rodrigoscama@yahoo.com.br http://scama.norbis.com.br

Programação Concorrente

Definição

Um programa que é executado apenas por um processo é chamado programa seqüencial. A grande maioria dos programas escritos são programas seqüenciais. Nesse caso existe apenas um fluxo de controle durante a execução. Isso permite, por exemplo, que o programador realize uma "execução imaginária" de seu programa apontando com o dedo, a cada instante, a linha do programa que está sendo executada no momento.

Um programa concorrente é executado simultaneamente por diversos processos que cooperam entre si, isto é, trocam informações. Para o programador realizar agora uma "execução imaginária", ele vai necessitar de vários dedos, um para cada processo que faz parte do programa.

O termo "programação concorrente" vem do inglês concurrent programming, onde concurrent quer dizer "acontecendo ao mesmo tempo". Além do termo acima, no Brasil também usamos o termo programação paralela.

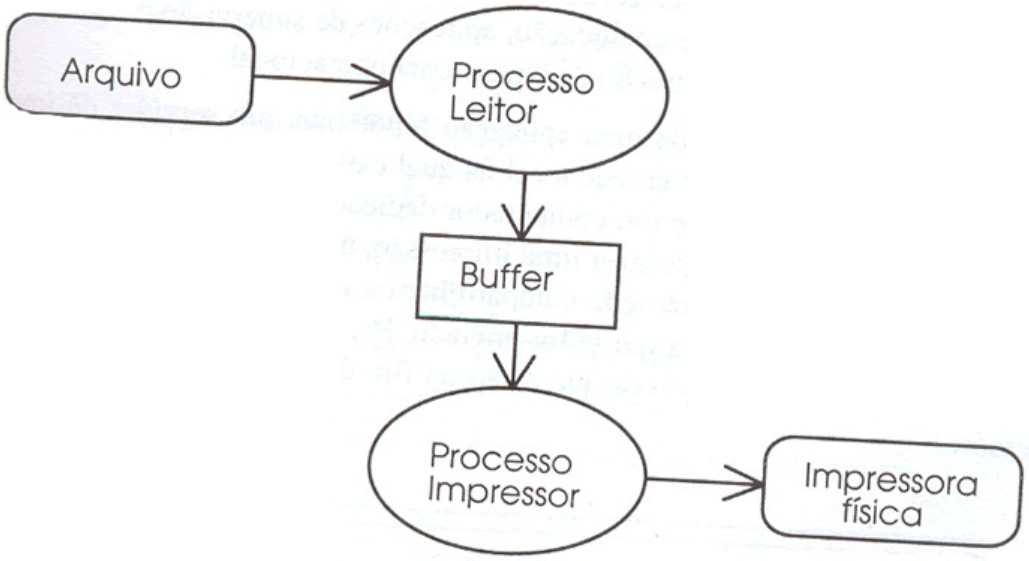
É comum em sistemas multiusuário que um mesmo programa seja executado simultaneamente por vários usuários, como um editor de textos. Entretanto, executar simultaneamente 10 instâncias de um editor de texto não o faz dele programa concorrente. Apenas o código é compartilhado pelos 10 processos. Cada processo executa sobre sua própria área de dados e ignora a existência de outras execuções do programa. Esses processos não cooperam entre si, isto é, não trocam informações. Nesse exemplo, temos apenas a execução de 10 instâncias no mesmo programa seqüencial, e não um programa concorrente.

Motivação

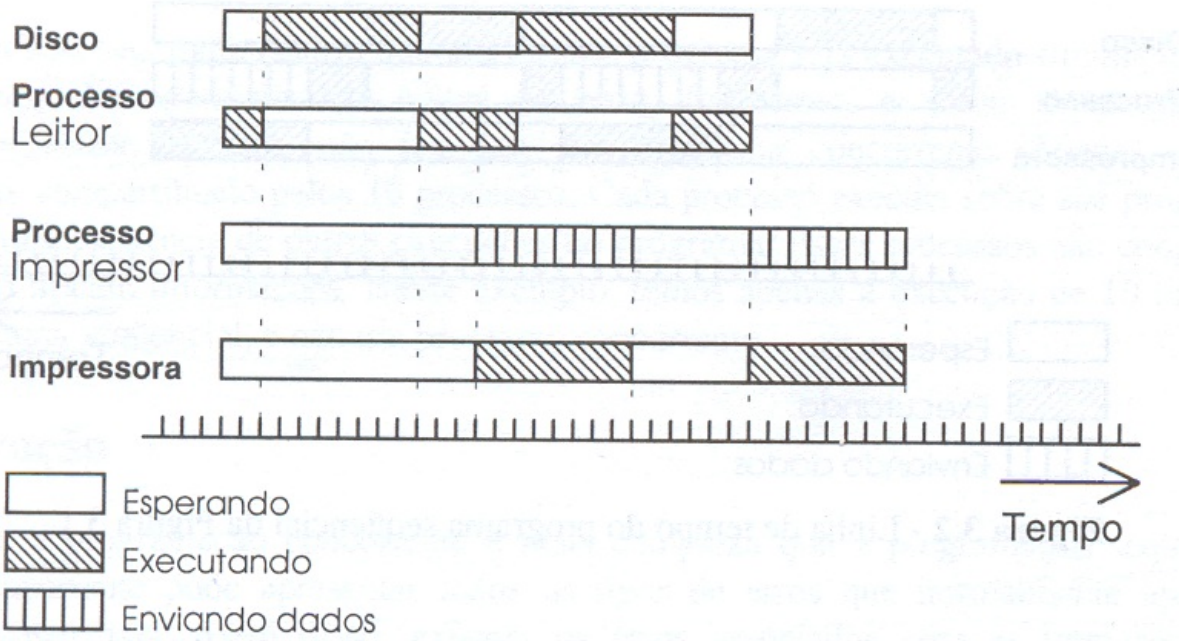
Notadamente, a programação concorrente é mais complexa que a programação seqüencial. Um programa concorrente pode apresentar todos os tipos de erros que normalmente aparecem em programas seqüenciais. Não contente, existem os erros associados com as interações entre os processos.

