Sistemas Operacionais

Profa. Dra. Kalinka Regina Lucas Jaquie Castelo Branco
kalinka@icmc.usp.br

Apresentação baseada nos slides
do Prof. Dr. Antônio Carlos Sementille e da Profa. Dra. Luciana A. F. Martimiano e nas transparências fornecidas no site de compra do livro "Sistemas Operacionais Modernos"

Comunicação de Processos

- Processos precisam se comunicar;
- Processos competem por recursos
- Três aspectos importantes:
  - Como um processo passa informação para outro processo;
  - Como garantir que processos não invadam espaços uns dos outros;
  - Dependência entre processos: sequência adequada.

Comunicação de Processos

Especificação de Execução Concorrente

Questão importante na estruturação de Algoritmos paralelos → **Como decompor um problema em um conjunto de processos paralelos**

Algumas formas de se expressar uma execução concorrente (usadas em algumas linguagens e sistemas operacionais)

- **Co-rotinas**
- **Declarações FORK/JOIN**
- **Declarações COBEGIN/COEND**
- **Declarações de Processos Concorrentes**

Comunicação de Processos

Co-Rotinas

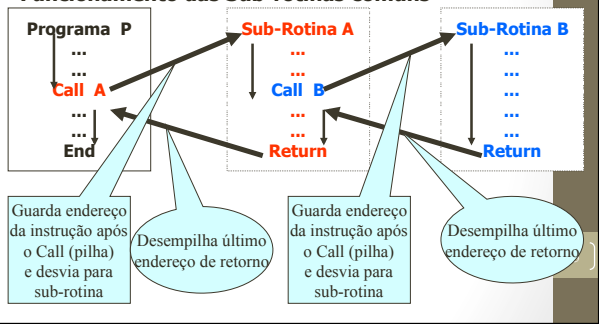
As co-rotinas são parecidas com sub-rotinas (ou procedimentos), diferindo apenas na forma de transferência de controle, realizada na chamada e no retorno

As co-rotinas possuem um ponto de entrada, mas pode representar diversos pontos intermediários de entrada e saída

A transferência de controle entre eles é realizada por meio do endereçamento explícito e de livre escolha do programador (por meio de comandos do tipo TRANSFER, do Modula-2)

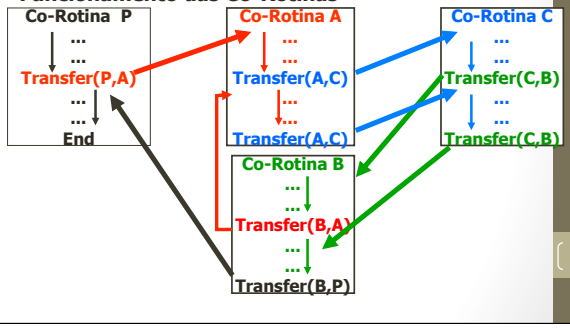
Comunicação de Processos

Funcionamento das Sub-rotinas comuns



Comunicação de Processos

Funcionamento das Co-Rotinas

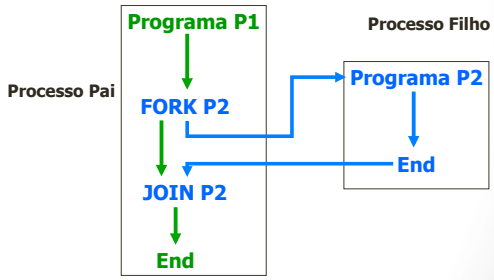


Comunicação de Processos

- **Declarações FORK/JOIN**
  - A declaração `FORK <nome do programa>` determina o início de execução de um determinado programa, de forma concorrente com o programa sendo executado.
  - Para sincronizar-se com o término do programa chamado, o programa chamador deve executar a declaração `JOIN <nome do programa chamado>.`
  - O uso do `FORK/JOIN` permite a concorrência e um mecanismo de criação dinâmica entre processos (criação de múltiplas versões de um mesmo programa -> processo-filho), como no sistema UNIX

Comunicação de Processos

Declarações FORK/JOIN



Comunicação de Processos

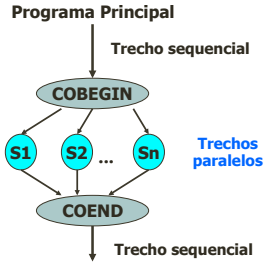
- **Declarações COBEGIN/COEND**
  - Constituem uma forma estruturada de especificar execução concorrente ou paralela de um conjunto de declarações agrupadas da seguinte maneira:

```
COBEGIN
  S1//S2//...//Sn
COEND
```

  Onde:
  - A execução deste trecho de programa provoca a execução concorrente das declarações S1, S2, ..., Sn.
  - Declarações Si podem ser qualquer declaração, incluindo o para COBEGIN/COEND, ou um bloco de declarações locais.
  - Esta execução só termina, quando todas as declarações Si terminarem.

Comunicação de Processos

Declarações COBEGIN/COEND




```
Program Paralelo;
/* declaração de var.e const. globais */
Begin
  /* trecho sequencial */
  ...
  COBEGIN /* trechos  paralelos */
    Begin  /* S1 */
      ...
    End;
    ...
    Begin /* Sn */
      ...
    End;
  COEND
  /* trecho sequencial */
  ...
End.
```


Comunicação de Processos

- **Declarações de Processos Concorrentes**
  - Geralmente, programas de grande porte são estruturados como conjunto de trechos sequenciais de programa, que são executados concorrentemente.
  - Poderiam ser utilizadas as co-rotinas, FORK/JOIN, ou Cobegin/Coend, porém a estrutura de um programa será mais clara se a especificação dessas rotinas explicitar que as mesmas são executadas concorrentemente.
  - Exemplo de linguagens: DP (*Distributed Process*): utiliza um único cobegin e coend – apenas 1 instância de programa; ADA: várias instâncias (processos podem ser criados dinamicamente – pode existir um número variável de processos).

Comunicação de Processos

Declarações de Processos Concorrentes

```
Program Conjunto_Processos;
/* declaração de var.e const. globais */
  ...
Processo Pi;
  /* declaração de var.e const. locais */
    ...
  End;
/* Outros  processos */
Processo Pn;
  /* declaração de var.e const. locais */
    ...
  End;
End.
```

Comunicação de Processos

Mecanismos Simples de Comunicação e Sincronização entre Processos

- **Em um sistema de multiprocessamento ou multiprogramação, os processos geralmente precisam se comunicar com outros processos.**
- **A comunicação entre processos é mais eficiente se for estruturada e não utilizar interrupções.**
- **A seguir, serão vistos alguns destes mecanismos e problemas da comunicação inter-processos.**

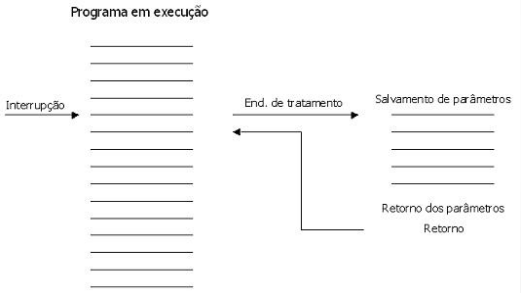
13

Comunicação de Processos

- ***O que são interrupções?***
  - Uma interrupção é um evento externo que faz com que o processador pare a execução do programa corrente e desvie a execução para um bloco de código chamado rotina de interrupção (normalmente são decorrentes de operações de E/S).
  - Ao terminar o tratamento de interrupção o controle retorna ao programa interrompido exatamente no mesmo estado em que estava quando ocorreu a interrupção.

14

Comunicação de Processos

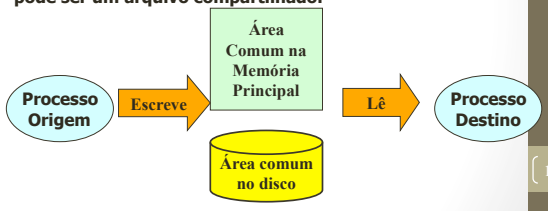


15

Comunicação de Processos

Condições de Corrida

- **Em alguns Sistemas Operacionais: os processos se comunicam por meio de alguma área de armazenamento comum. Esta área pode estar na memória principal ou pode ser um arquivo compartilhado.**



16

Comunicação de Processos

Condições de Corrida

- **Definição de condições de corrida: situações onde dois os mais processos estão lendo ou escrevendo algum dado compartilhado e o resultado depende de quem processa no momento propício.**
- **Depurar programas que contém condições de corrida não é fácil, pois não é possível prever quando o processo será suspenso.**

17

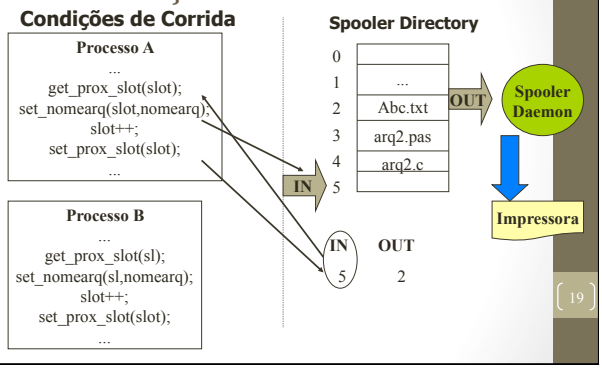
Comunicação de Processos

Condições de Corrida

- **Um exemplo: Print Spooler**
- **Quando um processo deseja imprimir um arquivo, ele coloca o nome do arquivo em uma lista de impressão (*spooler directory*).**
- **Um processo chamado "printer daemon", verifica a lista periodicamente para ver se existe algum arquivo para ser impresso, e se existir, ele os imprime e remove seus nomes da lista.**

