Fundamentos de Sistemas de Operação -2oTeste 15/12/2010

Sem consulta. **Duração total: 2h30 horas**

Questão 1.

a) Apresente a expressão que define o invariante de um semáforo, tal como definido por Dijkstra. Explique qual o significado desse invariante, quando as operações P ou V são executadas.

b) Pretende-se implementar as operações P(s) e V(s) sobre **semáforos genéricos**, segundo a definição de Dijkstra. Para esta implementação assuma que dispomos de **semáforos binários**, que só podem assumir os valores 0 ou 1 e que suportam as correspondentes operações Pb e Vb (em que o indice b indica tratar-se de um semáforo binário).

Apresente, em pseudo-código, uma implementação, em modo utilizador, das operações P(s) e V(s) sobre semáforos genéricos baseando-se em operações em semáforos binários. Nesta solução assuma que apenas dispõe de dois semáforos binários previamente criados e inicializados: **mutex** com o valor inicial 1 e **block** com o valor inicial 0. Assuma também que tem acesso ao valor inteiro s do semáforo que se pretende implementar.

c) Em linguagens de programação concorrente que suportam o conceitode monitor de sincronização, diga como são definidas as variáveis de tipo **condition** e quais as suas diferenças relativamente aos **semáforos**.

Questão 2. Considere o seguinte programa concorrente em que os dois processos P1 e P2 executam acções sobre as seguintes variáveis partilhadas: um inteiro **turn** e um vector de inteiros **flag[]**:

int turn; /* o valor inicial de **turn** é irrelevante para o programa */

int flag[2]; /* os valores iniciais são **flag[0] = flag[1] = 0** */

Processo P1

```
while (TRUE)    {
    1. flag[0] = 1;
    2. turn = 1;
    3. while (flag[1] == 1 && turn == 1) ;
    4. REGIÃO CRÍTICA
    5. flag[0] = 0;
}
```

Processo P2

```
while (TRUE) {
1.      flag[1] = 1;
2.      turn = 0;
3.      while (flag[0] == 1 && turn == 0) ;
4.      REGIÃO CRÍTICA
5.      flag[1] = 0;
        }
```

As acções numeradas de 1 a 3 (para a entrada) e 5 (para a saída) em cada processo pretendem implementar um protocolo de acesso que deve garantir a exclusão mútua da execução das duas regiões críticas indicadas. Diga se esta é uma solução correcta:

(i) Garante a exclusão mútua? Justifique.

(ii) Garante que não ocorrem situações de impasse em que os processos se bloqueiem mutuamente na tentativa de entrada nas suas regiões críticas? Justifique.

(iii) Evita situações de injustiça, em que um processo veja a sua tentativa de entrada eternamente adiada pelo facto de o outro processo aceder repetidamente à região crítica (em ciclo: entrando, acedendo e saindo)? Justifique.

Questão 3. Considere uma zona de memória partilhada por **N** processos concorrentes e com a capacidade máxima para conter, a cada momento, **um** único bloco de 1Kbytes. Admita que esta zona foi previamente reservada pelo SO e se mantém permanentemente em memória. Inicialmente a zona está vazia, ou seja tem um conteúdo indefinido.

O processo de identificador (pid) **P1** produz um bloco (também de 1Kbytes) de cada vez e depois tenta escrevê-lo naquela zona de memória. Cada um dos restantes processos, de identificadores **P2, ... PN**, deve tentar ler o conteúdo de cada bloco escrito por **P1**. Só após todos os processos (P2, ... PN) terem lido o conteúdo de um dado bloco, será este considerado lido e a zona de memória será considerada livre (permitindo que P1 escreva outro bloco na mesma zona de memória).

- O processo **P1** invoca a função **pôrBloco(char *buf)** que o bloqueia até que a zona de memória esteja livre e, então, copia o conteúdo apontado por **buf** para a zona de memória partilhada, invocando uma função auxiliar **colocar_bloco(char *buf)**, que se assume já implementada (mas não garante condições de sincronização). O bloco **buf** tem também o tamanho de 1Kbytes.
- Cada um dos processos **P2, ... PN** invoca a função **lerBloco(char *buf),** a qual executa a seguinte sequência de passos:

  1. bloqueia o processo até que exista um novo bloco para ler;

  2. obtém uma cópia do bloco, para uma zona de memória do processo apontada pelo argumento **buf**. Para obter a cópia do bloco, dispõe-se de uma função auxiliar **obter_bloco(char *buf)**, devolvendo o apontador para a memória para onde copia o bloco lido e que se assume já implementada, incluindo a reserva da memória (mas não garante condições de sincronização).

  3. aguarda, bloqueando-se até que todos os outros processos (do conjunto **P2-PN)** tenham também lido esse bloco e só então a função **lerBloco** retorna ao processo invocador. NOTA: não é necessário permitir que as leituras do bloco pelos processos P2-PN sejam simultâneas.

Apresente o código C das funções **pôrBloco(buf)** e **lerBloco(buf)**, que devem garantir todas as condições de sincronização deste problema, baseando-se exclusivamente em memória partilhada e em semáforos segundo a definição de Dijkstra. Nesta questão, não pode utilizar filas de mensagens, pipes ou sinais Unix.

Questão 4. Considere um problema semelhante ao da Questão 3, mas em que o processo de identificador (pid) **P1** produz um bloco (também de 1Kbytes) de cada vez e depois envia-o, com base em **comunicação por mensagens**, para cada um dos restantes processos, de identificadores **P2, ... PN**, que devem tentar receber o bloco enviado por **P1**. Só após todos os processos (P2, ... PN) terem recebido um dado bloco, se permitirá que P1 envie outro bloco aos processos (P2, ... PN).

- O processo **P1** invoca a função **enviarBloco(char *buf)** que envia um bloco, bloqueando depois o processo, tal que a função só retorne ao processo quando todos os processos (P2, ... PN) recebam esse bloco.
- Cada um dos processos **P2, ... PN** invoca a função **receberBloco(char *buf)** para receber um bloco, bloqueando depois o processo, tal que a função só retorne ao processo invocador quando todos os outros processos (P2, ... PN) tiverem também recebido esse bloco.

a) Apresente o código C das funções **enviarBloco** e **receberBloco** que devem garantir todas as condições de sincronização deste problema, baseando-se exclusivamente em filas de mensagens (message queues) Unix V. Deve indicar todas as filas que utilizar mas pode assumir que as filas já se encontram criadas e inicializadas. Nesta questão, não pode utilizar memória partilhada, semáforos, pipes ou sinais Unix.

b) Apresente o código C da seguinte função modificada:

int receberBloco(char *buf, int time)

tal que garanta que, se um bloco não tiver sido recebido após ter decorrido o intervalo de tempo indicado pelo argumento **time** (em segundos), então, a função **receberBloco** deve forçar o retorno ao processo invocador, retornando um indicador inteiro com o valor **-1**. Caso contrário, isto é, no caso de ser recebido um bloco antes do fim do intervalo **time** indicado, o valor retornado pela função deve ser **0**. Para esta questão utilize operações sobre filas de mensagens e sinais Unix.