FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mestrado em Engenharia Informática e Computação

SISTEMAS OPERATIVOS (EIC0027) – 2014/2015 - 2º semestre

Exame da Época Normal 16/Junho/2015

Duração: 2 horas
(com consulta)

NOME DA(O) ESTUDANTE: _____ **Nº:** _____

1.
Os sistemas operativos modernos permitem tirar partido dos períodos em que um processo não necessita do processador para executar outro processo.

a) [0.6] Que períodos são esses? Que nome se dá a esta técnica?

b) [0.6] Que informação deve ser guardada para retomar a execução de um processo?

c) [0.6] Como é que o sistema operativo garante que um programa que executa um ciclo infinito (por exemplo, **while (1) x++;**) não toma indefinidamente conta do processador?

2.

a) [0.5] Para usar um semáforo como se fosse um "mutex", qual deve ser o valor inicial do semáforo? Justifique.

b) [0.8] Explique por que não faz sentido inicializar um semáforo com um valor negativo.

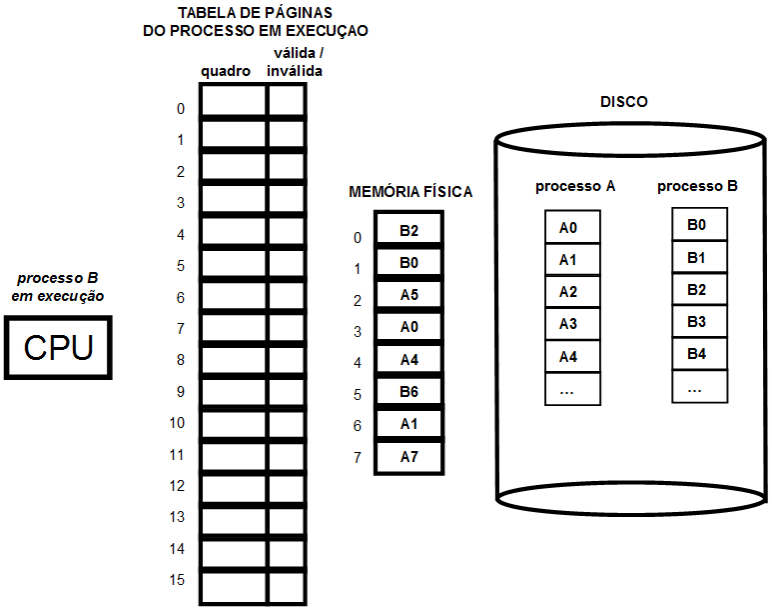
c) [1.2] Num sistema de gestão bancária correm vários processos que necessitam de transferir verbas entre contas de dois clientes.

c1) As transferências constituem secções críticas. Explique porquê.

c2) Mostre, através de um exemplo, que se não houver cuidado na escrita do código desses processos podem ocorrer situações de *deadlock*, quando se pretende minimizar o tempo de espera de cada processo para aceder às contas dos clientes. Justifique a resposta.

3.

Considere um sistema de gestão de memória, hipotético, com paginação "a pedido"/"por exigência" (*demand paging*), em que as páginas têm **1024** bytes. A figura seguinte apresenta um esquema do estado do sistema, quando estavam carregados para execução apenas dois processos, **A** e **B**; **Ai** e **Bi** representam as páginas dos processos.



a) [0.5] Neste sistema, qual o tamanho da memória virtual "vista" por cada processo? Justifique.

b) [0.5] Quantos bits tem um endereço lógico? E um endereço físico? Justifique.

c) [0.5] Preencha, na figura, a tabela de páginas do processo B.

(a preencher na figura da página anterior)

d) [1.0] Sabendo que as páginas foram, recentemente, acedidas pela seguinte ordem, **A0-B0-A1-A7-B2-A5-B0-A4-A0-B2-B6-A0**, que a substituição de páginas tem alcance global e que o algoritmo de substituição de páginas é o *LRU*, qual o endereço físico correspondente ao endereço lógico $\mathbf{2000_{10}}$? Justifique brevemente a resposta.

4. [1.2] Em Linux, explique a relação entre os conceitos de ficheiro, diretório e *i-node*.

NOME DA(O) ESTUDANTE: _____ **Nº:** _____

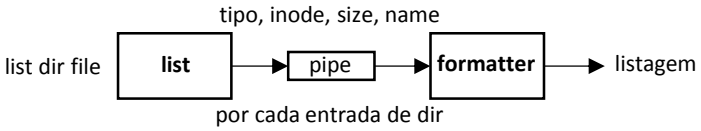
***Nota:* no código solicitado nas perguntas 5 e 6, pode omitir as diretivas de inclusão e, em geral, os testes de erro nas chamadas ao sistema**

5.

