# Teste 2017

(I)

```
int main(...)                    → P
{
    switch(fork()){              → P + F1
      case 0:  printf("A\n");    → F1
               break;            → F1
      default: printf("B\n");    → P
               wait(NULL)        → P
               print("C\n")      → P
    }                            → P + F
    return c;                    → P+F
}
```

a) P → processo pai

$F_1$ → processo filho

b)

| | |
|---|---|
| A | B |
| B | A |
| C | C |

c)

processo - é um programa em execução; o seu estado é definido pela atividade no momento; pode estar em vários estados; tem uma estrutura de so associado; pode ter vários

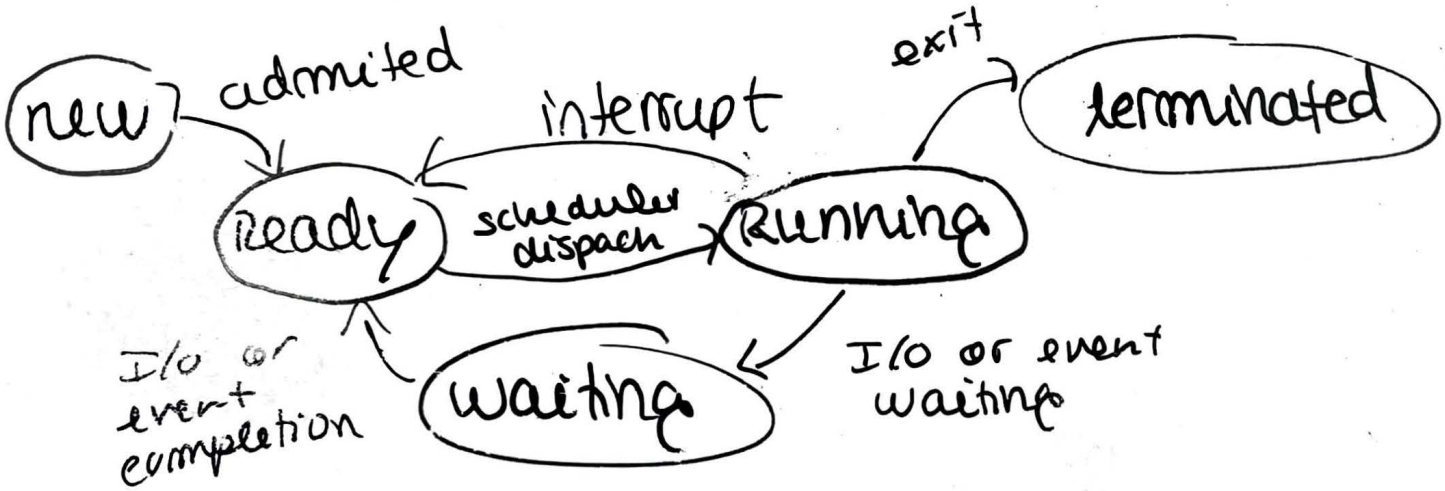
fluxos de execução - threads

thread - unidade básica do CPU constituído por threadid, stack, registers; As threads partilham data, ficheiros e código; A criação de processos envolve a cópia desses ficheiros (e data e código) mas não a sua partilha.

O que os destingue:

- o processo contém threads mas uma thread não pode conter um processo
- o processo demora ⊕ tempo a terminar, criar e mudar de contexto que as threads. É ⊖ eficiente e gasta ⊕ recursos.
- As threads usam memória partilhada e os processos não.

d) diagrama de estados



e)

Todas as linhas podem estar no estado Ready ou Running.

(Chamadas ao sistema 3, 5, 7
linhas I/O 4, 8
↳ podem estar em wait

Quando o programa termina., está em terminated

(II)

a)

```
int left(int f){
    int f_left = f;
    return f_left
}
```

```
int right(int f){
    int f_right =
        = (f+1) % N;
    return f_right
```

c) Para corrigir o problema temos que negar uma das condições de deadlock

- exclusão mutua
- espera c/retenção
- ñ libertação
- espera circular

⇒ negar a espera circular

filosofos 1 a 4 executam o código dado.

