LEIC
LERCI

Sistemas Operativos

Colectânea de Exercícios de Sincronização

Outubro de 2004

Exercícios de Sincronização

Resolva os seguintes exercícios recorrendo a trincos e semáforos para fazer a sincronização necessária. Os problemas marcados com asterisco são exercícios clássicos de sincronização.

A. Competição por um recurso

---

1. [Banco] Considere um banco com uma só conta. Nessa conta é possível efectuar as operações de depositar e levantar. Cada uma destas operações tem um parâmetro inteiro com o valor a depositar ou a levantar. A operação de depositar consiste unicamente em somar o valor ao saldo actual. A operação de levantar verifica se o saldo é suficiente para satisfazer o levantamento e em caso afirmativo subtrai o valor ao saldo. Considere que ambas as operações podem ser efectuadas por vários processos em simultâneo.

a. É necessária sincronização entre os vários processos que executam as operações?

b. Escreva o código das rotinas depositar e levantar em C.

---

2. [Somas concorrentes] Pretende-se efectuar a soma de N números utilizando K processos que em paralelo efectuam as somas parciais. Suponha que os N números se encontram numa lista ligada. Os processos executam um ciclo, retirando dois números, somando-os e devolvendo o resultado à lista. Os processos devem ser bloqueados quando não encontram números para somar. O último elemento que fica na lista é a soma total. A inicialização é efectuada por um processo dedicado que não deve considerar no exercício. Defina as estruturas de dados, os mecanismos de sincronização e programe as rotinas que fazem parte do ciclo dos processos. Para estruturar o programa considere que as rotinas `TirarNumero`()e `PorNumero`() efectuam os acessos à lista.

---

3. [Leitores-Escritores*] Considere uma base de dados e as operações de escrita e leitura `write_db()` e `read_db()` respectivamente. Considere ainda, que existem vários processos que lêem e escrevem na base de dados. Para evitar a corrupção da base de dados as actualizações não podem ser efectuadas em simultâneo por vários processos. Já as operações de leitura podem ocorrer em simultâneo sem que provocar a corrupção da base de dados, mas não podem ocorrer em simultâneo com as operações de escrita porque poderiam devolver dados incoerentes.

a. Dê um exemplo de como duas ou mais actualizações em simultâneo da base de dados pode provocar a sua corrupção.

b. Escreva, em C, o código que os processos que escrevem e lêem da base de dados terão que efectuar antes e depois de efectuarem a actualização e leitura, respectivamente. Adopte as designações `read_lock_db()`, `read_unlock_db()`, `write_lock_db()` e `write_unlock_db()`.

---

4. [Ponte estreita] Considere que existe uma ponte com uma única via por onde passam carros em ambos os sentidos. Para evitar colisões existe um semáforo luminoso em cada extremidade da ponte. Por cada carro que se aproxima da ponte de oeste é criado um processo que executa a rotina `carroDeOeste` e coloca o semáforo de oeste a vermelho. Quando um carro de oeste sai da ponte invoca uma rotina executa a rotina `carroSaiDeOeste` de modo a sinalizar a sua saída da ponte, colocando o semáforo a verde. Para os carros provenientes de Este o procedimento é semelhante, chamando neste caso a rotinas executa a rotina `carroDeEste` e executa a rotina `carroSaiDeEste`. Note que podem passar vários carros na mesma direcção em simultâneo, mas não em direcções opostas.

a. Escreva o código das rotinas: `carroDeOeste()`, `carroSaiDeOeste()`, `carroDeEste()` e `carroSaiDeEste()`. Note que o algoritmo dos leitores/escritores pode ser adaptado se considerarmos os carros de oeste "leitores tipo A", incompatíveis com os carros de este que são "leitores tipo B". A incompatibilidade entre os dois tipos de leitores é idêntica à incompatibilidade entre leitores e escritores.

b. Considere agora que não podem existir mais do que 5 carros na ponte em simultâneo (senão ela cai). Reescreva o código.

