Aula 08
Sistemas Operacionais I

Comunicação e Sincronismo de Processos

– Parte 02

Prof. Julio Cezar Estrella
jcezar@icmc.usp.br
Material adaptado de
Sarita Mazzini Bruschi

Processos

Introdução

Escalonamento de Processos

- **Comunicação entre Processos**
  - Condição de Disputa
  - Região Crítica
  - **Formas de Exclusão Mútua**
  - Problemas Clássicos

Threads

Deadlock

Soluções

Exclusão Mútua:

Espera Ocupada;

- Desabilitar Interrupções
- Primitiva *Lock*
- Alternância Estrita
- Solução de Peterson e Instrução TSL

Primitivas *Sleep/Wakeup*;

Semáforos;

Monitores;

Passagem de Mensagem;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

Todas as soluções apresentadas utilizam espera ocupada → processos ficam em estado de espera (*looping)* até que possam utilizar a região crítica:

- Tempo de processamento da CPU;
- Situações inesperadas:
  - Processos H (alta prioridade) e L (baixa prioridade)
  - L entra na RC
  - H começa a executar e quer entrar na RC

Comunicação de Processos – Primitivas *Sleep/Wakeup*

Para solucionar esse problema de espera, um par de primitivas *Sleep* e *Wakeup* é utilizado → BLOQUEIO E DESBLOQUEIO de processos.

A primitiva *Sleep* é uma chamada de sistema que bloqueia o processo que a chamou, ou seja, suspende a execução de tal processo até que outro processo o "acorde";

A primitiva *Wakeup* é uma chamada de sistema que "acorda" um determinado processo;

Ambas as primitivas possuem dois parâmetros: o processo sendo manipulado e um endereço de memória para realizar a correspondência entre uma primitiva *Sleep* com sua correspondente *Wakeup*;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

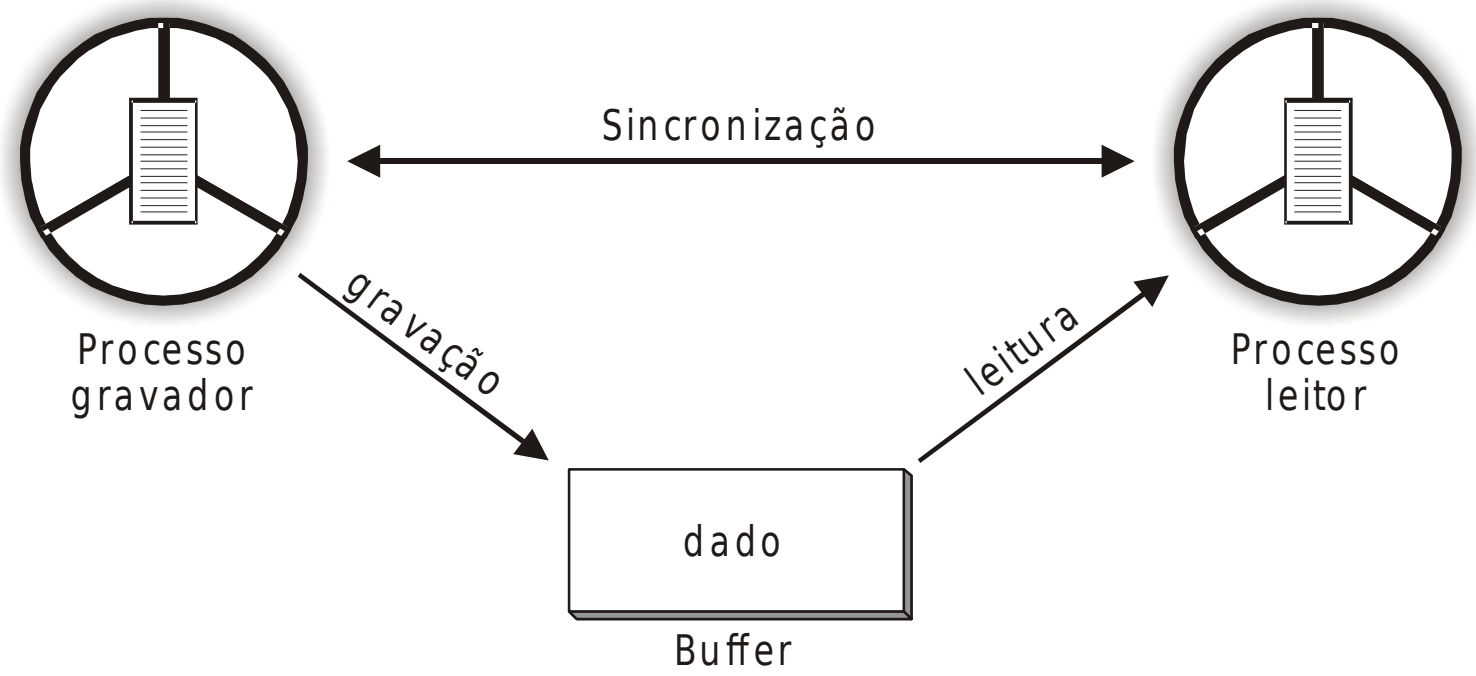
Problemas que podem ser solucionados com o uso dessas primitivas:

