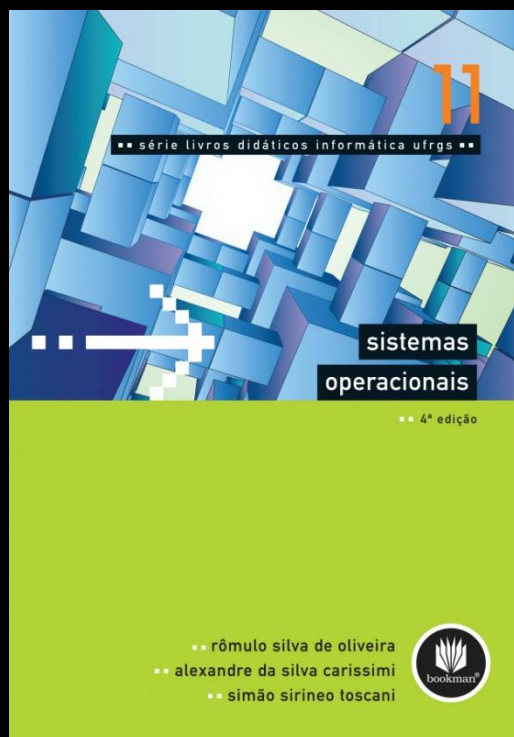


■ ■ série de livros didáticos informática ufrgs



**bookman®**  
EMPRESA DO GRUPO ARTMED  
[www.bookman.com.br](http://www.bookman.com.br)

**.inf**  
INSTITUTO  
DE INFORMÁTICA  
UFRGS



## Sistemas Operacionais

Rômulo Silva de Oliveira  
Alexandre da Silva Carissimi  
Simão Sirineo Toscani

# Agenda

- Fundamentos de programação concorrente
- Problemas da concorrência
- Região Crítica
- Semáforos, mutex e locks
- Remote Procedure Call - RPC
- Deadlocks

# Introdução

- Programa executado por apenas um processo é dito de programa sequencial
  - Existe apenas um fluxo de controle
- Programa concorrente é executado por diversos processos que cooperam entre si para realização de uma tarefa (aplicação)
  - Existem vários fluxos de controle
  - Necessidade de interação para troca de informações (sincronização)
- Emprego de termos
  - Paralelismo real: só ocorre em máquinas multiprocessadoras
  - Paralelismo “aparente” (concorrência): máquinas monoprocessadoras
  - Execução simultânea versus estar “em estado de execução” simultaneamente

# Programação concorrente

- Composta por um conjunto de processos sequenciais que se executam concorrentemente
- Processos disputam recursos comuns
  - e.g.: variáveis, periféricos, etc...
- Um processo é dito de cooperante quando é capaz de afetar, ou ser afetado, pela execução de outro processo

# Motivação para programação concorrente

- Aumento de desempenho:
  - Permite a exploração do paralelismo real disponível em máquinas multiprocessadoras
  - Sobreposição de operações de E/S com processamento
- Facilidade de desenvolvimento de aplicações que possuem um paralelismo intrínseco

