1.Comunicação entre processos

Frequentemente, os processos precisam se comunicar com outros processos, logo, há necessidade de comunicação entre os processos, preferivelmente de uma maneira bem estruturada que não utilize interrupções.

Existem três problemas relacionados à comunicação entre processos (IPC). O primeiro é de como um processo pode passar informações para outro. O segundo tem a ver com como garantir que dois ou mais processos não interfiram um com o outro quando envolvidos em atividades críticas. O terceiro diz respeito ao sequenciamento adequado quando estão presentes dependências. Se o processo A produz dados e o processo B os imprime, B tem de esperar até que A tenha produzido alguns dados, antes de começar a imprimir.

Em alguns sistemas operacionais, os processos que estão trabalhando juntos podem compartilhar algum armazenamento comum onde cada um deles pode ler e escrever. O armazenamento compartilhado pode estar na memória principal ou pode ser um arquivo. A localização exata do compartilhamento não muda a natureza da comunicação nem os problemas que surgem. Situações onde dois ou mais processos lêem e escrevem dados compartilhados e o resultado final depende da ordem de quem precisamente executa e quando, são chamadas de condições de corrida.

O segredo para evitar problemas de condições de corrida e em outras situações que envolve memória compartilhada, arquivos compartilhados e tudo mais compartilhado é encontrar alguma maneira de proibir que mais de um processo leia e modifique dados compartilhados ao mesmo tempo. Em outras palavras, é preciso uma exclusão mútua, uma maneira de garantirmos que, se um processo estiver utilizando um arquivo compartilhado, ou uma variável compartilhada, outros processos sejam impedidos de fazer a mesma coisa.

O problema de evitar as condições de corrida também pode ser formulado de uma maneira abstrata. Parte do tempo, um processo fica ocupado fazendo cálculos internos e outras coisas que não causam condições de corrida. Entretanto, às vezes um processo pode estar acessando memória compartilhada ou arquivos compartilhados. Essa parte do programa, em que a memória compartilhada é acessada, é chamada de região crítica ou seção crítica. Se pudéssemos organizar as coisas de tal modo que dois processos jamais estivessem em suas regiões críticas ao mesmo tempo, poderíamos evitar as condições de corrida. Para isso, é preciso que quatro condições sejam válidas.

1. Dois processos não podem estar simultaneamente dentro de uma região crítica
2. Nenhuma suposição pode ser feita sobre a velocidade ou sobre o número de CPUs
3. Nenhum processo executando fora de sua região crítica pode bloquear outros processos
4. Nenhum processo deve ter que esperar eternamente para entrar em sua região crítica

Existem algumas soluções para se obter exclusão mútua, para que enquanto um processo está ocupado atualizando a memória compartilhada em sua região crítica, nenhum outro processo entre em sua região crítica e cause problemas.

A mais simples das soluções é fazer cada processo desativar todas as interrupções imediatamente após entrar em sua região crítica e reativá-las imediatamente após sair dela. Com as interrupções desativadas, nenhuma interrupção de relógio pode ocorrer. Assim, quando um processo estiver desativando as interrupções, ele poderá examinar e atualizar a memória compartilhada sem medo de que qualquer outro processo intervenha. Esta estratégia é pouco atraente, pois não é aconselhável dar a processos de usuário o poder de desativar interrupções.

Frequentemente é conveniente que o próprio núcleo desative interrupções para algumas instruções, enquanto está atualizando variáveis ou listas. Se ocorresse uma interrupção, por exemplo, enquanto a lista de processos prontos estivesse em um estado inconsistente, poderiam ocorrer condições de corrida. A conclusão é: desativar interrupções é frequentemente uma técnica útil dentro do sistema operacional em si, mas não apropriada como um mecanismo de exclusão mútua geral para processos de usuário.

Como segunda alternativa, considere o fato de ter uma única variável (trava*) compartilhada, inicialmente com o valor 0. Quando um processo quer entrar em sua região crítica, ele primeiro testa o valor da variável trava. Se a trava for 0, o processo o irá configurar como 1 e entrará na região crítica. Se a trava já for 1, o processo apenas esperará até que ele se torne 0, assim, 0 significa que nenhum processo está em sua região crítica e 1 significa que algum processo está em sua região crítica.