Problema do Produtor/Consumidor (*bounded buffer* ou *buffer* limitado): dois processos compartilham um *buffer* de tamanho fixo. O processo produtor coloca dados no *buffer* e o processo consumidor retira dados do *buffer*;

Problemas:

- Produtor deseja colocar dados quando o *buffer* ainda está cheio;
- Consumidor deseja retirar dados quando o *buffer* está vazio;
- Solução: colocar os processos para “dormir”, até que eles possam ser executados;

Comunicação de Processos
Sincronização Produtor-Consumidor



Comunicação de Processos – Primitivas *Sleep*/*Wakeup*

Buffer: variável `count` controla a quantidade de dados presente no *buffer*.

Produtor:

Se `count` == valor máximo (buffer cheio)

Então

processo produtor é colocado para dormir

Produtor coloca dados no *buffer* e incrementa `count`

Comunicação de Processos – Primitivas *Sleep/Wakeup*

Consumidor:

Se count == 0 (buffer vazio)

Então

processo vai “dormir”

Retira os dados do *buffer* e decrementa count

Comunicação de Processos – Primitivas *Sleep/Wakeup*

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

void producer(void)
{
  int item;

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

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

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

Comunicação de Processos – Primitivas *Sleep/Wakeup*

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

void producer(void)
{
  int item;

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

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

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

Para aqui

Problema: se *wakeup* chega antes do consumidor dormir

Comunicação de Processos – Primitivas *Sleep/Wakeup*

Problemas desta solução: o acesso à variável count é irrestrito

O *buffer* está vazio e o consumidor acabou de checar a variável count com valor 0;

O escalonador (por meio de uma interrupção) decide que o processo produtor será executado; Então o processo produtor insere um item no *buffer* e incrementa a variável count com valor 1; Imaginando que o processo consumidor está dormindo, o processo produtor envia um sinal de *wakeup* para o consumidor;

No entanto, o processo consumidor não está dormindo, e não recebe o sinal de *wakeup*;

Assim que o processo consumidor é executado novamente, a variável count já tem o valor zero; Nesse instante, o consumidor é colocado para dormir, pois acha que não existem informações a serem lidas no *buffer*;

Assim que o processo produtor acordar, ele insere outro item no *buffer e* volta a dormir. Ambos os processos dormem para sempre...

