Aula 16 de FSO

José A. Cardoso e Cunha
DI-FCT/UNL

Este texto resume o conteúdo da aula teórica.

1 Objectivo

Objectivo da aula: Considerações sobre os contextos de utilização da comunicação por memória partilhada versus a comunicação por mensagens. Simulação de mensagens com base em memória partilhada e semáforos, num sistema multiprogramado com memória física partilhada, ou com um só CPU. Comunicação por mensagens. Envio e recepção explícitos. Recepção implícita, com notificação assíncrona de mensagens. Nomeação directa e indirecta dos participantes na comunicação. Exemplos.

2 Modelos de comunicação

Esquematizam-se a seguir as diversas possibilidades:

- por memória partilhada: com partilha de parte de espaços de endereçamento, tornada comum a processos distntos;
- por memória distribuída: com cópia de valores entre espaços de enderecemento (processos) diferentes:
  - fluxos de *bytes*, como por exemplo no Unix, por meio de *pipes*, sequências de *bytes* sem delimitadores de fronteiras, são copiados de processo para processo, através de *buffers* geridos pelo SO;
  - comunicação por mensagens: ao nível do SO, vectores de caracteres são enviados encapsulados em mensagens, transformadas em unidades de transmissão, ao nivel das primitivas oferecidas como chamadas ao SO, para enviar e para receber;

– comunicação por invocação de pontos de entrada de processos ou de objectos: chamadas de procedimentos ou de métodos remotos (*remote procedure call* - *RPC* ou *remote method invocation* - *RMI*);

Se a memória partilhada tem a vantagem de permitir uma comunicação mais eficiente, por não envolver cópia de bytes, no entanto, só é permitida em arquitecturas de computadores de um CPU ou de múltiplos CPUS com memória física partilhada através de *bus* comum.

A comunicação por mensagens pode ser realizada, quer num sistema de um CPU, num sistema de múltiplos CPUs com memória partilhada, ou numa arquitectura de memória distribuída, conforme se viu em aulas aneteriores. Envolve sempre cópias de bytes, sendo a única comunicação possível a nível físico, no caso de uma arquitectura de memória distribuída (por exemplo, em redes de computadores).

Em sistemas monoprocessadores ou multiprocessadores de memória partilhada, pode optar-se por utilizar modelos de comunicação entre processos por memória partilhada ou por mensagens. Se os primeiros são mais eficientes, os segundos têm, no entanto, sincronização associada.

Por exemplo, ao tentar receber uma mensagem, um processo pode bloquear se não houver mensagens pendentes para entrega. Isto pode ser garantido directamente pela primitiva de recepção de mensagens. De igual modo, ao enviar, um processo pode originar, indirectamente, a activação de um processo que esteja bloqueando aguardando.

O mesmo não acontece no caso da comunicação por memória partilhada: um processo pode ler da zona partilhada, seja qual for o conteúdo presente, mesmo que sejam valores ainda não inicializados, isto é, escritos com valores significativos por outros processos. Neste último caso isto é assim porque as primitivas que, geralmente, são disponibilizadas ao nível do SO (exemplo do Unix System V), apenas permitem inicializar zonas de memória partilhada e depois projectá-las no mapa de memória dos processos, a partir do que, tudo se passa sobretudo através do *hardware* da unidade de transformação de endereços (mmu - memory management unit ou tlb - translation lookaside buffer) que dá acesso à tabela de páginas de cada processo. Todo o controlo e sincronização ficam a cargo do programador, por exemplo, utilizando mecanismos de sincronização como os semáforos.

Assim, em monoprocessadores ou multiprocessadores de memória partilhada, o compromisso é entre maior eficiência (por evitar cópias) versus necessidade de lidar explicitamente com a sincronização dos processos concorrentes.

Na prática é frequente as aplicações concorrentes recorrerem aos dois tipos de modelos, por exemplo, utilizando memória partilhada para dar

acesso eficiente a estruturas de dados comuns, e utilizando mensagens para outros efeitos, por exemplo, para interacções directas entre processos.

