Aula 15 de FSO

José A. Cardoso e Cunha
DI-FCT/UNL

Este texto resume o conteúdo da aula teórica.

1 Objectivo

Objectivo da aula: Problema de exclusão mútua e regiões críticas. Instrução TEST-AND-SET. Semáforos: definição e exemplos.

2 Multiprocessador de memória partilhada

A figura 1 mostra o esquema de uma arquitectura de um multiprocessador com uma memória física partilhada.

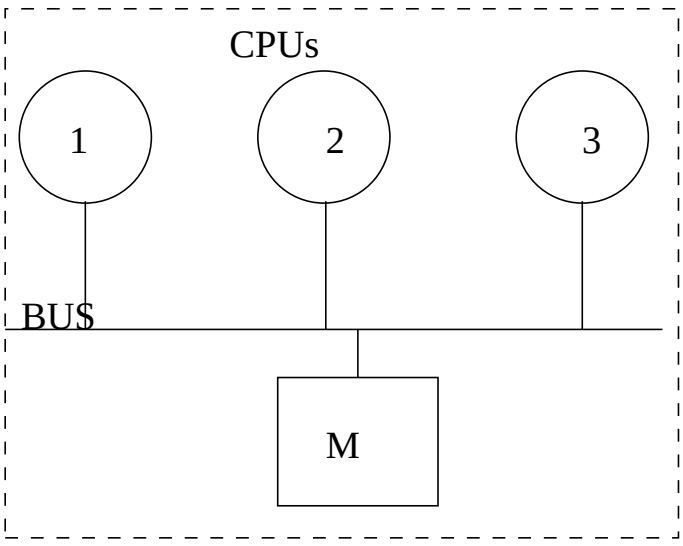

Figura 1: Multiprocessador de memória partilhada.

Se o sistema só tem um CPU, temos a arquitectura tradicional do computador sequencial de von Neumann, ao qual também se aplica a instrução máquina TEST-AND-SET, que a seguir se apresenta.

3 A instrução TEST-AND-SET

Esta instrução máquina, isto é, implementada como uma das instruções do processador (CPU) de um computador, vai-nos permitir, de uma forma atómica ou indivisível, ler o valor corrente de uma célula de memória e afectar-lhe um novo valor. Isto resolve o nosso problema de implementação do protocolo de entrada numa região crítica (veja a aula anterior). Nuns processadores, a instrução chama-se (impropriamente) TEST-AND-SET, enquanto noutros se chama EXCHANGE (por fazer a troca, entre si, dos conteúdos de uma célula de memória e de um registador do CPU).

Em pseudo-código, a instrução TEST-AND-SET descreve-se assim:

```
function TEST-AND-SET (var x: boolean): boolean;
(*indivisivel por hardware*)
begin
   TEST-AND-SET := x;
   x := true
end;
```

Ser indivisível quer dizer que, se dois processos concorrentes invocarem esta instrução só um deles conseguirá executá-la, de cada vez. No caso de um sistema multiprocessador com memória partilhada, em que a célula x está localizada na memória partilhada, como esta é acedida através do BUS comum a todos os CPUs, é fácil garantir a indivisibilidade, fechando (por hardware) o acesso ao BUS, durante a execução da instrução.

Isto permite a um CPU obter o valor corrente da célula x, deixando lá um novo valor e, depois, pode testar localmente se deve continuar a tentar ou se pode entrar na região crítica. As funções do protocolo de entrada e saída são assim descritas em pseudo-código:

```
function ENTRADA (var x: boolean);
 begin
   repeat
```

```
   until not TEST-AND-SET(x)
 end;

function SAIDA (var x: boolean);
 begin
   x := false
 end;
```

Imagine, por exemplo, que no início x está a *false*, indicando nenhum processo está na sua região crítica. Se houver dois processos a tentarem, concorrentemente, executar a instrução TEST-AND-SET(x), num sistema com dois CPUs, o BUS só atende o pedido de um deles, para aceder ao BUS (o hardware de controlo do acesso ao BUS tem um mecanismo de arbitragem, baseado em linhas de controlo para indicar *bus livre*, *pedido de acesso*, *libertação de bus*). O processo que não conseguiu, por o outro ter sido atendido primeiro, 'vê' o seu CPU ficar num estado de suspensão a nível do hardware (estado dito *HOLD ou WAIT*, que corresponde, de facto, à suspensão do ciclo de execução *fetch; execute* do CPU, à espera da autorização de acesso ao bus).

Entretanto o primeiro processo obtém o valor de x, que era *false*. A sua instrução TEST-AND-SET deixa em x o valor *true* e devolve *false*. Assim, no ciclo *repeat* do protocolo de ENTRADA, este processo vai conseguir continuar, entrando na sua região crítica.

Enquanto isto, o segundo processo, logo que o primeiro acaba a instrução TEST-AND-SET, vê o bus ser libertado e começa a executar a sua. Mas agora vai obter x com o valor *true* e deixa lá *true* (tanto faz, porque já estava). Mas este processo, no seu protocolo de ENTRADA, não vai conseguir sair do seu ciclo de *repeat*, enquanto o primeiro processo não executar o seu protocolo de SAIDA, pondo x a *false*, ao sair da sua região crítica.

4 Comentários sobre a instrução TEST-AND-SET

Esta é uma instrução existente em praticamente todos os processadores modernos.

Num sistema de múltiplos CPUs (multiprocessador), os processos ficam em espera activa, enquanto não puderem entrar na região crítica. Aplicam-se as observações da aula anterior, quanto ao número de processos ser igual ou menor do que o número de CPUs.

Em cada ciclo do seu protocolo de ENTRADA, os processos precisam de ir à memória consultar o valor de x. Isto pode originar, se houver muitos

CPUs, muitos conflitos de acesso ao BUS, com correspondentes atrasos dos processos. Na prática, isto limita o número típico de CPUs que se têm em arquitecturas destas, a valores da ordem de 16 - 32, mas nãp mais.

Apesar disso, há possibilidades de reduzir os conflitos. Se cada CPU tiver uma memória CACHE privada (isto é local, acessível sem ser preciso aceder ao BUS de sistema), é possível aliviar os conflitos. Um processo, da primeira vez que acede a x, traz uma cópia para a sua CACHE. A partir daí, no seu ciclo de *repeat*, nunca mais vai à memória partilhada ler x, usando sempre a sua cópia da CACHE. Logo que um processo actualizar x, todos os controladores de CACHE são avisados de que há um novo valor de x, e as cópias das CACHES são descartadas, pelo que os processos vão, um de cada vez, ler o novo valor à memória.

