Sistemas Operacionais I

Comunicação e Mecanismos de Sincronização

Prof. Carlos Eduardo de B. Paes
Departamento de Ciência da Computação
Pontifícia Universidade Católica de São Paulo

Estrutura da Aula

- Comunicação Interprocesso (IPC)
  - Condição de Corrida
  - Operações Atômicas
  - Região Crítica
  - Exclusão Mútua
  - Sincronização
- Mecanismos de Sincronização
  - Com Espera Ocupada (busy waiting)
  - Sem Espera Ocupada

Comunicação Interprocessos
(IPC - Interprocess Communication)

- Os processos freqüentemente se comunicam com os outros processos, existe portanto a necessidade de comunicar processos.
- Esta comunicação (cooperação) normalmente é através de compartilhamento de posições de memória. Analogamente, threads compartilham o mesmo espaço de endereçamento, ou seja, têm as mesmas variáveis globais

Comunicação Interprocessos Necessidades e Problemas

- São necessários mecanismos para processos realizarem comunicação e sincronização
- Por que IPC?
- Compartilhamento de informações
- Modularidade;
- Conveniência

Comunicação Interprocessos
Problemas

- Quais problemas podem ocorrer quando dois ou mais processos compartilham variáveis?
- Considere dois processos que compartilham as variáveis A e B. Qual será o resultado final? Importa a ordem na qual os processos executam?

Comunicação entre Processos
Problemas

PROCESSO 1	***PROCESSO 2***
A = B + 1	B = 2 * B

Qual é o resultado final ?

O que aconteceria se existirem múltiplos processadores?

Comunicação entre Processos
Problema

Transferência por Telefone

Aplicação pela Internet

Saldo Conta 10.012
R$4.500,00

Depositar
R$ 3.000,00

Retirar
R$ 4.000,00

Saldo Conta 10.012
R$3.500,00, R$ 7.500,00
ou R$ 500,00

Comunicação entre Processos
Condição de Corrida

- Situação em que dois ou mais processos estão acessando dados compartilhados, e o resultado final do processamento depende de quem roda quando.
- O objetivo é colocar ordem nas interações entre processos, precisamos prover algumas abstrações novas, algumas garantias sobre o que acontece e quando.

Comunicação entre Processos
Operações Atômicas

- São operações que não podem ser interrompidas. Não é possível ver as "partes" de uma operação atômica, mas apenas seu efeito final.
- Ou seja, não é possível ver a operação "em progresso".

Comunicação entre Processos
Operações Atômicas

- Exemplos:

