UNIVERSIDADE
DE ÉVORA

Relatório do 1º Trabalho de Sistemas Operativos I

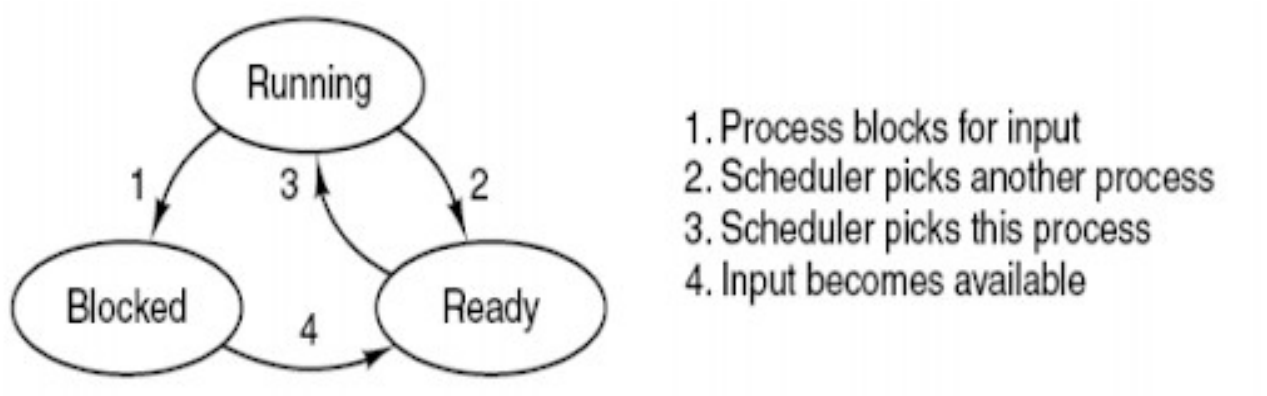
2019/2020

Engenharia Informática

-Gonçalo Correia, nº 43735

1. Introdução

Este trabalho tem como objetivo simular o funcionamento de um sistema operativo, mais concretamente, o funcionamento do escalonamento de processos, neste caso baseado na que utiliza modelos de 3 estados para lidar com determinados processos (figura abaixo a ilustrar este modelo).



Como se pode observar pela imagem acima, à medida que os processos entram no processador é feito um “scheduling”, composto por 3 etapas ou estados, READY, RUN e BLOCKED. Como os nomes, indicam, READY é o conjunto de processos que se encontram no processador, prontos para serem executados, esperando apenas para que outro processo termine ou seja bloqueado, como vamos ver mais à frente. Em RUN é o estado em que um processo está efetivamente a ser executado, apenas saindo deste estado quando este for terminado ou bloqueado por algum pedido de I/O, por exemplo. É em BLOCKED que se encontram estes processos que saíram de RUN devido a alguma interrupção e que quando saem deste estado voltam para READY à espera de serem executados de novo.
O escalonamento de processos pode diferir dependendo do tipo de algoritmo. Neste trabalho irá ser abordado First Come First Served (FCFS) e Round Robin:

- First Come First Served

No caso deste algoritmo de escalonamento, os processos vão sendo executados por “ordem de chegada” e tendem a continuar o seu processo de RUN até estarem completos ou até serem interrompidos, sendo enviados para BLOCKED quando isto acontece. Sendo que o próximo processo a ser executado vai ser o primeiro da “fila” de READY, ou seja, o que chegou há menos tempo, e assim sucessivamente até que todos os processos sejam executados.

- Round Robin

Em RR, existem algumas semelhanças a FCFS. Neste caso deparamo-nos com a existência de um quantum, ou seja, uma variável que determina quanto tempo é que cada processo fica no estado RUN, sendo que todos os processos vão estar todos a passar os mesmos instantes nesse estado. Quando estes são removidos de RUN devido ao limite do quantum ter sido atingido, os processos são postos de novo em READY, até que chegue novamente a vez de irem para o estado seguinte. Já a passagem de RUN para BLOCKED é exatamente igual a FCFS, ou seja, caso um processo seja bloqueado antes do quantum terminar, é passado para BLOCKED, tendo que passar depois para o READY antes de ir para RUN.

2. Implementação

No desenvolvimento deste trabalho resolvi criar apenas 3 funções, uma para executar o FCFS, outra para Round Robin e uma função main.