O 5 executa o seguinte código, de modo a pegar 1º no garfo da direita

b) f1

```
thunk();
forks[left(f)].down();
```

↳ s1 = 0

f2

```
thunk();
forks[left(f)].down();
```

↳ s2 = 0

Igual para f3, f4 e f5

⇓

s3 = 0
s4 = 0
s5 = 0

Deadlock (bloqueio) pois todos os semáforos estão a zero.
se todos os filósofos pegam 1º no garfo da esquerda quando alguem tentar pegar no da sua direita vai ficar bloqueado

c) (cont.)

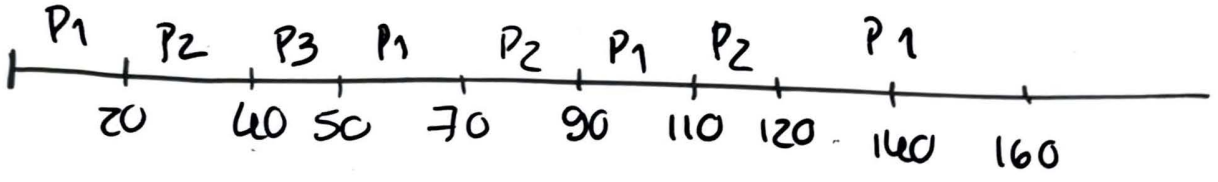
```
void filosofo(int f){
  while(true){
    thunk();
    forks[right(f)].down();
    forks[left(f)].down();
    ...
```

o resto é igual ao código dado

(IV)

$P_1$ – 100 ms
$P_2$ – 50 ms
$P_3$ – 10 ms

a) RR $q = 20$ ms




b) Tempos de espera

$P_1 = 30 + 20 + 10 = 60$

$P_2 = 20 + 30 + 20 = 70$

$P_3 = 40$

$$\overline{t}_{espera} = \frac{60 + 70 + 40}{3} = \frac{170}{3} \approx 56{,}7 \text{ ms}$$

c)

RR → processos são executados durante um tempo limitado; preemptivo

FCFS → é executado o 1º processo que chega; ñ preemptivo

vantagens do RR em relação ao FCFS

- livre de inanição, ou seja, como é um algoritmo preemptivo, não há o risco de algum processo não ser executado por causa de processos com maior prioridade
- garante um tempo de resposta ⊕ rápido (e menor tempo de espera)

vantagens do FCFS ???

- é ⊕ simples por ser não preemptivo (menos complexo)

escalonar aplicações → RR
calculo intensivo → FCFS

(V)

a)

| endereço | conteúdo |
|---|---|
| 0 | G |
| 1 | B |
| 2 | |
| 3 | C |
| 4 | F |
| 5 | A |
| 6 | |
| 7 | |

| endereço | conteúdo |
|---|---|
| 8 | D |
| 9 | E |
| 10 | |
| 11 | H |
| 12 | |
| 13 | |
| 14 | |
| 15 | |

b)

O SO garante que a tabela de página não é alterada por outros processos. Para isso há um base register e um limit register. O CPU compara todos os endereços gerados por um processo com esses registos. Caso estejam fora do limite, o SO emite uma ____ ??

Estes valores só podem ser alterados pelo SO.

O SO mantém uma cópia da ~~tabela de endereços~~ página de cada processo. Para evitar acessos inválidos à memória tbm se pode usar o valid-invalid bit na page table.

c) Objetivos da máquina virtual

- eficiência → partilha; manter na memória apenas o necessário
- segurança → impedir a atribuição de conteúdo de memória de outros processos
- transparência → acesso a mt memória como se toda a memória lhe pertencesse
- partilha de memória → vários processos acedem à mesma zona de memória de forma controlada

