COMUNICAÇÃO / SINCRONIZAÇÃO ENTRE PROCESSOS

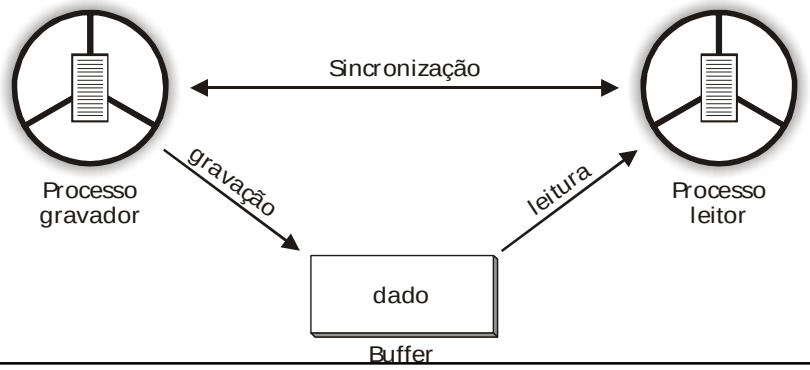
Prof. Maicon A. Sartin

Introdução

- Os processos podem trabalhar em conjunto de forma concorrente, compartilhando recursos do sistema como:
  - Arquivos, registros, dispositivos e áreas de memória
  - Condições de Corrida - Quando o recurso é compartilhado ao mesmo tempo
    - 2 ou mais processos estão lendo e gravando dados compartilhados e o resultado final depende da ordem de execução
- Ex.: 2 processos concorrentes trocam informações através de gravação e leitura em um buffer
  - A gravação só poderá acontecer se o buffer não estiver cheio
  - A leitura só poderá acontecer se existir dados no buffer

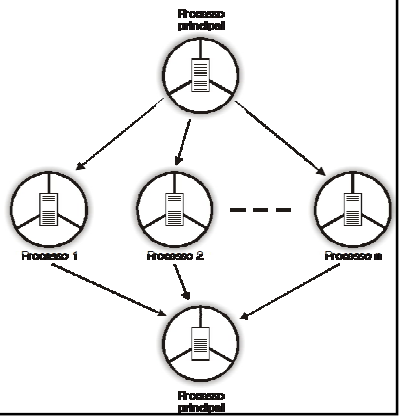
Introdução

- Comunicação/Sincronização entre processos
  - Os *mecanismos de sincronização* garantem:
    - A comunicação entre os processos concorrentes
    - O acesso a recursos compartilhados



Concorrência

- Programação Concorrente
  - Conjunto de processos (ou threads) sequenciais que podem executar independetemente um do outro, ou cooperativamente



Concorrência

- Conway, 1963 e Dennis & Horn, 1966:
  - Apresentaram as primeiras notações de concorrência em um programa , através dos comandos FORK e JOIN
  - São utilizados no SO Unix e Linux
  - O FORK cria um outro processo para execução do programa B
  - O JOIN sincroniza o programa A com o B, A só continua depois do término de B

```
PROGRAM A;
.
.
FORK B;
.
.
JOIN B;
.
.
END.
```