Mesmo com todas essas complicações, existem muitas áreas nas quais a programação concorrente é útil. Em sistemas nos quais existem vários processadores (máquinas paralelas ou sistemas distribuídos), é possível aproveitar esse paralelismo explicitamente e acelerar a execução do programa. Mesmo em sistemas com um único processador, existem razões para o seu uso em determinados tipos de aplicações.

Na figura abaixo temos um programa concorrente para realizar a impressão de um arquivo.



Dois processos dividem o trabalho. O processo leitor é responsável por ler registros do arquivo, formatar e colocar em um buffer na memória. O processo impressor retira os dados do buffer e envia para impressora. Supõe-se que os dois processos possuem acesso à memória onde está o buffer.

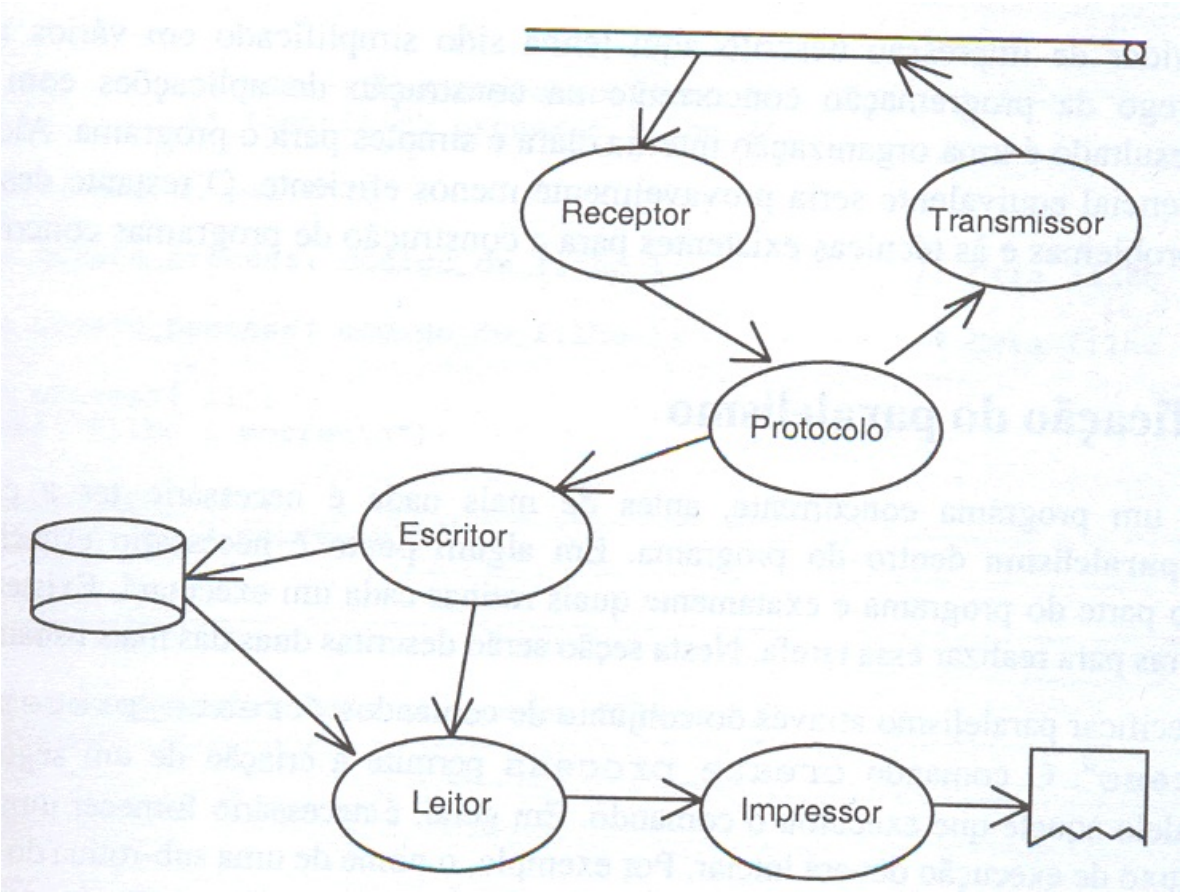


O programa concorrente é mais eficiente, pois consegue manter o disco e a impressora trabalhando simultaneamente. O tempo total para realizar a impressão do arquivo é menor quando a solução concorrente é empregada. É claro que a solução possui limitações. Se o leitor sempre for mais rápido, o buffer ficará cheio, e então o processo leitor terá que esperar até que o processo impressor retire algo do buffer. Por outro

lado, se o buffer ficar vazio, ele terá que esperar pelo processo leitor. De qualquer forma, a solução concorrente jamais será mais lenta que a solução seqüencial.

A maior motivação para a programação concorrente é a engenharia de software. Aplicações inerentemente paralelas – que possuem paralelismo intrínseco – são mais facilmente construídas se programação concorrente é utilizada.

Caso seu problema seja preparar um servidor de impressão, teremos esse paralelismo intrínseco. Uma das possíveis soluções para a organização interna do programa concorrente "servidor de impressão" é descrito abaixo.



