Sistemas Operacionais

Concorrência entre processos

Prof. Robson Siscoutto

e-mail: robson@unoeste.br

Sistemas Operacionais

Concorrência entre processos

- Introdução;
- Modelos do processo;
- Estados e mudanças do processo;
- Tipos de processos;
- Comunicação entre processos;
- Escalonamento de Processos;
- **Concorrência entre Processos**
  - **Problemas de sincronização;**
  - **Soluções de hardware e software;**
- Deadlock
- Threads

Concorrência entre Processos

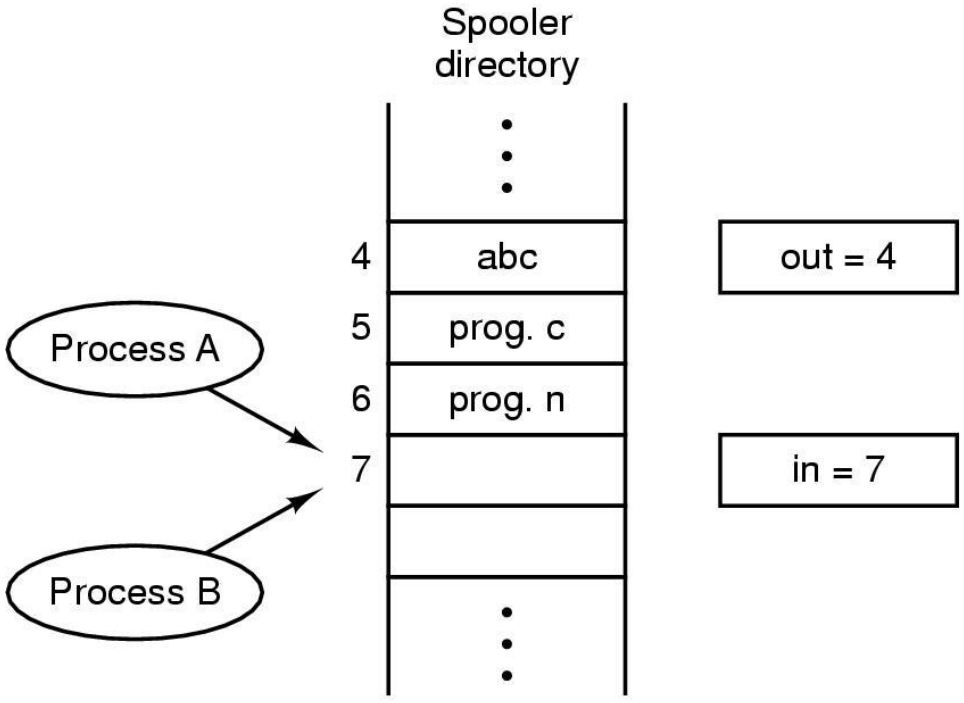
Comunicação Inter-processos

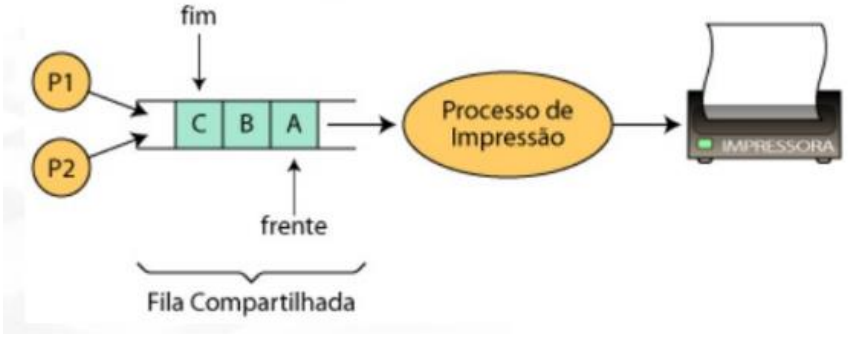
Condições de Corrida

- Situação onde dois ou mais processos estão acessando dados compartilhados e o resultado final do processamento depende de quem roda quando;

- Exemplo: Spool de impressão
  - Sempre que um processo deseja imprimir um arquivo ele entra com o nome do arquivo em um diretório de Spool;
  - Utiliza-se duas variáveis compartilhadas:
    - **in** – aponta para próxima entrada livre;
    - **out** – aponta p/ o próximo arquivo a ser impresso;

Spool Impressão Exemplo (1)





- Diretório de *spooler* com *n* entradas, cada uma capaz de armazenar um nome de arquivo.
- Servidor de impressão verifica se existem arquivos a serem impressos. Caso afirmativo, ele os imprime e remove os nomes do diretório.
- Variáveis compartilhadas: *out*, que aponta para o próximo arquivo a ser impresso; e *in*, que aponta para a próxima entrada livre no diretório.

Concorrência entre Processos

Comunicação Inter-processos

Condições de Corrida

- Problema:
  - Processo A lê de in == 7 e ocorre a mudança de contexto;
    - A armazenou 7 em sua variável interna;
  - Processo B lê in == 7 e coloca seu arquivo na posição 7;
    - Incrementa in = 8;
  - Processo A volta a executar e verifica sua variável interna = 7;
  - Processo A coloca seu arquivo na posição 7 e faz in = 8;
    - Sobrescreve o arquivo de B que esta em 7.
  - Para o gerenciador do spool tudo está correto;

Exemplo 2

```
Procedure echo();
var   out, in: character;
begin
      input (in, keyboard);
      out := in;
      output (out, display)
end.
```

- P1 invoca *echo()* e é interrompido imediatamente após a conclusão da função *input()*. Suponha que *x* tenha sido o caractere digitado, que agora está armazenado na variável *in*.
- P2 é despachado e também invoca *echo()*. Suponha que *y* seja digitado (*in* recebe *y*), sendo então exibido no dispositivo de saída.
- P1 retoma a posse do processador. O caractere exibido não é o que foi digitado (*x*), pois ele foi sobreposto por *y* na execução do processo P2. Conclusão: o caractere *y* é exibido duas vezes.
- Essência do problema: o compartilhamento da variável global *in*.

Concorrência entre Processos

Concorrência entre processos

Regiões Críticas

- Como **evitar** Condições de Corrida?
  - Proibir que mais de um processo acesse recursos compartilhado ao mesmo tempo.

  - Devemos implementar **Exclusão Mútua** de execução;
    - Deve afetar apenas os processo concorrentes que estejam acessando o recurso compartilhado;

  - Conceito de **Região Critica**:
    - É a parte do programa que faz o acesso ao recurso compartilhado;
    - Sempre que um processo desejar entrar ou sair de sua região critica ele deve verificar se ele pode ou não entrar;

Concorrência entre Processos

Comunicação Inter-processos - Concorrência

- Dificuldades:
  - Compartilhamento de recursos globais.
  - Gerência de alocação de recursos.
  - Localização de erros de programação (depuração de programas).
- Ação necessária:
  - Proteger os dados compartilhados (variáveis, arquivos e outros recursos globais).
  - Promover o acesso ordenado (controle de acesso) aos recursos compartilhados $\Rightarrow$ *sincronização de processos*.