Esta ideia contém a seguinte falha fatal. Suponha que um processo leia a trava e veja que ela é 0. Antes que ele possa configurar a trava como 1, outro processo é selecionado para execução, começa a executar e irá configurar a trava como 1. Quando o primeiro processo for executado novamente, ele também irá configurar trava como 1 e os dois processos estarão em suas regiões críticas ao mesmo tempo.

Combinando a ideia de alternância com a ideia de variáveis de trava e variáveis de alerta, o matemático holândes T. Dekker foi o primeiro a imaginar uma solução de software para o problema da exclusão mútua que não exigisse uma alternância estrita.

Em 1981, G.L. Peterson descobriu um modo muito mais simples de obter a exclusão mútua, tornando obsoleta a solução de Dekker. O algoritmo de Peterson consiste em duas rotinas escritas em C ANSI, o que significa que devem ser fornecidos protótipos de função para todas as funções definidas e utilizadas.

```
#define FALSE 0
#define TRUE 1
#define N 2 /* número de processos */

int turn; /* de quem é a vez ? */
int interested[N]; /* todos os valores são inicialmente 0 (FALSE) */

/* o processo é 0 ou 1 */
void enter_region(int process){
  int other; /* número do outro processo */
```

```
  other = 1 - process; /* o oposto do processo */
  interested[process] = TRUE; /* mostra que você está interessado */
  turn = process; /* configura flag */
  while (turn == process && interested[other] == TRUE) /* laço de espera */
}

/* process: quem está saindo */
void leave_region(int process){
  interested[process] = FALSE; /* indica saída da região crítica*/
}
```

Antes de utilizar as variáveis compartilhadas, cada processo chama *enter_region* com seu próprio número de processo 0 ou 1, como parâmetro. essa chamada fará com que ele espere, se necessário, até que seja seguro entrar. Depois que terminou de trabalhar com as variáveis compartilhadas, o processo chama *leave_region* para indicar que terminou e permitir que o outro processo entre, se assim o desejar.

Inicialmente, nenhum processo está em sua região crítica. Agora, o processo 0 chama *enter_region*. Ele indica seu interesse escrevendo TRUE no elemento de *array* a si associado e configurando *turn* como 0. Como o processo 1 não está interessado, *enter_region* retorna imediatamente. Se o processo 1 agora chamar *enter_region*, ele ficará parado até que *interested[0]* torne-se FALSE, um evento que só acontece quando o processo 0 chama *leave_region* para sair da região crítica.

Agora, considere o caso em que dois processos chamam *enter_region* quase simultaneamente. Ambos irão armazenar seu número de processo em *turn*. Qualquer que seja o armazenamento feito por último, é este que conta; o primeiro é perdido. Suponha que o processo 1 armazene por último, de modo que *turn* é 1. Quando os dois processos chegarem na instrução *while*, o processo 0 a executa zero vezes e entra em sua região crítica. O processo 1 entra no laço e não entra em sua região crítica.

A instrução TSL

Muitos computadores, especialmente aqueles projetados com múltiplos processadores em mente, têm uma instrução **TSL RX,LOCK** (test and lock), que funciona da seguinte maneira

Ele lê o conteúdo da palavra de memória **LOCK** no registrador **RX** e, então, armazena um valor diferente de zero no endereço de memória **LOCK**. É garantido que as operações de leitura e armazenamento da palavra são indivisíveis, logo, nenhum outro processador pode acessar a palavra de memória até que a instrução tenha terminado. A CPU que executa a instrução **TSL** bloqueia o barramento de memória, proibindo outras CPUs de acessar a memória até que ela tenha terminado.

