

# Comunicação & Sincronização (entre processos/ threads)



## Roteiro:

- Concorrência e Condições de Corrida
- Mecanismos de Sincronização:
  - Semáforo, Mutex e Variável de Condição Monitor
- Problemas Clássicos de Concorrência
- Envio de Mensagem
- Inter Process Communication em Unix:
  - Pipe, FIFO, Message Queue, Sockets

# Comunicação Inter-Processos (IPC)

- Existem inúmeras situações em que processos do sistema (ou do usuário) precisam interagir para se comunicar ou sincronizar as suas ações, por exemplo, no acesso compartilhado a dados ou recursos.

Comunicação Inter-Processos envolve:

- sincronização de atividades
- troca de dados

# Comunicação Inter-Processos(IPC)

- Processos (e threads) são entidades independentes que podem ser executados em qualquer ordem e interrompidos a qualquer momento
- A ordem de escalonamento é imprevisível
- Precisa-se de mecanismos para evitar problemas de inconsistência de dados compartilhados decorrentes da execução concorrente.



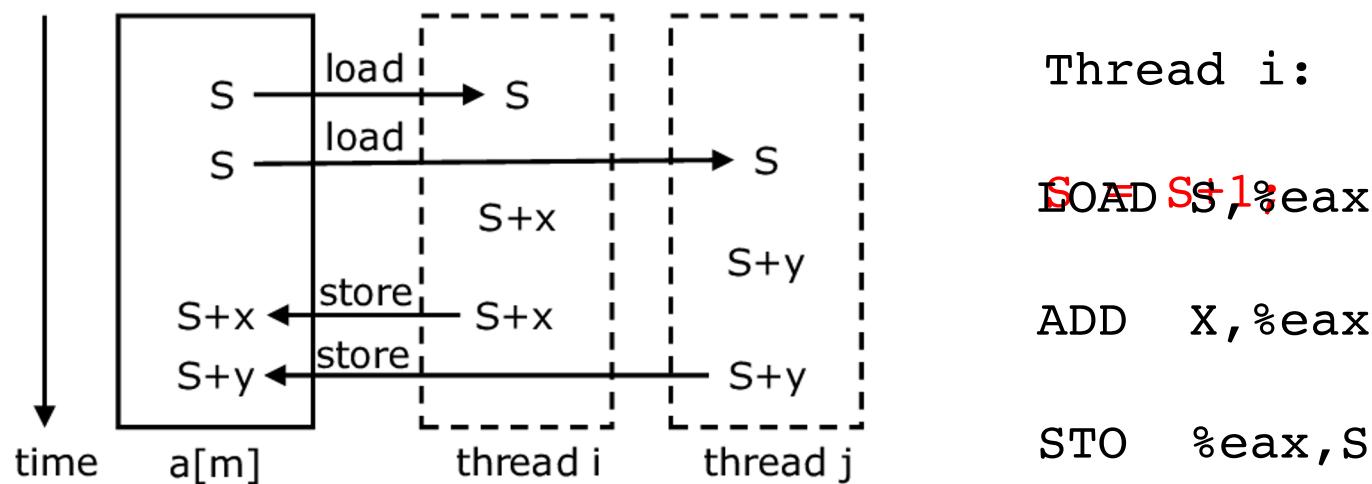
IPC entre processos



IPC entre threads

# Condição de Corrida

- Ocorre quando mais de um processo/ thread acessam uma memória compartilhada
- É a situação em que dois processos lêem e escrevem um dado compartilhado e o resultado final da manipulação depende da ordem em que os processos/thread são executados.

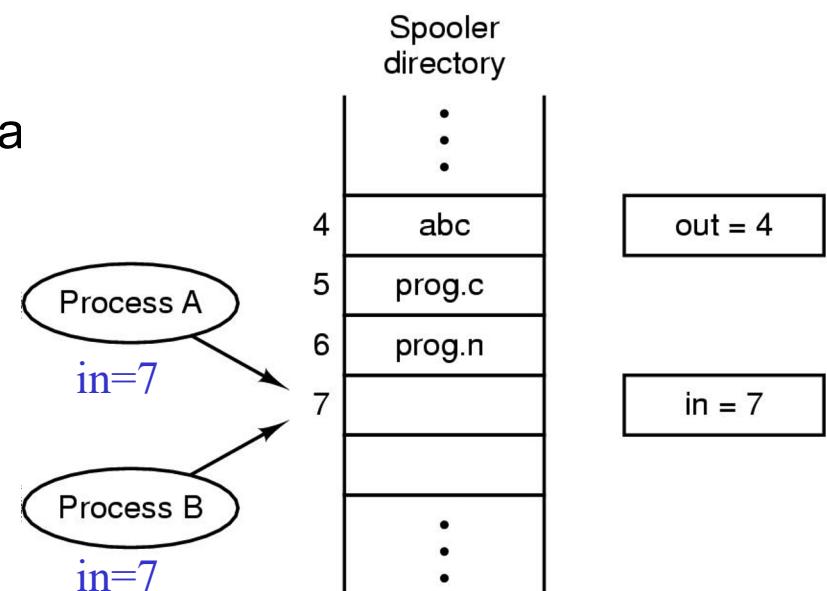


Depurar programas que contém condições de corrida é muito difícil, pois não é possível prever em que sequência as instruções de cada thread serão executadas.

# Condição de Corrida

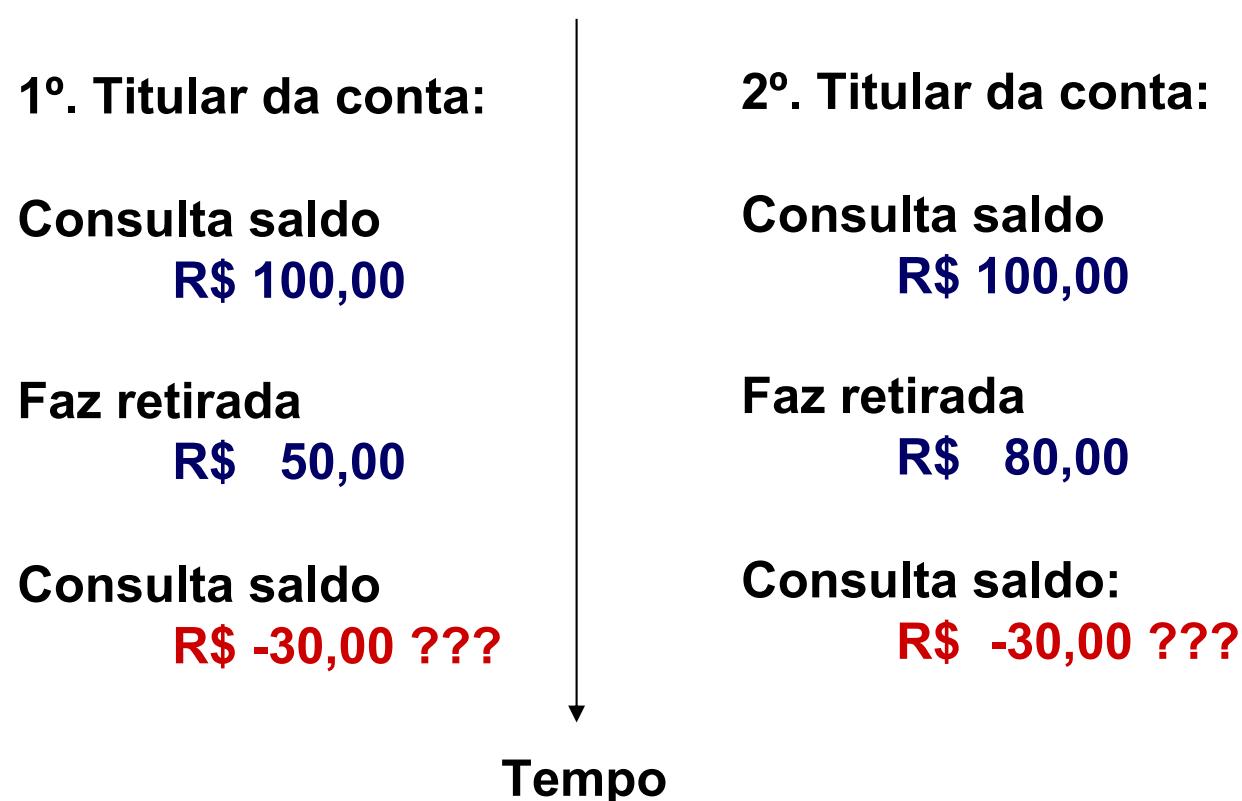
Exemplo: Dois processos concorrentes querem adicionar um job de impressão a fila de spooler:

- processo A lê memória compartilhada “in=7”, e logo depois é interrompido,
- Processo B faz o mesmo e inclui seu arquivo na posição 7 do spool de impressão
- Quando A retoma execução, sobrescreve a posição 7 com seu arquivo.