3 Implementação de comunicação por mensagens com base em semáforos e memória partilhada

A figura 1 mostra como se pode implementar a comunicação por mensagens, num monoprocessador ou multiprocessador de memória partilhada, utilizando semáforos como base.

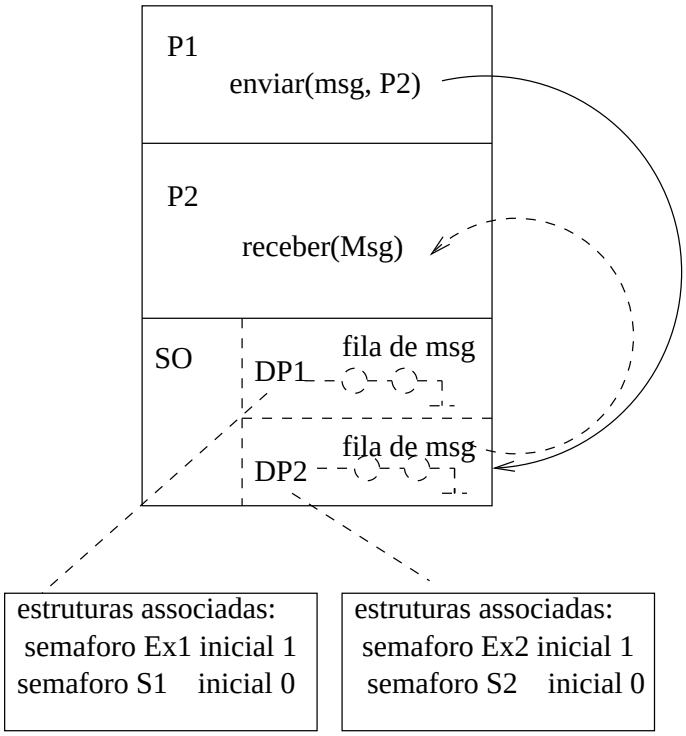

Figura 1: Mensagens com base em semáforos.

Na figura, admite-se uma primitiva *enviar(mensagem, processo)*, que envia uma mensagem directamente para o processo indicado. A implementação, a nível do SO, tem um descritor associado a cada processo, onde, para além do estado corrente do processo e de outra informação sobre o seu contexto de execução, existe uma fila de mensagens pendentes, cujo acesso é controlado

através de dois semáforos, um de exclusão mútua, outro utilizado como contador de mensagens.

```
enviar(mensagem m, processo pi):
 P(Exi);
 inserir(m, fila de mensagens de pi);
 V(Exi);
 V(Si);
```

A operação receber, que neste exemplo se assume da forma *receber(Mensagem)*, significando que um processo pj recebe uma mensagem de qualquer processo, sem indicar qual, pode ser assim realizada:

```
receber(mensagem M):
 P(Sj);
 P(Exj);
 remover(M, fila de mensagens de pj);
 V(Exj);
```

Este exemplo pretendeu apenas ilustrar como se pode implementar a comunicação por mensagens, num sistema *centralizado*, isto é, gerido por um SO único, que controla o acesso a uma memória partilhada, na qual se encontram os mapas de memória de todos os processos concorrentes, bem com as zonas de memória do próprio SO.

Nas secções seguintes, ver-se-ão as múltiplas dimensões da comunicação por mensagens, do ponto de vista lógico, isto é, do programador de aplicações, que acede as chamadas ao SO ou às bibliotecas que permitem este modelo de comunicação. Consideraremos sempre o caso de sistemas *centralizados*, na acepção do termo dada acima. O estudo dos modelos de comunicação em sistemas distribuídos far-se-á em disciplinas de anos seguintes, tais como Redes de Computadores, Sistemas Distribuidos I e II, e Sistemas de Computação Paralela e Distribuída.

4 As múltiplas dimensões da comunicação por mensagens

A figura 2 apresenta alguns aspectos do quadro geral dos modelos.

