Computação Concorrente (MAB-117)
Comunicação entre threads via troca de mensagens

Prof. Silvana Rossetto

[1]Departamento de Ciência da Computação (DCC)
Instituto de Matemática (IM)
Universidade Federal do Rio de Janeiro (UFRJ)
Março de 2012

1. Introdução

A interação entre processos/threads de uma aplicação tem dois requisitos fundamentais: (i) **sincronização**, para garantir acesso exclusivo aos recursos compartilhados; e (ii) **comunicação**, para permitir a troca de informações. Vimos como esses requisitos podem ser satisfeitos usando memória compartilhada e primitivas de bloqueio/desbloqueio de processos/threads. Outra abordagem é o uso de **primitivas de troca de mensagens**, as quais podem ser usadas tanto em ambientes de memória compartilhada (sistemas multiprocessador de memória compartilhada ou sistemas uniprocessador), quanto em ambientes de máquinas distribuídas [1].

Os sistemas de troca de mensagens são construídos a partir de duas primitivas básicas:

- **send (dst, msg)**
- **receive (src, msg)**

Uma thread envia informação para um **destinatário** (*dst*) na forma de uma **mensagem** (*msg*) usando a primitiva `send`. Uma thread recebe informação de uma **fonte** (*src*) executando a primitiva `receive` com a indicação de uma estrutura de dados para receber a **mensagem** (*msg*) [1].

Neste texto abordaremos algumas questões básicas relacionadas aos sistemas de troca de mensagens.

2. Sincronização

A troca de mensagens entre threads requer algum nível de sincronização:

- uma thread não pode receber uma mensagem que ainda não foi enviada; e
- é necessário especificar o que acontece com uma thread depois que ela invoca as primitivas `send/receive` [1].

Quando a **primitiva send é executada** há duas possibilidades: (i) a thread é bloqueada até a mensagem ser recebida; (ii) a thread continua sua execução após o envio da mensagem (sem bloqueio). Quando a **primitiva receive é executada** há também duas possibilidades: (i) se a mensagem já foi enviada, a mensagem é recebida e a execução da thread continua; (ii) se a mensagem ainda não foi enviada, a **thread pode ser bloqueada até a mensagem chegar** (mais usual), ou a **thread continua executando e a tentativa de recebimento é abandonada** (menos usual) [1].

Dadas as **diferentes possibilidades de comportamentos de emissores e receptores**, três combinações são mais comuns [1]:

1. **envio bloqueante, recebimento bloqueante**: emissor e receptor são bloqueados até a mensagem ser entregue (referenciado como *rendezvous*), caracteriza o tipo mais forte de sincronização;
2. **envio não-bloqueante, recebimento bloqueante**: o receptor é bloqueado até a mensagem chegar, é a combinação mais usual e permite que uma thread envie mensagens para vários destinos em sequência (comum em aplicações de requisições de serviços);
3. **envio não-bloqueante, recebimento não-bloqueante**: nenhuma parte fica bloqueada.

2.1. Bloqueio do emissor

O bloqueio do lado do emissor pode servir para três propósitos [2]:

1. **Gerência de recursos**: ex., a thread emissora não pode modificar dados de saída até que o sistema tenha copiado os valores antigos para uma localização segura.
2. **Semântica de falhas**: como a comunicação em rede é susceptível a falhas, o bloqueio do emissor garante que ele irá esperar até a mensagem ser entregue com sucesso.
3. **Parâmetros de retorno**: nos casos em que uma mensagem constitui uma requisição para a qual uma resposta é esperada, o emissor fica bloqueado até receber a resposta.

Para definir quanto tempo o emissor fica bloqueado, deve-se considerar as seguintes questões: (i) **semântica da sincronização**; (ii) **requisitos de bufferização**; e (iii) **descoberta de erros em tempo de execução** [2].

Com relação à **semântica de sincronização**, há três possibilidades:

1. **envio sem espera**: o emissor espera apenas pelo período de tempo suficiente para colocar a mensagem em um buffer de saída e passar a responsabilidade de envio para o sub-sistema de comunicação;
2. **envio com sincronização**: o emissor espera até a mensagem ser recebida;
3. **envio de invocação remota**: o emissor espera até receber uma resposta.

Com relação à **bufferização**, particularmente no caso do "envio sem espera", haverá um limite quando o espaço de buffer for preenchido, então o processo deverá bloquear de fato.

Com relação à **descoberta de erros**, quando o sub-sistema de comunicação não oferece o serviço de entrega confiável, a linguagem de programação ou biblioteca de comunicação tipicamente usa mensagens especiais para confirmação do sucesso das transmissões. Erros são reportados quando várias tentativas de envio falham.

3. Endereçamento

Os diferentes esquemas usados para especificar processos/threads nas primitivas send/receive podem ser agrupados em duas categorias: **endereçamento direto** e **endereçamento indireto** [1].

No **endereçamento direto**, a primitiva `send` inclui um identificador específico do destinatário da mensagem. A primitiva `receive` pode ser tratada de duas maneiras:

1. o receptor define explicitamente o emissor da mensagem (nem sempre é viável, por exemplo, um servidor de impressão pode receber requisições de vários clientes);
2. o receptor usa *endereçamento implícito*, nesse caso o valor do parâmetro fonte é preenchido quando a operação é concluída.

No **endereçamento indireto** as mensagens não são enviadas diretamente do emissor para um receptor, mas sim para uma estrutura de dados compartilhada consistindo de filas que podem armazenar mensagens temporareamente. Essas filas são geralmente denominadas **mailboxes**. Para duas threads se comunicarem, elas enviam/retiram mensagens de um mailbox apropriado. Essa forma de endereçamento permite grande flexibilidade devido ao **desacoplamento entre emissores e receptores**.