# Condição de Corrida

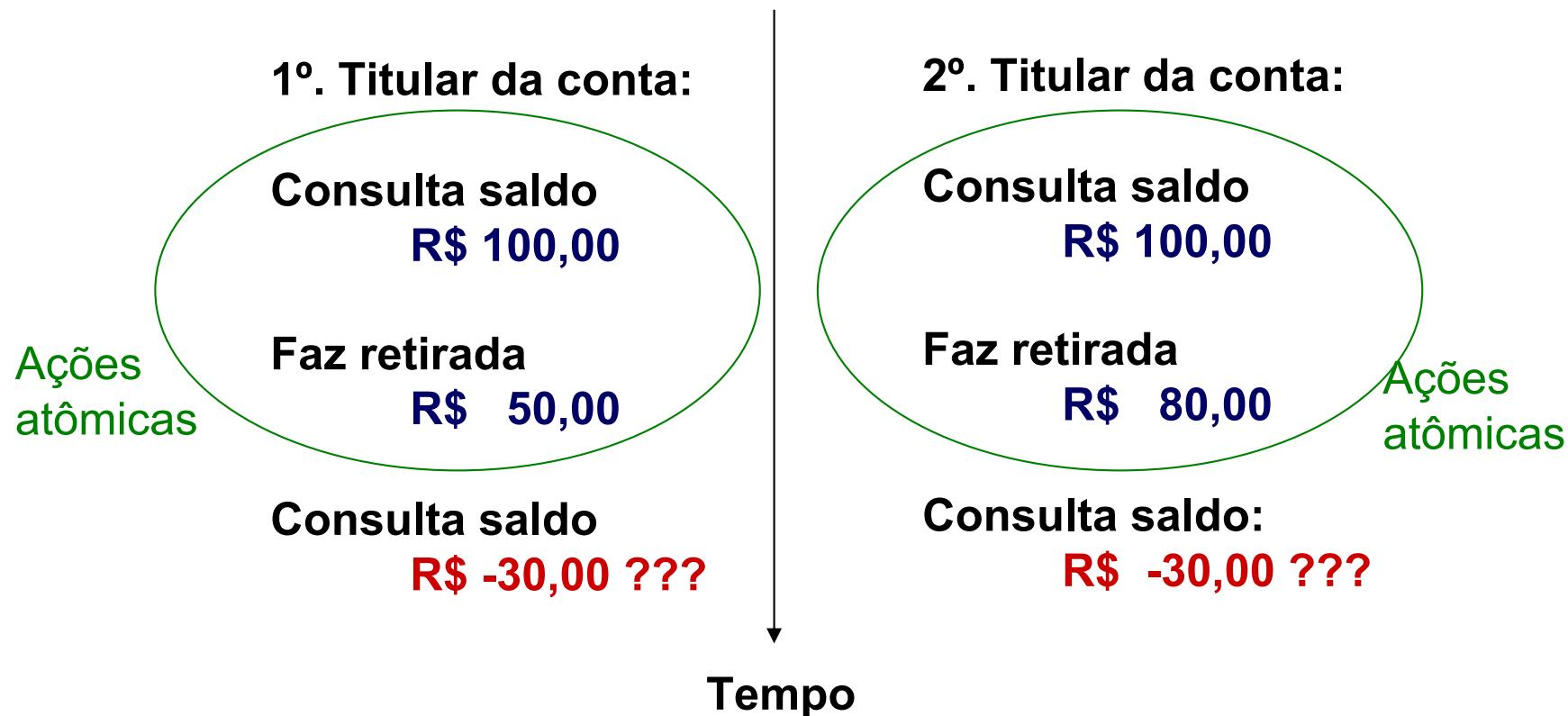
Exemplo: atualização de conta bancária conjunta  
(se o valor do saldo e a operação forem armazenados/executados em  
cada processo)



Explique o que aconteceu e indique como evitar que a conta fique negativa

# Condição de Corrida

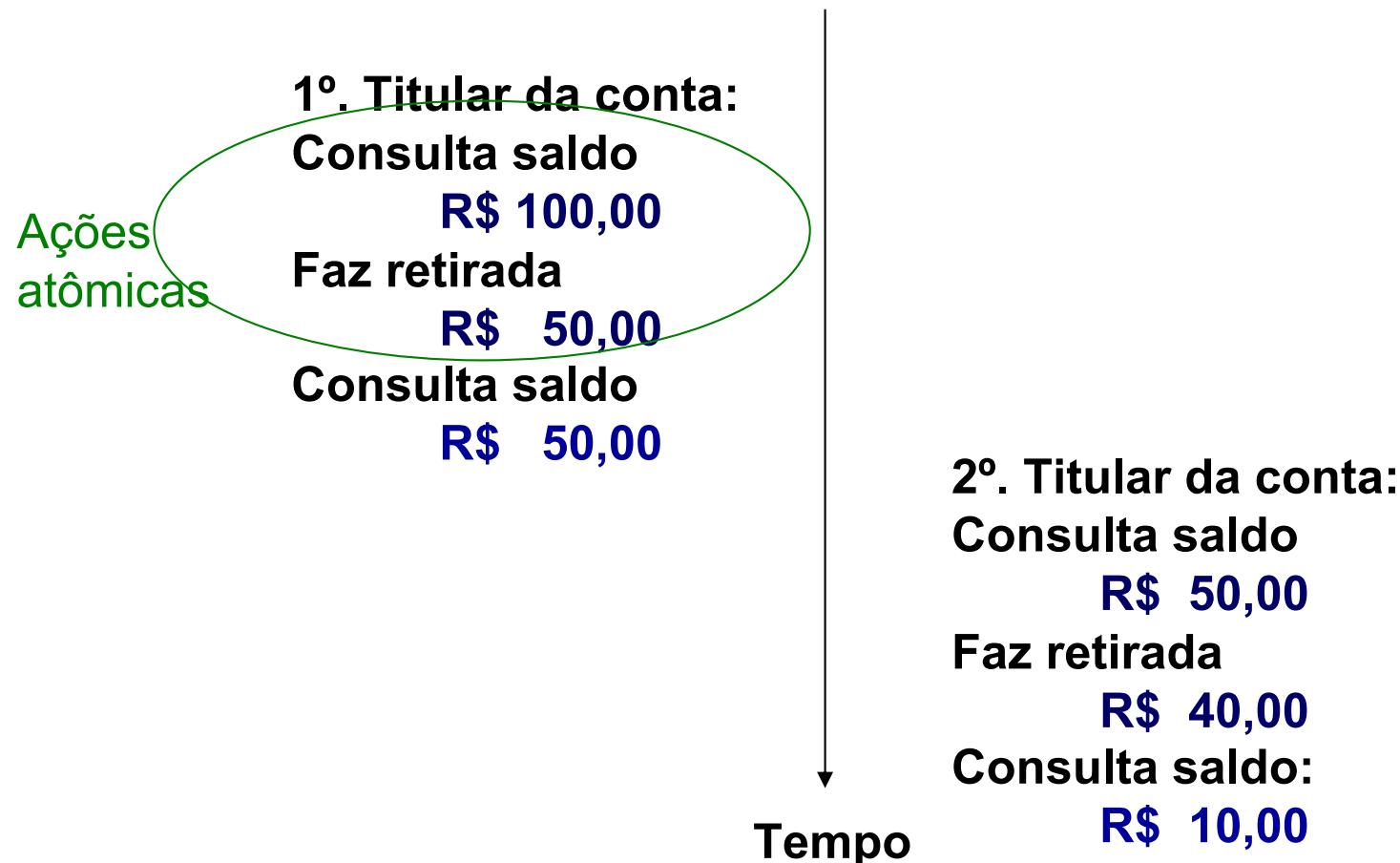
Exemplo: atualização de conta bancária conjunta



Explique o que aconteceu e indique como evitar que a conta fique negativa

# Condição de Corrida

Exemplo: atualização de conta bancária conjunta

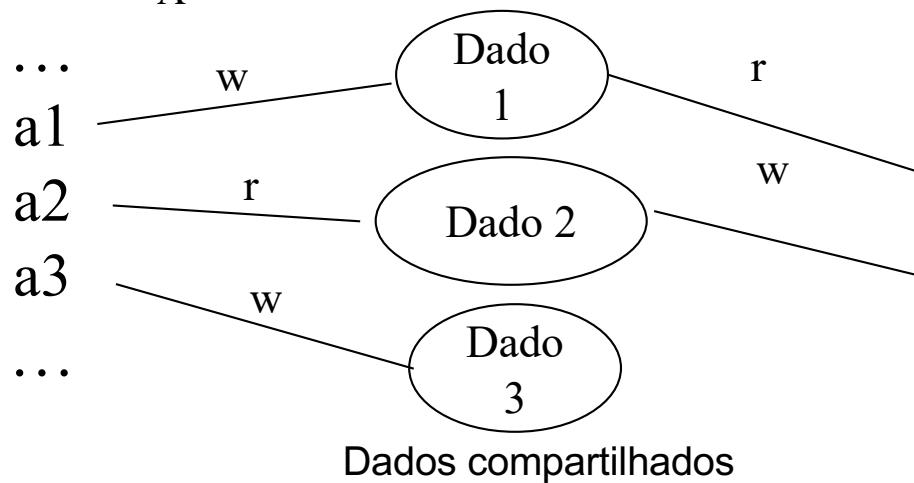


# Condição de Corrida

Ocorre sempre que...

- Cada processo/thread precisa executar uma sequência de ações ( $a_1, \dots, a_N$ ) que envolvem mais de um dado/recurso compartilhado, e
- os dados/recursos precisam manter um estado consistente entre si;
- e quando a execução de uma sequência de operações, um dos processos é interrompido pelo outro

Operações de  
Processo  $P_A$



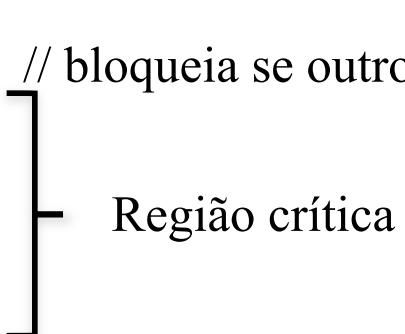
Operações de  
Processo  $P_B$

# Região Crítica

Para implementar uma região crítica deve haver um mecanismo/protocolo para garantir a entrada e saída segura (sincronizada, coordenada) nesta parte do código.