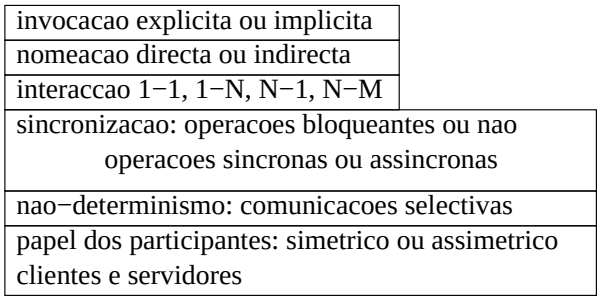

Figura 2: Dimensões de modelos de mensagens.

5 Invocação explícita ou implícita

Nas invocações explícitas, tem-se o caso ilustrado a seguir:

```
Processo P1        Processo P2

enviar(msg,P2)

                     receber(Msg)
```

Independentemente do modo como os participantes se nomeiam, neste caso, ambos têm de invocar explicitamente, nos seus programas, as primitivas para enviar e para receber.

Em alternativa, o receptor pode, no seu programa (possivelmente no início) declarar um procedimento que deve ser invocado automaticamente quando uma mensagem (qualquer ou específica, por exemplo, de um certo tipo, vinda de um certo destinatário, etc.) lhe for entregue. Nesse caso, o processo, que executa esse programa, deve ser automaticamente interrompido pelo SO (ou algum sistema de suporte), o procedimento declarado deve ser invocado e a mensagem chegada deve ser-lhe passada como um parâmetro de entrada (figura 3).

Este modelo, que se baseia num mecanismo de interrupção, tem um carácter intrinsecamente assíncrono, isto é, o momento de entrega da mensagem não sendo determinável pelo programa receptor, desencadeia uma interrupção durante a sua execução. As mensagens são entregues através da activação automática dos seus procedimentos de tratamento, declarados como(*handlers*) no início do programa. Num certo sentido, isto promove um modelo de

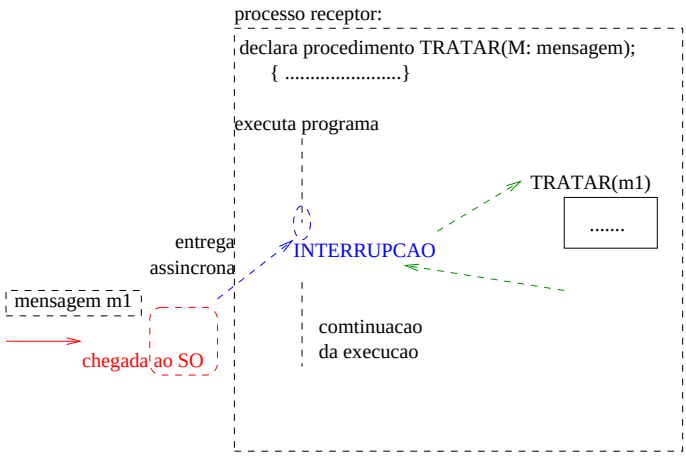

Figura 3: Notificação assíncrona de mensagens.

programação *reactivo*, quer dizer, tal que um processo é levado a reagir (invocar o procedimento TRATAR), logo que uma mensagem lhe é entregue pelo SO. Como o processo não conhece o momento de chegada das mensagens, este modelo permite lidar com esse não determinismo. Neste caso, a primitiva de recepção de mensagens não foi explicitamente invocada, ao contrário do modelo de envio e recepção explícitas.

A notificação assíncrona de mensagens, embora não existindo no Unix na forma como se apresenta acima, encontra uma leve aproximação a este modelo neste SO, no caso do mecanismo de *sinais* assíncronos, que veremos mais adiante. No entanto, outros SO (por exemplo, o VAX/VMS da DEC) e certos modelos de linguagens de programação concorrente (e.g. orientadas por objectos) incorporam este modelo. Como seria de esperar, a programação baseada nestes modelos exige cuidados, na sincronização entre a execução do programa e a dos procedimentos de tratamento de mensagens, de modo análogo ao que acontece quando se programa com interrupções num programa *assembly*, em que se impõem zonas de programa ininterrompíveis, para garantir que as estruturas de dados se mantêm coerentes.

6 Nomeação dos participantes

A figura 4 mostra dois modelos diferentes.