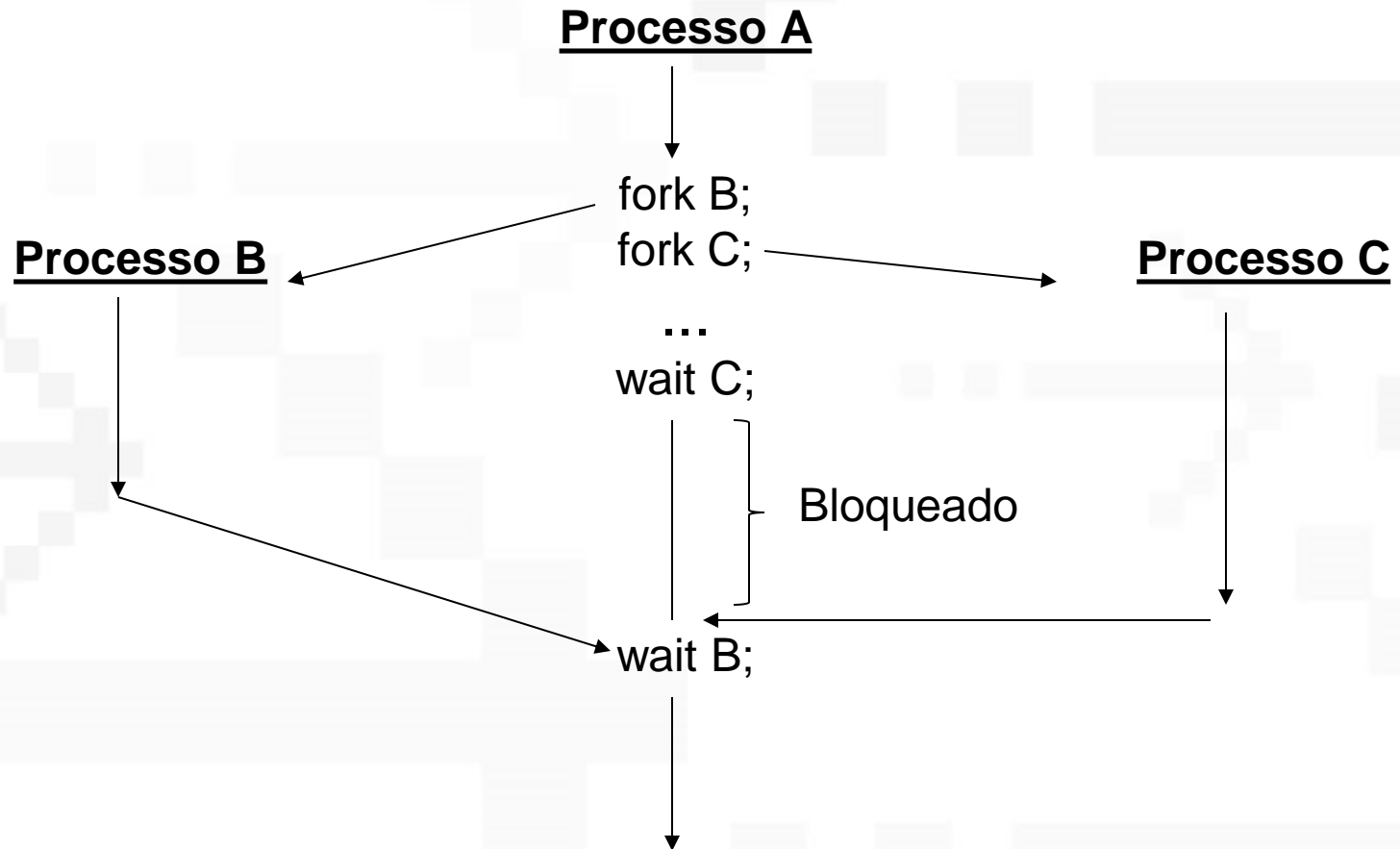
# Desvantagens da programação concorrente

- Programação complexa
- Aos erros “comuns” se adicionam erros próprios ao modelo
  - Diferenças de velocidade relativas de execução dos processos
- Aspecto não determinístico
  - Difícil depuração

# Especificação da paralelismo

- Necessidade de especificar o paralelismo definindo:
  - Quantos processos participarão
  - Quem fará o que
  - Dependência entre as tarefas (Grafo)
- Notação para expressar paralelismo
  - *fork/wait (fork/join)*
  - *parbegin/parend*

