



SCHEDULING

OPERATING SYSTEM



COLLABORATORS NAME & ID :
MARYAM ALIKARAMI 9731045
MAHAN AHMADVAND 9731071

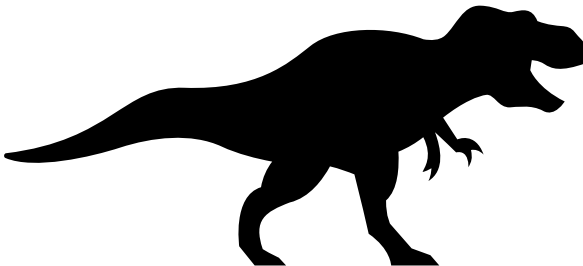




SCHEDULING



OPERATING SYSTEM



COLLABORATORS NAME & ID:
MARYAM ALIKARAMI 9731045
MAHAN AHMADVAND 9731071



OPERATING SYSTEM LABORATORY

SCHEDULING



INSTRUCTOR :

ARMAN GHEISARI

COLLABORATORS :

MARYAM ALIKARAMI

MAHAN AHMADVAND

QUESTION 4

برنامه ای به زبان c بنویسید که الگوریتم Round Robin را پیاده سازی کند.

(۱) تعداد فرآیندها را از کاربر دریافت کنید.

(۲) زمان سرویس دهی هر فرآیند را از کاربر دریافت کنید.

(۳) کوانتوم زمانی را از کاربر دریافت کنید.

(۴) ترتیب انجام فرآیندها را نشان دهید.

(۵) متوسط زمان انتظار هر فرآیند را حساب کنید.

ابتدا می خواهیم کمی در مورد این الگوریتم توضیح دهیم :

این الگوریتم یک الگوریتم preemptive و responsive است

و در واقع هر پردازش اجازه دارد از CPU به اندازه ی time

quantum استفاده کند و زمانی که به اندازه ی time

quantum از CPU استفاده کرد، نوبت به پردازش ی بعدی

میرسد و پردازش ی فعلی به آخر صف ready queue می رود و

به همین ترتیب.

معمولا time quantum را به اندازه ی 10 تا 100 میلی

ثانیه در نظر می گیرند.

توجه شود که نباید عدد time quantum را بزرگ انتخاب

کنیم زیرا ممکن است در واقع همان FIFO رخ دهد، یعنی عملا

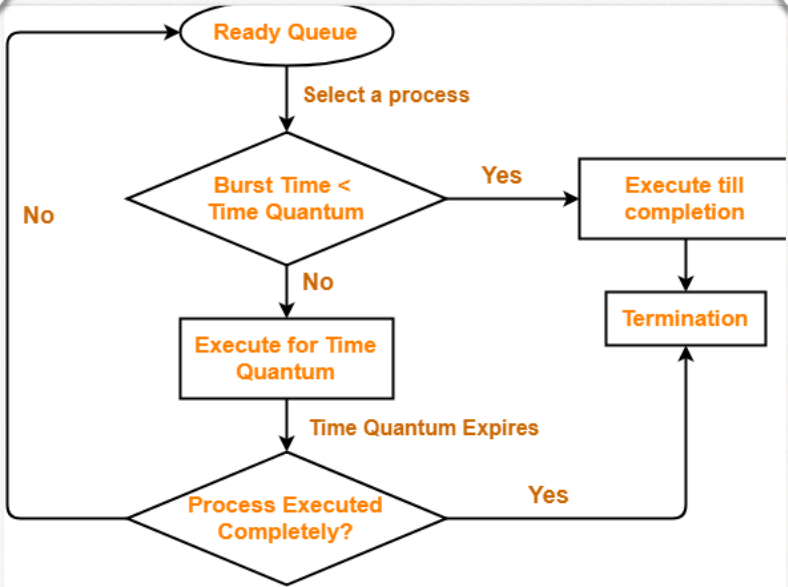
FCFS خواهیم داشت، همچنین نباید این عدد را کوچک اختیار

کنیم، زیرا باعث می شود که context switch زیاد رخ دهد و

این باعث سربار زیاد می شود(زیرا می دانیم که context

switch، سربار دارد).

