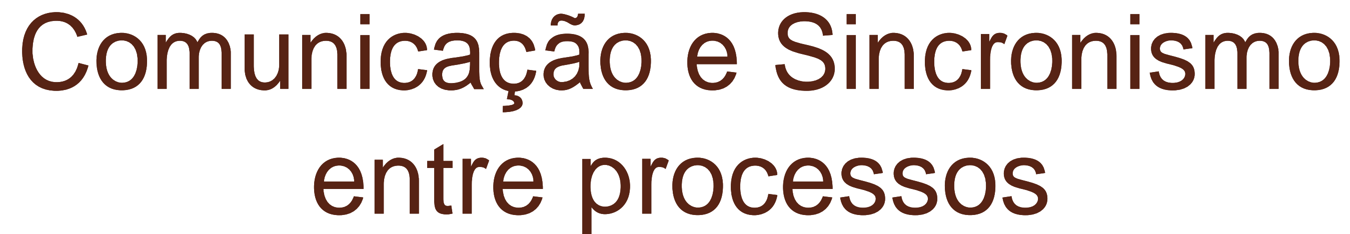


Comunicação e Sincronismo entre processos

Prof. Marcos Ribeiro Quinet de Andrade
Universidade Federal Fluminense - UFF
Pólo Universitário de Rio das Ostras - PURO

Comunicação entre Processos

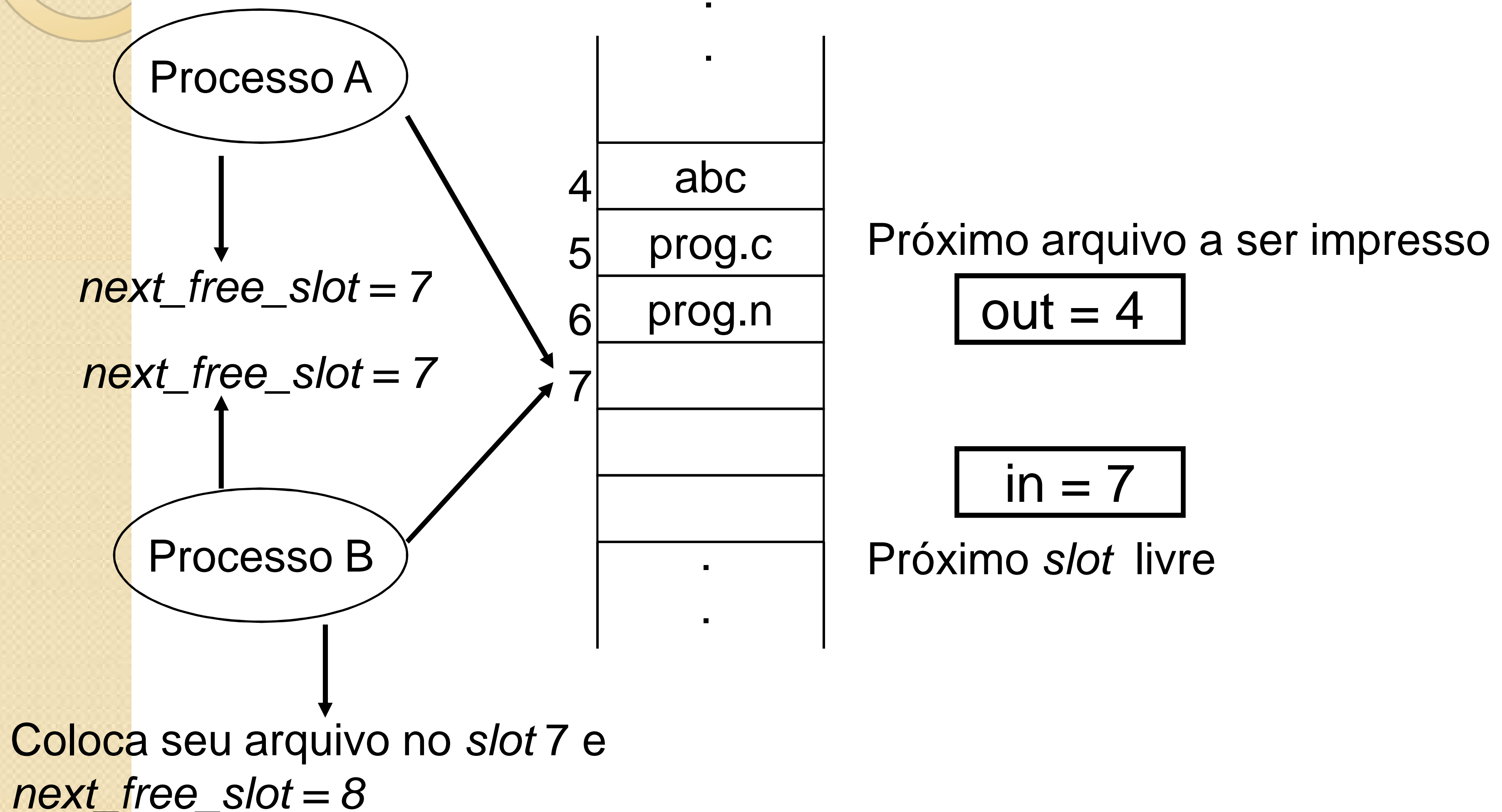
- Processos precisam se comunicar;
- Processos competem por recursos;
- Três aspectos importantes:
  - Como um processo passa informação para outro processo;
  - Como garantir que processos não invadam espaços uns dos outros;
  - Dependência entre processos: sequência adequada;