---

5. [Searchers, inserters, deleters] Três tipos de processos partilham acesso a uma lista simplesmente ligada: *searchers*, *inserters* e *deleters*. Os *searchers* apenas examinam a lista, logo podem executar-se concorrentemente entre si. Os *inserters* devem ser mutuamente exclusivos para evitar inserções paralelas. No entanto, uma inserção pode prosseguir em paralelo com qualquer número de procuras. Finalmente, os *deleters* podem aceder à lista um de cada vez e devem ser mutuamente exclusivas com procuras e inserções. Escreva o código para os três tipos de processos que assegurem estas propriedades.

---

6. [Parque de estacionamento] Considere um parque de estacionamento com capacidade máxima para MAX viaturas. Existem três tipos de utilizadores deste parque: docentes, funcionários e alunos. Enquanto há lugares livres no parque, a ordem de entrada é por ordem de chegada. A partir do momento em que o parque fica cheio, i.e. a entrada de viaturas fica condicionada à saída de outras, deve ser dada prioridade às viaturas dos docentes, funcionários e alunos (por esta ordem). Utilizando semáforos com as habituais operações Esperar e Assinalar, programe em pseudo-código (C) as funções `EntraViatura(int tipo)` e `SaiViatura()`.

---

7. [Passeios turísticos] Suponha que existem N passageiros e um autocarro que pode ter no máximo C passageiros, onde C < N. Os passageiros repetidamente esperam para entrar no autocarro e fazer um passeio pela cidade. O carro só pode circular quando está completamente cheio. Quando o autocarro chega, C passageiros devem invocar `EntrarNoCarro()`. Seguidamente, o carro deve invocar `Partir()`. Mais tarde os passageiros devem invocar `SairDoCarro()`, depois do carro invocar `Chegar()`. Escreva o código dos passageiros e do autocarro que assegura este comportamento.

B. Competição por múltiplos recursos

8. [Múltiplos recursos] Suponha um sistema que permite a utilização de seis recursos – R1, R2, R3, R4, R5 e R6 – de natureza distinta (por exemplo, R1 = disk drive, R2 = printer, etc). Só existe uma instância de cada recurso. Considere quatro processos concorrentes – P1, P2, P3 e P4 – que só podem começar a correr uma vez garantida a posse dos recursos que necessitam para a execução da sua tarefa. Esta estratégia simples, mas ineficiente do ponto de vista do rendimento do sistema, garante a impossibilidade de interblocagem (deadlock) entre os processos (situação possível caso estes requisitassem cada recurso apenas quando estritamente necessário). A tabela de recursos, para cada processo, está descrita de seguida.

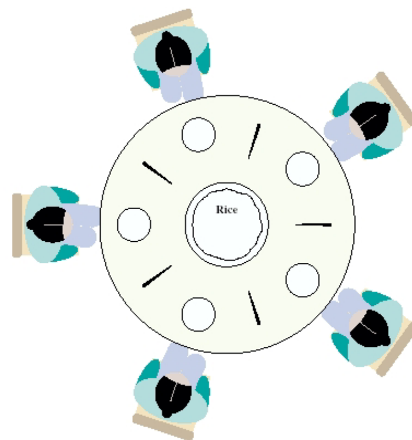
	R1	R2	R3	R4	R5	R6
P1	Sim	Sim	Sim		Sim	
P2	Sim	Sim		Sim	Sim	
P3		Sim		Sim	Sim	Sim
P4		Sim	Sim	Sim		Sim

A cada recurso corresponde um semáforo (inicializado a 0) e o sistema deve operar por ciclos do seguinte modo. Um processo, dito agente, escolhe aleatoriamente uma das configurações especificadas na tabela e executa uma operação assinalar nos semáforos correspondentes. A partir desta informação, o processo adequado deverá ser posto em execução por um algoritmo de cooperação a conceber. Entretanto o agente fica bloqueado à espera que o processo escolhido lhe comunique o seu fim de tarefa. Aí, o agente torna a escolher nova configuração de recursos e o ciclo repete-se.

Escreva um programa em C que lance os processos P1, P2, P3, P4 e que coordene a execução destes segundo as normas descritas.