Comunicação de Processos – Primitivas *Sleep/Wakeup*

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

Soluções

Exclusão Mútua:

Espera Ocupada;

Primitivas *Sleep/Wakeup*;

Semáforos;

Monitores;

Passagem de Mensagem;

Comunicação de Processos – Semáforos

Idealizados por E. W. Dijkstra (1965);

Variável inteira que armazena o número de sinais *wakeups* enviados;

Um semáforo pode ter valor 0 quando não há sinal armazenado ou um valor positivo referente ao número de sinais armazenados;

Duas primitivas de chamadas de sistema: *down (sleep)* e *up (wake);*

Originalmente P (*down*) e V (*up*) em holandês*;*

Comunicação de Processos – Semáforos

Down: verifica se o valor do semáforo é maior do que 0; se for, o semáforo é decrementado; se o valor for 0, o processo é colocado para dormir sem completar sua operação de ***down***;

Todas essas ações são chamadas de **açõ**es atômicas;

Ações atômicas garantem que quando uma operação no semáforo está sendo executada, nenhum processo pode acessar o semáforo até que a operação seja finalizada ou bloqueada;

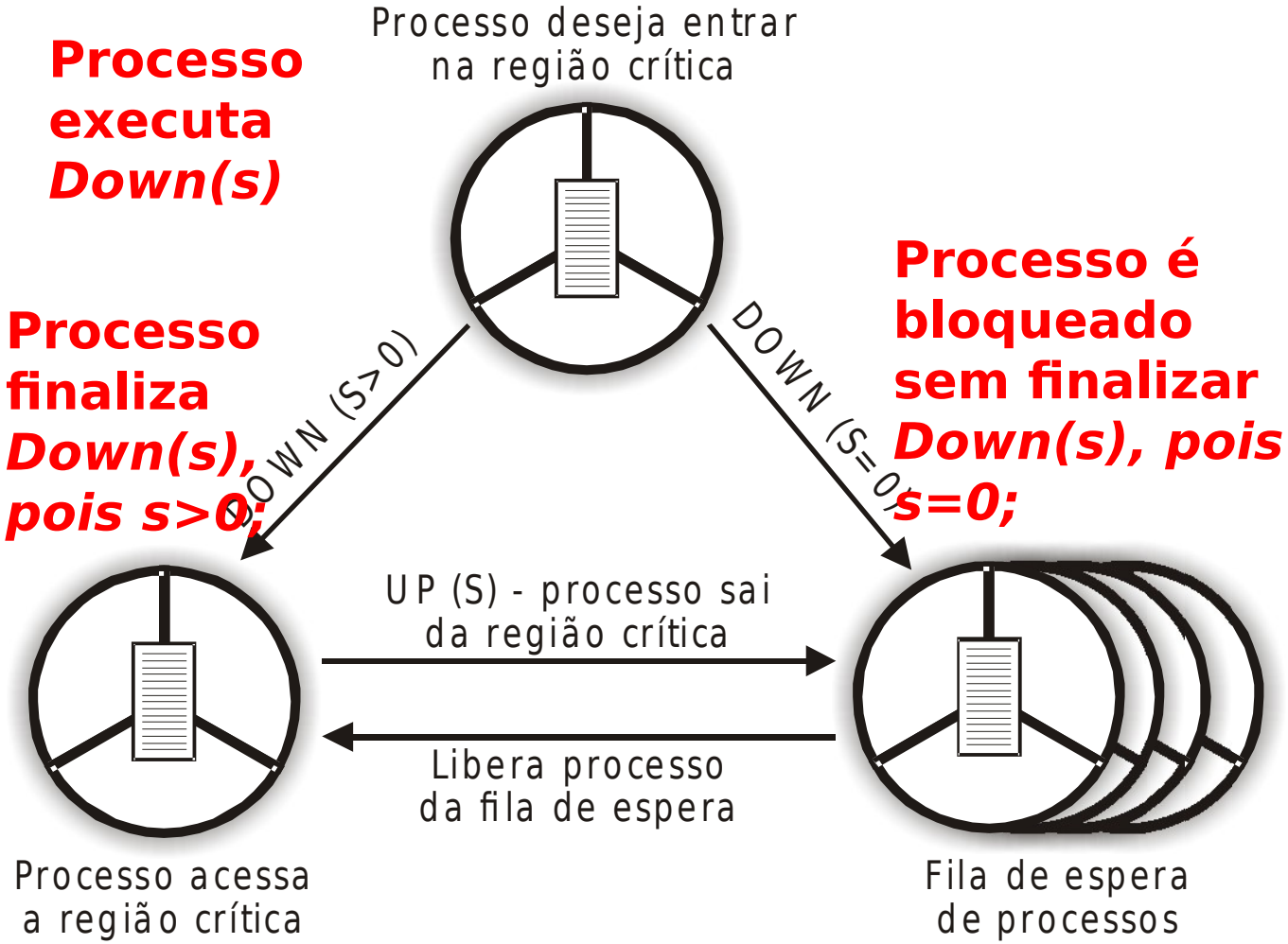
Comunicação de Processos – Semáforos

Up: incrementa o valor do semáforo, fazendo com que algum processo que esteja dormindo possa terminar de executar sua operação ***down***;

Semáforo Mutex: garante a exclusão mútua, não permitindo que os processos acessem uma região crítica ao mesmo tempo

Também chamado de **semáforo binário**

Comunicação de Processos – Semáforo Binário



Comunicação de Processos – Semáforos

Problema produtor/consumidor: resolve o problema de perda de sinais enviados;

Solução utiliza três semáforos:

Full: conta o número de *slots* no *buffer* que estão ocupados; iniciado com 0; resolve sincronização;

Empty: conta o número de *slots* no *buffer* que estão vazios; iniciado com o número total de *slots* no *buffer*; resolve sincronização;

Mutex: garante que os processos produtor e consumidor não acessem o *buffer* ao mesmo tempo; iniciado com 1; também chamado de **semáforo binário**; Permite a exclusão mútua;

Comunicação de Processos – Semáforos

```
# include “prototypes.h”
# define N 100

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