18

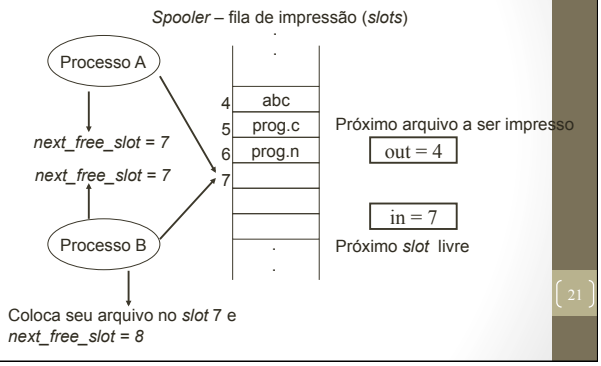
Comunicação de Processos



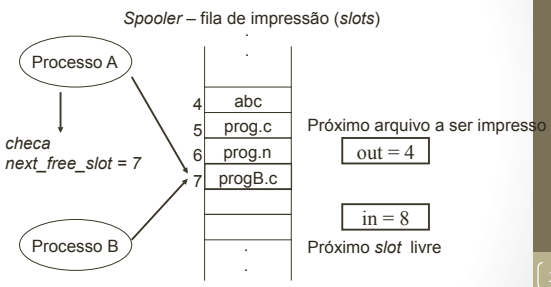
Comunicação de Processos – *Race Conditions*

- *Race Conditions*: processos acessam recursos compartilhados concorrentemente;
  - Recursos: memória, arquivos, impressoras, discos, variáveis;
- Ex.: Impressão: quando um processo deseja imprimir um arquivo, ele coloca o arquivo em um local especial chamado ***spooler*** (tabela). Um outro processo, chamado ***printer spooler***, checa se existe algum arquivo a ser impresso. Se existe, esse arquivo é impresso e retirado do *spooler*. Imagine dois processos que desejam ao mesmo tempo imprimir um arquivo...

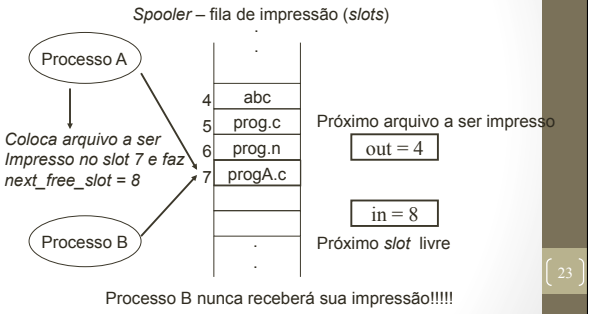
Comunicação de Processos - *Race Conditions*



Comunicação de Processos - *Race Conditions*



Comunicação de Processos - *Race Conditions*



Comunicação de Processos – *Regiões Críticas*

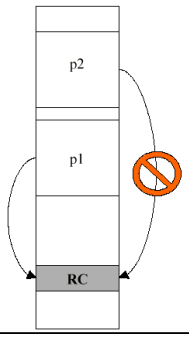
Regiões Críticas

Uma solução para as condições de corrida é proibir que mais de um processo leia ou escreva em uma variável compartilhada ao mesmo tempo.

Esta restrição é conhecida como exclusão mútua, e os trechos de programa de cada processo que usam um recurso compartilhado e são executados um por vez, são denominados seções críticas ou regiões críticas (R.C.).

Comunicação de Processos – *Regiões Críticas*

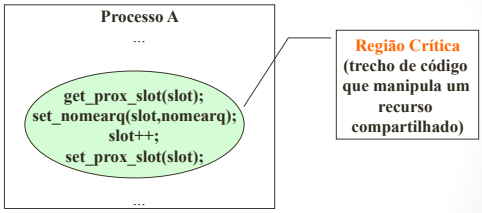
Regiões Críticas



25

Comunicação de Processos – *Regiões Críticas*

Regiões Críticas



26

Regiões Críticas e Exclusão Mútua

- **Região crítica**
  - seção do programa onde são efetuados acessos (para leitura e escrita) a recursos compartilhados por dois ou mais processos
  - é necessário assegurar que dois ou mais processos não se encontrem simultaneamente na região crítica

27

Comunicação de Processos – *Regiões Críticas*

Pergunta: isso quer dizer que uma máquina no Brasil e outra no Japão, cada uma com processos que se comunicam, nunca terão Condições de Disputa?

28

Ex.: Vaga em avião

1. Operador OP1 (no Brasil) lê Cadeira1 vaga;
2. Operador OP2 (no Japão) lê Cadeira1 vaga;
3. Operador OP1 compra Cadeira1;
4. Operador OP2 compra Cadeira1;

29

Solução simples para exclusão mútua

- Caso de venda no avião:
  - apenas um operador pode estar vendendo em um determinado momento;
- Isso gera uma fila de clientes nos computadores;
- Problema: ineficiência!

30

Comunicação de Processos – Regiões Críticas

- Como solucionar problemas de *Race Conditions*???
  - Proibir que mais de um processo leia ou escreva em recursos compartilhados concorrentemente (ao "mesmo tempo")
    - **Recursos compartilhados → regiões críticas**;
  - Exclusão mútua: garantir que um processo não terá acesso à uma região crítica quando outro processo está utilizando essa região;

31

Comunicação de Processos – Exclusão Mútua

- assegura-se a exclusão mútua recorrendo aos mecanismos de sincronização fornecidos pelo SO
- Estas afirmações são válidas também para as *threads* (é ainda mais crítico, pois todas as *threads* dentro do mesmo processo compartilham os mesmos recursos)

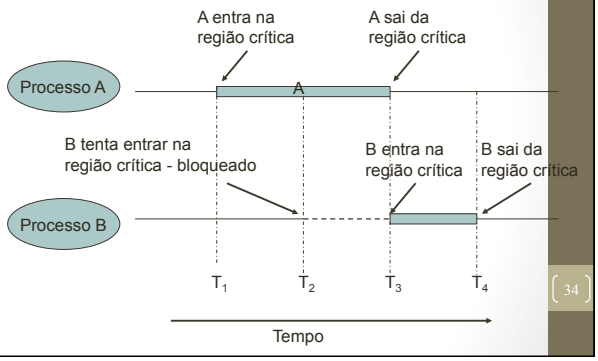
32

Regiões Críticas e Exclusão Mútua

- Regras para programação concorrente (condições para uma boa solução)
  - Dois ou mais processos não podem estar simultaneamente dentro de uma região crítica
  - Não se podem fazer assunções em relação à velocidade e ao número de CPUs
  - Um processo fora da região crítica não deve causar bloqueio a outro processo
  - Um processo não pode esperar infinitamente para entrar na região crítica

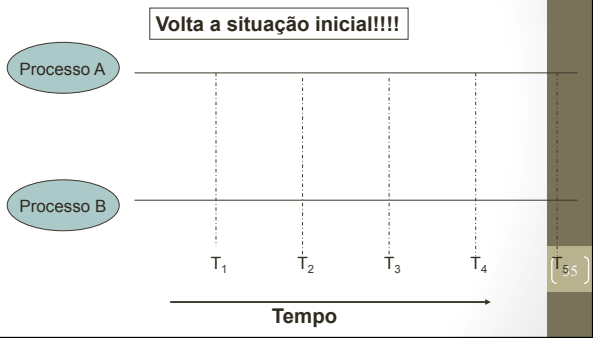
33

Comunicação de Processos – Exclusão Mútua



34

Comunicação de Processos – Exclusão Mútua



35

Soluções

- Exclusão Mútua:
  - **Espera Ocupada**;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - Monitores;
  - Passagem de Mensagem.

36

Comunicação de Processos – Exclusão Mútua

- Espera Ocupada (*Busy Waiting)*: constante checagem por algum valor;
- Algumas soluções para Exclusão Mútua com Espera Ocupada:
  - Desabilitar interrupções;
  - Variáveis de Travamento (*Lock)*;
  - Estrita Alternância (*Strict Alternation)*;
  - Solução de Peterson e Instrução TSL;

Comunicação de Processos – Exclusão Mútua

- Desabilitar interrupções:
  - Processo desabilita todas as suas interrupções ao entrar na região crítica e habilita essas interrupções ao sair da região crítica;
  - Com as interrupções desabilitadas, a CPU não realiza chaveamento entre os processos;
    - Viola condição 2;
  - Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções e não ser finalizado;
    - Viola condição 4;

Comunicação de Processos – Exclusão Mútua

Exclusão Mútua com Espera Ocupada

Desabilitando as Interrupções

SOLUÇÃO MAIS SIMPLES: cada processo desabilita todas as interrupções (inclusive a do relógio) após entrar em sua região crítica, e as reabilita antes de deixá-la.

DESVANTAGENS:
- Processo pode esquecer de reabilitar as interrupções;
- Em sistemas com várias CPUs, desabilitar interrupções em uma CPU não evita que as outras acessem a memória compartilhada.

CONCLUSÃO: é útil que o kernel tenha o poder de desabilitar interrupções, mas não é apropriado que os processos de usuário usem este método de exclusão mútua.