Num sistema de um único CPU (monoprocessador), mas executando múltiplos processos concorrentes, em multiprogramação, sob o controlo de um SO, a utilização do TEST-AND-SET origina situações em que cada processo, ao tentar testar o x, no seu ciclo de *repeat* fica em execução durante todo o tempo do TIME-SLICE, que o SO lhe atribuiu. Durante este tempo, se x estiver a *true*, ou seja, outro processo anteriormente entrou na sua região crítica e ainda não saiu, então o valor de x não é alterado, seguramente. Donde, o tempo do TIME-SLICE é, de facto, totalmente desperdiçado. Apesar de ineficiente, a solução é correcta, pois, ao fim do TIME-SLICE, este processo é obrigado a largar o CPU, para dar a vez de execução a outro processo.

No entanto, mais valia que o processo, que está a tentar entrar, mas não pode (por o recurso estar ocupado por outro processo), fosse posto num estado de bloqueio, numa fila de espera gerida pelo SO, até que, de facto, o outro processo, saindo da região crítica, lhe permitisse prosseguir.

É esta alternativa, que veremos na secções seguintes, sendo, geralmente, mais eficiente sempre que tenhamos menos CPUs do que processos.

5 Sincronização de processos baseada em mecanismos de bloqueio

Num SO que suporta multiprogramação, um processo pode passar do estado activo, em que o CPU está executando as instruções do programa desse processo, para o estado bloqueado, devido a ter de aguardar até que alguma condição seja satisfeita. Exemplos dessas condições são: aguardar a chegada de dados, quando se invoca uma operação *read* e os *buffers* de entrada estão vazios; aguardar até que um *buffer*, correntemente cheio, tenha espaço para

se inserirem novos elementos; aguardar a chegada de uma mensagem; etc..

Na década de 1960, quando os problemas de programação concorrente começaram a ser identificados, o investigador holandês E.W.Dijkstra inventou um mecanismo de sincronização de processos, a que deu o nome de *semáforo*. Trata-se de um mecanismo genérico, que pode ser aplicado a um grande diversidade de situações envolvendo processos concorrentes. Em particular, veremos a seguir a sua aplicação para resolver os problemas da exclusão mútua, do produtor-consumidor através de um *buffer*, dos leitores-escritores de uma base de dados, dos clientes-servidores, e muitos outros problemas.

A principal característica que distingue o semáforo das soluções anteriores do problema da exclusão mútua é o facto de o semáforo ser um mecanismo bloqueante, permitindo assim, evitar o desperdício de tempo, típico das soluções baseadas em espera activa.

6 Definição de semáforo

Um semáforo é definido como um tipo de dados abstracto, isto é, caracterizado pelos valores que pode assumir e pelas operações que o manipulam, mas não pela sua implementação.

Assim, um semáforo é definido por uma variável inteira, não negativa, dita *o valor do semáforo (s)*, e por três operações indivisíveis:

- inicializar: atribui um valor inicial ao semáforo (s_init);
- P: o processo invocador aguarda até que $s > 0$ e, então, decrementa s de uma unidade (P = **P**roberen, significa testar, em holandês);
- V: o processo invocador incrementa o valor (s) de uma unidade (V = **V**erhogen, significa incrementar, em holandês).

A cada momento, verifica-se a seguinte igualdade, dita o invariante do semáforo:

```
s = s_init + NopsV - NopsP
```

em que NopsV é o número de operações V completadas, e NopsP é o número de operações P completadas, até àquele momento.

A figura 2 ilustra o funcionamento do semáforo com base numa analogia com uma caixa com bolas, em que cada bola representa uma unidade do valor do semáforo.

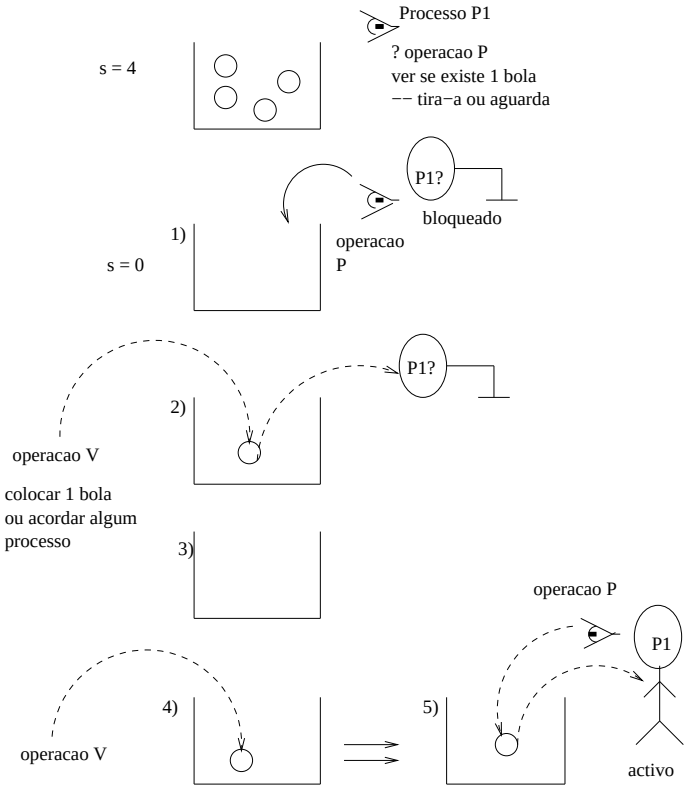

Figura 2: Semáforos.

A figura ilustra, no caso da parte 4), a capacidade de memória do semáforo: uma operação V, feita sobre um semáforo 'vazio', isto é, com s = 0, deixa lá o valor s = 1. Quando, mais tarde, um processo invocar uma operação P, esta não bloqueia o processo, pelo que a execução deste prossegue, como se o semáforo tivesse armazenado temporariamente um 'sinal' de sincronização para o processo.

Se um processo invocar a operação P quando s = 0, o processo bloqueia-se, até que um outro processo invoque V. Neste caso, o efeito de V, é correspondente ao de incrementar o valor de s, para 1, mas é 'imediatamente' seguido pelo completar da operação P, que estava bloqueada, pelo que é como se o valor do semáforo não chegasse a ser modificado. Veja, na figura, a parte 2), em que o efeito de V é assimilado à passagem directa da bola ao processo que estava bloqueado no P.

A indivisibilidade destas operações tem de ser garantida pela implementação.

Significa que, se dois ou mais processos concorrentes invocarem um P ou um V sobre um mesmo semáforo, então é executada uma operação de cada vez. Ou seja, o código das operações P e V corresponde a regiões críticas. De igual modo, se houver dois ou mais processos bloqueados num semáforo, e um outro processo invocar um V, apenas um daqueles processos é bloqueado.

7 Exemplo de implementação de semáforos ao nível do SO

Ao nível do SO, os semáforos podem ser implementados pelo núcleo e as operações *inicializar, P e V* disponibilizadas sob a forma de chamadas ao sistema. É o que acontece em SO, como o UNIX (System V, conjunto de funções IPC - Inter Process Communication) (veja no manual e nas aulas práticas, funções *semop*). O facto de serem implementadas pelo núcleo, torna estas operações indivisíveis.