Código em um processo:

```
...
Enter-region;      // bloqueia se outro processo estiver dentro
instr 1;
instr 2;
instr 3;
...
Exit-region;       // sai da região, e libera outros processos esperando
...
```

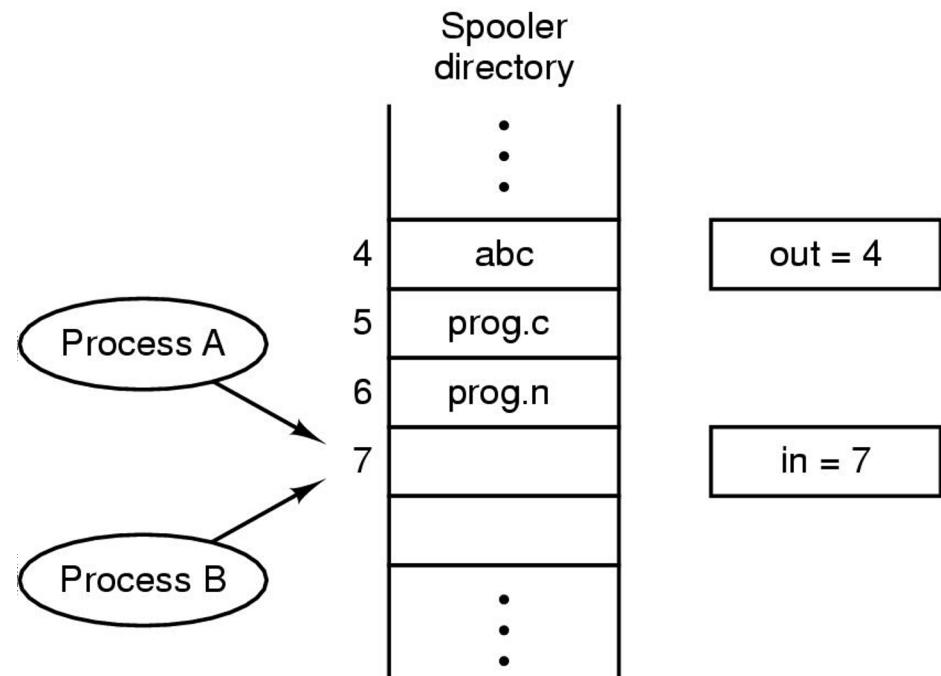


Veremos algumas possíveis abordagens e mecanismos para se garantir exclusão mútua de RCs

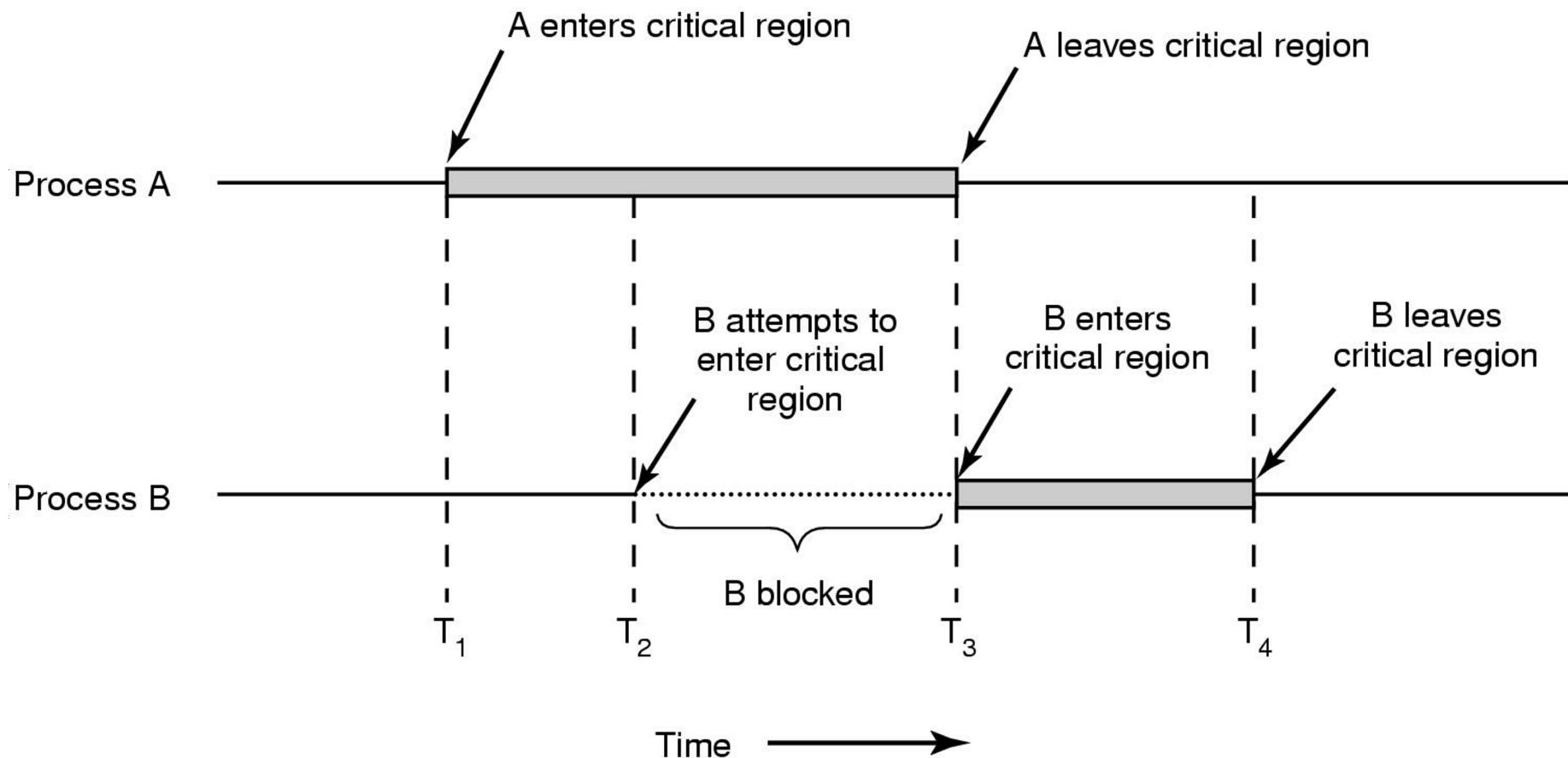
# Usando Regiões Críticas

```
Process {
...
    Enter-Region()
    copy file to Spooler[in];
    in = (in+1)%slots;
    Exit-Region()
...
}

Spooler {
    while(1) {
        Enter-Region();
        print Spooler[out];
        out = (out+1)%slots;
        Exit-Region();
    }
}
```



# Região Crítica



Exclusão mútua usando Regiões Críticas

# Formas de Implementar Regiões Críticas

## Alternativas:

1. Desabilitar interrupções de hardware
  - ➡ Pode ser feito pelo núcleo, mas não por um processo em modo usuário
2. Processos compartilham uma flag “lock”: se lock=0, trocar valor para 1 e processo entra RC, senão processo espera
  - ➡ Se leitura e atribuição do lock não for atômica, então exclusão mútua não é garantida
3. Alternância regular de acesso por dois processos (PID= 0; PID= 1)
  - ➡ Só funciona para 2 processos e apenas se a frequência de acesso ao recurso for mais ou menos a mesma.