Comecei por criar uma struct processo, que contém os vários parâmetros que cada processo contém, ou seja, o seu PID, o instante em que o processo chega ao CPU (t_inicio) ,dois arrays, um para guardar os burst times de cada processo e outro para guardar os tempos de espera no acesso a I/O e também dois index para cada um destes arrays, cada um determinando o tamanho de cada array, tais serão usados mais à frente nas funções do simulador propriamente dito. Depois disto fiz a criação dos 4 processos p1, p2 , p3 e p4 para que possa ser lida a informação do ficheiro e colocada nestes.

```
struct process{
    int PID;
    int t_inicio;
    int burst[5];
    int index_b;
    int io[5];
    int index_io;
};
```

Em seguida, já dentro da função main, tratei da parte da leitura dos ficheiros, neste caso, do ficheiro “input1.txt”. Utilizando vários fscans para guardar o conteúdo do ficheiro nesta struct. Começando por ler os PID e t_inicio e de seguida ir guardando os tempos de burst e de espera nos respetivos arrays até que apareça algum valor maior que 100, tratando-se de um PID, logo, é sinal de que temos de iniciar o scan de um novo processo repetindo este mesmo procedimento, até que cheguemos ao fim do ficheiro daí ir definindo uma variável depois de definir os burst times (o último dígito do ficheiro irá sempre ser um burst time) para que se essa variável for igual a EOF (end of file) os fscans terminarem a sua tarefa. (exemplo abaixo de um dos procedimentos para o scan de um processo).

```
while(aux <= 100){
  fscanf(p, "%d", &aux);

  if(aux >= 100){
    break;
  } else{
    p1.burst[b_index] = aux;
    p1.index_b ++;
    b_index ++;
    int state = fscanf(p, "%d", &aux);
    if(aux >= 100 || state == EOF){
      break;
    }else{
      p1.io[io_index] = aux;
      p1.index_io ++;
      io_index ++;
    }
  }
}
```

Nesta função foram foram inicializadas várias variáveis que funcionam como índice para ajudar a percorrer os arrays da struct.

Ainda na função main, foi criado uma espécie de menu, de modo a que o utilizador possa escolher qual dos algoritmos quer executar (FCFS ou RR).

Função FCFS:

Para esta função, que toma como argumentos os 4 processos do tipo “struct process”, inicialmente comecei por criar dois arrays para READY e BLOCKED e para RUN apenas uma variável do tipo int, uma vez que só pode estar um processo neste estado. Criei uma série de índices para percorrer os arrays e variáveis finished para saber quando um processo termina, de modo a que deixe de continuar a “correr”.

Resolvi criar todo o conteúdo do algoritmo dentro de um for que representa os vários instantes (cada ciclo for corresponde a um instante do clock), que está previamente definido para 60 instantes, podendo ser alterado para outros valores.

De seguida começo por criar if’s para quando uma função “entra no CPU”, a colocar devidamente na fila de READY ou em RUN, caso não esteja nenhum processo a correr. Isto também se aplica para quando um processo sai de BLOCKED, como vamos ver mais à frente.

```
if(p2.t_inicio <= clock && run != p2.PID && blocked[0] != p2.PID && blocked[1] != p2.PID && finishedp2 == 0){

  if(run == 0){
    run = p2.PID;
    p2counterb = 0;
  } else if(p2.PID != ready[0] && p2.PID != ready[1] && p2.PID != ready[2]) {
    for( int readyc = 0; readyc < 3; readyc ++){
       if(ready[readyc] == 0){
        ready[readyc] = p2.PID;
        break;
       } else{
         continue;
       }
    }
  }
}
```

Imagem 3: Exemplo de um dos if’s utilizados para dar “enqueue” nos processos.

O primeiro processo a entrar vai diretamente para o estado RUN, mas este é um caso diferente, pois o resto dos processos têm que esperar que outros processos acabem de correr. Quando um processo entra em RUN, é reordenada a queue de READY, de forma a retirar o processo que estava anteriormente em primeiro e avançar os restantes na fila. Depois é feito um if para “atrasar” o processo, pois caso contrário os processos que entram em RUN iriram sobrepor-se ao último ciclo do RUN anterior.

