**WEEK1**

**Program1.).**Write a program to create a child process using system call fork().

**SOURCE CODE:**

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

int main(){

pid\_t p\_id;

p\_id=fork();

if(p\_id<0){

printf("fork failed");

return 1;

}

else if(p\_id==0){

printf("i am child process.... \n");

}

else{

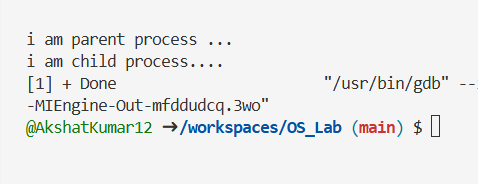
printf("i am parent process ...\n");

}

return 0;

}

**OUTPUT:**



**Program2.)**Write a program to print process Id's of parent and child process i.e. parent should print its own and its child process id while child process should print its own and its parent process id. (use getpid(), getppid())

**SOURCE CODE:**

 #include<stdio.h>

 #include<unistd.h>

 #include<sys/types.h>

 int main(){

    pid\_t id;

    id=fork();

    if(id<0){

        printf("sorry fork failed. \n");

        return 1;

    }

    else if(id==0){

        printf("child process is running .....\n");

        printf("child p\_id  : %d \n",getpid());

        printf("parent p\_id(child) : %d \n ",getppid());

    }

    else{

        printf("parent process is running ....\n");

        printf("parent p\_id : %d\n",getpid());

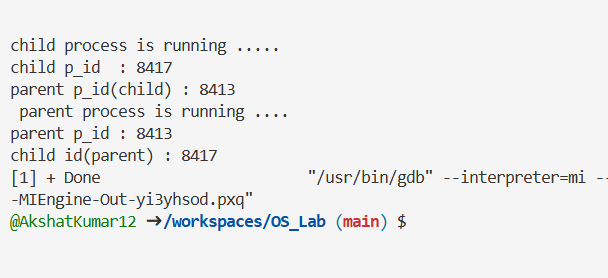
        printf("child id(parent) : %d \n",id);

    }

return 0;

}

**OUTPUT:**



**Program3.)**Write a program to create child process which will list all the files present in your system. Make sure that parent process waits until child has not completed its execution. (use wait(), exit()) What will happen if parent process dies before child process? Illustrate it by creating one more child of parent process.

**SOURCE CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

    pid\_t pid1, pid2;

    pid1 = fork();

    if (pid1 < 0) {

        printf("Fork failed!\n");

        return 1;

    } else if (pid1 == 0) {

        printf("Child Process 1 (PID: %d): Listing files...\n", getpid());

        execlp("ls", "ls", "-l", (char \*)NULL);

        exit(0);

    } else {

        wait(NULL);

        printf("Parent Process (PID: %d): First child completed.\n", getpid());

        pid2 = fork();

        if (pid2 < 0) {

            printf("Fork failed!\n");

            return 1;

        } else if (pid2 == 0) {

            printf("Child Process 2 (PID: %d): I am the second child.\n", getpid());

            sleep(5);

            printf("Child Process 2 (PID: %d): Work done.\n", getpid());

            exit(0);

        } else {

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

            exit(0);

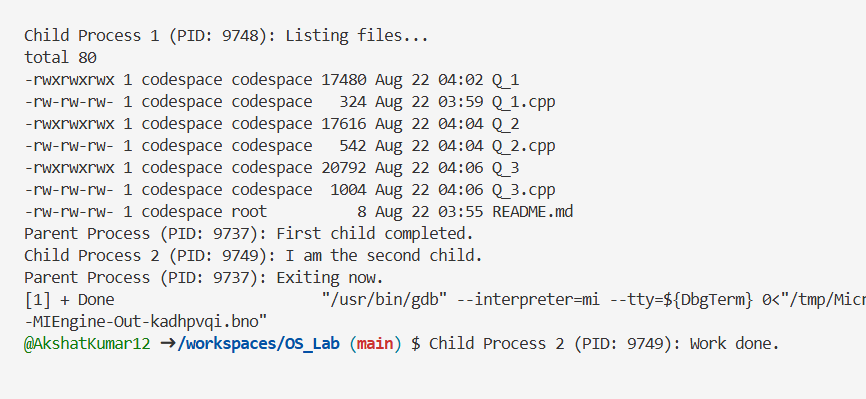
        }

    }

return 0;