Cada círculo representa um processo. Cada flecha representa a passagem de dados de um processo para o outro. Essa passagem de dados pode ser feita, por exemplo, através de variáveis compartilhadas pelos processos envolvidos na comunicação. O processo "receptor" é responsável por receber mensagens da rede local. Ele faz isso através de chamadas de sistema apropriadas e descarta as mensagens com erro. As mensagens são passadas para o processo "protocolo". Ele analisa o conteúdo das mensagens recebidas "a luz do protocolo de comunicação suportado pelo servidor de impressão. É possível que seja necessário a geração e o envio de mensagens de resposta. O processo

"protocolo" gera mensagens a serem enviadas e passa-as para o processo "transmissor", que as envia através de chamadas de sistema apropriadas. Algumas mensagens contêm pedaços de arquivos a serem impressos. É suposto aqui que as mensagens são da ordem de alguns Kbytes, assim sendo, um arquivo deve ser dividido em várias mensagens para transmissão através da rede. Quando o processo "protocolo" identifica uma mensagem que contém um pedaço de arquivo, ele passa esse pedaço de arquivo para o processo "escritor". Passa também a identificação do arquivo ao qual o pedaço em questão pertence. Cabe ao processo "escritor" usar as chamadas de sistema apropriadas para escrever no disco. Quando o pedaço de arquivo em questão é o último de seu arquivo, o processo "escritor" passa para o processo "leitor" o nome do arquivo, que está pronto para ser impresso.

O processo "leitor" executa um laço externo no qual ele pega um nome de arquivo, envia o conteúdo para o processo "impressor" e então remove o arquivo lido. O envio de conteúdo para o processo "impressor" é feito através de um laço interno composto pela leitura de uma parte do arquivo e pelo envio dessa parte. Finalmente, o processo "impressor" é encarregado de envia os pedaços de arquivo que ele recebe para a impressora.

Especificação do Paralelismo

Para construir um programa concorrente, antes de mais nada, é necessário ter a capacidade de especificar o paralelismo dentro do programa. Em algum ponto é necessário especificar quantos processos farão parte do programa e exatamente quais rotinas cada um executará. Existem, na prática, diversas maneiras para realizar essa tarefa. Aqui vamos ver as duas mais usuais.

É possível especificar paralelismo através do conjunto de comandos: "`create_process`", "`exit`" e "`wait_process`". O comando `create process` permite a criação de um segundo fluxo de execução, paralelo àquele que executou o comando. Em geral, é necessário fornecer uma indicação de onde o novo fluxo de execução deverá iniciar. Por exemplo, uma sub-rotina do programa.

Os comandos "`exit`" e "`wait_process`" são auxiliares ao `create_process`. Quando o comando `exit` é executado, o fluxo de controle que o executa é imediatamente terminado. Um comando imediatamente após o comando `exit` jamais será executado. O comando `wait_process` permite que um fluxo de execução espere outro fluxo terminar. Em geral, é necessário fornecer uma indicação do fluxo de execução cujo término está sendo esperado. Por exemplo, um número inteiro retornado pelo comando `create_process`.

Vamos usar a sintaxe da linguagem C para exemplificar. Nesse programa, o processo inicial (pai) executa duas vezes o comando

`create_process`, criando dois processos adicionais (filhos 1 e 2). Ele então espera que cada um dos filhos termine, escreve uma mensagem para cada um e termina também. Os dois processos criados executam a mesma função "código do filho". Eles apenas colocam uma mensagem na tela e terminam. Uma possível saída para esse programa é:

```
Alo do pai
Alo do filho
Alo do filho
Filho1 morreu
Filho 2 morreu
```

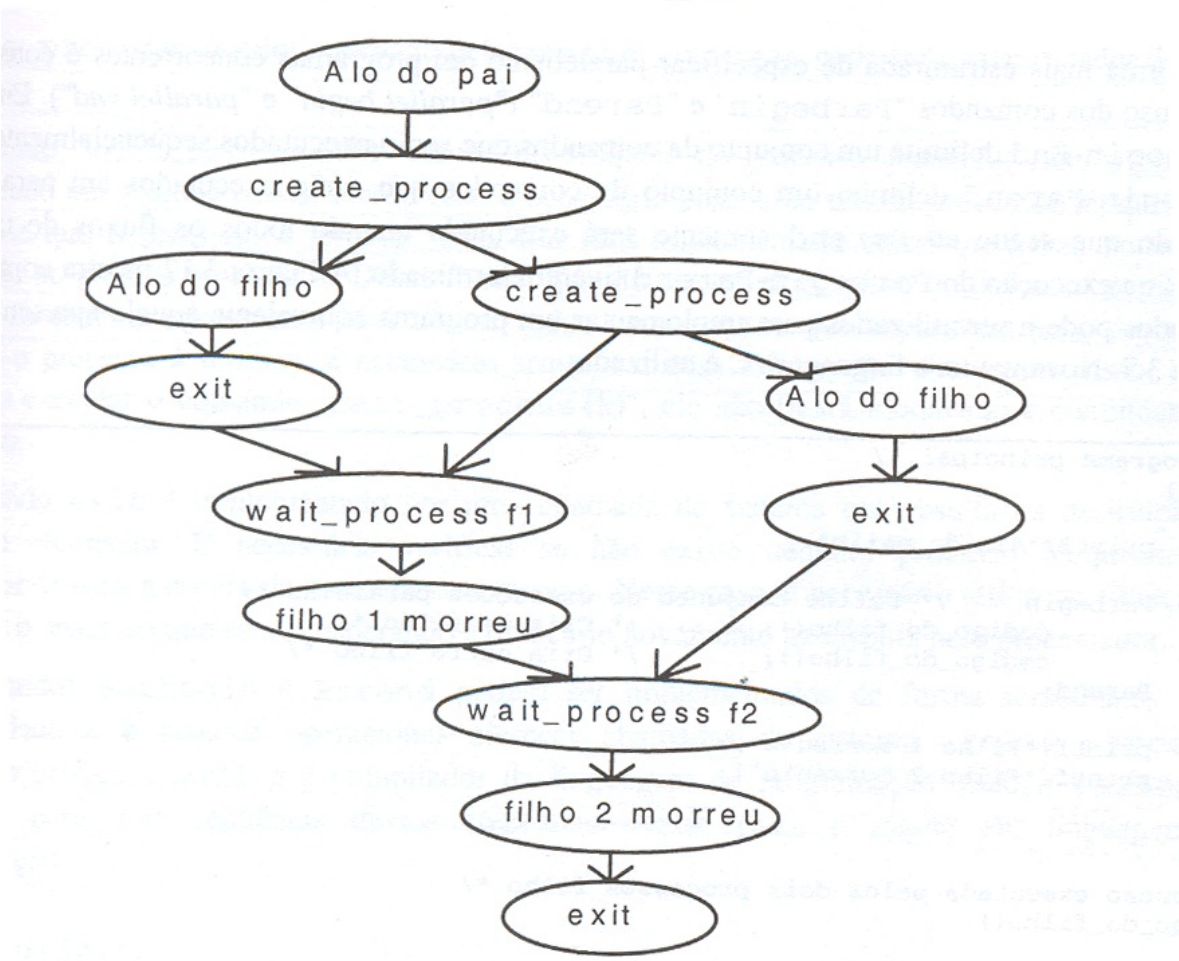
O importante aqui é salientar que processos paralelos podem ser executados em qualquer ordem. Na saída anterior, o segundo filho foi executado antes que o pai percebesse a morte do primeiro filho. Caso o segundo filho tivesse executado após a morte do primeiro filho, a saída seria:

```
Alo do pai
Alo do filho
Filho1 morreu
Alo do filho
Filho 2 morreu
```

