Name: Dhrubahari Ranabhat
Registration number: 11813479
Email address: dhurva.ranabhat123@gmail.com
GitHub link:https://github.com/DhrubahariRanabhat/OSCA/

Question:
Implement the preemptive Priority CPU scheduling algorithm. Compare the average waiting time per process considering context switch time over head with average waiting time per process not considering context switch time over head.

Code:
```
#include <stdio.h>
#include <stdlib.h>
struct Processes
{
int burstTime, priority, arrivalTime, flag, remainingTime, completionTime, 
turnaroundTime, waitingTime, cpu, rt, id, cot;
};

void main()
{
struct Processes process[10],tmp;
int a, totalNumber, totalTime = 0, highest;
printf("Enter the number of process for the scheduler:\t");
a:
scanf("%d", &totalNumber);
if (totalNumber > 8 && totalNumber < 1)
goto a;
for (a = 0; a < totalNumber; a++)
{
printf("\n\nEnter the details for the process P%d", a + 1);
printf("Arrival time\t");
scanf("%d", &process[a].arrivalTime);
printf("burst time\t\t");
scanf("%d", &process[a].burstTime);
process[a].remainingTime = process[a].burstTime;
printf("Priority\t\t");
scanf("%d", &process[a].priority);
totalTime += process[a].burstTime;
totalTime += process[a].arrivalTime;
process[a].flag = -1;
process[a].id = a + 1;
process[a].cot = 0;
}

int b;
```

```
for (a = 0; a < totalNumber; a++)
{
for (b = a + 1; b < totalNumber; b++)
{
if (process[a].arrivalTime > process[b].arrivalTime)
{
tmp = process[a];
process[a] = process[b];
process[b] = tmp;
}
}
}

process[9].priority = 999;
process[9].remainingTime = 9999;
process[9].flag = -1;
int previousId, first = 1;
int last = 0;
system("CLS");
printf("GANTT CHART\n\n");
int currentTime = process[0].arrivalTime;
printf("\n|");
while (currentTime < 100)
{
highest = 9;

for (a = 0; a < totalNumber; a++)
{
if (currentTime - process[a].cot >= 2)
{
if (process[a].flag == -2)
{
process[a].priority -= 2;
}
}
}

for (a = 0; a < totalNumber; a++)
{
if (process[a].remainingTime > 0)
{
if (process[a].priority < process[highest].priority)
{
if (process[a].arrivalTime <= currentTime)
{
highest = a;
}
}
}
}

printf("Selected P%d\n", process[highest].id);

if (process[highest].flag == -2)
{
process[highest].flag = -1;
}

if (process[highest].id != process[previousId].id)
{
process[previousId].flag = -2;
process[previousId].cot = currentTime;
if (process[highest].id != 0)
printf("%d - |", currentTime);
currentTime += 2;
}

if (process[highest].flag == -1)
{
process[highest].flag = 0;
process[highest].cpu = currentTime;
}

if (process[highest].id != 0)
printf("%d P%d |", currentTime, process[highest].id);

previousId = highest;
process[highest].remainingTime--;
currentTime++;

printf("Current time %d\n", currentTime);

if (process[highest].remainingTime == 0 && process[highest].id != 0)
{
process[highest].flag = 1;
process[highest].completionTime = currentTime;
process[highest].turnaroundTime = process[highest].completionTime -
process[highest].arrivalTime;
process[highest].waitingTime = process[highest].turnaroundTime -
process[highest].burstTime;
process[highest].rt = process[highest].cpu - process[highest].arrivalTime;
last = highest;
```

```
continue;
}
}
printf("%d", process[last].completionTime);
printf("\n\n\nProcess\t\tArrival Time\tBurst Time\tCompletion Time\tTurn Around
Time \tWait Time \t\n");
for (a = 0; a < totalNumber; a++)
{
printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", a + 1, process[a].arrivalTime,
process[a].burstTime, process[a].completionTime, process[a].turnaroundTime,
process[a].waitingTime, process[a].rt);
}
}
```

1. Description:

Preemptive priority scheduling algorithm is such type of algorithm where the job which is being executed can be stopped when the job with highest priority comes into the scenario. In this algorithm the priority of the job is compared with all of the available processes and the process that is currently running and whenever the process with highest priority comes then the priority will be given to the highest priority job.

1.In this problem we have to use Preemptive Priority CPU scheduling algorithm.

2. First of all, CPU schedules the job which has low arrival time and after that schedules the processes by interrupting the processor after the highest priority job comes and does consider the completion of the process in this execution which is first execution.

3.Than again at the second preemption the scheduler than checks for the number of process waiting for the processor and allots the processor to the process having high priority then the currently executing job and but similar to first iteration considers the completion of the processes in this iteration.

4. Than at last the CPU checks the number of processes waiting in the queue for the processor after the execution and gives the processor to the process which has highest priority to complete than the other processes to go in the terminated state.

2. Algorithm:

```
Step -1: Declare array arrivalTime[ ],burstTime[ ],priority[], remainingTime[ ],

waitingTime[ ], turnAroundTime[ ] and int variable n.

Step -2: Take input number of process in size.

Step -3: Repeat for int i= 0,1,2.... size-1

    Take arrival time,burst time and the priority of the process as input.

Step -4:Compare the arrival time of all the processes.

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

                              if(p[i].arrival_time > p[j].arrival_time){
```

```
                                        temp = p[i];
                                        p[i] = p[j];
                                        p[j] = temp;
```

Step -5: Execute the process and see whether there are other processes arriving on that time or not. If the process/job is arriving on that time than compare the priority of those job and gives the CPU to the job having highest priority.

```
                    if (process[a].priority < process[highest].priority)
          {
            if (process[a].arrivalTime <= currentTime)
            {
              highest = a;
```

Step -6: Again execute the next process and repeat the step no 5.