}

**OUTPUT:**



**WEEK2**

P**rogram1.).** Write a program to open a directory and list it contents (use opendir(), readdir(), closedir() ).

**SOURCE CODE:**

#include<stdio.h>

#include<unistd.h>

#include<dirent.h>

#include<stdlib.h>

int main(){

DIR \*dir;

struct dirent \* entry;

dir=opendir(".");

if(dir==NULL){

printf("unable to open directory .....\n");

return 1;

}

printf("contents of the current directory ....\n");

while((entry=readdir(dir))!=NULL)

{

printf("%s\n",entry->d\_name);

}

closedir(dir);

return 0;

}

**OUTPUT:**



**Program2.).** Write a program to show working of execlp() system call by executing ls command.

**SOURCE CODE:**

#include <stdio.h>

#include <unistd.h>

int main()

{

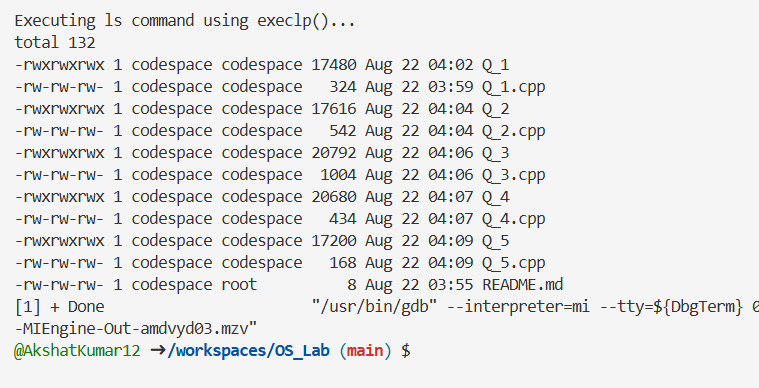
printf("Executing ls command using execlp()...\n");

execlp("ls", "ls", "-l", (char \*)NULL);

return 0;

}

**OUTPUT:**



**Program3.).** Write a program to read a file and store your details in that file. Your program should also create one more file and store your friends details in that file. Once both files are created, print lines which are matching in both files.

**SOURCE CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_LINE\_LENGTH 256

int main() {

FILE \*myFile, \*friendFile;

char myLine[MAX\_LINE\_LENGTH];

char friendLine[MAX\_LINE\_LENGTH];

myFile = fopen("mydetails.txt", "w");

if (myFile == NULL) {

perror("Failed to create mydetails.txt");

return 1;

}

fprintf(myFile, "Name: Akshat Kumar\n");

fprintf(myFile, "University: Graphic Era Hill University\n");

fprintf(myFile, "Location: Dehradun\n");

fprintf(myFile, "Hobby: Coding\n");

fclose(myFile);

friendFile = fopen("friendsdetails.txt", "w");

if (friendFile == NULL) {

perror("Failed to create friendsdetails.txt");

return 1;

}

fprintf(friendFile, "Name: Himanshu\n");

fprintf(friendFile, "University: Graphic Era Hill University\n");

fprintf(friendFile, "Location: Bihar\n");

fprintf(friendFile, "Hobby: Coding\n");

fclose(friendFile);

myFile = fopen("mydetails.txt", "r");

friendFile = fopen("friendsdetails.txt", "r");

if (myFile == NULL || friendFile == NULL) {

perror("Failed to open one of the files for reading");

return 1;