Abaixo o código:

```
/* Programa principal */
main()
{
      int f1;      /* Identifica processo filho 1*/
      int f2;      /* Identifica processo filho 2*/

      printf("Alo do pai\n");

      f1 = create_process( codigo_do_filho );          /* Cria filho 1 */

      f2 = create_process( codigo_do_filho );          /* Cria filho 2 */

      wait_process( f1);
      printf("Filho 1 morreu\n");

      wait_process( f2);
      printf("Filho 2 morreu\n");

      exit();
}

/* Funcao executada pelos dois processos filhos */
codigo_do_filho()
{
      printf("Alo do filho\n");
      exit();
}
```

Essa é uma característica importante dos programas concorrentes. Duas execuções consecutivas do mesmo programa, com os mesmos dados de entrada, podem gerar resultados diferentes. Isso não é necessariamente um erro. Cabe ao programador fazer com que todos os resultados possíveis sejam igualmente corretos. Mais adiante, serão apresentados mecanismos capazes de controlar explicitamente a ordem de execução dos processos, se assim for desejado. Para ilustrar o programa acima, vejamos o gráfico dele:



Agora vamos supor que o programador deseje que o processo do filho 1 termine para depois iniciarmos o processo do filho 2. Abaixo vai o código:

```
/* Programa principal */
main()
{
      int f1;       /* Identifica processo filho 1*/
      int f2;       /* Identifica processo filho 2*/

      printf("Alo do pai\n");

      f1 = create_process( codigo_do_filho );          /* Cria filho 1 */
      wait_process( f1);
      printf("Filho 1 morreu\n");

      f2 = create_process( codigo_do_filho );          /* Cria filho 2 */
      wait_process( f2);
      printf("Filho 2 morreu\n");

      exit();
}

/* Funcao executada pelos dois processos filho */
codigo_do_filho()
{
      printf("Alo do filho\n");
      exit();
}
```

Uma forma mais estruturada de especificar paralelismo em programas concorrentes é conseguida com o uso dos comandos "`parbegin`" e "`parend`" ("`parallel begin`" e "`parallel end`"). Enquanto o par `Begin-End` delimita um conjunto de comandos que serão executados seqüencialmente, o par `Parbegin-Parend` delimita um conjunto de comandos que serão executados paralelamente. O comando que segue ao `Parend` somente será executado quando todos os fluxos de controle criados na execução do `Parbegin-Parend` tiverem terminado. Segue-se outra maneira de trabalhar a programação original.

```
/* Programa principal */
main()
{
      printf("Alo do pai\n");

      Parbegin     /* Define conjunto de execuções paralelas */
            codigo_do_filho();       /* Cria um filho */
            codigo_do_filho();       /* Cria outro filho */
      Parend;

      printf("Filho 1 morreu\n");
      printf("Filho 2 morreu\n");

}

/* Funcao executada pelos dois processos filho */
codigo_do_filho()
{
      printf("Alo do filho\n");
}
```

Outra possibilidade é o sistema operacional oferecer chamadas de sistema create_process, wait_process e exit e o compilador da linguagem de programação traduzir Parbegin-Parend para uma seqüência dessas chamadas. Vamos ver um pedaço da programação:

```
Inicio();
Parbegin
     Comando_1();
     Comando_2();
Parend;
Fim();
```