Comunicação entre Processos

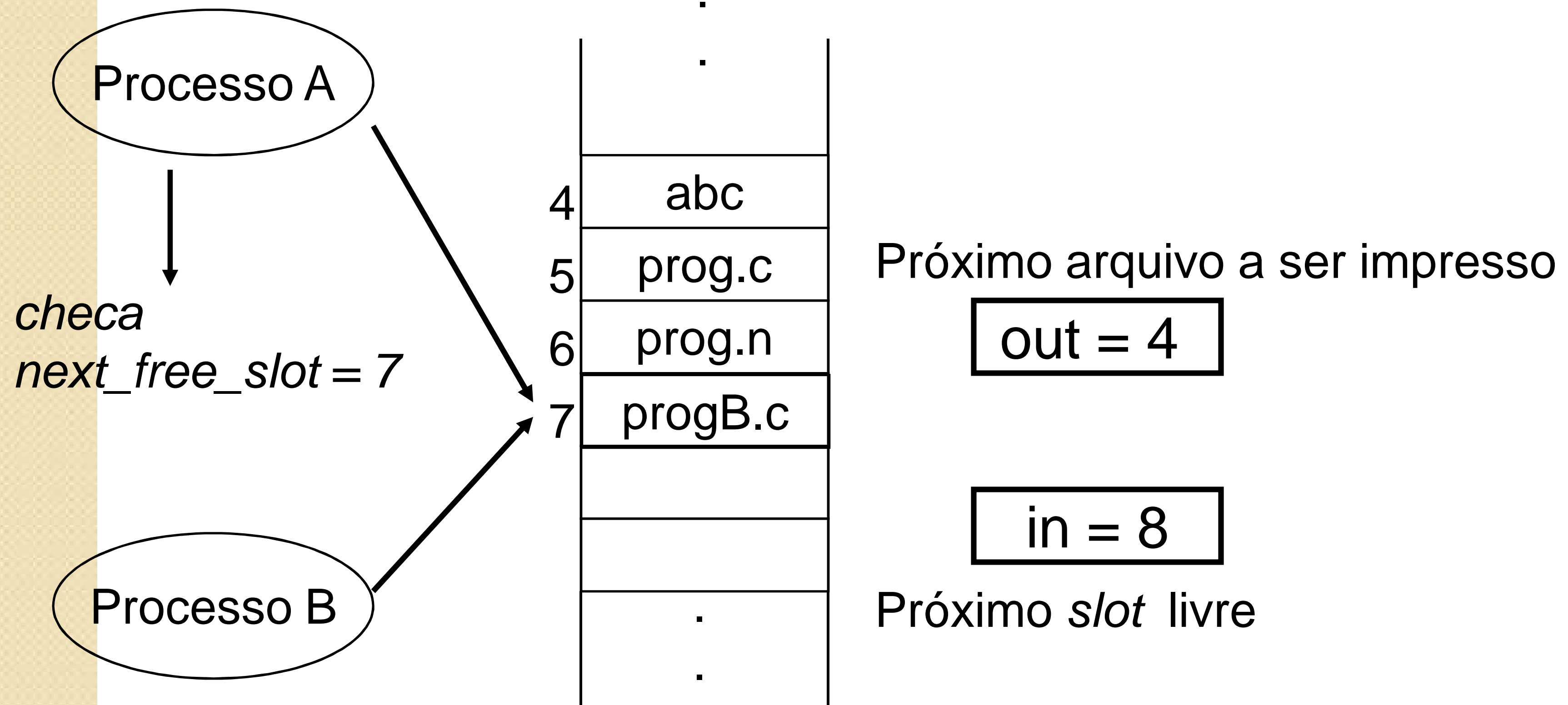
- ***Condições de disputa*** *(race conditions)*: processos acessam recursos compartilhados concorrentemente;
  - Recursos: memória, arquivos, impressoras, discos, variáveis;

- Ex.: Impressão: quando um processo deseja imprimir um arquivo, ele coloca o arquivo em um local especial chamado ***spooler*** (tabela). Um outro processo, chamado ***printer spooler***, checa se existe algum arquivo a ser impresso. Se existe, esse arquivo é impresso e retirado do *spooler*. Imagine dois processos que desejam imprimir um arquivo ao mesmo tempo...

