



## RELATÓRIO - TRABALHO PRÁTICO

### SISTEMAS OPERATIVOS I

#### *SIMULADOR DE ESCALONAMENTO*

0	READY		RUN 100	BLOCKED
1	READY		RUN 100	BLOCKED
2	READY		RUN 100	BLOCKED
3	READY	110	RUN 100	BLOCKED
4	READY	110 122	RUN 100	BLOCKED
5	READY		RUN	BLOCKED 100

**Trabalho realizado por:**

Diogo Castanho, 42496



## 1.1 - Introdução

O objetivo deste trabalho é implementar um escalonamento FCFS, e Round Robin, Quantum=3 mas configurável (#define) na linguagem C.

O simulador pretendido baseia-se numa arquitetura sobre o modelo de 3 estados que consome programas constituídos por um conjunto instruções.

As instruções são codificadas por uma sequência de números inteiros representando alternadamente o tempo (burst) de CPU e um tipo de pedido I/O (por exemplo acesso ao disco), e.g

**5 1 3 3 5**

Esta sequência por exemplo, é equivalente à seguinte sequência:

- 5 instantes no CPU
- acesso a I/O com espera de 1 instante
- 3 instantes no CPU
- acesso a I/O com espera de 3 instantes
- 5 instantes no CPU

Cada sequência de instruções tem sempre um número ímpar de elementos, isto é, termina sempre com um burst de CPU.

Os ficheiros de teste têm a indicação do PID e do instante de entrada, seguida da sequência de instruções, separadas por espaços, e.g.

(Continuação 1.1)

Assim, a implementação do simulador ainda terá algumas restrições:

Quando no mesmo instante, um processo novo ou vindo de BLOCKED, e /ou do RUN pretendem entrar na fila de READY:

- O vindo do BLOCKED tem prioridade;
- Seguido do de RUN;
- Por fim o processo novo;

O output deverá apresentar em cada instante a lista de processos (indicando os PIDs) em cada um dos estados: READY, RUN e BLOCKED, e.g.

0		READY->			RUN->	100		BLOCKED->	
1		READY->			RUN->	100		BLOCKED->	
2		READY->			RUN->	100		BLOCKED->	
3		READY->	110		RUN->	100		BLOCKED->	
4		READY->	110 122		RUN->	100		BLOCKED->	
5		READY->	122		RUN->	110		BLOCKED->	100
6		READY->	122 100		RUN->	110		BLOCKED->	
7		READY->	100 101		RUN->	122		BLOCKED->	110
8		READY->	100 101		RUN->	122		BLOCKED->	110
9		READY->	100 101 110		RUN->	122		BLOCKED->	
10		READY->	100 101 110		RUN->	122		BLOCKED->	
11		READY->	100 101 110		RUN->	122		BLOCKED->	
12		READY->	101 110		RUN->	100		BLOCKED->	122

**Legenda:** Exemplo do funcionamento do simulador implementado;

## 1.2 - Decisões tomadas na realização do trabalho e observações

Inicialmente, tentou perceber-se o problema em causa (neste caso, a programação de um simulador de escalonamento RR e FCFS seguindo diversas regras impostas inicialmente) lendo mais que uma vez o enunciado proposto.

Após se perceber bem os objetivos do trabalho, começou-se por fazer o escalonamento FCFS por considerar-se mais “simples”.

Após vários testes de algoritmo falhados, principalmente devido a erros de sintaxe (erros na digitação do código por vezes, por falta de concentração) e erros semânticos (onde o programa funcionava, porém, não fazia o pedido inicialmente), foi-se implementando o programa e corrigindo aspetos função a função.

Ainda assim, após diversas tentativas e revoluções no código, chegou-se a um output que parece pelo menos parece o correto no escalonamento FCFS. Já o ROUND ROBIN apresentou diversos problemas e não foi possível concluir o mesmo talvez nem apresentar output's que eram objetivo inicialmente.

## 1.3 - Desenvolvimento do programa

Para além das funções básicas e frequentes da biblioteca stdio.h, tais como *printf* e *scanf*, foram implementadas outras funções necessárias para o bom funcionamento do simulador e implementadas estruturas e filas para gerir trocas de processos entre **READY**, **RUN** e **BLOCKED**.