No modelo (a), dito de *nomeação directa* ambos os processos se nomeiam explicitamente. Sendo assim, têm de co-existir simultaneamente, isto é, se B

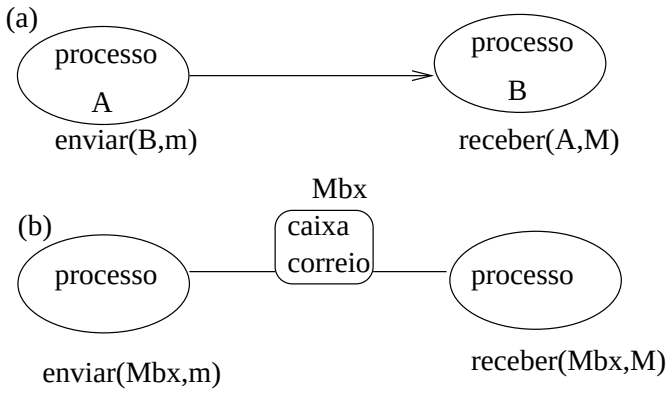

Figura 4: Nomeação directa e indirecta.

não existe quando A tenta a comunicação, ocorre um erro. Também, quando há a criação de novos processos no sistema, cujos identificadores não eram conhecidos dos processos existentes até esse momento, há que tornar esses identificadores conhecidos, para que possam envolver-se em comunicações.

Uma variante deste modelo, especialmente adequada para representar interacções de clientes com servidores, permite que o processo receptor, não sabendo quem quererá comunicar com ele, indique pretender receber de um qualquer emissor: uma primitiva da forma *receber(M)*, na qual só tem a mensagem a receber, M, como parâmetro de saída.

No modelo (b), dito de *nomeação indirecta*, existe um intermediário na comunicação entre os processos, na figura designado genericamente por *caixa de correio*. Neste modelo, cada participante activo (isto é, cada processo), dirige-se ao intermediário, para enviar ou receber mensagens. O intermediário, em geral, tem um papel passivo, isto é, limita-se a gerir filas de mensagens, actuando como um distribuidor indirecto: aceita as que lhe enviam e distribui-as pelos destinatários, quando estes lhas pedirem.

O modelo (b) facilita as interacções em que os processos não possam co-existir simultaneamente no sistema, por serem eventualmente executados em momentos diferentes. Também facilita a interacção entre processos, de uma forma anónima, se assim o quisermos, na medida em que qualquer processo possa enviar para a caixa de correio, e qualquer processo possa receber qualquer mensagem da caixa, sem ter de ser explicitamente nomeado pelo emissor. Um modelo destes pode, por exemplo, ser usado, para um qualquer processo no sistema, notificar todos os outros ou um subconjunto

dos processos existentes, sem ter de os nomear explicitamente. Qualquer novo processo que seja criado, pode começar a utilizar a caixa, desde que lhe tenha acesso.

Como veremos mais tarde, os modelos de nomeação indirecta podem assumir outras formas, para além de uma simples caixa de correio: os dispositivos intermediários podem ter um papel activo, discriminando a entrega de mensagens.

Em geral, podem ser os seguintes, os atributos das caixas de correio:

- um nome global, único no sistema
- direitos de acesso para protecção
- disciplinas possíveis de entrega das mensagens pendentes: FIFO; prioridades; conforme o tipo, tal como definido pelo emissor, etc.
- operações explícitas de criação e destruição
- operações para lhes associar canais ou portas de comunicação.

Por exemplo, as primitivas disponíveis no Unix System V, pertencentes à categoria IPC - *InterProcess Communication* de chamadas ao SO, permitem:

- criar ou ter acesso a uma fila de mensagens:
  msgqid = msgget(key, options)
  sendo *key* um nome global (isto é, pode ser acedido por processos, mesmo que de diferentes 'famílias', e em que *msgqid* é um identificador único, também global, devolvido pelo SO, para acesso mais eficiente à fila.
- enviar mensagem:
  msgsnd(msgqid, msg, cont, options)
  sendo *msgqid* o identificador acima referido e *msg* uma estrutura em C, com dois campos: o tipo da mensagem e o seu texto (um vector de caracteres); o tipo é um inteiro, atribuível pelo emissor, que possibilita ao receptor discriminar a obtenção de mensagens, conforme o seu tipo.
- receber mensagem:
  cont = msgrcv(msgqid, msg, maxcont, type, options)
  sendo *msg* um apontador para uma estrutura onde será colocada informação sobre a mensagem recebida, e sendo *type* um dos seguintes valores:
  - *type* = 0 indica que a ordem de entrega é FIFO