Comunicação de Processos – Exclusão Mútua

- Variáveis *Lock*:
  - O processo que deseja utilizar uma região crítica atribui um valor a uma variável chamada *lock*;
  - Se a variável está com valor 0 (zero) significa que nenhum processo está na região crítica; Se a variável está com valor 1 (um) significa que existe um processo na região crítica;
  - Apresenta o mesmo problema do exemplo do *spooler de impressão*;

Comunicação de Processos – Exclusão Mútua

- Variáveis *Lock* - Problema:
  - Suponha que um processo A leia a variável *lock* com valor 0;
  - Antes que o processo A posso alterar a variável para o valor 1, um processo B é escalonado e altera o valor de *lock* para 1;
  - Quando o processo A for escalonado novamente, ele altera o valor de *lock* para 1, e ambos os processos estão na região crítica;
    - Viola condição 1;

Comunicação de Processos – Exclusão Mútua

- Variáveis *Lock: **lock==0**;*

```
while(true){
    while(lock!=0); //loop
    lock=1;
    critical_region();
    lock=0;
    non-critical_region();
}
```

Processo A

```
while(true){
    while(lock!=0); //loop
    lock=1;
    critical_region();
    lock=0;
    non-critical_region();
}
```

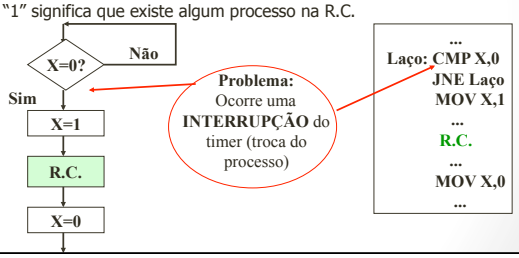
Processo B

Comunicação de Processos – Exclusão Mútua

Exclusão Mútua com Espera Ocupada

Variáveis de Trava

Consiste no uso de uma **variável, compartilhada, de trava**. Se a variável está em zero, significa que nenhum processo está na R.C., e "1" significa que existe algum processo na R.C.



Comunicação de Processos – Exclusão Mútua

- *Strict Alternation*:
  - Fragmentos de programa controlam o acesso às regiões críticas;
  - Variável `turn`, inicialmente em `0`, estabelece qual processo pode entrar na região crítica;

```
while (TRUE)
{
        while (turn!=0); //loop
        critical_region();
        turn = 1;
        noncritical region();}
```

(Processo A)

```
while (TRUE){
        while (turn!=1); //loop
        critical_region();
        turn = 0;
        noncritical region();}
```

(Processo B)

Comunicação de Processos – Exclusão Mútua

- Problema do *Strict Alternation:*
  1. Suponha que o Processo B é mais rápido e saí da região crítica;
  2. Ambos os processos estão fora da região crítica e `turn` com valor 0;
  3. O processo A termina antes de executar sua região não crítica e retorna ao início do *loop*; Como o `turn` está com valor zero, o processo A entra novamente na região crítica, enquanto o processo B ainda está na região não crítica;
  4. Ao sair da região crítica, o processo A atribui o valor 1 à variável `turn` e entra na sua região não crítica;

Comunicação de Processos – Exclusão Mútua

- Problema do *Strict Alternation:*
  5. Novamente ambos os processos estão na região não crítica e a variável `turn` está com valor 1;
  6. Quando o processo A tenta novamente entrar na região crítica, não consegue, pois `turn` ainda está com valor 1;
  7. **Assim, o processo A fica bloqueado pelo processo B que NÃO está na sua região crítica, violando a condição 3**;

Comunicação de Processos – Exclusão Mútua

- Solução de Peterson e Instrução TSL (*Test and Set Lock*):
  - Uma variável (ou programa) é utilizada para bloquear a entrada de um processo na região crítica quando um outro processo está na região;
  - Essa variável é compartilhada pelos processos que concorrem pelo uso da região crítica;
  - Ambas as soluções possuem fragmentos de programas que controlam a entrada e a saída da região crítica;

Comunicação de Processos – Exclusão Mútua

- Instrução TSL: utiliza registradores do hardware;
  - TSL RX, LOCK; (lê o conteúdo de ***lock*** em RX, e armazena um valor diferente de zero (0) em ***lock*** – operação indivisível);
  - ***Lock*** é compartilhada
    - Se ***lock***==0, então região crítica "liberada".
    - Se ***lock***<>0, então região crítica "ocupada".

```
enter_region:
    TSL REGISTER, LOCK        | Copia lock para reg. e lock=1
    CMP REGISTER, #0          | lock valia zero?
    JNE enter_region          | Se sim, entra na região crítica,
                              | Se não, continua no laço
    RET                       | Retorna para o processo chamador

leave_region
     MOVE LOCK, #0            | lock=0
     RET                      | Retorna para o processo chamador
```

Comunicação de Processos – Exclusão Mútua

Instrução TSL (Test and Set Lock)

Esta solução é implementada com **uso do hardware**.

Muitos computadores possuem uma instrução especial, chamada **TSL (test and set lock)**, que funciona assim: ela lê o conteúdo de uma palavra de memória e armazena um valor diferente de zero naquela posição.

Em sistemas multiprocessados: esta instrução trava o barramento de memória, proibindo outras UCPs de acessar a memória até ela terminar.

Comunicação de Processos – Exclusão Mútua

Instrução TSL (Test and Set Lock) - Exemplo

```
ENTRA_RC:
    TSL reg, flag ; copia flag para reg
                  ; e coloca 1 em flag
    CMP reg,0     ; flag era zero?
    JNZ ENTRA_RC  ; se a trava não
                  ; estava ligada,
                  ; volta ao laço
    RET

SAI_RC:
    MOV flag,0  ;  desliga flag
    RET
```

```
PROCESSO
...
CALL ENTRA_RC
...
R.C.
...
CALL SAI_RC
...
```


Comunicação de Processos – Exclusão Mútua

Exclusão Mútua com Espera Ocupada

Considerações Finais

Espera Ocupada: quando um processo deseja entrar na sua região crítica, ele verifica se a entrada é permitida. Se não for, o processo ficará em um laço de espera, até entrar.

Desvantagens:
- desperdiça tempo de UCP;
- pode provocar "**bloqueio perpétuo**" (deadlock) em sistemas com prioridades.

Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - **Primitivas *Sleep/Wakeup***;
  - Semáforos;
  - Monitores;
  - Passagem de Mensagem;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Todas as soluções apresentadas utilizam espera ocupada → processos ficam em estado de espera (*looping)* até que possam utilizar a região crítica:
  - Tempo de processamento da CPU;
  - Situações inesperadas;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Para solucionar esse problema de espera, um par de primitivas *Sleep* e *Wakeup* é utilizado → BLOQUEIO E DESBLOQUEIO de processos.
- A primitiva *Sleep* é uma chamada de sistema que bloqueia o processo que a chamou, ou seja, suspende a execução de tal processo até que outro processo o "acorde";
- A primitiva *Wakeup* é uma chamada de sistema que "acorda" um determinado processo;
- Ambas as primitivas possuem dois parâmetros: o processo sendo manipulado e um endereço de memória para realizar a correspondência entre uma primitiva *Sleep* com sua correspondente *Wakeup*;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Problemas que podem ser solucionados com o uso dessas primitivas:
  - Problema do Produtor/Consumidor (*bounded buffer*): dois processos compartilham um *buffer* de tamanho fixo. O processo produtor coloca dados no *buffer* e o processo consumidor retira dados do *buffer*;
  - Problemas:
    - Produtor deseja colocar dados quando o *buffer* ainda está cheio;
    - Consumidor deseja retirar dados quando o *buffer* está vazio;
    - Solução: colocar os processos para "dormir", até que eles possam ser executados;

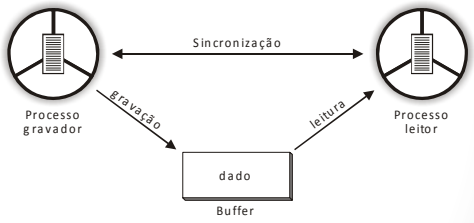
Comunicação de Processos – Primitivas *Sleep/Wakeup*

- *Buffer*: uma variável `count` controla a quantidade de dados presente no *buffer*.
- Produtor: Antes de colocar dados no *buffer*, o processo produtor checa o valor dessa variável. Se a variável está com valor máximo, o processo produtor é colocado para dormir. Caso contrário, o produtor coloca dados no *buffer* e o incrementa.

Comunicação de Processos – Primitivas *Sleep/Wakeup*

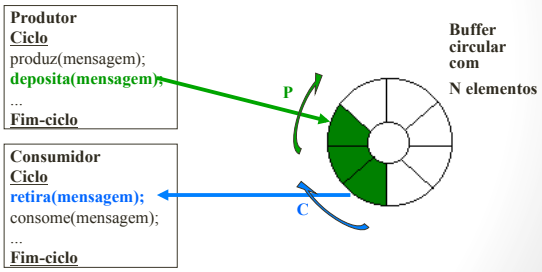
- Consumidor: Antes de retirar dados no *buffer*, o processo consumidor checa o valor da variável `count` para saber se ela está com 0 (zero). Se está, o processo vai "dormir", senão ele retira os dados do *buffer* e decrementa a variável;

Comunicação de Processos Sincronização Produtor-Consumidor



Comunicação de Processos Sincronização Produtor-Consumidor

- **O Problema do Produtor/Consumidor atuando sobre um Buffer Circular**



Comunicação de Processos Sincronização Produtor-Consumidor

- **O Problema do Produtor/Consumidor atuando sobre um Buffer Circular**

Restrições do Problema:

- o produtor não deve exceder a capacidade finita do buffer;
- o consumidor não poderá consumir mensagens mais rapidamente do que forem produzidas;
- as mensagens devem ser retiradas do buffer na mesma ordem que forem colocadas;
- restrição de exclusão mútua no acesso ao buffer circular.