## (1.3 - Continuação)

**O trabalho foi realizado e organizado pela seguinte ordem:**

**1º**

A primeira parte a ser implementada foi a criação de structs.

Foi criada uma struct Processo onde é possível aceder ao:

- PID;
- Tempo de chegada;
- Tempos de burst;
- Tempos de wait;
- Índice de burst (Para controlar qual o burst a ser decrementado);
- Índice de wait (Para controlar qual o wait a ser decrementado);
- Número de burst's;
- Número de wait's;

de cada processo.

### (1.3 - Continuação)

#### 2º

Foi criada uma struct Fila onde é possível aceder ao:

- Tamanho da fila;
- Primeiro elemento da fila;
- Último elemento da fila;
- Capacidade da fila;
- Array de processos na fila;

#### 3º

Com a necessidade de implementação de filas para gerir trocas de processos foram criadas as seguintes funções:

- **void criarFila (Fila \*f, int c)**, a função não retorna nada, porém irá criar uma fila alocando memória com a capacidade do segundo argumento.
- **void inserir (Fila \*f, Processo \*processo)**, a função não retorna nada, porém irá inserir o processo introduzido no segundo argumento na fila correspondente ao segundo argumento.
- **void remover(Fila \*f)**, a função também não retorna nada, serve apenas para remover um elemento da fila. Não é necessário um argumento para o processo pois o processo removido será sempre o primeiro da fila.

### (1.3 - Continuação)

→ **int estaVazia(Fila \*f)**, a função retorna “verdadeiro” apenas se a fila estiver vazia.

→ **void mostrarFila(Fila \*f)**, a função não tem qualquer valor de retorno. É chamada tendo como argumento a fila em questão e esta mostra o conteúdo da mesma. Neste caso, processo a processo.

Para gerir os processos considerou-se que apenas eram necessárias estas funções correspondentes às filas para auxiliar o simulador.

4º

Foi lido o input inicial de um ficheiro sendo colocados todos os inteiros dentro de um array dinâmico, isto é, com capacidade dinâmica (alterável).

5º

Foi criada a função **Processo \*novo\_processo()** para inicializar um novo processo.

Dentro desta, são inicializadas as variáveis dos índices para percorrer o array de burst e o array de wait com valor 0.

## (1.3 - Continuação)

6º

Foi criada a função **int dgtlen(int n)**, sendo esta usada para por exemplo, quando percorrer o array onde estão guardados os valores do input fazer a contagem do numero de dígitos de um PID e contabilizar na contagem do número de processos visto que, neste trabalho o PID é único e tem sempre 3 dígitos diferindo assim de outros valores do input.

A função em si devolve o número de dígitos do valor introduzido como argumento.

7º

Em seguida era necessário, guardar o input por processos, “arrumando” cada um dentro da “sua estrutura”.

Assim, foi criada uma função **Processo \*arruma\_processo(int lista[], int indice\_inicial, int indice\_final)** cujo os argumentos serão o array onde estão guardados os valores do input lido, o índice onde começa o processo e o índice onde termina o processo.

A função irá atribuir os valores (limitados por esses índices) às respetivas variáveis da estrutura **Processo** (ex. PID, burst, wait.).



## (1.3 - Continuação)

8º

Depois de implementado tudo o que foi explicado anteriormente e ter sido lido o input, foi calculado o tempo total de burst do mesmo para saber o número de instantes máximo que o simulador irá apresentar.