Concorrência entre Processos
Comunicação Inter-processos

Regiões Críticas

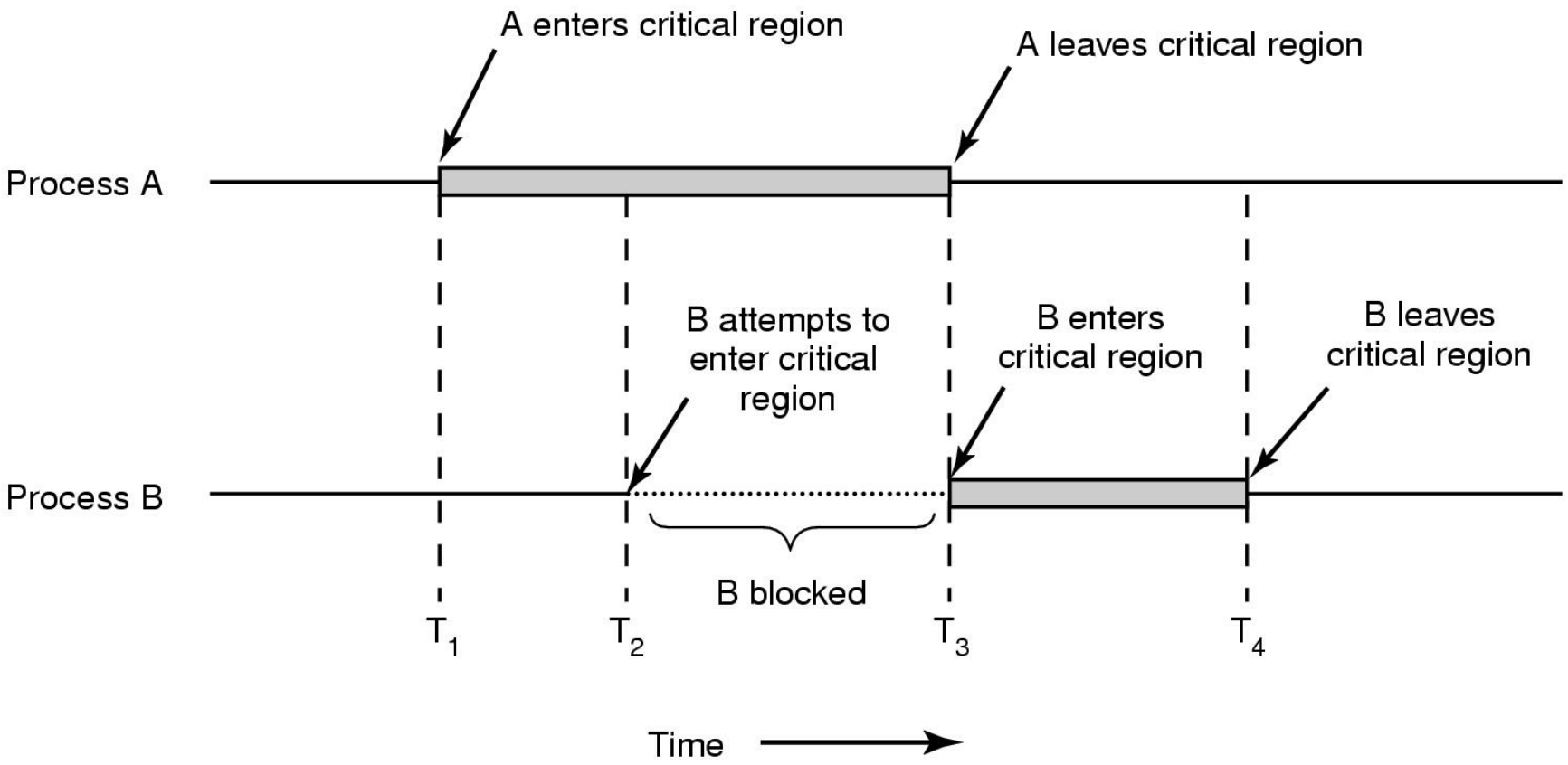
- O esquema descrito anteriormente apesar de evitar condições de corrida, mas só ele não é suficiente para permitir que processos paralelos cooperem corretamente;

- Quatro condições devem ser garantidas:
  1. Dois ou mais processos não podem estar simultaneamente na R.C;
  2. Consideração sobre velocidade do processo deve ser ignoradas;
  3. Processos fora da R.C não podem bloquear outro processo;
  4. Processo não pode ficar esperando indefinidamente p/ entrar na sua Região critica.

Concorrência entre Processos

Comunicação Inter-processos

Exclusão Mútua usando Regiões Críticas



Tipos de Soluções

- **Soluções de Hardware**
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- **Soluções de software com *busy wait***
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Decker
  - Algoritmo de Peterson
- **Soluções de software com bloqueio**
  - Sleep / Wakeup, Semáforos, Monitores

Concorrência entre Processos

Comunicação Inter-processos

Exclusão Mútua

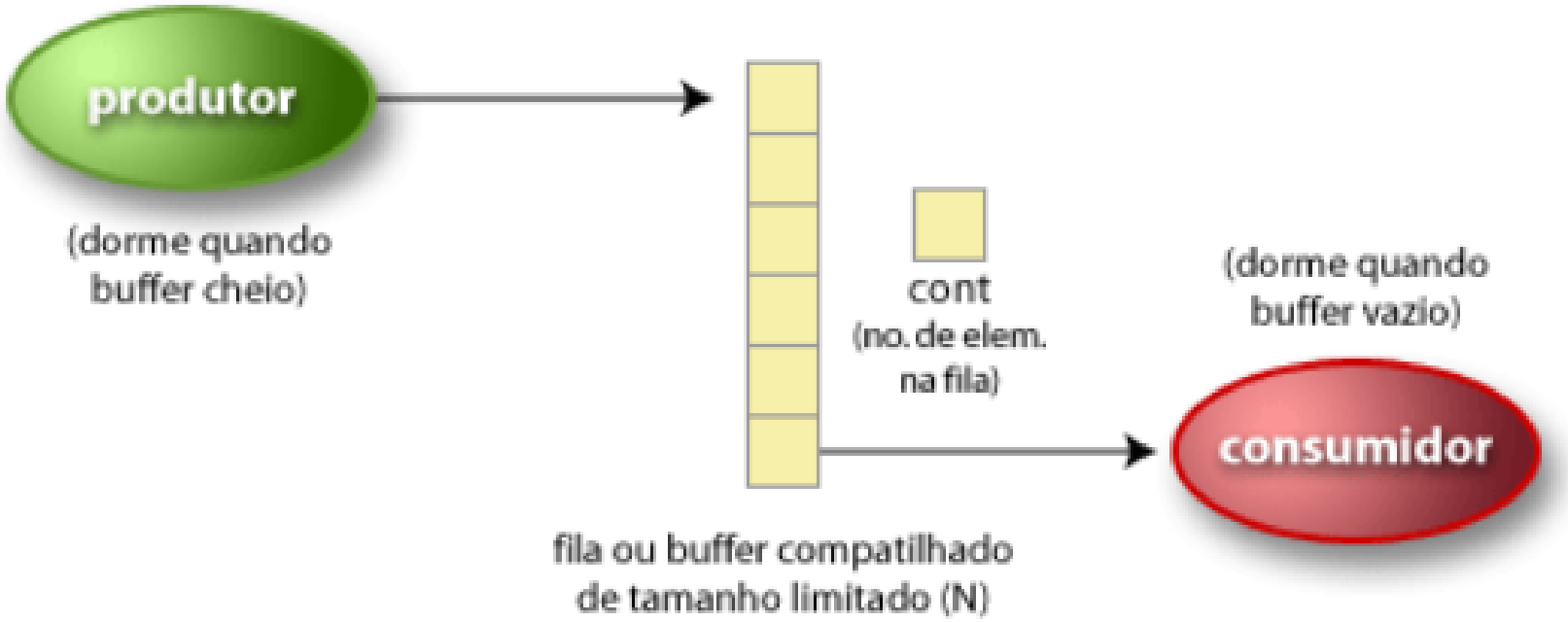
Soluções de Hardware

- Inibição de interrupções
- Instrução TSL (apresenta *busy wait*)

Concorrência entre Processos
Comunicação Inter-processos

- **Inibição das Interrupções**:
  - Cada processo inibe as interrupções logo após seu ingresso em uma região crítica e Habilitando-as outra vez imediatamente antes de deixá-la.
    - DI = disable interrupt EI = enable interrupt
  - Sem interrupção, o processador não consegue realizar a mudança de contexto, logo o processo executando não é trocado;
  - Desvantagens:
    - Se o processo não habilitar as interrupções, o que acontece?
    - Em sistemas multiprocessados, a inibição ocorre somente no processador onde o processo está executando;
    - Conflito com o Kernel, pois o kernel pode estar realizando operações críticas e neste momento o processo inibi as interrupções.