3.1. Formato das mensagens

Uma mensagem é normalmente constituída de duas partes:

- **cabeçalho**: contém informações sobre a mensagem (identificação da fonte e destino, tamanho do dados, etc.)
- **corpo**: contém o conteúdo da mensagem.

4. Implementação de mailboxes em Java

Threads dentro de um mesmo processo podem acessar **objetos do tipo canal** implementados na memória compartilhada[3]. Em Java, podemos definir classes que implementam diferentes tipos de canais, baseados em interfaces que especificam a forma (semântica) das operações **send/receive** suportadas pelo canal. O código abaixo mostra a definição da interface `Channel`:

```
interface Channel {
   void send(Object m); //envia uma mensagem cujo conteudo eh um objeto
   Object receive();    //recebe uma mensagem que contem um objeto
}
```

Podemos distinguir três tipos de canais, baseado no número de threads emissoras e receptoras que podem acessá-lo [3]:

1. **Caixa de mensagem** (*mailbox*): vários emissores e vários receptores podem acessar o canal;
2. **Porta** (*port*): vários emissores e apenas um receptor podem acessar o canal;
3. **Enlace** (*link*): um par emissor e receptor pode acessar o canal.

Cada tipo de canal pode ter implementações no modo **síncrono** (bloqueante) ou **assíncrono** (não-bloqueante) para as operações send/receive.

5. Exclusão mútua via troca de mensagens

Quando processos/threads de uma aplicação interagem via troca de mensagens, as próprias primitivas de comunicação oferecem suporte para sincronizar ou coordenar as várias atividades da aplicação.

Assumindo o uso das primitivas `send` não-bloqueante e `receive` bloqueante, o pseudo-código abaixo mostra como implementar exclusão mútua entre processos de uma aplicação [1].

```
const int N //número de processos da aplicação
void P(int i) {
   message msg;
   while(true) {
      receive(mailbox, msg);
      //executa a seção crítica
      send(mailbox, msg);
      //executa fora da seção crítica
   }
}

void main() {
   createMailbox(mailbox);
   send(mailbox, null);
   //inicia os N processos
}
```

Na solução apresentada acima, um conjunto de processos compartilham um *mailbox* que é usado por todos os processos para enviar e receber mensagens. O *mailbox* é inicializado com uma única mensagem de conteúdo vazio. O processo que deseja entrar na seção crítica tenta primeiro receber uma mensagem. Se o *mailbox* estiver vazio, o processo é bloqueado, caso contrário o processo executa a seção crítica e devolve uma mensagem para o *mailbox*. Podemos concluir que a mensagem funciona como um *token* ("bastão") que é passado de um processo para o outro.

Quando mais de uma processo pode executar concorrentemente, a solução apresentada assume as seguintes premissas:

- se há apenas uma mensagem no *mailbox*, ela é entregue a apenas um processo, os demais são bloqueados;
- se o *mailbox* estiver vazio, todos os processos ficam bloqueados, e quando uma mensagem torna-se disponível, apenas um processo é ativado.

Essas premissas são normalmente verdadeiras em todos os sistemas de troca de mensagens [1].

6. O problema do produtor/consumidor usando troca de mensagens

O pseudo-código abaixo mostra uma solução para o **problema do produtor/consumidor** usando troca de mensagens [1].

```
void produtor() {
   message pmsg;
   while(true) {
      receive(pode_produzir, pmsg);
      pmsg = produz_item();
      send(pode_consumir, pmsg);
   }
}
```

```
void consumidor() {
   message cmsg, msg_vazia;
   while(true) {
      receive(pode_consumir, cmsg);
      send(pode_produzir, msg_vazia);
      consome(cmsg);
   }
}

void main() {
   int N //tamanho do buffer
   message msg_vazia;
   createMailbox(pode_produzir);
   createMailbox(pode_consumir);
   for(int i=1; i<=N; i++)
      send(pode_produzir, msg_vazia);
   //inicia os processos consumidores e produtores
}
```

A solução aproveita o fato das mensagens servirem tanto para enviar dados quanto sinais de sincronização. Dois *mailboxes* são usados: `pode_produzir` e `pode_consumir`. Inicialmente o mailbox `pode_produzir` é preenchido com um número N de mensagens vazias, correspondente a capacidade do buffer. O número de mensagens nesse mailbox diminui com cada novo item gerado por um processo produtor e aumenta com cada item retirado por um processo consumidor. Quando um produtor gera um novo item ele envia uma mensagem contendo o item gerado para o mailbox `pode_consumir`. Quando há ao menos uma mensagem nesse mailbox, um processo consumidor pode consumir o item.

A solução apresentada permite vários produtores e vários consumidores executando concorrentemente, desde que tenham acesso aos dois mailboxes.

Exercícios

1. Por que em geral a operação de envio de mensagem precisa bloquear?
2. Quais são as principais opções de sincronização para o emissor de uma mensagem?
3. Diferencie endereçamento direto e endereçamento indireto.

Referências

[1] W. Stallings. *Operating Systems – Internals and Design Principles*. Pearson - Prentice Hall, 6 edition, 2009.

[2] M. L. Scott. *Programming Language Pragmatics*. Morgan-Kaufmann, 2 edition, 2006.

[3] Richard H. Carver and Kuo-Chung Tai. *Modern multithreading*. Wiley, 2006.