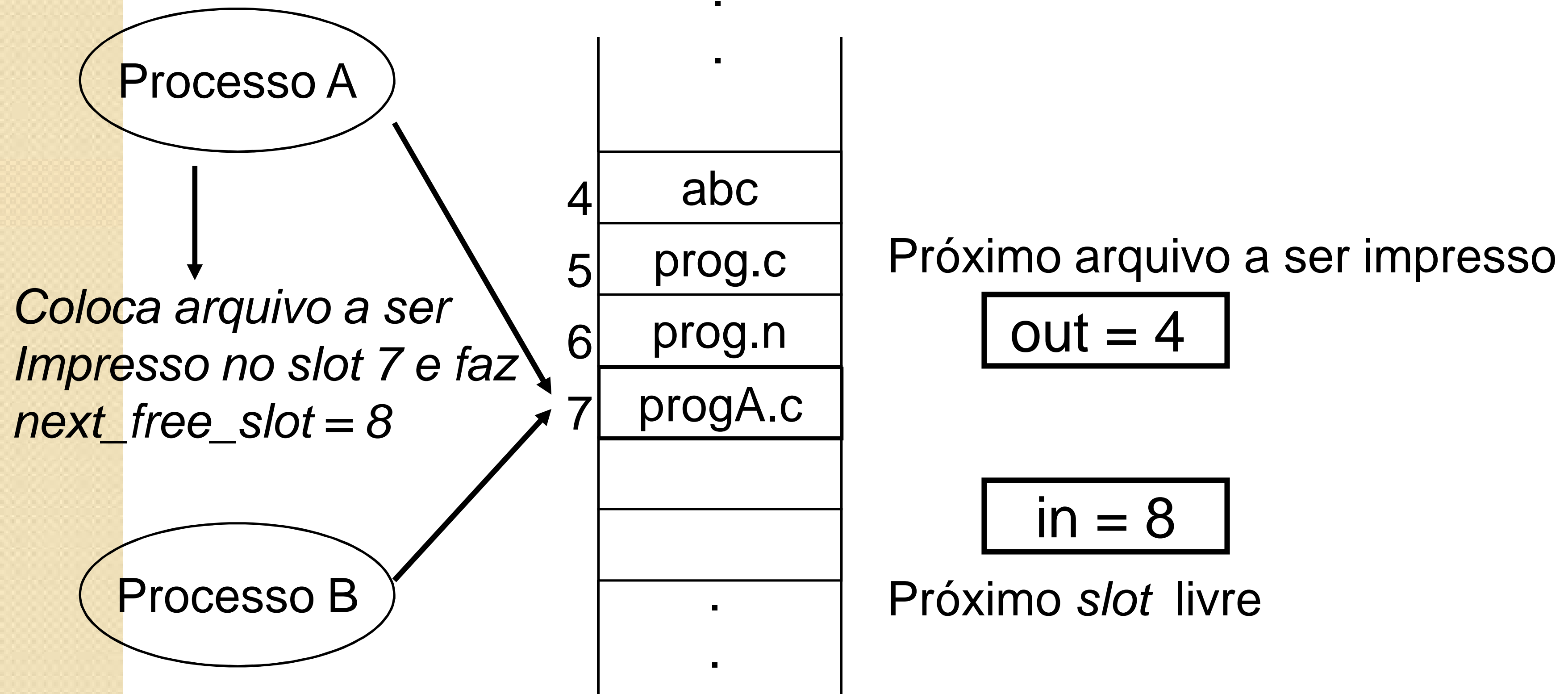
Comunicação entre Processos



Comunicação entre Processos



Comunicação entre Processos



Processo B nunca receberá sua impressão!!!!!

Comunicação entre Processos – Regiões Críticas

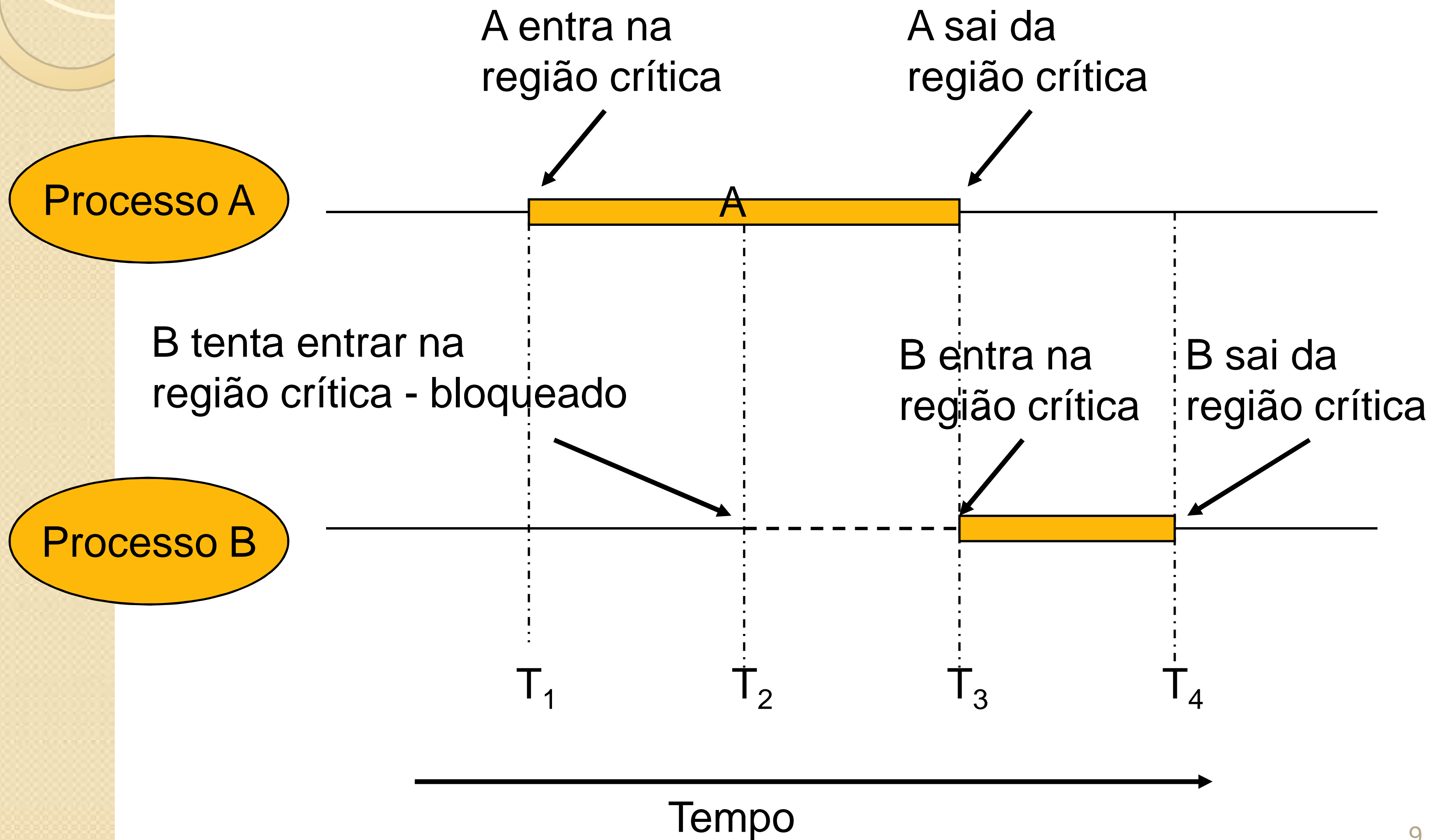
- Como solucionar problemas de *Race Conditions*???
  - Proibir que mais de um processo leia ou escreva em recursos compartilhados concorrentemente (ao “mesmo tempo”)
    - **Recursos compartilhados → regiões críticas**;
  - Exclusão mútua: garantir que um processo não terá acesso à uma região crítica quando outro processo está utilizando essa região;