Comunicação de Processos Sincronização Produtor-Consumidor

- **Exemplo do Problema do Produtor/Consumidor usando Sleep e Wakeup**

Para os casos extremos de ocupação do buffer (cheio/vazio), deverão funcionar as seguintes **regras de sincronização**:

- se o produtor tentar depositar uma mensagem no **buffer cheio**, ele será suspenso até que o consumidor retire pelo menos uma mensagem do buffer;
- se o consumidor tenta retirar uma mensagem do **buffer vazio**, ele será suspenso até que o produtor deposite pelo menos uma mensagem no buffer.

Comunicação de Processos – Primitivas *Sleep/Wakeup*

```
# define N 100
int count = 0;

void producer(void)
{
  int item;

  while (TRUE) {
   item = produce_item();
   if (count == N)
        sleep();
   insert_item(item);
   count = count + 1;
   if (count == 1)
        wakeup(consumer)
   }
}
```

```
void consumer(void)
{
  int item;

  while (TRUE) {
   if (count == 0)
        sleep();
   item = remove_item();
   count = count - 1;
   if (count == N - 1)
        wakeup(producer)
   consume_item(item);
   }
}
```


Comunicação de Processos – Primitivas *Sleep/Wakeup*

- **Exemplo do Problema do Produtor/Consumidor usando Sleep e Wakeup**

```
#define N 100
int contador = 0;

produtor()
{
  while(TRUE)
   {
     produz_item();
     if (contador==N)    Sleep();
     deposita_item();
     contador + = 1;
     if (contador==1)
              Wakeup(consumidor);
   }
}
```

```
consumidor()
{
  while(TRUE)
   {
     if (contador==0)    Sleep();
     retira_item();
     contador - = 1;
     if (contador==N-1)
              Wakeup(produtor);
   consome_item();
   }
}
```

interrupção

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Problemas desta solução: Acesso à variável `count` é irrestrita
  - O *buffer* está vazio e o consumidor acabou de checar a variável `count` com valor 0;
  - O escalonador (por meio de uma interrupção) decide que o processo produtor será executado; Então o processo produtor insere um item no *buffer* e incrementa a variável `count` com valor 1; Imaginando que o processo consumidor está dormindo, o processo produtor envia um sinal de *wakeup* para o consumidor;
  - No entanto, o processo consumidor não está dormindo, e não recebe o sinal de *wakeup*;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Assim que o processo consumidor é executado novamente, a variável `count` já tem o valor zero; Nesse instante, o consumidor é colocado para dormir, pois acha que não existem informações a serem lidas no *buffer*;
- Assim que o processo produtor acordar, ele insere outro item no *buffer e* volta a dormir. Ambos os processos dormem para sempre...

- Solução: *bit* de controle recebe um valor `true` quando um sinal é enviado para um processo que não está dormindo. No entanto, no caso de vários pares de processos, vários *bits* devem ser criados sobrecarregando o sistema!!!!

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- **Exemplo do Problema do Produtor/Consumidor usando Sleep e Wakeup**

Problema: pode ocorrer uma condição de corrida, se a variável contador for utilizada sem restrições.

Solução: Criar-se um **"bit de wakeup".** Quando um Wakeup é mandado à um processo já acordado, este bit é setado. Depois, quando o processo tenta ir dormir, se o bit de espera de Wakeup estiver ligado, este bit será desligado, e o processo será mantido acordado.

Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - **Semáforos**;
  - Monitores;
  - Passagem de Mensagem;

Comunicação de Processos – Semáforos

- Variável utilizada para controlar o acesso a recursos compartilhados
  - semáforo=0 → recurso está sendo utilizado
  - semáforo>0 → recurso livre
- Operações sobre semáforos
  - down → executada sempre que um processo deseja usar um recurso compartilhado
  - up → executada sempre que um processo liberar o recurso

Comunicação de Processos – Semáforos

- down(semáforo)
  - Verifica se o valor do semáforo é maior que 0
  - Se for, semáforo=semáforo – 1
  - Se não for, o processo que executou o down bloqueia
- up(semáforo)
  - semáforo=semáforo + 1
  - Se há processos bloqueados nesse semáforo, escolhe um deles e o desbloqueia
    - Nesse caso, o valor do semáforo permanece o mesmo

Operações sobre semáforos são atômicas.

Comunicação de Processos – Semáforos

- Semáforos usados para implementar exclusão mútua são chamados de ***mutex*** (*mutual exclusion semaphor*) ou binários, por apenas assumirem os valores 0 e 1
  - Recurso é a própria região crítica
- Vamos resolver o problema do produtor consumidor usando semáforos???
  - *mutex* → exclusão mútua
  - *full* e *empty* → sincronização

Comunicação de Processos – Semáforos

- Idealizados por E. W. Dijkstra (1965);
- Variável inteira que armazena o número de sinais *wakeups* enviados;
- Um semáforo pode ter valor 0 quando não há sinal armazenado ou um valor positivo referente ao número de sinais armazenados;
- Duas primitivas de chamadas de sistema: `down` *(sleep)* e `up` *(wake);*
- Originalmente P (`down`) e V (`up`) em holandês*;*

Comunicação de Processos – Semáforos

- ***Down***: verifica se o valor do semáforo é maior do que 0; se for, o semáforo é decrementado; Se o valor for 0, o processo é colocado para dormir sem completar sua operação de ***down***;
- Todas essas ações são chamadas de **ações atômicas**;
  - **Ações atômicas** garantem que quando uma operação no semáforo está sendo executada, nenhum processo pode acessar o semáforo até que a operação seja finalizada ou bloqueada;

Comunicação de Processos – Semáforos

- ***Up***: incrementa o valor do semáforo, fazendo com que algum processo que esteja dormindo possa terminar de executar sua operação ***down***;
- ***Semáforo Mutex***: garante a exclusão mútua, não permitindo que os processos acessem uma região crítica ao mesmo tempo
  - Também chamado de **semáforo binário**

Comunicação de Processos – Semáforos

SEMÁFOROS

Mecanismo criado para solucionar o problema de armazenar múltiplos WAKEUPS (E.W.Dijkstra)

O mecanismo envolve a utilização de uma variável compartilhada chamada **semáforo**, e de duas operações primitivas indivisíveis que atuam sobre ela.

A variável compartilhada pelos processos, poderá assumir valores inteiros não negativos e sua manipulação será restrita às operações **P** e **V** (ou **Down** e **Up**, ou **Wait** e **Signal**, respectivamente).

Comunicação de Processos – Semáforos

SEMÁFOROS

As operações que atuam em um semáforo denominado **s**, incluindo seus efeitos são definidos a seguir:

P(s): Espera até que s>0 e então decrementa s;

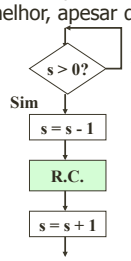
V(s): Incrementa s;

Comunicação de Processos – Semáforos

SEMÁFOROS

1ª. Implementação – Espera ocupada

Esta implementação é através da espera ocupada: não é a melhor, apesar de ser fiel à definição original.



P(s): Espera até que s>0 e então decrementa s;

V(s): Incrementa s;

Comunicação de Processos – Semáforos

SEMÁFOROS

2ª. Implementação – Associando uma fila Qi a cada semáforo si

Quando se utiliza este tipo de implementação, o que é muito comum, as primitivas P e V apresentam o seguinte significado:

P(si): se si>0 e então decrementa si (e o processo continua)
senão bloqueia o processo, colocando-o na fila Qi;

V(si): se a fila Qi está vazia então incrementa si
senão acorda processo da fila Qi;

Comunicação de Processos – Semáforos

SEMÁFOROS

O semáforo é um mecanismo bastante geral para resolver problemas de sincronismo e exclusão mútua.

Tipos de Semáforos

Semáforo geral: se o semáforo puder tomar qualquer valor inteiro não negativo;

Semáforo binário (booleano): só pode tomar os valores 0 e 1.