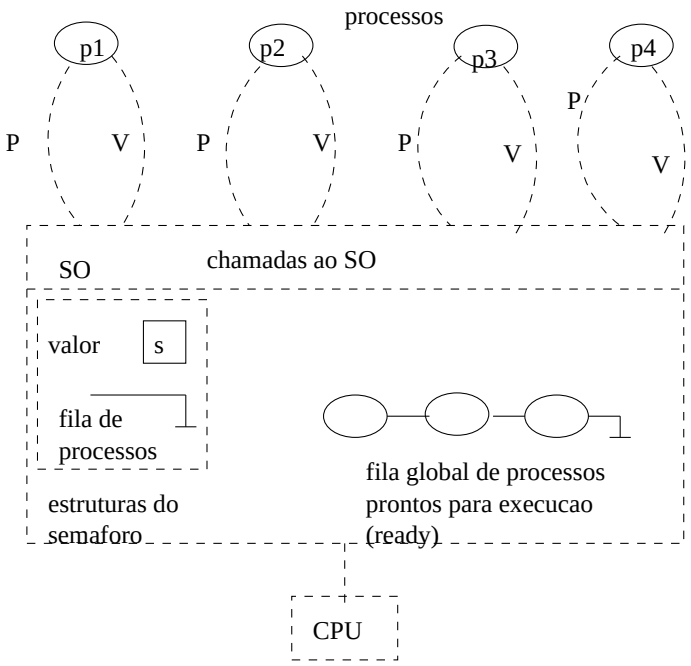

Figura 3: Implementação de semáforos.

Internamente ao núcleo vêem-se na figura 3 as duas estruturas básicas geridas por cada semáforo: uma variável inteira, representando o valor (s), e

uma fila de espera de processos, inicialmente vazia, na qual os processos que invocam P quando s = 0, ficam bloqueados até que outros façam V.

O pseudo-código das operações P e V é dado na figura 4.

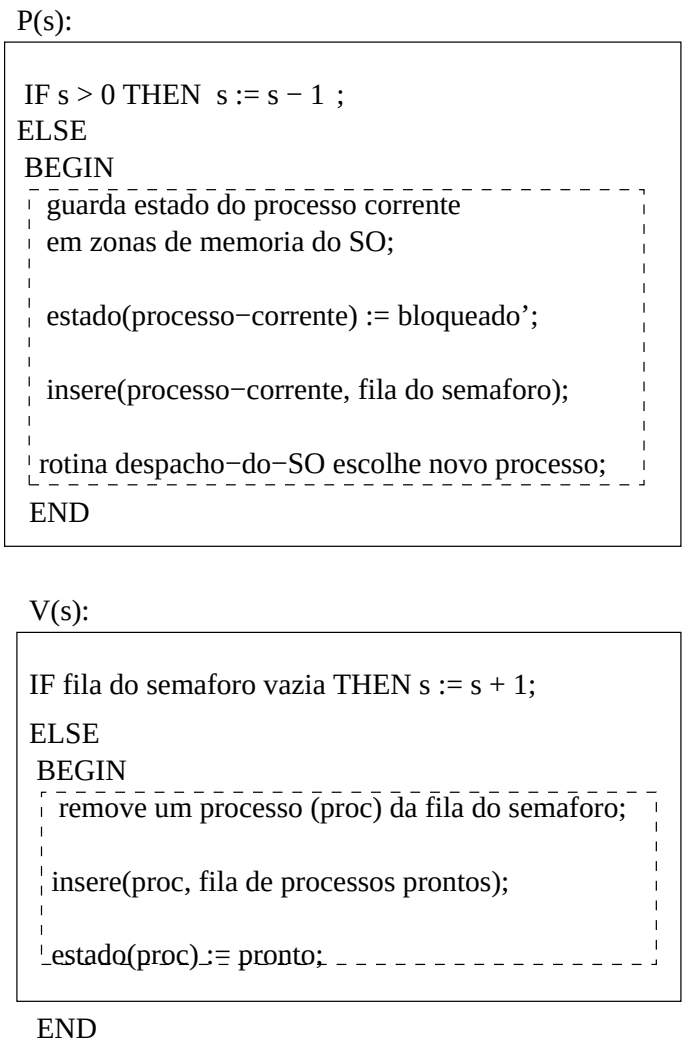

Figura 4: Operações P e V sobre semáforos.

A figura 5 ilustra as transições de estado dos processos, em função das operações P e V.

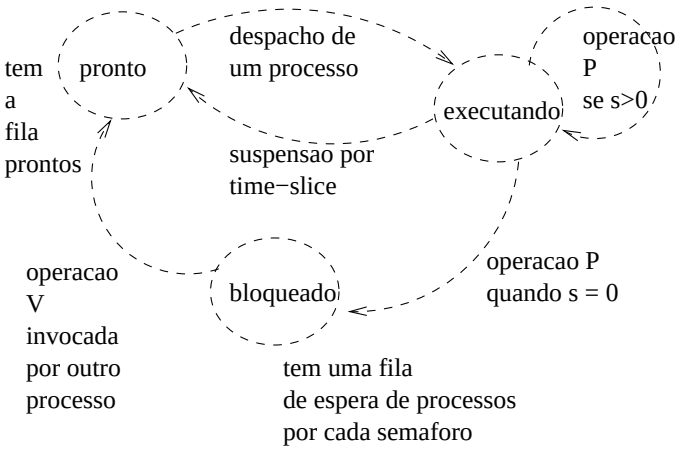

Figura 5: Transições de estados devidas aos semáforos.

8 Exemplo1: exclusão mútua

O problema das regiões críticas, regiões de código de processos concorrentes, que se excluem mutuamente, resolve-se muito facilmente com semáforos. Inicialize-se um semáforo de nome *exmut* a 1. Em cada processo concorrente, faça-se o seguinte:

```
P(exmut);
regiao critica
V(exmut);
```

9 Exemplo2: produtor-consumidor através de um *buffer*

Considere que dois ou mais processos concorrentes comunicam entre si através de um *buffer* em memória partilhada.

9.1 Caso de *buffer* ilimitado

Se o *buffer* é ilimitado, os processos que colocam elementos no buffer - ditos produtores -, nunca se bloqueiam pois o buffer nunca fica cheio. Contudo, os processos que retiram elementos do buffer - ditos consumidores -, podem ter de ser bloquear quando tentam retirar elementos, mas o buffer está vazio

(por exemplo, logo que se iniciam os processos, o buffer está vazio, e há que garantir que um consumidor, que tenha sido mais rápido na tentiva de retirar elementos, seja posto a aguardar, até que um produtor coloque algum elemento no buffer).