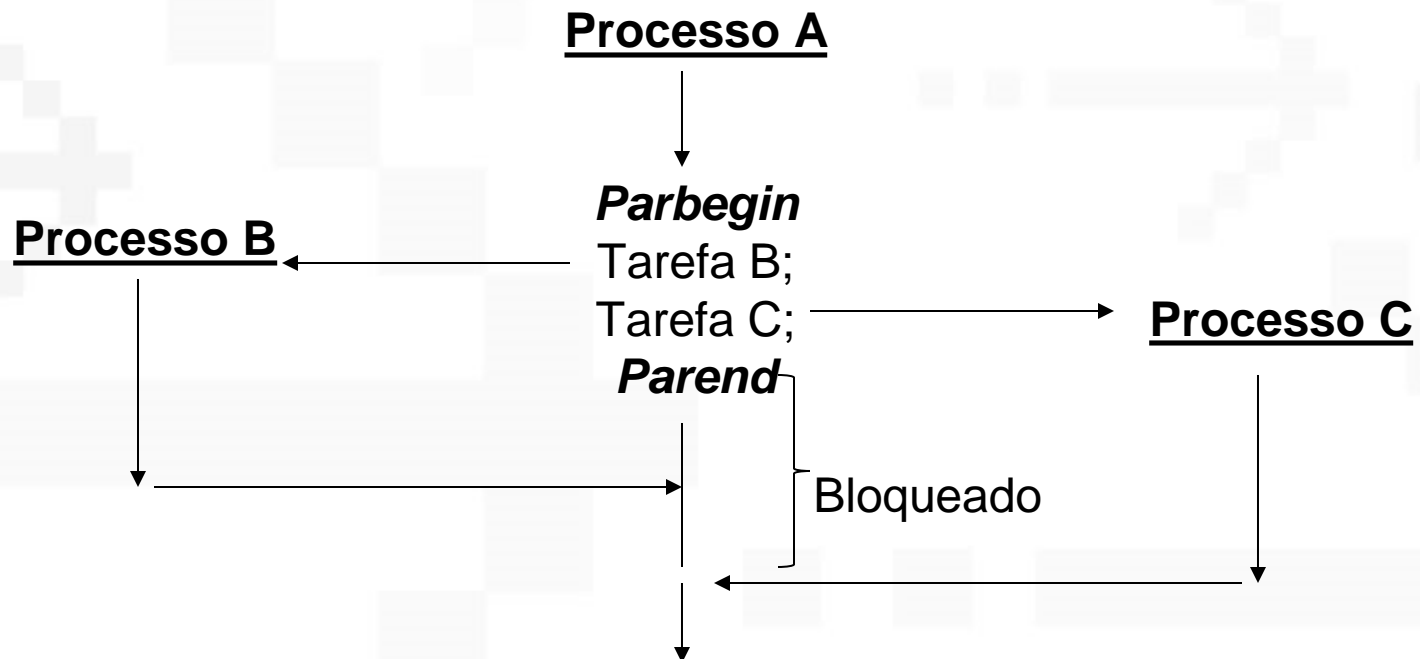
# Fork/wait





# *Parbegin/parend*

- Comandos empregados para definir uma sequência de comandos a serem executados concorrentemente
- A primitiva *parend* funciona como um ponto de sincronização (barreira)



# Comentários gerais

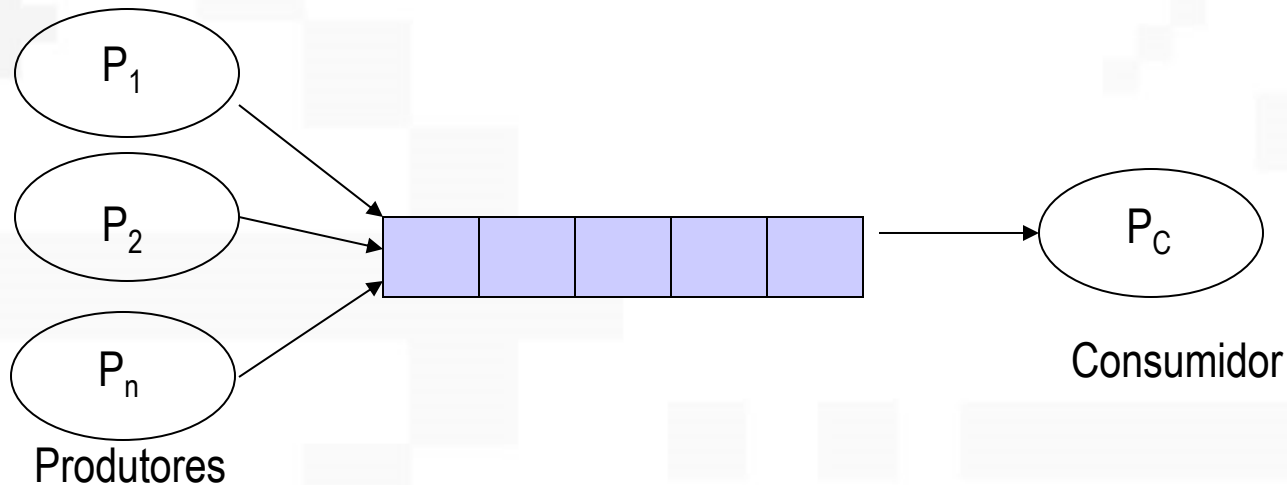
- Primitivas de mais alto nível para descrição do paralelismo do tipo parbegin/parend podem ser traduzidas por pré-compiladores e/ou interpretadores para primitivas de mais baixo nível
- Processos paralelos podem ser executados em qualquer ordem
  - Duas execuções consecutivas do mesmo programa, com os mesmos dados de entrada, podem gerar resultados diferentes
    - Não é necessariamente um erro
  - Possibilidade de forçar a execução em uma determinada ordem