Comunicação de Processos – Semáforos

Problema da Exclusão Mútua com Semáforos

```
Program exclusao_mutua;
Var Mutex: semaphore;
Begin                   /* inicio do programa principal */
      Mutex:=1;  /* condição inicial */
      Cobegin    /* inicio dos processos concorrentes */
                  Begin /* Processo 1 */
                              Repeat
                                    ...
                                    P(Mutex);
                                          Seção_crítica_1;
                                    V(Mutex);
                                    ...
                              until false;
                  End;

                  /* outros processos*/
      Coend
End.
```

79

Comunicação de Processos – Semáforos

SEMÁFOROS e a Sincronização Baseada em Condições

Sincronização de processos baseada em condições → **Cada condição é representada por um semáforo**

Se um processo precisa verificar se uma condição é verdadeira antes de prosseguir

verifica semáforo P(s)

- **Condição falsa = (s=0) bloqueia processo**
- **Condição verdadeira = (s>0) processo continua**

80

Comunicação de Processos – Semáforos

SEMÁFOROS e a Sincronização Baseada em Condições

Se um processo torna uma condição verdadeira, deve sinalizar isto através do semáforo, usando V(s)

V(s)

- **Se fila Q(s) não estiver vazia, desbloqueia um processo**
- **Se fila de bloqueados estiver vazia, incrementa s**

81

Comunicação de Processos – Semáforos

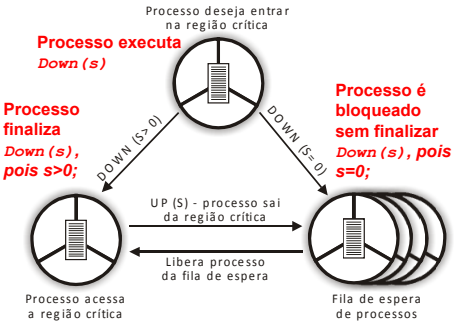
Problema do Produtor/ Consumidor usando Semáforos

```
Program Produtor_consumidor;
Const max = ...;
Type msg=...;
Var mensagem: msg;
Var buffer:...;
Var p, c: 0..max –1; /* ponteiros do buffer*/
Var cheio, vazio, mutex: semaphore;

Procedure Depositar(m:msg)
Begin ... End;

Procedure Retirar (var m:msg)
Begin ... End;
/* inicio do programa principal */
Begin
            cheio:=0;
            vazio:=max;
            mutex:=1;
            p:=0;
            c:=0;
```

```
Cobegin  /* inicio dos processos concorrentes */
  Begin  /* processo Produtor */
    Repeat
            ...
            produção da mensagem;
            P(vazio);  /* sincronização */
            P(mutex);  /* exclusão mútua */
              Depositar(mensagem);
            V(mutex); /*liberação da exclusão mútua */
            V(cheio); /*sincronização para o consumidor */
    Until false;
  End;

  Begin  /* processo Consumidor */
    Repeat
            P(cheio);  /* sincronização */
            P(mutex);  /* exclusão mútua */
              Retirar(mensagem);
            V(mutex); /*liberação da exclusão mútua */
            V(vazio); /*sincronização para o produtor */
            consumo da mensagem;
            ...
    Until false;
  End
Coend
End.
```

82

Comunicação de Processos – Semáforo Binário



83

Comunicação de Processos – Semáforos

- Problema produtor/consumidor: resolve o problema de perda de sinais enviados;
- Solução utiliza três semáforos:
  - `Full`: conta o número de *slots* no *buffer* que estão ocupados; iniciado com 0; resolve sincronização;
  - `Empty`: conta o número de *slots* no *buffer* que estão vazios; iniciado com o número total de *slots* no *buffer*; resolve sincronização;
  - `Mutex`: garante que os processos produtor e consumidor não acessem o *buffer* ao mesmo tempo; iniciado com 1; também chamado de **semáforo binário**; Permite a exclusão mútua;

84

Comunicação de Processos – Semáforos

```
# include "prototypes.h"
# define N 100

typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer (void){
  int item;
  while (TRUE){
    produce_item(&item);
    down(&empty);
    down(&mutex);
    enter_item(item);
    up(&mutex);
    up(&full);
  }
}
```

```
void consumer (void){
  int item;
  while (TRUE){
    down(&full);
    down(&mutex);
    remove_item(item);
    up(&mutex);
    up(&empty);
    consume_item(item);
  }
}
```


Comunicação de Processos – Semáforos

- Problema: erro de programação pode gerar um *deadlock*;
  - Suponha que o código seja trocado no processo produtor;

```
..                      ..
down(&empty);           down(&mutex);
down(&mutex);           down(&empty);
..                      ..
```

  - Se o *buffer* estiver cheio, o produtor será bloqueado com `mutex = 0;` Assim, a próxima vez que o consumidor tentar acessar o *buffer*, ele tenta executar um `down` sobre o `mutex`, ficando também bloqueado.

Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - **Monitores**;
  - Passagem de Mensagem;

Comunicação de Processos – Monitores

- Idealizado por Hoare (1974) e Brinch Hansen (1975)
- **Monitor**: primitiva (unidade básica de sincronização) de alto nível para sincronizar processos:
  - Conjunto de procedimentos, variáveis e estruturas de dados agrupados em um único módulo ou pacote;
- Somente um processo pode estar ativo dentro do monitor em um mesmo instante; outros processos ficam bloqueados até que possam estar ativos no monitor;

Comunicação de Processos – Monitores

```
monitor example
  int i;
  condition c;

  procedure A();
  .
  end;
  procedure B();
  .
  end;
end monitor;
```

Estrutura básica de um Monitor

Dependem da linguagem de programação → Compilador é que garante a exclusão mútua.
- JAVA

Todos os recursos compartilhados entre processos devem estar implementados **dentro** do **Monitor**;

Comunicação de Processos – Monitores

- Execução:
  - Chamada a uma rotina do monitor;
  - Instruções iniciais → teste para detectar se um outro processo está ativo dentro do monitor;
  - Se positivo, o processo novo ficará bloqueado até que o outro processo deixe o monitor;
  - Caso contrário, o processo novo executa as rotinas no **monitor**;

Comunicação de Processos – Monitores

- *Condition Variables* (`condition`): variáveis que indicam uma condição; e
- Operações Básicas: `WAIT e SIGNAL`
  `wait (condition)` → bloqueia o processo;
  `signal (condition)` → "acorda" o processo que executou um `wait` na variável `condition` e foi bloqueado;

Comunicação de Processos – Monitores

- Variáveis condicionais não são contadores, portanto, não acumulam sinais;
- Se um sinal é enviado sem ninguém (processo) estar esperando, o sinal é perdido;
- Assim, um comando `WAIT` deve vir antes de um comando `SIGNAL`.

Comunicação de Processos – Monitores

- Como evitar dois processos ativos no monitor ao mesmo tempo?
- (1) Hoare → colocar o processo mais recente para rodar, suspendendo o outro!!! (*sinalizar e esperar*)
- (2) B. Hansen → um processo que executa um `SIGNAL` deve deixar o monitor imediatamente;
  - O comando `SIGNAL` deve ser o último de um procedimento do monitor;

A condição (2) é mais simples e mais fácil de se implementar.

Comunicação de Processos – Monitores

```
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;
```


Comunicação de Processos – Monitores

- A exclusão mútua automática dos procedimentos do monitor garante que, por exemplo, se o produtor dentro de um procedimento do monitor descobrir que o buffer está cheio, esse produtor será capaz de terminar a operação de `WAIT` sem se preocupar, pois o consumidor não estará ativo dentro do monitor até que `WAIT` tenha terminado e o produtor tenha sido marcado como não mais executável;

Comunicação de Processos – Monitores

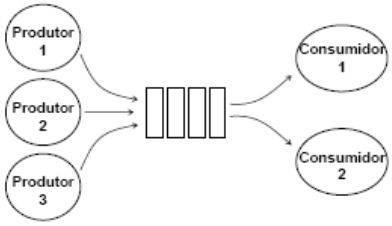
- Limitações de semáforos e monitores:
  - Ambos são boas soluções somente para CPUs com memória compartilhada. Não são boas soluções para sistema distribuídos;
  - Nenhuma das soluções provê troca de informações entre processo que estão em diferentes máquinas;
  - Monitores dependem de uma linguagem de programação – poucas linguagens suportam Monitores;

Problemas Clássicos De Comunicação entre processos

Produtor - Consumidor

- Um sistema é composto por entidades produtoras e entidades consumidoras.
- Entidades produtoras
  - Responsáveis pela produção de itens que são armazenados em um buffer (ou em uma fila)
  - Itens produzidos podem ser consumidos por qualquer consumidor
- Entidades consumidoras
  - Consomem os itens armazenados no buffer (ou na fila)
  - Itens consumidos podem ser de qualquer produtor

Produtor - Consumidor



Leitores - Escritores

- Um sistema com uma base de dados é acessado simultaneamente por diversas entidades. Estas entidades realizam dois tipos de operações:
  - Leitura
  - Escrita
- Neste sistema é aceitável a existência de diversas entidades lendo a base de dados.
- Porém, se um processo necessita escrever na base, nenhuma outra entidade pode estar realizando acesso à base.

O Problema dos Leitores e Escritores

```
typedef int semaphore;              /* use sua imaginação */
semaphore mutex = 1;                /* controla o acesso a 'rc' */
semaphore db = 1;                   /* controla o acesso a base de dados */
int rc = 0;                         /* número de processos lendo ou querendo ler */

void reader(void)
{
     while (TRUE) {                 /* repete para sempre */
          down(&mutex);             /* obtém acesso exclusivo a 'rc' */
          rc = rc + 1;              /* um leitor a mais agora */
          if (rc == 1) down(&db);   /* se este for o primeiro leitor ... */
          up(&mutex);               /* libera o acesso exclusivo a 'rc' */
          read_data_base( );        /* acesso aos dados */
          down(&mutex);             /* obtém acesso exclusivo a 'rc' */
          rc = rc - 1;              /* um leitor a menos agora */
          if (rc == 0) up(&db);     /* se este for o último leitor ... */
          up(&mutex);               /* libera o acesso exclusivo a 'rc' */
          use_data_read( );         /* região não crítica */
     }
}

void writer(void)
{
     while (TRUE) {                 /* repete para sempre */
          think_up_data( );         /* região não crítica */
          down(&db);                /* obtém acesso exclusivo */
          write_data_base( );       /* atualiza os dados */
          up(&db);                  /* libera o acesso exclusivo */
     }
}
```

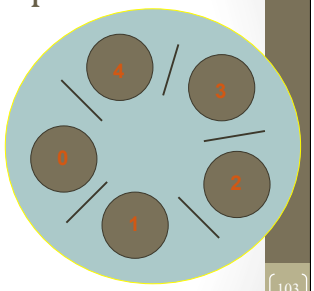
Uma solução para o problema dos leitores e escritores

Problemas clássicos de comunicação entre processos

- Cinco filósofos estão sentados ao redor de uma mesa circular para o jantar.
  - Cada filósofo possui um prato para comer espaguete.
  - Como o espaguete é muito escorregadio, é necessário a utilização de dois garfos.
  - Entre cada par de pratos existe um garfo.