Para usar a instrução **TSL**, será utilizado uma variável compartilhada, **LOCK**, para coordenar o acesso à memória compartilhada. Quando **LOCK** é 0, qualquer processo pode configurá-la com 1 utilizando a instrução TSL e, então, ler ou modificar a memória compartilhada. Ao terminar, o processo configura **LOCK** novamente como 0, utilizando uma

instrução *move* normal. A solução para regiões críticas através desta instrução é mostrada abaixo.

enter_region:
TSL REGISTER, LOCK | copia LCOK no registrador e configura LOCK como 1
CMP REGISTER, #0 | LOCK era zero ?
JNE ENTER_REGION | se não era zero, LOCK estava configurada, logo, entra no loop
RET | retorna para quem fez a chamada e entra na região crítica

leave_region:
MOVE LOCK, #0 | armazena o valor 0 em LOCK
RET |retorna para quem fez a chamada

A solução agora para o problema da região crítica agora é simples. Antes de entrar em sua região crítica, um processo chama *enter_region*, que faz a espera ativa até que a trava esteja livre, em seguida, ele adquire a trava e retorna. Depois da região crítica, o processo chama *leave_region*, que armazena o valor 0 em LOCK. Como acontece em todas as soluções baseadas em regiões críticas, os processos devem chamar *enter_region* e *leave_region* nos momentos certos para o método funcionar. Se um processo trapacear, a exclusão mútua falhará.

1.1 Sleep e Wakeup

Tanto a solução de Peterson como a solução TSL são corretas, porém, ambas têm o defeito de exibir a espera ativa. A estratégia destas duas soluções desperdiça tempo de CPU como também pode ter efeitos inesperados como o ***problema da inversão de prioridade***.

Para isso, usa-se certas primitivas simples, como *sleep* e *wakeup*. *Sleep* é uma chamada de sistema que causa o bloqueio do processo que fez a chamada. A chamada *wakeup* tem um parâmetro, o processo a ser despertado. Como alternativa, *sleep* e *wakeup* têm um parâmetro, um endereço de memória utilizado para fazer as instruções *sleep* corresponderem às instruções *wakeup*.

Considere o problema do ***produtor-consumidor***. Dois processos compartilham um buffer de tamanho fixo. Um deles, o ***produtor***, coloca informações no buffer e o outro, ***consumidor*** as retira.

O problema surge quando o produtor quer colocar um novo item no buffer, mas este já está cheio. A solução é o produtor ser bloqueado (*sleep*) esperando o consumidor remover um ou mais itens e permitir que o produtor seja desbloqueado (*wakeup*). De maneira semelhante, se o consumidor quiser remover um item do buffer este estiver vazio, ele bloqueará (*sleep*) até que o produtor coloque algo no buffer e o desbloqueie (*wakeup*).

Para controlar o número de itens no buffer, é preciso de uma variável *count*. Se o número máximo de itens que o buffer pode armazenar for *N*, o código do produtor primeiro testará se *count* é *N*. Se for, o produtor bloqueará, se não for, o produtor adicionar um item e incrementará *count*.

O código do consumidor é semelhante. Primeiro testa *count* para ver se é 0. Se for, bloqueará, caso contrário, removerá um item e decrementará o contador. Cada um dos processos também testa para ver se o outro está bloqueado e se deve desbloqueá-lo.

```
#define N 100 /* número de entradas no buffer */
int count = 0; /* número de itens do buffer */

void producer(void){
  int item;
  while (TRUE){                 /* repete para sempre */
    item = produce_item();     /*gera o próximo item */
    if (count == N) sleep();    /* se o buffer estiver cheio, bloqueia */
    insert_item(item);          /* coloca item no buffer */
    count = count + 1;          /* incrementa a contagem de itens no buffer */
    if (count == 1) wakeup(consumer); /* o buffer estava vazio ? */
  }
}

void consumer(void){
  int item;
  while (TRUE){                 /* repete para sempre */
    if (count == 0) sleep();    /* se o buffer estiver vazio, bloqueia */
    item = remove_item();       /* retira item do buffer */
    count = count - 1;          /* decrementa a contagem de itens no buffer */
    if (count == N - 1) wakeup(producer); /* o buffer estava cheio ? */
    consume_item(item);     /* imprime o item */
  }
}
```

Uma condição de corrida pode ocorrer porque o acesso a *count* é irrestrito. A seguinte situação possivelmente pode ocorrer. O buffer está vazio e o consumidor acabou de ler *count* para ver se é 0. Nesse instante, o escalonador decide parar temporariamente de executar o consumidor e começa a executar o produtor. O produtor insere um item no buffer, incrementa *count* e avisa que ela agora é 1. Deduzindo que *count* era 0 e, assim que o consumidor deve estar dormindo, o produtor chama *wakeup* para acordá-lo.