# O problema do compartilhamento de recursos

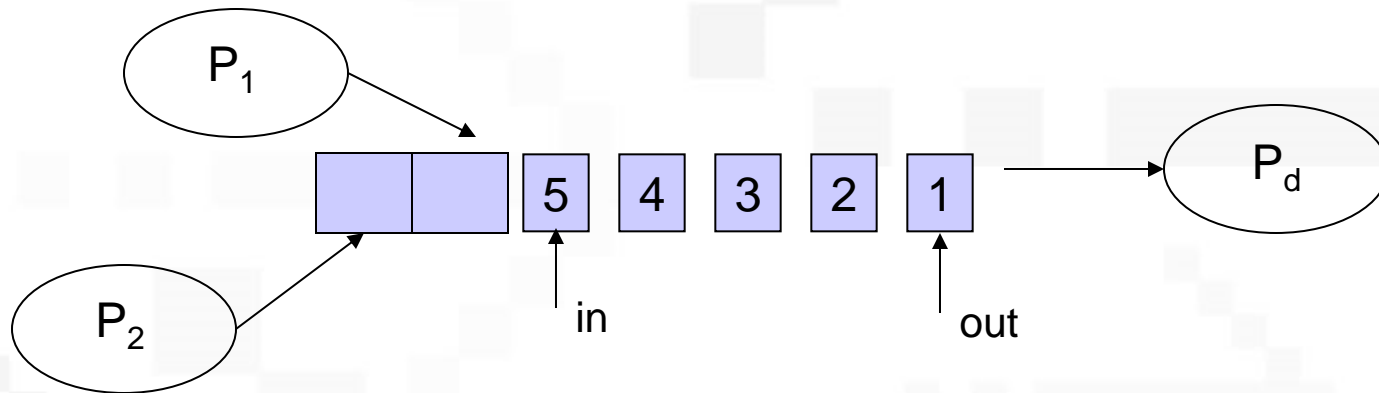
- A programação concorrente implica em um compartilhamento de recursos
  - Variáveis compartilhadas são recursos essenciais para a programação concorrente
- Acessos a recursos compartilhados devem ser feitos de forma a manter um estado coerente e correto do sistema

# Exemplo: relação produtor-consumidor (1)

- Relação produtor-consumidor é uma situação bastante comum em sistemas operacionais
- Servidor de impressão:
  - Processos usuários produzem “impressões”
  - Impressões são organizadas em uma fila a partir da qual um processo (consumidor) os lê e envia para a impressora



# Exemplo: relação produtor-consumidor (2)



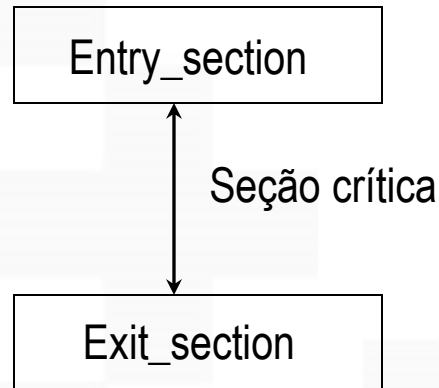
- Suposições:
  - Fila de impressão é um *buffer* circular
  - Existência de um ponteiro (*in*) que aponta para uma posição onde a impressão é inserida para aguardar o momento de ser efetivamente impressa
  - Existência de um ponteiro (*out*) que aponta para a impressão que está sendo realizada

# O problema da seção crítica

- Corrida (*race condition*)
  - Situação que ocorre quando vários processos manipulam o mesmo conjunto de dados concorrentemente e o resultado depende da ordem em que os acessos são feitos
- Seção crítica:
  - Segmento de código no qual um processo realiza a alteração de um recurso compartilhado

# Necessidade da programação concorrente

- Eliminar corridas
- Criação de um protocolo para permitir que processos possam cooperar sem afetar a consistência dos dados
- Controle de acesso a seção crítica:
  - Garantir a exclusão mútua



# Propriedades para exclusão mútua

- Regra 1 - Exclusão mútua
  - Dois ou mais processos não podem estar simultaneamente em uma seção crítica
- Regra 2 - Progressão
  - Nenhum processo fora da seção crítica pode bloquear a execução de um outro processo
- Regra 3 - Espera limitada
  - Nenhum processo deve esperar infinitamente para entrar em uma seção crítica
- Regra 4
  - Não fazer considerações sobre o número de processadores, nem de suas velocidades relativas



# Obtenção da exclusão mútua

- Desabilitação de interrupções
- Variáveis especiais do tipo *lock*
- Alternância de execução

# Desabilitação de interrupções

- Não há troca de processos com a ocorrência de interrupções de tempo ou de eventos externos
- Desvantagens:
  - Poder demais para um usuário
  - Não funciona em máquinas multiprocessadoras (SMP) pois apenas a CPU que realiza a instrução é afetada (violação da regra 4)

Seção crítica { CLI ;Desliga interrupções  
STI ;Ativa interrupções

# Variáveis do tipo *lock*

- Criação de uma variável especial compartilhada que armazena dois estados:
  - Zero: livre
  - 1: ocupado
- Desvantagem:
  - Apresenta *Race conditions*