Exemplo - Problema do produtor-consumidor



Exemplo - Problema do produtor-consumidor

- variável *N* indica quantos itens ainda podem ser colocados no *buffer*.

```
Produtor                    Consumidor
DI                          DI
LDA N                       LDA N       direct address   ac <- Mem[N]
DCR AC                      INC  AC
STA N                       STA N        Mem[N] <- ac
EI                          EI
```

Problemas da Solução DI/EI

- É desaconselhável dar aos processos de usuário o poder de desabilitar interrupções.
- Não funciona com vários processadores.
- Inibir interrupções por um longo período de tempo pode ter consequências danosas. Por exemplo, perde-se a sincronização com os dispositivos periféricos.
  - OBS: inibir interrupções pelo tempo de algumas poucas instruções pode ser conveniente para o *kernel* (p.ex., para atualizar uma estrutura de controle).

Concorrência entre Processos

Comunicação Inter-processos

- **Instrução TSL – Test and Set Locked:**
  - Instrução básica disponibilizada por computadores **Multiprocessados**;
  - **Funcionamento**:
    - Transfere o conteúdo de um endereço de memória para um registrador, e depois armazena um valor não nulo em tal endereço;
    - A transferência do conteúdo é indivisível (bloqueia o barramento de memória);
    - Utiliza uma variável compartilhada FLAG, para coordenar o acesso a memória;
      - FLAG = 0, qualquer processo poder setar para 1 e ler ou escrever na memória compartilhada;
      - Ao terminar, FLAG = 0 (usando instrução MOVE)

Concorrência entre Processos

Comunicação Inter-processos

- Instrução TSL – Test and Set Locked:

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET | return to caller
```

A Instrução TSL

- **Vantagens da TSL**:
  - Simplicidade de uso (embora sua implementação em hardware não seja trivial).
  - Não dá aos processos de usuário o poder de desabilitar interrupções.
  - Presente em quase todos os processadores atuais.
  - Funciona em máquinas com vários processadores.

- **Desvantagens**:
  - Espera ocupada (*busy wait*).
  - Possibilidade de postergação infinita (starvation)
    - "processo azarado" sempre pega a variável *lock* com o valor 1

Concorrência entre Processos

Comunicação Inter-processos

Exclusão Mútua

Soluções de software com Busy Wait (Espera Ocupada)

- Variável de bloqueio
- Alternância estrita
- Algoritmo de Decker
- Algoritmo de Peterson

Soluções com *Busy Wait*

- ***Busy wait* = espera ativa ou espera ocupada**.
- Basicamente o que essas soluções fazem é:
  - Quando um processo quer entrar na sua R.C. ele verifica se a entrada é permitida. Se não for, **ele espera em um laço** (improdutivo) até que o acesso seja liberado.
    - Ex: While (vez == OUTRO) do {nothing};
  - **Consequência**: desperdício de tempo de CPU.
- Problema da **inversão de prioridade**:
  - Processo *LowPriority* está na sua R.C. e é interrompido. Processo *HighPriority* é selecionado mas entra em espera ativa. Nesta situação, o processo *LowPriority* nunca vai ter a chance de sair da sua R.C.

Concorrência entre Processos
Comunicação Inter-processos

- **Variáveis de Bloqueio/Travamento**:
  - Única variável compartilhada, cujo valor inicia com 0;
  - Esta variável deve ser testada pelo processo quando desejar entrar na sua região crítica;
  - Se for 0, o processo muda para 1 e entra na região crítica;
  - Se for 1, o processo deve esperar até voltar a 0;
  - Desvantagens:
    - Mesmo problema do exemplo de diretório de Spool;

1ª Tentativa - Variável de Bloqueio

- Variável de bloqueio, compartilhada, indica se a R.C. está ou não em uso.
  - *turn = 0* $\Rightarrow$ R.C. livre *turn = 1* $\Rightarrow$ R.C. em uso
- Tentativa para *n* processos:

```
var turn: 0..1
turn := 0

Process Pi:
...
while turn == 1 do {nothing};
turn := 1;
< critical section >
turn := 0;
...
```

Problemas da 1ª Tentativa

- A proposta não é correta pois dois processos podem concluir "simultaneamente" que a R.C. está livre,

- Os dois processos podem testar o valor de *turn* antes que essa variável seja feita igual a *true* por um deles.

```
var turn: 0..1
turn := 0

Process Pi:
...
while turn == 1 do {nothing};
turn := 1;
< critical section >
turn := 0;
...
```

Problemas da 1ª Tentativa

```
int *turn;

int main(){
      int shmid = shmget (IPC_CREAT, 2*sizeof(int), IPC_CREAT | 0666);
      turn = (int*)shmat (shmid, NULL, 0);

     turn[0] = 0;

     if (fork() > 0) {
         while (turn[0] == 1)  printf ("estou bloqueado - processo %d: \n", getpid());
         turn[0] = 1;
          printf ("< critical section %d: >\n", getpid());
         turn[0] = 0;
      } else {
              while (turn[0] == 1)  printf ("estou bloqueado - processo %d: \n", getpid());
              turn[0] = 1;
              printf ("< critical section %d: >\n", getpid());
              turn[0] = 0;
     }
    //libera alocacao
     shmdt(turn);
     shmctl(shmid, IPC_RMID, NULL);
    exit(1);
}
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

Concorrência entre Processos
Comunicação Inter-processos

- **Estrita Alternância**:
  - Teste contínuo do valor de uma variável, aguardando que ela assuma um determinado valor;
    - Variável TURN = 0, processo A entra na região;
    - Variável TURN = 1, processo B entra na região;

    - Tentativa para 2 processos: P1 e P2

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

(a)

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

(b)

Problemas da 2ª Tentativa

- O algoritmo garante a exclusão mútua, mas **obriga a alternância** na execução das R.C.
  - Garante exclusão mutua;

- Não é possível a um **mesmo processo entrar duas vezes** consecutivamente na sua R.C.
  - Logo, a "velocidade" de entrada na R.C. é ditada pelo processo mais lento.

- Se um **processo falhar ou terminar**, o outro não poderá mais entrar na sua R.C., ficando bloqueado permanentemente.
  - Não garante progresso;

Concorrência entre Processos
Comunicação Inter-processos

- **Estrita Alternância**:
  - Desvantagens:
    - Consome muito tempo de processamento;
    - Processo mais lento que o outro pode interferir na Estrita alternância
      - Imaginemos que processo A é mais longo que o processo B
      - Logo B deve ficar esperando A terminar;
      - Processo A deixa a R.C e faz TURN=1 e continua executando;
      - Processo B executa rapidamente, faz TURN=0 e volta para inicio do Loop;
      - O processo B não poderá entrar na sua Região Crítica pois TURN = 0, e o processo A ainda não mudou TURN=1 pois continua executando fora da sua região crítica;

Concorrência entre Processos

Comunicação Inter-processos

Estrita Alternância: Viola a condição 3

```
Whle (TRUE) {
        while (turn != 0)  //espera
        critical_region( );
        turn = 1;
        noncritical_region( );
}
```

```
xxxxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxx
xxxxxxxxxxx
```

(a)

```
Whle (TRUE) {
        while (turn != 1)  //espera
        critical_region( );
        turn = 0;
        noncritical_region( );
}
```

(b)

3ª Tentativa

- O problema da tentativa anterior é que ela guarda a ***identificação*** do processo que pode entrar na R.C.
  - Entretanto, o que se precisa, de fato, é de informação de ***estado*** dos processos (i.e., se eles ***querem*** entrar na R.C.)