Problemas clássicos de comunicação entre processos

- Problema do Jantar dos Filósofos
  - Cinco filósofos desejam comer espaguete; No entanto, para poder comer, cada filósofo precisa utilizar dois garfo e não apenas um. Portanto, os filósofos precisam compartilhar o uso do garfo de forma sincronizada.
  - Os filósofos comem e pensam;

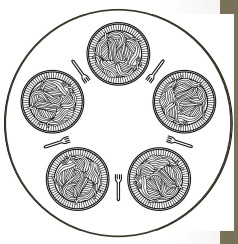


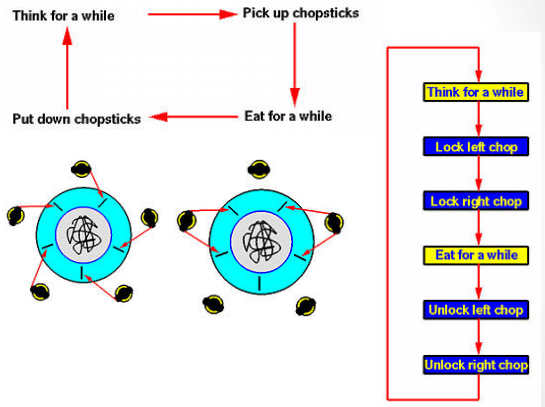
Problemas clássicos de comunicação entre processos

- **A vida do filósofo consiste na alternância de períodos de alimentação e reflexão.**
  - **Quando um filósofo fica com fome, ele tenta pegar os garfos a sua volta (garfos a sua esquerda e direira), em qualquer ordem, um de cada vez.**
  - **Se o filósofo conseguir pegar os dois garfos ele inicia seu período de alimentação. Após algum tempo ele devolve os garfos a sua posição original e retorna ao período de reflexão**

Problemas clássicos de comunicação entre processos

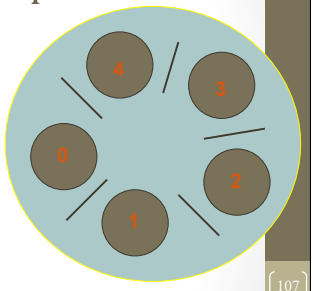
- Filósofos comem/pensam
- Cada um precisa de 2 garfos para comer
- Pega um garfo por vez
- Como prevenir deadlock





Problemas clássicos de comunicação entre processos

- Problemas que devem ser evitados:
  - *Deadlock* – todos os filósofos pegam **um garfo** ao mesmo tempo;
  - *Starvation* – os filósofos **fiquem indefinidamente pegando garfos simultaneamente**;



Solução 1 para Filósofos (1/2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

Solução 1 para Filósofos (2/2)

- Problemas da solução 1:
  - Execução do `take_fork(i)` → Se todos os filósofos pegarem o garfo da esquerda, nenhum pega o da direita → *Deadlock*;
- Se modificar a solução (mudança 1):
  - Verificar antes se o garfo da direita está disponível. Se não está, devolve o da esquerda e começa novamente → *Starvation (Inanição);*
  - Tempo fixo ou tempo aleatório (rede Ethernet);
    - Serve para sistemas não-críticos;

Solução 1 para Filósofos (2/2)

- Se modificar a solução (mudança 2):

```
#define N 5                                     /* number of philosophers */
 semaphore mutex = 1;
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        down(&mutex);
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
        up(&mutex);
    }
}
```

Somente um filósofo come!

Solução 2 para Filósofos usando Semáforos (1/3)

- Não apresenta:
  - *Deadlocks*;
  - *Starvation*;
- Permite o máximo de "paralelismo";

Solução 2 para Filósofos usando Semáforos (2/3)

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N–1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                          /* repeat forever */
        think( );                           /* philosopher is thinking */
        take_forks(i);                      /* acquire two forks or block */
        eat( );                             /* yum-yum, spaghetti */
        put_forks(i);                       /* put both forks back on table */
    }
}
```

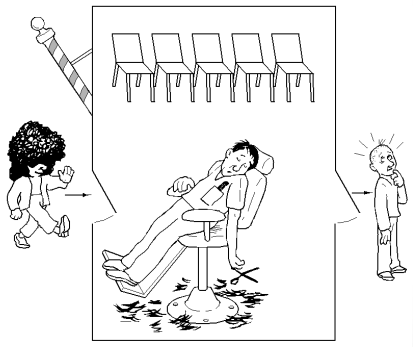

Solução 2 para Filósofos usando Semáforos (3/3)

```
void take_forks(int i)                      /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                           /* enter critical region */
    state[i] = HUNGRY;                      /* record fact that philosopher i is hungry */
    test(i);                                /* try to acquire 2 forks */
    up(&mutex);                             /* exit critical region */
    down(&s[i]);                            /* block if forks were not acquired */
}

void put_forks(i)                           /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                           /* enter critical region */
    state[i] = THINKING;                    /* philosopher has finished eating */
    test(LEFT);                             /* see if left neighbor can now eat */
    test(RIGHT);                            /* see if right neighbor can now eat */
    up(&mutex);                             /* exit critical region */
}

void test(i)                                /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```


Barbeiro Sonolento

- **Uma barbearia possui:**
  - **1 barbeiro**
  - **1 cadeira de barbeiro**
  - **N cadeira para espera de clientes**
- **Se, em um determinado momento, não houverem clientes para serem atendidos, o barbeiro dorme.**
  - **Quando um cliente chega, ele acorda e atende o cliente.**
  - **Quando um cliente chega e o barbeiro estiver atendendo um cliente, ele aguarda sua vez sentado na cadeira de espera.**
  - **Quando um cliente chega e não existem cadeiras de espera disponíveis, o cliente vai embora.**

O Problema do Barbeiro Sonolento (1)



O Problema do Barbeiro Sonolento (2)

```
#define CHAIRS 5                    /* número de cadeiras para os clientes à espera */
typedef int semaphore;              /* use sua imaginação */
semaphore customers = 0;            /* número de clientes à espera de atendimento*/
semaphore barbers = 0;              /* número de barbeiros à espera de clientes */
semaphore mutex = 1;                /* para exclusão mútua */
int waiting = 0;                    /* clientes estão esperando (não estão cortando) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* vai dormir se o número de clientes for 0 */
          down(&mutex);             /* obtém acesso a 'waiting' */
          waiting = waiting – 1;    /* decresce de um o contador de clientes à espera */
          up(&barbers);             /* um barbeiro está agora pronto para cortar cabelo */
          up(&mutex);               /* libera 'waiting' */
          cut_hair();               /* corta o cabelo (fora da região crítica) */
     }
}

void customer(void)
{
     down(&mutex);                  /* entra na região crítica */
     if (waiting < CHAIRS) {        /* se não houver cadeiras livres, saia */
          waiting = waiting + 1;    /* incrementa o contador de clientes à espera */
          up(&customers);           /* acorda o barbeiro se necessário */
          up(&mutex);               /* libera o acesso a 'waiting' */
          down(&barbers);           /* vai dormir se o número de barbeiros livres for 0 */
          get_haircut( );           /* sentado e sendo servido */
     } else {
          up(&mutex);               /* a barbearia está cheia; não espere */
     }
}
```

Solução para o problema do barbeiro sonolento

Alguns links interessantes

- http://www.anylogic.pl/fileadmin/Modele/Traffic/filozof/Dining%20Philosophers%20-%20Hybrid%20Applet.html
- http://www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html
- http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/deadlock.html
- http://users.erols.com/ziring/diningAppletDemo.html

Problemas clássicos de comunicação entre processos

- Sugestão de Exercícios:
  - Entender a solução para o problema dos **Filósofos** utilizando semáforos:
    - Identificando a(s) **região(ões) crítica(s)**;
    - Descrevendo **exatamente** como a solução funciona;
  - Entender a solução para o problema dos Produtores/Consumidores utilizando monitor:
    - Identificando a(s) **região(ões) crítica(s)**;
    - Descrevendo **exatamente** como a solução funciona;

Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - Monitores;
  - **Passagem de Mensagem**;

Comunicação de Processos – Passagem de Mensagem

Mecanismos Mais Elaborados de Comunicação e Sincronização entre Processos

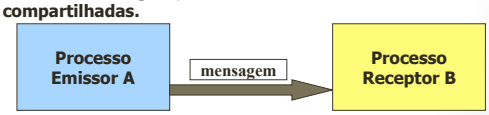
A troca de mensagens é um mecanismo de comunicação e sincronização que exige do S.O., tanto a sincronização quanto a comunicação entre os processos.

Os mecanismos já considerados exigem do S.O. somente a sincronização, deixando para o programador a comunicação de mensagens através da memória compartilhada.

Os mecanismos estudados até agora asseguram a exclusão mútua, mas não garantem um controle sobre as operações desempenhadas sobre o recurso.