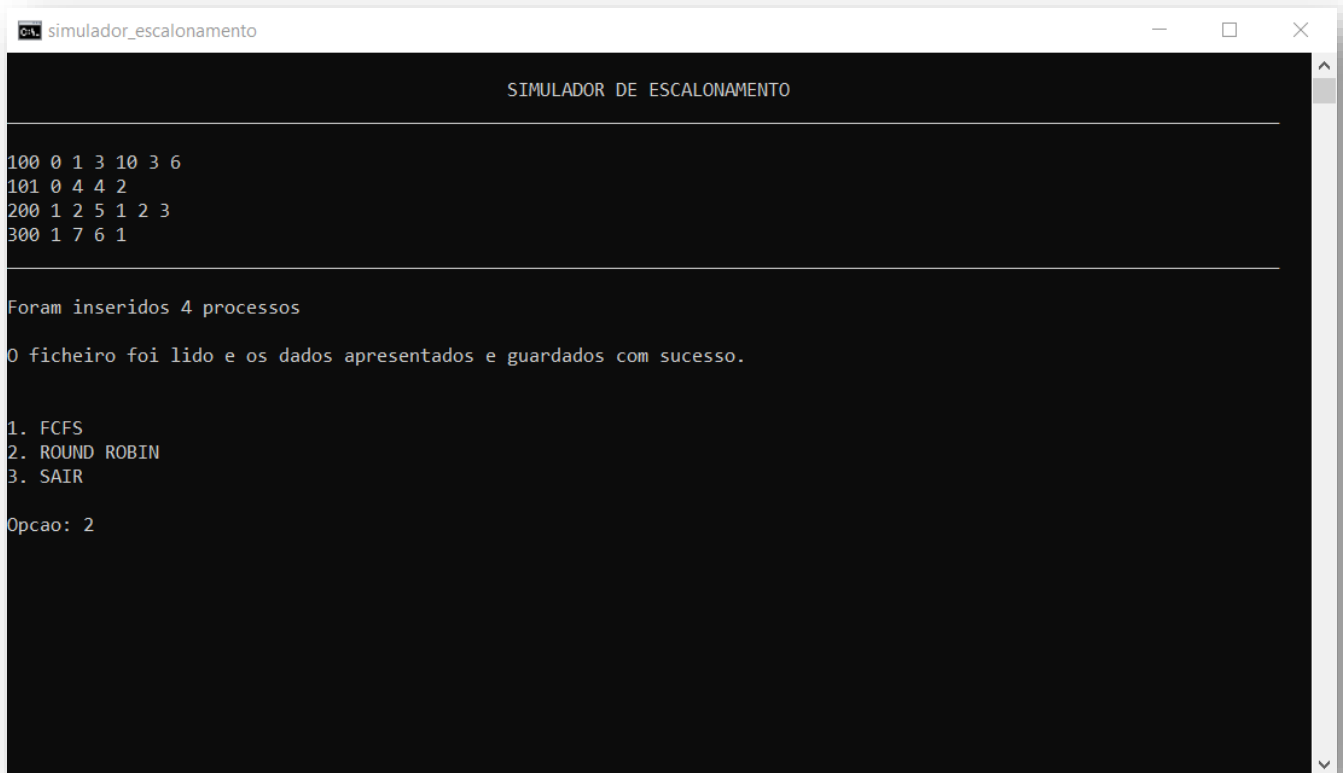
Após isso e a contagem do número de processos deu-se início à simulação.

9º

Já na simulação, foi criada então posteriormente (e por necessidade) a função **void ready\_check(Fila \*f, Processo \*lista[], int instante){**.

A função serve, neste caso, para inserir um processo na **fila ready** assim que o seu tempo de chegada corresponder ao instante atual do ciclo.

## 1.4 – Demonstração do output gerado pelo simulador para o escalonamento FCFS



```
simulador_escalonamento

SIMULADOR DE ESCALONAMENTO

100 0 1 3 10 3 6
101 0 4 4 2
200 1 2 5 1 2 3
300 1 7 6 1

Foram inseridos 4 processos

O ficheiro foi lido e os dados apresentados e guardados com sucesso.

1. FCFS
2. ROUND ROBIN
3. SAIR

Opcao: 2
```

Após inserida a opção 2 (FCFS) o programa apresenta:

*(imagem inserida na página seguinte)*

## 1.4 – Demonstração do output gerado pelo simulador para o escalonamento

### FCFS

(Continuação)

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-> 101 200
8	READY-> 100	RUN-> 300	BLOCKED-> 101 200
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-> 100 101 200	RUN-> 300	BLOCKED->
13	READY-> 100 101 200	RUN-> 300	BLOCKED->
14	READY-> 101 200	RUN-> 100	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-> 200 300	RUN-> 101	BLOCKED-> 100
25	READY-> 200 300	RUN-> 101	BLOCKED-> 100
26	READY-> 300	RUN-> 200	BLOCKED-> 100
27	READY-> 100	RUN-> 300	BLOCKED-> 200
28	READY->	RUN-> 100	BLOCKED-> 200
29	READY-> 200	RUN-> 100	BLOCKED->
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->	RUN-> 200	BLOCKED->
35	READY->	RUN-> 200	BLOCKED->
36	READY->	RUN-> 200	BLOCKED->