- um valor de *type* positivo indica que deve obter a primeira mensagem, na ordem FIFO, que tenha o tipo indicado
- um valor de *type* negativo, indica que deve obter a primeira mensagem com tipo igual ou inferior ao valor absoluto de *type*

6.1 Caixas de correio globais e privadas

Em geral, uma caixa de correio é realizada como uma fila de mensagens e suportada por estruturas auxiliares que permitem ao SO saber, a cada momento, quantas e quais as mensagens pendentes, e quais os processos que estão a aguardar a entrega de mensagens.

Num sistema centralizado, por exemplo, como o Unix System V, as caixas de correio (designadas por *message queues* ou filas de mensagens) são assim geridas pelo SO, sendo-lhes reservadas zonas de memória do SO. Neste tipo de sistemas, é aceitável que a caixa de correio admita um modelo de interacção n*m, isto é, quaisquer n procesos podem enviar mensagens e quaisquer m processos podem receber mensagens, desde que tenham as devidas permissões de acesso.

Contudo, num sistema de memória distribuída, o modelo n*m não é muito adequado, na medida em que não é fácil decidir em que memória e em que computador se deve reservar espaço para representar a fila de mensagens:

- num computador determinado, que assim actua como local de uma espécie de servidor centralizado, distribuidor de mensagens? Mas então este pode tornar-se um ponto de 'engarrafamento' do sistema, onde afluem todos os pedidos de envio e recepção, mesmo vindo de máquina remotas (isto é, cujos tempos de transmissão de mensagens podem ser apreciáveis);

- duplicada em cada computador onde haja potenciais participantes (isto é, emissores ou receptores de mensagens? Mas então há que manter cada uma das cópias da fila de mensagens actualizada em cada computador, sempre que algum processo envia ou recebe. Isto é muito difícil, porque os tempos de transmissão de mensagens não são desprezáveis.

Assim, compreende-se que haja um modelo limitado, em que a caixa de correio, ao invés de ser 'global', no sentido do modelo n*m, passa a ser 'privada', correspondendo a um modelo n*1, ou seja, quaisquer n procesos podem enviar mensagens, mas apenas um pode receber. Este último processo diz-se o dono da caixa de correio, pelo que, num sistema distribuído se torna

agora mais fácil decidir em que computador se deve localizar a caixa: no computador onde se executa o processo dono da caixa.

A figura 5 esquematiza este modelo.

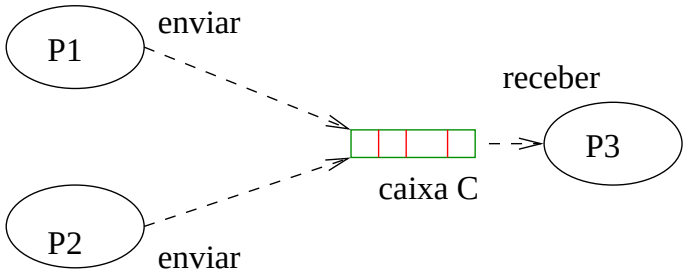

Figura 5: Caixas de correio privadas: modelo n*1.

As caixas de correio privadas, modelo n*1, por vezes designadas por *portas de comunicação* são as mais frequentes em sistemas distribuídos. Encontram-se, por exemplo, no Unix, associadas a um mecanismo que permite abrir e fechar canais virtuais de comunicação (*sockets*) em cada processo, os quais podem depois ser interligados de modo a permitir comunicação entre processos em computadores diferentes.

Neste modelo, é possível que um mesmo processo tenha uma ou mais caixas privadas (portas) associadas, como se ilustra na figura 6.