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

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

# Região Crítica com Espera Ocupada

```
#define FALSE 0
#define TRUE 1
#define N      2           /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;               /* number of the other process */

    other = 1 - process;     /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;          /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

## Solução de Peterson:

- Variável *turn* e o vetor *interested[]* são variáveis compartilhadas
- Se dois processos PID = {0,1} executam simultaneamente *enter\_region*, o primeiro valor de *turn* será sobreescrito (e o processo correspondente vai entrar), mas *interested[first]* vai manter o registro do interesse do segundo processo

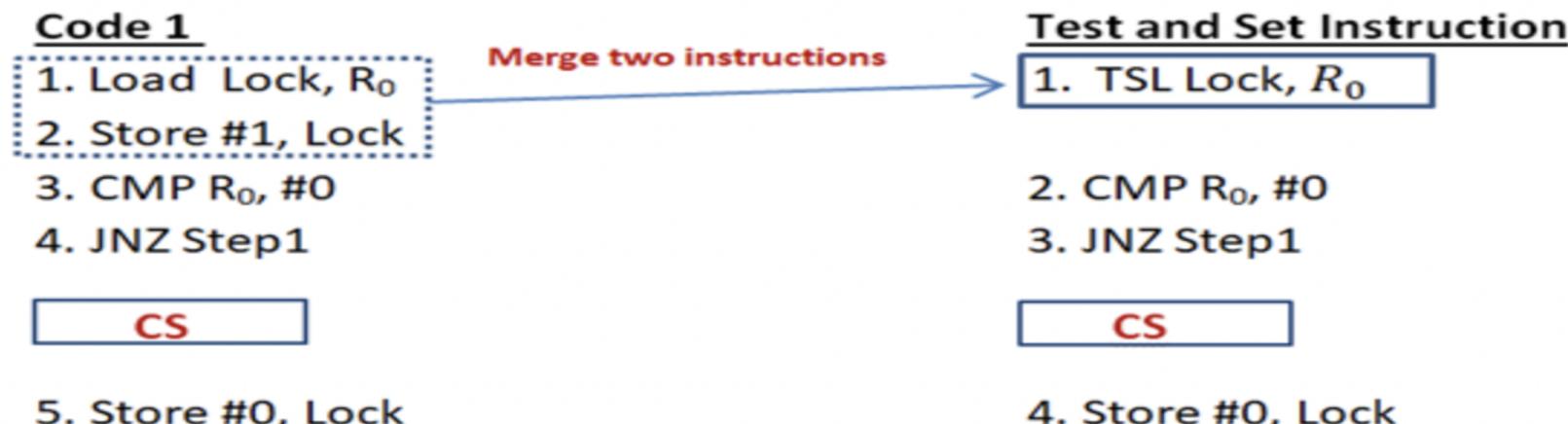
# Exclusão Mútua com Espera Ocupada - Test and Set Lock

Algumas arquiteturas possuem uma instrução de máquina especial, **Test-and-Set-Lock (TSL)** para cópia de um lock para um registrador e atribuição de um valor diferente de zero ( $\neq 0$ ) de forma atômica!

Processos que desejam entrar RC/SC executam TSL como forma de testar e setar a variável lock atomicamente:

- se registrador (e lock=0), então entram na RC, senão esperam em loop

Initially Lock = 0



# Espera Ocupada vs. Bloqueio em system calls

- Solução de Peterson e Test-and-Set-Lock têm o problema da espera ocupada que consome ciclos de processamento durante a espera `Enter_region()`
- Outro problema da espera Ocupada: Inversão de prioridades

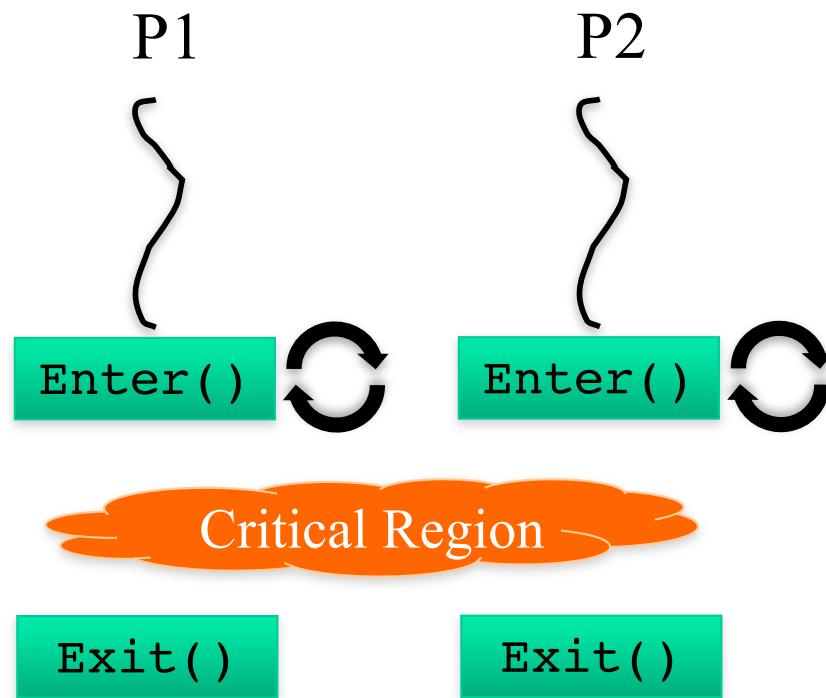
Se um processo com baixa prioridade estiver na RC, demorará mais a ser escalonado (e a sair da RC), pois processos de maior prioridade estarão executando em espera ocupada (para entrar na RC).

A alternativa: System calls que bloqueiam o processo e o fazem esperar por um sinal de outro processo

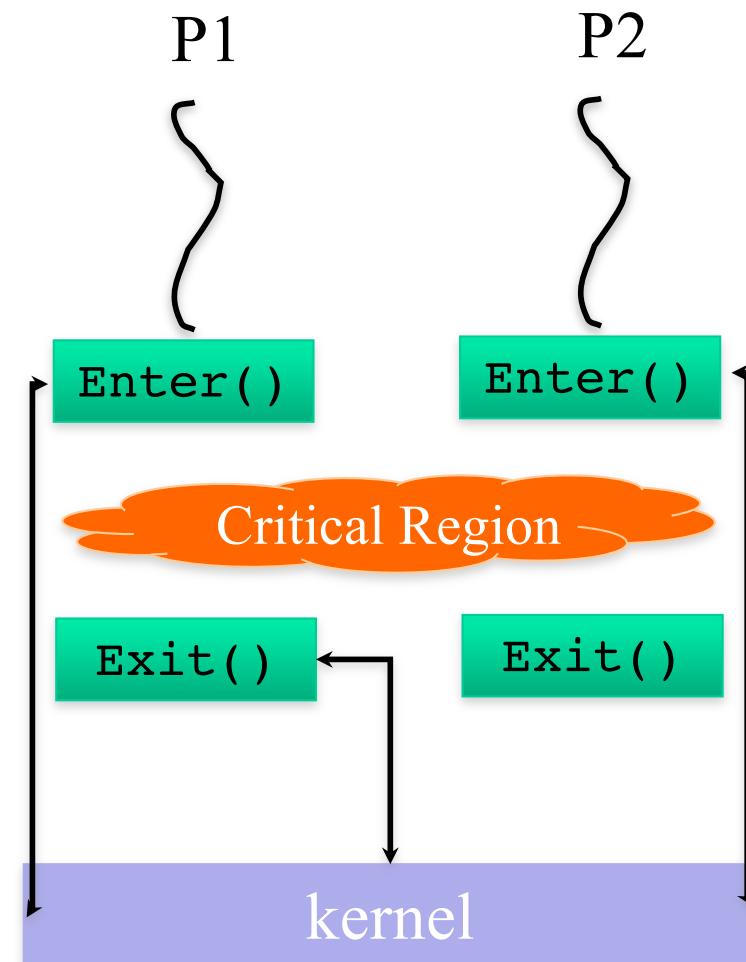
Por exemplo:

- *sleep:: suspend o processo até que seja acordado*
- *wakeup(PID):: sinal para acordar o processo PID*

# Espera Ocupada *versus* system call para o núcleo



Espera Ocupada: consome ciclos de processamento.



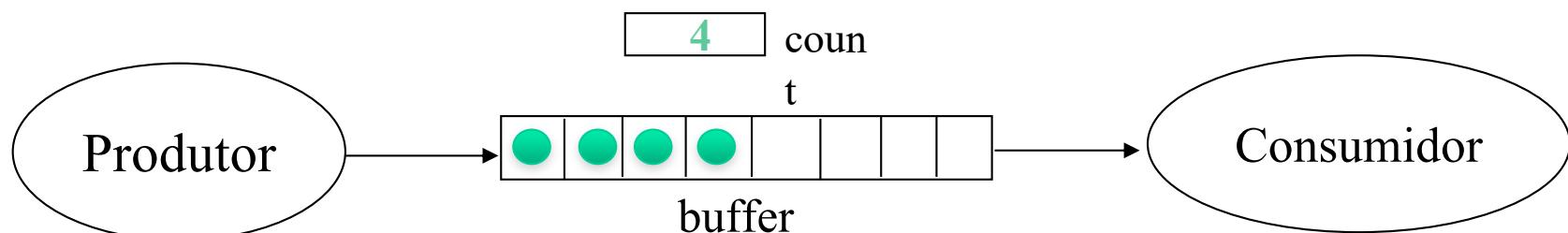
system calls:.o núcleo garante a atomicidade

# Problema do Produtor e Consumidor

Sincronização de dois processos (um consumidor, outro produtor de itens) que compartilham um buffer em uma região crítica, e que usam uma variável compartilhada *count* para controlar o fluxo de controle.

- se  $\text{count}=N$ , produtor deve esperar, e
- se  $\text{count}=0$  consumidor deve esperar,

Cada processo deve “acordar” o outro processo quando o conteúdo do buffer mudar a ponto de permitir o prosseguimento do processamento



Esse tipo de sincronização está relacionada ao estado do recurso: no caso, se itens podem ser retirados ou adicionados do buffer -> Sincronização de condição

# Problema do Produtor e Consumidor usando sleep & wakeup

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                                /* number of items in the buffer */

void producer(void)
{
    int item;

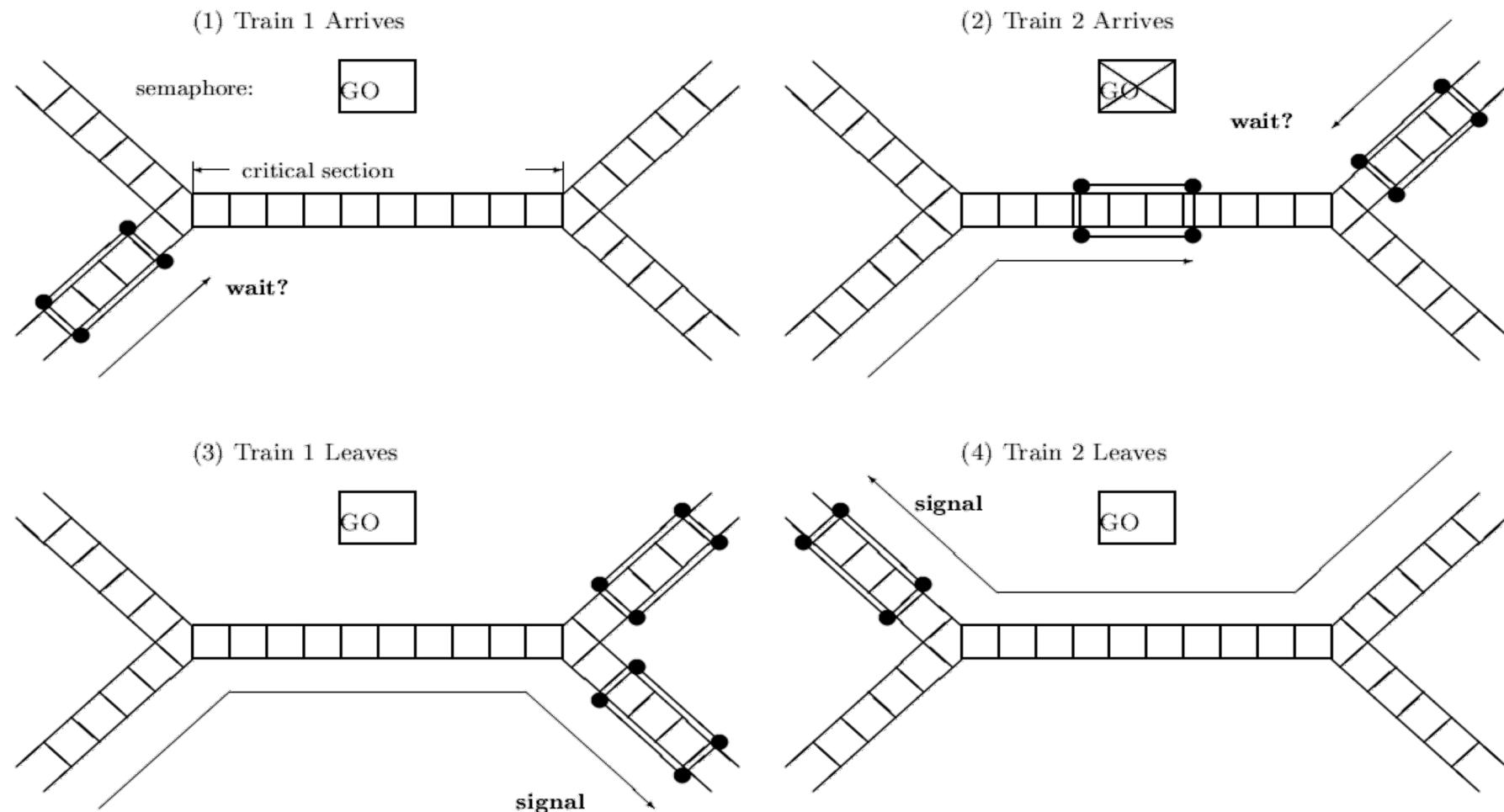
    while (TRUE) {                            /* repeat forever */
        item = produce_item();                /* generate next item */
        if (count == N) sleep();              /* if buffer is full, go to sleep */
        insert_item(item);                  /* put item in buffer */
        count = count + 1;                  /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);    /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        if (count == 0) sleep();              /* if buffer is empty, got to sleep */
        item = remove_item();                /* take item out of buffer */
        count = count - 1;                  /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);                /* print item */
    }
}
```

# Semáforos

Em 1965 E.W. Dijkstra propôs o conceito de semáforos como mecanismo básico para sincronização entre processos. A inspiração: sinais de trens.