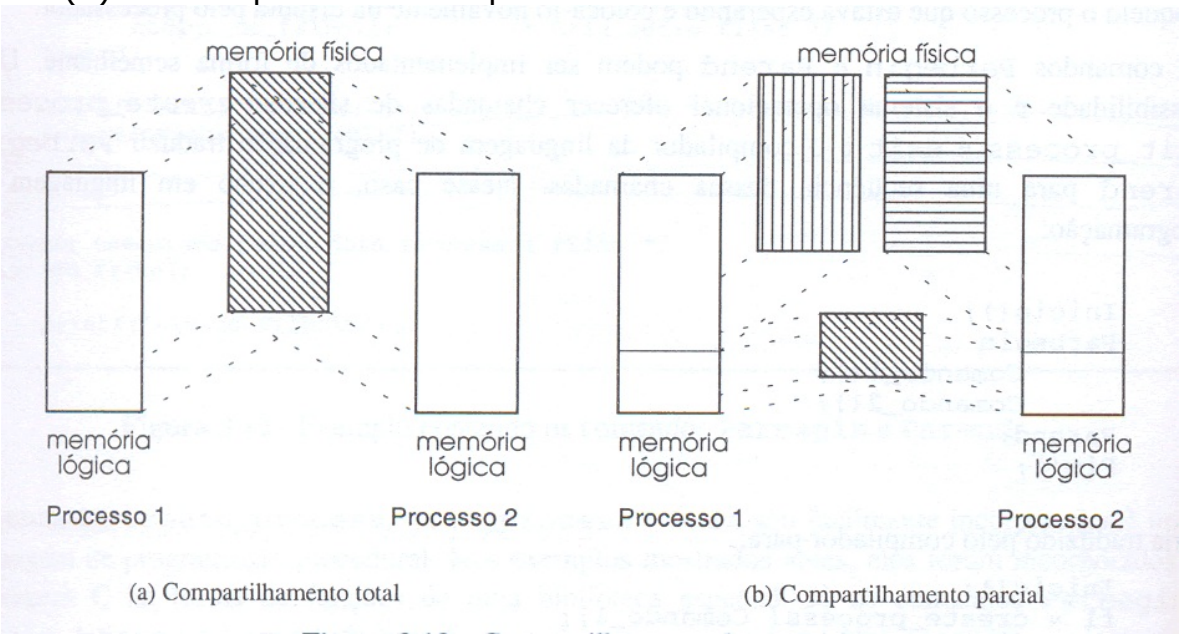
Que, no compilador ficará:

```
Inicio();
f1 = create_process( Comando_1);
f2 = create_process( Comando_2);
wait_process( f1);
wait_process( f2);
Fim();
```

A especificação de paralelismo é um aspecto importante da programação concorrente, mas não é o único.

Problemas de seção crítica

A quantidade exata de memória compartilhada entre os processos pode variar conforme o programa. Processos podem compartilhar todo o seu espaço de endereçamento, apenas um segmento de memória, algumas estruturas de dados ou algumas variáveis. O sistema operacional arranja para que processos acessem as mesmas posições de memória. Abaixo vemos em (a) o compartilhamento total da memória e em (b) um compartilhamento parcial.



Sempre que o processo escritor completa a escrita de um arquivo no disco, ele deve passar o nome desse arquivo para o processo leitor. É possível que, nesse momento, o processo leitor ainda não tenha concluído a leitura dos arquivos recebidos anteriormente. Logo, uma forma natural de implementar essa comunicação é usar uma fila de nomes de arquivos. Ao completar a escrita de um arquivo, o processo escritor insere o nome do arquivo no fim da fila. Ao concluir a leitura de um arquivo, o processo leitor retira do início da fila o nome do próximo arquivo a ser lido. A impressão dos arquivos vai ocorrer na mesma ordem na qual eles terminaram de ser escritos no disco.

Como o tamanho da fila é variável, vamos usar uma lista encadeada para implementá-la. Cada elemento da fila será composto por dois campos: um nome de arquivo e um apontador para o próximo elemento da fila. Vamos também ,manter um apontador para o início da fila e outro para o fim da fila, acelerando a remoção e a inserção de elementos. Usando C, ficaria assim a estrutura:

```
Struct registro{
     Char nome_do_arquivo[64];
```

```
     Struct registro *próximo;
}
struct registro *inicio = NULL;          /* aponta primeiro
elemento */
struct registro *fim = NULL;             /* aponta último
elemento */
```

A seguir, o código **Escritor**, que testa se a fila está vazia, comparando a variável "inicio" com o valor "NULL". Caso a fila esteja vazia, tanto "inicio" como "fim" passam a apontar o novo e único elemento da fila. Caso a fila não esteja vazia, "inicio" não é alterado. O campo "próximo" do atual último elemento passa a apontar para o novo elemento, que também deve ser apontado por "fim", pois é o novo último elemento. Abaixo, o código.

```
void escritor( void)
{
     struct registro *registro_novo;

     ...   /* Novo registro é apontado por "registro_novo" */

     /* Novo nome é sempre inserido no fim da fila */
     registro_novo->proximo = NULL;

     if( inicio == NULL )
          {     /* Fila vazia */
                inicio = registro_novo;
                fim = registro_novo;
          }
     else
          {     /* Fila não vazia */
                fim->proximo = registro_novo;
                fim = registro_novo;
          }
}
```

Já o processo **Leitor**, fica preso em um laço até que exista algum elemento na fila. Ele então retira o primeiro elemento da fila, fazendo a variável "inicio" apontar para o segundo elemento da fila, ou seja, para aquele indicado no campo "próximo" do atual primeiro elemento. Caso o elemento removido fosse o único da fila, seu campo "próximo" conteria "NULL", e esse valor seria copiado para "inicio". Nesse caso, o valor "NULL" também deverá ser copiado para a variável "fim". Abaixo, o código.

```
void leitor( void)
{
      struct registro *ler;

      ...

      /* Espera até existir algum nome na fila */
      while( inicio == NULL )
            ;

      /* Retira o primeiro nome da fila */
      ler = inicio;
      inicio = inicio->proximo;

      /* Se era o único, acerta apontador de fim da fila */
      if( inicio == NULL )
            fim = NULL;

      ...   /* Lê o arquivo cujo nome é indicado por "ler" */
}
```

Os códigos acima funcionarão perfeitamente enquanto forem executados em momentos distintos. Os problemas surgem quando ambos os processos tentam acessar a mesma estrutura de dados ao mesmo tempo. Isso pode acontecer pois os processos Escritor e Leitor trabalham com relativa independência, e não é possível determinar antecipadamente em que instantes vão ocorrer os chaveamentos de contexto.