Por outro lado, se há múltiplos produtores e consumidores, há que garantir exclusão mútua no acesso à casas do *buffer*.

Para isso temos dois semáforos: o semáforo *exmut*, inicializado a 1, protege o acesso às casas do buffer; o semáforo *s_preenchido*, inicializado a 0, vai servir de contador do número de casas do buffer, correntemente preenchido.

O pseudo-código dos produtores e dos consumidores é dado na figura 6

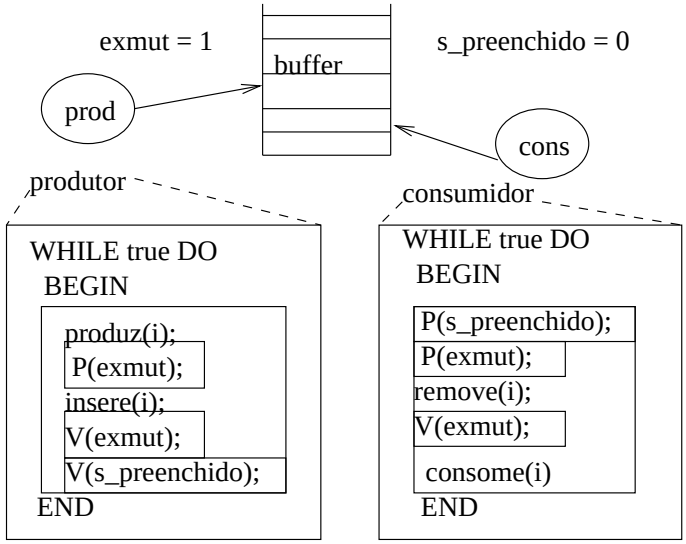

Figura 6: *Buffer* ilimitado.

9.2 Caso de *buffer* limitado

Se o *buffer* é limitado, os processos que colocam elementos no buffer - ditos produtores -, também se podem bloquear pois o buffer pode ficar cheio. Se o buffer tiver uma capacidade de N elementos, um novo semáforo *s_livres*, inicializado a N e representando o número de casas vazias do buffer a cada momento, permite resolver o problema de forma simples, como se indica na figura 7.

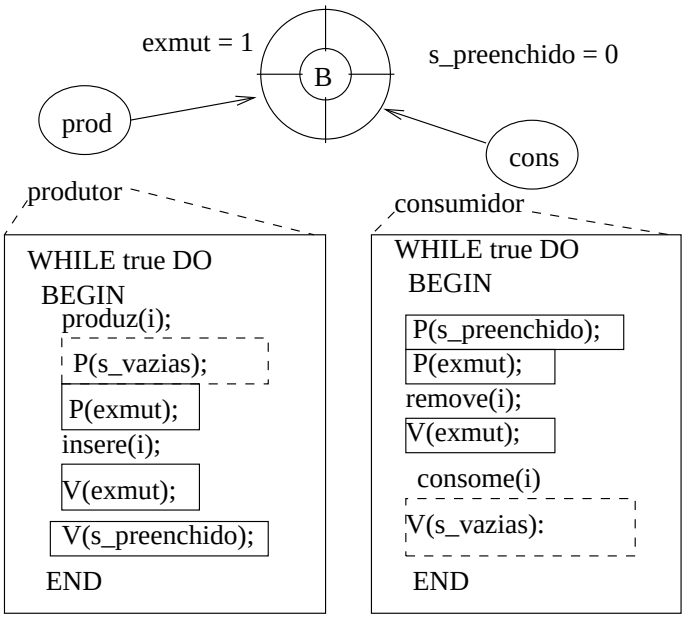

Figura 7: *Buffer* limitado.

Este exemplo ilustra a utilização de semáforos como contadores de recursos, permitindo que os processos se sincronizem facilmente entre si, através das operações V() que, incrementam os contadores, correspondendo a 'libertar' ou a 'gerar' recursos, e das operações P() que tentam decrementar os contadores, correspondendo a 'pedir' ou 'reservar' os recursos.

10 Exemplo3: troca síncrona de sinais entre processos

Um semáforo, inicializado a 0, pode ser utilizador para fazer um processo aguardar por um sinal de activação gerado por outro processo.

Por exemplo:

```
semaforo s  inicialmente 0;

processo P1                    processo P2
.....                          .....

P(s); (*1*)                    .....

                               V(s);

.....                          .....
```

Se P1 for o primeiro a aceder ao semáforo, bloqueia-se até que P2 execute o V(s). Caso contrário, P2 incrementa o valor do semáforo e, quando P1 invocar P(s) já não se bloqueia. Em qualquer dos casos, a execução de P1, a partir do ponto (1), fica condicionada ao sinal gerado por P2.

Esta troca de sinais diz-se *síncrona* porque P1 tem de invocar explicitamente a operação P(s), na qual se bloqueia potencialmente aguardando a execução de V(s).

11 Exemplo4: sincronização por semáforos no controlo de operações de entrada e saída

Os semáforos, bem como outros mecanismos de sincronização, permitem oferecer aos processos concorrentes que se executam sob o controlo do SO, um ambiente em que os mecanismos de interrupção de programas, detectados ao nível da máquina hardware, são escondidos e, quando necessário, convenientemente transformados em indicações dadas aos programas, a um nível lógico e conforme a linguagem de programação utilizada.

A figura 8 ilustra esta virtualização, no que se refere à sincronização.

Como exemplo, vejamos o caso das operações de leitura ou de escrita de dados em dispositivos periféricos ('ficheiros', no caso do Unix). Um programador numa linguagem como o Pascal ou o C, quando invoca uma instrução READ, espera que, logo após o retorno, os dados pedidos estejam

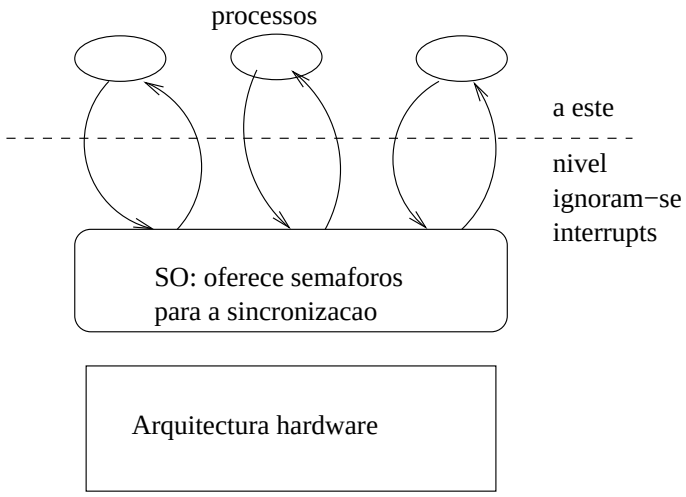

Figura 8: Virtualização das interrupções *hardware*.

disponíveis. Isto significa que, caso os dados não estejam acessíveis ao programa, no momento em que este invoca a operação, o processo será bloqueado e só retomará a execução logo que os dados vierem. Este modo de operação diz-se síncrono ou bloqueante e é o mais conveniente, em geral, para um programador numa linguagem convencional, que assume um modo sequencial de execução. Este modo também 'facilita' a depuração de programas (*debugging*, pois permite 'pontos de controlo' nos quais se sabe assumir qual o estado correcto do processo.