Comunicação entre Processos – Exclusão Mútua

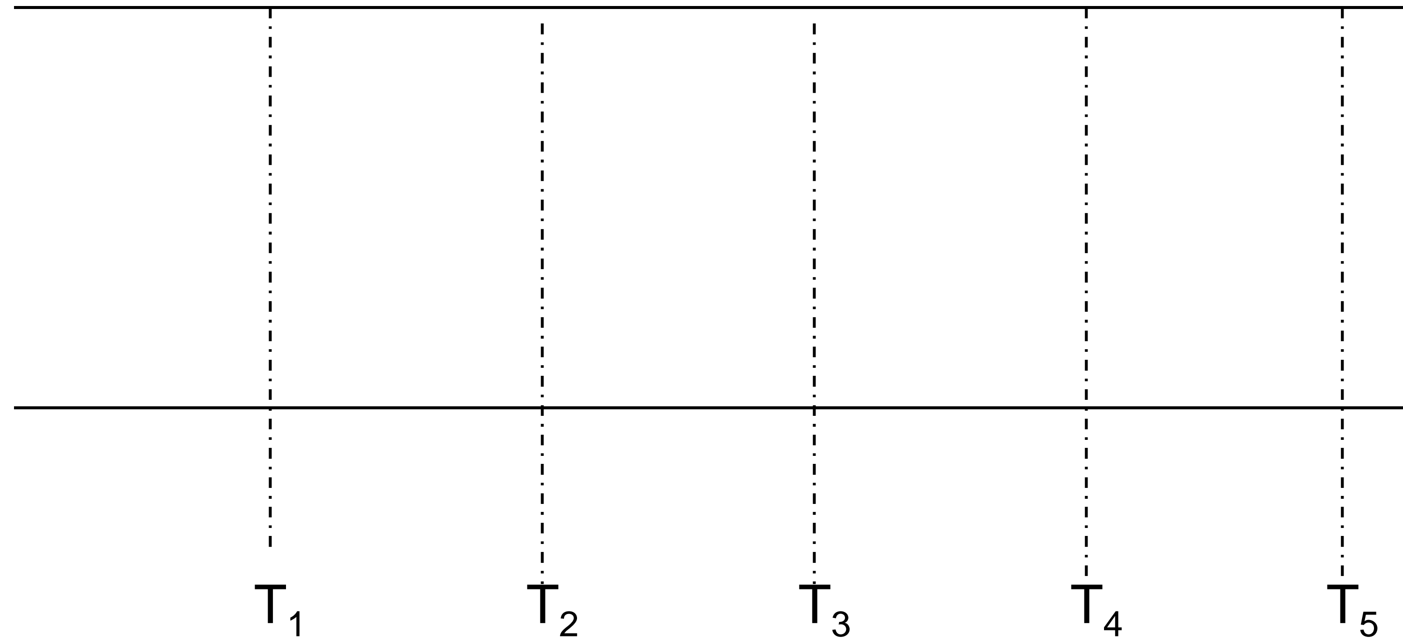
- Quatro condições para uma boa solução:

1. Dois processos não podem estar simultaneamente em regiões críticas;
2. Nenhuma restrição deve ser feita com relação à CPU;
3. Processos que não estão em regiões críticas não podem bloquear outros processos que desejam utilizar regiões críticas;
4. Processos não podem esperar para sempre para acessarem regiões críticas;

Comunicação entre Processos – Exclusão Mútua



Comunicação entre Processos – Exclusão Mútua



Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - Monitores;
  - Passagem de Mensagem;

Comunicação entre Processos – Exclusão Mútua

- Espera Ocupada (*Busy Waiting)*: constante checagem por algum valor;
- Algumas soluções para exclusão mútua com espera ocupada:
  - Desabilitar interrupções;
  - Variáveis de Travamento (*Lock)*;
  - Estrita Alternância (*Strict Alternation)*;
  - Solução de Peterson e Instrução TSL;

Comunicação entre Processos – Exclusão Mútua

- Desabilitar interrupções:
  - Processo desabilita todas as suas interrupções ao entrar na região crítica e habilita essas interrupções ao sair da região crítica;
  - Com as interrupções desabilitadas, a CPU não realiza chaveamento entre os processos;
    - Viola condição 2 (Nenhuma restrição deve ser feita com relação à CPU);
  - Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções e não ser finalizado;
    - Viola condição 4 (Processos não podem esperar para sempre para acessarem regiões críticas);

Comunicação entre Processos – Exclusão Mútua

- Variáveis *Lock*:
  - O processo que deseja utilizar uma região crítica atribuí um valor a uma variável chamada *lock*;
  - Se a variável está com valor 0 (zero) significa que nenhum processo está na região crítica; Se a variável está com valor 1 (um) significa que existe um processo na região crítica;
  - Apresenta o mesmo problema do exemplo do *spooler de impressão*;

Comunicação entre Processos – Exclusão Mútua

- Variáveis *Lock* - Problema:
  - Suponha que um processo A leia a variável *lock* com valor 0;
  - Antes que o processo A posso alterar a variável para o valor 1, um processo B é escalonado e altera o valor de *lock* para 1;
  - Quando o processo A for escalonado novamente, ele altera o valor de *lock* para 1, e ambos os processos estão na região crítica;
    - Viola condição 1 (Dois processos não podem estar simultaneamente em regiões críticas);