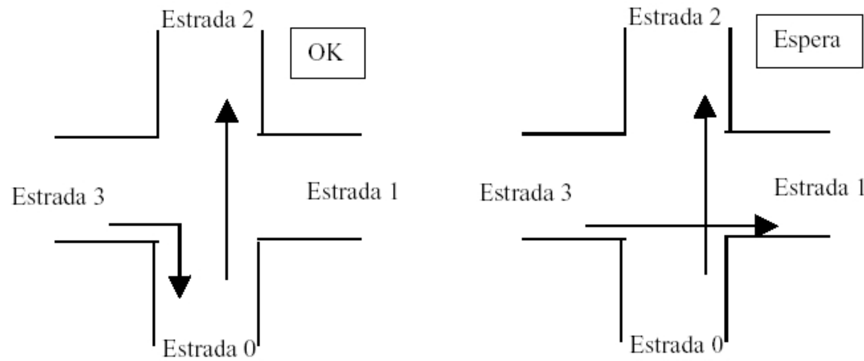
---

9. [Jantar dos Filósofos*] Cinco filósofos estão sentados à volta de uma mesa redonda. Cada filósofo tem à sua frente uma taça de arroz. Entre cada par de filósofos existe um pauzinho chinês. A vida de um filósofo alterna entre comer e pensar, mas para comer necessita de dois pauzinhos. Quando um filósofo tem fome, tenta adquirir um pauzinho da direita e outro da esquerda, independentemente da ordem. Se conseguir obter ambos, o filósofo come durante um tempo, depois pousa os pauzinhos e recomeça a pensar. Caso contrário o filósofo tem que aguardar (com fome) até ter ambos os pauzinhos disponíveis.

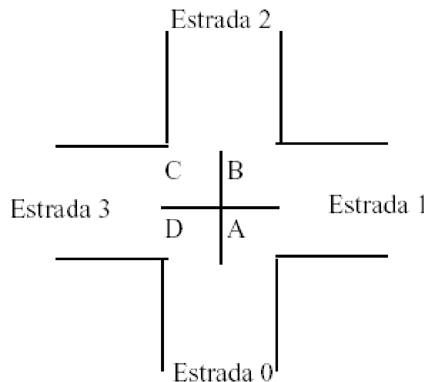


Escreva o código das tarefas, cada uma representando um filósofo, que obedeça à especificação do problema mas evitando situações de interblocagem (*deadlock*).

10. [Cruzamento] Imagine um cruzamento de quatro estradas em que os automóveis podem entrar por qualquer uma das estradas e sair por qualquer outra (não pode sair pela que entrou). Dependendo das estradas de entrada e saída alguns automóveis podem estar no cruzamento em simultâneo. Por exemplo, na figura 1, se um automóvel pretender seguir da estrada 0 para a estrada 2 e outro pretender seguir da 3 para a 0 podem entrar no cruzamento em simultâneo, já se um automóvel pretender seguir da estrada 0 para a estrada 2 e outro da estrada 3 para a 1 então terão que esperar.



Na figura 2, o cruzamento está dividido em quadrantes, se os caminhos pretendidos pelos automóveis cruzarem o mesmo quadrante então os caminhos não são permitidos em simultâneo, caso contrário são.



Admita que possui uma função que lhe indica se dois caminhos X e Y usam quadrantes comuns – `int useComonRegion(int X_inicio, int X_fim, int Y_inicio, int Y_fim)` – e a função a função que efectua a travessia do cruzamento:

```
void car(int startLane, int endLane) {
   request(startLane, endLane);
   enter_cross();
   leave(startLane, endLane);
}
```

Realize as funções `request()` e `leave()`. Declare todas as variáveis globais que necessitar.
Nota: Considere as semelhanças com o exemplo do jantar dos filósofos.

C. Cooperação entre processos

---

11. [Produtores-Consumidores*] Considere um *buffer* circular com capacidade para armazenar `N` items de informação. O *buffer* é usado para transmitir informação dos processos produtores para os processos consumidores. Em cada acesso ao *buffer*, cada produtor deposita apenas um item de informação no *buffer* e cada consumidor retira apenas um item de informação. Quando um produtor tenta introduzir informação no *buffer* estando este repleto, deve passar ao estado bloqueado; o mesmo deve suceder a um consumidor que tenta extrair um item quando o *buffer* está vazio. Os processos assim bloqueados devem ser acordados apenas quando o conteúdo do *buffer* assim o justificar.

```
#define N = ...;

struct item_t buffer[N];

int IdxProxLeitura, IdxProxEscrita;


void Produtor(int i)
{
    item_t item;
    while(1)
    {
        // codigo arbitrario que preenche item
        DepositaItem(Item);
    }
}


void Consumidor(int i)
{
    item_t item;
    while(1)
    {
        RetiraItem(item);
        // codigo arbitrario que processa Item
    }
}
```