Noutros contextos, por exemplo, quando se programam os serviços oferecidos por um SO, o modo de operação síncrono pode ser considerado demasiado restritivo. Por exemplo, se estivermos a programar um servidor que deva poder atender diversos tipos de pedidos e tenha, além disso, tarefas a executar, enquanto outros pedidos não se cumprem, pode interessar dispor de um modo de operação assíncrono ou não bloqueante. Também no caso de um programa que efectue operações de leitura, que podem demorar, mas o programa tem outras tarefas para executar, convém ter a possibilidade de o programador especificar, quando invoca uma operação READ, se quer esperar, bloqueando-se, pelo resultado, ou se não quer esperar, retornando de imediato, mas sem os dados disponíveis. Neste segundo caso, portanto, terá o programa de, mais tarde, testar se os dados já estarão disponíveis, em geral com base noutra primitiva.

Vejamos como se pode fazer isto com semáforos.

A figura 9 ilustra a interacção de um processo utilizador, quando invoca uma operação de entrada e saída (DO_IO), em modo síncrono ou em modo assíncrono.

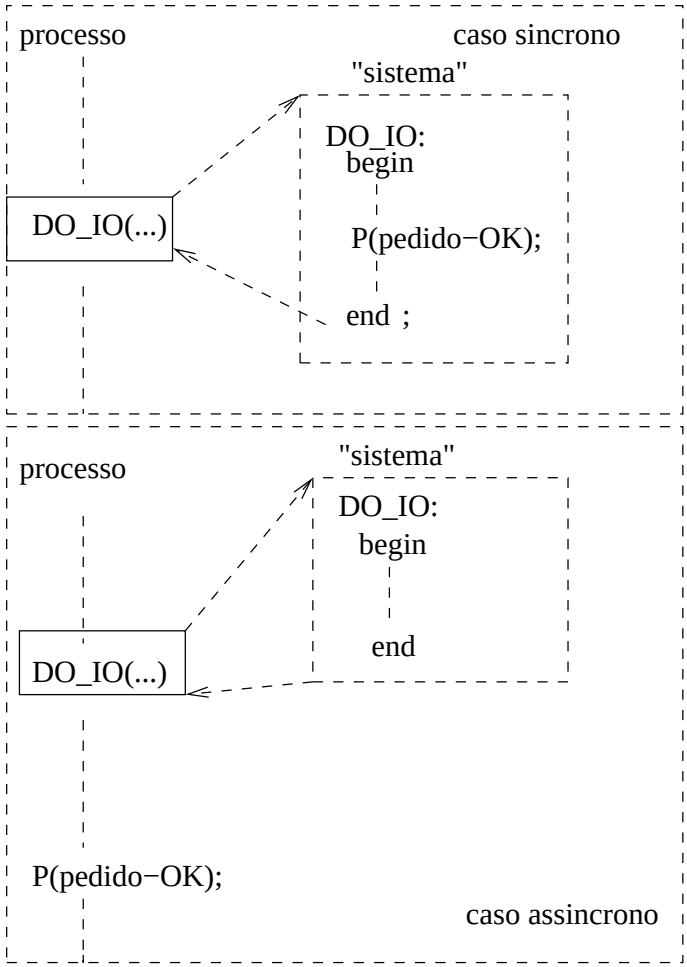

Figura 9: Invocação de uma operação de entrada/saída.

Note-se a utilização de um semáforo *pedido-OK*, privado ao processo utilizador (isto é, no qual só este pode fazer P()), inicialmente 0, e que serve para aguardar o fim da operação pedida. No caso síncrono, a espera é escondida no interior da rotina DO_IO (seja interna ao SO ou a uma biblioteca de suporte).

No caso assíncrono, a rotina DO_IO apenas desencadeia o pedido, internamente

ao SO, e o retorno dá-se, 'de imediato' ao processo invocador, o qual fica responsável por testar o fim da operação mais tarde, invocando explicitamente uma operação P(). A este respeito, a utilização de semáforos não é muito flexível, pois, quando o processo quiser testar o fim da operação, só poderá fazê-lo através de um P(), com o efeito possível de bloquear o processo, no caso de a operação ainda não se ter bloqueado. Para este efeito, gostaríamos de dispor de uma variante 'não bloqueante' da operação P(), tal que o processo pudesse evitar bloquear-se no caso de o valor do semáforo ainda ser 0. Contudo, na definição original, Dijkstra define a operação P como só sendo completada após ter conseguido decrementar o valor do semáforo. Assim, aquela variante de P() iria contra a definição original.

11.1 Variantes nas operações sobre semáforos

Na prática, as implementações de semáforos a nível de SO, acrescentaram variantes às definições das primitivas P e V, tal como introduzidas por Dijkstra. Por exemplo, no conjunto de primitivas IPC do Unix System V, os semáforos permitem:

- consultar o valor corrente do semáforo, sem se bloquear;
- efectuar múltiplas operações P ou V, sobre vectores de semáforos;
- etc. (veja o manual)

Algumas variantes são interessantes, por exemplo, permitir múltiplas operações P sobre múltiplos semáforos, realizadas como operações indivisíveis, pode ser utilizador para prevenir situações de *deadlock*. Veja-se o exemplo seguinte, em que s1 e s2 são dois semáforos inicializados a 1, e a numeração indica uma possível ordem de execução das acções:

```
processo P1                  processo P2

P(s1);   (1)                 P(s2);   (2)
P(s2);   (3)                 P(s1);   (4)
.......                      ........
```

Ambos os processos ficam bloqueados eternamente: P1 em (3), P2 em (4).

Se dispusermos de uma operação P(s1, s2), tal que equivalha a efectuar duas operações P, de forma indivisível (quer dizer que, ou consegue realizar ambos os P(), ou então bloqueia-se em realizar nenhum), não ocorrem situações de *deadlock*.

Quanto às variantes que permitem consultar o valor corrente de um semáforo, sem o processo se bloquear, a sua utilização é perigosa e exige cuidados especiais. De facto, com essa variante, dois processos correntes podem consultar o mesmo valor de um semáforo e, concorrentemente, desencadearem acções incompatíveis. Foi exactamente este problema que a definição de semáforo resolveu, quando exigiu que as operações permitissem aguardar e decrementar, de forma indivisível. Portanto, esta variante é altamente desaconselhada.