Infelizmente, o consumidor ainda não está logicamente bloqueado, portanto, o sinal para despertar é perdido. Na próxima vez que o consumidor for executado, ele testará o valor de *count* lido anteriormente, verificará que ele é 0 e bloqueará. Cedo ou tarde, o produtor preencherá o buffer e também bloqueará. Ambos ficarão eternamente bloqueados. A essência do problema aqui é um sinal para despertar, enviado para um processo que ainda não está bloqueado, é perdido. Se ele não fosse perdido, tudo funcionaria.

Uma correção rápida seria modificar as regras para adicionar um ***bit de espera por despertar*** ao quadro geral. Quando um sinal para despertar for enviado para um processo que ainda está acordado, esse bit é ativado. Posteriormente, quando o processo for ser bloqueado,

e o bit de espera por despertar estiver ativado, ele será desativado, e o processo permanecerá desbloqueado. O bit de espera por despertar é um cofrinho de sinais de despertar.

Embora o bit de espera por despertar resolva o problema, com três ou mais processos nos quais um bit de espera por despertar é insuficiente.

1.2 Semáforos

Um semáforo tem o valor 0, indicando que nenhum sinal para despertar foi salvo, ou um valor positivo, caso um ou mais sinais para despertar estivessem pendentes. Dijkstra propôs ter duas operações, *down* e *up* (generalizações de *sleep* e *wakeup*). Em um semáforo, a operação *down* verifica se o valor é maior do que 0. Se for, ele decrementa o valor e simplesmente continua. Se o valor for 0, o processo é bloqueado sem completar a operação *down*. Verificar o valor, alterá-lo e, possivelmente, ser bloqueado, tudo é feito como uma única e indivisível ação atômica. É garantido que, uma vez iniciada, uma operação de semáforo, nenhum outro processo pode acessar o semáforo até que a operação tenha terminado ou sido bloqueada. Essa atomicidade é absolutamente essencial para resolver problemas de sincronismo e evitar condições de corrida.

A operação *up* incrementa o valor do semáforo passado como parâmetro. Se um ou mais processos estavam bloqueados neste semáforo, um deles é escolhido pelo sistema (aleatoriamente) e autorizado a completar sua operação *down*. Assim, depois de uma operação *up* em um semáforo contendo processos bloqueados, o semáforo ainda será 0, mas haverá nele um processo bloqueado a menos.

A maneira normal é implementar *up* and *down* como chamadas de sistema, com o sistema operacional desativando brevemente todas as interrupções enquanto está testando o semáforo, atualizando-o e bloqueando o processo se necessário. Se várias CPUs estiverem sendo utilizadas, cada semáforo deverá ser protegido por uma variável do tipo trava, com a instrução TSL usada para garantir que apenas uma CPU por vez examine o semáforo. Entenda que utilizar TSL para impedir que várias CPUs acessem o semáforo ao mesmo tempo é muito diferente da espera ativa por parte do produtor ou do consumidor, aguardando o outro esvaziar ou preencher o buffer.

```
#define N 100                       /* número de entradas no buffer */
typedef int semaphore;              /* os semáforos são um tipo especial de inteiro */
semaphore mutex = 1;                        /* controla o acesso à região crítica */
semaphore empty = N;                /* conta as entradas livres do buffer */
semaphore full = 0;                 /* conta as entradas ocupadas do buffer */

void producer(void){
int item;

while(TRUE){
  item = produce_item();            /* produz algo para colocar no buffer */
  down(&empty);                     /* decrementa a contagem de entradas livres */
  down(&mutex);                     /* entra na regiã crítica */
```

```
  insert_item(item);              /* colocar um novo item no buffer */
  up(&mutex);                     /* sai da região crítica */
  up(&full);                      /* incrementa a contagem de entradas ocupadas */
}
}

void consumer(VOID){
int item;

while(TRUE){
  down(&full);                    /* decrementa a contagem de entradas ocupadas */
  down(&mutex);                   /* entra na região crítica */
  item = remove_item();           /* retira item do buffer */
  up(&mutex);                     /* sai da região crítica */
  up(&empty);                     /* incrementa a contagem de entradas livres
*/
  consume_item(item);             /* faz algo com o item */
}
}
```