Step -7: if(remainingBTime[Process_no]==0 && indicator==1)

```
        remainingProcess--
        remainingTime[Process_no]=currentTime
        turnAroundTime[Process_no]=remainingTime[Process_no]-
        arrivalTime[Process_no]
        waitingTime[Process_no]=turnAroundTime[Process_no]-
        burstTime[Process_no]
        indicator = 0
```

Step -8: End the repetition of step no 5 and set:- Process_no=0

Repeat Step -9 to Step - 10 while remaining process != 0

Step -9: Pick a job whose burst time is lesser among all of them.

Step -10: Execute that job and set

```
remainingTime[Process_no]=currentTime
turnAroundTime[Process_no]=remainingTime[Process_no]-arrivalTime[Process_no]
waitingTime[Process_no]=trunaroundTime[Process_no]-burstTime
```

Step -11: End of step step 10 loop.

Step -12: Repeat for int I = 0,1,2.... Size-1

A) Calculate average waiting time and average burst time.

B) Print all the information.

3. Time Complexity:

The time complexity of the code to this problem is $O(n^2)$.

4. Code Snippet:

```
#include <stdio.h>

#include <stdlib.h>
struct Processes
{
int burstTime, priority, arrivalTime, flag, remainingTime, completionTime, 
turnaroundTime, waitingTime, cpu, rt, id, cot;
};

void main()
{
struct Processes process[10],tmp;
int a, totalNumber, totalTime = 0, highest;
printf("Enter the number of process for the scheduler:\t");
a:
scanf("%d", &totalNumber);
if (totalNumber > 8 && totalNumber < 1)
goto a;
for (a = 0; a < totalNumber; a++)
{
printf("\n\nEnter the details for the process P%d", a + 1);
printf("Arrival time\t");
scanf("%d", &process[a].arrivalTime);
printf("burst time\t\t");
scanf("%d", &process[a].burstTime);
process[a].remainingTime = process[a].burstTime;
printf("Priority\t\t");
scanf("%d", &process[a].priority);
totalTime += process[a].burstTime;
totalTime += process[a].arrivalTime;
process[a].flag = -1;
process[a].id = a + 1;
process[a].cot = 0;
}

int b;
for (a = 0; a < totalNumber; a++)
{
for (b = a + 1; b < totalNumber; b++)
{
if (process[a].arrivalTime > process[b].arrivalTime)
{
tmp = process[a];
process[a] = process[b];
process[b] = tmp;
}
}
}

process[9].priority = 999;
process[9].remainingTime = 9999;
process[9].flag = -1;
int previousId, first = 1;
int last = 0;
system("CLS");
printf("GANTT CHART\n\n");
int currentTime = process[0].arrivalTime;
printf("\n|");
while (currentTime < 100)
{
highest = 9;

for (a = 0; a < totalNumber; a++)
{
if (currentTime - process[a].cot >= 2)
{
if (process[a].flag == -2)
{
process[a].priority -= 2;
}
}
}

for (a = 0; a < totalNumber; a++)
{
if (process[a].remainingTime > 0)
{
if (process[a].priority < process[highest].priority)
{
if (process[a].arrivalTime <= currentTime)
{
highest = a;
}
}
}
}
```

```
printf("Selected P%d\n", process[highest].id);

if (process[highest].flag == -2)
{
process[highest].flag = -1;
}

if (process[highest].id != process[previousId].id)
{
process[previousId].flag = -2;
process[previousId].cot = currentTime;
if (process[highest].id != 0)
printf("%d - |", currentTime);
currentTime += 2;
}

if (process[highest].flag == -1)
{
process[highest].flag = 0;
process[highest].cpu = currentTime;
}

if (process[highest].id != 0)
printf("%d P%d |", currentTime, process[highest].id);

previousId = highest;
process[highest].remainingTime--;
currentTime++;

printf("Current time %d\n", currentTime);

if (process[highest].remainingTime == 0 && process[highest].id != 0)
{
process[highest].flag = 1;
process[highest].completionTime = currentTime;
process[highest].turnaroundTime = process[highest].completionTime -
process[highest].arrivalTime;
process[highest].waitingTime = process[highest].turnaroundTime -
process[highest].burstTime;
process[highest].rt = process[highest].cpu - process[highest].arrivalTime;
last = highest;
continue;
}
}
printf("%d", process[last].completionTime);
printf("\n\n\nProcess\t\tArrival Time\tBurst Time\tCompletion Time\tTurn Around
Time \tWait Time \t\n");
for (a = 0; a < totalNumber; a++)
{
printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", a + 1, process[a].arrivalTime,
process[a].burstTime, process[a].completionTime, process[a].turnaroundTime,
process[a].waitingTime, process[a].rt);
}
}
```

5. Boundary Conditions:

Total no. of processes must be less than equal to 10 because of while declaring an array only 10 size is allocated.

The processes Burst Time and Arrival Time must be reliable without any random values. If the very random input is passed than the program does not return the desired output.

6. Test Cases:

PID	Arrival Time	Priority	Burst Time	Waiting Time	Turnaround Time
P1	1	5	4	11 ms	15 ms
P2	2	2	5	14 ms	19 ms
P3	3	6	6	5 ms	11 ms
P4	0	4	1	0 ms	1 ms
P5	4	7	2	3 ms	5 ms
P6	5	8	3	0 ms	3 ms

GitHub Link: https://github.com/DhrubahariRanabhat/OSCA/