Nos instantes seguintes um processo vai correr tantas vezes quanto o número correspodente ao burst time atual.
Quando acaba de correr um processo, é alterada a variável run para o PID que estiver em primeiro na fila de READY e o processo que acabou de correr é posto no array blocked.

```
if(run == p3.PID){
  if(ready[0] == p3.PID){
    ready[0] = ready[1];
    ready[1] = ready[2];
    ready[2] = 0;
  }
  if(p3counterb == -1){
    p3counterb ++;
  }
  else{
   printf("%d ", p3.PID);
   p3counterb ++;
  }
  if(p3counterb == p3.burst[p3x]){
    p3counterb = -1;
    p3x ++;
    if(p3x >= p3.index_b){
      finishedp3 = 1;
    }
    if(finishedp3 == 0){
      if(blocked[0] == 0){
        blocked[0] = p3.PID;
      } else{
          blocked[1] = p3.PID;
        }
    }
    run = ready[0];
  }
}
```

Imagem 4: Processo de RUN no programa usando If

Já em BLOCKED os processos são mantidos tanto tempo quanto for o valor que está inserido no array “io”, sendo depois colocados de novo na queue.

```
if (p4.PID == blocked[0]){
  if(p4counterio == -1){
    p4counterio ++;
  }
  else{
   printf("%d", p4.PID);
   p4counterio ++;
  }
  if(p4counterio == p4.io[p4y]){
    p4counterio = -1;
    p4y++;
    blocked[0] = blocked[1];
    blocked[1] = 0;
  }
}
```

Função roundrobin:

Para a criação desta função, apenas alterei a fase RUN em relação ao FCFS, porque de resto, são bastante idênticos. Comecei por adicionar um #define MAX 3, sendo este MAX o nosso quantum (ajustável) deste algoritmo. Criei index's para cada um dos processos de modo a contar os instantes que estes passam no RUN, de modo a que (no caso de MAX = 3) os processos apenas corram no máximo 3 vezes seguidas, sendo colocados de novo na fila após atingir esse patamar.

```
if(run == p1.PID){
  if(ready[0] == p1.PID){
    ready[0] = ready[1];
    ready[1] = ready[2];
    ready[2] = 0;
  }
  if(p1counterb == -1 || p1aux == 1){
    if(p1counterb == -1){
      p1counterb ++;
    } p1aux = 0;
  }
  else{
   printf("%d ", p1.PID);
   p1counterb ++;
   rrcounterp1 ++;
  }
  if(p1counterb == p1.burst[p1x]){
    p1counterb = -1;
    p1x++;
    rrcounterp1 = 0;
    if(p1x >= p1.index_b){
      finishedp1 = 1;
    }
    if(finishedp1 == 0){
      if(blocked[0] == 0){
        blocked[0] = p1.PID;
      } else{
          blocked[1] = p1.PID;
        }
    }
    run = ready[0];
  }
  else if(rrcounterp1 == MAX){
    rrcounterp1 = 0;
    p1aux = 1;
    if(ready[0] != 0){
        run = ready[0];
    }
  }
}
```

Imagem 6: Aspeto de um RUN na função roundrobin

Outputs de ambas as funções com o ficheiro pedido:

```
0  | READY 101          RUN 100 BLOCKED
1  | READY 200 300      RUN 101 BLOCKED 100
2  | READY 200 300      RUN 101 BLOCKED 100
3  | READY 200 300      RUN 101 BLOCKED 100
4  | READY 200 300 100  RUN 101 BLOCKED
5  | READY 300 100      RUN 200 BLOCKED 101
6  | READY 300 100      RUN 200 BLOCKED 101
7  | READY 100          RUN 300 BLOCKED 101
8  | READY 100          RUN 300 BLOCKED 101
9  | READY 100 101      RUN 300 BLOCKED 200
10 | READY 100 101      RUN 300 BLOCKED 200
11 | READY 100 101      RUN 300 BLOCKED 200
12 | READY 100 101      RUN 300 BLOCKED 200
13 | READY 100 101      RUN 300 BLOCKED 200
14 | READY 100 101 200  RUN BLOCKED 300
15 | READY 101 200      RUN 100 BLOCKED 300
16 | READY 101 200      RUN 100 BLOCKED 300
17 | READY 101 200      RUN 100 BLOCKED 300
18 | READY 101 200      RUN 100 BLOCKED 300
19 | READY 101 200      RUN 100 BLOCKED 300
20 | READY 101 200 300  RUN 100 BLOCKED
21 | READY 101 200 300  RUN 100 BLOCKED
22 | READY 101 200 300  RUN 100 BLOCKED
23 | READY 101 200 300  RUN 100 BLOCKED
24 | READY 101 200 300  RUN 100 BLOCKED
25 | READY 200 300      RUN 101 BLOCKED 100
26 | READY 200 300      RUN 101 BLOCKED 100
27 | READY 300          RUN 200 BLOCKED 100
28 | READY 100          RUN 300 BLOCKED 200
29 | READY 100          RUN BLOCKED 200
30 | READY 200          RUN 100 BLOCKED
31 | READY 200          RUN 100 BLOCKED
32 | READY 200          RUN 100 BLOCKED
33 | READY 200          RUN 100 BLOCKED
34 | READY 200          RUN 100 BLOCKED
35 | READY 200          RUN 100 BLOCKED
36 | READY              RUN 200 BLOCKED
37 | READY              RUN 200 BLOCKED
38 | READY              RUN 200 BLOCKED
39 | READY              RUN BLOCKED
```

