Resolução Exame 2016-2017

1. Considere o excerto de código seguinte, parte de uma implementação do problema do jantar dos filósofos. Cada filósofo é implementado por uma *thread*, que, após inicialização, executa a função `philosopher`, e é identificado pelo parâmetro `id`, que varia entre 0 e N-1.

```
1   sem_t fork[N];
2
3   #define left(i) i
4   #define right(i) ((i+1)\%N)
5   void philosopher(int id)
6   {
7       while (true)
8       {
9           think();
10          down(fork[left(id)]);
11          down(fork[right(id)]);
12          eat();
13          up(fork[left(id)]);
14          up(fork[right(id)]);
15      }
16  }
```

(a) A possibilidade de ocorrência de *deadlock* pressupõe a satisfação em simultâneo de 4 condições. Quais são?

- Hold and Wait - Recursos sao usados por processos que estao à espera de poder aceder a um processo em uso.
- Circular Wait - Se todos os filosofos adquirirem o garfo da esquerda, ha uma chain em que cada filosofo espera pelo garfo de outro filosofo.
- Non-Preemptive - Recursos em uso nao conseguem ser libertados por outros processos.
- Mutual exclusion - apenas um processo pode usar um recurso at a time. Se outro processo pede acesso, deve esperar ate que o recurso seja libertado.

(b) A implementação do jantar dos filósofos apresentada pode conduzir a *deadlock*. Mostre que as 4 condições anteriores são satisfeitas.

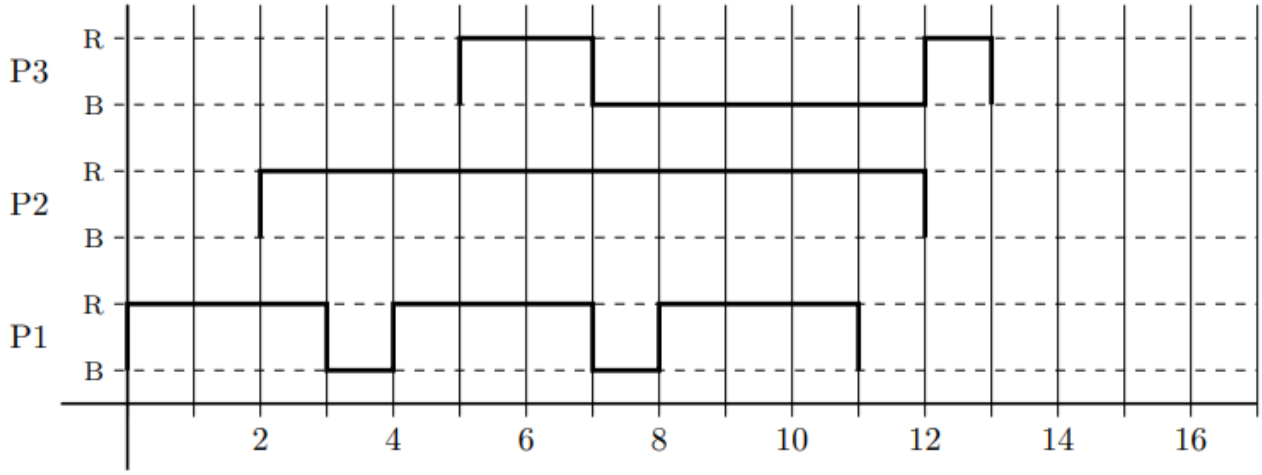
- Hold and Wait - Filosofos pegam no garfo da esquerda primeiro e depois pegam no da direita. Se o da direita estiver em uso, vao estar em posse do garfo da esquerda enquanto esperam que o da direita fique disponivel.
- Circular Wait - Se todos os filosofos adquirirem o garfo da esquerda, ha uma chain em que cada filosofo espera pelo garfo de outro filosofo.
- Non-Preemptive - Apenas o filosofo consegue colocar na mesa garfos que esteja a usar.