Seção crítica

```
While (lock==1);  
lock=1;  
  
lock=0;
```

# Alternância

- Desvantagem
  - Teste contínuo do valor da variável compartilhada provoca o desperdício do tempo do processador (*busy waiting*)
  - Viola a regra 2 se a parte não crítica de um processo for muito maior que a do outro

```
while (TRUE) {  
    while (turn!=0);  
        critical_section();  
    turn=1;  
    non_critical_section();  
}
```

```
while (TRUE) {  
    while (turn!=1);  
        critical_section();  
    turn=0;  
    non_critical_section();  
}
```

# Implementação de mecanismos para exclusão mútua

- Algorítmica:
  - Combinação de variáveis do tipo *lock* e alternância (Dekker 1965, Peterson 1981)
- Primitivas:
  - Mutex
  - Semáforos
  - Monitor

# Mutex

- Variável compartilhada para controle de acesso a seção crítica
- CPU são projetadas levando-se em conta a possibilidade do uso de múltiplos processos
- Inclusão de duas instruções *assembly* para leitura e escrita de posições de memória de forma atômica.
  - CAS: *Compare and Store*
    - Copia o valor de uma posição de memória para um registrador interno e escreve nela o valor 1
  - TSL: *Test and Set Lock*
    - Lê o valor de uma posição de memória e coloca nela um valor não zero

# Primitivas *lock* e *unlock*

- O emprego de mutex necessita duas primitivas

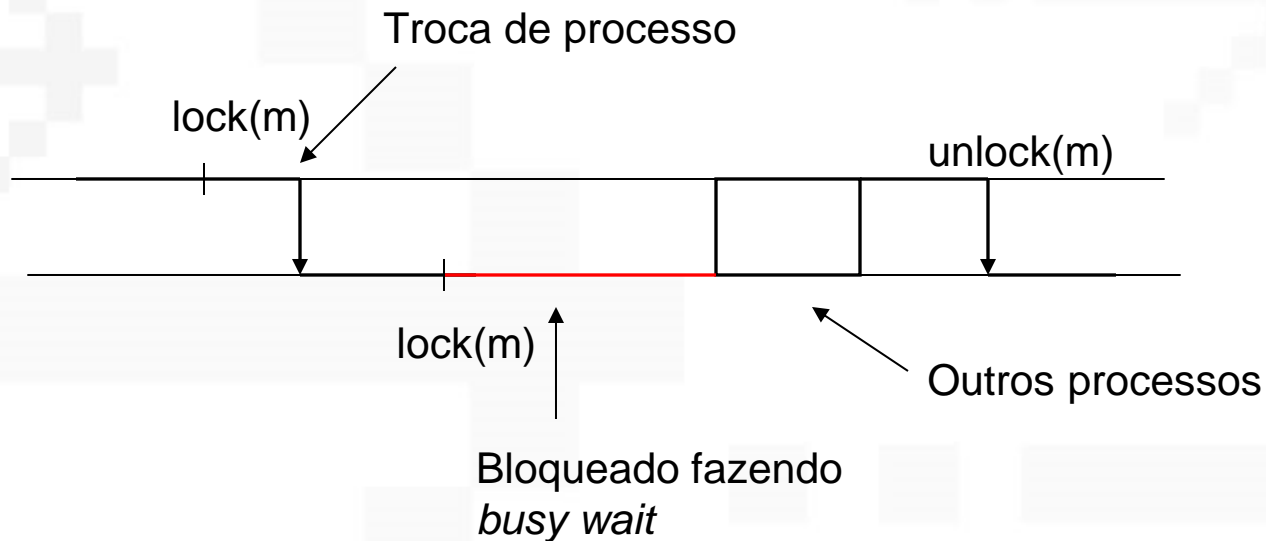
```
enter_region:  tst register,flag  
               cmp register,0  
               jnz  enter_region  
               ret
```

```
leave_region:  mov flag,0  
               ret
```

```
Seção crítica { lock(flag);  
               unlock(flag);
```

# Primitivas *lock* e *unlock*: problemas (1)

- *Busy waiting (spin lock)*
- Confiar no processo (programador)
  - Fazer o *lock* e o *unlock* corretamente
- Inversão de prioridades





# Primitivas *lock* e *unlock*: problemas (2)

- Solução:
  - Bloquear o processo ao invés de executar *busy waiting*
  - Baseado em duas novas primitivas
    - *sleep*: Bloqueia um processo a espera de uma sinalização
    - *wakeup*: Sinaliza um processo