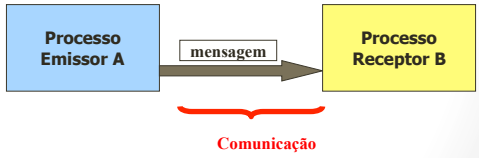
Comunicação de Processos – Passagem de Mensagem

- O uso de troca de mensagens para a manipulação de um recurso compartilhado **assegura a exclusão mútua**, e **impõe restrições nas operações** a serem desempenhadas sobre ele.
- Os mecanismos já considerados exigem do S.O. somente a sincronização, deixando para o programador a comunicação de mensagens através da memória compartilhada.
- **Esquema de troca de mensagens:** os processos enviam e recebem mensagens, em vez de ler e escrever em variáveis compartilhadas.



Comunicação de Processos – Passagem de Mensagem

- **A sincronização entre processos:** é garantida pela restrição de que uma mensagem só poderá ser recebida depois de ter sido enviada.
- A transferência de dados de um processo para outro, após ter sido realizada a sincronização, estabelece a comunicação.



Comunicação de Processos – Passagem de Mensagem

Primitivas de Troca de Mensagens

- De forma genérica, uma mensagem será enviada quando um processo executar o seguinte comando:

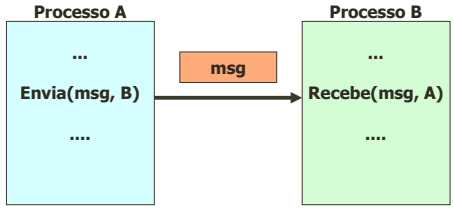
Envia (mensagem, processo_receptor)
ou
Send(message, receiver)

- Uma mensagem será recebida quando um processo executar o seguinte comando:

Recebe(mensagem, processo_emissor)
ou
Receive(message, sender)

Comunicação de Processos – Passagem de Mensagem

Primitivas de Troca de Mensagens



Comunicação de Processos – Passagem de Mensagem

Primitivas de Troca de Mensagens

- As primitivas podem ser de dois tipos:
  - **Bloqueantes:** quando o processo que a executar ficar bloqueado até que a operação seja bem sucedida (ou seja, quando ocorrer a entrega efetiva da mensagem ao processo destino, no caso da emissão, ou o recebimento da mensagem pelo processo destino, no caso de recepção).
  - **Não bloqueantes:** quando o processo que executar a primitiva, continuar sua execução normal, independentemente da entrega ou do recebimento efetivo da mensagem pelo processo destino.

Comunicação de Processos – Passagem de Mensagem

Exemplo de Comunicação usando Troca de Mensagens

```
Program emissor_receptor;
Type msg=...;
Var mensagem : msg;
Begin  /* inicio do programa principal */
  Cobegin /* inicio dos processos concorrentes */
    Begin /* processo emissor – E */
      repeat
        ...;
        produz uma mensagem;
        Send(mensagem, R);
      until  false
    End;

    Begin /* processo receptor – R */
      repeat
        Receive(mensagem, E);
        Consome a  mensagem;
        ...;
      until  false
    End
  Coend
End.
```

Comunicação de Processos – Passagem de Mensagem

Os sistemas de troca de mensagens possuem alguns problemas e estudos de projetos interessantes, principalmente quando os processos comunicantes estão em máquinas diferentes, conectadas por uma rede de comunicação. Os principais são:

- **Perda de mensagens**
- **Perda de reconhecimento**
- **Nomeação de Processos**
- **Autenticação**
- **Estudos de Projeto para quando emissor e receptor estiverem na mesma máquina**

127

Comunicação de Processos – Passagem de Mensagem

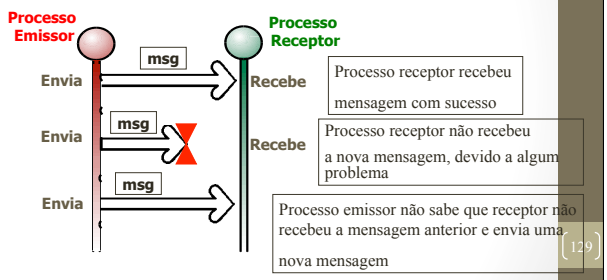
- **Perda de Mensagens**

uma solução é que o receptor, ao receber uma nova mensagem, envie uma mensagem especial de reconhecimento (ACK). Se o emissor não receber um ACK a tempo, deve retransmitir a mensagem.

128

Comunicação de Processos – Passagem de Mensagem

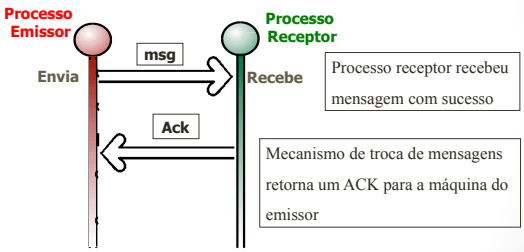
Perda de Mensagens: o problema



129

Comunicação de Processos – Passagem de Mensagem

Perda de Mensagens: uma solução



130

Comunicação de Processos – Passagem de Mensagem

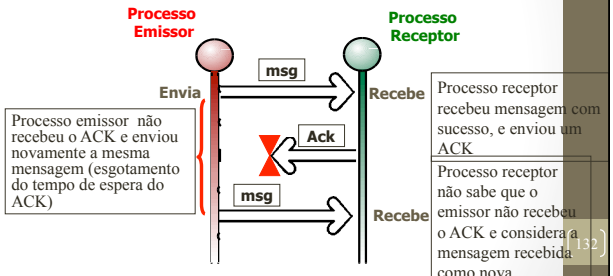
- **Perda de Reconhecimento (ACK)**

causa o problema de se receber mensagens idênticas.
Uma das soluções é a numeração de mensagens.

131

Comunicação de Processos – Passagem de Mensagem

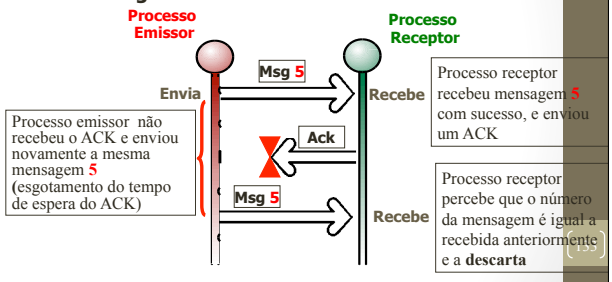
Perda de Reconhecimento: o problema



132

Comunicação de Processos – Passagem de Mensagem

Perda de Reconhecimento: uma solução é numerar as mensagens



Comunicação de Processos – Passagem de Mensagem

Nomeação de Processos

os processos devem ser nomeados de maneira única, para que o nome do processo especificado no Send ou Receive não seja ambíguo.

Ex: processo@máquina (normalmente existe uma autoridade central que nomeia as máquinas).

Quando o número de máquinas é muito grande: processo@máquina.domínio.

Comunicação de Processos – Passagem de Mensagem

Autenticação

permitir a comunicação e acessos apenas dos usuários autorizados. Uma solução é criptografar as mensagens com uma chave conhecida apenas por usuários autorizados.

Comunicação de Processos – Passagem de Mensagem

Estudo de projeto para quando emissor e receptor estão na mesma máquina

para aumento do desempenho, pensa-se em registradores especializados para a troca de mensagens.

Comunicação de Processos – Passagem de Mensagem

Combinação de Primitivas

existem quatro maneiras de se combinar as primitivas de troca de mensagens:

Envia Bloqueante – Recebe Bloqueante ⇨ síncrono

Envia Bloqueante – Recebe Não Bloqueante

Envia Não Bloqueante – Recebe Bloqueante

} **Semi síncrono**

Envia Não Bloqueante – Recebe Não Bloqueante ⇨ **Assíncrono**

Comunicação de Processos – Passagem de Mensagem

- Podem ser implementadas como procedimentos:
  - `send (destination,&message);`
  - `receive (source,&message);`
- O procedimento `send` envia para um determinado destino uma mensagem, enquanto que o procedimento `receive` recebe essa mensagem em uma determinada fonte; Se nenhuma mensagem está disponível, o procedimento `receive` é bloqueado até que uma mensagem chegue.

Comunicação de Processos – Passagem de Mensagem

- Problemas desta solução:
  - Mensagens são enviadas para/por máquinas conectadas em rede; assim mensagens podem se perder ao longo da transmissão;
  - Mensagem especial chamada ***acknowledgement*** → o procedimento `receive` envia um ***acknowledgement*** para o procedimento `send`. Se esse ***acknowledgement*** não chega no procedimento `send`, esse procedimento retransmite a mensagem já enviada;

Comunicação de Processos – Passagem de Mensagem

- Problemas:
  - A mensagem é recebida corretamente, mas o ***acknowledgement*** se perde.
  - Então o `receive` deve ter uma maneira de saber se uma mensagem recebida é uma retransmissão → cada mensagem enviada pelo `send` possui uma identificação – sequência de números; Assim, ao receber uma nova mensagem, o `receive` verifica essa identificação, se ela for semelhante a de alguma mensagem já recebida, o `receive` descarta a mensagem!