# Semáforos

Um semáforo é um objeto que consiste de um contador, uma lista de espera de processos, e que pode ser manipulado por duas operações  $P()$  e  $V()$  (ou  $down()$  e  $up()$ )

As duas operações **são atômicas!** Não podem ser interrompidas

- No caso da exclusão mútua as instruções *down* e *up* funcionam como protocolos de entrada e saída das regiões críticas:
  - ***down/wait***: é executada quando o processo deseja entrar na região crítica. Decrementa o semáforo de 1. Se valor ficar negativo, processo bloqueia.
  - ***up/signal***: quando o processo sai da sua região crítica, incrementa o semáforo de 1, possivelmente liberando algum processo bloqueado.

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <semaphore.h>

#define THREAD_NUM 4
sem_t semaphore;

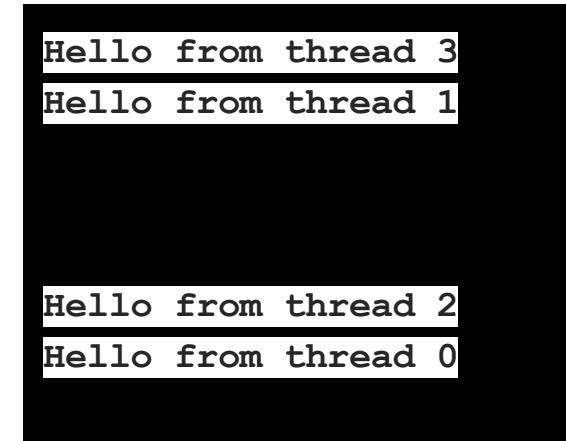
void* routine(void* args) {
    sem_wait(&semaphore);
    sleep(1);
    printf("Hello from thread %d\n", *(int*)args);
    sem_post(&semaphore);
    free(args);
}

int main(int argc, char *argv[]) {
    pthread_t th[THREAD_NUM];
    sem_init(&semaphore, 0, 2);
    int i;
    for (i = 0; i < THREAD_NUM; i++) {
        int* a = malloc(sizeof(int));
        *a = i;
        if (pthread_create(&th[i], NULL, &routine, a) != 0) {
            perror("Failed to create thread");
        }
    }
    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    sem_destroy(&semaphore);
    return 0;
}

```

## Exemplo:

4 threads executando print em paralelo a depender do valor so semáforo.



```

Hello from thread 3
Hello from thread 1
Hello from thread 2
Hello from thread 0

```

Obs: sem\_post (em pthreads) é equivalente a sem\_signal

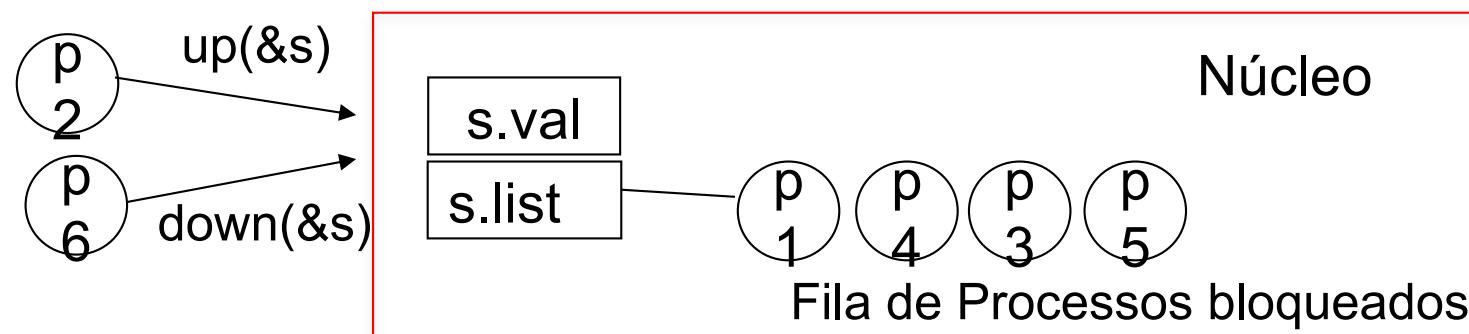
# Semáforo - Implementação

**val**: é um contador que representa o número de processos/threads que podem entrar em uma Região Crítica.

**list**: A cada semáforo está associado uma lista de processos bloqueados.

Semântica das operações:

- **down(&s)** :: decrementa s.val. Se s.val < 0 processo invocador bloqueia nesta chamada e é colocado no final da fila. Senão, continua execução
- **up(&s)** :: Incrementa s.val, desbloqueia um dos processos bloqueados (se houver) e continua execução.



Obs: down() e up() são implementadas como chamadas do núcleo (system call), e durante a sua execução o núcleo desabilita temporariamente as interrupções (para garantir a atomicidade).

# Semáforos - Implementação

## Semaphore Structure:

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```

## Down Operation

```
down (semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}
```

## Up Operation :

```
up (semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```

# Mutex: semáforos binários

Um **Mutex**, é um semáforo binário (que tem somente dois estados):

Ou seja, os valores s.val estão em [-1,0], representando “ocupado” e “livre”

E as operações recebem outro nome:

- **mutex\_lock** (é equivalente ao down)
- **mutex\_unlock** (é equivalente ao up)

Os mutexes são usados para implementar exclusão mútua simples, isto é, **onde apenas um processo pode estar na região crítica.**

Obs: mutex\_lock e mutex\_unlock precisam ser chamados sempre pela mesma thread

# Problema do Produtor e consumidor

Precisamos garantir:

1. Acesso ao buffer compartilhado em exclusão mútua
2. Que o produtor não tente inserir elementos no buffer quando este está cheio
3. Que o consumidor não tente consumir itens do buffer se este estiver vazio

Ideia:

- usar mutex para controlar o acesso exclusivo ao buffer
- semáforo empty (como contador dos campos vazios do buffer) e
- semáforo full (como contador dos campos cheios do buffer)

# Semáforos: Exemplo de uso

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

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

/\* number of slots in the buffer \*/  
/\* semaphores are a special kind of int \*/  
/\* controls access to critical region \*/  
/\* counts empty buffer slots \*/  
/\* counts full buffer slots \*/

/\* TRUE is the constant 1 \*/  
/\* generate something to put in buffer \*/  
/\* decrement empty count \*/  
/\* enter critical region \*/  
/\* put new item in buffer \*/  
/\* leave critical region \*/  
/\* increment count of full slots \*/

/\* infinite loop \*/  
/\* decrement full count \*/  
/\* enter critical region \*/  
/\* take item from buffer \*/  
/\* leave critical region \*/  
/\* increment count of empty slots \*/  
/\* do something with the item \*/

## Semáforos:

sem\_empty = contador  
de campos vazios  
sem\_full = contador de  
campos preenchidos  
mutex = exclusão mutua

# Variável de Condição

Uma variável de condição é um tipo especial de variável usada para determinar se uma determinada condição foi atendida ou não.

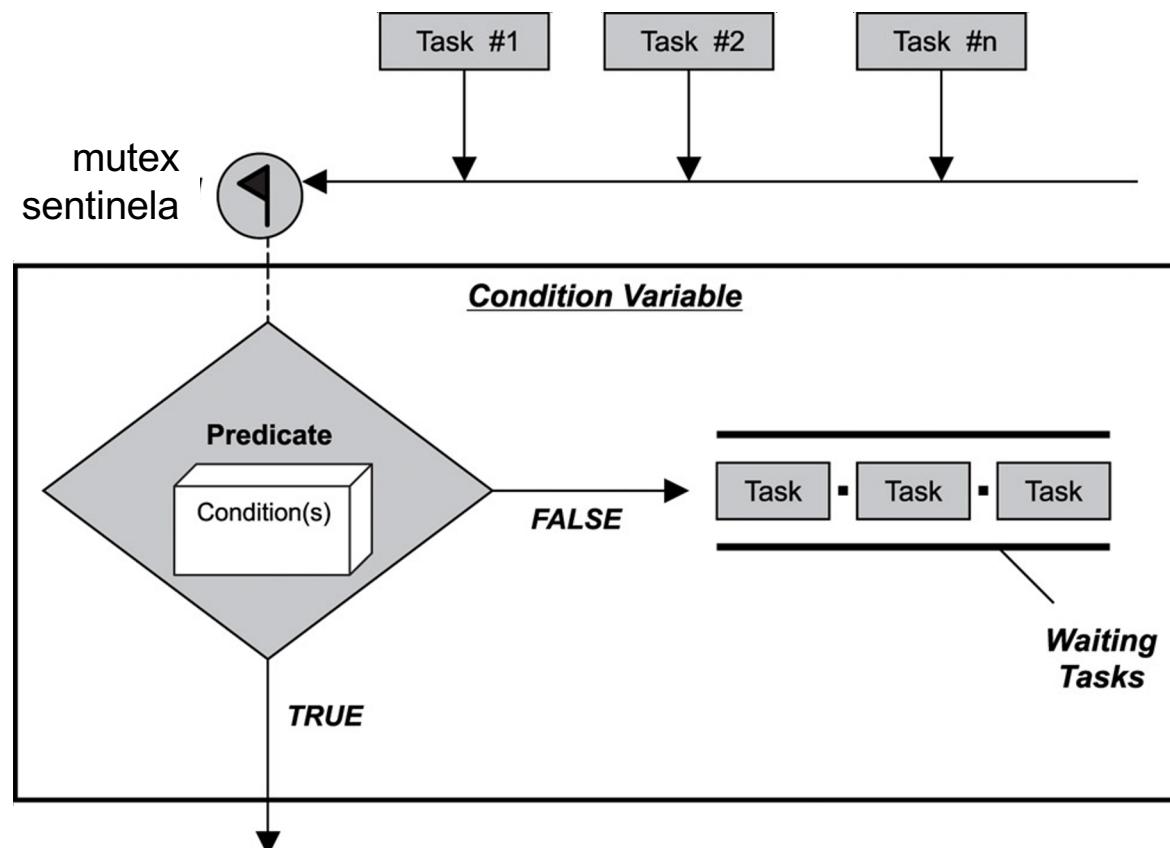
Ela é usada para comunicar threads quando determinadas condições se tornam verdadeiras.