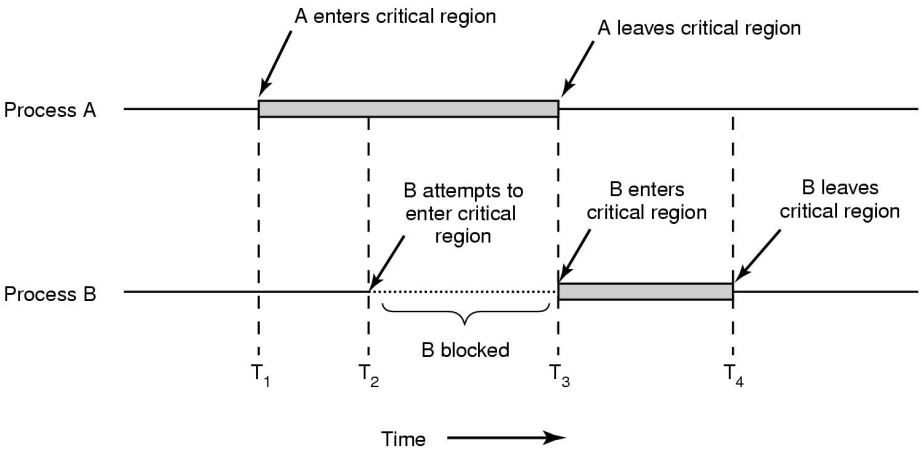
```
PROGRAM B;
.
.
.
.
END.
```

Concorrência

- Problema em Compatilhamento de Recursos
  - Compartilhando a mesma variável em processos diferentes e nos 2 processos existem atribuição a esta variavel
- Solução
  - Impedir que 2 ou mais processos acessem um mesmo recurso ao mesmo tempo
- Exclusão Mútua (Mutual Exclusion)
  - Quando um processo estiver acessando determinado recurso, os outros deverão esperar até que ele termine seu acesso
  - Exclusividade em processos concorrentes
  - Região Crítica
    - A parte do código do programa onde é feito o acesso ao recurso compartilhado

Concorrência

- Região Crítica



Solução

- Quatro condições são necessárias para garantir a exclusão mútua
  1. Dois ou mais processos não podem estar simultaneamente em suas regiões críticas
  2. Nenhum processo fora da sua região crítica pode bloquear outro processo
  3. Nenhum processo pode esperar infinitamente para entrar na região crítica
  4. Não se deve fazer nenhuma consideração quanto ao número de CPUs e suas velocidades

Outros Problemas

- Problema na velocidade de execução
  - Onde um processo fica dependente do outro para entrar na região crítica
- Starvation
  - Onde um processo nunca consegue acessar a região crítica e o recurso compartilhado
- Sincronização Condicional
  - Quando o recurso compartilhado não está pronto para ser acessado
  - Assim, o processo é colocado em estado de espera até o recurso estiver pronto

Propostas para Exclusão Mútua

1. Desabilitar Interrupções
2. Espera Ocupada
3. Bloqueio de Processos
4. Semáforos
5. Monitores

Soluções

- Desabilitar Interrupções
  - Cada processo, logo após entrar na região crítica, desabilita as interrupções e reabilita as interrupções antes de sair
  - Pode causar problemas, pois não é uma boa idéia dar direitos aos processos de inibir interrupções

Soluções

- Espera Ocupada
  - Permanece testando uma variável continuamente até que a seção crítica seja liberada
  - Deve ser evitado pois desperdiça tempo de CPU
  - É utilizada quando há uma expectativa razoável de que espera seja curta
  - Viola a condição 3, mostrada anteriormente

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

Processo 0

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

Processo 1

Soluções

- Solução de Peterson
  - A primeira solução para o problema EM entre 2 processos foi apresentada pelo matemático Dekker (1965)
  - Peterson (1981) criou uma solução mais simples utilizando espera ocupada com variáveis de bloqueio e variáveis de aviso
  - Porém, ambas as soluções têm o defeito de necessitar da espera ocupada