Comunicação de Processos – Passagem de Mensagem

- Problemas:
  - Desempenho: copiar mensagens de um processo para o outro é mais lento do que operações com semáforos e monitores;
  - Autenticação → Segurança;

Comunicação de Processos – Passagem de Mensagem

```
#define N 100                                   /* number of slots in the buffer */

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

     while (TRUE) {
          item = produce_item();                /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

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

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```


Comunicação de Processos – Passagem de Mensagem

Mecanismos de Comunicação Síncronos

Existem três mecanismos de comunicação síncronos mais importantes:

- **Rendez-vous**
- **Rendez-vous Estendido**
- **Chamada Remota de Procedimento**

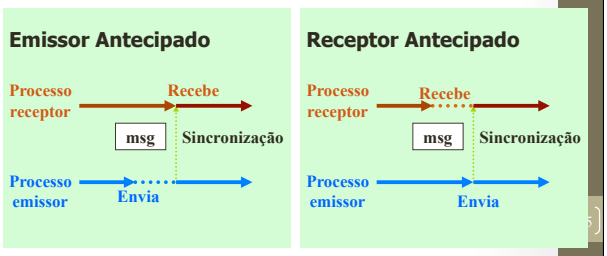
Comunicação de Processos – Passagem de Mensagem

Rendez-vous

é obtido por meio de primitivas Envia e Recebe bloqueantes colocadas em processos distintos; a execução destas primitivas em tempos diferentes, fará com que o processo que executar a primitiva antes do outro fique bloqueado até que ocorra a sincronização entre os dois processos, e a consecutiva transferência da mensagem; em seguida, ambos os processos continuarão seu andamento em paralelo.
Ex.: linguagem CSP.

Comunicação de Processos – Passagem de Mensagem

Rendez-vous



Comunicação de Processos – Passagem de Mensagem

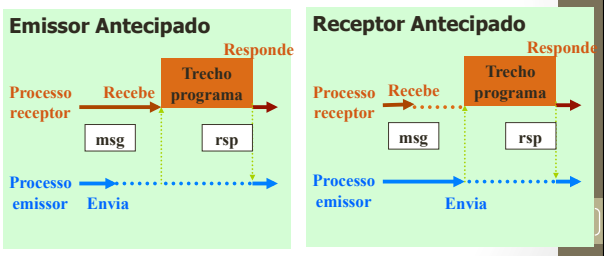
- **Rendez-vous Estendido**

caracteriza-se por apresentar uma estrutura de comunicação onde um processo consegue comandar a execução de um trecho de programa previamente estabelecido, pertencente a outro processo, envolvendo sincronização e, eventualmente, troca de mensagem.
Ex: linguagem ADA.

146

Comunicação de Processos – Passagem de Mensagem

Rendez-vous Estendido



Comunicação de Processos – Passagem de Mensagem

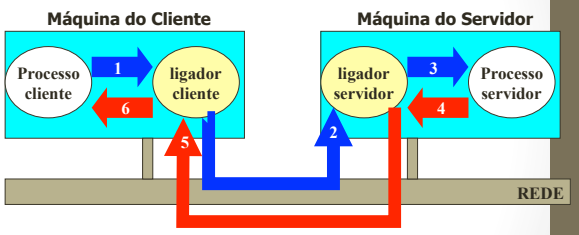
- **Chamada Remota de Procedimento (RPC - Remote Procedure Call)**

apresenta uma estrutura de comunicação na qual um processo pode comandar a execução de um procedimento situado em outro processador. O processo chamador deverá ficar bloqueado até que o procedimento chamado termine. Tanto a chamada quanto o retorno podem envolver troca de mensagem, conduzindo parâmetros.
Ex: linguagem DP.

148

Comunicação de Processos – Passagem de Mensagem

Chamada Remota de Procedimento



(1) e (3) são chamadas de procedimento comuns
(2) e (5) são mensagens
(4) e (6) são retornos de procedimento comuns

149

Comunicação de Processos – Passagem de Mensagem

Vantagem da Chamada Remota de Procedimento

- Cliente e servidor não precisam saber que as mensagens são utilizadas.

Eles as veem como chamadas de procedimento locais.

150

Comunicação de Processos – Passagem de Mensagem

Problemas da Chamada Remota de Procedimento

- **Dificuldade da passagem de parâmetros por referência:** **por exemplo, se as máquina servidora e cliente possuem diferentes representações de informação (necessidade de conversão e desconversão).**
- **Diferenças de arquitetura:** **por exemplo, as máquinas podem diferir no armazenamento de palavras.**
- **Falhas semânticas:** **por exemplo, o servidor pára de funcionar quando executava uma RPC. O que dizer ao cliente? Se disser que houve falha e o servidor terminou a chamada logo antes de falhar, o cliente pode pensar que falhou antes de executar a chamada. Ele pode tentar novamente, o que pode não ser desejável. Principais abordagens: "no mínimo uma vez", "exatamente uma vez" e "talvez uma vez".**

Comunicação de Processos Outros mecanismos

- RPC – *Remote Procedure Call*
  - Rotinas que permitem comunicação de processos em diferentes máquinas;
  - Chamadas remotas;
- MPI – *Message-passing Interface*;
  - Sistemas paralelos;
- RMI Java – *Remote Method Invocation*
  - Permite que um objeto ativo em uma máquina virtual Java possa interagir com objetos de outras máquinas virtuais Java, independentemente da localização dessas máquinas virtuais;

Comunicação de Processos Outros mecanismos

- *Pipe*:
  - Permite a criação de filas de processos;
  - `ps -ef | grep alunos`;
  - Saída de um processo é a entrada de outro;
  - Existe enquanto o processo existir;
- *Named pipe*:
  - Extensão de pipe;
  - Continua existindo mesmo depois que o processo terminar;
  - Criado com chamadas de sistemas;
- *Socket*:
  - Par endereço IP e porta utilizado para comunicação entre processos em máquinas diferentes;
  - Host X (192.168.1.1:1065) Server Y (192.168.1.2:80);

Comunicação de Processos – Passagem de Mensagem

Outros Mecanismos de Comunicação Baseados na Troca de mensagens

Existem diversos mecanismos atuais baseados na troca de mensagens, além dos discutidos. Os mais representativos são:

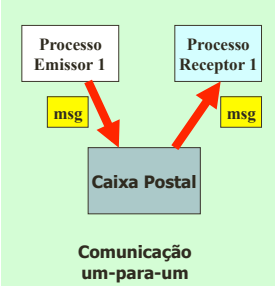
- **Caixas Postais**
- **Portos**

Comunicação de Processos – Passagem de Mensagem

- **Caixas Postais (*Mailboxes*)**
- **São filas de mensagens não associadas, a princípio, com nenhum processo.**
- **Os processos podem enviar e receber mensagens das caixas postais, tornando o mecanismo adequado para comunicar diversos remetentes a diversos receptores.**
- **Uma restrição ao uso de caixas postais é a sua necessidade de envio de duas mensagens para comunicar o remetente com o receptor: uma do remetente à caixa postal, e outra da caixa postal para o receptor.**

Comunicação de Processos – Passagem de Mensagem

Caixas Postais



Comunicação de Processos – Passagem de Mensagem

- **Portos (Ports)**
- **consiste num elemento do sistema que permite a comunicação entre conjuntos de processos através do mascaramento de um de seus identificadores. Cada porto é como uma caixa postal, porém com um dono, que será o processo que o criar.**

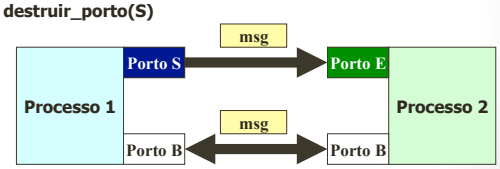
157

Comunicação de Processos – Passagem de Mensagem

Portos

Cria_porto(S, saída, msg)
conecta_porto(S, E)
envia_porto(S, msg)
desconecta_porto(S, E)
destruir_porto(S)

Cria_porto(E, entrada, msg)
recebe_porto(E, msg)
destruir_porto(E)



158

Comunicação de Processos – Passagem de Mensagem

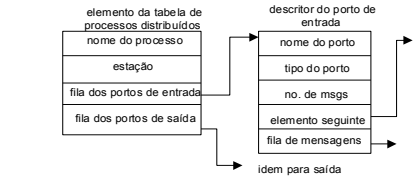
Outras Características dos Portos

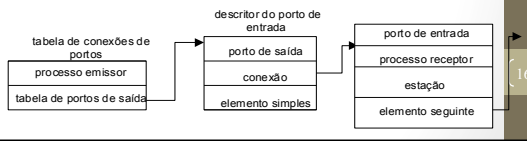
- **A criação e a interligação de portos e caixas postais pode ser feita de maneira dinâmica. A necessidade de enfileiramento das mensagens enviadas, torna necessário o uso de "buffers", para o armazenamento intermediário.**
- **A comunicação entre processos locais ou remotos, em um sistema estruturado com portos, será feita pela execução de primitivas síncronas (ou assíncronas) do tipo envia e recebe.**

159

Comunicação de Processos – Passagem de Mensagem

Uma Idéia para a Implementação de Portos



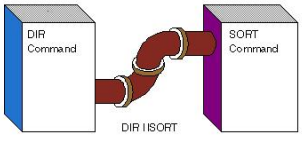


160

Comunicação de Processos – Passagem de Mensagem

Comunicação de Processos no UNIX: PIPES

- **Pipes: – usados no sistema operacional UNIX para permitir a comunicação entre processos. Um pipe é um modo de conectar a saída de um processo com a entrada de outro processo, sem o uso de arquivos temporários. Um pipeline é uma conexão de dois ou mais programas ou processos através de pipes.**



161

Comunicação de Processos – Passagem de Mensagem

PIPES

Exemplos:

sem uso de pipes (usando arquivos temporários)

```
$ ls > temp
$ sort < temp
```

com uso de pipes

```
$ ls | sort
$ wc poem
 8  46  263
$ wc ch* > wc.out &   /* uso do background &*/
   6944  /* id. do processo */
$ pr ch*| lpr &
```

162