Uma variável de condição é uma fila em que as threads (dentro de uma região crítica) se colocam quando estão esperando por uma condição que é modificável por outra thread. (**cond\_wait()**)

Quando isso acontece essa thread pode sinalizar a mudança de condição para as threads que estão esperando pela condição. (**cond\_signal()**)

# Variável de Condição

Ideia central: bloquear e esperar ao mesmo tempo para uma condição ser satisfeita e o acesso exclusivo a uma sessão crítica



Ao contrário de Semáforos, variáveis de condição não possuem um contador (não têm uma memória)

Fonte: <http://www.embeddedlinux.org.cn/>

# Variável de Condição

- Variáveis de condição sempre estão associadas a um mutex.
- Permite que threads em uma região crítica sejam suspensos até uma condição estiver satisfeita.
- Threads/processos que estejam na região crítica abrem mão do lock da exclusão mútua deixando um outro processo entrar (para que possa fazer a condição ficar satisfeita)
- Operações: **wait**, **signal** (ou **notify**) e **broadcast**

# Produtor consumidor com variáveis de condição

```
variáveis compartilhadas
cond_t not_empty, not_full;
int slots =0;
mutex_t lock;
```

Produtor:

```
Produce_one(void)
mutex_lock(&lock)
while (slots == N)
    cond_wait(&not_full, &lock);
my = get_empty(&buffer);
Fill(&buffer[my]);
slots++;
cond_signal(&not_empty);
mutex_unlock(&lock);
}
```

Consumidor:

```
Consume_one(void)
mutex_lock(&lock)
while (slots == 0)
    cond_wait(&not_empty, &lock);
my = get_filled(&buffer);
Use(&buffer[my]);
slots--;
cond_signal(&not_full);
mutex_unlock(&lock);
}
```

# Monitor – Objeto com sincronização

## Idéia central:

- Usar o princípio de encapsulamento de dados também para a exclusão mútua,
- Os dados internos do monitor só são acessados através dos **procedimentos** do Monitor;
- **A cada momento, apenas um único procedimento do monitor pode ser executado;**
- O monitor gerencia as threads esperando pelo término da execução do procedimento “em execução”

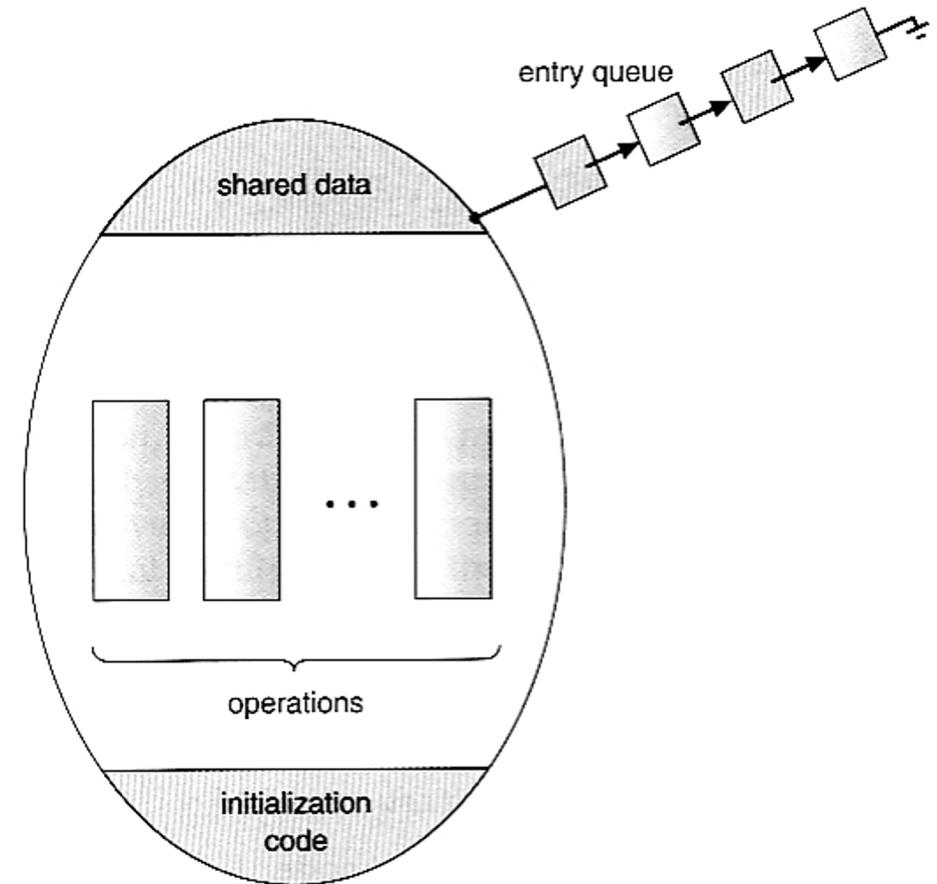


Figure 6.17 Schematic view of a monitor.

# Monitor

- É um objeto com uma coleção de procedimentos que manipulam dados compartilhados internos ao monitor.
- Existe um único mutex associado ao Monitor.
- Cada procedimento adquire o mutex.lock antes de começar a executar e faz mutex.unlock ao retornar.
- Assim, cada procedimento pode acessar os dados compartilhados do Monitor em regime de exclusão mútua.
- Variáveis de condição são usadas para bloquear procedimentos que necessitem de certa condição.

# Monitor

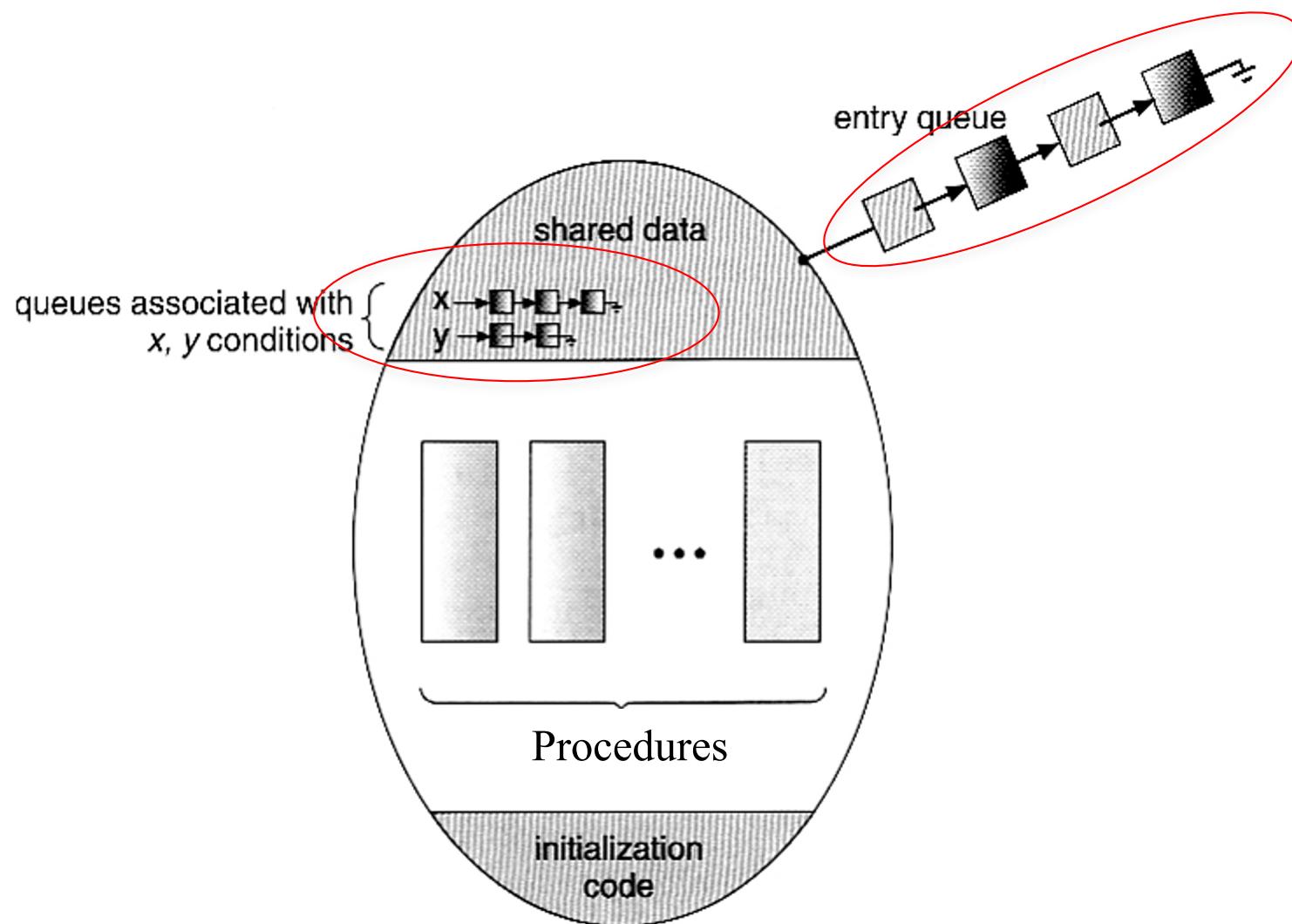


Figure 6.18 Monitor with condition variables.

