FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO
Mestrado Integrado em Engenharia Informática e Computação
Exame de Sistemas Operativos – Época Normal

21 de Janeiro de 2008 **Duração: 2 h 15 m**

(Responda às questões 1 a 4 na mesma folha e às questões 5 e 6 em duas folhas separadas)

1.

[2] O algoritmo de escalonamento a usar num sistema operativo poderia depender do tipo de processos que se espera que ele tenha de controlar. Para cada um dos tipos de sistemas de computação (A, B e C) abaixo descritos, diga qual (quais) dos algoritmos de escalonamento a seguir indicados seria(m) mais apropriados: *First-Come-First-Served, Shortest-Job-First*, *Priority*, *Round-Robin* e *Multilevel-Feedback-Queue*. No caso de algum destes algoritmos poder ser executado com ou sem preempção, diga qual das duas versões escolheria. Justifique as suas respostas.

- A- sistema partilhado, cujos utilizadores executam, essencialmente, edição de texto e pesquisas na Web;
- B- sistema de controlo de tráfego aéreo onde são executados múltiplos processos que fazem o seguimento e o controlo da rota de aviões, precavendo situações de colisão;
- C- sistema de processamento em *batch*, em que se pretende minimizar o tempo médio de espera dos *jobs*.

2.

a) [1.5] Considere a seguinte tentativa de implementação das operações `wait()` e `signal()` sobre semáforos.

a.1) Qual o objectivo das chamadas `disable_interrupts()` e `enable_interrupts()`?

a.2) Qual o(s) problema(s) que há em implementar estas operações recorrendo à inibição de interrupções ?

```
wait(S):
  disable_interrupts();
  while (S.count <=0);
  S.count--;
  enable_interrupts();
```

```
signal(S):
  disable_interrupts();
  S.count++;
  enable_interrupts();
```

a.3) Mostre que esta tentativa de implementação não funciona.

b) [2.5] Pretende-se implementar uma "barreira de N processos", isto é, garantir que cada processo, de um conjunto de processos, espera, em determinado ponto da sua execução, que todos os outros processos do conjunto atinjam um certo ponto da respectiva execução. Tomando um conjunto de 3 processos como exemplo, teríamos a situação ilustrada, ao lado. Indique como implementaria uma "barreira de N processos", recorrendo a semáforos. Admita que dispõe das seguintes funções que operam sobre semáforos: `init(sem,value)`, `wait(sem)` e `signal(sem)`.

P1	**P2**	**P3**
...	...	...
...	**B:** espera que P1 chegue a A e que P3 chegue a C	...
A: espera que P2 chegue a B e que P3 chegue a C	...	**C:** espera que P1 chegue a A e que P2 chegue a B
...	...	...
...	...	...
...	...	...

c) [1] Um servidor tem ligadas a si 5 impressoras e tem N processos em execução. Cada processo pode requisitar até M impressoras. Para cada um dos seguintes valores de N e M, indique, justificando, se há ou não possibilidade de ocorrência de *deadlocks*: **c.1)** N=1, M=3; **c.2)** N=2, M=3; **c.3)** N=4, M=3 e **c.4)** N>2, M=1.

3.

[2] Um programador que tinha desenvolvido um programa de manipulação de matrizes bidimensionais de grande dimensão, em linguagem Fortran, verificou que depois de traduzir o programa para linguagem C, recorrendo a um tradutor automático, o programa passou a executar muito mais lentamente, embora o sistema de computação fosse o mesmo e a "carga do sistema" semelhante à anterior. Sabendo que os elementos de uma matriz são guardados, em Fortran, coluna a coluna (isto é, os elementos da primeira coluna, seguidos dos da segunda coluna, etc.) e, em C, linha a linha (isto é, os elementos da primeira linha, seguidos dos da segunda linha, etc.), encontra alguma explicação para que isto tenha acontecido, que possa ser justificada pela técnica de gestão de memória utilizada? Justifique a sua resposta e descreva as condições gerais para a ocorrência de situações semelhantes.

4.