Vamos agora ilustrar uma seqüência de eventos capaz de gerar um erro. Suponha que a fila contenha apenas um elemento. (A) Nesse momento o processo Leitor conclui a leitura do arquivo anterior e vai retirar um novo nome de arquivo da fila. Ele faria a mesma coisa da imagem anterior, porém, ao chegar ao comando

```
If(inicio==NULL)
```

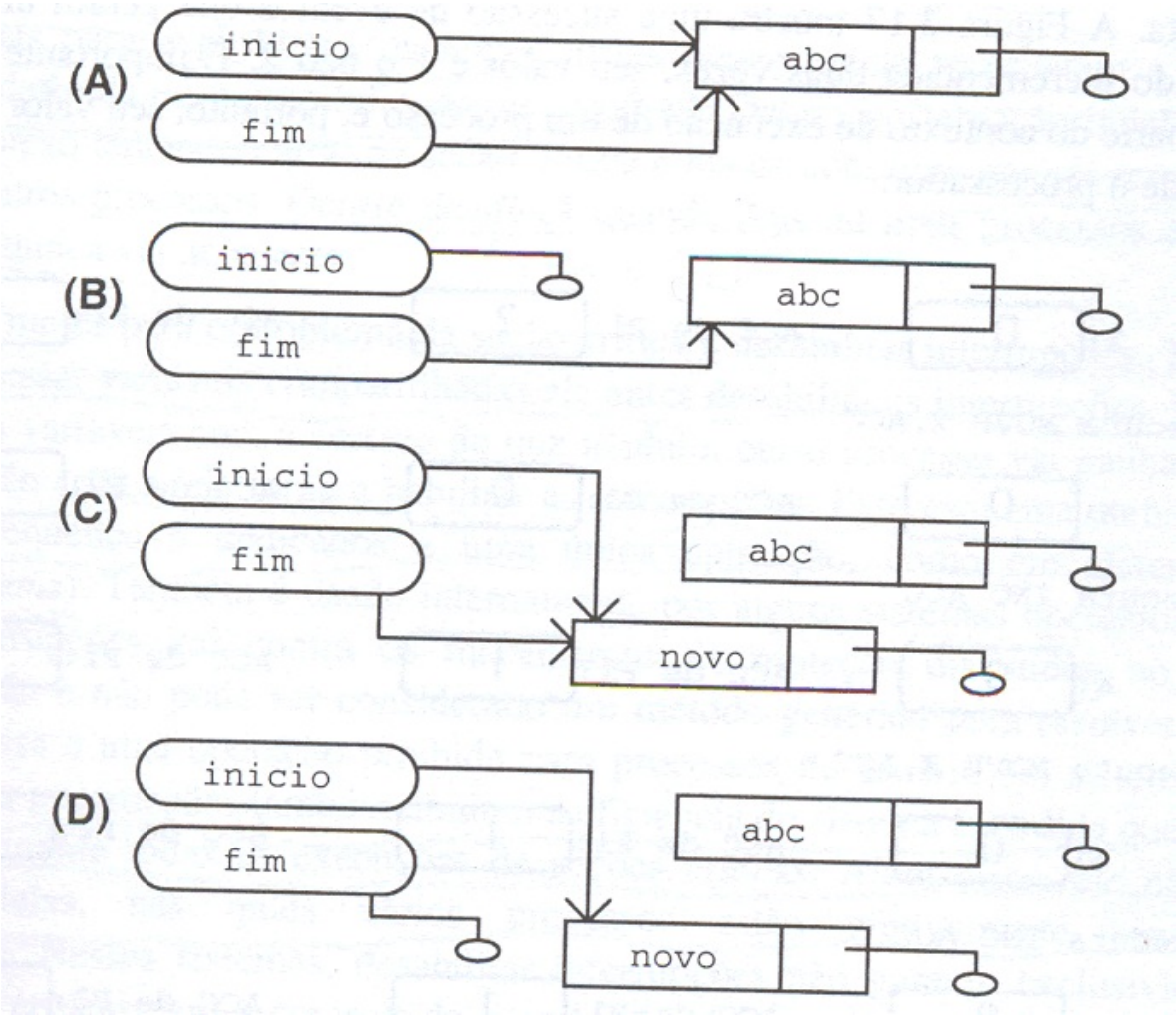
termina sua fatia de tempo, e ele é suspenso. A estrutura de dados fica inconsistente (B), pois o Leitor não teve tempo de concluir sua alteração.

Vamos supor agora que o processo Escritor recebe o processador. Ele concluiu a recepção de um arquivo e vai colocar o seu nome na fila. O processo Escritor executa o código mostrado na figura do processo Escritor e deixa a fila como em (C). Observe que o novo elemento foi inserido.

Depois de algum tempo é novamente a vez do processo Leitor executar. Ele retoma sua execução do ponto em que havia sido suspenso, ou seja, no comando

```
Fim = NULL;
```

Esse comando faz com que a fila adquira a forma mostrada em (D), o que vai resultar em um erro quando um novo elemento for inserido na fila.



Esse tipo de erro é típico de programas concorrentes. Para que o erro se manifeste, é necessária uma seqüência específica de eventos e chaveamentos de contexto. Dessa forma, o programa vai executar corretamente muitas vezes até executar um erro. Caso o programador execute novamente o programa com o objetivo de identificar o erro, provavelmente ele não acontecerá. Ou seja, é muito difícil reproduzir um erro em programas concorrentes.

O erro aqui descrito acontece porque dois processos, Escritor e Leitor, acessam a mesma estrutura de dados simultaneamente. A solução para isso é controlar o acesso dos processos a essas variáveis compartilhadas de modo a garantir que um processo não acesse a estrutura de dados enquanto o outro a está acessando.

Vamos chamar de **seção crítica** aquela parte do código de um processo que acessa uma estrutura de dados compartilhada. O problema da sessão crítica está em garantir que, quando um processo

está executando sua seção crítica, nenhum outro processo entre na sua respectiva seção crítica.

Uma solução para o problema de seção crítica estará correta quando apresentar as seguintes 4 propriedades:

Existe exclusividade mútua entre os processos com referência a execução das respectivas seções críticas.

Quando um processo P deseja entrar na seção crítica e nenhum outro processo está executando a sua seção crítica, o processo P não é impedido de entrar.