# Semáforos

- Mecanismo proposto por Dijkstra (1965)
- Duas primitivas:
  - P (*Proberen*, testar)
  - V (*Verhogen*, incrementar)
- Semáforo é um tipo abstrato de dados:
  - Um valor inteiro
  - Fila de processo

# Implementação de semáforos

- Primitivas P e V

```
P(s): s.valor = s.valor - 1  
      Se s.valor < 0 {  
        Bloqueia processo (sleep);  
        Insere processo em S.fila;  
      }
```

```
V(s): s.valor = s.valor + 1  
      Se S.valor <= 0 {  
        Retira processo de S.fila;  
        Acorda processo (wakeup);  
      }
```

- Necessidade de garantir a atomicidade nas operações de incremento (decremento) e teste da variável compartilhada *s.valor*
  - Uso de *mutex*
- Dependendo dos valores assumidos por *s.valor*
  - Semáforos binários: *s.valor* = 1
  - Semáforos contadores: *s.valor* = n

# Semáforos *versus* mutex

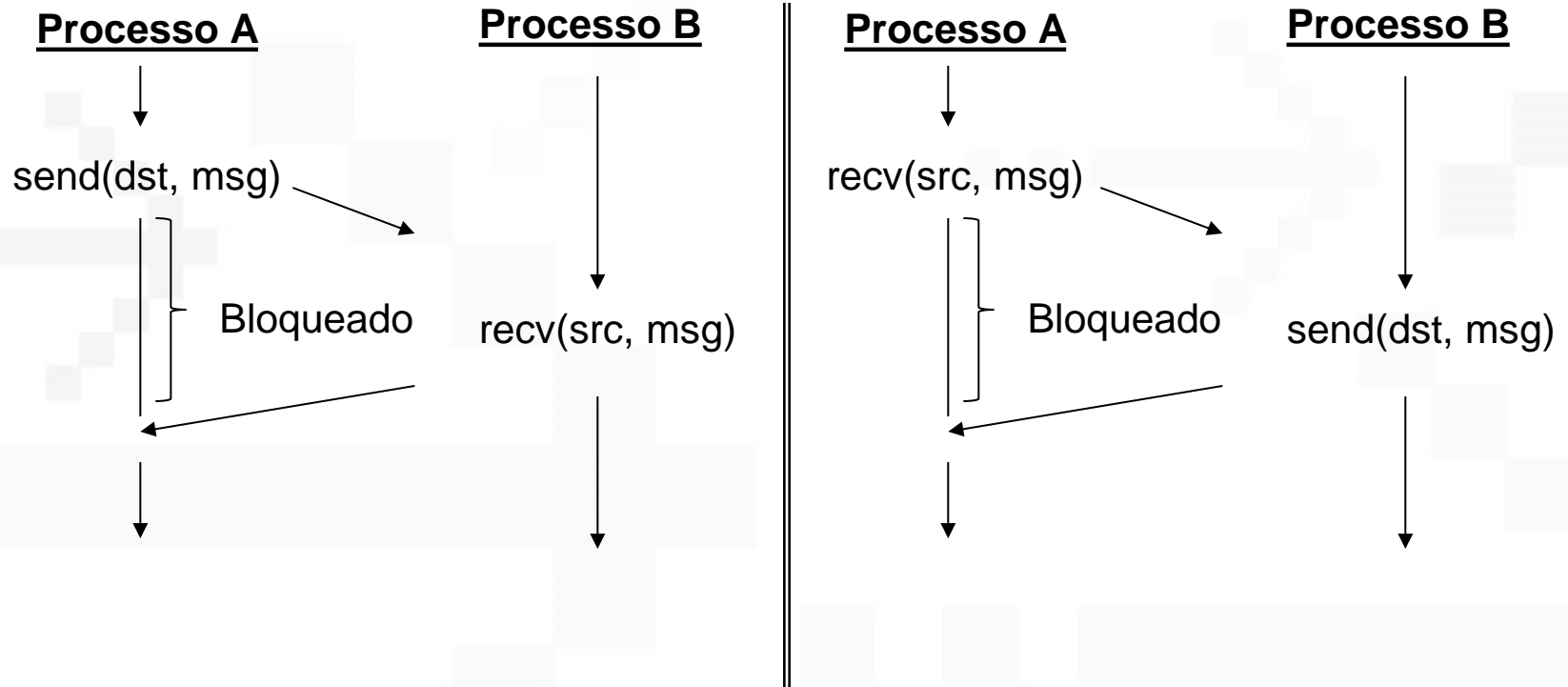
- Primitivas *lock* e *unlock* são necessariamente feitos por um mesmo processo
  - Acesso a seção crítica
- Primitivas *P* e *V* podem ser realizadas por processos diferentes
  - Gerência de recursos