Round robin algorithm flow chart



Round Robin Scheduling

نکته: توجه کنیم که time quantum را طوری انتخاب می کنیم تا 80 درصد cpu burst ها کوچکتر از time quantum باشند.

حال می خواهیم این الگوریتم را با توجه به موارد خواسته شده در سوال پیاده سازی کنیم :

```
1 #include<stdio.h>
2
3 int main()
4 {
5
6     int count = 0;
7     int numberOfProcesses = 0;
8     int time = 0;
9     int remainingProcesses = 0;
10    int flag = 0;
11    int timeQuantum = 0;
12    int waitingTime = 0;
13    int turnaroundTime = 0;
14    int arrivalTime[100];
15    int burstTime[100];
16    int needingTime[100];
17
18    printf("Enter Total number of Processes:");
19    scanf("%d",&numberOfProcesses);
20
21    remainingProcesses = numberOfProcesses;
22
23    for(count = 0; count < numberOfProcesses; count++)
24    {
25        int processesNumber = count + 1;
26        printf("Enter Arrival Time and Burst Time for Process Number %d :",processesNumber);
27        scanf("%d",&arrivalTime[count]);
28        scanf("%d",&burstTime[count]);
29        needingTime[count]=burstTime[count];
30    }
31
32    printf("Enter the Time Quantum(q:");
33    scanf("%d",&timeQuantum);
34    printf("\nProcess\t TurnaroundTime\t WaitingTime\n\n");
35
36    time = 0;
37    count = 0;
38    while(remainingProcesses > 0)
39    {
40        if(needingTime[count] <= timeQuantum && needingTime[count] > 0)
41        {
42            time+=needingTime[count];
43            needingTime[count] = 0;
44            flag = 1;
45        }
46        else if(needingTime[count] > 0)
47        {
48            needingTime[count] -= timeQuantum;
49            time+=timeQuantum;
50            int processNumber = count + 1;
51            printf("%d\t\t %d\t\t %d\n", processNumber, processTurnaroundTime,processWaitingTime);
52        }
53        if(needingTime[count] == 0 && flag == 1)
54        {
55            remainingProcesses--;
56            int processNumber = count + 1;
57            int processTurnaroundTime = time - arrivalTime[count];
58            int processWaitingTime = time - arrivalTime[count] - burstTime[count];
59            printf("%d\t\t %d\t\t %d\n", processNumber, processTurnaroundTime,processWaitingTime);
60
61            waitingTime += time-arrivalTime[count]-burstTime[count]; // Turnaround Time - burst Time
62            turnaroundTime += time-arrivalTime[count]; // departure Time(time variable in here) - arrival Time
63
64            flag=0;
65        }
66        if(count == numberOfProcesses - 1)
67            count=0;
68    }
```

```

68     else if(arrivalTime[count + 1] <= time)
69         count++;
70     else
71         count=0;
72 }
73
74 // printing Average Waiting Time
75 printf("\nAverage Waiting Time = %f\n",waitingTime*1.0/numberOfProcesses);
76
77 // printing Average TurnaroundTime
78 printf("Average Turnaround Time = %f\n",turnaroundTime*1.0/numberOfProcesses);
79
80 // processes order
81
82 return 0;
83 }

```

ابتدا متغیر `count` را تعریف کرده ایم، زیرا به آن به عنوان یک شمارنده در قسمت های مختلف کد نیاز خواهیم داشت. متغیر `numberOfProcesses` تعداد پردازش های موجود را نگه می دارد که به عنوان ورودی از کاربر تعداد پردازش ها را پرسیده ایم، سپس متغیر `time` را تعریف کرده ایم و با استفاده از آن می خواهیم زمان را `trace` کنیم.

همچنین متغیر `timeQuantum` مقدار `time quantum` را در خود نگه داری می کند و این مقدار را از کاربر ورودی گرفتیم.