Soluções

- Solução de Peterson

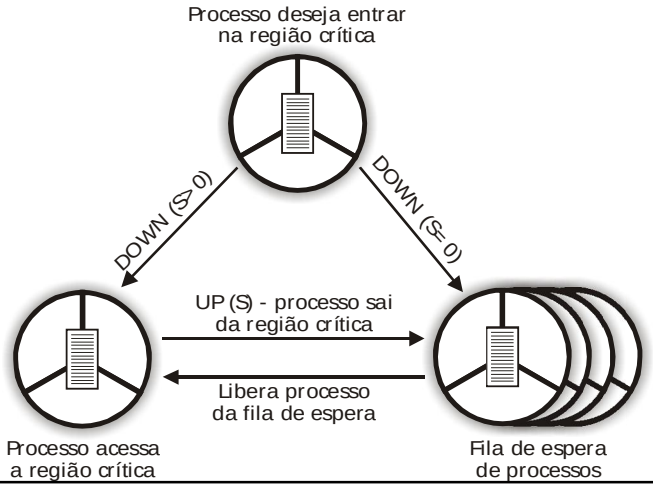
```
#define FALSE       0
#define TRUE        1
#define N           2                                   /*  número de processos */

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

void enter_region(int process);                         /*  o process é 0 ou 1 */
{
 int other;                                             /*  número dos outros processos */

 other = 1 – process;                                   /*  o oposto do processo */
 interested[process] = TRUE;                            /*  mostra que você está interessado */
 turn = process;                                        /*  define o sinalizador */
 while (turn == process && interested[other] == TRUE)   /*  declaração nula */;
}

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

Soluções

- Semáforos
  - Proposto por Dijkstra (1968) mais geral e simples de ser implementada
  - Um Semáforo é uma variável inteira, não negativa, manipulada por duas instruções:
    - DOWN e UP (ou P e V)
      - Funcionam como protocolos de E/S da região crítica
      - São implementadas como rotinas do sistema (SC)
  - O semáforo fica associado a um recurso compartilhado
    - S > 0
      - Nenhum processo está utilizando o recurso
      - Pode executar a região crítica
    - S = 0
      - Fica impedido o acesso
      - O processo fica no estado de espera em uma fila associada ao semáforo

Soluções

- Utilização de Semáforo

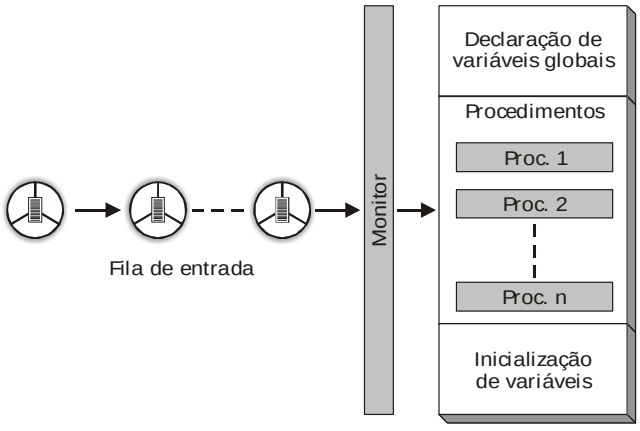


Soluções

- Monitores
  - No uso de semáforos
    - O programador deve tomar cuidado para não levar à problemas de sincronização imprevisiveis e difíceis de reproduzir.
  - Os monitores são mecanismos de sincronização de alto nível
    - Propostos por Hansen (1973) e Hoare (1974)
    - É um conjunto de procedimentos, variáveis e estrutura de dados definidos dentro de um módulo.
    - Tentam tornar mais fácil o desenvolvimento e correção de programas concorrentes
    - Principal característica é a implementação automática de EM em seus procedimentos

Soluções

- Estrutura do Monitor



Soluções

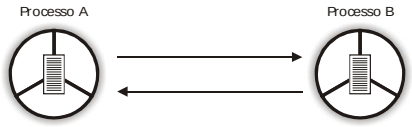
- Troca de Mensagens
  - É um mecanismo de comunicação e sincronização entre processos
  - Implementado através de duas rotinas do sistema
    - SEND (receptor, mensagem);
    - RECEIVE (transmissor, mensagem);

Soluções

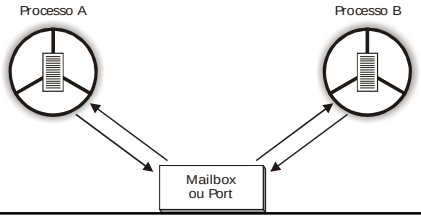
- Troca de Mensagens
  - SEND e RECEIVE
    - Desvantagem
      - Não são executados com EM
      - Pode haver perda de mensagens
        - Contornado com ACK (mensagem do receptor para o transmissor de ok)
    - Vantagem
      - Pode-se obter dados de outro processo
      - Restrição da ordem de ocorrência dos eventos
        - Uma mensagem não pode ser lida, se não foi enviada

Soluções

- Troca de Mensagens
  - SEND e RECEIVE
    - Comunicação Síncrona e Endereçamento Direto

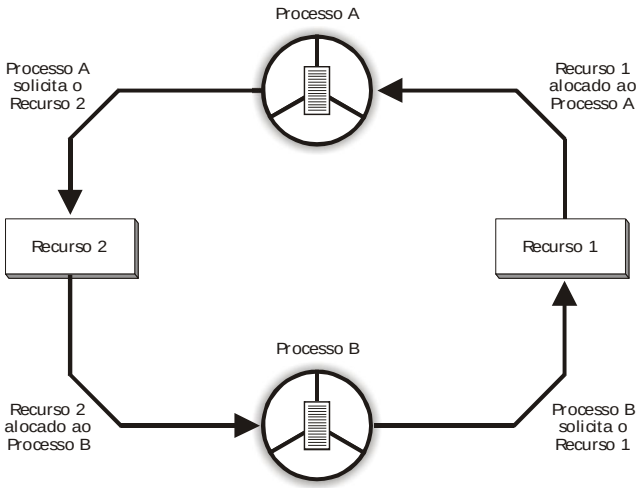


    - Comunicação Assíncrona e Endereçamento Indireto



Deadlock

- **Deadlock – Espera Circular**

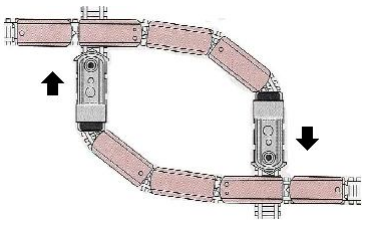


Deadlock

- Quatro condições para que ocorram situações de Deadlock, segundo Coffman (1971):
  1. Cada recurso só pode estar alocado a um único processo em um determinado instante (EM)
  2. Um processo, além dos recursos já alocados, pode estar esperando por outros recursos
  3. Um recurso não pode ser liberado de um processo só porque outros processos desejam o mesmo recurso (não-preempção)
