U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA E COMPUTAÇÃO | 2º ANO
EIC0027| SISTEMAS OPERATIVOS | 2018/2019 – 2º SEMESTRE

EXAME - ÉPOCA NORMAL | 2019-06-11

Com consulta. Duração: 2h30m

**Nome do estudante:** _____ **Número:** _____

**1.** [5.0]

| Indique quais das seguintes afirmações são verdadeiras e quais são falsas, assinalando respetivamente com **V** ou **F.**<br>**NOTA** - A pontuação nesta pergunta será dada pela fórmula: máximo ( 0, (nº_respostas_certas – nº_respostas_erradas)*0.25 ) | **V / F** |
|---|---|
| A melhor forma que um programador tem de prevenir a existência de *deadlocks* nos seus processos é impedir a ocorrência de "espera circular". | |
| O código seguinte escreve 8 vezes a palavra `Hello`: `for (int i=1; i<=3; i++) { fork(); } printf("Hello ");` | |
| As chamadas exec fazem simultaneamente duas coisas: criam um novo processo e carregam o código de um programa a executar nesse processo. | |
| O escalonamento do tipo *Multilevel Feedback Queue* atribui um *quantum* (fatia de tempo) menor a processos com maior prioridade. | |
| Num sistema operativo, a implementação de preempção baseia-se no uso de um temporizador. | |
| Uma *race condition* é o que acontece quando vários processos estão a tentar aceder a uma secção crítica mas só um consegue entrar. | |
| Em diferentes processos, o mesmo endereço lógico corresponde a diferentes endereços físicos. | |
| O princípio da localidade de referência é a base da técnica de gestão de memória virtual. | |
| No contexto dos sistemas operativos, multiprogramação é sinónimo de multiprocessamento. | |
| Em Linux, se a chamada `readdir()` for invocada após a leitura da última entrada de um diretório, será lida automaticamente a primeira entrada do diretório. | |
| Em Linux, a chamada `execlp("ls", "ls","*.c",NULL)` cria um processo que mostrará na saída standard todas as entradas do diretório corrente cujo nome tem a extensão `".c"`. | |
| O comando `ps`, da *shell* (interpretador de comandos) de Linux, permite obter os sinais pendentes para um processo. | |
| Em Linux, um *i-node* contém, entre outros dados, o nome do ficheiro ou o nome do diretório que lhe está associado. | |
| Em Linux, após uma chamada `fork()`, o processo filho pode aceder aos ficheiros que estiverem abertos no processo pai, sem os reabrir. | |
| Enquanto um processo permanecer no estado *zombie*, o seu *Process Control Block* não pode ser destruído. | |
| Num sistema em que a gestão de memória é baseada em paginação simples, os endereços lógicos têm 32 bits e as páginas tem 16 KBytes, a tabela de páginas terá $2^{18}$ elementos. | |
| Uma das limitações dos sinais como mecanismo de comunicação entre processos é que não há garantia que todos os sinais enviados a um processo sejam recebidos por este. | |
| Em Linux, um *thread* é implementado através de uma função cujo protótipo tem de ser do tipo: `void* thrFunc(void*)` . | |
| Em Linux, um *mutex* só pode ser usado por *threads* de um mesmo processo. | |
| Uma secção crítica é um pedaço de código que tem de ser executado rapidamente e sem interrupções. | |

**Nome do estudante:** _____ **Número:** _____

**2.** [2.0]

**a)** [0.5] Explique por que não faz sentido inicializar um semáforo com um valor negativo.

**b)** [1.5] Pretende-se implementar 2 processos, **P1** e **P2**, que executem as suas tarefas de forma alternada, sendo **P1** o primeiro a executar (considerando o exemplo ao lado, a execução consistiria na sequência: **doWork1()**-**doWork2()**-**doWork1()**-...). Estes processos partilham com outro processo **P0**, criado previamente, uma variável booleana (**goOn**), criada e controlada por **P0**, que indica se **P1** e **P2** devem ou não continuar o seu trabalho; esta variável pode ser atualizada a qualquer momento por **P0**. Recorrendo a semáforos, indique como implementaria a parte do código de **P1** e **P2** que garante a sincronização de todos os processos; indique também como e onde deve ser feita a inicialização dos semáforos.