[2] Considere um sistema de ficheiros de tipo Unix. Para cada um dos seguintes itens indique qual o local ou estrutura de dados em que ele é guardado: 1) nome de um ficheiro; 2) tamanho de um ficheiro; 3) data da última modificação de um ficheiro; 4) número do *i-node* associado a um ficheiro; 5) lista de blocos de dados que compõem um ficheiro; 6) permissões de acesso a um ficheiro; 7) descritor associado a um ficheiro; 8) apontador do ficheiro - local do ficheiro onde será efectuada a próxima operação de leitura/escrita; 9) lista de blocos livres do disco.

(continua no verso)

NOTA: nos programas pode omitir os ficheiros de inclusão e, em geral, os testes de erro nas chamadas ao sistema.

5.

Pretende-se escrever uma versão muito simplificada do utilitário de Unix **find**, que serve para encontrar, de forma recursiva, a partir de um directório base, ficheiros que satisfaçam algumas características e sobre eles executar algumas acções. O programa a desenvolver, **gotcha-thr**, ao contrário de **find**, vai tentar encontrar apenas um ficheiro e será estruturado em várias partes, sendo as essenciais: inicialização, paralelização da pesquisa, pesquisa propriamente dita e execução de acção sobre o que for encontrado.

Exemplos de utilização:

```
sh> gotcha-thr /home/luis -name abc.txt -print
```

(procura abaixo de **/home/luis** um ficheiro com o nome **abc.txt** e mostra, no ecrã, o *path* e nome do primeiro que encontrar)

```
sh> gotcha-thr . -mmin -3 -delete
```

(procura abaixo de . um ficheiro modificado há menos de 3 minutos e apaga o primeiro encontrado)

a) [1] Escreva uma rotina, **void args(int argc, char *argv[])** que leia os argumentos passados a **gotcha-thr** e, se estiverem em número correcto, os coloque em apontadores globais (**char *basedir**; **char *test**; **char *testparam**; **char *action**), se não termina o programa. Note que **gotcha-thr** tem, sempre, além do seu próprio nome, 4 argumentos; o primeiro deve ser um nome de um directório e os restantes têm um significado que depende do tipo de operação solicitada.

b) A rotina **void parallel(char *dir)**, abaixo apresentada, deve verificar se o directório **dir** tem sub-directórios e, se assim for, criar um novo *thread* para cada um deles, passando-lhe o correspondente nome do sub-directório. Complete a rotina **parallel()**, escrevendo cada uma das partes de código assinaladas que:

b.1) [1] (PARTE A) verifica se **dir** é mesmo um directório;

b.2) [0.5] (PARTE B) constrói **name** para cada entrada de **dir**;

b.3) [1] (PARTE C) cria um novo *thread* com função **void *thr(void *farg)** (nota: antes de escrever esta parte do código, comece por analisar a rotina **thr()**, abaixo apresentada).

c) [1] A função **int find(char *dir, char *test, char *testparam)**, abaixo apresentada, pesquisa o directório **dir**, testando os ficheiros relativamente à característica **test**, com parâmetro **testparam**, e retorna **NOTFOUND**, **FIRST** ou **TOOLATE** conforme nada tenha encontrado, tenha sido o primeiro *thread* a encontrar o que se procurava ou tenha encontrado depois de outros *threads*. Utilizando as variáveis globais que achar conveniente, escreva o código de sincronização (PARTE D) que garanta que só um dos *threads* consegue retornar FIRST.

```
void parallel(char *dir)
{
  DIR *od;
  struct dirent *rd;
  struct stat buf;
  char name[BUFFSIZE];
  ...
  // PARTE A. Verificar se 'dir' é mesmo um directório.
  //          Se não for, invocar pthread_exit().
  ...
  if ((od=opendir(dir)) == NULL)
    { perror("opendir"); exit(1); }
  while ((rd=readdir(od)) != NULL)
  {
   // PARTE B. Construir 'name' de uma entrada de 'dir'
   ...
   if (stat(name, &buf) == -1) perror("stat");
   if (is_directory)
   { // PARTE C. Criar um thread.
        ...
   }
  } // while
  ...
}
```

```
int find(char *dir, char *test, char *testparam)
{
  int first = 0;
  char *found = NULL; // nome do ficheiro encontrado

  // código de pesquisa (test e testparam) em 'dir'
  // que, eventualmente, altera 'found'
  ...
  ...
  if (found != NULL)
  {
   // PARTE D. código de sincronização
   ...
   return (first==1 ? FIRST : TOOLATE);
  }
  return (NOTFOUND);
}
```

```
struct thr_arg { char dirname[BUFFSIZE]; void *mem; };

...

void *thr(void *farg)
{
 parallel(((struct thr_arg *)farg)->dirname);
 if ((find(((struct thr_arg *)farg)->dirname, test, testparam)) == FIRST)
  execute_action (.....);
 free (((struct thr_arg *)farg)->mem);
 pthread_exit(NULL);
}
```