- Mutual exclusion - Apenas um filosofo pode usar um garfo at a time. Se outro filosofo quer usar um garfo em uso, vai ter que esperar ate que seja posto na mesa. Garfos nao sao partilhaveis., por causa do semaforo.

(c) Altere o código dado de modo a evitar a ocorrência de dealock. Explique a sua solução, indicando a(s) condição(ões) que negou.

Para evitar deadlock, temos que arranjar uma soluçao para pelo menos uma das condiçoes anteriores.
Uma soluçao seria apenas levantar o garfo esquerdo se o direito tambem tiver disponivel. Estariamos a negar o hold and wait.
Outra soluçao seria libertar o garfo da esquerda caso o da direita nao estivesse disponivel. Tambem negaria o hold and wait.
Outra soluçao: Cada garfo tem um id, igual ao id do filosofo à sua esquerda. Cada filosofo adquire o garfo com menor id. This way, philosophers 0 to N − 2 acquire first the left fork, while philosopher N − 1 acquires first the right one. Negaria o circular wait.

meter `if left and right fork available {sem_down}...` *Nao sei se ta bem*



(a) Distinga processos CPU-intensivos e processos I/O-intensivos. Em que categoria coloca os processos P1, P2 e P3? Justifique a sua resposta.

Processos CPU-intensivos sao processos que têm poucos CPU-bursts longos.
Processos I/O-intensivos sao processos que têm muitos CPU-bursts pequenos.
Processos CPU-intensivos precisam mais frequentemente do processador e processos I/O-intensivos precisam mais frequentemente que ocorram eventos externos.

P1 - CPU-intensivo
P2 - CPU-intensivo, mas mais que o P1
P3 - I/O-intensivo

Tanto P1 como P2 gastam mais unidades de tempo em açoes de RUN, enquanto que no P3 acontence o contrario.

(b) Compare as políticas de escalonamento do processador designadas por FCFS (First Come First Served) e Round Robin.

- FCFS ou First Come First Served escolhe para dispatch o processo mais velho na FIFO. Não existe transições de time-out.
- Round Robin tem transições de time-out, pelo que é necessário atribuir um time quantum a cada processo dispatched.

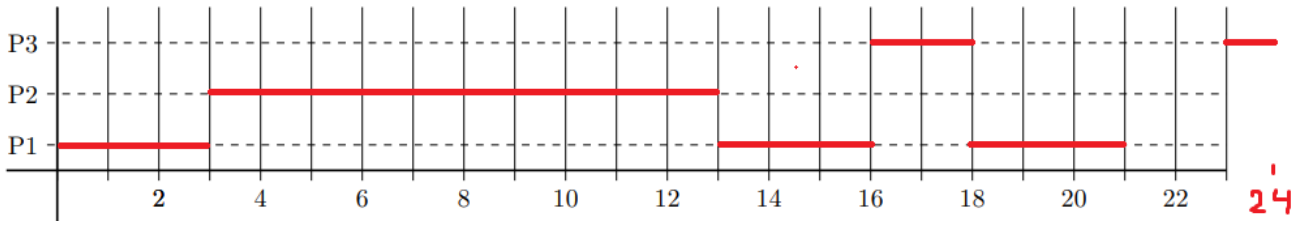
(c) Distinga multiprocessamento e multiprogramação.