11.2 Como simular comportamento assíncrono, com semáforos e processos?

Vimos antes que, com as operações P(), no modo assíncrono, o processo utilizador bloqueia, logo que tente testar se a operação pedida já se completou. Utilizando a chamada ao SO fork() do Unix, é possível que o processo, de cada vez que quer testar, delegue num processo filho essa tarefa, e depois só precisa de comunicar com o filho para saber se a operação já acabou ou não. A figura 10 ilustra esta solução.

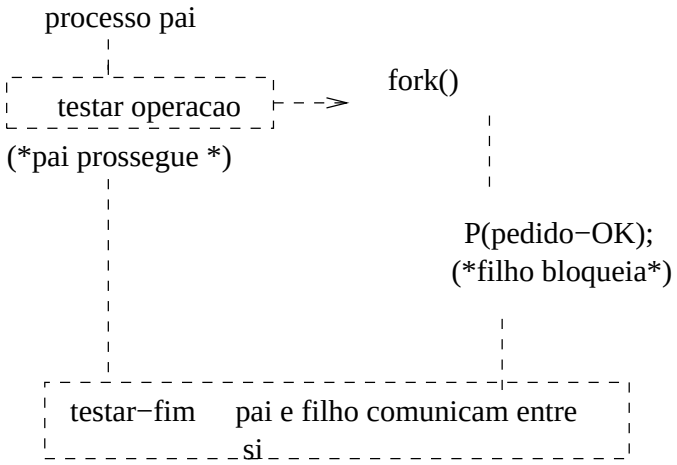

Figura 10: Comportamento assíncrono utilizando semáforos e processos.

Apesar de funcionar, esta solução, em geral, não é a mais conveniente,

pelo que o mais habitual é encontrarmos, em diversas chamadas ao SO, os dois possíveis modos de operação: bloqueante e não bloqueante.

Alguns exemplos, no caso do Unix:

- *read* de um *pipe*, sem bloqueio no caso de estar vazio; utilizando a chamada ao SO *select*;
- receber uma mensagem de uma caixa de correio, sem bloqueio no caso de não haver mensagens pendentes;
- *waitpid()*, aguardar a terminação de um processo filho, criado por *fork*, sem bloqueio no caso do filho ainda não ter terminado.

12 Exemplo5: sincronização do processo utilizador, do processo servidor e do controlador de um dispositivo periférico

A figura 11 mostra o lado do 'sistema' correspondente ao serviço de um pedido de *DO_IO*.

Na figura, a estrutura BP (bloco-pedido) descreve o tipo e os argumentos da operação pedida e inclui informação adicional. Nomeadamente, cada bloco inclui o nome de um semáforo *pedido-OK*, no qual a rotina *DO_IO* ficará a aguardar o fim do tratamento do pedido.

Como a figura ilustra, cada pedido é inserido numa fila de pedidos (tipo BP), para que um processo controlador possa tratar os pedidos. Para este efeito, podemos ter, associados à fila, dois semáforos:

- *pedidos*: conta o número de pedidos pendentes na fila;
- *fila-exmut*: garante exclusão mútua no acesso à fila, ou seja garante que as operações *inserir* (invocada por *DO_IO*) e *remover* (invocada pelo controlador) são regiões críticas.

A figura 12 mostra o esquema simplificado do processo controlador.

O semáforo *fim-op*, inicializado a 0, permite ao controlador aguardar o fim da operação física do periférico, através de uma operação P(fim-op). O dispositivo periférico, ao terminar a operação, gera um pedido de interrupção *hardware*, o qual desencadeia a invocação da correspondente *rotina de serviço de interrupção*. Esta, depois de efectuar as acções adequadas, indica ao controlador o fim da operação, através de uma operação V(fim-op). O

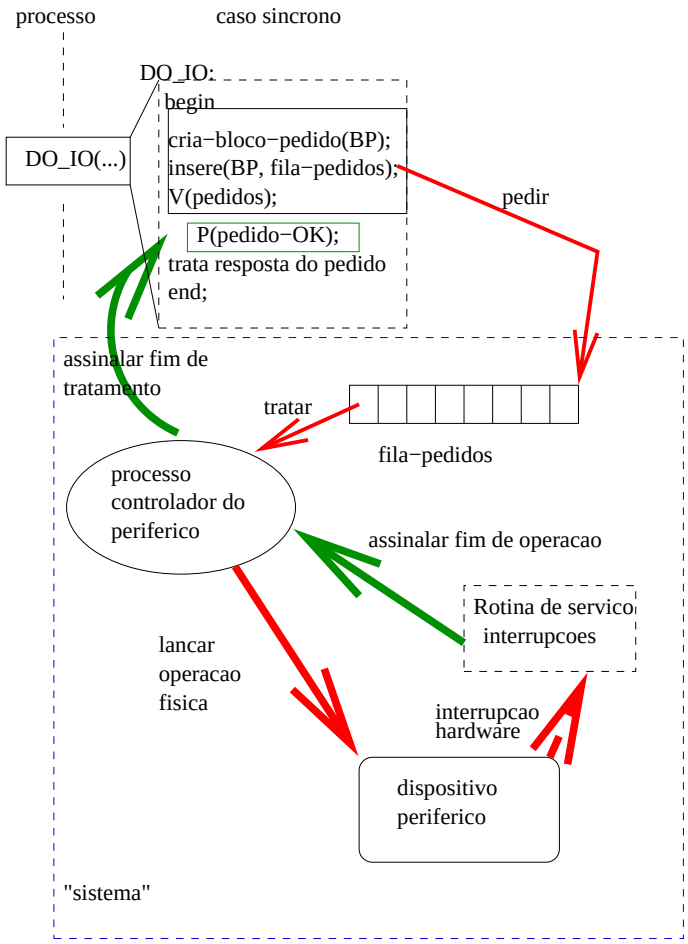

Figura 11: Arquitectura interna do serviço de *DO_IO*.

controlador pode, então, assinalar o fim do tratamento do pedido ao processo utilizador, cuja execução ficara bloqueada em P(pedidos-OK), no interior da rotina *DO_IO*. Por isso, o controlador assinala isto, fazendo V(pedidos-OK).

