %d\n",idleTime);

printf("Average Turn Around Time: %f\n",averageTurnAroundTime);

printf("Average Waiting Time: %f\n",averageWaitingTime);

printf("Throughput: %f: \n",throughput);

}

**OUTPUT:**

**A screenshot of a computer

Description automatically generated**

**Q7. Write a program in “C” Language to Implement Non-Preemptive Priority CPU Scheduling Algorithm.**

#include<stdio.h>

#include<limits.h>

#include<stdbool.h>

typedef struct process {

int id;

int priority;

int arrival;

int burst;

int completion;

int turn\_around\_time;

int waiting\_time;

int response\_time;

bool completed;

} process;

int maxPriorityInd(process\* processes, int n, int currTime){

int ind = -1;

int priority = INT\_MAX;

int arrival = INT\_MAX;

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

if(processes[i].priority <= priority && !processes[i].completed && processes[i].arrival <= currTime){

if(processes[i].priority == priority){

if(processes[i].arrival >= arrival){

continue;

}

}

ind = i;

priority = processes[i].priority;

arrival = processes[i].arrival;

}

}

return ind;

}

int main(){

int n;

scanf("%d", &n);

process processes[n];

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

printf("Enter Priority, Arrival And Burst Time Of Process %d: ", i + 1);

scanf("%d %d %d",&processes[i].priority,&processes[i].arrival, &processes[i].burst);

processes[i].id = i + 1;

processes[i].response\_time = -1;

processes[i].completed = false;

}

int currTime = 0;

int totalProcesses = 0;

int idleTime = 0;

float averageTurnAroundTime = 0;

float averageWaitingTime = 0;

currTime += processes[0].burst;

processes[0].response\_time = 0;

processes[0].completed = true;

processes[0].completion = currTime;

processes[0].turn\_around\_time = processes[0].completion - processes[0].arrival;

processes[0].waiting\_time = processes[0].turn\_around\_time - processes[0].burst;

averageTurnAroundTime += processes[0].turn\_around\_time;

averageWaitingTime += processes[0].waiting\_time;

totalProcesses++;

while(totalProcesses < n){

int highestPriorityInd = maxPriorityInd(processes, n, currTime);

if(highestPriorityInd == -1){

currTime++;

idleTime++;

continue;

}

if(processes[highestPriorityInd].response\_time == -1){

processes[highestPriorityInd].response\_time = currTime;

}

currTime += processes[highestPriorityInd].burst;

processes[highestPriorityInd].completed = true;

processes[highestPriorityInd].completion = currTime;

processes[highestPriorityInd].turn\_around\_time = processes[highestPriorityInd].completion - processes[highestPriorityInd].arrival;

processes[highestPriorityInd].waiting\_time = processes[highestPriorityInd].turn\_around\_time - processes[highestPriorityInd].burst;

averageTurnAroundTime += processes[highestPriorityInd].turn\_around\_time;

averageWaitingTime += processes[highestPriorityInd].waiting\_time;

totalProcesses++;

}

float cpu\_util = ((currTime - idleTime)/(1.0\*currTime))\*100;

averageTurnAroundTime /= (1.0\*n);

averageWaitingTime /= (1.0\*n);

float throughput = ((1.0)\*n)/currTime;

printf("\nPID\tPR\tAT\tBT\tCT\tTAT\tWT\tRT\n");

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

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].id, processes[i].priority,processes[i].arrival, processes[i].burst, processes[i].completion, processes[i].turn\_around\_time, processes[i].waiting\_time, processes[i].response\_time);

}

printf("CPU Utilization: %f\n",cpu\_util);

printf("Idle Time: %d\n",idleTime);

printf("Average Turn Around Time: %f\n",averageTurnAroundTime);

printf("Average Waiting Time: %f\n",averageWaitingTime);

printf("Throughput: %f: \n",throughput);

}

**OUTPUT:**

**A screenshot of a computer program

Description automatically generated**