a. Escreva o código das rotinas `DepositaItem()` e `RetiraItem()` e complete adequadamente o programa, de modo a satisfazer os requisitos de manipulação especificados para o *buffer* (não é necessário explicitar campos para o tipo item_t, mas utilize as variáveis `IdxProxLeitura` e `IdxProxEscrita` para efectuar o acesso circular ao *buffer*).

b. Considere a seguinte alteração: existe apenas um produtor e os consumidores não devem ficar bloqueados ao tentar retirar um item de informação no *buffer*. Isto é, cada consumidor deve previamente detectar se o *buffer* está vazio. Em caso afirmativo, não é retirado um item (mas sim tomado outro rumo de acção arbitrário); caso contrário, é retirado um item do *buffer*. Modifique o programa de modo a cumprir este novo protocolo de utilização do *buffer*.

---

12. [Pipe Unix] Neste problema pretende-se reproduzir, à custa de semáforos, os mecanismos de sincronização inerentes à utilização de um *pipe* (*buffer* circular no sistema Unix. Considere um *buffer* circular com capacidade para armazenar N itens de informação.

Cada processo da classe produtor/consumidor pode tentar depositar/extrair um número variável de itens, mas sempre entre 1 e N. As regras de utilização do *buffer* são as seguintes:

- Se o *buffer* não tem espaço para satisfazer um pedido de depósito de n itens por parte de um produtor, este passa ao estado bloqueado. A sua execução é retomada assim que qualquer consumidor retire itens do *buffer* e deve ser dirigida no sentido de efectuar nova tentativa de escrita;
- Se um consumidor tentar ler o *buffer* vazio, deve passar ao estado bloqueado. A sua execução deve ser retomada assim que qualquer produtor introduza informação no *buffer*;
- Quando um consumidor tenta ler mais informação do que a que existe no *buffer*, extrai todos os itens disponíveis do *buffer* e retorna (não fica bloqueado).

Implemente este protocolo de cooperação na manipulação do *buffer* usando como referência o problema dos produtores-consumidores.

---

13. [Passagem de testemunho] Considere N processos `Test` que se executam num ciclo infinito a circular um testemunho (*token*) que é um inteiro com o valor inicial 0. O testemunho começa no processo 0, passa para o processo 1, depois para o 2 e assim sucessivamente até ao processo N-1 que o passa novamente ao processo 0, repetindo-se este funcionamento num ciclo infinito. De cada vez que o testemunho passa por um processo, é incrementado. De cada vez que o testemunho passa pelo processo 0, o seu valor é escrito no terminal. Cada processo tem um identificador `pid` compreendido entre 0 e N-1. Programe os processos `Test` usando variáveis partilhadas e semáforos.

---

14. [Barbeiro*] Numa barbearia existe uma cadeira onde o barbeiro corta cabelo e N cadeiras para os clientes que estão à espera, aguardarem sentados. Se não existem clientes, o barbeiro senta-se na cadeira e adormece. Quando um cliente chega, ele tem que acordar o barbeiro dorminhoco para lhe cortar o cabelo. Se entretanto chegarem mais clientes enquanto o barbeiro estiver a cortar o cabelo ao primeiro, ou esperam numa cadeira livre ou vão-se embora se já não houver mais cadeiras livres. Implemente o código das tarefas `Barbeiro` e `Cliente`.

---

15. [Hilzer's Barbershop] Numa barbearia existem três cadeiras, três barbeiros e um local de espera que pode acomodar quatro pessoas num sofá e que tem um área em que os clientes podem estar à espera em pé. O número máximo de clientes que pode estar na sala é de 20. Um cliente não pode entrar na loja se esta estiver totalmente cheia com clientes à espera. Se o cliente puder entrar, senta-se no sofá ou aguarda em pé se não há lugar no sofá. Quando um barbeiro está livre o cliente que está à espera há mais tempo no sofá é

servido e, se existirem clientes à espera em pé, aquele que está em pé há mais tempo toma o lugar no sofá. Quando o corte de cabelo de um cliente terminou, qualquer barbeiro pode aceitar pagamento, mas porque existe apenas uma caixa registadora, o pagamento é aceite para um cliente de cada vez. Os barbeiros dividem o seu tempo entre cortar cabelo, aceitar pagamento e dormir na cadeira à espera que um cliente chegue.

