FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mestrado em Engenharia Informática e Computação

SISTEMAS OPERATIVOS – 2009/2010 - 2º semestre

Exame da Época Normal 6/Julho/2010

PARTE A – sem consulta
Duração: 60 minutos

NOME DO ESTUDANTE: _____ **Nº:** $\mathcal{E}I$_____

PARTE A-1 [7 valores]**: Perguntas com resposta de escolha múltipla**

- **Para cada pergunta só há uma resposta correcta; assinale-a com um X.**
- **Para anular uma resposta que assinalou faça um O em volta do X, para voltar a assinalar essa resposta desenhe o X à esquerda da caixa.**
- **Cada resposta correcta vale 0.5 valores; cada resposta errada vale –0.5/3 valores; as respostas não-assinaladas, ininteligíveis ou ambíguas valem zero valores.**

1.

Multiprogramação é

□ - o desenvolvimento de um programa, em simultâneo, por vários programadores.

□ - a execução interlaçada de processos, num único processador (*CPU*), tendo em vista a maximização da sua utilização.

□ - o uso de dois ou mais processadores para executar, em paralelo, partes de um programa.

2.

Os processadores (*CPUs*) modernos suportam, pelo menos, dois modos de operação, vulgarmente designados por modo utilizador e modo supervisor (/sistema/privilegiado/*kernel*). O objectivo deste duplo modo de operação é

□ - impedir que os processos dos utilizadores possam executar directamente certas instruções-máquina, as quais só podem ser executas em modo supervisor, através de "chamadas ao sistema" feitas por esses processos.

□ - permitir que o código do sistema operativo execute mais rapidamente, comutando o processador para o modo supervisor.

□ - permitir que os processos dos utilizadores que sejam considerados prioritários executem mais rapidamente, sendo para tal necessário comutar o processador para modo supervisor

3.

Durante a sua "vida", um processo vai passando por vários estados. De acordo com o modelo de 5 estados, apresentado nas aulas, um processo diz-se que está no estado "bloqueado"

□ - se estiver em "deadlock".

□ - se estiver à espera de um evento ou de um recurso.

□ - se estiver na fila de processos prontos, à espera de usar o processador.

4.

Qual das seguintes afirmações é falsa ?

□ - A comunicação entre *threads* de um mesmo processo é facilitada pelo facto de todas as *threads* poderem aceder às variáveis globais do processo.

□ - Num programa *multithreaded*, cada *thread* tem um *program counter* que lhe está associado.

□ - O tempo de comutação entre *threads* de um mesmo processo é maior que o tempo de comutação entre processos.

5.

Qual das seguintes afirmações sobre o algoritmo *Round-Robin* de escalonamento do processador é falsa ?

□ - Cada processo pode usar o processador até esgotar uma fatia de tempo que lhe foi atribuída.

□ - É, no essencial, uma versão preemptiva do algoritmo *First-Come First-Served.*

□ - Favorece os processos que fazem uso intensivo de operações de entrada/saída (*I/O-bound*).

6.

No contexto da sincronização de processos, uma secção crítica é um bloco de código que

□ - acede a variáveis ou recursos comuns a mais do que um processo/*thread*, variáveis e recursos esses que têm de ser usados em exclusão mútua.

□ - tem obrigatoriamente de ser executado em modo supervisor.

□ - tem de ser executado num tempo muito curto, sob pena de o sistema de computação deixar de funcionar correctamente.

7.

Qual das seguintes afirmações acerca dos semáforos é falsa ?

□ - No essencial, um semáforo é um contador sobre o qual podem ser executadas duas operações básicas, wait() e signal().

□ - O contador do semáforo pode ser inicializado com qualquer valor, positivo, nulo ou negativo.

□ - A operação wait() decrementa o contador do semáforo e a operação signal() incrementa o contador.

8.

Considere os seguintes excertos do código de dois processos, P1 e P2, que executam concorrentemente.
S1 e S2 são dois semáforos binários (ou *mutexes*) inicializados com o valor 1.
Qual das seguintes afirmações é verdadeira ?
Ao executar ambos os processos concorrentemente, eles

□ - podem entrar em "deadlock".

□ - nunca entrarão em "deadlock".

□ - entrarão sempre em "deadlock".

processo P1	processo P2
…	…
wait(S1)	**wait(S2)**
wait(S2)	**wait(S1)**
…	…
signal(S2)	**signal(S1)**
signal(S1)	**signal(S2)**
…	…

9.