Nenhum processo pode ter seu ingresso na seção crítica postergado indefinidamente, ou seja, ficar esperando para sempre.

A solução não depende das velocidades relativas dos processos.

Soluções erradas para o problema da seção crítica normalmente apresentam a possibilidade de postergação indefinida ou a possibilidade de deadlock. Ocorre postergação indefinida quando um processo está preso tentando entrar na seção crítica e nunca consegue por ser sempre preterido em benefício de outros processos. Ocorre deadlock quando dois ou mais processos estão à espera de um evento que nunca vai ocorrer.

Uma solução simples para o problema da seção crítica é desabilitar interrupções. Toda vez que um processo vai acessar variáveis compartilhadas ele antes desabilita as interrupções. Ao final da seção crítica, ele torna a habilitar as interrupções. Esse esquema é efetivamente usado em sistemas pequenos e dedicados a uma única aplicação como em sistemas embutidos (embedded systems – celulares, por exemplo)

Por outro lado, desabilitar as interrupções não pode ser considerado um modo genérico de acabar com os problemas, até porque vai contra os mecanismos de proteção do próprio SO. Essa é uma operação proibida para processos de usuários de sistemas gerais. Desabilitar interrupções também diminui a eficiência do sistema, uma vez que os periféricos não são atendidos durante todas as execuções de seções críticas. Além disso, não funciona em máquinas paralelas, onde vários processos são executados simultaneamente.

rodrigoscama@yahoo.com.br http://scama.norbis.com.br

Spin-Lock

Uma solução possível para problemas de seção crítica é o chamdo spin-lock, ou Test-and-set. Essa solução consiste em uma instrução de máquina que troca (swap) o valor contido em uma posição da memória com o valor contido em um registrador. A posição da memória e o registrado a serem utilizados são especificados como operandos da instrução. Temos, então:

Instrução SWAP (reg,mem)

[mem] -> aux "copia conteúdo da posição [mem] para um registrador auxiliar"
reg -> [mem] "copia conteúdo do registrador reg para a posição [mem] da memória"
aux -> reg "copia conteúdo do registrador auxiliar para o registrador reg"

É essencial que o processador execute toda a instrução SWAP sem interrupções e sem perder o acesso exclusivo ao barramento do computador. Isso é normal em máquinas com apenas um processador.

A seção crítica será protegida por uma variável que ocupará a posição [mem] da memória. Essa variável é geralmente chamada de *lock* (fechadura). Quando lock contém "0", a seção crítica está livre. Quando lock contém "1", ela está ocupada. A variável é sempre inicializada com "0".

Antes de entrar na seção crítica, um processo precisa "fechar a porta", colocando "1" em lock. Entretanto, ele só pode fazer isso se a porta estiver aberta. Ao sair da seção crítica, basta colocar "0" na variável lock.

A vantagem do spin-lock é sua simplicidade, aliada ao fato de que não é necessário desabilitar interrupções. A desvantagem dele é o busy-waiting, ou seja, o processo que está no laço de espera executando "swap" ocupa o processador enquanto espera. Não contente, existe a possibilidade de postergação indefinida, ou seja, um processo "azarado" perde sempre a disputa de quem "pega antes" o "0".

Na prática, o spin-lock é muito usado em situações onde a seção crítica é pequena, fazendo com que seus problemas desapareçam.

rodrigoscama@yahoo.com.br http://scama.norbis.com.br

Semáforos

O mecanismo de sincronização semáforo foi inventado em 1965 pelo matemático holandês E.W. Dijkstra. O semáforo é um tipo abstrato de dado composto por um valor inteiro e uma fila de processos.Somente duas operações são permitidas no semáforo: **P** (testar) e **V** (incrementar).

Quando um processo executa a operação P em um semáforo, o seu valor inteiro é decrementado. Caso o novo valor do semáforo seja negativo, o processo é bloqueado e ele vai pro fim da fila. Quando um processo executa a operação V, o seu valor inteiro é incrementado. Caso exista algum processo bloqueado na fila desse semáforo, o primeiro processo da fila é liberado. Observe o funcionamento através do semáforo **S**:

```
P(S):
     S.valor = S.valor -1;
     Se S.valor < 0
          Então bloqueia o processo , insere em S.fila

V(S):
     S.valor = S.valor +1;
     Se S.fila não está vazia
          Então retira processo P de S.fila, acorda P
```

Para que semáforos funcionem corretamente, é essencial que as operações P e V sejam atômicas, isto é, uma operação P ou V não pode ser interrompida.

Assim, a proteção de seção crítica torna-se muito simples. Para cada estrutura de dados compartilhada, deve ser criado um semáforo S inicializado com o valor 1. Todo processo, antes de acessar essa estrutura, deve executar um P(S), ou seja, a operação P sobre o semáforo S associado com a estrutura de dados em questão. Ao sair da seção crítica, o processo executa V(S).

Uma variação muito comum de semáforos são as construções mutex ou semáforo binário. Nesse caso, temos um semáforo capaz de assumir apenas os valores 0 e 1. Ele pode ser visto como uma variável tipo mutex, a qual assume apenas os valores livre e ocupado. Nesse caso, P e V são chamados, respectivamente, de *lock* e *unlock*.