# Troca de mensagens

- Primitivas do tipo mutex e semáforos são baseadas no compartilhamento de variáveis
  - Necessidade do compartilhamento de memória
  - Sistemas distribuídos não existe memória comum
- Novo paradigma de programação
  - Troca de mensagens
- Primitivas
  - *send e receive*
  - RPC (*Remote Procedure Call*)

# Primitivas *send* e *receive* (1)

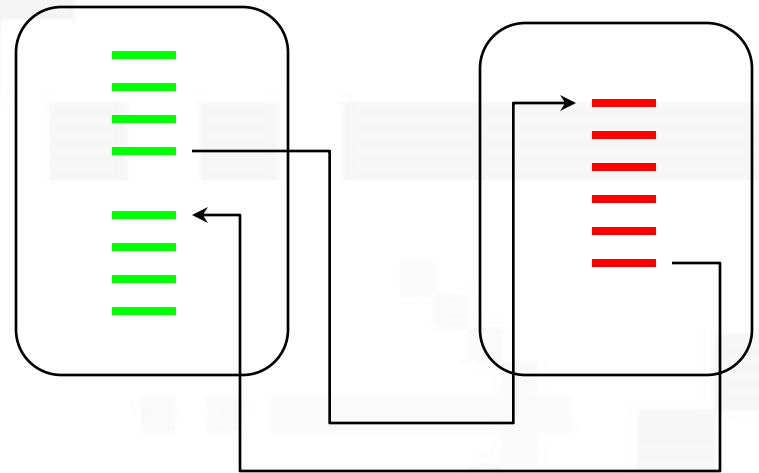
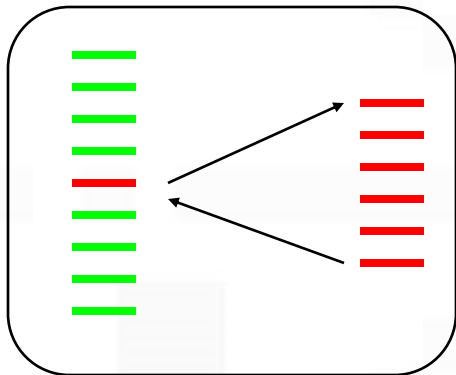
- Diferentes comportamentos em função da semântica das primitivas *send* e *receive*
- Funcionamento básico:



# Primitivas *send* e *receive* (2)

- Bibliotecas de comunicação
  - e.g. *sockets*, MPI, PVM, etc.
- Grande variedade de funções
  - Ponto a ponto
  - Em grupo
  - Primitivas para testar *status* e andamento de uma comunicação
  - Diferentes semânticas
    - Bloqueante
    - Não bloqueante
    - *Rendez vous*

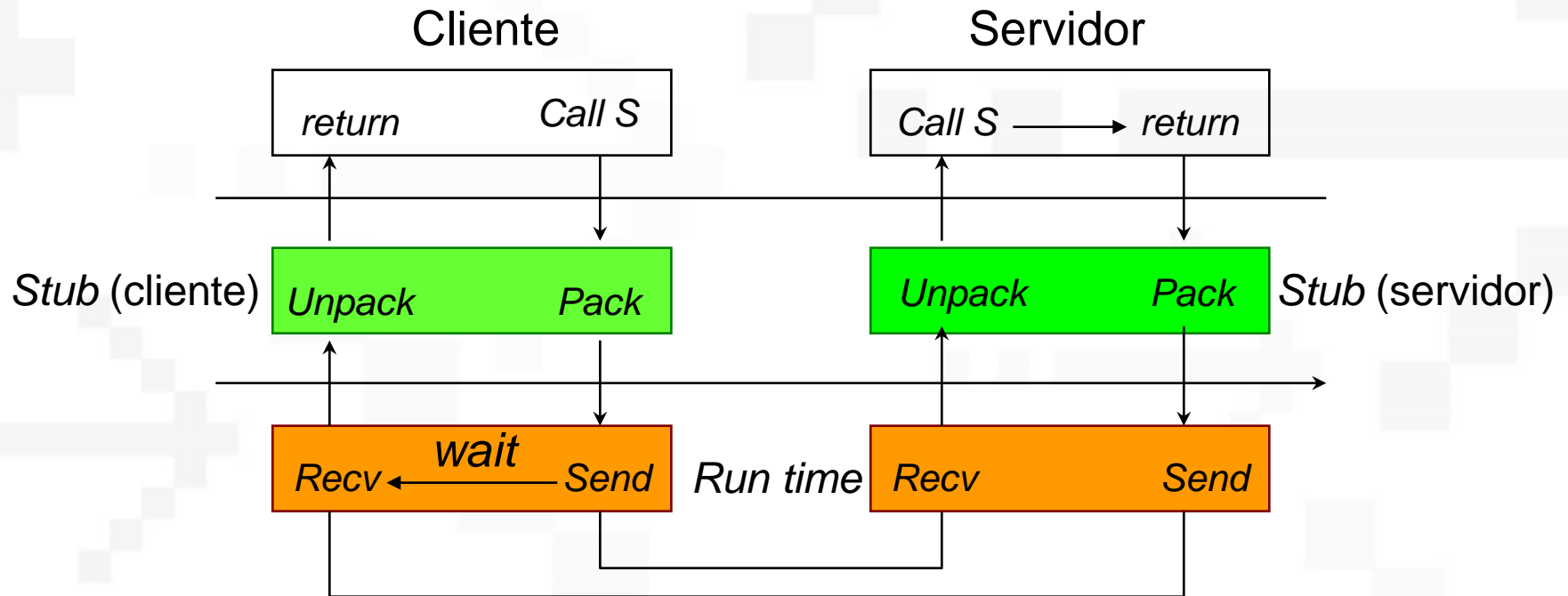
# Remote Procedure Call (1)