Essa solução utiliza três semáforos: um chamado *full*, para contar o número de entradas que estão ocupadas, um chamado *empty*, para contar o número de entradas que estão livres, e um chamado *mutex*, para garantir que o produtor e o consumidor não acessem o buffer ao mesmo tempo. Inicialmente, *full* é 0, *empty* é igual ao número de entradas no buffer e *mutex* é 1. Os semáforos inicializados com 1 e utilizados por dois ou mais processos para garantir que apenas um deles possa entrar em sua região crítica por vez, são chamados de ***semáforos binários***. Se cada processo executa uma operação *down* imediatamente antes de entrar em sua região crítica é uma operação *up* imediatamente após sair dela, a exclusão mútua é garantida.

1.2.1 Mutex

Os *mutexes* são bons apenas para gerenciar a exclusão mútua de algum recurso ou parte de código compartilhado. Sua implementação é fácil e eficiente, o que os torna particularmente úteis nas implementações de *threads* em espaço de usuário. Um *mutex* é uma variável que pode ter dois estados, livre ou ocupado. Consequentemente, apenas 1 bit é exigido para representá-lo, mas, na prática, frequentemente é usado um valor inteiro, com 0 significando livre e todos os outros valores significando ocupado. Quando um processo (*thread*) precisa acessar uma região crítica, ele chama *mutex_lock*. Se o *mutex* está corretamente livre, a chamada é bem-sucedida e o processo que fez a chamada pode entrar na região crítica. Por outro lado, se o *mutex* está ocupado, o processo que faz a chamada é bloqueado até que o processo que se encontra na região crítica tenha terminado e chame *mutex_unlock*. Se vários processos estiverem bloqueados no *mutex*, um deles será escolhido aleatoriamente e poderá entrar na região crítica.

1.2.2 Monitores

Suponha que duas operações *down* no código do produtor tivessem sua ordem invertida, de modo que o *mutex* fosse decrementado antes de *empty* e não depois. Se o buffer estivesse completamente cheio, o produtor seria bloqueado, com *mutex* configurado como 0. Consequentemente, na próxima vez que o consumidor tentasse acessar o buffer, ele executaria uma operação *down* em *mutex*, agora 0, e também seria bloqueado. Os dois processos permaneceriam bloqueados para sempre e mais nenhum trabalho seria feito. Essa situação é chamada de ***deadlock***.

Para tornar mais fácil escrever programas corretamente, Brinch Hansen e Hoare propuseram uma primitiva de sincronismo de mais alto nível chamada ***monitor***. Suas propostas diferiam ligeiramente.

Um monitor é um conjunto de rotinas, variáveis e estruturas de dados, todas agrupadas em um tipo especial de módulo ou pacote. Os processos podem chamar as rotinas presentes em um monitor sempre que quiserem, mas não podem acessar diretamente as estruturas de dados internas do monitor a partir das rotinas declaradas fora dele

Os monitores têm uma propriedade importante que os torna úteis para obter exclusão mútua: a qualquer instante, apenas um processo pode estar ativo em um monitor. Os monitores são uma construção de linguagem de programação, de modo que o compilador sabe que eles são especiais e pode manipular chamadas para as rotinas do monitor de forma diferente de outras chamadas de procedimentos. Em geral, quando um processo chama uma rotina do monitor, suas primeiras instruções verificam se algum outro processo está ativo dentro do monitor. Se assim for, o processo que fez a chamada será suspenso até que o outro processo tenha saído do monitor. Se nenhum outro processo estiver usando o monitor, o processo que fez a chamada poderá entrar.