  4. Um processo pode ter de esperar por um recurso alocado a outro processo e vice-versa (espera circular)

Deadlock

- Esse problema existe em qualquer sistema multiprogramável
- As soluções implementadas devem considerar o tipo de sistema e o impacto em seu desempenho
- Existem mecanismos de prevenção, detecção e correção de deadlocks



Deadlock

- Prevenção
  - É preciso garantir que uma das quatro condições necessárias para sua ocorrência nunca se satisfaça
    - A 1° condição, já foi vista anteriormente que a sua ausência pode causar sérios danos ao sistema
    - A 2° condição tem duas opções possíveis:
      - O processo que possui um recurso garantido não poderá acessar um novo
      - Na criação do processo aloca-se os recursos necessários para sua execução
      - Problemas
        - Grande desperdício na utilização dos recursos
        - Dificuldade em determinar o número de recursos necessários
        - Possibilidade de um processo ocasionar starvation

Deadlock

- Prevenção
  - A 3° condição pode ser evitada quando permiti-se que um recurso seja retirado de um processo
    - Pode ocasionar sérios problemas, o processo pode perder todo o seu trabalho
    - Possibilidade de sofrer starvation
  - A Última condição é excluindo a possibilidade de espera circular
    - Forçar o processo a ter apenas um recurso de cada vez
    - Se necessitar de um novo recurso deve liberar o primeiro
  - O sistema ficará limitado se evitar ocorrência de qualquer uma das condições

Deadlock

- Solução para Prevenção
  - É possível evitar o deadlock através do algoritmo do Banqueiro proposto por Dijkstra (1968)
    - Mantém todas as condições
    - Exige que os processos informem o número máximo de cada tipo de recurso necessário a sua execução
    - Antes de garantir ao processo um recurso, verifica se o pedido de recurso pode causar um deadlock
  - Apesar de evitar o deadlock esta solução possui várias limitações
    - Necessidade de um número fixo de processos ativos e de recursos do sistema
      - Pela dificuldade em prever esses números, impede que esta solução seja implementada

Deadlock

- Detecção
  - É o mecanismo que determina a existência de um deadlock
  - Permite identificar os recursos e processos envolvidos no problema
  - Para detectar a presença do deadlock:
    - O SO deve manter uma estrutura de dados capaz de identificar cada recurso do sistema
    - O algoritmo percorre toda a estrutura sempre que um recurso for solicitado e ele não pode ser garantido imediatamente
    - Esse ciclo de busca pode variar dependendo do tipo do sistema
    - Pode causar overhead no sistema

Deadlock

- Correção
  - Para corrigir primeiro tem que se detectar o deadlock
  - Na maioria dos SOs uma solução é :
    - Eliminar um ou mais processos envolvidos e desalocar seus recursos
    - Os processos eliminados não podem ser recuperados
    - A escolha dos processos são feitas aleatoriamente ou por prioridade
  - Outra solução seria o rollback:
    - Suspender um processo e liberar seus recursos até que o ciclo de espera termine
    - Depois retornar o processo sem perder seu processamento
    - Gera overhead e sua implementação é complexa

Referências

TANEMBAUN, A. S.; WOODHULL, A. S. "Sistemas Operacionais – Projeto e Implementação". Bookman, 2000.

MACHADO, F. B. "Arquitetura de sistemas operacionais". LTC, 1997.

ANDL JUNIOR, P. . Notas sobre Sistemas Operacionais. Itatiba, 2004.

GUALEVE, J. A. F. "Sistemas Operacionais". Brasília: Universidade Católica de Brasília, 2006.