- Cada processo deve então ter a sua própria "**chave de intenção**". Assim, se falhar, ainda será possível a um outro entrar na sua R.C.

- A solução se baseia no uso de uma variável ***array*** para **indicar a intenção** de entrada na R.C.

3a. Tentativa

- Antes de entrar na sua R.C, o processo examina a variável de tipo ***array***. Se ninguém mais tiver manifestado interesse, o processo indica a sua intenção de ingresso ligando o bit correspondente na variável de tipo ***array*** e prossegue em direção a sua R.C.

```
var flag: array[0..1] of boolean;
flag[0]:= false; flag[1]:= false;
```

```
Process P0:
...
while flag[1] do {nothing};
flag[0] := true;
< critical section >
flag[0] := false;
...
```

```
Process P1:
...
while flag[0] do {nothing};
flag[1] := true;
< critical section >
flag[1] := false;
...
```

Problemas da 3ª Tentativa

- Agora, se um processo **falha fora da sua R.C**. não haverá nenhum problema, nenhum processo ficará eternamente bloqueado devido a isso. Entretanto, se o **processo falhar dentro da R.C**., o problema ocorre.

- Não assegura exclusão mútua, pois cada processo pode chegar à conclusão de que o outro não quer entrar e, assim, entrarem simultaneamente nas R.C.
  - Isso acontece porque existe a possibilidade de cada processo testar se o outro não quer entrar (comando *while*) *antes* de um deles marcar a sua intenção de entrar.

4a. Tentativa

- A idéia agora é que cada **processo marque a sua intenção de entrar *antes* de testar a intenção do outro**, o que elimina o problema anterior.
- É o mesmo algoritmo anterior, porém com uma troca de linha.

```
Process P0:
...
flag[0] := true;
while flag[1] do
{nothing};
< critical section >
flag[0] := false;
...
```

```
Process P1:
...
flag[1] := true;
while flag[0] do
{nothing};
< critical section >
flag[1] := false;
...
```

Problemas da 4ª Tentativa

- **Garante a exclusão mútua** mas se um processo falha dentro da sua R.C. (ou mesmo após *setar* o seu *flag*) o outro processo ficará eternamente bloqueado.

- Uma falha fora da R.C. não ocasiona nenhum problema para os outros processos.

- **Problemão**:
  - Todos os processos ligam os seus *flags* para *true* (marcando o seu desejo de entrar na sua R.C.). Nesta situação todos os processos ficarão presos no *while* em um *loop* eterno (situação de *deadlock*).

5ª Tentativa

- Na tentativa anterior o processo assinalava a sua intenção de entrar na R.C. sem saber da intenção do outro, não havendo oportunidade dele mudar de ideia depois (i.e., mudar o seu estado para "*false*").

- A **5a. tentativa corrige este problema**:
  - Após testar no *loop*, se o outro processo também quer entrar na sua R.C, em caso afirmativo, o processo com a posse da UCP declina da sua intenção, dando a vez ao parceiro.

```
Process P0:
...
flag[0] = 1;
while (flag[1])
   {
     flag[0] = 0;
     <delay for a short time>
     flag[0] = 1;
   }
< critical section >
flag[0] = 0;
...
```

```
Process P1:
...
flag[1] = 1;
while (flag[0])
   {
     flag[1] = 0;
     <delay for a short time>
     flag[1] = 1;
   }
< critical section >
flag[1] = 0;
...
```

5a. Tentativa (cont.)

Entrega da Implementação do Vetor de Intenção

```
int main(){
     int shmid = shmget (IPC_CREAT, 2*sizeof(int), IPC_CREAT | 0666);
     turn = (int*)shmat (shmid, NULL, 0);

    turn[0] = 0;

    if (fork() > 0) {
       while (turn[0] == 1)  printf ("estou bloqueado - processo %d: \n", getpid());
       turn[0] = 1;
        printf ("< critical section %d: >\n", getpid());
       turn[0] = 0;
     } else {
           while (turn[0] == 1)  printf ("estou bloqueado - processo %d: \n", getpid());
           turn[0] = 1;
           printf ("< critical section %d: >\n", getpid());
           turn[0] = 0;
     }
    //libera alocacao
     shmdt(turn);
     shmctl(shmid, IPC_RMID, NULL);
    exit(1);
}
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

5a. Tentativa (cont.)

- **Esta solução é quase correta**. Entretanto, existe um **pequeno problema**: a possibilidade dos processos ficarem cedendo a vez um para o outro "indefinidamente" (problema da "**mútua cortesia**")
  - **Livelock**
- Na verdade, essa é uma situação **muito difícil de se sustentar durante um longo tempo na prática**, devido às velocidades relativas dos processos. Entretanto, ela é uma possibilidade teórica, o que invalida a proposta como solução geral do problema.

5a. Tentativa – Exemplo

P_0 seta *flag[0]* para *true*.

P_1 seta *flag[1]* para *true*.

P_0 testa *flag[1]*.

P_1 testa *flag[0]*.

P_0 seta *flag[0]* para *false*.

P_1 seta *flag[1]* para *false*.

P_0 seta *flag[0]* para *true*.

P_1 seta *flag[1]* para *true*.

Solução de Dekker - matematico holandês

- Trata-se da **primeira solução correta** para o problema da exclusão mútua de dois processos (proposta na década de 60).
- O algoritmo combina as idéias de variável de **bloqueio e *array* de intenção**.
- É similar ao algoritmo anterior mas usa uma **variável adicional (*vez/turn*)** para realizar o desempate, no caso dos dois processos entrarem no *loop* de mútua cortesia.

Algoritmo de Dekker

```
var flag: array[0..1] of boolean;
    turn: 0..1; //who has the priority

flag[0] := false
flag[1] := false
turn := 0   // or 1
```

```
Process p0:
     flag[0] := true
     while flag[1] {
         if turn ≠ 0 {
            flag[0] := false
            while turn ≠ 0 {}
            flag[0] := true
          }
     }

    // critical section
    ...
    // end of critical section
    turn := 1
    flag[0] := false
```

```
Process  p1:
   flag[1] := true
   while flag[0] {
      if turn ≠ 1 {
        flag[1] := false
        while turn ≠ 1 {}
        flag[1] := true
      }
   }