Comunicação entre Processos – Exclusão Mútua

- Variáveis *Lock*: `lock==0`;

```
while(true){
     while(lock!=0); //loop
     lock=1;
     critical_region();
     lock=0;
     non-critical_region();
}
```

Processo A

```
while(true){
     while(lock!=0); //loop
     lock=1;
     critical_region();
     lock=0;
     non-critical_region();
}
```

Processo B

Comunicação entre Processos – Exclusão Mútua

- *Strict Alternation*:
  - Fragmentos de programa controlam o acesso às regiões críticas;
  - Variável `turn`, inicialmente em `0`, estabelece qual processo pode entrar na região crítica;

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

(Processo A)
`turn` **0**

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

(Processo B)
`turn` **1**

Comunicação entre Processos – Exclusão Mútua

- Problema do *Strict Alternation:*

1. Suponha que o Processo B é mais rápido e sai da região crítica;
2. Ambos os processos estão fora da região crítica e `turn` com valor 0;
3. O processo A termina antes de executar sua região não crítica e retorna ao início do *loop*; Como o `turn` está com valor zero, o processo A entra novamente na região crítica, enquanto o processo B ainda está na região não crítica;
4. Ao sair da região crítica, o processo A atribui o valor 1 à variável `turn` e entra na sua região não crítica;

Comunicação entre Processos – Exclusão Mútua

- Problema do *Strict Alternation:*

5. Novamente ambos os processos estão na região não crítica e a variável `turn` está com valor 1;
6. Quando o processo A tenta novamente entrar na região crítica, não consegue, pois `turn` ainda está com valor 1;
7. **Assim, o processo A fica bloqueado pelo processo B que NÃO está na sua região crítica, violando a condição 3**;

Comunicação entre Processos – Exclusão Mútua

- Solução de Peterson e Instrução TSL (*Test and Set Lock*):
  - Uma variável (ou programa) é utilizada para bloquear a entrada de um processo na região crítica quando um outro processo está na região;
  - Essa variável é compartilhada pelos processos que concorrem pelo uso da região crítica;
  - Ambas as soluções possuem fragmentos de programas que controlam a entrada e a saída da região crítica;

Comunicação entre Processos – Exclusão Mútua

- Solução de Peterson

```
#define FALSE 0
#define TRUE 1
#define N 2

int turn;
int interested[N];

void enter_region(int process)
{
  int other;

  other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (turn ==  process) && interested[other] == TRUE) ;
}

void leave_region(int process)
{
  interested[process] = FALSE;
}
```

Comunicação entre Processos – Exclusão Mútua

- Instrução TSL: utiliza registradores do hardware;
  - TSL RX, LOCK; (lê o conteúdo de ***lock*** em RX, e armazena um valor diferente de zero (0) em ***lock*** – operação indivisível);
  - ***Lock*** é compartilhada
    - Se ***lock***==0, então região crítica "liberada".
    - Se ***lock***<>0, então região crítica "ocupada".

```
enter_region:
    TSL REGISTER, LOCK          | Copia lock para reg. e lock=1
    CMP REGISTER, #0            | lock valia zero?
    JNE enter_region            | Se sim, entra na região crítica,
                                | Se não, continua no laço
    RET                         | Retorna para o processo chamador

leave_region
    MOVE LOCK, #0               | lock=0
    RET                         | Retorna para o processo chamador
```

Comunicação entre 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;

Comunicação entre 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 entre 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;

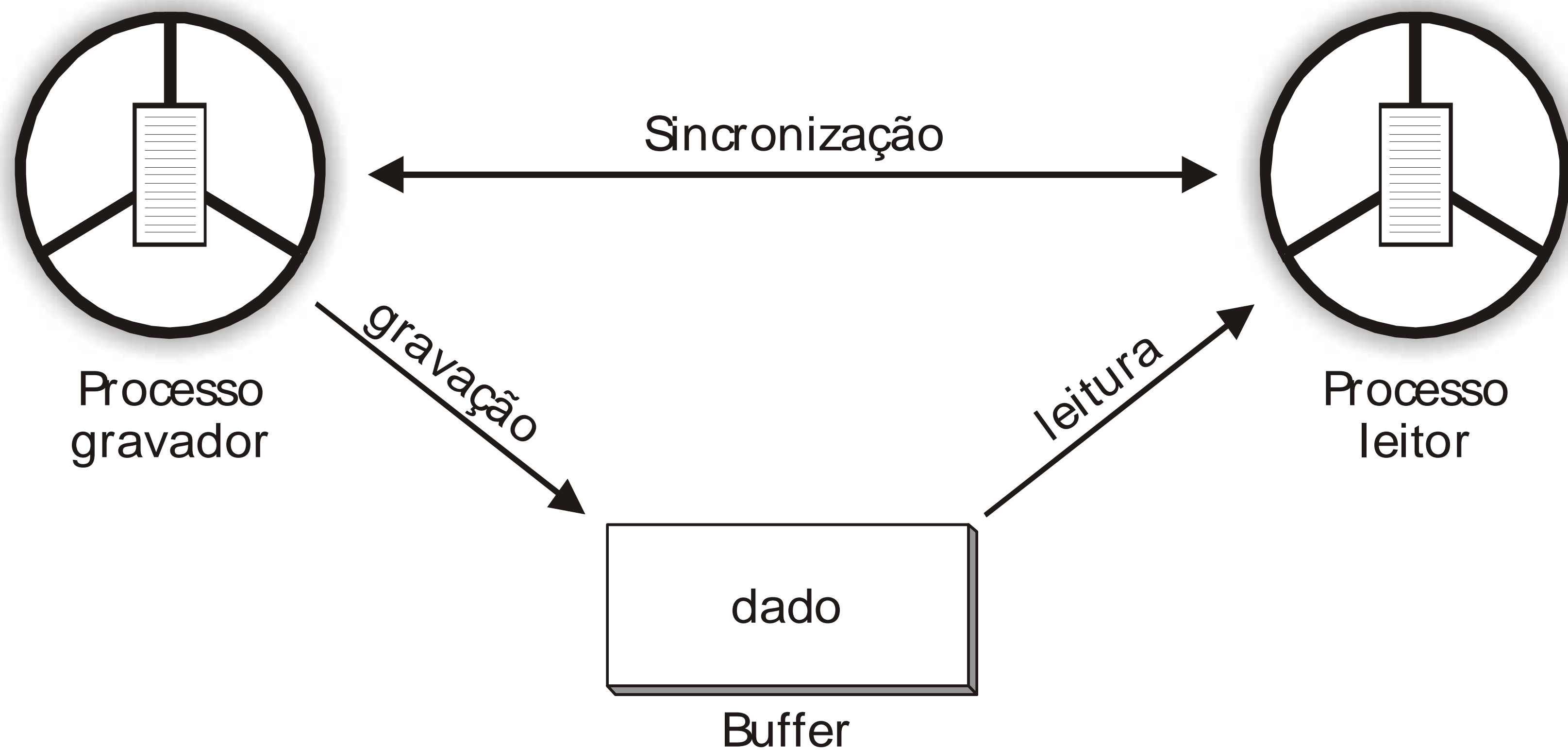
Comunicação entre Processos – Primitivas *Sleep/Wakeup*