Considere que dispõe das funções **init(sem,value)**, **wait(sem)** e **signal(sem)** que operam sobre semáforos.

```
P1
...
...
while (goOn)
  doWork1();
...
...
```

```
P2
...
...
while (goOn)
   doWork2();
...
...
```

**3.** [1.5]

Um processo está em execução num computador cujo sistema operativo usa paginação a pedido (*demand paging*). O processo, ao qual foram atribuídos 4 quadros (*frames*), faz a seguinte sequência de referência às suas páginas: 1, 2, 3, 4, 1, 5, 4, 3, 6, 5, 1, 3, 2, 4, 6, 1, 5, 2, 4.

**a)** [0.5] Considerando que a substituição de páginas é feita, com alcance local, usando o algoritmo *LRU*, e que nenhuma página está carregada inicialmente, indique quantas faltas de página seriam geradas pela sequência mostrada. Justifique a resposta, recorrendo a um diagrama.

**b)** [0.5] Em determinado período da sua execução o processo entrou em *thrashing.* **1)** Explique como é possível constatar esse facto. **2)** Indique a sequência de referências em que isso aconteceu, sublinhando os números na sequência abaixo.

1)

2) **1, 2, 3, 4, 1, 5, 4, 3, 6, 5, 1, 3, 2, 4, 6, 1, 5, 2, 4** (sublinhar os números das páginas correspondentes ao período em que aconteceu *thrashing*)

**c)** [0.5] **1)** Explique por que é que o processo entrou em *thrashing* e **2)** indique uma medida adequada para evitar essa situação.

1)

2)

**4.** [1.0]

A estrutura dos *i-nodes* dos sistemas de ficheiros do tipo Linux foi pensada para suportar ficheiros de tamanhos muito diversos (desde alguns KByte a alguns TByte). **1)** Descreva sucintamente a forma como isto é conseguido e **2)** justifique um potencial problema de desempenho no acesso a ficheiros muito grandes.

1)

2)

**Nome do estudante:** _____ **Número:** _____

**Nota: nas questões seguintes apenas é necessário fazer tratamento de erros se tal for solicitado explicitamente**

**5.** [5.0]

Em Linux, o utilitário **bc** pode ser usado para fazer cálculos aritméticos. Quando invocado com as opções **–q** e **-i**, este programa fica à espera que o utilizador escreva uma expressão aritmética na entrada padrão (stdin) e apresenta de imediato o resultado na saída padrão (stdout), como se ilustra ao lado.

Escreva um programa que use **bc** como coprocesso para calcular o resultado de um conjunto de expressões. O programa deve ler do teclado as expressões, uma a uma, até que o utilizador tecle apenas **<enter>**, e guardar num ficheiro de texto a expressão e o seu resultado, como se ilustra ao lado. O nome do ficheiro de texto deve ser fornecido ao programa como argumento da linha de comando. Se o programa for executado várias vezes com o mesmo ficheiro como argumento, as expressões e correspondentes resultados devem ser acrescentadas ao ficheiro. Considere que todas as expressões são válidas.

**NOTA**: para implementar este programa não use as funções **popen()** nem **system()**.

*Exemplo de execução do utilitário* **bc***:*

```
$ bc -qi
2+3
5
(13-7)*5+1
31          // NOTA: o utilizador teclou CTRL-D
$ _
```

*Exemplo de conteúdo do ficheiro:*

```
5-7 = -2
(2+3)*((8-3)/5+1) = 10
...
```

**Nome do estudante:** _____ **Número:** _____

**6.** [2.5]

Um processo cria um *pipe* com nome (**FIFO**), tendo permissões efetivas de leitura e escrita por qualquer outro processo. Vários processos escritores enviam mensagens (todas do mesmo tamanho) para o **FIFO**; vários *threads* de um processo leitor lêem concorrentemente essas mensagens.