}

printf("\nMatching lines in both files:\n");

while (fgets(myLine, sizeof(myLine), myFile) != NULL) {

fseek(friendFile, 0, SEEK\_SET);

while (fgets(friendLine, sizeof(friendLine), friendFile) != NULL) {

myLine[strcspn(myLine, "\n")] = '\0';

friendLine[strcspn(friendLine, "\n")] = '\0';

if (strcmp(myLine, friendLine) == 0) {

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

}

}

}

fclose(myFile);

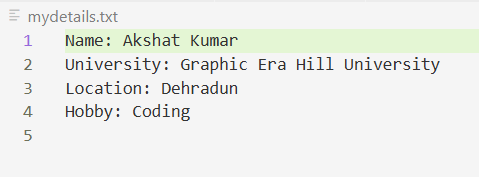
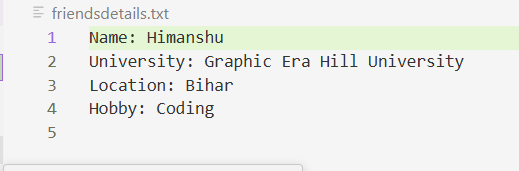
fclose(friendFile);

return 0;

}

**OUTPUT:**





**WEEK3**

**Program1.) FCFS** – First Come First Served : process which arrives first will get the CPU first.

**SOURCE CODE:**

#include <stdio.h>

#include <stdlib.h>

struct Process {

int pid;

int at;

int bt;

int ct;

float tat;

float wt;

int rt;

int st;

};

int compare(const void \*p1, const void \*p2) {

int a = ((struct Process \*)p1)->at;

int b = ((struct Process \*)p2)->at;

if (a < b)

return -1;

else

return 1;

}

int main() {

int n;

float swt = 0, stat = 0;

float cu = 0, throughput = 0;

float awt = 0, atat = 0;

int sbt = 0;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process p[n];

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

printf("For Process %d\n", i + 1);

p[i].pid = i + 1;

printf("Enter the value of AT and BT: ");

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

}

qsort((void \*)p, n, sizeof(struct Process), compare);

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

if (i == 0) {

p[i].ct = p[i].at + p[i].bt;

} else if (p[i - 1].ct <= p[i].at) {

p[i].ct = p[i].at + p[i].bt;

} else {

p[i].ct = p[i - 1].ct + p[i].bt;

}

p[i].tat = p[i].ct - p[i].at;

p[i].wt = p[i].tat - p[i].bt;

p[i].rt = p[i].wt;

sbt += p[i].bt;

swt += p[i].wt;

stat += p[i].tat;

}

awt = swt / n;

atat = stat / n;

int max = 0;

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

p[i].st = p[i].rt + p[i].at;

if (p[i].ct > max) {

max = p[i].ct;

}

}

cu = (sbt / (float)max) \* 100;

throughput = n / (float)max;

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

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

printf("P%d\t%d\t%d\t%d\t%d\t%.2f\t\t%.2f\t\t%d\n",

p[i].pid, p[i].at, p[i].bt, p[i].st, p[i].ct, p[i].tat, p[i].wt, p[i].rt);

}

printf("\nSum of Turn Around Time: %.2f\nAverage of Turn Around Time: %.2f\n", stat, atat);

printf("Sum of Waiting Time: %.2f\nAverage of Waiting Time: %.2f\n", swt, awt);