Cabe ao compilador implementar a exclusão mútua em entradas de monitor, mas uma maneira comum é utilizar um *mutex* ou semáforo binário. Embora os monitores ofereçam uma maneira fácil de obter exclusão mútua, não se torna o suficiente. Também é preciso uma maneira de bloquear os processos quando eles não podem prosseguir. No problema do produtor-consumidor, é muito fácil colocar todos os testes de buffer cheio e buffer vazio em rotinas do monitor.

A introdução de ***variáveis de condição*** ajuda a bloquear o produtor quando o buffer estiver vazio junto com duas operações sobre elas, *wait* e *signal*. Quando uma rotina do monitor descobre que não pode continuar, ela executa uma operação *wait*, uma variável de condição, digamos *full*. Essa ação causa o bloqueio do processo que fez a chamada. Ela também permite que outro processo, anteriormente proibido de entrar no monitor, agora entre.

Esse outro processo, por exemplo, o consumidor, pode despertar seu parceiro que está bloqueado, executando uma operação *signal* na variável de condição que está esperando. Para evitar a existência de dois processos simultaneamente ativos no monitor, é preciso de uma regra dizendo o que acontece após uma operação *signal*.

Hoare propôs deixar o processo recentemente desbloqueado executar, suspendendo o outro. Brinch propôs refinar o problema, exigindo que um processo que execute uma operação *signal* deve sair do monitor imediatamente. Se uma operação *signal* é executada em uma variável de condição em que vários processos estão esperando, apenas um deles, determinado pelo escalonador do sistema, é desbloqueado.

Um esboço da solução do problema do produtor-consumidor com monitores. Apenas uma rotina do monitor por vez está ativa.

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;

  count := 0;
end monitor;

procedure producer;
begin
  while true do
  begin
    item = produce_item;
    ProducerConsumer.insert(item)
  end
end;

procedure consumer;
begin
  while true do
  begin
```

```
    item = ProducerConsumer.remove;
    consume_item(item)
  end
end;
```

A diferença fundamental entre *sleep wakeup* e *wait signal* é que *sleep wakeup* falhavam porque, enquanto um processo estava sendo bloqueado, um outro já tentava desbloqueá-lo. Com os monitores, isso não acontece. A exclusão mútua automática em rotinas do monitor garante que se, digamos, o produtor existente dentro de uma rotina do monitor descobre que o buffer está cheio, ele poderá completar a operação *wait* sem precisar preocupar-se com a possibilidade do escalonador trocar para o consumidor exatamente antes da operação *wait* terminar. O consumidor nem mesmo será autorizado a entrar no monitor até que a operação *wait* termine e o produtor seja identificado como não executável.

Tornando automática a exclusão mútua de regiões críticas, os monitores tornaram a programação paralela muito menos sujeita a erros do que com semáforos. Mas eles também têm alguns inconvenientes. C e a maioria das outra linguagens não tem monitores, portanto, não é razoável esperar que seus compiladores imponham regras de exclusão mútua.

Essas mesmas linguagens também não tem semáforos, mas é fácil adicioná-los: basta adicionar à biblioteca duas rotinas curtas em código *assembly*, para produzir as chamadas de sistemas *up* e *down*.

Naturalmente, os sistemas operacionais precisam saber da existência dos semáforos, mas pelo menos se você tiver um sistema operacional baseado em semáforos, ainda poderá escrever programas de usuário para ele em C ou C++. Com monitores, você precisa de uma linguagem que os tenha incorporado.

Outro problema dos monitores e também dos semáforos, é que eles foram projetados para resolver o problema dos monitores, e também em uma ou mais CPUs que têm acesso a uma memória comum. Colocando os semáforos na memória compartilhada e protegendo-os com instruções TSL, podemos evitar as condições de corrida. Quando usamos um sistema distribuído, composto de várias CPUs, cada uma com sua própria memória privativa e conectadas por uma rede local, essas primitivas se tornam inaplicáveis. A conclusão é que os semáforos são de nível muito baixo e os monitores não são utilizáveis, exceto em poucas linguagens de programação.

1.2.3 Passagem de mensagens

Esse método de comunicação entre processos utiliza duas primitivas *send* e *receive*, as quais, como semáforos e ao contrário dos monitores, são chamadas de sistema em vez de construções da linguagem. Como tal, elas podem ser facilmente colocadas em funções de biblioteca.