   // critical section
   ...
   // end of section
   turn := 0
   flag[1] := false
```

Algoritmo de Dekker (cont.)

- Quando *P0* quer entrar na sua R.C. ele coloca seu *flag* em *true*. Ele então vai checar o *flag* de *P1*.
- Se o *flag* de *P1* for *false*, então *P0* pode entrar imediatamente na sua R.C.; do contrário, ele consulta a variável *turn*.
- Se *turn = 0* então *P0* sabe que é a sua vez de insistir e, deste modo, fica em *busy wait* testando o estado de *P1*.
- Em certo ponto, *P1* notará que é a sua vez de declinar. Isso permite ao processo *P0* prosseguir.
- Após *P0* usar a sua R.C. ele coloca o seu *flag* em *false* para liberá-la, e faz *turn = 1* para transferir o direito para *P1*.

Algoritmo de Dekker (cont.)

- Algoritmo de Dekker resolve o problema da exclusão mútua
- Uma solução deste tipo só é aceitável se houver um número de CPUs igual (ou superior) ao número de processos que se devam executar no sistema. Porquê?
  - Poderíamos nos dar 'ao luxo' de consumir ciclos de CPU,
  - Situação rara na prática (em geral, há mais processos do que CPUs)
  - Isto significa que a solução de Dekker é pouco usada.

Entrega da Implementação do
Algoritmo de Vetor de Interrupção (5 tentativa) e Dekker
Implementação usando fork e ponteiro para exclusão mutua
Entrega código documentado e teste de mesa.
+
Decker para 3 processos

Algoritmo de Dekker (cont.)

- Contudo, a solução de Dekker mostrou que é possível resolver o problema inteiramente por software, isto é, sem exigir instruções máquina especiais.
- Devemos fazer uma modificação significativa do programa se quisermos estender a **solução de 2 para N processos**:
  - flag[] com N posições; variável turn passa a assumir valores de 1..N; alteração das condições de teste em todos os processos

Solução de Peterson

- Usa variável compartilhada TURN;
- Dois procedimentos:
  - enter_region:
    - Verifica se é seguro entrar na região crítica, caso contrário ele fica esperando;
  - leave_region:
    - Libera a entrada da região crítica para outro processo;

Concorrência entre Processos
Comunicação Inter-processos

- **Solução de Peterson**:
  - Funcionamento:
    - Processo 0 chama ***enter_region*** e seta a posição correspondente no vetor ***Interested [0] = True;***
    - Seta ***TURN = 0****;*
    - Se o processo 1 chamar ***enter_region***, ele ficará preso no Laço até ***interested [0] = FALSO***, o que ocorrerá somente quando o processo 0 chamar ***leave_region****.*
    - Mesmo que os processos 0 e 1 chamem ***enter_region*** ao mesmo tempo, somente um deles entrar na sua região critica e o outro ficará bloqueado até o processo que entrou sair da R.C;

Concorrência entre Processos

Comunicação Inter-processos

- **Solução de Peterson**:

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
      int other;                        /* number of the other process */

      other = 1 – process;              /* the opposite of process */
      interested[process] = TRUE;       /* show that you are interested */
      turn = process;                   /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
      interested[process] = FALSE;      /* indicate departure from critical region */
}
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

Algoritmo de Peterson – Versão 2

- O truque do algoritmo consiste no seguinte:
  - **Ao marcar a sua intenção de entrar, o processo já indica (para o caso de empate) que a vez é do outro.**

```
flag[0]   := false
flag[1]   := false
turn      := 0

Process P0:
    flag[0] := true
    turn := 1
    while ( flag[1] && turn == 1 ){
         // do nothing
    }
    // critical section
    ...
    // end of critical section
    flag[0] := false
```

```
Process P1:
    flag[1] := true
    turn := 0
    while ( flag[0] && turn == 0 ){
          // do nothing
    }
    // critical section
    ...
    // end of critical section
    flag[1] := false
```

Solução de Peterson (cont.)

- Exclusão mútua é atingida.
  - Uma vez que *P0* tenha feito *flag[0] = true*, *P1* não pode entrar na sua R.C.
  - Se *P1* já estiver na sua R.C., então *flag[1] = true* e *P0* está impedido de entrar.
- Bloqueio mútuo (deadlock) é evitado.
  - Supondo *P0* bloqueado no seu *while*, isso significa que *flag[1] = true* e que *turn = 1*
  - se flag[1] = true e que turn = 1, então P1 por sua vez entrará na sua seção crítica
  - Assim, *P0* só pode entrar quando **ou** *flag[1]* tornar-se *false* **ou** *turn* passar a ser 0.

Concorrência entre Processos

Comunicação Inter-processos

Exclusão Mútua

Soluções de software com Bloqueio

- Sleep / Wakeup
- Semáforos
- Monitores

Concorrência entre Processos

Comunicação Inter-processos

Primitivas SLEEP/WAKEUP

Primitivas SLEEP/WAKEUP

- Uma nova solução para o problema Espera Ocupada:
  - Espera ocupada gasta muito tempo de processamento, pois fica testando uma variável para ter permissão de entrar em sua R.C.;
  - Problema da Prioridade Invertida:
    - Processos de maior prioridade ficam impedidos de entrar em sua região crítica pois um processo de menor prioridade dentro da sua região critica não consegue entrar para executar e liberar seu RC;
- Sleep/Wakeup bloqueia a continuação da execução de um determinado processo sem gastar tempo de execução;

Primitivas SLEEP/WAKEUP

- Funcionamento:
  - Primitiva SLEEP:
    - Chamada de sistema que bloqueia o processo que a chamou;
  - Primitiva WAKEUP:
    - “Acorda” o processo que está suspenso pela primitiva SLEEP;
    - Esta chamada possui um parâmetro que identifica o processo a ser acordado;

Primitivas SLEEP/WAKEUP

- Exemplo: Problema do Produtor-Consumidor
  - Dois processos compartilham um buffer de tamanho fixo;
  - O processo Produtor coloca informações no buffer;
  - O processo Consumidor retira informações do buffer;
  - Quando o buffer estiver cheio o produtor deve dormir;
  - Quando o buffer estiver vazio o consumidor deve dormir;
  - Problema – Condições de corridas semelhante ao diretório de Spool, só que no buffer;

Primitivas SLEEP/WAKEUP

- Problema de **Condições de Corrida** no problema Prod/Cons.
  - Utiliza uma variável **COUNT** para controlar o numero de itens no buffer
  - Se COUNT = 0, consumidor dorme; Se COUNT = N, produtor dorme;
  - Situação do problema: Buffer vazio
    - Consumidor lê COUNT para testar se é 0 e ocorre mudança contexto;
    - Produtor passa a executar e coloca um item no buffer, COUNT = 1;
    - Produtor tenta acorda consumidor através da WAKEUP, mas a chamada é perdida, pois consumidor não está dormindo;
    - Quando consumidor voltar a executar ele vai dormir, pois COUNT lido será 0;
    - Produtor quando chegar COUNT = N também vai dormir;
    - Ambos dormirão eternamente;

Primitivas SLEEP/WAKEUP

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Concorrência entre Processos

Comunicação Inter-processos

SEMÁFOROS

SEMÁFOROS

- **Variável inteira, não negativa**, manipulada por duas **instruções atômicas (indivisíveis)**:
  - **DOWN , Wait ou P** = *proberen* (testar)
    - Verifica se o valor do semáforo é maior que 0;
    - Se for, **decrementa-o em 1** e o processo continua sua execução;
    - Se for **zero**, o processo é posto para **dormir** na fila do semáforo;
  - **UP, Signal ou V** = *verhogen* (incrementar): **Incrementa o valor** do semáforo e um processo que aguarda na fila de espera deste semáforo é acordado;
- As chamadas trabalham as **pares**. Para cada DOWN um
- Pode haver vários processos sobre um semáforo:
  - Deve ser escolhido um de algum forma (randomicamente p.ex.);

SEMÁFOROS

- Semáforos que assumem somente os valores *0* e *1* são denominados ***semáforos binários*** **ou** ***mutex***.
- Neste caso, P e V são chamadas de ***LOCK*** **e** ***UNLOCK***, respectivamente.