Imagem 7: Output da função fcfs

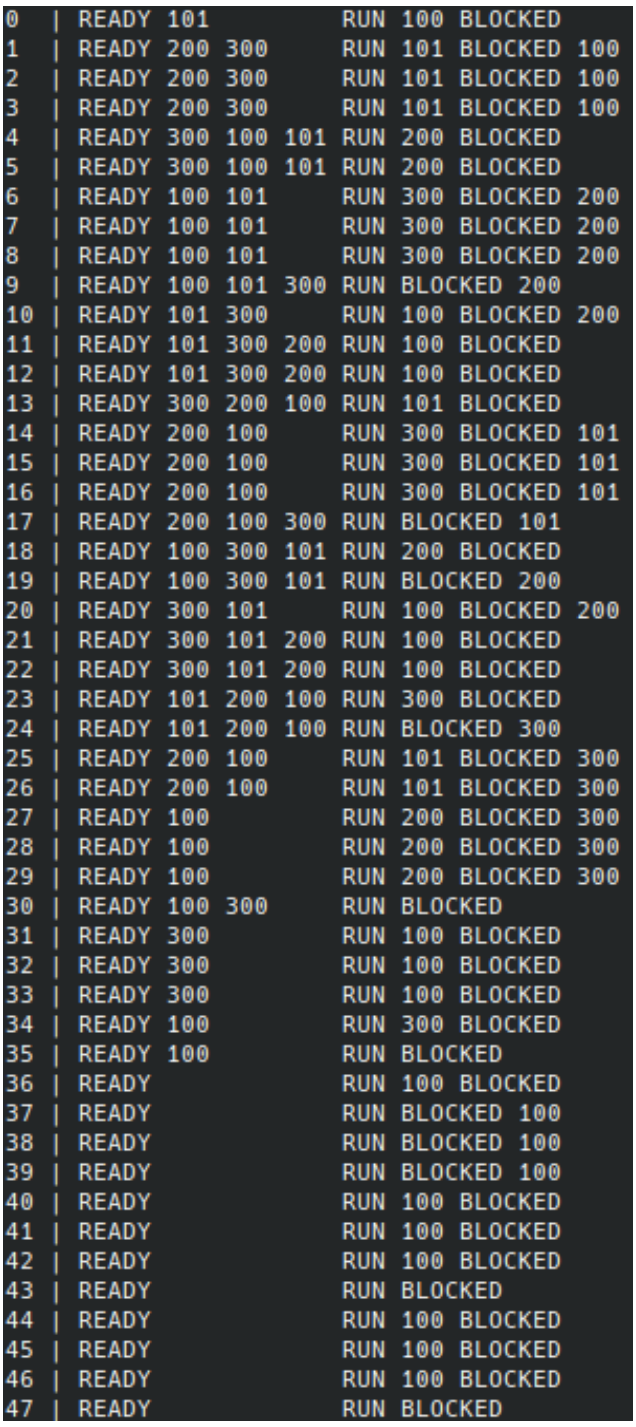


Imagem 8: Ouput da função roundrobin

3. Problemas e situações adversas

No decorrer da implementação deste código surgiram inúmeras dificuldades, visto que a linguagem C não está dominada por mim a 100% e gostava de reconhecer algumas falhas que aconteceram ao longo deste trabalho.
Olhando para os outputs colocados nas páginas acima verificam-se 2 coisas que correspondem a problemas que não consegui resolver. Tais são o facto de por vezes no RUN não existir nenhum processo e de em blocked apenas haver um processo de cada vez, quando poderiam e deveriam existir mais do que um. Outro aspeto deve-se ao facto de que este programa apenas suporta 4 processos no máximo, de modo a corresponder aos inputs dispostos pelo professor no Moodle, quando por ventura poderia ter sido otimizado para ler mais processos, dependendo do input.

4. Conclusão

Com a realização deste trabalho consegui desenvolver bastante a minha capacidade de trabalhar com a linguagem C, que era bastante reduzida, mas principalmente, trabalhar com o escalonamento dos processos dentro do CPU de um computador, o que me ajudou a perceber de forma muito mais clara este tema e consequentemente deixou-me com muito mais interesse nesta disciplina e a realizar as próximas tarefas com uma maior eficácia e empenho.