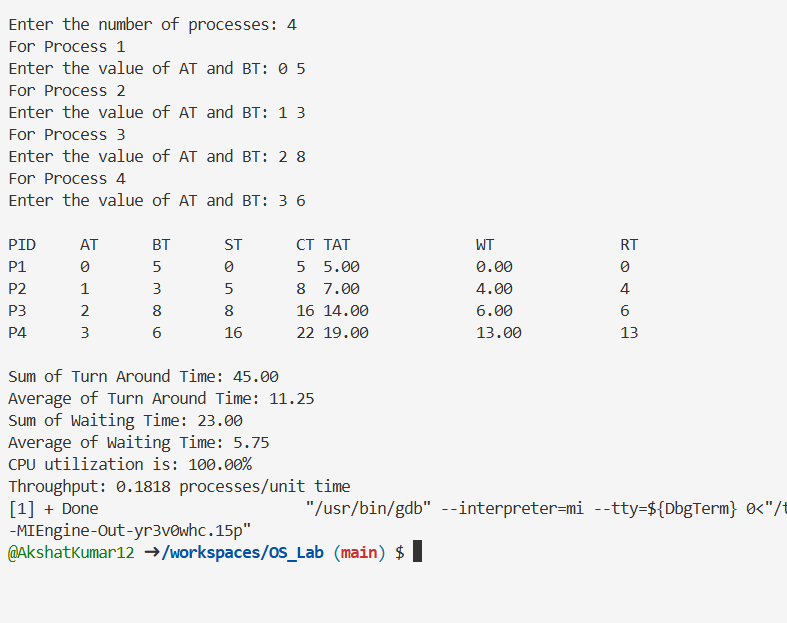
printf("CPU utilization is: %.2f%%\n", cu);

printf("Throughput: %.4f processes/unit time\n", throughput);

return 0;

}

**OUTPUT:**



**Program2.) SJF NP** – Shortest Job First Non-Preemptive : process which needs CPU for least amount will get the CPU first. Here non-preemptive means currently running process leaves CPU voluntarily a er completing its execution.

**SOURCE CODE:**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

struct process\_struct

{

int pid;

int at;

int bt;

int ct, wt, tat, rt, start\_time;

} ps[100];

int findmax(int a, int b) { return a > b ? a : b; }

int findmin(int a, int b) { return a < b ? a : b; }

int main()

{

int n;

bool is\_completed[100] = {false}, is\_first\_process = true;

int current\_time = 0, completed = 0;

int sum\_tat = 0, sum\_wt = 0, sum\_rt = 0, total\_idle\_time = 0, prev = 0, length\_cycle;

float cpu\_utilization;

int max\_completion\_time, min\_arrival\_time;

printf("Enter total number of processes: ");

scanf("%d", &n);

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

{

ps[i].pid = i + 1;

printf("\nEnter AT and BT for Process %d: ", i + 1);

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

}

while (completed != n)

{

int min\_index = -1;

int minimum = INT\_MAX;

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

{

if (ps[i].at <= current\_time && is\_completed[i] == false)

{

if (ps[i].bt < minimum)

{

minimum = ps[i].bt;

min\_index = i;

}

if (ps[i].bt == minimum)

{

if (ps[i].at < ps[min\_index].at)

{

minimum = ps[i].bt;

min\_index = i;

}

}

}

}

if (min\_index == -1)

{

current\_time++;

}

else

{

ps[min\_index].start\_time = current\_time;

ps[min\_index].ct = ps[min\_index].start\_time + ps[min\_index].bt;

ps[min\_index].tat = ps[min\_index].ct - ps[min\_index].at;

ps[min\_index].wt = ps[min\_index].tat - ps[min\_index].bt;

ps[min\_index].rt = ps[min\_index].wt; // For non-preemptive SJF

sum\_tat += ps[min\_index].tat;

sum\_wt += ps[min\_index].wt;

sum\_rt += ps[min\_index].rt;

total\_idle\_time += (is\_first\_process == true) ? 0 : (ps[min\_index].start\_time - prev);

completed++;

is\_completed[min\_index] = true;

current\_time = ps[min\_index].ct;

prev = current\_time;

is\_first\_process = false;

}

}

max\_completion\_time = INT\_MIN;

min\_arrival\_time = INT\_MAX;

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

{

max\_completion\_time = findmax(max\_completion\_time, ps[i].ct);

min\_arrival\_time = findmin(min\_arrival\_time, ps[i].at);