III

Recursos disponíveis

| R1 | R2 | R3 |
|---|---|---|
| 0 | 2 | 3 |

Estado dos processos

| | Recursos já adquiridos | | | Recursos a pedir | | |
|---|---|---|---|---|---|---|
| | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 3 | 1 | 4 | 2 | 0 | 0 |
| P2 | 2 | 1 | 0 | 0 | 2 | 1 |
| P3 | 2 | 1 | 0 | 3 | 1 | 0 |

a) Sim, os processos podem terminar

sequência → P2 → P1 → P3

$$\text{Disponíveis} = \begin{bmatrix} 0 \\ 2 \\ 3 \end{bmatrix} \begin{matrix} R1 \\ R2 \\ R3 \end{matrix} = D$$

$$\text{Adquiridos} = \begin{bmatrix} 3 & 1 & 4 \\ 2 & 1 & 0 \\ 2 & 1 & 0 \end{bmatrix} = A$$

$$\text{Need} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 1 \\ 3 & 1 & 0 \end{bmatrix} = N$$

1º P2

Para o processo ser executado, os recursos necessários têm que ser ≤ aos disponíveis

$$\begin{bmatrix}0\\2\\1\end{bmatrix} \leq \begin{bmatrix}0\\2\\3\end{bmatrix} \Leftrightarrow N \leq D$$

sobraram

$$\begin{bmatrix}0\\0\\2\end{bmatrix}$$

$$\begin{bmatrix}2\\1\\0\end{bmatrix} + \begin{bmatrix}0\\2\\1\end{bmatrix} = \begin{bmatrix}2\\3\\1\end{bmatrix}$$

$$D = \begin{bmatrix}2\\3\\3\end{bmatrix} \begin{matrix}R1\\R2\\R3\end{matrix}$$

2º P1

$$\begin{bmatrix}2\\0\\0\end{bmatrix} \leq \begin{bmatrix}2\\3\\3\end{bmatrix} \Leftrightarrow N \leq D \rightarrow \text{pode ser executado}$$

sobraram

$$\begin{bmatrix}0\\3\\3\end{bmatrix}$$

$$\begin{bmatrix}3\\1\\4\end{bmatrix} + \begin{bmatrix}2\\0\\0\end{bmatrix} = \begin{bmatrix}5\\1\\4\end{bmatrix}$$

$$D = \begin{bmatrix}5\\4\\7\end{bmatrix} \begin{matrix}R1\\R2\\R3\end{matrix}$$

3º P3

$$\begin{bmatrix}3\\1\\0\end{bmatrix} \leq \begin{bmatrix}5\\4\\7\end{bmatrix}$$

$$\begin{bmatrix}2\\1\\0\end{bmatrix} + \begin{bmatrix}3\\1\\0\end{bmatrix} = \begin{bmatrix}5\\2\\0\end{bmatrix}$$

sobraram

$$\begin{bmatrix}2\\3\\7\end{bmatrix}$$

$$D = \begin{bmatrix}7\\5\\7\end{bmatrix}$$

b)

P3 pede $\begin{bmatrix}0\\1\\0\end{bmatrix}$

$$\begin{bmatrix}0\\1\\0\end{bmatrix} \leq \begin{bmatrix}3\\1\\0\end{bmatrix} \checkmark \rightarrow \begin{bmatrix}0\\1\\0\end{bmatrix} \leq \begin{bmatrix}0\\2\\3\end{bmatrix}$$

→ Recursos disponíveis

Novo estado:

$$D = \begin{bmatrix}0\\1\\3\end{bmatrix}$$

$$A = \begin{bmatrix}3 & 1 & 4\\2 & 1 & 0\\2 & 2 & 1\end{bmatrix} \qquad Need = \begin{bmatrix}2 & 0 & 0\\0 & 2 & 1\\3 & 0 & 0\end{bmatrix}$$

Para os processos serem executados, o nº de recursos pedidos tem que ser ≤ aos disponíveis

$$\begin{bmatrix}2\\0\\0\end{bmatrix}\ \begin{bmatrix}0\\2\\1\end{bmatrix}\ \begin{bmatrix}3\\0\\0\end{bmatrix} \not\leq \begin{bmatrix}0\\1\\3\end{bmatrix}$$

↓ ↓ ↓
$P_1$ $P_2$ $P_3$

→ Falso

⇓

deadlock

c)

```
void print_avail_resources(void)
{
  s_mutex.down();
  { printf("%d\n", info.avail_r1);
    printf("%d\n", info.avail_r2);
    printf("%d\n", info.avail_r3);
  }
  s_mutex.up();
}
```