Operações Atômicas

Tocar a Campainha
Desligar a Luz

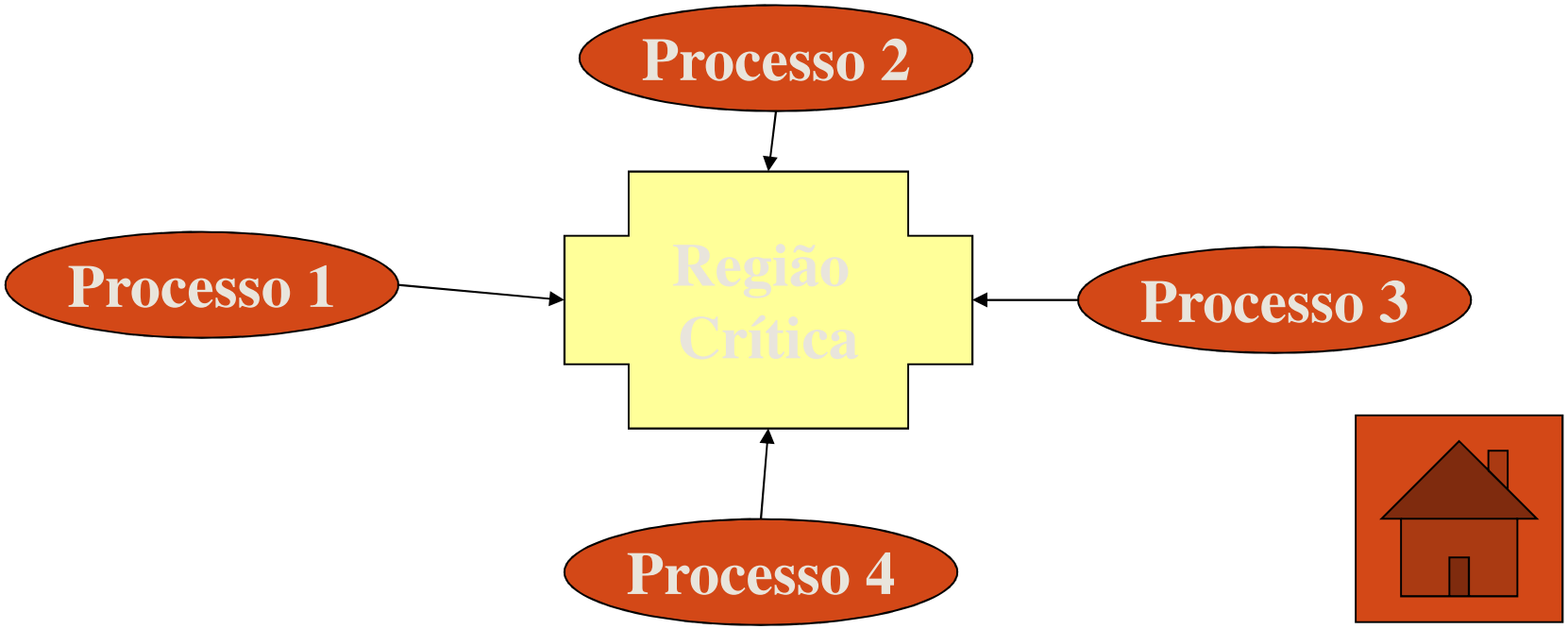
Operações Não-Atômicas

Encher um copo de água
Caminhar até a porta

Operações atômicas são relevantes em outras áreas além de Sistemas Operacionais. Elas são a base para transações atômicas , que por sua vez formam a base para uma área denominada Processamento de Transações; essa área trata de problemas de coordenação de acessos múltiplos e concorrentes a bancos de dados. Bancos eletrônicos são uma das aplicações importantes da área.

Comunicação entre Processos
Região Crítica

- É a parte do programa, onde o processamento pode levar a ocorrência de condições de corrida.



Comunicação entre Processos
Exclusão Mútua

- É a garantia que dois ou mais processos não acessem a região crítica ao mesmo tempo.
- Condições:
  - Dois os mais processos não podem estar simultaneamente dentro de suas regiões críticas.
  - Nenhum processo que esteja rodando for a da RC pode bloquear a execução de outro processo.
  - Nenhum processo pode ser obrigado a esperar indefinidamente para entrar em sua RC.

Comunicação entre Processos
Exclusão Mútua

- Propriedades importantes para um mecanismo de exclusão mútua :
  - **Ser justo** : se vários processos estão esperando, dar acesso a todos eventualmente.
  - **Eficiente** : não utilizar quantidades substanciais de recursos quando estiver esperando. Em particular, evitar "espera ocupada".
  - **Simples** : deve ser fácil de utilizar.

Comunicação entre Processos
Exclusão Mútua

- Propriedades importantes dos processos que utilizam os mecanismos:
  - Trancar sempre antes de utilizar o recurso compartilhado
  - Destrancar sempre que terminar o uso de dado compartilhado
  - Não trancar de novo se já estiver trancado o recurso.
  - Não destrancar se não foi você que trancou (pode haver exceções a esta regra)
  - Não ficar muito tempo dentro de seções críticas.

Comunicação entre Processos
Sincronização

- Consiste na utilização de operações atômicas com o objetivo de garantir o correto funcionamento de processos cooperantes.

- Exemplo: *O Problema de Espaço na Geladeira*

Comunicação entre Processos Sincronização - Exemplo

Pessoa A

- 6:00 – Olha a geladeira: sem cerveja
- 6:05 – Sai para a padaria
- 6:10 – Chega na padaria
- 6:15 – Sai da padaria
- 6:20 – Chega em casa: guarda a cerveja
- 6:25 - ?????
- 6:30 - ?????

Pessoa B

- ??? ?
- ??? ?
- Olha a geladeira: sem cerveja
- Sai para a padaria
- Chega na padaria .
- Sai da padaria
- Chega em casa: Ah ! Não !

Comunicação entre Processos
Sincronização - Exemplo

- Neste exemplo temos um problema de sincronização entre os processos (pessoas). Apenas um processo (pessoa) pode fazer alguma coisa em determinado momento. Exemplo : apenas uma pessoa pode sair para comprar cerveja em qualquer momento.
- Temos que tornar o acesso a região crítica uma operação atômica (comprar cerveja), ou seja, apenas um processo (pessoa) pode executar de cada vez.

Comunicação entre Processos
Mecanismos de Sincronização

- Para a garantia da exclusão mútua existem mecanismos de sincronização
  - Os mecanismos de sincronização podem ser classificados de duas formas:
    - Com espera ocupada (busy waiting)
    - Sem espera ocupada

Mecanismos de Sincronização

Inibição das Interrupções (*busy waiting*)

- O processo ao entrar na região crítica inibe as interrupções, garantindo assim que o processo não será interrompido por exceder o tempo de processamento cedido a ele.

- Ao sair da região crítica o processo habilita as interrupções.

