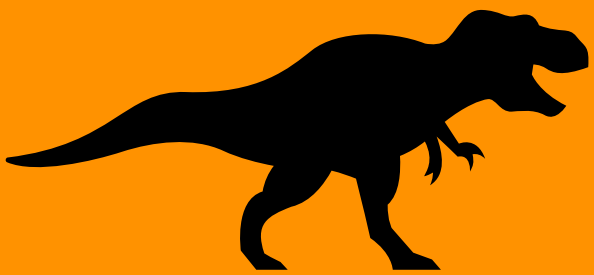
SCHEDULING

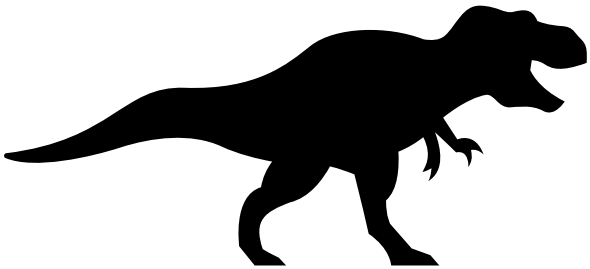
OPERATING SYSTEM



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.

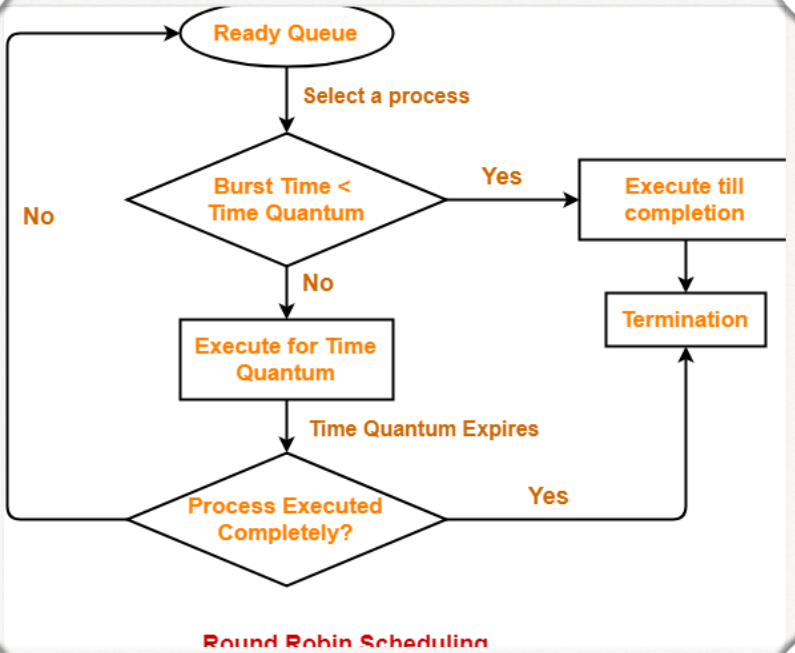
ابتدا می خواهیم کمی در مورد این الگوریتم توضیح دهیم :

این الگوریتم یک الگوریتم preemptive و responsive است و در واقع هر پردازه 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 پردازه ی بعدی میرسد و پردازه 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 درنظر 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



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 :

```c
#include<stdio.h>

int main()
{

  int count = 0;
  int numberOfProcesses = 0;
  int time = 0;
  int remainingProcesses = 0;
  int flag = 0;
  int timeQuantum = 0;
  int waitingTime = 0;
  int turnaroundTime = 0;
  int arrivalTime[100];
  int burstTime[100];
  int needingTime[100];

  printf("Enter Total number of Processes:");
  scanf("%d",&numberOfProcesses);

  remainingProcesses = numberOfProcesses;

  for(count = 0;count < numberOfProcesses;count++)
  {
    int processesNumber = count + 1;
    printf("Enter Arrival Time and Burst Time for Process Number %d :",processesNumber);
    scanf("%d",&arrivalTime[count]);
    scanf("%d",&burstTime[count]);
    needingTime[count]=burstTime[count];
  }

  printf("Enter the Time Quantum(q):");
  scanf("%d",&timeQuantum);
  printf("\nProcess\t TurnaroundTime WaitingTime\n\n");

  time = 0;
  count = 0;
  while(remainingProcesses > 0)
  {
    if(needingTime[count] <= timeQuantum && needingTime[count] > 0)
    {
      time+=needingTime[count];
      needingTime[count] = 0;
      flag = 1;
    }
    else if(needingTime[count] > 0)
    {
      needingTime[count] -= timeQuantum;
      time+=timeQuantum;
    }
    if(needingTime[count] == 0 && flag == 1)
    {
      remainingProcesses--;
      int processNumber = count + 1;
      int processTurnaroundTime = time - arrivalTime[count];
      int processWaitingTime = time - arrivalTime[count] - burstTime[count];
      printf("P%d\t \t%d\t \t%d\n", processNumber, processTurnaroundTime,processWaitingTime);

      waitingTime += time-arrivalTime[count]-burstTime[count]; // Turnaround Time - burst Time
      turnaroundTime += time-arrivalTime[count]; // departure Time(time variable in here) - arrival Time

      flag=0;
    }
    if(count == numberOfProcesses - 1)
      count=0;
    else if(arrivalTime[count + 1] <= time)
      count++;
    else
```

```
68      else
69          count=0;
70      }
71
72      // printing Average Waiting Time
73      printf("\nAverage Waiting Time = %f\n",waitingTime*1.0/numberOfProcesses);
74
75      // printing Average TurnaroundTime
76      printf("Average Turnaround Time = %f\n",turnaroundTime*1.0/numberOfProcesses);
77
78      return 0;
79  }
```