متغیر `waitingTime` برای ذخیره کردن مجموع `waiting` `time` کل پردازش ها و متغیر `turnaroundTime` برای ذخیره کردن مجموع `turnaround time` کل پردازش ها می باشند.

سه آرایه ی `arrivalTime`، `burstTime` و `needingTime` به ترتیب، زمان رسیدن پردازش ها، `CPU Burst` و زمانی که هر پردازش نیاز دارد در هر مرحله تا کار خود را به طور کامل انجام دهد، می باشند که این مقدار به مرور زمان کم شده و زمانی که پردازش ی موردنظر کار خود را به طور کامل انجام داد به صفر می رسد.

متغیر `remainingProcesses` را نیز قبل تر تعریف کردیم و ابتدا آن را برابر کل پردازه ها قرار می دهیم اما به مرور زمان مقدار آن کم می شود تا به صفر برسد، زیرا در انتها پردازه ای نخواهد بود که سرویس داده نشده باشد.

حال مطابق کد از کاربر تعداد پردازه ها و `arrival time` و `burst time` هر کدام را گرفته ایم، سپس `time quantum` را نیز از کاربر گرفته ایم (در حلقه ی `for` موجود در خط 23).

قبل از حلقه ی `while` مقدار `time` که با آن قرار است زمان `trace` کنیم و نیز مقدار شمارنده یعنی `count` را برابر صفر قرار داده ایم.

حال ابتدا چک می کنیم که اگر پردازه ای وجود داشته باشد که هنوز به آن به اندازه ی موردنیازش (به اندازه ی `CPU Burst` اش)، به آن `CPU` نداده باشیم یعنی `> remaining processes` 0 پس وارد حلقه `while` می شویم، دو سناریو پیش می آید:

اگر پردازه ی موردنظر به زمانی کمتر از `time quantum` برای کار خود نیاز داشته باشد، این مقدار زمان بزرگتر از صفر باشد، متغیر `time` را به اندازه ی `needingTime` زیاد می کنیم و سپس `needingTime` را برابر صفر می کنیم زیرا زمان `time quantum` برای انجام شدن کامل این پردازه کافی است، همچنین مقدار `flag` را نیز 1 می کنیم (از این `flag` بعدا استفاده خواهیم کرد).

اگر پردازه ی موردنظر زمانی بیشتر از `time quantum` برای کار خود نیاز داشته باشد و این زمان مقداری مثبت باشد، به اندازه ی `time quantum` به این پردازه `CPU` می دهیم (`needingTime -= timeQuantum`)، سپس به اندازه ی `time quantum` به زمان (`time`) اضافه می کنیم.

حال چک می کنیم که اگر پردازش ای کار خود را به طور کامل انجام داده بود و دیگر CPU را نیاز نداشت(==needingTime 0 و flag برابر 1 بود)، مقدار remainingProcesses را یکی کم می کنیم و سپس برای پردازش ی موردنظر turnaround time و waiting time را محاسبه می کنیم.

$TurnaroundTime = departure\ time(\text{time in code}) - arrival\ time$

$WaitingTime = TurnaroundTime - burst\ Time$

حال این مقادیر را برای هر پردازش چاپ می کنیم و مجموع هر کدام را برای تمامی پردازش ها حساب می کنیم و بر تعداد کل پردازش ها تقسیم می کنیم و تا در نهایت مقادیر average turnaroundTime و average waitingTime را بدست آوریم.