Mecanismos de Sincronização

Inibição das Interrupções (*busy waiting*)

- Exemplo: Em Assembly (8088)

```
CLI     ; desabilita interrupções
        PUSH   AX
        PUSH   BX
          .
          .             ; acesso a região crítica
          .
        STI     ; habilita interrupções
```

Mecanismos de Sincronização

Inibição das Interrupções (*busy waiting*)

- Desvantagens:
  - **Desaconselhável em modo usuário**
  - **Não funciona com multiprocessadores**
- Vantagem:
  - **Útil no modo supervidor (dentro do kernel)**

Mecanismos de Sincronização

Variável de Travamento (*busy waiting*)

- Consiste em uma solução de software que utiliza uma variável de travamento para gerenciar o acesso a região crítica.

- O processo que deseja entrar na RC muda para 1 esta variável, e ao sair coloca o valor 0 na variável de travamento.

Mecanismos de Sincronização

Variável de Travamento (*busy waiting*)

Processo A

```
while (TRUE) {
    while (var_trava !=0);
     var_trava = 1;
     regiao_crítica();
     var_trava = 0;
}
```

Processo B

```
while (TRUE) {
    while (var_trava !=0);
     var_trava = 1;
     regiao_crítica();
     var_trava = 0;
}
```

Mecanismos de Sincronização

Variável de Travamento (*busy waiting*)

- Vantagem:
  - **Útil em multiprocessadores para RCs curtas, pois em alguns casos não necessita realizar troca de contexto do processador que está na RC.**
- Desvantagens:
  - **Necessita de exclusão mútua no acesso a variável de travamento;**
  - **Mecanismo de espera ocupada.**

Mecanismos de Sincronização

Estrita Alternância (*busy waiting*)

- Esta técnica consiste em que cada processo possui um valor de acesso a região crítica.
- O processo trava a entrada de outro processo quando ele acessa a RC, liberando o acesso ao sair da RC.
- Para esta solução é usado uma variável *turn* que sinaliza a entrada de cada processo.

Mecanismos de Sincronização

Variável de Travamento (*busy waiting*)

Processo A

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

Processo B

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

Mecanismos de Sincronização

Variável de Travamento (*busy waiting*)

- Vantagem:
  - **Garante a exclusão mútua**
- Desvantagens:
  - **Viola requisito de progresso**
  - **Mecanismo de espera ocupada.**

Mecanismos de Sincronização

Solução de Peterson (*busy waiting*)

- Em 1981, G. L. Peterson descobriu uma forma muito mais simples de se obter a exclusão mútua.

- A idéia é implementar dois procedimentos que são compartilhados, um para entrar na região crítica e outro para sair.

Mecanismos de Sincronização

Solução de Peterson (*busy waiting*)

- O processo que deseja acessar a RC chama o procedimento de entrada (*enter_region*) passando o "numero" do processo, e ao deixar a RC o processo chama o procedimento de saída (*leave_region*).

Mecanismos de Sincronização

Solução de Peterson (*busy waiting*)

```
#include "prototypes.h"
#define FALSE    0
#define TRUE     1
#define N        2                        /* numero de processos */

int turn;                                 /* de que é a vez  ? */
int interested[N];                        /* todos os valores iniciados com zero (FALSE) */

void enter_region (int process)   /* processo : quem está entrando (0 ou 1) */
{
  int other;                              /* número dos outros processos */
  other = 1 - process;                    /* o processo oposto */
  interested[process] = TRUE;             /* mostra que você está interessado */
  turn = process;                         /* seta a flag */
  while (turn == process && interested[other] == TRUE) /* null */ ;
}

void leave_region (int process)  /* processo : quem está saindo (0 ou 1) */
{
  interested[process] = FALSE;   /* indica saída da região crítica */
}
```

Mecanismos de Sincronização

Solução de Peterson (*busy waiting*)

- Vantagem:
  - **Garante a sincronização**
- Desvantagem:
  - **Mecanismo de espera ocupada.**

Mecanismos de Sincronização

Instrução Test and Set Lock (*busy waiting*)

- Solução de sincronização em hardware
- Alguns computadores projetados para suportar múltiplos processadores implementam um instrução básica chamada test and set lock(tsl).
- Basicamente esta instrução realiza a transferência de um posição de memória para um registrador, e depois armazena nesta posição um valor não nulo.

Mecanismos de Sincronização

Instrução Test and Set Lock (*busy waiting*)

```
enter_region:
        tsl  reg, flag          ; copia o valor de flag no registrador e seta o flag em 1
        cmp  reg, #0            ; o flag é zero ?
        jnz  enter_region       ; se não for zero, lock é setado, então oprocesso entra em loop
        ret                     ; retorna a quem chamou; efetua entrada na RC
leave_region:
        mov  flag, #0           ; guarda 0 em flag
        ret                     ; retorna
```

Mecanismos de Sincronização

Considerações Importantes

- Tanto a solução de Peterson e TSL são corretas, mas ambas tem o defeito de usar a espera ocupada em suas implementações.
- *Busy Waiting* além de consumir tempo de CPU pode causar problema de inversão de prioridade.
  - Pode acontecer de um processo de prioridade baixa entrar na RC e, em seguida, um processo de prioridade mais alta entra em execução e fica ocupando a CPU com a espera ocupada, impedindo o processo de prioridade baixa de sair da RC.