```
Down(S):
  If S > 0
       Then S := S – 1
  Else bloqueia processo (coloca-o na fila de S)


UP(S):
 If algum processo dorme na fila de S
       Then acorda processo
 S := S + 1
```


Linguagens como Java oference essa estrutura: java.util.concurrent.Semaphore

Uso de Semáforos (1)

- Exclusão mútua (**semáforos binários**):

```
...
Semaphore mutex = 1;          /*var semáforo = 1*/
```

Processo P_1	**Processo P_2**	...	**Processo P_n**
...	...		...
Down(mutex)	Down(mutex)		Down(mutex)
// R.C.	// R.C.		// R.C.
UP(mutex)	UP(mutex)		UP(mutex)
...	...		...

Uso de Semáforos (2)

- Alocação de Recursos (**semáforos contadores**):

```
...
Semaphore S := 3;    /*var. semáforo, iniciado com
                       qualquer valor inteiro */
```

Processo P_1	Processo P_2	Processo P_3
...	...	...
Down(S)	Down(S)	Down(S)
//usa recurso	//usa recurso	//usa recurso
UP(S)	UP(S)	UP(S)
...	...	...

Uso de Semáforos (3)

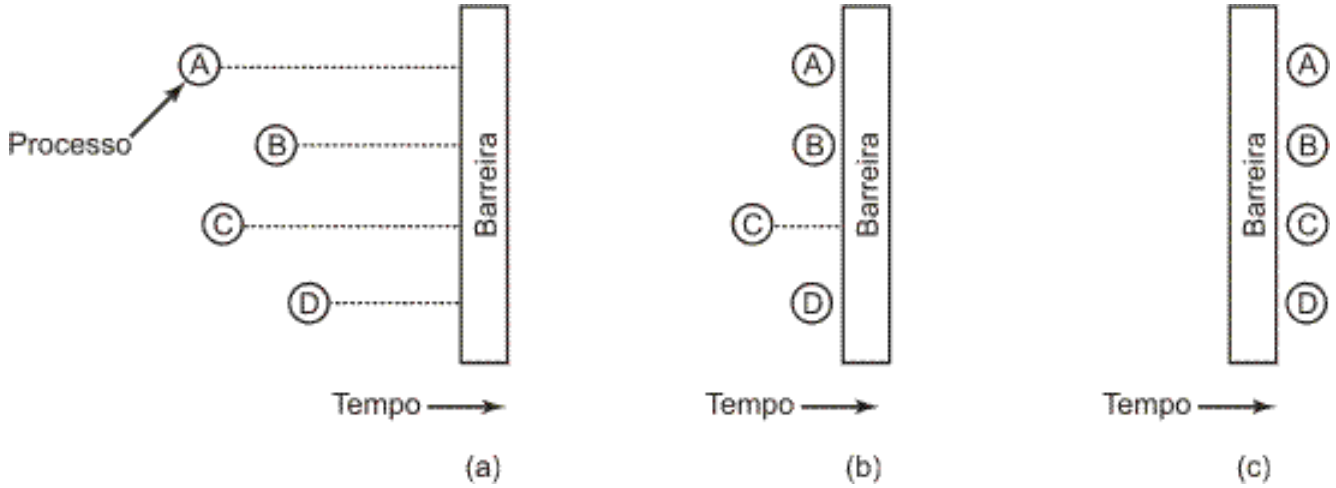
- **Relação de precedência** entre processos:
  (Ex: executar *p1_rot2* somente depois de *p0_rot1*)

```
semaphore S = 0 ;
parbegin
      begin                          /* processo P0*/
            p0_rot1()
            UP(S)
            p0_rot2()
      end
      begin                          /* processo P1*/
            p1_rot1()
            Down(S)
            p1_rot2()
      end
parend
```

Uso de Semáforos (4)

- Sincronização do tipo **barreira**: (*n-1* processos aguardam o *n-ésimo* processo para todos prosseguirem)

```
Y,Z: semaphore initial 0;
...

P1:                    P2:
   ...                    ...
   UP(Z);                 UP(Y);
   Down(Y);               Down(Z);
   A;                     B;
   ...                    ...
```



- Uso de uma barreira
  a) processos se aproximando de uma barreira
  b) todos os processos, exceto um, bloqueados pela barreira
  c) último processo chega, todos passam

SEMÁFOROS

- **Exemplo: Problema do Produtor-Consumidor Usando Semáforos com Buffer Limitado**
  - Utiliza três semáforos:
    - FULL = 0
      - Conta o numero de posições do buffer que foram preenchidas;
    - EMPTY = N
      - Conta o numero de posições do buffer ainda vazias;
    - MUTEX = 1
      - Assegurar que somente um processo acesse o buffer por vez (ler/escrever) - gerencia a exclusão mutua;

SEMÁFOROS

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