Multiprocessamento é a execução de vários processos ao mesmo tempo, uma vez que há mais do que um processador no sistema. Podem ser executados N processos simultaneamente, sendo N o número de processadores reais no sistema. - PARALELISMO

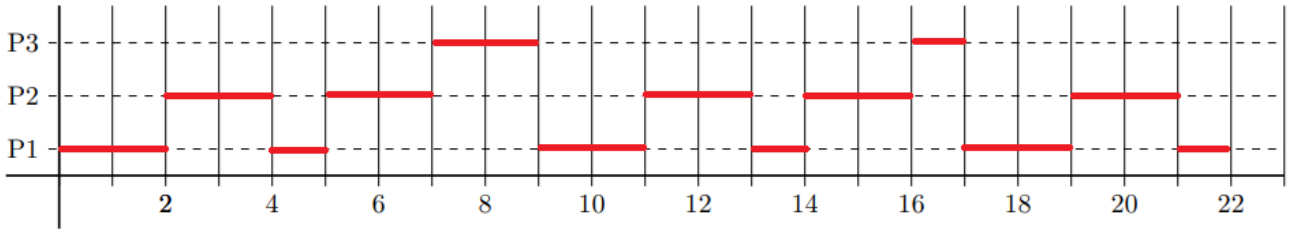
Multiprogramação é a ilusão de multiprocessamento. Apenas existe 1 processador e o scheduler faz dispatches e preempts frequentes para criar a ilusão (a seres humanos) que os processos estão a correr simultaneamente. - CONCORRÊNCIA

(d) Considere que os 3 processos representados acima correm num ambiente monoprocessador. Usando os gráficos abaixo, trace os diagramas temporais de escalonamento do processador pelos processos P1, P2 e P3, considerando as políticas de escalonamento FCFS e Round Robin, esta com um *time quantum* (*time slot* atribuído a cada processo) de 2.

FCFS



Round Robin com time quantum de 2



3. Considere o programa apresentado a seguir, onde `delay()` é uma função que gera um atraso com tempo aleatório em *busy waiting* (ou seja, não bloqueante).

```
1  int main(void)
2  {
3      printf("msg 0\n");
4      int pid = fork();
5      switch (pid)
6      {
7          case 0:
8              delay();
9              printf("msg 1\n");
10             printf("msg 2\n");
11             break;
12         default:
13             delay();
14             printf("msg 3\n");
15             wait(NULL);
16             printf("msg 4\n");
17     }
18     return 0;
19 }
```

(a) Assumindo que o `fork` não falha, que linhas do programa anterior são executadas no processo pai e no processo filho? Justifique sucinta e adequadamente a sua resposta.

A função `fork()` devolve valor 0 ao filho e um valor inteiro positivo ao pai. Negativo se falhar.
Logo o filho executa as linhas [5, 11] e 18 e o pai executa todas menos as [7, 11].