}

length\_cycle = max\_completion\_time - min\_arrival\_time;

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

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

{

printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

ps[i].pid, ps[i].at, ps[i].bt,

ps[i].ct, ps[i].tat, ps[i].wt, ps[i].rt);

}

cpu\_utilization = (float)(length\_cycle - total\_idle\_time) / length\_cycle;

printf("\nAverage Turn Around Time = %.2f", (float)sum\_tat / n);

printf("\nAverage Waiting Time = %.2f", (float)sum\_wt / n);

printf("\nAverage Response Time = %.2f", (float)sum\_rt / n);

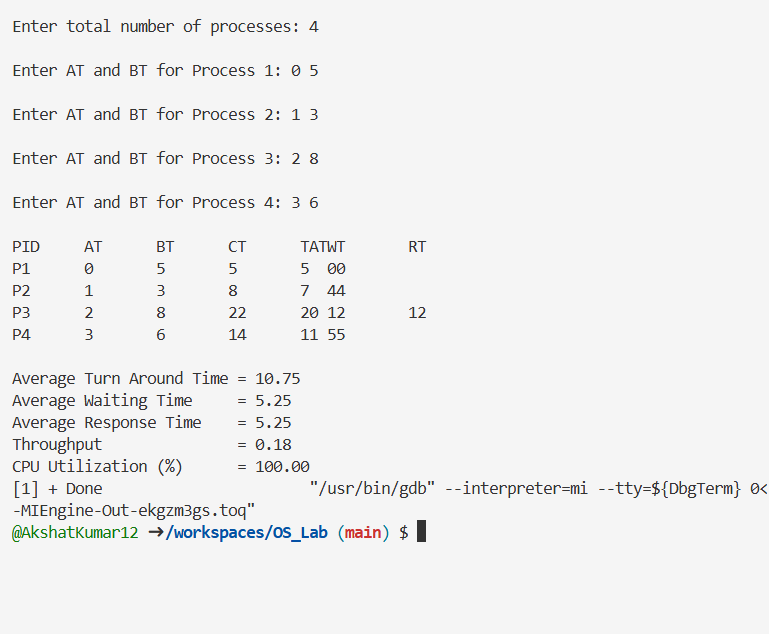
printf("\nThroughput = %.2f", n / (float)length\_cycle);

printf("\nCPU Utilization (%%) = %.2f\n", cpu\_utilization \* 100);

return 0;

}

**OUTPUT:**



**Program3.) SJF P** – Shortest Job First Preemptive – Here preemptive means opera ng system decides when to move currently running process.

**SOURCE CODE:**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

struct process\_struct

{

int pid;

int at;

int bt;

int ct, wt, tat, rt, start\_time;

} ps[100];

int findmax(int a, int b) { return a > b ? a : b; }

int findmin(int a, int b) { return a < b ? a : b; }

int main()

{

int n;

int bt\_remaining[100];

bool is\_completed[100] = {false}, is\_first\_process = true;

int current\_time = 0, completed = 0, prev = 0;

float sum\_tat = 0, sum\_wt = 0, sum\_rt = 0, total\_idle\_time = 0, length\_cycle;

float cpu\_utilization;

int max\_completion\_time, min\_arrival\_time;

printf("Enter total number of processes: ");

scanf("%d", &n);

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

{

ps[i].pid = i + 1;

printf("\nEnter AT and BT for Process %d: ", i + 1);

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

bt\_remaining[i] = ps[i].bt;

}

while (completed != n)

{

int min\_index = -1;

int minimum = INT\_MAX;

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

{

if (ps[i].at <= current\_time && is\_completed[i] == false)

{

if (bt\_remaining[i] < minimum)

{

minimum = bt\_remaining[i];

min\_index = i;

}

if (bt\_remaining[i] == minimum)

{

if (ps[i].at < ps[min\_index].at)