- Base de comunicação do DCE
- Composto por :
  - núcleo executivo (*run time*)
  - Interfaces para a geração de aplicações (Interface de programação)



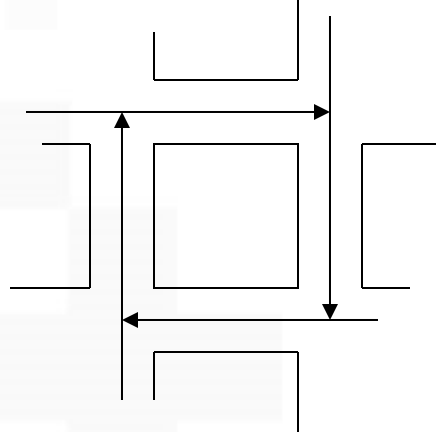
## Remote Procedure Call (2)



- Linguagem própria para descrever funções (chamada/definição)
- Compilador (*rpcgen*) para gerar *stubs* e ligar com programa aplicativo
- comunicação é toda gerada pelo *run-time* de forma transparente

# Deadlock

- Situação na qual um, ou mais processos, fica impedido de prosseguir sua execução devido ao fato de cada um estar aguardando acesso a recursos já alocados por outro processo



# Condições para ocorrência de *deadlocks* (1)

- Para que ocorra um *deadlock* quatro condições devem ser satisfeitas simultaneamente:
  1. Exclusão mútua:
    - Todo recurso ou está disponível ou está atribuído a um único processo
  2. Segura/espera:
    - Os processo que detém um recurso podem solicitar novos recursos

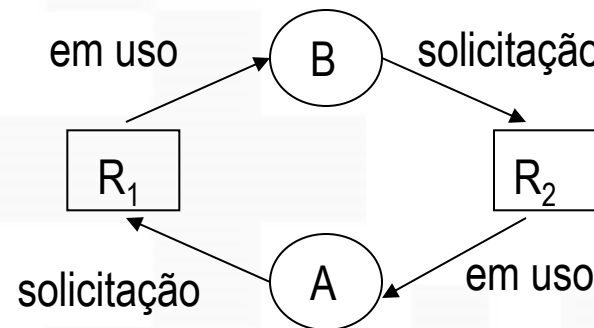
# Condições para ocorrência de *deadlocks* (2)

## 3. Recurso não preemptível:

- Um recurso concedido não pode ser retirado de um processo por outro

## 4. Espera circular:

- existência de um ciclo de 2 ou mais processos cada um esperando por um recurso já adquirido (em uso) pelo próximo processo no ciclo



# Estratégias para tratamento de *deadlocks*

- Ignorar
- Detecção e recuperação
  - Monitoração dos recursos liberados e alocados
  - Eliminação de processos
- Impedir ocorrência cuidando na alocação de recursos
  - Algoritmo do banqueiro
- Prevenção (por construção)
  - Evitar a ocorrência de pelo menos uma das quatro condições necessárias

# Leituras complementares

- R. Oliveira, A. Carissimi, S. Toscani; Sistemas Operacionais. Editora Bookman, 2010.
  - Capítulo 3
- A. Silberchatz, P. Galvin, G. Gagne; Applied Operating System Concepts. Addison-Wesley, 2000, (1ª edição).
  - Capítulo 6 e 7
- W. Stallings; Operating Systems. (4<sup>th</sup> edition). Prentice Hall, 2001.
  - Capítulo 5 e 6