- Os clientes invocam as funções: `EntrarNaLoja()`, `SentarNoSofa()`, `SentarNaCadeira()`, `Pagar()` e `SairDaLoja()`. Os barbeiros invocam as funções `CortarCabelo` e `AceitarPagamento`. Existem assim as seguintes restrições:
- Os clientes não podem invocar `EntrarNaLoja` se a loja está cheia.
- Se o sofá está cheio, um cliente que entrou na loja não pode invocar `SentarNoSofa` até que um dos clientes do sofá invoque `SentarNaCadeira()`.
- Se as três cadeiras estão ocupadas, um cliente não pode invocar `SentarNaCadeira()` até que um dos clientes a cortar cabelo invoque `Pagar()`.
- O cliente tem que chamar `Pagar` antes do barbeiro poder `AceitarPagamento()`.
- O barbeiro tem que `AceitarPagamento()` antes que o cliente chame `SairDaLoja()`.

Escrever o código do `Cliente` e do `Barbeiro` que assegurem este comportamento.

---

16. [Fumadores] Existem três fumadores. Cada um prepara um cigarro e depois fuma-o, repetindo estas duas acções continuamente. Para fazer um cigarro, o fumador necessita três coisas: tabaco, mortalha e fósforos. Cada fumador tem apenas uma das coisas, com uma quantidade infinita de abastecimento. Além dos fumadores, existe um agente que tem abastecimento infinito dos três ingredientes. Inicialmente os fumadores estão à espera. O agente coloca aleatoriamente dois ingredientes na mesa e avisa o fumador que necessita desses dois ingredientes. O fumador retira os ingredientes da mesa e fuma o cigarro por um determinado tempo (aleatório). Ao libertar a mesa, o ciclo repete-se e o agente coloca então outros dois ingredientes na mesa. Programe as tarefas `Fumador` e `Agente` de forma a simular o comportamento indicado.

---

17. [Jantar de Gauleses] Uma tribo gaulesa janta em comunidade a partir de uma mesa enorme com espaço para M javalis grelhados. Quando um gaulês quer comer, serve-se e retira um javali da mesa a menos que esta já esteja vazia. Nesse caso o gaulês acorda o cozinheiro e aguarda que este reponha javalis na mesa. O código seguinte representa o código que implementa o gaulês e o cozinheiro.

```
void Gaules()
{
   while(true)
   {
      Javali j = RetiraJavali();
      ComeJavali(j);
   }
}
```

```
void Cozinheiro()
{
   while(true)
   {
      ColocaJavalis(M);
   }
}
```

Implemente o código das funções `RetiraJavali()` e `ColocaJavalis()` incluindo código de sincronização que previna *deadlock* e acorde o cozinheiro apenas quando a mesa está vazia.

---

18. [Pai Natal] O Pai Natal dorme na sua loja no Pólo Norte e pode apenas ser acordado em duas situações: (1) pelas nove renas que regressam de férias do Pacífico Sul ou (2) por alguns dos elfos que estão com dificuldades a construir brinquedos. Para que o Pai Natal consiga dormir, os elfos apenas o acordam quando três deles tiverem problemas. Enquanto três elfos estiverem com o Pai Natal, se houver outros (três) elfos com problemas, estes devem esperar que os primeiros saiam. Se o Pai Natal acordar e descobrir que tem não só elfos à espera, mas também as renas, os elfos têm que esperar para depois do Natal, pois é necessário ir preparar o trenó. Só a última rena a chegar é que vai avisar o Pai Natal. Até que a última rena chegue, as outras renas esperam num covil até serem atreladas no trenó.