# Monitor

**monitor example**

```
integer i;  
condition c;
```

```
procedure producer( );
```

```
. . .  
. wait(c)  
. . .  
end;
```

```
procedure consumer( );
```

```
. . .  
. signal(c)  
. . .  
end;
```

```
end monitor;
```

- Monitor é um elemento da linguagem de programação que combina o encapsulamento de dados com o controle para acesso sincronizado
- Usa-se variáveis de condição (com operações **wait** e **signal**), quando o procedimento em execução não consegue completar e precisa que outro procedimento seja completado;

# Monitor: Exemplo de Uso

```
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;
```

## Resolvendo o problema do produtor-consumidor com monitores

- Exclusão mútua dentro do monitor e controle explícito de sincronização garante a coerência do estado do buffer
- buffer tem  $N$  entradas

# Principal Diferença entre Monitores e Semáforos

Monitor:

- só serve para threads, já que processos não têm acesso a dados compartilhados (as instâncias de monitor)
- Monitor é um mecanismo mais abstrato, que estende o encapsulamento para a exclusão mútua

Semáforo:

- É um mecanismo de sincronização mais básico e pode ser provido por núcleos de S.O. ou runtime de uma linguagem programação (i.e. máquina virtual)
- podem ser usados por processos e threads

# Problemas Clássicos de Concorrência

- Produtor-Consumidor 
- Leitores e Escritores excludentes
- Barbeiro Dorminhoco
- Jantar dos Filósofos
- Sincronização de Barreira

# Impasse e Inanição (Starvation)

As duas situações a serem evitadas em problemas de concorrência são:

**Impasse (Definição):** Um conjunto de processos está em situação de impasse (deadlock) se cada entidade pertencente ao conjunto está bloqueada esperando por um evento (ou recurso) que somente outra entidade no mesmo conjunto pode gerar (ou liberar).

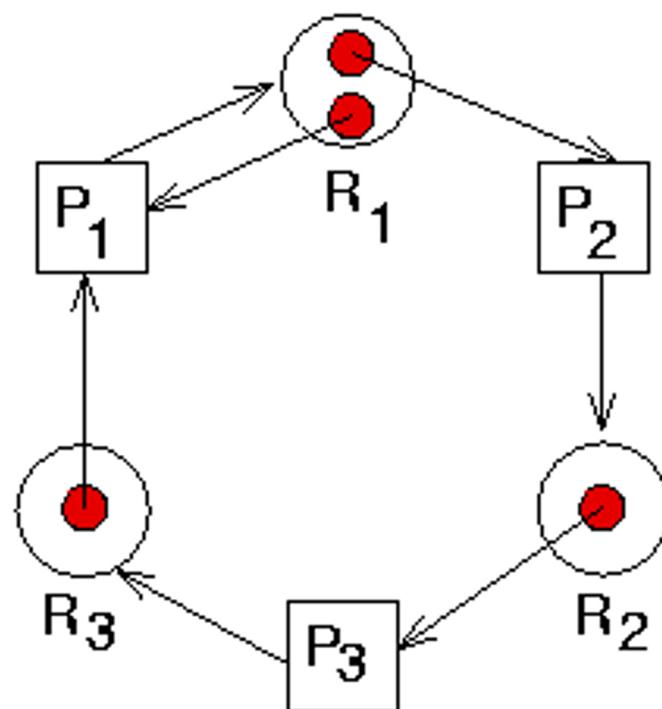
**Inanição (Definição):** Uma processo está em inanição (starvation) quando ocorrem negativas de acesso a um determinado recurso perpetuamente, impedindo que o processo execute o restante das tarefas, sem que o mesmo esteja bloqueado.

P.S.: quando o processo está com baixa prioridade ou as demandas dos outros processos são muito mais frequentes

# Grafo de Alocação de Recursos

Representação de uma situação de alocação e demanda por recursos por processos

Se houver um ciclo, então têm-se um impasse

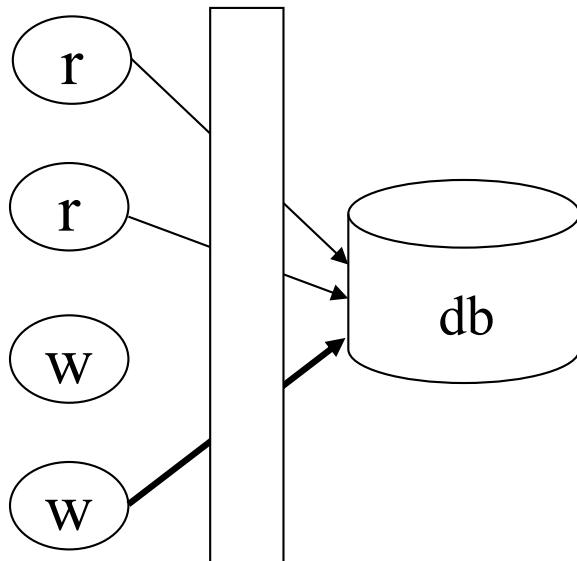


$R \rightarrow P$ : recurso R  
alocado ao processo P

$P \rightarrow R$ : processo P  
requisitando o recurso R

# Problema dos Leitores e Escritores Excludentes: solução com semáforos

Objetivo: Vários leitores podem entrar a Sessão Crítica ao mesmo tempo, mas escritores precisam executar em exclusão mútua.

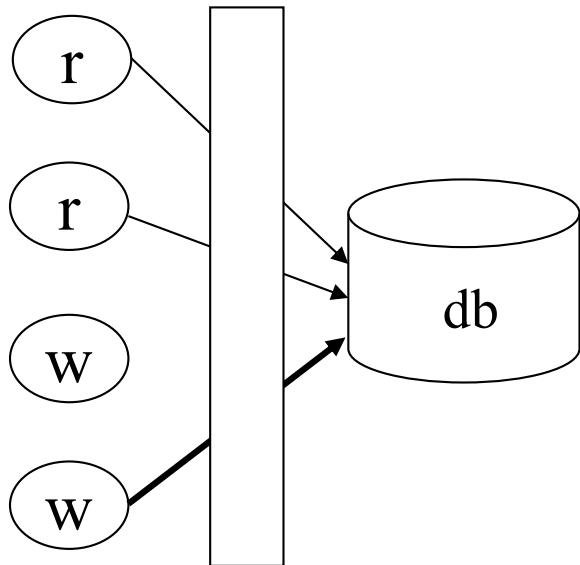


```
typedef int semaphore;           /* use your imagination */  
semaphore mutex = 1;            /* controls access to 'rc' */  
semaphore db = 1;               /* controls access to the database */  
int rc = 0;                     /* # of processes reading or wanting to */
```

```
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);  
    }  
}
```

# Problema dos Leitores e Escritores Excludentes: solução com semáforos



Há aqui um risco potencial de impasse?

Mas pode-se definir prioridades diferentes.

Nessa variante, dá-se prioridade para os leitores, que só são bloqueados se um escritor (writer) estiver acessando db.

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

/* use your imagination */
/* controls access to 'rc' */
/* controls access to the database */
/* # of processes reading or wanting to */

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();
    }
}

/* repeat forever */
/* get exclusive access to 'rc' */
/* one reader more now */
/* if this is the first reader ... */
/* release exclusive access to 'rc' */
/* access the data */
/* get exclusive access to 'rc' */
/* one reader fewer now */
/* if this is the last reader ... */
/* release exclusive access to 'rc' */
/* noncritical region */

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

/* repeat forever */
/* noncritical region */
/* get exclusive access */
/* update the data */
/* release exclusive access */
```

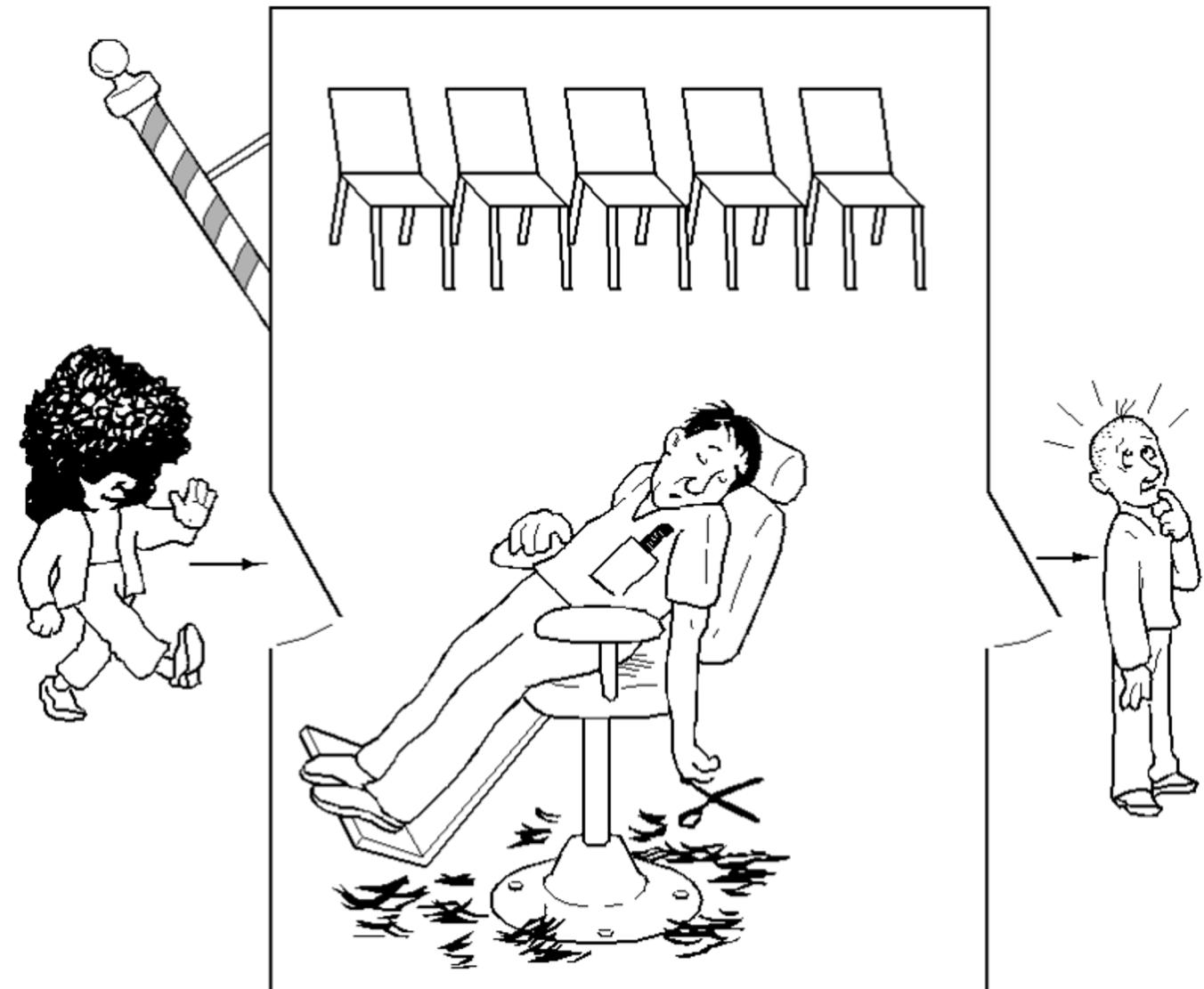
# O Problema do Barbeiro Dorminhoco

Pode haver até 5 clientes esperando pelo serviço.

Se todas cadeiras estão ocupadas, cliente segue adiante sem cortar cabelo.

Se todos os barbeiros estão ocupados, cliente espera.

Se não há clientes, barbeiro tira uma soneca, até que chega um novo cliente



# O Barbeiro Dorminhoco: usando Semáforos