Observações:

- ilustrou-se um caso de um servidor sequencial ou iterativo, que só trata um pedido de cada vez; variantes, possivelmente mais eficazes, em função dos tipos de pedidos, serão vistas adiante;
- omitiram-se os pormenores dos tratamentos intermédios, incluindo os

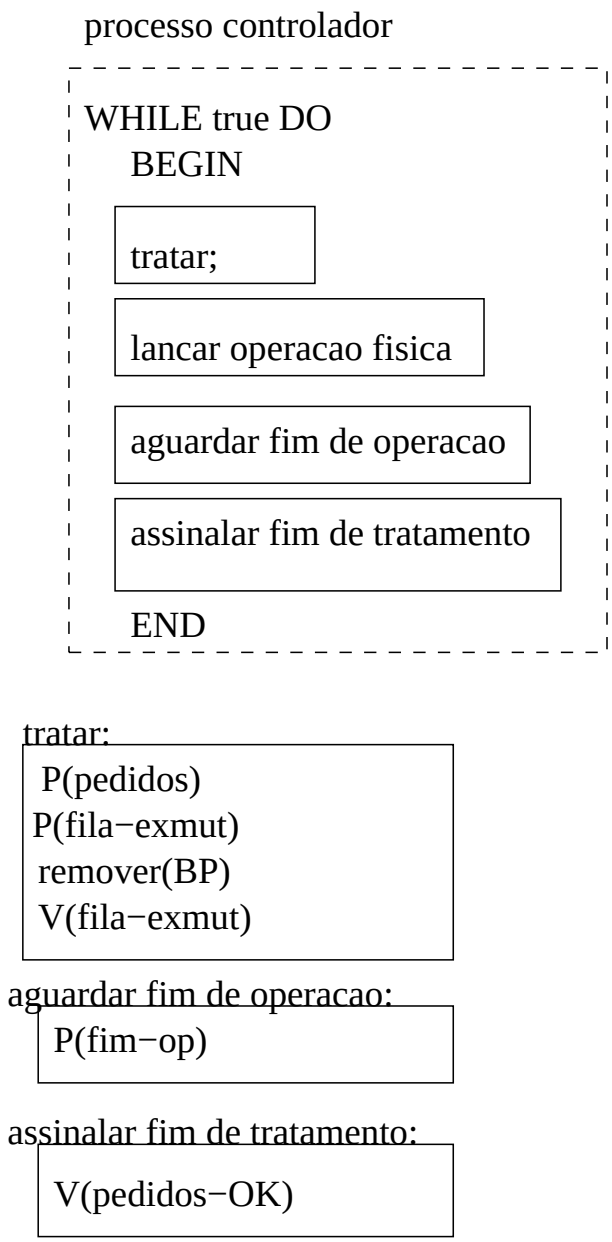

Figura 12: Processo controlador.

das verificações de erros;

- procurou-se ilustrar a expressividade dos semáforos, para se programarem diferentes tipos de interacções e sincronização entre processos;
- a arquitectura ilustrada, ainda que viável na prática, não pretende representar as soluções reais que se encontram nos SO existentes.

13 Exemplo6: escalonamento de processos em 'tempo real'

A figura 13 apresenta uma organização de processos concorrentes, em que, N processos (p1..PN), uma vez criados no início da execução (no 'arranque' da aplicação), ficam bloqueados em semáforos privados, sem1..semN, todos inicializados a 0.

Cada um destes processos terá associados, numa agenda de tarefas, dois atributos:

- o programa da tarefa a executar;
- a *deadline* ao fim da qual o processo deverá ser activado, para executar a sua tarefa (indicada por Ti, para o processo pi).

Este tipo de aplicação é habitual em sistemas de controlo em 'tempo real', nos quais, para além de tarefas que devem ser activadas de forma muito rápida, em reacção a eventos detectados no ambiente exterior ao computador, existe um conjunto de tarefas que devem ser activadas periodicamente, para, por exemplo, realizarem amostragens de valores sinais exteriores, armazenando-os para posterior processamento. São estas últimas tarefas (e não as primeiras, de natureza 'reactiva') que consideramos neste exemplo.

A *deadline* de cada tarefa é, neste exemplo, medida em termos de unidades de tempo (*tiques*) de um temporizador *hardware* (um contador digital programável), que, ao fim de cada tique, gera um pedido de interrupção *hardware*. A correspondente rotina de serviço de interrupção, invocada por cada tique, efectua uma operação V(sem0) num semáforo, no qual está bloqueado um processo escalonador p0, também criado no início, e cuja função é desencadear a activação dos processos pi.

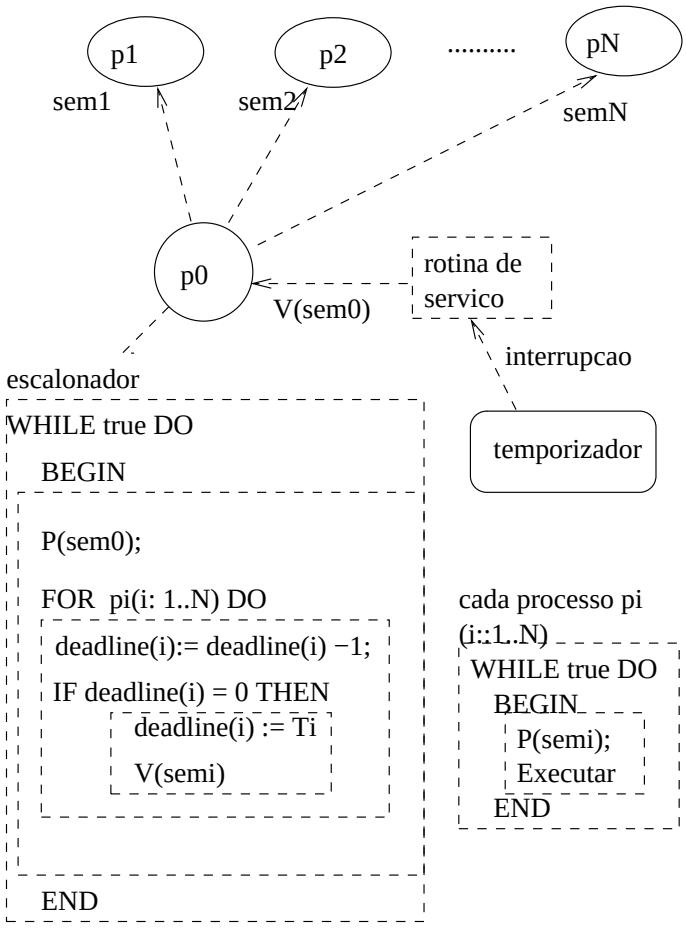

Figura 13: Escalonamento periódico de processos.

14 Exemplo7: o problema dos leitores e dos escritores

O problema dos leitores-escritores representa um padrão habitual de comportamento de processos concorrentes, que acedem a bases de dados partilhadas: os processos leitores, só fazendo consultas, podem partilhar, entre si, o acesso; os processos escritores, fazendo actualizações das estruturas partilhadas, exigem acesso exclusivo.

A figura 14 ilustra o esquema do problema.