Mecanismos de Sincronização

Primitivas **Sleep** e **Wakeup**

- Esta solução consiste na utilização de duas primitivas para resolver o problema da espera ocupada.

- A primitiva ***Sleep*** é uma chamada de sistema que bloqueia o processo que chamou, isto é, suspende a execução do processo, até que outro processo o acorde

- A primitiva ***Wakeup*** acorda o processo que foi colocado para dormir.

Mecanismos de Sincronização

Primitivas **Sleep** e **Wakeup**

- Problema do Produtor e Consumidor:
  - Conhecido com o problema do buffer de tamanho reduzido;
  - Dois processos compartilham um buffer de tamanho fixo;
  - O produtor produz itens no buffer e o consumidor retira itens do buffer.
  - O produtor não pode produzir itens quando o buffer estiver cheio
  - O consumidor não pode consumir itens com buffer vazio

Mecanismos de Sincronização

Primitivas **Sleep** e **Wakeup**

```
#include "prototypes.h"

#define FALSE    0
#define TRUE     1
#define N        100            /* numero de posições no buffer */

int count = 0;                  /* numero de itens no buffer */

void producer(void)
{
  int item;

   while (TRUE) {               /* loop */
        produce_item(&item);    /* geração do próximo item */
        if (count == N) sleep(); /* se buffer cheio, processo vai dormir */
        enter_item(item);       /* colocação do item no buffer */
        count = count + 1;      /* incrementa a contagem do item no buffer */
        if (count == 1) wakeup(consumer); /* o buffer está vazio ? */
   }
}
```