Em Unix/Linux, para obter uma lista dos processos em execução usa-se o comando

□ - ls

□ - cp

□ - ps

10.

Em Unix/Linux, qual das chamadas seguintes não devolve um descritor de ficheiro (*file descriptor*) ?

□ - open()

□ - mkfifo()

□ - pipe()

11.

Em Unix/Linux, a chamada ao sistema wait() (*não confundir com a operação de espera sobre um semáforo, genericamente designada por wait()*) permite que

□ - um processo espere que qualquer um dos seus processos-filhos termine.

□ - qualquer processo-filho espere que o seu processo-pai termine.

□ - qualquer processo espere que qualquer outro termine.

12.

Em Unix/Linux, um sinal é um mecanismo muito básico de comunicação entre processos. Qual das seguintes afirmações é verdadeira, relativamente ao uso de sinais:

□ - A única funcionalidade da chamada ao sistema kill() é enviar o sinal SIGKILL a um processo.

□ - A chamada ao sistema signal() é usada para enviar um sinal.

□ - A chamada ao sistema signal() é usada para instalar o *handler* de um sinal.

13.

Em Unix/Linux, para que dois processos, "pai" e "filho", possam comunicar entre si através de um *pipe* sem nome (*unnamed pipe*):

□ - o *pipe* pode ser criado antes ou depois de ser criado o processo-filho.

□ - o *pipe* tem de ser criado depois de ser criado o processo-filho.

□ - o *pipe* tem de ser criado, antes de ser criado o processo-filho.

14.

Em Unix/Linux, uma das vantagens dos *FIFO*s (ou *named pipes*) relativamente aos *unnamed pipes* é que

□ - no *FIFO*, a informação é sempre lida pela mesma ordem em que foi escrita.

□ - podem ser usados para transferir informação entre quaisquer dois processos que estejam em execução num mesmo computador.

□ - com um único *FIFO* é possível a comunicação bidireccional.

==========

PARTE A-2 [*4 valores*]**: Perguntas com resposta livre**

1.

Considere o extracto apresentado ao lado, em código *C-like*, de um programa que implementa um processo consumidor no "problema dos produtores-consumidores", usando um *buffer* circular, de capacidade **N**.

sem_wait() e **sem_signal()** representam as operações básicas sobre semáforos.

I é uma variável partilhada entre todos os consumidores.

```
programa consumidor
...
do
   sem_wait(mayConsume);
   sem_wait(m);
   item = buffer[I];  //extrai item
   I = (I + 1) % N;
   sem_signal(m);
   sem_signal(mayProduce);
   ...              // processa item
while ...;
...
```

a) [*1 valor*] Qual deverá ser o valor inicial dos semáforos **m**, **mayConsume** e **mayProduce** ? Justifique a resposta.

b) [*1 valor*] Escreva o programa produtor, em código *C-like*.

```
programa produtor
...
do









while ...;
...
```

2.

Considere o seguinte programa através do qual se pretendia criar 7 *threads*, cada uma dos quais deveria escrever uma saudação contendo o seu número de identificação (de 0 a 6). O resultado de uma execução deste programa foi o indicado na coluna da direita.

```
#include ...

void *PrintHello(void *threadnum) {
   printf("Hello from thread no. %d!\n", *(int *) threadnum);
   pthread_exit(NULL);
}

int main() {
   int t;  pthread_t tid[7];
   for(t=0; t< 7; t++){
      printf("Creating thread %d\n", t);
      pthread_create(&tid[t], NULL, PrintHello, (void *)&t);
   }
   pthread_exit(0);
}
```

```
Creating thread 0
Creating thread 1
Creating thread 2
Hello from thread no. 2!
Hello from thread no. 2!
Hello from thread no. 2!
Creating thread 3
Creating thread 4
Creating thread 5
Hello from thread no. 5!
Creating thread 6
Hello from thread no. 7!
Hello from thread no. 5!
Hello from thread no. 7!
```

a) [*1 valor*] O resultado não é o esperado! Explique, sucintamente, o que aconteceu.

b) [*1 valor*] Indique a correcção necessária para que cada *thread* escreva o seu número de identificação correctamente.