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 پردازه های موجود را نگه می دارد که به عنوان ورودی از کاربر تعداد پردازه 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 پردازه ها و متغیر turnaroundTime برای ذخیره کردن مجموع turnaround time کل پردازه ها می باشند.

سه آرایه ی arrivalTime، burstTime و needingTime به ترتیب، زمان رسیدن پردازه ها، CPU Burst و زمانی که هر پردازه 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 پردازه 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 را 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 اندازهی 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 پردازه ای کار خود را به طور کامل انجام داده بود و دیگر CPU را نیاز نداشت(== needingTime

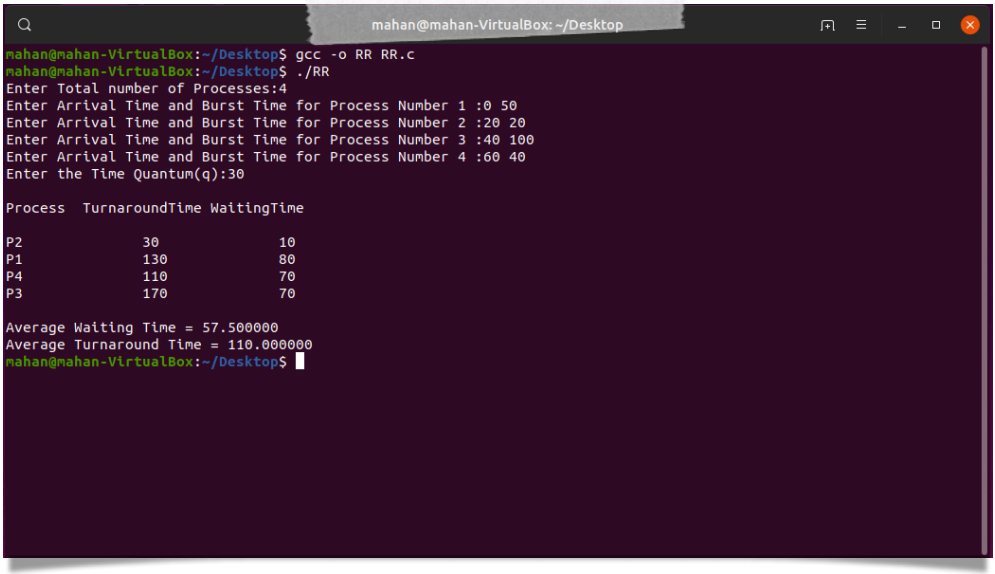
0 و flag برابر 1 بود)، مقدار remainingProcesses را یکی کم می کنیم و سپس برای پردازه ی موردنظر turnaround time و waiting time را محاسبه می کنیم.

TurnaroundTime = departure time(time in code) - arrival time

WaitingTime = TurnaroundTime - burst Time

حال این مقادیر را برای هر پردازه چاپ می کنیم و مجموع هر کدام را برای تمامی پردازه ها حساب می کنیم و بر تعداد کل پردازه 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 :



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

QUESTION 2

توضیحات مربوط به الگوریتم shortest job first یا SJF :

در این الگوریتم زمان بندی اولویت با پردازه 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 فهیمدن CPU Burst هر کدام از پردازه ها می باشد، زیرا برای استفاده از این الگوریتم باید CPU Burst هر یک از پردازه ها را بدانیم، که دو راه برای این کار وجود دارد، یکی آن که از خود کاربر بپرسیم 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, $n_1, \ldots, n_m$
- 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

  $\text{WLOG, can assume exists } n_j < n_k, \text{ not just } n_j \leq n_k$
- 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