SO 2017

I

a) int main (int argc, char *argv[]) → PAI
{ → PAI
switch (fork()) { → PAI + FILHO
case 0: printf("A\n"); → FILHO
break; → FILHO
default: printf("B\n"); → PAI
wait(NULL); → PAI
printf("C\n"); → PAI
} → PAI + FILHO
return 0; → PAI + FILHO
}

b)
A	B
B	A
C	C

c) Processo — programa em execução. É, portanto, uma entidade ativa. Consiste no programa counter, registos, stack, data section. Pode estar em vários estados e existe na estrutura do SO associado a ele — process control block. Um processo pode ter vários fluxos de instrução — threads.

Thread — Unidade básica do CPU constituída por thread id, PC, register set, stack. As threads partilham código, data e ficheiros, enquanto a criação de processos envolve a cópia destes, mas não a sua partilha (cópia do espaço de endereçamento). Assim, os processos só partilham recursos por memória partilhada e por mensagens. Assim, é + económico mudar de contexto com threads, aumenta a responsividade e a scalability.

d) = 2019

e) Todas as linhas podem estar no estado ready ou running. Chamadas ao sistema — 3, 5, 7 — podem estar em wait. Quando o programa termina, estão em terminated — todas linhas Σ10 — 4, 8 —

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;
}

b) f1
think();
forks[left(f)].down()
↳ S1 = 0

f2
think();
forks[left(f)].down()
↳ S2 = 0

f3
think();
forks[left(f)].down()
↳ S3 = 0

f4
think;
forks[left(f)].down()
↳ S4 = 0

f5
think;
forks[left(f)].down()
↳ S5 = 0

forks[right(f)].down();
↳ bloqueia, pois todos os mutexes já estão a 0.
Ou seja, se todos os filósofos pegarem 1.º no garfo à sua esquerda, qdo algum tentam pegar no da direita, vai ficar bloqueado.

c) Para evitar deadlocks temos que negar 1 das 4 condições:
exclusão mútua, espera com retenção, não libertação, espera circular.
Uma forma de o fazer era adicionar semáforos e negar a espera circular:
Os filósofos 1 a 4 executam o código dado. E o filósofo 5 executa o seguinte:
void filosofo (int f)
while(true){
think;
forks[right(f)].down();
forks[left(f)].down();
eat();
forks[left(f)].up();
forks[right(f)].up();
}
Ou seja, pelo menos um filósofo começa a pegar nos garfos pela direita

III

a) Available = $\begin{bmatrix}0\\2\\3\end{bmatrix}$ R_1 R_2 R_3 Allocation = $\begin{matrix}P_1\\P_2\\P_3\end{matrix}\begin{bmatrix}3&1&4\\2&1&0\\2&1&0\end{bmatrix}$ Need = $\begin{bmatrix}2&0&0\\0&2&1\\3&1&0\end{bmatrix}$

work

Finish = $\begin{bmatrix}0\\0\\0\end{bmatrix}$

1º $\begin{bmatrix}0\\2\\3\end{bmatrix} \leq \begin{bmatrix}0\\2\\1\end{bmatrix}$ ⇒ $\begin{cases}work = \begin{bmatrix}2\\3\\3\end{bmatrix}\\ finish = \begin{bmatrix}0\\1\\0\end{bmatrix}\end{cases}$ P_2

A sequência é segura P_2, P_1, P_3

2º $\begin{bmatrix}2\\3\\3\end{bmatrix} \leq \begin{bmatrix}2\\0\\0\end{bmatrix}$ ⇒ $\begin{cases}work = \begin{bmatrix}5\\4\\7\end{bmatrix}\\ finish = \begin{bmatrix}1\\1\\0\end{bmatrix}\end{cases}$ P_1

3º $\begin{bmatrix}5\\4\\7\end{bmatrix} \leq \begin{bmatrix}3\\1\\0\end{bmatrix}$ ⇒ $\begin{cases}work = \begin{bmatrix}7\\5\\7\end{bmatrix}\\ finish = \begin{bmatrix}1\\1\\1\end{bmatrix}\end{cases}$ P_3

b) Request P_3 = $\begin{bmatrix}0\\1\\0\end{bmatrix}$

$\begin{bmatrix}0\\1\\0\end{bmatrix} \leq \begin{bmatrix}3\\1\\0\end{bmatrix}$ ✓ → $\begin{bmatrix}0\\1\\0\end{bmatrix} \leq \begin{bmatrix}0\\2\\3\end{bmatrix}$ ✓

novo estado:

⇒ Available = $\begin{bmatrix}0\\1\\3\end{bmatrix}$ Allocation = $\begin{bmatrix}3&1&4\\2&1&0\\2&2&0\end{bmatrix}$ Need = $\begin{bmatrix}2&0&0\\0&2&1\\3&0&0\end{bmatrix}$

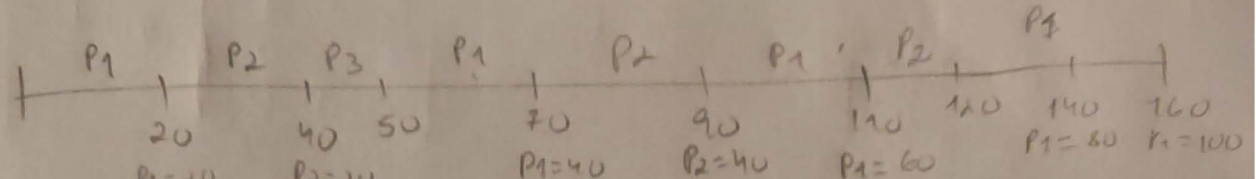
É preciso ver se o novo estado é seguro

$\begin{bmatrix}0\\1\\3\end{bmatrix} \leq$ nenhum processo ⇒ deed lock!

c)
```
void print_avail_resources(void)
{
    s_mutex.down();
    info→ printf("%d\n", info.avail_R1);
    printf("%d\n", info.avail_R2);
    printf("%d\n", info.avail_R3);
    s_mutex.up();
}
```

IV

a) P_1 100
P_2 50
P_3 10



b) $\overline{st}_{espera} = \frac{30 + 20 + 10 + 20 + 30 + 20 + 40}{3} =$

$= \frac{60 + 70 + 40}{3} = \frac{140}{3} = 46,7$

c) RR ⇒ cada processo no CPU por um tempo limite de q.
Um processo nunca espera mais do que $(n-1) \times q$.
É uma versão time sharing e preemptiva do FCFS. $\overline{st}$ é
usualmente longo. Se q ↑ ⇒ ~ FCFS e se q ↓ ⇒ a atenção
às mudanças de contexto. É + complexo do que o FCFS
que basicamente usa um FIFO ~~mas~~ o $\overline{st}$ nesta depende
se o + do custo é executado 1º. É não preemptivo.
exclusão aplicar — RR
cálculo intensivo — FCFS

V = 2019