Pretende-se listar os nomes dos subdiretórios e ficheiros de um diretório que é diretamente descendente do diretório *home* do utilizador. Essa listagem deve ter uma letra inicial, **f** ou **d**, seguida de um espaço e do nome do ficheiro ou diretório, em linhas consecutivas. Apenas para um dos ficheiros (indicado especialmente) são também listados, na mesma linha e separados por espaço, o número do *inode* e o tamanho em blocos. Apresenta-se ao lado um exemplo, em que **cde** é o ficheiro especial.

```
...
d sub
f abc
f cde 10143 5
f xyz
...
```

Para o efeito, utilizam-se 2 programas, comunicando entre si por um *pipe* (figura ao lado).



O programa **list**, tem dois argumentos, onde o primeiro deverá conter o nome do diretório a listar (relativo ao diretório *home*) e o segundo o nome de um ficheiro que, se existir no diretório, deverá ser listado no modo especial. Exemplo de invocação em que o diretório é **dir1** e o ficheiro especial é **cde**: **list dir1 cde**.

O programa **formatter** recebe na sua entrada *standard* uma sequência de conjuntos de 4 valores, até ao 'fim de ficheiro', em que cada conjunto é constituído por três inteiros e uma *string* (incluindo o 0 terminador). Os três inteiros indicam o tipo (1 indica ficheiro, 2 diretório e 100 o ficheiro especial), número do *inode*, e tamanho em blocos. A *string* contém o nome. O programa **formatter** produz na saída *standard* a listagem especificada, usando apenas para as linhas de tipo 100 os valores do *inode* e tamanho.

a) [1.2] Explique a diferença entre diretório *home* e diretório corrente. Indique como poderia obter o diretório *home* do utilizador que executa o programa (não escreva código).

b) [1.2] Escreva, para o programa **list**, a secção de código que declara as variáveis necessárias, verifica se o diretório a listar (o primeiro argumento) existe, e percorre, num ciclo, todas as suas entradas, copiando o nome de cada uma delas para a variável **char *name**. Assuma que o nome do diretório *home* é apontado por **char *home**.

c) [1.2] Escreva a secção de código do programa **list** que declara as variáveis necessárias e cria um *pipe* que o liga à entrada *standard* de **formatter**, lançado em execução nessa secção de código. O programa **formatter** encontra-se no diretório *home*.

d) [1.2] No ciclo que escreveu na alínea b) acrescente agora o envio dos dados a **formatter** de modo a produzir a listagem esperada. Indique apenas as alterações ao código de b).

e) [1.2] Indique o que acontece aos dois programas, **list** e **formatter**, se **list** receber o sinal **SIGKILL** a meio do ciclo que está a enviar os dados a **formatter**.

NOME DA(O) ESTUDANTE: _____ **Nº:** _____

6.

Um processo cria 2 *threads* para preencherem e processarem um *buffer* de dados, garantindo sempre que o processamento de uma posição do *buffer* ocorre após o seu preenchimento. O *buffer* tem 10 posições, sendo cada uma constituída por uma determinada estrutura de dados (**struct Data**). O preenchimento faz-se chamando uma função **void fill(struct Data * d)** e o processamento chamando **void process(struct Data d)**. Os ciclos seguintes constituem o código significativo dos dois *threads*:

```
Thread 1:                          Thread 2:
     ...                                ...
     for (k=0; k<10; k++) {             for (k=0; k<10; k++) {
          fill(&buffer[k]);                  process(buffer[k]);
     ...                                ...
```

a) [1.2] Indique duas formas dos dois *threads* terem acesso ao mesmo *array*, **buffer[ ]**. Para cada uma das formas, faça a declaração de variáveis e escreva as linhas de código relevantes do programa global, supondo **struct Data** já definido.

b) [1.2] Antes de lançar os dois *threads*, a função **main()** do processo deve criar e inicializar um semáforo, como variável global, com o valor **SEM_INI** definido numa diretiva **#define**. Declare e escreva as linhas de código para a criação e inicialização adequada desse semáforo.

c) [1.2] Para garantir que a chamada a **process()** na posição **k** do *buffer* se faz sempre depois da chamada a **fill()** na mesma posição **k** no outro *thread*, é necessário colocar, dentro dos ciclos anteriormente apresentados, chamadas a **sem_wait()** e a **sem_post()**. Reescreva esses ciclos com essas chamadas e indique o valor de inicialização **SEM_INI** de modo a garantir, de forma eficiente, a condição anterior.

d) [1.2] Diga, justificando, se seria possível ou conveniente substituir o semáforo por um *mutex*.

e) [1.2] Sabendo que os tempos de execução das chamadas a **fill()** e **process()** são respetivamente **tf** e **tp** e que todas as outras instruções dos *threads* demoram um tempo desprezável (exceto os bloqueamentos em **sem_wait()**), apresente os cálculos do tempo total mínimo de execução dos 2 *threads*, nos casos em que **tf > tp** e **tf < tp**, sabendo que há pelo menos 2 processadores no sistema.

FIM