- *Buffer*: uma variável `count` controla a quantidade de dados presente no *buffer*.
- Produtor: Antes de colocar dados no *buffer*, o processo produtor checa o valor dessa variável. Se a variável está com valor máximo, o processo produtor é colocado para dormir. Caso contrário, o produtor coloca dados no *buffer* e o incrementa.

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

- Consumidor: Antes de retirar dados no *buffer*, o processo consumidor checa o valor da variável `count` para saber se ela está com 0 (zero). Se está, o processo vai "dormir", senão ele retira os dados do *buffer* e decrementa a variável;

Comunicação entre Processos Sincronização Produtor-Consumidor



Comunicação entre 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 entre Processos – Primitivas *Sleep/Wakeup*

- Problemas desta solução: Acesso à variável `count` é irrestrita
  - 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*;

Comunicação entre Processos – Primitivas *Sleep/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...

- 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!!!!

Comunicação entre 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*;*

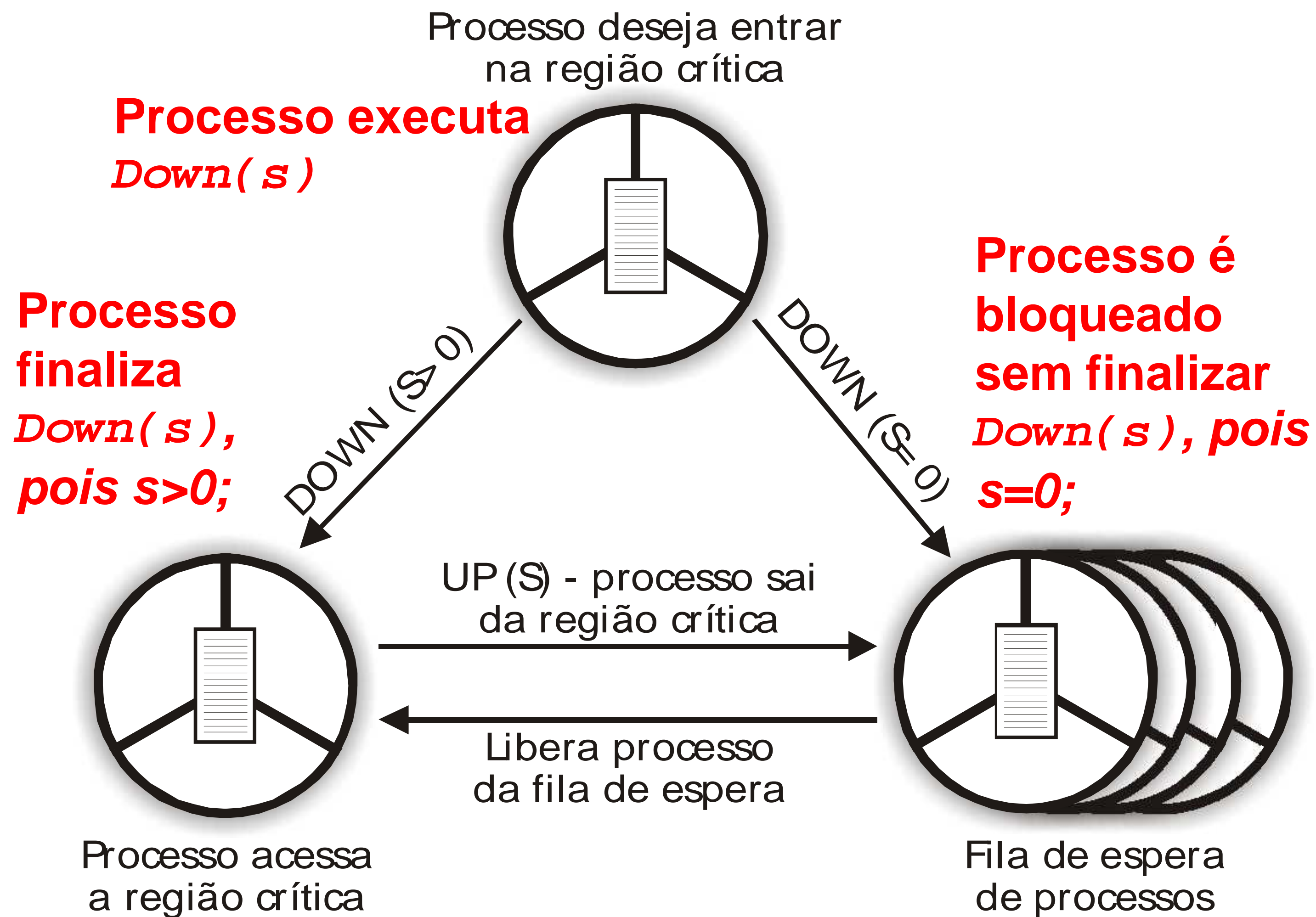
Comunicação entre Processos – Semáforos

- ***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 **ações 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 entre 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 entre Processos – Semáforo Binário



Comunicação entre 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 entre 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 entre 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.