Mecanismos de Sincronização

Primitivas **Sleep** e **Wakeup**

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

   while (TRUE) {              /* loop*/
        if (count == 0) sleep(); /* se o buffer vazio, processo vai dormir */
        remove_item(&item);      /* retirada de um item do buffer */
        count = count - 1;       /* decrementa a contagem de item no buffer */
        if (count == N-1) wakeup(producer); /* o buffer está cheio ? */
        consume_item(item);      /* consome o item */
   }
}
```

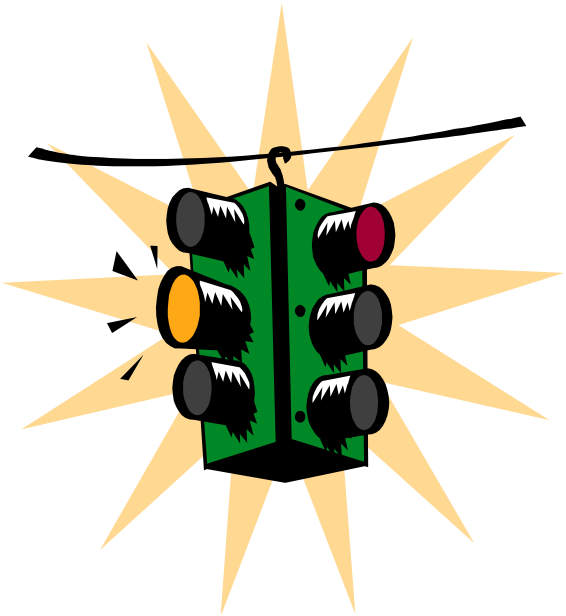
Mecanismos de Sincronização

Primitivas **Sleep** e **Wakeup**

- Problemas com a solução apresentada:
  - Acesso concorrente ao buffer: somente um processo pode estar manipulando o buffer para que não ocorra inconsistência na manutenção da quantidade dos elementos. Acesso a variável *count* deve ocorrer em RC;
  - Produtor pode dormir para sempre: basta que ocorra uma execução do consumidor entre o teste do buffer cheio e o *sleep*. Essas duas instruções devem ocorrer em um RC.
  - Consumidor pode dormir para sempre: basta que ocorra um execução do produtor entre o teste do buffer vazio e o *sleep*. A solução é agrupar essas instruções em um RC.

Mecanismos de Sincronização

Semáforos

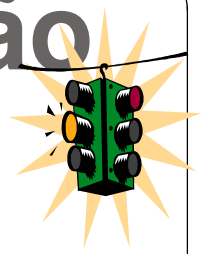


- Proposto pelo matemático holandês ***E. W. Dijkstra*** em 1965.

- Consiste de uma solução geral e simples de ser implementada, para os problemas de sincronização de processos concorrentes.

Mecanismos de Sincronização

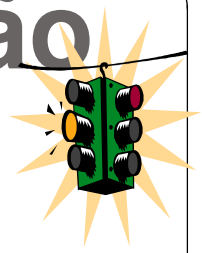
Semáforos



- Semáforo → tipo abstrato de dado composto por um valor inteiro e uma fila de processos
- Apenas duas operações são permitidas sobre semáforos:
  - **P** (do holandês *proberen*, testar) e **V** (do holandês *verhogen*, incrementar).

Mecanismos de Sincronização

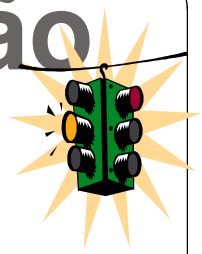
Semáforos



- Quando um processo executa a operação **P** sobre um semáforo, o seu valor inteiro é decrementado. Caso o novo valor do semáforo seja negativo, o processo é bloqueado e inserido no fim da fila desse semáforo.
- Quando um processo executa a operação **V** sobre um semáforo, o seu valor inteiro e incrementado. Caso exista algum processo bloqueado na fila desse semáforo, o primeiro processo da fila é liberado.

Mecanismos de Sincronização

Semáforos



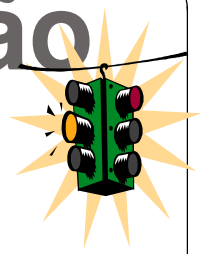
- Podemos sintetizar as operações com semáforos da seguinte forma:

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

V(S):
S.valor = S.valor+1;
Se S.valor >= 0 então retira P de S.fila, acorda P`

Mecanismos de Sincronização

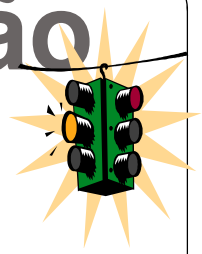
Semáforos



- Para funcionar corretamente, é essencial que as operações P e V sejam atômicas.
- Semáforos binários são aqueles que só podem ter os valores 0 e 1 (conhecido também como **mutex**)
- Semáforos contadores podem assumir qualquer valor inteiro não-negativo

Mecanismos de Sincronização

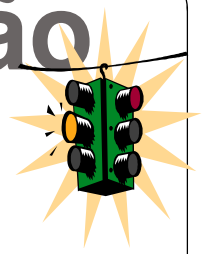
Semáforos



- Semáforos são:
  - Independentes de máquina
  - Simples de usar
  - Funcionam com muitos processos
  - Podem existir várias regiões críticas diferentes controladas por vários semáforos
  - Muitos recursos podem ser adquiridos simultaneamente
  - Permitem múltiplos processos numa região crítica de uma vez, se isso for desejável
  - Tipicamente, semáforos não são providos pelo hardware

Mecanismos de Sincronização

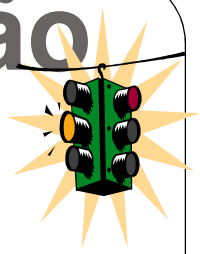
Semáforos



- Problema do Produtor e Consumidor usando semáforos.
  - Semáforos utilizados: *mutex* (exclusão mútua), *empty* e *full* (sincronização);
  - O semáforo *full* trava o consumidor quando o buffer está vazio;
  - O semáforo *empty* trava o produtor quando o buffer está cheio;

Mecanismos de Sincronização

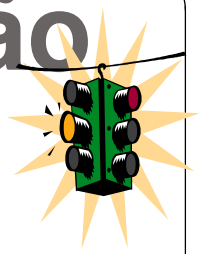
Semáforos



```
#include "prototypes.h"
#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) {
              produce_item(&item);     /* generate next item */
              down(&empty);            /* decrement empty count */
              down(&mutex); /* enter critical region */
              enter_item(item);        /* put item in buffer */
              up(&mutex);              /* leave critical region */
              up(&full);               /* increment count of full slots */
         }
}
```

Mecanismos de Sincronização

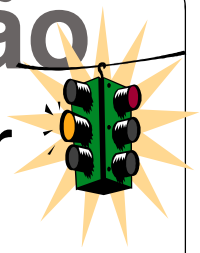
Semáforos