{

minimum = bt\_remaining[i];

min\_index = i;

}

}

}

}

if (min\_index == -1)

{

current\_time++;

}

else

{

if (bt\_remaining[min\_index] == ps[min\_index].bt)

{

ps[min\_index].start\_time = current\_time;

total\_idle\_time += (is\_first\_process == true) ? 0 : (ps[min\_index].start\_time - prev);

is\_first\_process = false;

}

bt\_remaining[min\_index] -= 1;

current\_time++;

prev = current\_time;

if (bt\_remaining[min\_index] == 0)

{

ps[min\_index].ct = current\_time;

ps[min\_index].tat = ps[min\_index].ct - ps[min\_index].at;

ps[min\_index].wt = ps[min\_index].tat - ps[min\_index].bt;

ps[min\_index].rt = ps[min\_index].start\_time - ps[min\_index].at;

sum\_tat += ps[min\_index].tat;

sum\_wt += ps[min\_index].wt;

sum\_rt += ps[min\_index].rt;

completed++;

is\_completed[min\_index] = true;

}

}

}

max\_completion\_time = INT\_MIN;

min\_arrival\_time = INT\_MAX;

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

{

max\_completion\_time = findmax(max\_completion\_time, ps[i].ct);

min\_arrival\_time = findmin(min\_arrival\_time, ps[i].at);

}

length\_cycle = max\_completion\_time - min\_arrival\_time;

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

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

{

printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

ps[i].pid, ps[i].at, ps[i].bt,

ps[i].ct, ps[i].tat, ps[i].wt, ps[i].rt);

}

cpu\_utilization = (float)(length\_cycle - total\_idle\_time) / length\_cycle;

printf("\nAverage Turn Around Time = %.2f", (float)sum\_tat / n);

printf("\nAverage Waiting Time = %.2f", (float)sum\_wt / n);

printf("\nAverage Response Time = %.2f", (float)sum\_rt / n);

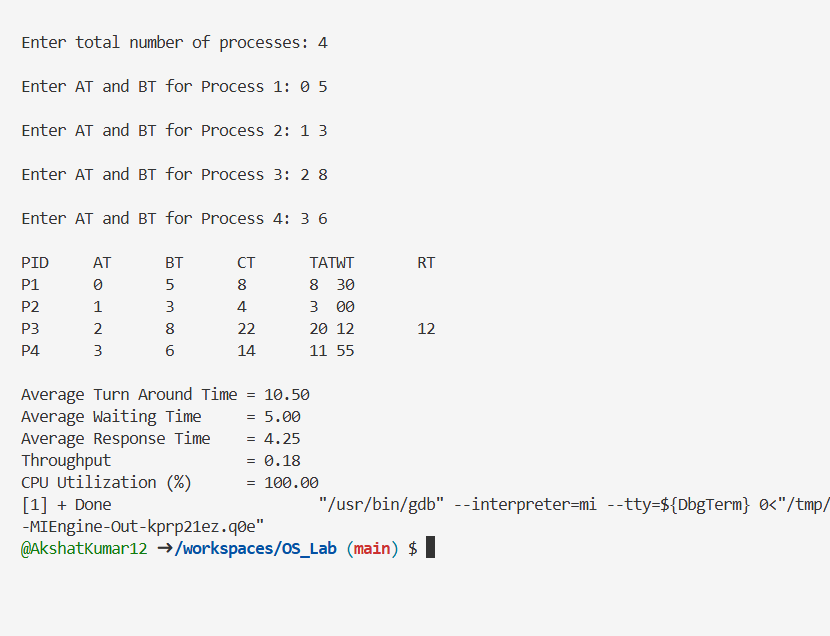
printf("\nThroughput = %.2f", n / (float)length\_cycle);

printf("\nCPU Utilization (%%) = %.2f\n", cpu\_utilization \* 100);

return 0;