As condições a satisfazer por uma boa solução são:

- um escritor exclui qualquer outro processo;

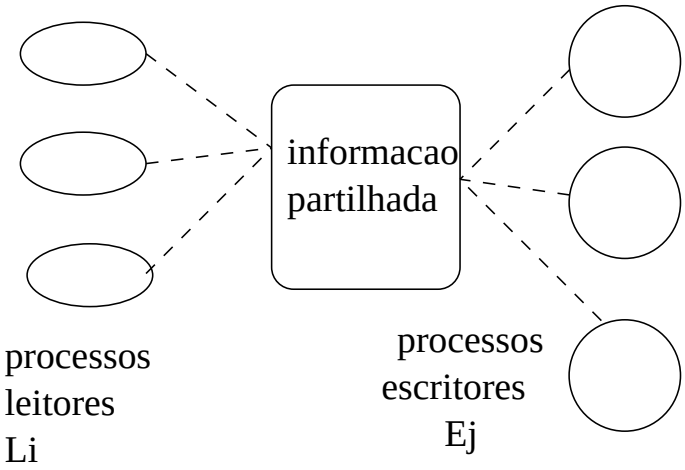

Figura 14: Leitores e escritores.

- qualquer número de leitores podem aceder concorrentemente;
- um fluxo de leitores Li não deve monopolizar o acesso, se houver um escritor à espera;
- idem, para um fluxo de escritores.

As duas últimas condições exprimem a ausência de privação (*starvation*), uma situação possível em sistemas concorrentes, quando a competição entre processos deixa algum processo eternamente privado de aceder a algum recurso, por os pedidos dos outros serem sempre atendidos prioritariamente. Uma forma de impedir a ocorrência de privação é impor uma disciplina de atendimento dos pedidos, baseada numa ordenação segundo a ordem de chegada. Deixa-se como exercício, a realização de uma solução que satisfaça as quatro condições indicadas.

Deixa-se também como exercício o estudo da correcção e das propriedades da seguinte solução do problema dos leitores-escritores (figura 15).

15 Exemplo 8: processos clientes e servidores

A figura 16 ilustra o esquema do problema.

O tipo de interacção de processos ditos *clientes* com processos ditos *servidores* é muito habitual e importante, porque é uma forma de suportar o

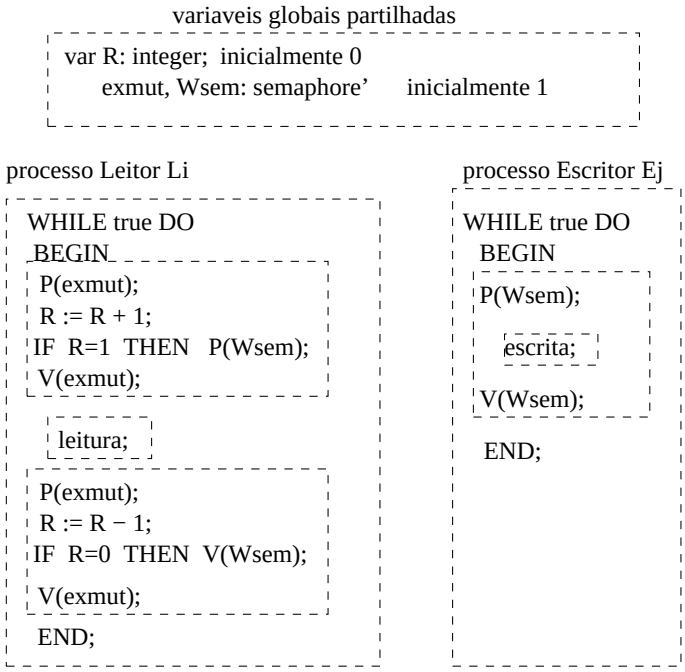

Figura 15: Leitores e escritores.

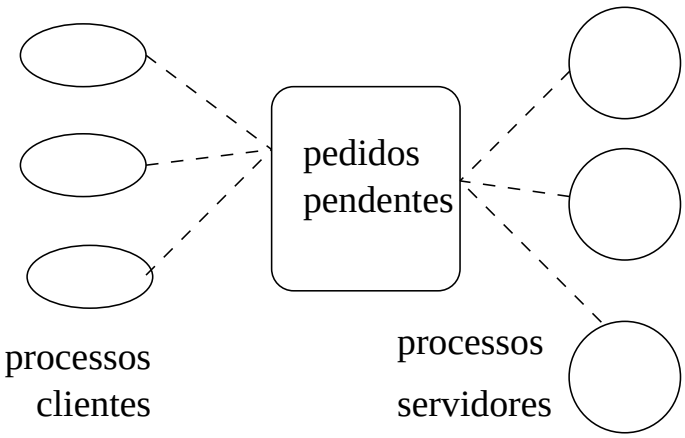

Figura 16: Clientes e servidores.

acesso, por parte dos processos utilizadores, a uma diversidades de serviços,

que são disponibilizados através de uma interface, invocada pelos clientes e implementada pelos servidores.

Neste exemplo, dado estarmos a estudar aplicações de semáforos que só são utilizáveis em sistemas de memória partilhada (de um ou múltiplos processadores), consideramos que a comunicação entre clientes e servidores se faz através de estruturas em memória partilhada. Nomeadamente, admite-se existir, em memória partilhada, uma fila única de pedidos, na qual cada cliente coloca um pedido, e da qual qualquer dos servidores retira um pedido, de cada vez, para seguidamente o tratar. Admite-se que todos os servidores são idênticos e capazes de tratar qualquer dos pedidos, dentro de um certo conjunto que caracteriza o tipo de serviço oferecido. Por exemplo, um serviço de ficheiros ofereceria as operações habituais de acesso a ficheiros. Admite-se ainda que cada servidor é sequencial e iterativo, isto é, trata apenas um pedido de cada vez. Em aulas seguintes veremos outras dimensões deste problema. Por agora, só queremos ilustrar outra utilização dos semáforos.

Cada cliente tem o seguinte pseudo-código:

```
P(exmut);
coloca-um-pedido-na-fila;
V(exmut);
V(pedidos);
...
```

O semáforo exmut é inicializado a 1 e protege o acesso à fila de pedidos, que está em memória partilhada, sendo, por isso, um semáforo global, ou seja utilizado por todos os processos clientes e servidores. O semáforo pedidos é inicializado a 0 e conta os pedidos pendentes na fila de pedidos.

Dependendo do tipo de pedido, cada cliente poderá prosseguir a execução, de forma assíncrona, logo que coloca o pedido na fila, ou então, ter de esperar por uma resposta, que lhe será comunicada por um servidor. Esta sincronização também se pode fazer por semáforos, o que se deixa aqui como exercício.

Cada servidor tem o seguinte pseudo-código:

```
WHILE true DO
BEGIN
  P(pedidos);
  P(exmut);
  remove-um-pedido-da-fila;
  V(exmut);
```

```
   trata-o-pedido();
   ...
END;
```

Cada servidor, conforme o tipo de pedido exigir resposta ou não, encarrega-se da resposta, após o que volta ao início do ciclo de serviço, aguardando novos pedidos.