```
void consumer(void)
{
     int item;
     while (TRUE) {              /* repeat forever */
       down(&full);            /* decrement count of full slots */
       down(&mutex);           /* enter critical region */
       remove_item(&item);     /* take item from buffer */
       up(&mutex);             /* leave critical region */
       up(&empty);             /* increment count of full slots */
       consume_item(item);     /* use item */
     }
}
```

Mecanismos de Sincronização

Semáforos – Produtor e Consumidor



- Problema na seqüência de implementação dos semáforos:
  - Os downs do produtos foram invertidos:mutex foi diminuído antes de empty.
  - O buffer está cheio, então o produtor fica bloqueado (mutex = 0)
  - A próxima vez que o consumidor tentasse acessar o buffer, ele ficará bloqueado no down(mutex).

Os processos estão em DEADLOCK !!

Mecanismos de Sincronização

Monitores

- Esta solução foi proposta por *Hoare* (1974) e *Brinch Hansen* (1975) com o objetivo de facilitar a escrita de programas paralelos por meio do uso de primitivas de sincronização de alto nível chamada monitor
- Um monitor é um conjunto de procedimentos, estruturas de dados, todas agrupadas em um módulo especial. Os processos só podem acessar as estruturas de dados encapsuladas no monitor através de seus procedimentos

Mecanismos de Sincronização

Monitores

- Monitores são parecidos com o tipo abstrato de dados classes, mas com sincronização embutida no mecanismo.

- Os monitores têm um propriedade muito importante que os torna úteis na implementação da exclusão mútua: *Somente um processo pode está ativo dentro do monitor em um dado instante de tempo*.

Mecanismos de Sincronização

Monitores

- Os monitores são construções de linguagem de alto nível, portanto o compilador fica responsável em gerar o código com esta estrutura.
- Exemplos de linguagens que implementam o conceito de monitor : Mesa/Cedar (Xerox), Concurrent Euclid e Java.

Mecanismos de Sincronização

Monitores

- Exemplo:

```
monitor  exemplo
            integer i;
            condition c;

procedure  produtor(x)
            .
            .
end

procedure consumidor(x)
            .
            .
end
end monitor
```

Mecanismos de Sincronização
Monitores

- Propriedades Importantes de Monitores:
  - Apenas um processo pode executar o código do monitor a cada instante de tempo;
  - Exclusão mútua implícita é associada a cada monitor, por meio de um monitor *lock*
  - O *lock* deve ser adquirido para um processo "entrar"no monitor
  - Ao sair do monitor, o *lock* é liberado

Mecanismos de Sincronização

Monitores

- Produtor e Consumidor: é necessário bloquear os processo quando eles não podem prosseguir.
- Solução: variáveis de condição.
  - **wait**(condição) : causa o bloqueio do processo de chamada
  - **signal**(condição) : acorda um processo previamente bloqueado

Mecanismos de Sincronização

Monitores

```
monitor ProdutorConsumidor

  condition full, empty;
  int count = 0;
  int item;

  void enter(void);
  {
    if (count == N) { wait(full); }
    enter_item(item);
    count++;
    if (count == 1) { signal(empty); }
  }
```

```
void remove(void);
  {
    if (count == 0) { wait(empty); }
    remove_item(&item);
    count--;
    if (count == N-1) { signal(full); }
  }

end monitor;
```

Mecanismos de Sincronização

Monitores

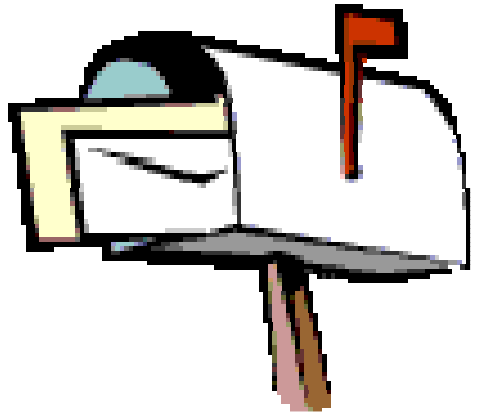
```
void producer(void);
{
  while (TRUE) {
    produce_item(&item);
    ProducerConsumer.enter();
  }
}