}

**OUTPUT:**



**WEEK4**

**Program1.) Priority** **Scheduling**– process which has highest priority will get CPU first.

**SOURCE CODE:**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

struct **process\_struct**

{

    int pid;

    int at;

    int bt;

    int priority;

    int ct, wt, tat, rt, start\_time;

} ps[100];

int **findmax**(int a, int b) { return a > b ? a : b; }

int **findmin**(int a, int b) { return a < b ? a : b; }

int **main**()

{

    int n;

**bool** is\_completed[100] = {**false**}, is\_first\_process = **true**;

    int current\_time = 0, completed = 0, total\_idle\_time = 0, prev = 0, length\_cycle;

    float cpu\_utilization;

    int max\_completion\_time, min\_arrival\_time;

    float sum\_tat = 0, sum\_wt = 0, sum\_rt = 0;

**printf**("Enter total number of processes: ");

**scanf**("%d", &n);

    for (int i = 0; i < n; i++)

    {

        ps[i].pid = i + 1;

**printf**("\nEnter AT, BT and Priority for Process %d: ", i + 1);

**scanf**("%d %d %d", &ps[i].at, &ps[i].bt, &ps[i].priority);

    }

    while (completed != n)

    {

        int max\_index = -1;

        int maximum = **INT\_MIN**;

        for (int i = 0; i < n; i++)

        {

            if (ps[i].at <= current\_time && is\_completed[i] == **false**)

            {

                if (ps[i].priority > maximum)

                {

                    maximum = ps[i].priority;

                    max\_index = i;

                }

                else if (ps[i].priority == maximum)

                {

                    if (ps[i].at < ps[max\_index].at)

                    {

                        maximum = ps[i].priority;

                        max\_index = i;

                    }

                }

            }

        }

        if (max\_index == -1)

        {

            current\_time++;

        }

        else

        {

            ps[max\_index].start\_time = current\_time;

            ps[max\_index].ct = ps[max\_index].start\_time + ps[max\_index].bt;

            ps[max\_index].tat = ps[max\_index].ct - ps[max\_index].at;

            ps[max\_index].wt = ps[max\_index].tat - ps[max\_index].bt;

            ps[max\_index].rt = ps[max\_index].start\_time - ps[max\_index].at;

            total\_idle\_time += (is\_first\_process == **true**) ? 0 : (ps[max\_index].start\_time - prev);

            sum\_tat += ps[max\_index].tat;

            sum\_wt += ps[max\_index].wt;

            sum\_rt += ps[max\_index].rt;

            completed++;

            is\_completed[max\_index] = **true**;

            current\_time = ps[max\_index].ct;

            prev = current\_time;

            is\_first\_process = **false**;

        }

    }

    max\_completion\_time = **INT\_MIN**;

    min\_arrival\_time = **INT\_MAX**;

    for (int i = 0; i < n; i++)

    {

        max\_completion\_time = **findmax**(max\_completion\_time, ps[i].ct);

        min\_arrival\_time = **findmin**(min\_arrival\_time, ps[i].at);

    }

    length\_cycle = max\_completion\_time - min\_arrival\_time;

    cpu\_utilization = (float)(length\_cycle - total\_idle\_time) / length\_cycle;

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

    for (int i = 0; i < n; i++)

    {

**printf**("P%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

               ps[i].pid, ps[i].at, ps[i].bt, ps[i].priority,

               ps[i].ct, ps[i].tat, ps[i].wt, ps[i].rt);

    }

**printf**("\nAverage Turn Around Time = %.2f", sum\_tat / n);

**printf**("\nAverage Waiting Time     = %.2f", sum\_wt / n);

**printf**("\nAverage Response Time    = %.2f", sum\_rt / n);

**printf**("\nThroughput               = %.2f", n / (float)length\_cycle);

**printf**("\nCPU Utilization (%%)      = %.2f\n", cpu\_utilization \* 100);

    return 0;

}

**OUTPUT:**