Comunicação entre 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 entre 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 → o Compilador é que garante a exclusão mútua.

Java possui este recurso

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

Comunicação entre 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 entre 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 entre 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 entre 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 entre 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 entre Processos – Monitores

- A exclusão mútua automática dos procedimentos do monitor garante que, por exemplo, se o produtor dentro de um procedimento do monitor descobrir que o buffer está cheio, esse produtor será capaz de terminar a operação de `WAIT` sem se preocupar, pois o consumidor não estará ativo dentro do monitor até que `WAIT` tenha terminado e o produtor tenha sido marcado como não mais executável;

Comunicação entre 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;

Comunicação entre 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 entre 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 entre 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 entre 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 – sequência 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 entre 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 entre 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 entre Processos
Outros mecanismos

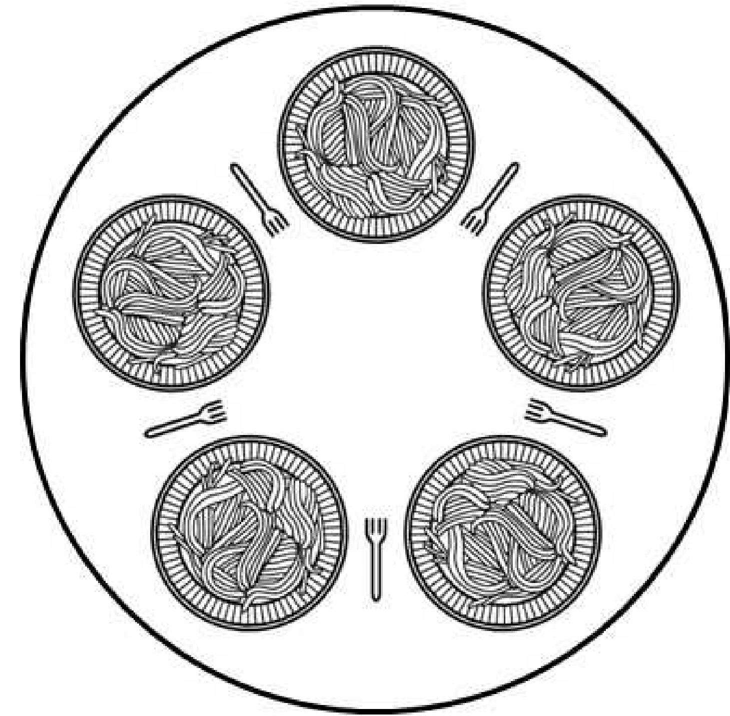
- RPC – *Remote Procedure Call*
  - Rotinas que permitem comunicação de processos em diferentes máquinas;
  - Chamadas remotas;
- MPI – *Message-passing Interface*;
  - Sistemas paralelos;
- RMI Java – *Remote Method Invocation*
  - Permite que um objeto ativo em uma máquina virtual Java possa interagir com objetos de outras máquinas virtuais Java, independentemente da localização dessas máquinas virtuais;
- Web Services
  - Permite que serviços sejam compartilhados através da Web

Comunicação entre Processos
Outros mecanismos

- *Pipe*:
  - Permite a criação de filas de processos;
  - `ps -ef | grep produtor_consumidor`;
  - 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);

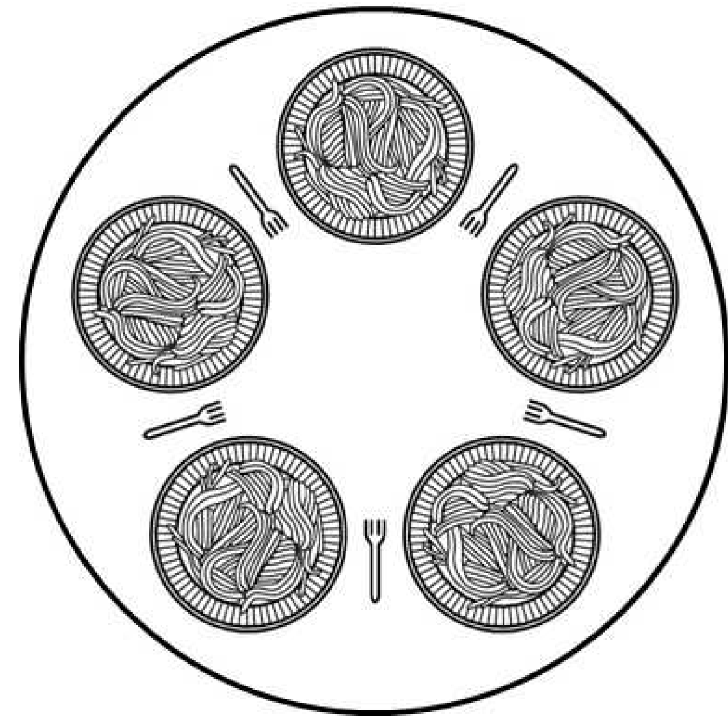
Problemas clássicos de comunicação entre processos

- Problema do Jantar dos Filósofos
  - Cinco filósofos desejam comer espaguete; No entanto, para poder comer, cada filósofo precisa utilizar dois garfo e não apenas um. Portanto, os filósofos precisam compartilhar o uso do garfo de forma sincronizada.
  - Os filósofos comem e pensam;