void consumer(void);
{
  while (TRUE) {
    ProducerConsumer.remove();
    consume_item(item);
  }
}
```

Mecanismos de Sincronização

Monitores X Semáforos

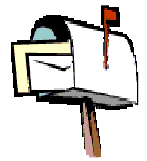
- Monitor é um conceito de alto nível (linguagem de programação) e semáforos de baixo nível (chamadas de SO);
- Monitor é mais fácil de utilizar, mas é necessário o suporte da LP ou Compilador;
- Semáforos estão disponíveis em vários SOs.
- Monitores forçam o programador a isolar sincronizações complexas em módulos especiais

Comunicação Interprocessos
Troca de Mensagens



- Até agora os seguintes mecanismos para comunicação e cooperação entre processos foram apresentados:
  - Memória Compartilhada
  - Semáforos
  - Monitores (locks e variáveis de condição)

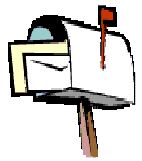
Comunicação Interprocessos
Troca de Mensagens



- Semáforos, memória compartilhada e monitores são mecanismos de sincronização, que normalmente assumem a existência de memória física compartilhada (eficiência !!).

- Existem outras classes mais gerais de abstrações para comunicação entre processos:
  - Troca de Mensagens
  - PIPES (comuns a todas implementações de UNIX)
  - Chamadas remotas de procedimentos (Remote Procedure Calls - RPC) : existente em muitos sistemas cliente-servidor.

Comunicação Interprocessos
Troca de Mensagens

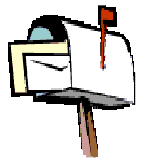


- Este tipo de sistema permite a comunicação/sincronização de processos sem utilizar dados compartilhados.

- Mensagem : um item de informação que é passado de um processo para outro processo

- Este método de comunicação utiliza duas primitivas para realizar a comunicação entre os processos : SEND e RECEIVE.

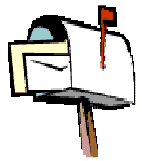
Comunicação Interprocessos
Troca de Mensagens



- Estas primitivas são disponibilizadas através de chamadas de sistema,com o seguinte formato :

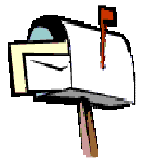
```
send(destino,&mensagem);
receive(fonte,&mensagem);
```

Comunicação Interprocessos
Troca de Mensagens



- A primitiva *send* envia uma mensagem para um certo destino, e a primitiva *receive* recebe uma mensagem de uma determinada fonte.
- Se não houver nenhuma mensagem a ser recebida, o receptor pode ser bloqueado até que uma mensagem chegue.
- Aspecto importante sobre sistemas baseados em mensagens: **perda da mensagem**.

Comunicação Interprocessos
Troca de Mensagens

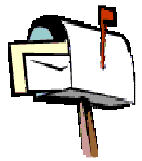


- Podemos ter problemas na comunicação de processos que estiverem em máquinas diferente, ou seja, conectadas por um rede.

As mensagens podem ser perdidas na rede.

- Para prevenir a perda de mensagens, o transmissor deve ser programados de forma que, tão logo tenha recebido uma mensagem, o receptor envie de volta ao transmissor uma mensagem especial de reconhecimento (acknowledgement - ack).

Comunicação Interprocessos
Troca de Mensagens

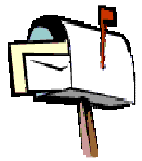


- Se o transmissor não receber esta mensagem num certo intervalo, ele deve retransmitir a mensagem.

- Outro problema que pode ocorrer é a mensagem chegar no receptor, mas o reconhecimento for perdido.

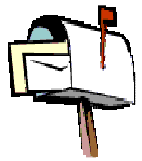
  - Neste caso, o transmissor transmite de novo a mensagem, e o receptor a recebe duas vezes. Pode-se resolver este problema colocando números seqüenciais em cada mensagem original.
  - Se o receptor receber uma mensagem com o mesmo número da anterior, ele sabe que tal mensagem é duplicada e que pode ser ignorada.

Comunicação Interprocessos
Troca de Mensagens



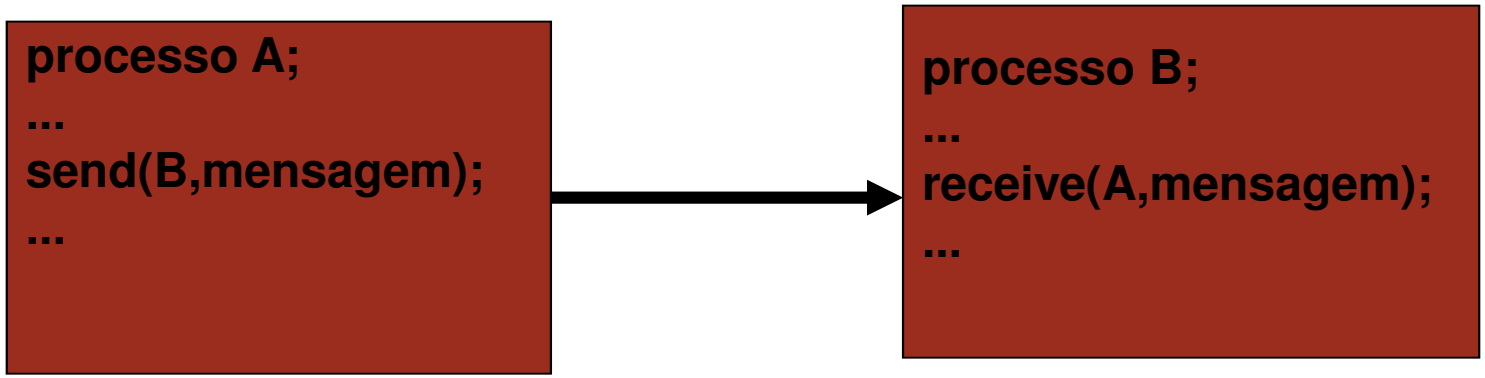
- Temos outros problemas as serem considerados, como autenticação das mensagens, criptografia das mensagens, etc….
- Mensagens é um mecanismo mais adequado para comunicação e sincronização entre processos, tanto em sistemas centralizados como em sistemas distribuídos. Uma mensagem pode conter dados ou comandos de execução, ou mesmo código a ser transmitido entre dois ou mais processos.

Comunicação Interprocessos
Troca de Mensagens

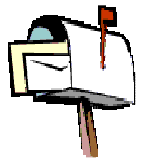


Considerações Importantes

- O processo pode se comunicar com o receptor de forma direta ou indireta.