(b) Considerando que a execução de um `printf` é atómica, além da saída

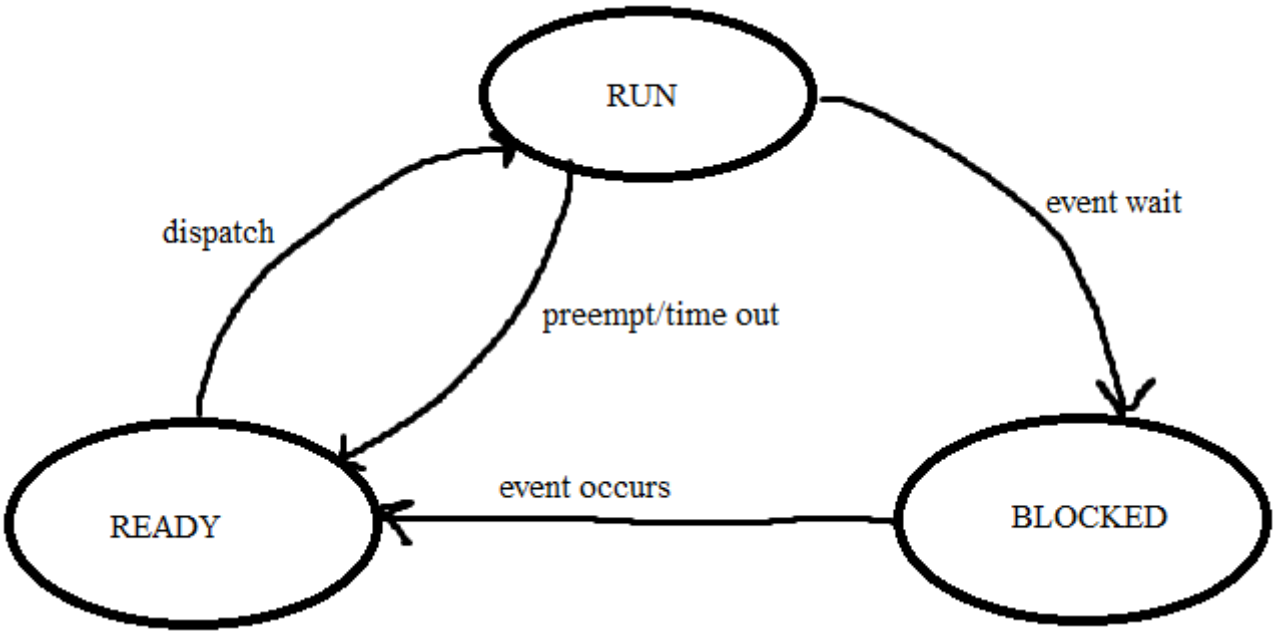
```
msg 0
msg 1
msg 2
msg 3
msg 4
```

apresente outras possíveis saídas (em termos de *standard output*) que podem resultar da execução do programa anterior. Justifique sucinta e adequadamente a sua resposta.

O wait(NULL) espera ate que um processo filho dê exit. Por isso, meio que sincroniza.
0-3-1-2-4

Sempre que o 3 é printado, o 4 so pode ser printado no fim de executar o processo filho (se existente). 0-x-x-x-4 sempre.

(c) O escalonador de processador de baixo nível típico possui 3 estados, normalmente designados de `RUN`, `READY_TO_RUN` e `BLOCKED`. Trace o diagrama de estados para um escalonador de baixo nível, considerando os estados anteriores. Para cada transição considerada, explique o seu papel e quando é que ocorre.



- Preempt significa que um processo de maior prioridade que o processo atualmente a correr entrou em estado READY e portanto, toma posse do processador, tendo o outro que abandonar.
- Time out significa que o processo a correr esgotou o seu time quantum.
- Dispatch mete um processo a correr.
- Event wait significa que um processo está à espera que aconteça um evento externo.
- Event occurs significa que ocorreu o evento externo.

(d) Considerando que a execução do programa resulta na saída apresentada na alínea (b) e que os `printf` nunca bloqueam o processo, o processo pai pode passar pelo estado `BLOCKED`? Justifique sucinta e adequadamente a sua resposta.

Na situação em b), não. (Seria preempted acho eu, mas blocked nao) O processo pai chama o wait que bloqueia o seu processo ate algum filho terminar processo, mas no caso do b), todos os filhos já têm o processo terminado quando chegamos à wait(NULL). Logo, nunca fica blocked à espera deste evento.

4. Considere um sistema de memória virtual paginada, onde a cada processo é atribuído um máximo de 10 páginas e em que a memória principal do sistema tem 5 *frames*. Considere ainda que um processo (único) executou a seguinte sequência de referências, em termos de páginas de memória acedidas: 1, 2, 3, 4, 5, 3, 4, 9, 6, 7, 1, 7, 8, 1, 7, 8, 9, 5, 4, 5, 2.

(a) A tabela seguinte representa, ao longo do tempo, as páginas residentes nas *frames* de memória. Complete-a considerando que o algoritmo de substituição de páginas utilizado é o FIFO. Preencha apenas as células da tabela quando há mudança de página.

