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:

```
i am parent process ...
i am child process....
[1] + Done                       "/usr/bin/gdb" --
-MIEngine-Out-mfddudcq.3wo"
@AkshatKumar12 →/workspaces/OS_Lab (main) $
```

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:

```
child process is running .....
child p_id  : 8417
parent p_id(child) : 8413
 parent process is running ....
parent p_id : 8413
child id(parent) : 8417
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --
-MIEngine-Out-yi3yhsod.pxq"
@AkshatKumar12 →/workspaces/OS_Lab (main) $
```

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:

```
Child Process 1 (PID: 9748): Listing files...
total 80
-rwxrwxrwx 1 codespace codespace 17480 Aug 22 04:02 Q_1
-rw-rw-rw- 1 codespace codespace   324 Aug 22 03:59 Q_1.cpp
-rwxrwxrwx 1 codespace codespace 17616 Aug 22 04:04 Q_2
-rw-rw-rw- 1 codespace codespace   542 Aug 22 04:04 Q_2.cpp
-rwxrwxrwx 1 codespace codespace 20792 Aug 22 04:06 Q_3
-rw-rw-rw- 1 codespace codespace  1004 Aug 22 04:06 Q_3.cpp
-rw-rw-rw- 1 codespace root          8 Aug 22 03:55 README.md
Parent Process (PID: 9737): First child completed.
Child Process 2 (PID: 9749): I am the second child.
Parent Process (PID: 9737): Exiting now.
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Micr
-MIEngine-Out-kadhpvqi.bno"
@AkshatKumar12 ➜/workspaces/OS_Lab (main) $ Child Process 2 (PID: 9749): Work done.
```

WEEK2

Program1.). 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:

```
contents of the current directory ....
README.md
Q_3
Q_2.cpp
.git
Q_4
..
.
Q_3.cpp
Q_2
Q_1
Q_4.cpp
Q_1.cpp
.vscode
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft
-MIEngine-Out-bdjyfsbw.kek"
@AkshatKumar12 →/workspaces/OS_Lab (main) $
```

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:

```
Executing ls command using execlp()...
total 132
-rwxrwxrwx 1 codespace codespace 17480 Aug 22 04:02 Q_1
-rw-rw-rw- 1 codespace codespace   324 Aug 22 03:59 Q_1.cpp
-rwxrwxrwx 1 codespace codespace 17616 Aug 22 04:04 Q_2
-rw-rw-rw- 1 codespace codespace   542 Aug 22 04:04 Q_2.cpp
-rwxrwxrwx 1 codespace codespace 20792 Aug 22 04:06 Q_3
-rw-rw-rw- 1 codespace codespace  1004 Aug 22 04:06 Q_3.cpp
-rwxrwxrwx 1 codespace codespace 20680 Aug 22 04:07 Q_4
-rw-rw-rw- 1 codespace codespace   434 Aug 22 04:07 Q_4.cpp
-rwxrwxrwx 1 codespace codespace 17200 Aug 22 04:09 Q_5
-rw-rw-rw- 1 codespace codespace   168 Aug 22 04:09 Q_5.cpp
-rw-rw-rw- 1 codespace root          8 Aug 22 03:55 README.md
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0
-MIEngine-Out-amdvyd03.mzv"
@AkshatKumar12 ➜ /workspaces/OS_Lab (main) $
```

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:

```
Matching lines in both files:
University: Graphic Era Hill University
Hobby: Coding
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=
-MIEngine-Out-0kyemeak.mt3"
@AkshatKumar12 →/workspaces/OS_Lab (main) $
```

friendsdetails.txt

```
1   Name: Himanshu
2   University: Graphic Era Hill University
3   Location: Bihar
4   Hobby: Coding
5
```

mydetails.txt

```
1   Name: Akshat Kumar
2   University: Graphic Era Hill University
3   Location: Dehradun
4   Hobby: Coding
5
```

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:

```
Enter the number of processes: 4
For Process 1
Enter the value of AT and BT: 0 5
For Process 2
Enter the value of AT and BT: 1 3
For Process 3
Enter the value of AT and BT: 2 8
For Process 4
Enter the value of AT and BT: 3 6

PID     AT      BT      ST      CT TAT             WT             RT
P1      0       5       0       5  5.00            0.00           0
P2      1       3       5       8  7.00            4.00           4
P3      2       8       8       16 14.00           6.00           6
P4      3       6       16      22 19.00           13.00          13

Sum of Turn Around Time: 45.00
Average of Turn Around Time: 11.25
Sum of Waiting Time: 23.00
Average of Waiting Time: 5.75
CPU utilization is: 100.00%
Throughput: 0.1818 processes/unit time
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/t
-MIEngine-Out-yr3v0whc.15p"
@AkshatKumar12 →/workspaces/OS_Lab (main) $
```

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:

```
Enter total number of processes: 4

Enter AT and BT for Process 1: 0 5

Enter AT and BT for Process 2: 1 3

Enter AT and BT for Process 3: 2 8

Enter AT and BT for Process 4: 3 6

PID     AT      BT      CT      TATWT       RT
P1      0       5       5       5  00
P2      1       3       8       7  44
P3      2       8       22      20 12       12
P4      3       6       14      11 55

Average Turn Around Time = 10.75
Average Waiting Time     = 5.25
Average Response Time    = 5.25
Throughput               = 0.18
CPU Utilization (%)      = 100.00
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<
-MIEngine-Out-ekgzm3gs.toq"
@AkshatKumar12 →/workspaces/OS_Lab (main) $
```

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:

```
Enter total number of processes: 4

Enter AT and BT for Process 1: 0 5

Enter AT and BT for Process 2: 1 3

Enter AT and BT for Process 3: 2 8

Enter AT and BT for Process 4: 3 6

PID     AT      BT      CT      TATWT       RT
P1      0       5       8       8  30
P2      1       3       4       3  00
P3      2       8       22      20 12       12
P4      3       6       14      11 55

Average Turn Around Time = 10.50
Average Waiting Time     = 5.00
Average Response Time    = 4.25
Throughput               = 0.18
CPU Utilization (%)      = 100.00
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/
-MIEngine-Out-kprp21ez.q0e"
@AkshatKumar12 →/workspaces/OS_Lab (main) $
```

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:

```
Enter total number of processes: 4

Enter AT, BT and Priority for Process 1: 0 5 3

Enter AT, BT and Priority for Process 2: 1 3 2

Enter AT, BT and Priority for Process 3: 2 8 4

Enter AT, BT and Priority for Process 4: 3 6 1

PID     AT      BT      PR      CT TAT      WT      RT
P1      0       5       3       5  50       0
P2      1       3       2       16 15       12      12
P3      2       8       4       13 11       3       3
P4      3       6       1       22 19       13      13

Average Turn Around Time = 12.50
Average Waiting Time     = 7.00
Average Response Time    = 7.00
Throughput               = 0.18
CPU Utilization (%)      = 100.00
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${
-MIEngine-Out-zvh51e0n.zri"
```