- Direta : O nome é fornecido de forma direta



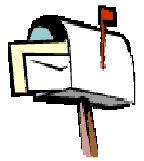
Comunicação Interprocessos
Troca de Mensagens



Considerações Importantes

- As mensagens podem ser passadas por valor ou por referencia
- A mensagem passada por valor é copiada diretamente no espaço de endereçamento do destinatário (cópia).
- A mensagem passada por referência, o remetente envia o ponteiro (endereço) da mensagem para o destinatário.

Comunicação Interprocessos
Troca de Mensagens

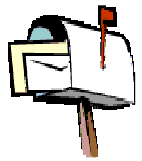


Considerações Importantes

- Problemas na passagem de mensagens por referência :
  - Impossível em sistemas distribuídos
  - Eficiente mas não seguro

- Inconvenientes da passagem por valor :
  - cópia da mensagem consome tempo de CPU e memória deve-se minimizar o número de cópias

Comunicação Interprocessos
Troca de Mensagens

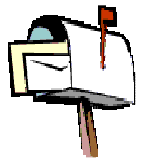


Considerações Importantes

- Independente da forma de endereçamento entre os processos, a comunicação entre eles pode bloquear (operações blocantes) ou não os processo envolvidos (operações não-blocantes).

- Basicamente, existem duas formas de comunicação entre processos através de troca de mensagens: comunicação síncrona ou assíncrona.

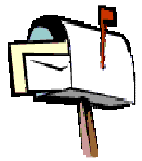
Comunicação Interprocessos
Troca de Mensagens



Comunicação Síncrona

- A comunicação é dita síncrona, quando um processo envia uma mensagem (send) e fica esperando, até que o processo receptor leia a mensagem, ou quando um processo tenta receber uma mensagem (receive) e fica esperando, até que o processo transmissor grave alguma mensagem.
  - Este tipo de comunicação dispensa a necessidade de buffer; porém, a execução dos processo fica limitada ao tempo de processamento das mensagens.

Comunicação Interprocessos
Troca de Mensagens

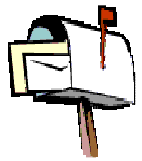


Comunicação Síncrona

- Este mecanismo também é conhecido como ***rendezvous***.

- Vantagens :
  - baixo "overhead"
  - facilidade de implementação
  - remetente sabe que a mensagem foi recebida pelo destinatário
- Inconveniências:
  - operação síncrona
  - indesejável em processos de tempo real, servidores, …

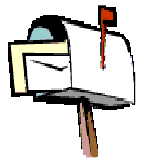
Comunicação Interprocessos
Troca de Mensagens



Comunicação Assíncrona

- Na comunicação assíncrona, nem receptor permanece aguardando o envio de um mensagem, nem o transmissor o seu recebimento.
- Neste caso, além da necessidade de buffer (ex: *mailboxes*) para armazenar as mensagens, deve haver outros mecanismos de sincronização que permitam ao processo identificar se uma mensagem já foi enviada ou recebida.

Comunicação Interprocessos
Troca de Mensagens

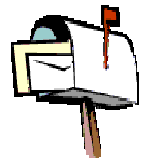


Comunicação Assíncrona

- Vantagem :
  - maior paralelismo na execução do processos = aumenta o grau de concorrência do sistema.

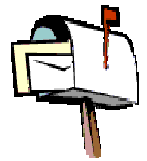
- Problema do Produto e Consumidor utilizando troca de mensagens

Comunicação Interprocessos
Troca de Mensagens



```
#define N 100          /* number of slots in the buffer */
#define MSIZE 4        /* message size */

typedef int message[MSIZE];

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

  while(TRUE) {
    produce_item(&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 */
  }
}
```

Comunicação Interprocessos
Troca de Mensagens



```
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 */
    extract_item(&m, &item); /* take item out of message */
    send(producer, &m);      /* send back empty reply: ACK */
    consume_item(item);      /* use item */
  }
}
```

Condição de Corrida

Problema de Compartilhamento de Recursos

```
READ(Arq_Contas, Reg_Cliente);
READLN(Valor_Dep_Ret);
Reg_Cliente.Saldo = Reg_Cliente.Saldo-Valor_Dep_Ret;
WRITE(Arq_Contas, Reg_Cliente);
```

Condição de Corrida

Problema de Compartilhamento de Recursos

CAIXA	Comando	Saldo ARQ	Valor Dep/Ret	Saldo Memória
1	READ	4.500	?	4.500
1	READLN	4.500	- 4.000	4.500
1	=	4.500	-4.000	500
2	READ	4.500	?	4.500
2	READLN	4.500	+ 3.000	4.500
2	=	4.500	+3.000	7.500
1	WRITE	500	-4.000	500
2	WRITE	7.500	+3.000	7.500