- Cada processo tem max de 10 paginas
- Mem principal tem 5 frames

	1	2	3	4	5	3	4	9	6	7	1	7	8	1	7	8	9	5	4	5	2
F5					5								8								
F4				4							1										
F3			3							7											2
F2		2							6										4		
F1	1							9										5			

O processo quer aceder a cenas, entao precisa que essas cenas estejam na memoria. Cada frame "aguenta" com uma pagina. O stor preencheu as primeiras 5 vezes. Agora temos as frames todas ocupadas com as paginas de 1 a 5. 6a coluna queremos aceder à pagina 3. Já a tinhamos metido em memoria antes, por cima nao é preciso fazer nada, 7a igual. 8a coluna queremos aceder à pagina 9, e nao se encontra em memoria. Logo temos que fazer uma substituiçao. Com metodo FIFO substitui-se a Frame que está há mais tempo ocupada pela mesma pagina (mais velha ig), portanto a F1. Usa-se esta logica para o resto.

(b) A tabela seguinte representa, ao longo do tempo, as páginas residentes nas *frames* de memória. Complete-a considerando que o algoritmo de substituição de páginas utilizado é o LRU (Least Recently Used). Preencha apenas as células da tabela quando há mudança de página.

Fica

	1	2	3	4	5	3	4	9	6	7	1	7	8	1	7	8	9	5	4	5	2
F5					5					7											2
F4				4									8								
F3			3								1								4		
F2		2							6									5			
F1	1							9													

Nao sei se tá bem mas acho que sim

(c) O algoritmo LRU tem um custo de implementação elevado e é pouco eficiente. Uma aproximação menos exigente e relativamente eficiente é o algoritmo NRU (Not Recently Used). Descreva o princípio de funcionamento deste algoritmo.

R:

Se virmos as coisas pelo lado oposto, podemos aproximarmo-nos de uma solução usando uma estratégia **NRU** (do inglês *Not Recently Used*), onde são apenas usados os bits `Ref` e `Mod` que são processados tipicamente por uma MMU convencional. Periodicamente, o sistema de operação percorre a lista dos frames ocupados e coloca a zero o bit `Ref`. Assim, quando ocorre uma *page fault*, os frames da lista de frames ocupados enquadram-se numa das classes da tabela da Figura 4.11.

classe	Ref	Mod
0	0	0
1	0	1
2	1	0
3	1	1

A seleção da página a substituir será então feita entre aquelas pertencentes à classe de ordem mais baixa existente atualmente na lista dos frames ocupados.

---

5. Considere o sistema de ficheiros `sofsxx`, semelhante aos `sofs15` e `sofs16`, com blocos de tamanho 512 bytes (2^9) e *clusters* de 4 blocos.

(a) Sabendo que as referências aos clusters têm 32 bits e desprezando os blocos usados pelo superbloco e pela tabela de nós-i (*inodes*), calcule o tamanho máximo em bytes que um disco formatável em `sofsxx` pode ter? Apresente os cálculos necessários para justificar a sua resposta.

REFS = 32 bits

BLK_size = 2^9 bytes

Cluster size = 4 BLK

Num Clusters = 2^{32}

Num Blocos = 4 * 2^{32} = 2^{34}

Tamanho disco = 2^9 * 2^{34} = 2^{43}, ou seja, 8 TBytes.

(b) Considerando que um nó-i do `sofsxx` possui 5 referências diretas, 1 indireta, 1 duplamente indireta e 1 triplamente indireta, calcule o tamanho máximo em bytes que um ficheiro pode ter? Apresente os cálculos necessários para justificar a sua resposta.

$$Filesize = (5 * 4 + \frac{2^9}{4} * 4 + \frac{2^9 * 2^9}{4} * 4 + \frac{2^9 * 2^9 * 2^9}{4} * 4) * 2^9$$

fica = 2^9 * (20 + 2^9 + 2^{18} + 2^{27})

= 2^{38}

(c) O `sofsxx` suporta *hard links* e *soft links* (atalhos). Explique a diferença entre ambos.

Hard link aponta para o mesmo nó-i do ficheiro para o qual é um atalho.
Soft link aponta para outro nó-i e aponta para o caminho onde está o ficheiro.