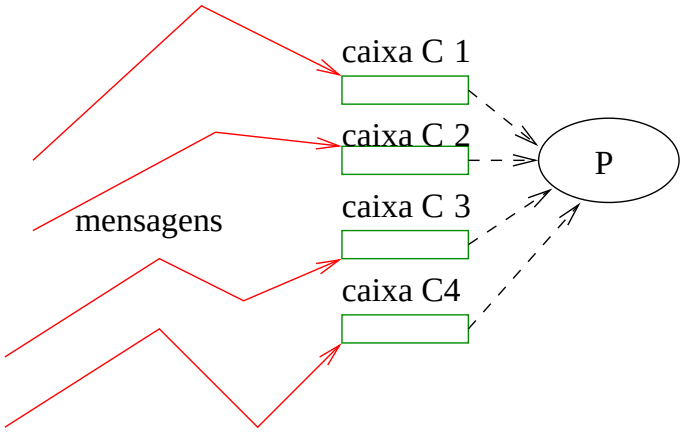

Figura 6: Caixas de correio privadas: modelo n*1.

Neste último caso, o processo tem o problema de, não sabendo a ordem

pela qual as mensagens chegam às diversas caixas, ter decidir de que forma aguarda pela chegada de mensagens:

- aguarda, bloqueando-se primeiro à espera de mensagens na caixa C1, por exemplo, invocando *receber(C1, Msg)*? Mas, nesse caso, pode ficar eternamente bloqueado, se 'nunca' chegarem mensagens a C1; e se, entretanto, forem chegando mensagens às outras caixas, estas não são nunca recebidas;

- não aguarda nunca em nenhuma caixa, fazendo, pelo contrário, testes sem bloqueio, para verificar se, em cada caixa, existem ou não mensagens pendentes para entrega: *testar-mensagens(Ci,Resposta)*, em que Resposta indica se sim ou não; mas este método pode originar um grande desperdício de tempo de CPU, como é habitual em soluções baseadas em espera activa, pois, não havendo mensagens em nenhuma caixa, o processo ficará em ciclo de testes sucessivos, até chegar alguma mensagem;

- aguarda, bloqueando-se, mas sem se comprometer com nenhuma caixa específica, isto é, invoca uma primitiva do tipo
  receber-qualquer(Caixa, Msg)
  em que *Caixa* devolve o identificador da 'primeira' caixa na qual surgiu uma mensagem, sendo esta devolvida em *Msg*;

- declara um procedimento de tratamento (*handler*) para ser invocado, de forma assíncrona, logo que uma mensagem chegue a qualquer das caixas; ou declara um *handler* para cada uma das caixas; neste caso, o processo pode executar um programa qualquer, e ser automaticamente interrompido quando uma mensagem chegar.

Como se vê, há múltiplas possíveis soluções para este problema. A nível do SO, em geral, são suportadas primitivas que permitem ao programador optar por diversas soluções, conforme o problema.

6.2 Outros modelos de comunicação por mensagens

Existem outras possibilidades de nomeação dos processos quanto à forma como comunicam entre si. Embora não sejam estudadas nesta disciplina, mencionam-se as formas de comunicação por *difusão de mensagens* (*broadcast*), em que um processo envia uma mesma mensagem para todos os processos presentes no sistema, num dado momento. Uma forma particular de difusão (designada por *multicast*) corresponde a enviar uma mensagem para os processos

que pertencem a um dado *grupo de processos*, sendo estes em geral processos que têm alguma relação entre si, por exemplo, participam na resolução conjunta de um dado problema, ou têm interesses comuns, por exemplo, estão interessados em receber mensagens sobre um dado assunto. Estas formas de comunicação correspondem a um modelo 1*n, isto é, um emissor e n receptores, sendo particularmente utilizadas em sistemas distribuídos. Nestes modelos, os destinatários não são nomeados individualmente, mas como um todo, ou indirecta e implicitamente, através de um nome do grupo ao qual pertencem. Compete ao sistema que suporta este modelo, a gestão dos grupos, isto é, saber a cada momento quais os processos que pertencem a cada grupo, de modo a propagar as mensagens para todos eles. O programador não se preocupa com esta gestão.