```
DOWN(S):
  SE S > 0 ENTÂO S := S – 1
  SENÂO bloqueia processo


UP(S):
 SE algum processo dorme na fila de S
  ENTÂO acorda processo
  S := S + 1
```

Deficiência dos Semáforos (1)

- Exemplo: suponha que os dois ***down*** do código do produtor estivessem **invertidos**. Neste caso, *mutex* seria diminuído antes de *empty*. Se o *buffer* estivesse completamente cheio, o produtor bloquearia com *mutex = 0*. Portanto, da próxima vez que o consumidor tentasse acessar o *buffer* ele faria um *down* em *mutex*, agora zero, e também bloquearia. **Os dois processos ficariam bloqueados eternamente**.

- **Conclusão**: erros de programação com semáforos podem levar a resultados imprevisíveis.

Deficiência dos Semáforos (2)

- Embora semáforos forneçam uma abstração flexível o bastante para tratar diferentes tipos de problemas de sincronização, ele é **inadequado em algumas situações**.

- Semáforos são uma abstração de alto nível baseada em primitivas de baixo nível, que provêm atomicidade e mecanismo de bloqueio, com manipulação de filas de espera e de escalonamento. Tudo isso contribui para que a **operação seja lenta**.

- Para alguns recursos, isso pode ser tolerado; para outros esse tempo mais longo é **inaceitável**.
  - Ex: (Unix) Se o bloco desejado é achado no *buffer cache*, *getblk()* tenta reservá-lo com P(). Se o *buffer* já estiver reservado, não há nenhuma garantia que ele conterá o mesmo bloco que ele tinha originalmente.

Concorrência entre Processos

Comunicação Inter-processos

MONITOR

Monitores (1)

- Sugeridos por Dijkstra (1971) e desenvolvidos por Hoare (1974) e Brinch Hansen (1975), são estruturas de **sincronização de alto nível**, que têm por objetivo impor (forçar) uma boa estruturação para programas concorrentes.

- **Motivação**:
  - Sistemas baseados em algoritmos de exclusão mútua ou **semáforos estão sujeitos a erros de programação**.
  - Embora estes devam estar inseridos no código do processo, não existe nenhuma reivindicação formal da sua presença.
  - Assim, erros e omissões (deliberadas ou não) podem existir e a **exclusão mútua pode não ser atingida**.

MONITOR

- Facilitar a vida o programador;
  - A implementação de Semáforos deve ser feita com muito cuidado;

- Monitor é uma primitiva de alto nível para sincronização de processos;

- **Um monitor é um conjunto de procedimentos, variáveis e estruturas de dados, todas agrupadas juntas em um módulo especial;**

- Os processos podem chamar os procedimentos do monitor sempre que quiserem, mas não podem acessar diretamente as estruturas de dados e variáveis internas do monitor;

Monitores (2)

- Solução:
  - Tornar obrigatória a exclusão mútua. Uma maneira de se fazer isso é **colocar as seções críticas em uma área acessível somente a um processo de cada vez**.

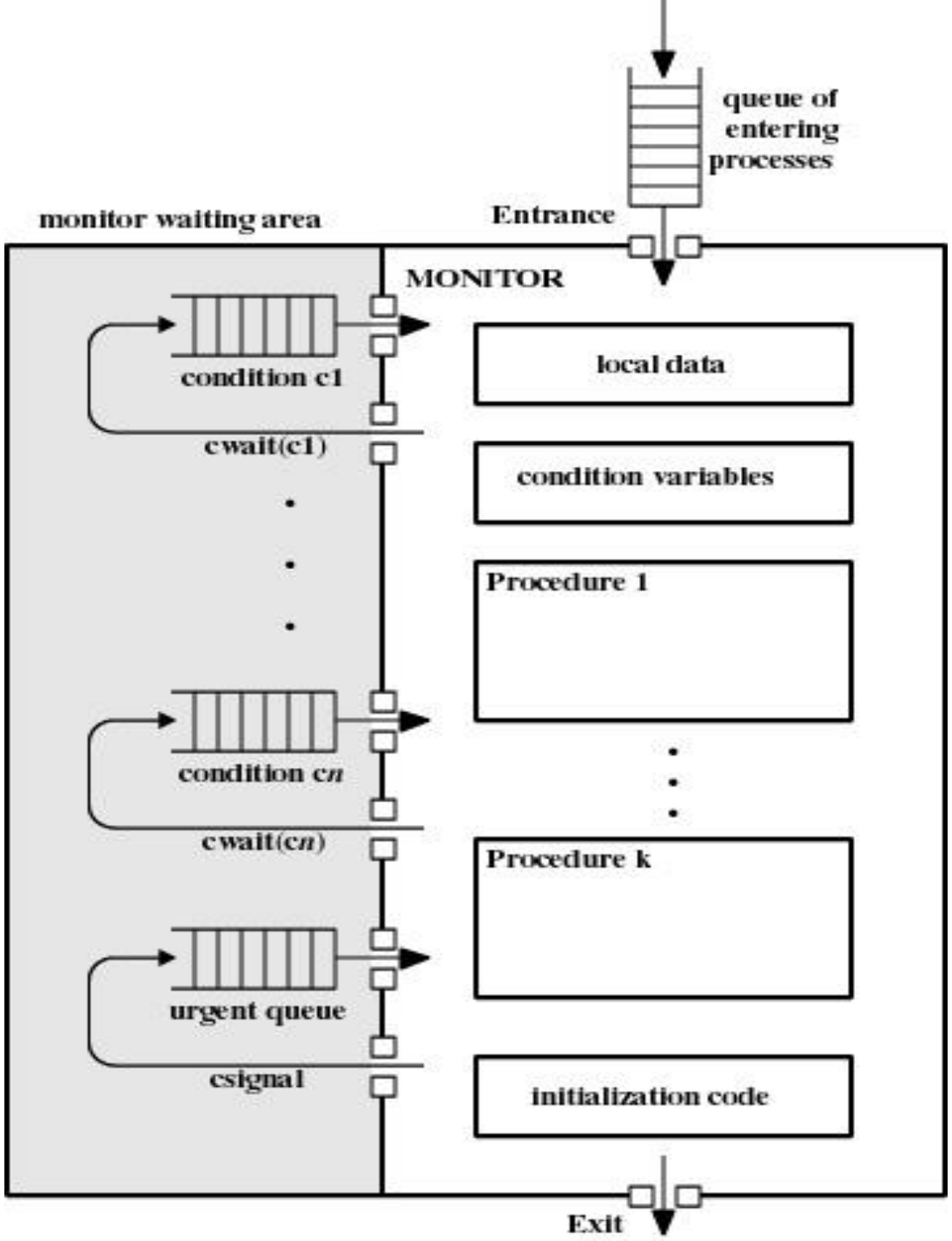
- Ideia central:
  - Em vez de codificar as **regiões críticas** dentro de cada processo, podemos **codificá-las como procedimentos** (*procedure entries*) do monitor. Assim, quando um processo precisa referenciar dados compartilhados, ele simplesmente chama um procedimento do monitor.
  - **Resultado**: o código da **região crítica não é mais duplicada** em cada processo.

MONITOR

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

MONITOR

- Propriedade importante para exclusão mútua:
  - Somente um processo pode estar ativo dentro do monitor em um dado instante de tempo;
- O compilador é responsável por implementar a exclusão mútua sobre o monitor;
- Normalmente dentro dos procedimentos do monitor são colocados as regiões críticas;



Visão da Estrutura de um Monitor

```
Processo P1
  Begin
  ...
  Chamada a um procedimento do monitor
  ...
  End
```

```
Processo P2
  Begin
  ...
  Chamada a um procedimento do monitor
  ...
  End
```

```
Processo P3
  Begin
  ...
  Chamada a um procedimento do monitor
  ...
  End
```

Chamada de procedimento do Monitor

Variáveis de Condição (1)

- São **variáveis que estão associadas a condições** que provocam a suspensão e a reativação de processos. Permitem, portanto, sincronizações do tipo *sleep-wakeup*.

- **Só podem ser declaradas dentro do monitor** e são sempre usadas como argumentos de dois comandos especiais:
  - *Wait* (ou *Delay*)
  - *Signal* (ou *Continue*)

MONITOR

- Variáveis de Condição:
  - Duas operações possíveis:
    - WAIT
      - Faz com que o monitor suspenda o processo que fez a chamada. O monitor armazena as informações sobre o processo suspenso em uma estrutura de dados (fila) associada à variável de condição.
    - SIGNAL
      - Faz com quer o monitor reative UM dos processos suspensos na fila associada à variável de condição.

MONITOR

- Regras sobre operações sobre variáveis de condição:
  - O processo acordado ganha o direito de executar, suspendendo o processo que o acordou;
  - O processo que executar um SIGNAL é obrigado a deixar o monitor em seguida à chamada da primitiva;
    - Operação **SIGNAL aparece como a última instrução** no procedimento do monitor;

MONITOR

```
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;
```

Troca de Mensagens

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

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

     while (TRUE) {
          item = produce_item( );               /* gera alguma coisa para colocar no buffer */
          receive(consumer, &m);                /* espera que uma mensagem vazia chegue */
          build_message(&m, item);              /* monta uma mensagem para enviar */
          send(consumer, &m);                   /* envia item para consumidor */
     }
}

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

     for (i = 0; i < N; i++) send(producer, &m);  /* envia N mensagens vazias */
     while (TRUE) {
          receive(producer, &m);                /* pega mensagem contendo item */
          item = extract_item(&m);              /* extrai o item da mensagem */
          send(producer, &m);                   /* envia a mensagem vazia como resposta */
          consume_item(item);                   /* faz alguma coisa com o item */
     }
}
```

O problema do produtor-consumidor com N mensagens

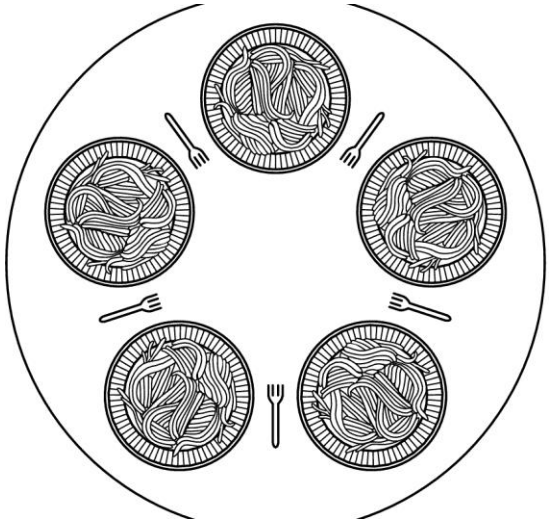
Concorrência entre Processos

Comunicação Inter-processos

Problemas Clássicos de Comunicação entre Processos

Problema dos Filósofos Glutões

- 5 filósofos ao redor de uma mesa;
- Cada filósofo tem a sua frente um prato de macarrão;
- Para comer é necessário dois garfos;
- Entre cada prato existe um garfo;