حال می خواهیم درستی کد خود را بررسی کنیم :

```
mahan@mahan-VirtualBox: ~/Desktop
mahan@mahan-VirtualBox:~/Desktop$ gcc -o RR RR.c
mahan@mahan-VirtualBox:~/Desktop$ ./RR
Enter Total number of Processes:4
Enter Arrival Time and Burst Time for Process Number 1 :0 50
Enter Arrival Time and Burst Time for Process Number 2 :20 20
Enter Arrival Time and Burst Time for Process Number 3 :40 100
Enter Arrival Time and Burst Time for Process Number 4 :60 40
Enter the Time Quantum(q):30

Process  TurnaroundTime  WaitingTime
P1
P2          30           10
P3
P4
P1          130           80
P3
P4          110           70
P3
P3          170           70

Average Waiting Time = 57.500000
Average Turnaround Time = 110.000000
mahan@mahan-VirtualBox:~/Desktop$
```

همانطور که مشاهده می کنیم مقادیر به درستی چاپ شده اند و کد به درستی کار می کند.

توجه شود که فرض کرده ایم که کاربر موردنظر arrival time ها را به صورت صعودی و به ترتیب وارد کرده است، در نتیجه نیازی به sort کردن پرده ها در سوال نبود، زیرا در صورت سوال نیز چنین چیزی پرسیده نشده است.

QUESTION 2

توضیحات مربوط به الگوریتم shortest job first یا SJF :
در این الگوریتم زمان بندی اولویت با پردازش ای است که زمان
CPU Burst کمتری دارد، همچنین این الگوریتم یک الگوریتم
زمان بندی non-preemptive است و نسخه ی preemptive آن،
shortest-remaining-time-first نامیده می شود.

مشکل این الگوریتم فهمیدن CPU Burst هر کدام از پردازش
ها می باشد، زیرا برای استفاده از این الگوریتم باید CPU
Burst هر یک از پردازش ها را بدانیم، که دو راه برای این کار
وجود دارد، یکی آن که از خود کاربر پرسیم یا اینکه تخمین
بزنیم و یا می توانیم از میانگین CPU Burst های قبلی نیز
استفاده کنیم.

ثابت می شود که الگوریتم زمان بندی SJF در واقع بهترین و
کمترین Average Waiting Time را به ما می دهد(می توانیم
با برهان خلف ثابت کنیم).

توجه شود که در این الگوریتم ممکن است قحطی زدگی نیز
پیش آید و به این صورت است که ممکن است تعداد job هایی
با CPU Burst کم انقدر زیاد باشند که CPU به job هایی با
CPU Burst بزرگ نرسد.

روند اثبات اینکه که الگوریتم زمان بندی SJF بهترین و کمترین Average Waiting Time را به ما می دهد به صورت زیر است :

Proof that SJF optimal for average wait time

Proof by contradiction:

- Assume given m process in order of increasing runtime length, n_1, \dots, n_m
- SJF returns that order (since runs shortest jobs first).
- Assume there exists a different ordering that has the lowest average wait time; since the order is different than SJF, its not in increasing order, and so there must exist processes j and k where $n_j < n_k$ but process k runs before j
WLOG, can assume exists $n_j < n_k$, not just $n_j \leq n_k$
- Let w_i be wait time for process i under this different ordering, with average wait time $\frac{1}{m} \sum_{i=1}^m w_i$

Proof that SJF optimal for average wait time

Proof by contradiction:

- Let t_1 be the index when process k is run, t_2 when process j is run (e.g. $t_1 = 5$ means process k is fifth process run)
- We know $t_1 < t_2$. Now pretend we swap process k and j , so process j now runs at index t_1 , and k at t_2

	Case 1: wait before t_1	Case 2: wait between t_1 & t_2	Case 3: wait after t_2
Before swap	$\sum_{i=1}^{t_1} w_i$	$\sum_{i=t_1+1}^{t_2} w_i$	$\sum_{i=t_2+1}^m w_i$
	$=$	$>$	$=$
After swap	$\sum_{i=1}^{t_1} w_i$	$\sum_{i=t_1+1}^{t_2} w_i - (n_k - n_j)$	$\sum_{i=t_2+1}^m w_i$

Proof that SJF optimal for average wait time

Proof by contradiction:

- Total wait time is less after swapping process j and k , but the ordering before the swap was supposed to be optimal! ⚡
- So, must be that there does not exist such an ordering, that has lower total wait time (and so lower average wait time) than the SJF ordering



THE END