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

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

Comunicação de Processos – Semáforos

Problema: erro de programação pode gerar um *deadlock*;

Suponha que o código seja trocado no processo produtor;

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

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

Soluções

Exclusão Mútua:

Espera Ocupada;

Primitivas *Sleep/Wakeup*;

Semáforos;

Monitores;

Passagem de Mensagem;

Comunicação de Processos – Monitores

Idealizado por Hoare (1974) e Brinch Hansen (1975)

Monitor: primitiva (unidade básica de sincronização) de alto nível para sincronizar processos:

Conjunto de procedimentos, variáveis e estruturas de dados agrupados em um único módulo ou pacote;

Somente um processo pode estar ativo dentro do monitor em um mesmo instante; outros processos ficam bloqueados até que possam estar ativos no monitor;

Comunicação de Processos – Monitores

```
monitor example
   int i;
   condition c;

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

Estrutura básica de um Monitor

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

- JAVA

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

Comunicação de Processos – Monitores

Execução:

Chamada a uma rotina do monitor;

Instruções iniciais → teste para detectar se um outro processo está ativo dentro do monitor;

Se positivo, o processo novo ficará bloqueado até que o outro processo deixe o monitor;

Caso contrário, o processo novo executa as rotinas no **monitor**;

Comunicação de Processos – Monitores

Condition Variables (condition): variáveis que indicam uma condição; e

Operações Básicas: *WAIT e SIGNAL*

wait (condition) → bloqueia o processo;

signal (condition) → "acorda" o processo que executou um *wait* na variável *condition* e foi bloqueado;

Comunicação de Processos – Monitores

Variáveis condicionais não são contadores, portanto, não acumulam sinais;

Se um sinal é enviado sem ninguém (processo) estar esperando, o sinal é perdido;

Assim, um comando `WAIT` deve vir antes de um comando `SIGNAL`.

Comunicação de Processos – Monitores

Como evitar dois processos ativos no monitor ao mesmo tempo?