```
send(destination, &message);
receive(source, &message);
```

A primeira chamada envia uma mensagem para determinado destino, enquanto a segunda recebe uma mensagem de determinada origem. Se nenhuma mensagem estiver disponível, o destinatário é bloqueado até uma chegar. Como alternativa, ele pode retornar imediatamente com um código de erro.

Os sistemas de passagem de mensagem têm muitos problemas desafiadores e questões de projeto que não aparecem nos semáforos nem nos monitores, especialmente se os processos que estão se comunicando estiverem em máquinas diferentes conectadas por uma rede. Para evitar perda de mensagens, o remetente e o destinatário podem concordar que, assim que a mensagem for recebida, o destinatário enviará de volta uma mensagem especial de ***reconhecimento ou confirmação***.

Agora considere o que acontece se a mensagem em si é recebida corretamente, mas o sinal de reconhecimento é perdido. O remetente retransmitirá a mensagem, portanto, o destinatário a receberá duas vezes. É fundamental que o destinatário possa diferenciar entre uma nova mensagem e a retransmissão de uma antiga. Normalmente, esse problema é resolvido colocando-se números em sequência consecutivos em cada mensagem original.

Os sistemas de mensagem também têm de lidar com a questão de como os processos são identificados, para que o processo especificado em uma chamada *send* ou *receive* não seja ambíguo. A ***autenticação*** também é um problema nos sistemas de mensagem.

```
#define N 100          /* número de entradas no buffer */

void producer(void){
int item;
message m;             /* buffer de mensagens */

while(TRUE){
  item = produce_item();     /* produz algo para colocar no buffer */
  receive(consumer, &m);     /* espera a chegada de uma mensagem vazia */
  build_message(&m, item);          /* espera a chegada de uma mensagem vazia */
  send(consumer, &m);        /* envia item para o consumidor */
}
}

void consumer(void){
int item, i;
message m;

for (i = 0; i < N; i++) send(producer, &m);/* envia N mensagens vazias */
while(TRUE){
  receive(producer, &m);                   /* recebe a mensagem contendo o item */
  item = extract_item(&m);                 /* extrai o item da mensagem */
  send(producer, &m);                      /* envia de volta resposta vazia */
```

```
  consume_item(item);                    /* faz algo com o item */
}
}
```

O problema do produtor-consumidor pode ser resolvido com passagem de mensagens e nenhuma memória compartilhada acima.

Supomos que todas as mensagens têm o mesmo tamanho e que as mensagens enviadas, mas ainda não recebidas, são automaticamente armazenadas em buffer pelo sistema operacional. Nessa solução, é utilizado um total de N mensagens, análogo às N entradas em um buffer de memória compartilhada. O consumidor começa enviando N mensagens vazias para o produtor. Quando o produtor tem um item para enviar ao consumidor, ele pega uma mensagem vazia e envia de volta uma cheia. Dessa maneira, o número total de mensagens no sistema permanece constante com o tempo, de modo que elas podem ser armazenadas em uma quantidade de memória previamente conhecida.

Se o produtor trabalhar mais rápido do que o consumidor, todas as mensagens serão usadas e o produtor será bloqueado esperando o consumidor enviar de volta uma mensagem vazia. Se o consumidor trabalhar mais rápido, então o inverso acontecerá: todas as mensagens vazias, esperando o produtor enchê-las: o consumidor será bloqueado, esperando uma mensagem cheia.