Assim, depois das nove renas chegarem, o Pai Natal invoca o método `PrepararTreno()` e as então as nove renas devem invocar `AtrelarATreno()`. Quando o terceiro elfo chega, o Pai Natal invoca `AjudarOsElfos()`. Concorrentemente, os três elfos devem invocar `PedirAjuda()`. Todos os três elfos devem ser ajudados, invocando `SerAjudado()`, antes que outros elfos entrem. O Pai Natal deve correr num ciclo de forma a ajudar o conjunto de elfos que conseguir. Assumimos que existem exactamente nove renas, mas pode existir um qualquer número de elfos. Implemente as tarefas que implementam o Pai Natal, as Renas e os Elfos.

D. Objectos avançados de sincronização

---

19. [Barreira] Uma barreira é um objecto que oferece duas primitivas: `associar` e `chegar`. Os processos chamam `associar` para se associarem à barreira. Quando um processo chama `chegar`, bloqueia-se até que todos os outros processos associados à barreira chamem `chegar`. Quando o último processo chamar `chegar`, todos os processos que estiveram associados à barreira e que fizeram `chegar` serão desbloqueados.

Programe o tipo de dados barreira e as as rotinas `associar` e `chegar`, usando semáforos. Admita que os processos são bem comportados, como representados na figura seguinte, ou seja, primeiro chamam `associar` e só esses é que depois chamam `chegar`.

```
barreira_t b;

init_i(){
    ...
    associar(&b); // associa-se à barreira
    ...
}

proc_i(){
    ...
    chegar(&b); // bloqueia-se até que todos chamem chegar
    ...         // todos continuam "ao mesmo tempo"
}
```

---

20. [Monitor] Algumas linguagens de alto nível, como por exemplo o Java e o C#, baseiam o seu modelo de sincronização em *monitores* que são mais fáceis de programar e mais seguros de utilizar do que semáforos. Um monitor define uma região de código protegida por um trinco associado ao monitor. Esse código é delimitado pelas primitivas do monitor `Entrar` e `Sair` e assegura automaticamente que esse código é executado em exclusão mútua. Objectos especiais chamados *variáveis de condição* disponibilizam primitivas

que permitem bloquear tarefas dentro de monitores (`Aguardar`) e assinalar tarefas bloqueadas em monitores (`Notificar`). A descrição das funções relevantes é a seguinte:

- `CriaMonitor() -> monitor_t`: cria e inicializa um monitor;
- `CriaVarCondicao(monitor_t monitor) -> varcond_t`: cria uma variável de condição associada ao monitor específico;
- `Entrar(monitor_t monitor)`: entra no monitor e obtém o trinco do monitor;
- `Aguarda(varcond_t cvar)`: bloqueia o processo libertando o trinco do monitor;
- `Notificar(varcond_t cvar)`: se ninguém está a aguardar, não faz nada (i.e. a variável de condição não guarda memória), caso contrário acorda uma das tarefas em espera. A tarefa que invocou `Notificar` continua na posse do trinco do monitor;
- `Sair(monitor_t monitor)`: sai do monitor e liberta o trinco do monitor.

Como exemplo, o programa seguinte representa uma implementação do problema dos produtores/consumidores usando monitores e variáveis de condição.

```
Item_t buffer[N];
int iprods = 0, icons = 0, count = 0;
monitor_t monitor = CriaMonitor();
varcond_t vazio = CriaVarCondicao(monitor);
varcond_t cheio = CriaVarCondicao(monitor);
```

```
void DepositaItem(Item_t item)
{
    Entra(monitor);
    while(cont == N)
    {
        Aguardar(cheio);
    }
    buffer[iprods] = item;
    iprods = (iprods + 1) % N
    count++;
    Notificar(vazio);
    Sair(monitor);
}
```

```
void RetiraItem(Item_t* item)
{
    Entra(monitor);
    while(cont == 0)
    {
        Aguardar(vazio);
    }
    *item = buffer[icons];
    icons = (icons + 1) % N
    count--;
    Notificar(cheio);
    Sair(monitor);
}
```

a. Programe as estruturas de dados `monitor_t` e `varcond_t`, e implemente as respectivas funções de manipulação de acordo com a semântica descrita usando semáforos e/ou trincos.

b. Volte a implementar o exercício **4.a** usando monitores e variáveis de condição com a semântica aqui descrita.