Problema dos Leitores e Escritores

1. Problema dos Leitores e Escritores

   - Processos acessando uma base de dados ao mesmo tempo;
   - Durante uma escrita nenhum processo poderá ler ou escrever na base de dados;
   - Problema: como podemos programar este idéia;

Problema do Barbeiro Dorminhoco

1. Problema do Barbeiro Dorminhoco

- Um barbeiro, uma cadeira e diversos lugares de espera;
  - Se houver lugares esperam, caso contrário vão embora
- Se não houver clientes o barbeiro senta na cadeira e cai no sono;
- Problema: programar tanto o barbeiro quanto os clientes, sem que haja ocorrência de condições de corrida;

Referências

- Livro do Tanenbaum
  - Sistemas Operacionais Modernos
  - **www.cs.vu.nl/~ast**
- Livro do Silberschatz
  - Operating System Concepts
  - **www.bell-labs.com/topic/books/aos-book/**
- Livro do Machado e Maia
  - Arquitetura de Sistemas Operacionais.

Referências

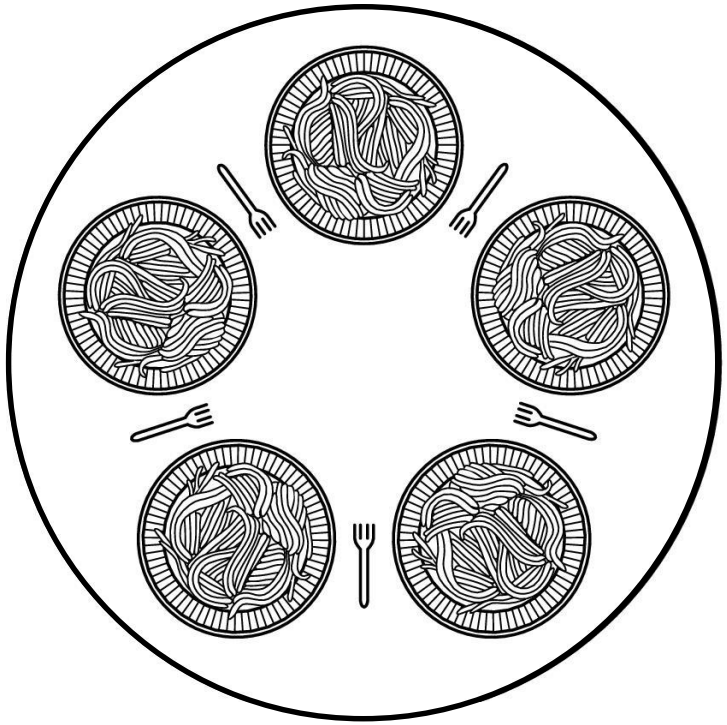
- Silberschatz A. G.; Galvin P. B.; Gagne G.; "Fundamentos de Sistemas Operacionais", 6a. Edição, Editora LTC, 2004.
  - Capítulo 7 (até seção 7.3 inclusa)
- A. S. Tanenbaum, "Sistemas Operacionais Modernos", 2a. Edição, Editora Prentice-Hall, 2003.
  - Seção 2.3 (até 2.3.3 inclusa)
- Deitel H. M.; Deitel P. J.; Choffnes D. R.; “Sistemas Operacionais”, 3ª. Edição, Editora Prentice-Hall, 2005
  - Capítulo 5 (até seção 5.4.2 inclusa)

Referências

- A. S. Tanenbaum, "Sistemas Operacionais Modernos", 2a. Edição, Editora Prentice-Hall, 2003.
  - Seções 2.3.7
- Silberschatz A. G.; Galvin P. B.; Gagne G.; "Fundamentos de Sistemas Operacionais", 6a. Edição, Editora LTC, 2004.
  - Seção 7.7
- Deitel H. M.; Deitel P. J.; Choffnes D. R.; “Sistemas Operacionais”, 3ª. Edição, Editora Prentice-Hall, 2005
  - Seções 6.2 e 6.3
- Monitores em Java
  - Link prog concorrente em java
    - http://www.mcs.drexel.edu/~shartley/ConcProgJava/monitors.html
  - Capitulo sobre java monitors
    - http://java.sun.com/developer/Books/performance2/chap4.pdf

Jantar dos Filósofos (1)

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



Jantar dos Filósofos (2)

```
#define N 5                                 /* número de filósofos */

void philosopher(int i)                     /* i: número do filósofo, de 0 a 4 */
{
    while (TRUE) {
        think( );                           /* o filósofo está pensando */
        take_fork(i);                       /* pega o garfo esquerdo */
        take_fork((i+1) % N);               /* pega o garfo direito; % é o operador modulo */
        eat( );                             /* hummm! Espaguete! */
        put_fork(i);                        /* devolve o garfo esquerdo à mesa */
        put_fork((i+1) % N);                /* devolve o garfo direito à mesa */
    }
}
```

Uma solução errada para o problema do jantar dos filósofos

Jantar dos Filósofos (3)

```
#define N           5           /* número de filósofos */
#define LEFT        (i+N-1)%N   /* número do vizinho à esquerda de i */
#define RIGHT       (i+1)%N     /* número do vizinho à direita de i */
#define THINKING    0           /* o filósofo está pensando */
#define HUNGRY      1           /* o filósofo está tentando pegar garfos */
#define EATING      2           /* o filósofo está comendo */
typedef int semaphore;          /* semáforos são um tipo especial de int */
int state[N];                   /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;            /* exclusão mútua para as regiões críticas */
semaphore s[N];                 /* um semáforo por filósofo */

void philosopher(int i)         /* i: o número do filósofo, de 0 a N-1 */
{
    while (TRUE) {              /* repete para sempre */
        think( );               /* o filósofo está pensando */
        take_forks(i);          /* pega dois garfos ou bloqueia */
        eat( );                 /* hummm! Espaguete! */
        put_forks(i);           /* devolve os dois garfos à mesa */
    }
}
```

Uma solução para o problema do jantar dos filósofos (parte 1)

Jantar dos Filósofos (4)

```
void take_forks(int i)                  /* i: o número do filósofo, de 0 a N–1 */
{
     down(&mutex);                      /* entra na região crítica */
     state[i] = HUNGRY;                 /* registra que o filósofo está faminto */
     test(i);                           /* tenta pegar dois garfos */
     up(&mutex);                        /* sai da região crítica */
     down(&s[i]);                       /* bloqueia se os garfos não foram pegos */
}

void put_forks(i)                       /* i: o número do filósofo, de 0 a N–1 */
{
     down(&mutex);                      /* entra na região crítica */
     state[i] = THINKING;               /* o filósofo acabou de comer */
     test(LEFT);                        /* vê se o vizinho da esquerda pode comer agora */
     test(RIGHT);                       /* vê se o vizinho da direita pode comer agora */
     up(&mutex);                        /* sai da região crítica */
}

void test(i)                            /* i: o número do filósofo, de 0 a N–1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

Uma solução para o problema do jantar dos filósofos (parte 2)

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

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