(1) Hoare → colocar o processo mais recente para rodar, suspendendo o outro!!! (*sinalizar e esperar*)

(2) B. Hansen → um processo que executa um `SIGNAL` deve deixar o monitor imediatamente;

- O comando `SIGNAL` deve ser o último de um procedimento do monitor;

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

Comunicação de Processos – Monitores

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Comunicação de Processos – Monitores

Se parar aqui

Consumidor não pode entrar no 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;
```

Comunicação de Processos – Monitores

Devido a exclusão mútua automática dos procedimentos do monitor, tem-se:

o produtor dentro de um procedimento do monitor descobre que o buffer está cheio

produtor termina a operação de `WAIT` sem se preocupar

consumidor só entrará no monitor após produtor dormir

Comunicação de Processos – Monitores

Limitações de semáforos e monitores:

Ambos são boas soluções somente para CPUs com memória compartilhada. Não são boas soluções para sistema distribuídos;

Nenhuma das soluções provê troca de informações entre processo que estão em diferentes máquinas;

Monitores dependem de uma linguagem de programação – poucas linguagens suportam Monitores;

Soluções

Exclusão Mútua:

Espera Ocupada;

Primitivas *Sleep/Wakeup*;

Semáforos;

Monitores;

Passagem de Mensagem;

Comunicação de Processos – Passagem de Mensagem

Provê troca de mensagens entre processos rodando em máquinas diferentes;

Utiliza-se de duas primitivas de chamadas de sistema: *send* e *receive;*

Comunicação de Processos – Passagem de Mensagem

Podem ser implementadas como procedimentos:

```
send (destination,&message);
```

```
receive (source,&message);
```

O procedimento `send` envia para um determinado destino uma mensagem, enquanto que o procedimento `receive` recebe essa mensagem em uma determinada fonte; Se nenhuma mensagem está disponível, o procedimento `receive` é bloqueado até que uma mensagem chegue.

Comunicação de Processos – Passagem de Mensagem

Problemas desta solução:

Mensagens são enviadas para/por máquinas conectadas em rede; assim mensagens podem se perder ao longo da transmissão;

Mensagem especial chamada ***acknowledgement*** → o procedimento `receive` envia um ***acknowledgement*** para o procedimento `send`. Se esse ***acknowledgement*** não chega no procedimento `send`, esse procedimento retransmite a mensagem já enviada;

Comunicação de Processos – Passagem de Mensagem

Problemas:

A mensagem é recebida corretamente, mas o ***acknowledgement*** se perde.

Então o `receive` deve ter uma maneira de saber se uma mensagem recebida é uma retransmissão → cada mensagem enviada pelo `send` possui uma identificação – seqüência de números; Assim, ao receber uma nova mensagem, o `receive` verifica essa identificação, se ela for semelhante a de alguma mensagem já recebida, o `receive` descarta a mensagem!

Comunicação de Processos – Passagem de Mensagem

Problemas:

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

Autenticação → Segurança;

Comunicação de Processos – Passagem de Mensagem

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

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

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

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

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

Comunicação de Processos
Outros mecanismos

RPC – *Remote Procedure Call*

Rotinas que permitem comunicação de processos em diferentes máquinas;

Chamadas remotas;

MPI – *Message-passing Interface*;

Sistemas paralelos;

RMI Java – *Remote Method Invocation*

Permite que um objeto ativo em uma máquina virtual Java possa interagir com objetos de outras máquinas virtuais Java, independentemente da localização dessas máquinas virtuais;

Web Services

Permite que serviços sejam compartilhados através da Web

Comunicação de Processos
Outros mecanismos

Pipe:

Permite a criação de filas de processos;

`ps -ef | grep <nome_usuário>`;

Saída de um processo é a entrada de outro;

Existe enquanto o processo existir;

Named pipe:

Extensão de pipe;

Continua existindo mesmo depois que o processo terminar;

Criado com chamadas de sistemas;

Socket:

Par endereço IP e porta utilizado para comunicação entre processos em máquinas diferentes;

Host X (192.168.1.1:1065) Server Y (192.168.1.2:80);