FIM DA PARTE A

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mestrado em Engenharia Informática e Computação

SISTEMAS OPERATIVOS – 2009/2010 - 2º semestre

Exame da Época Normal 6/Julho/2010

PARTE B – com consulta de livros ou apontamentos
Duração: 60 minutos

1. [*3 valores*]

Escreva um programa (**`rlslc`** - *recursive ls line counter*) que, recorrendo ao comando de Unix/Linux **`ls`** e usando as primitivas **`pipe()`**, **`fork()`**, **`dup2`**( ) e **`exec`***xx*(), mostra no écrã uma listagem recursiva de um directório e, no final, indica o número de linhas de texto obtidas. A listagem a obter deve ser a mesma que é dada pelo comando **`ls –lasR dir`**, onde **`dir`** representa o nome do directório. O directório **`dir`** deve ser passado a **`rlslc`** como argumento da linha de comando (ex: **`./rlslc /home/users/username/SOPE`**).
Assuma que dispõe de uma função **`int readline(int fd, char * line)`** que lê uma linha de texto, de um ficheiro cujo descritor é **`fd`** para o endereço **`line,`** e devolve o número de *bytes* lidos ou zero quando o texto se esgotar. Cada linha tem no máximo **`LINE_LEN`** caracteres.

2. [*6 valores*]

Considere o desenvolvimento de um sistema de monitorização de recursos informáticos, para o qual já se dispõe de duas funções:
- a funcão **`char** get_computer_names()`** que devolve um vector, *null terminated* (terminado por um apontador nulo), de nomes de computadores a monitorizar;
- a função **`void get_and_print_computer_resources(char* name)`** que tem como parâmetro o nome de um computador, obtém a lista dos seus recursos e apresenta-a na saída *standard* (**`stdout`**).

a) [*3 valores*] Com base nestas duas funções, pretende-se desenvolver o sistema de monitorização, modelizado como um sistema produtor-consumidor constituído por duas *threads* e usando um *buffer* circular, de *strings*, de capacidade **`N`**: **`char* buffer[N]`**. O acesso a este *buffer* deve ser devidamente sincronizado por semáforos.
A *thread* **`producer`** deve, em ciclo infinito, obter um vector de nomes de computadores a monitorizar, invocando **`get_computer_names()`**, e inserir os nomes, um por um, no *buffer* circular.
A *thread* **`consumer`** deve, também em ciclo infinito, extrair do *buffer* circular o nome de um computador e invocar a função **`get_and_print_computer_resources()`**, passando o nome desse computador como parâmetro.
Escreva o código do programa que implementa este sistema. NOTA: na solução que apresentar deve usar os semáforos estritamente necessários, tendo em conta que existe apenas 1 produtor e 1 consumidor.

b) [*1 valor*] Por uma questão de eficiência, a *thread* **`consumer`** deve paralelizar a execução da função **`get_and_print_computer_resources()`**. Nesse sentido, proceda às seguintes alterações:

i. Implemente o código de uma *thread* **`show_resources`**, muito simples, que se limita a invocar a função **`get_and_print_computer_resources()`** para um único computador, cujo nome é recebido como parâmetro na criação da *thread* e, de seguida, termina.

ii. Altere o código de **`consumer`** para, em cada ciclo, em vez de extrair um só computador do *buffer* circular e chamar directamente a função **`get_and_print_computer_resources()`**, extrair 10 computadores e, de seguida, proceder à criação de 10 *threads* **`show_resources`**, uma por cada computador, esperando que estas terminem antes de iniciar o próximo ciclo.

c) [*1 valor*] Altere a *thread* **`show_resources()`** para, antes e depois de invocar a função **`get_and_print_computer_resources()`**, escrever na saída *standard* (**`stdout`**) uma mensagem identificativa do número do computador que vai ser (ou que acabou de ser) monitorizado. Exemplo:

```
"Vai ser obtida informação do computador nº 1"
…… // aqui fica a informação dada pela rotina get_and_print_computer_resources()
"Foi obtida a informação do computador nº1"
```

Este número do computador é um valor sequencial que deverá ser recebido (também) como parâmetro na criação da *thread* **`show_resources`**(). Altere também a *thread* **`consumer`** para incrementar esse número do computador a ser monitorizado e passá-lo, como parâmetro, para a *thread* **`show resources`**(), em adição ao parâmetro 'nome do computador' que já era passado na solução mais simplificada da alínea **b.ii)**.

d) [*1 valor*] Introduza as alterações necessárias ao código da *thread* **`consumer`** da alínea **c)** para permitir a execução concorrente de múltiplas instâncias desta *thread*.

FIM