**a)** [0.5] Quem deve criar o **FIFO**? Porquê?

**b)** [0.5] Escreva o código de criação do **FIFO**, de nome **srvfifo**, no diretório **/users/tmp** (considere que a **umask** tem o valor **0000**).

**c)** [0.5] Diga se é possível que um escritor sobreponha, no **FIFO**, a sua mensagem à mensagem de outro escritor ou que dois processos leiam, do **FIFO**, a mesma mensagem. Se algum destes dois problemas puder acontecer, explique como poderia(m) ser evitado(s). Considere que as mensagens têm um tamanho inferior a **PIPE_BUF** bytes.

**d)** [1.0] Considere que, a dada altura, se pretende evitar que haja mais escritas no **FIFO**, mantendo-se a capacidade de leitura das mensagens já lá escritas, mas ainda não lidas.

**d.1)** Explique como poderia isso ser conseguido. Escreva o código que deveria ser executado e por que processo(s).

**d.2)** Explique como ficaria um leitor a saber quando o **FIFO** está vazio, isto é, quando já não há nele mais mensagens. Escreva o código respetivo.

**d.3)** Explique como ficaria um escritor a saber que já não será possível escrever no FIFO. Não escreva código.

d.1)

d.2)

d.3)

**Nome do estudante:** _____ **Número:** _____

**7.** [3.0]

Um programador que pretende criar um programa para executar sequencial e repetidamente um conjunto de *threads*, começou por escrever uma versão preliminar do código, apresentada abaixo (ver alínea **d)**), em que cada *thread* apenas escreve o seu número de ordem. Ao executar o programa com 3 *threads* verificou-se que aparecia no ecrã a sequência 1 2 3 1 2 3 1 2 3 …, como pretendido.

**a)** [0.5] Justifique a necessidade de utilização do *array* **arg[]**, na função **main()** do código abaixo apresentado.

**b)** [0.5] O que aconteceria se a função **main()** terminasse com a instrução **exit(0)** em vez de **pthread_exit(NULL)** ?

**c)** [0.5] Diga, justificando, se a espera de um *thread* pela sua vez de escrever o seu número se faz com bloqueio ou com *busy waiting* (espera ativa).

**d)** [1.5] Um programador experiente sugeriu que poderiam ser usadas *condition variables* (variáveis de condição), para garantir a sincronização dos *threads*, mantendo a escrita sequencial e repetida: cada *thread* deve sinalizar o *thread* seguinte quando chegar a vez deste escrever o seu número no ecrã. Indique as alterações necessárias para implementar esta solução (apenas as alterações).

```c
// NOTE: for simplicity, error treatment was ommited
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int numThreads;
int turn = 0; // The first thread to run must have thrIndex=turn=0

void * thr(void *arg)
{
  int thrIndex = *(int*)arg; // The effective indexes are 0,1,2,...
  while (1)
  {
    pthread_mutex_lock(&mutex);
    if (thrIndex == turn)
    {
      printf("%d ", thrIndex + 1); // The numbers shown are 1,2,3,...
      turn = (turn + 1) % numThreads;
    }
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}

int main()
{
  printf("Number of threads ? "); scanf("%d", &numThreads);
  int *arg = (int *) malloc(sizeof(int)*numThreads);
  pthread_t *tid = (pthread_t *) malloc(sizeof(pthread_t)*numThreads);
  for (int i = 0; i < numThreads; i++)
  {
    arg[i] = i;
    pthread_create(&tid[i], NULL, thr, (void*)&arg[i]);
  }
  pthread_exit(NULL);
}
```

*Indique aqui as alterações ao código, assinalando adequadamente o ponto do programa em que devem ser inseridas:*

**FIM**