6.

Em sistemas embebidos existe um mecanismo de *hardware* (*watchdog timer*) que, se não for periodicamente acedido pelo *software*, provoca um *reset* e reinicialização do sistema. Isto evita que o dispositivo fique inoperacional se ocorrerem problemas no *software* de controlo.

Para sistemas embebidos multiprocesso apenas um processo de controlo (**checker**) deverá aceder periodicamente ao *watchdog timer*, mas esse processo deverá garantir que todos os outros processos se encontram em execução normal.

Para esse efeito, o processo inicial reserva uma zona de memória partilhada, com um *slot* associado a cada novo processo, inicializa cada *slot* com o valor 1, e verifica periodicamente se todos os *slots* se encontram a zero, acedendo então ao *watchdog* e voltando a colocar os *slots* a 1. Cada um dos novos processos, por sua vez, acede periodicamente ao seu *slot* e coloca-o a zero.

A estrutura de cada *slot* é

```
struct slot { pid_t pid; int alive;},
```

e consiste no identificador de um processo e num inteiro que deverá ser, periodicamente, colocado a 0 por esse processo e a 1 pelo **checker**.

a) [1] Para o **checker**, escreva o código da função

`int init_alive(int num, struct slot **shmadd, int *shmid, int *semid)`, que

- reserva uma zona de memória partilhada com espaço para `num` *slots* e a associa ao processo corrente, devolvendo através de `shmadd` o seu endereço e através de `shmid` o seu identificador, e
- cria um semáforo de controlo dessa memória, cujo identificador deve ser devolvido em `*semid`.

Os *slots* e o semáforo devem ser inicializados com valores adequados e a função deverá devolver 0 ou 1 conforme tenha ou não sucesso.

b) [1] Escreva a função do **checker**

```
int check_proc(int num, struct slot **shmadd, int *shmid, int *semid),
```

que verifica se todos os *slots* têm o campo `alive` a 0 e em seguida os coloca a 1. A função deve devolver 0, se todos os *slots* tiverem o valor 0, ou 1, em caso contrário.

c) Para que todos os processos possam usar este mecanismo, devem registar-se perante o **checker**, criando este para o efeito um *FIFO* de nome "**reg"**. Os processos que pretendam registar-se devem criar um *FIFO* de resposta com um nome privado, criar e preencher a estrutura

```
struct msg{ pid_t pid; char[40] reply_fifo;}
```

com a sua *pid* e o nome do *FIFO* privado, e escrevê-la no *FIFO* "**reg"**.

c.1) [1] Escreva a função do **checker**

```
int open_fifo(),
```

que cria e abre para leitura, se não existir, um *FIFO* de nome "**reg"** de tal modo que operações de `read()` subsequentes não bloqueiem; a função deve devolver 0 em caso de insucesso ou o descritor do *FIFO*.

c.2) [1.5] Escreva a função do **checker**

```
pid_t read_fifo(int fifofd),
```

que verifica se existe alguma mensagem no *FIFO*. Se houver mensagens, deve ler uma, activar um *slot*, criar e preencher a estrutura

```
struct ipcids{ int shmid; int semid;}
```

e escrevê-la no `reply_fifo`; `shmid` e `semid` são os identificadores que devem ser usados para os novos processos poderem aceder à zona de memória partilhada e ao semáforo. A função deve devolver a *pid* do processo que se pretende registar ou 0 se não houver mensagens.

FIM

JAS/JSC/MMC