Problemas clássicos de comunicação entre processos

- Problemas que devem ser evitados:
  - *Deadlock* – todos os filósofos pegam **<u>um garfo</u>** ao mesmo tempo;
  - *Starvation* – os filósofos **<u>fiquem indefinidamente pegando garfos simultaneamente</u>**;



Solução 1 para Filósofos (1/2)

```
#define N 5                              /* number of philosophers */

void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                        /* philosopher is thinking */
        take_fork(i);                    /* take left fork */
        take_fork((i+1) % N);            /* take right fork; % is modulo operator */
        eat( );                          /* yum-yum, spaghetti */
        put_fork(i);                     /* put left fork back on the table */
        put_fork((i+1) % N);             /* put right fork back on the table */
    }
}
```

Solução 1 para Filósofos (2/2)

- Problemas da solução 1:
  - Execução do `take_fork(i)` → Se todos os filósofos pegarem o garfo da esquerda, nenhum pega o da direita → *Deadlock*;
- Se modificar a solução (mudança 1):
  - Verificar antes se o garfo da direita está disponível. Se não está, devolve o da esquerda e começa novamente → *Starvation (Inanição);*
  - Tempo fixo ou tempo aleatório (rede Ethernet);
    - Serve para sistemas não-críticos;

Solução 1 para Filósofos (2/2)

- Se modificar a solução (mudança 2):

```
#define N 5                              /* number of philosophers */
 semaphore mutex = 1;
void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                      /* philosopher is thinking */
          take_fork(i);                  /* take left fork */
          take_fork((i+1) % N);          /* take right fork; % is modulo operator */
          eat( );                        /* yum-yum, spaghetti */
          put_fork(i);                   /* put left fork back on the table */
          put_fork((i+1) % N);           /* put right fork back on the table */
     }
}
```

down(&mutex); (inserido antes de take_fork(i);)

up(&mutex); (inserido após put_fork((i+1) % N);)

Somente um filósofo come!

Solução 2 para Filósofos usando Semáforos (1/3)

- Não apresenta:
  - *Deadlocks*;
  - *Starvation*;
- Permite o máximo de "paralelismo";

Solução 2 para Filósofos usando Semáforos (2/3)

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N−1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */

void philosopher(int i)         /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {              /* repeat forever */
        think( );               /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat( );                 /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}
```

Solução 2 para Filósofos usando Semáforos (3/3)

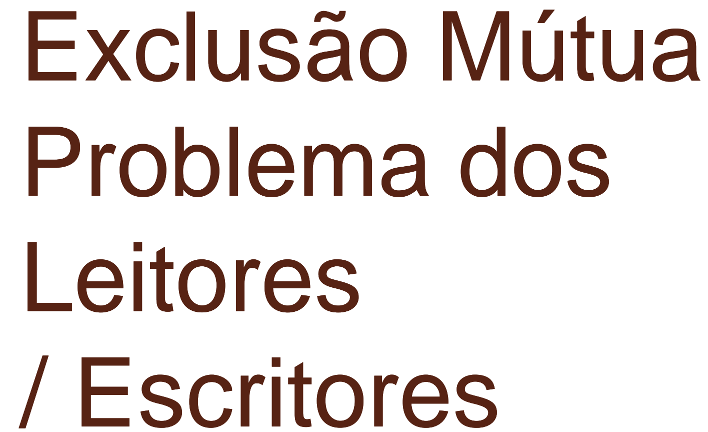
```
void take_forks(int i)              /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                 /* enter critical region */
      state[i] = HUNGRY;            /* record fact that philosopher i is hungry */
      test(i);                      /* try to acquire 2 forks */
      up(&mutex);                   /* exit critical region */
      down(&s[i]);                  /* block if forks were not acquired */
}

void put_forks(i)                   /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                 /* enter critical region */
      state[i] = THINKING;          /* philosopher has finished eating */
      test(LEFT);                   /* see if left neighbor can now eat */
      test(RIGHT);                  /* see if right neighbor can now eat */
      up(&mutex);                   /* exit critical region */
}

void test(i)                        /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

Exclusão Mútua
Problema dos Leitores / Escritores

- O problema dos Leitores e Escritores modela o acesso compartilhado a uma base de dados. Processos leitores e processos escritores competem por um acesso a essa base. É possível que vários processos leitores acessem a base ao mesmo tempo, no entanto, quando um processo escritor está escrevendo (modificando) a base de dados, nenhum outro processo pode realizar um acesso, nem mesmo um processo leitor.



Exclusão Mútua Problema dos Leitores / Escritores

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
        while (TRUE) {
                down(&mutex);
                rc = rc + 1;
                if (rc == 1) down(&db);
                up(&mutex);
                read_data_base( );
                down(&mutex);
                rc = rc – 1;
                if (rc == 0) up(&db);
                up(&mutex);
                use_data_read( );
        }
}


void writer(void)
{
        while (TRUE) {
                think_up_data( );
                down(&db);
                write_data_base( );
                up(&db);
        }
```

Exclusão Mútua
Problema do Barbeiro

- Na barbearia há um barbeiro, uma cadeira de barbeiro e *n* cadeiras para os clientes esperarem para ser atendidos. Quando não há clientes, o barbeiro senta-se na cadeira do barbeiro e dorme. Quando um cliente chega, ele precisa acordar o barbeiro para ser atendido. Se outros clientes chegarem enquanto o barbeiro estiver ocupado cortando o cabelo de algum cliente, eles se sentam se houver cadeiras disponíveis para clientes, senão eles vão embora se todas as cadeiras para clientes estiverem ocupadas.

Exclusão Mútua Problema do Barbeiro

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting – 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair( );              /* cut hair (outside critical region) */
     }
}


void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```