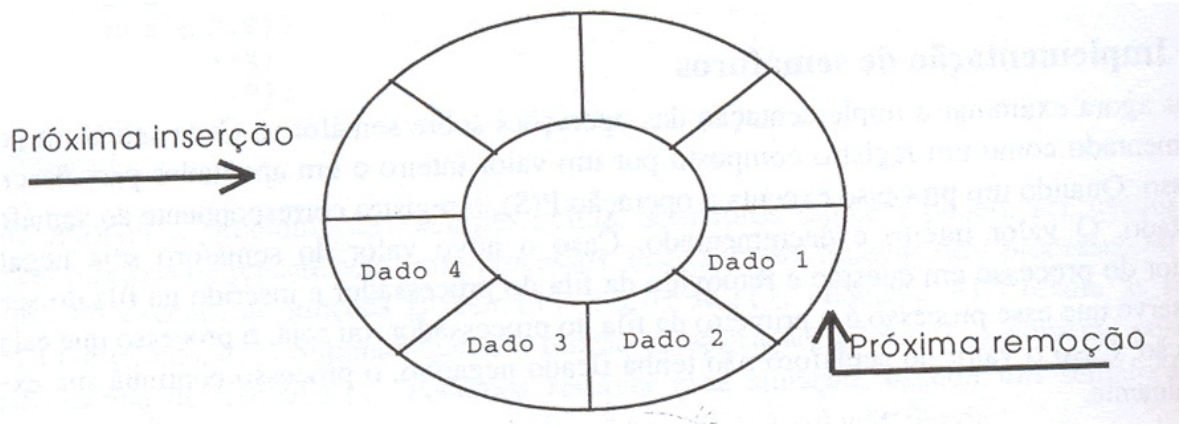
Vejamos seu funcionamento:

```
lock(x):
     Se x está livre
          Então -> marca x como ocupado
          Senão -> insere processo no fim da fila "x"
unlock(x):
```

```
Se fila "x" está vazia
     Então -> marca x como livre
     Senão -> libera processo do início da fila "x"
```

Problema do Produtor-consumidor

É comum, em programas concorrentes, a existência de um processo que produz continuamente informações que serão usadas por outro processo. Essa situação é chamada de problema de produtor-consumidor. O dado produzido e consumido pode ser um número, uma string, um registro, etc. Entre produtor e consumidor existe um buffer no qual os dados já produzidos mas ainda não consumidos são armazenados temporariamente. Dessa forma, produtor e consumidor podem trabalhar em paralelo. Quando o produtor tenta colocar um dado no buffer e ele está cheio, o produtor deve ser bloqueado até que o consumidor retire algo do buffer, liberando a entrada. O mesmo acontece com o consumidor quando tenta retirar um dado do buffer vazio. O buffer é tipicamente implementado de maneira circular, como vemos abaixo.



Mensagens

Agora vamos considerar programas concorrentes construídos a partir da troca de mensagens entre processos. Uma aplicação imediata para esse mecanismo está nos sistemas distribuídos. Nesse caso, processos executando em diferentes máquinas não possuem variáveis compartilhadas. Eles trocam informações através de mensagens via rede de comunicação.

Tipicamente o mecanismo que permite a troca de mensagens é implementado pelo sistema operacional. Ele é acessado através de duas chamadas de sistema básicas: *send* e *receive*.

A chamada *send* possui pelo menos dois parâmetros: a mensagem a ser enviada e o destinatário da mensagem. Existem dois tipos de endereçamento:
Direto – quando a mensagem é endereçada explicitamente a um processo em particular, e o processo remetente deve conhecer o identificador do processo destinatário.
Indireto – Quando o sistema operacional implementa um recurso chamado de caixa postal.Mensagens não são enviadas para processos, e sim para caixas postais. As mensagens devem ser retiradas da caixa postal por outro processo. O sistema operacional normalmente fornece chamadas de sistema que permitem a criação e a destruição de caixas postais. Também é o sistema operacional que controla quais processos possuem o direito de ler ou escrever na caixa postal. O processo remetente somente pode enviar mensagens para as caixas postais das quais ele conhece o identificador, ou para caixas postais que possuam identificador público. Além disso, deve possuir o direito de escrita sobre a caixa em questão.
A chamada *receive* possui pelo menos um parâmetro: o endereço da variável do processo onde deverá ser colocada a mensagem lida.

Quando endereçamento indireto é usado, a chamada receive possui um segundo parâmetro o identificador da caixa postal em questão. Um processo somente pode retirar mensagens das caixas postais das quais ela conhece o identificador, ou das caixas postais que possuem um identificador público. Além disso, ele deve possuir o direito de leitura sobre a caixa postal em questão.

É importante observar que caixas postais permitem uma comunicação de N para N processos, enquanto o endereçamento direto permite uma comunicação de 1 para 1 processo. Existem sistemas que permitem a criação de grupos de processos. Nesse caso, um processo pode enviar uma mensagem para um grupo de processos, ao invés de um processo em particular.

Alguns sistemas oferecem buffers para as mensagens que foram enviadas, mas ainda não recebidas. Nesse caso, o processo remetente retorna imediatamente da chamada send e o sistema operacional armazena a mensagem. Ela será entregue quando o processo destinatário executar um receive. Como os recursos do sistema são limitados, uma solução alternativa é armazenar até um determinado limite, medido em número de mensagens ou em bytes. Quando esse limite é atingido, o sistema operacional passa a bloquear os processos remetentes cujos respectivos destinatários ainda não executaram o receive.

Uma forma popular de usar mensagens é a **chamada remota de procedimento (RPC, remote procedure call)**. Nesse caso, o processo chamador utiliza o Send para solicitar a execução de uma

determinada rotina em outro processo, passando os parâmetros para essa rotina como parte da mensagem. Em seguida, ele executa um receive, para receber os resultados da execução da rotina. No outro lado, um processo executa um receive para receber o pedido de execução de uma rotina, executa a rotina com os parâmetros recebidos na mensagem e utiliza send para enviar os resultados da execução ao processo chamador. Dessa forma, a comunicação entre processos baseada em mensagens é disfarçada de uma simples chamada de sub-rotina.

Em sistemas distribuídos, podem ocorrer erros na rede de comunicação. É responsabilidade dos protocolos de comunicação prover mecanismos para a recuperação de falhas simples como, por exemplo, a perda de uma mensagem. Em caso extremo, a máquina na qual está um dos processos envolvidos na comunicação pode parar completamente. Nesse caso, o SO deve ter cuidado de detectar a situação e não deixar outros processos envolvidos na comunicação bloqueados para sempre.