```
#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 */
    }
}
```

# Barbeiro Dorminhoco usando Monitor

mon: monitor  
sleeping\_barber

## Barber code:

```
while (true)
begin
    mon.get_customer();
    cut_hair();
end;
```

## customer code:

```
mon.hair_cut();
```

## Monitor:

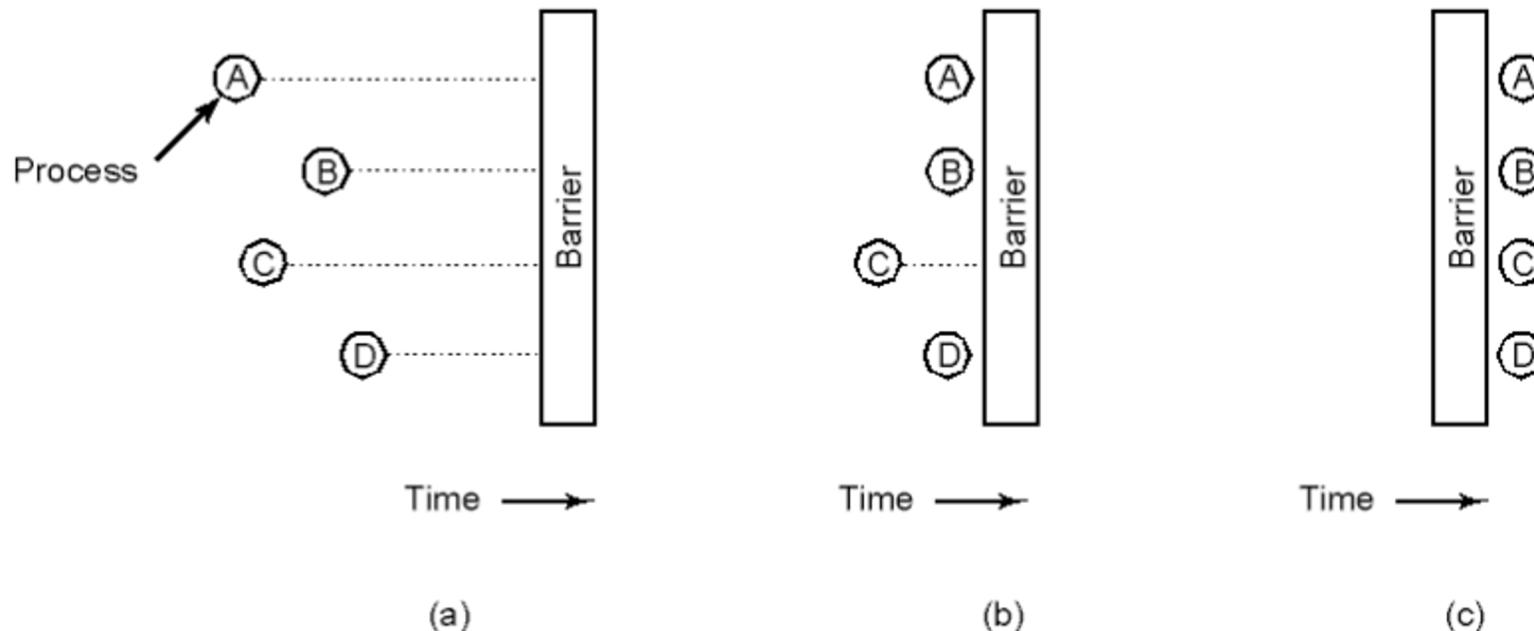
```
type sleeping_barber = monitor
    var number_of_customer: integer;
    var x_barber: condition;
    var x_customer: condition;

procedure entry get_customer()
begin
    if (number_of_customer = 0)
        x_barber.wait; /* barber has to wait for customers */
    number_of_customers := number_of_customers - 1;
    x_customer.signal; /* inform customer that the barber is ready */
end

procedure entry hair_cut()
begin
    number_of_customer := number_of_customer + 1;
    x_barber.signal; /* inform the barber that a customer is in */
    x_customer.wait; /* wait for the barber */
    do_haircut();
end

begin /*initialization*/
    number_of_customer = 0;
end
end /* monitor */
```

# Sincronização de Barreira



- Quando todos os processos precisam alcançar um mesmo estado, antes de prosseguir (exemplo: processamento paralelo “em rodadas” com troca de informações)
  - Processos progridem a taxas distintas
  - Todos que chegam a barreira, são bloqueados para esperar pelos demais
  - Quando o retardatário chega, todos são desbloqueados e podem prosseguir

# Sincronização de Barreira

Sejam N processos que precisam esperar mutuamente na barreira.

```
Process {
    bool last = false;
    Semaphore barrier;
    Mutex m;
    Int count = N
    Init (&barrier, 0)

    down(&m)
        count--;
        if (count == 0) last= true;
    up(&m)
    if (NOT last) down (&barrier); // espera pelos demais processos
    else for (i=0; i< N; i++) up (&barrier); // desbloqueia todos
    ...
}
```

# O Jantar dos Filósofos

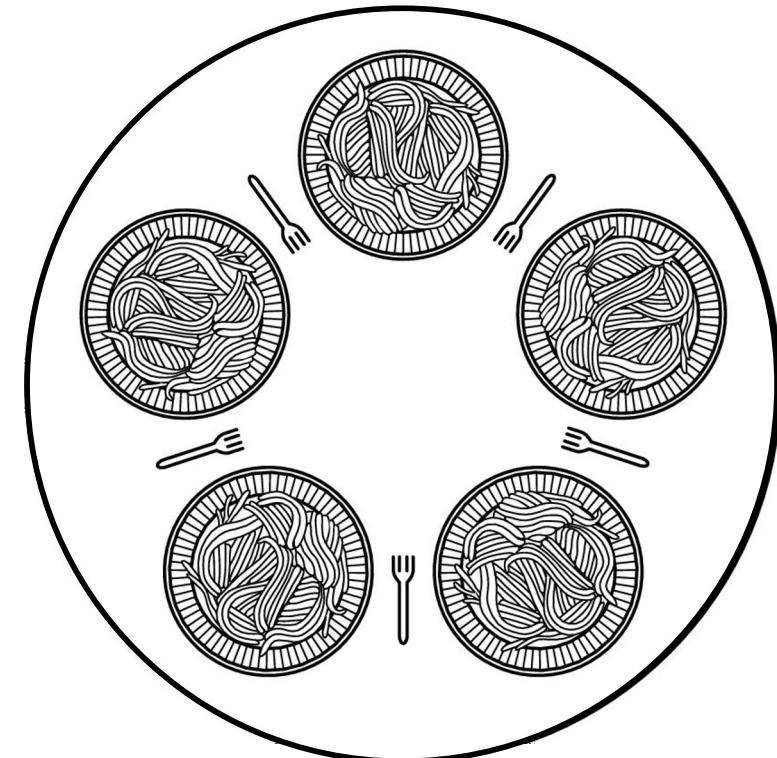
## Sincronização para compartilhamento de recursos 2 a 2

### Definição do Problema:

- Filósofos só tem 3 estados: *com\_fome*, *comendo* e *pensando*;
- Para comer, precisam de dois garfos, cada qual compartilhado com os seus vizinhos;
- Só é possível pegar um garfo por vez, isso é feito aleatoriamente.
- Assim que obtém um garfo, um filósofo não solta o mesmo

### Questões globais:

- Como evitar um impasse (deadlock)?
- Como garantir que nenhum filósofo morra de fome? (starvation)



Esse problema é considerado um clássico de sincronização em S.O., mesmo não ocorrendo exatamente assim no mundo real.

Motivo: exemplifica muitas situações em que os processos precisam adquirir mais de um recurso ao mesmo tempo, recursos compartilhados com outros processos

# O Jantar dos Filósofos

Uma solução trivial:

- Se há  $N$  recursos (garfos), deixar que no máximo  $N-1$  filósofos sentem à mesa, e para isso, usar um semáforo contador, “mesa”, e inicializá-lo com o valor  $N-1$ .
  - Então, mesmo que filósofos vizinhos disputem um garfo, os filósofos das pontas sempre conseguirão comer, devolverão os garfos, e seus vizinhos conseguiram comer, etc.
- O sistema fica livre de impasses

# O Jantar dos Filósofos

## Tentativa 1:

```
#define N 5                                     /* number of philosophers */  
  
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */  
{  
    while (TRUE) {  
        think();  
        take_fork(i);  
        take_fork((i+1) % N);  
        eat();  
        put_fork(i);  
        put_fork((i+1) % N);  
    }  
}
```

Cada filósofo tenta pegar primeiro o garfo esquerdo, e depois o garfo direito.

Problema: Se todos pegarem o esquerdo ao mesmo tempo, teremos um impasse?

**Tentativa 2:** Aguarde até obter garfo esquerdo; Se garfo direito estiver disponível, ok, senão devolve também o garfo esquerdo e volta a esperar pelo garfo esquerdo.

Qual é o problema?

# Jantar dos Filósofos

```
#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;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

/\* number of philosophers \*/  
/\* number of i's left neighbor \*/  
/\* number of i's right neighbor \*/  
/\* philosopher is thinking \*/  
/\* philosopher is trying to get forks \*/  
/\* philosopher is eating \*/  
/\* semaphores are a special kind of int \*/  
/\* array to keep track of everyone's state \*/  
/\* mutual exclusion for critical regions \*/  
/\* one semaphore per philosopher \*/  
  
/\* i: philosopher number, from 0 to N-1 \*/  
  
/\* repeat forever \*/  
/\* philosopher is thinking \*/  
/\* acquire two forks or block \*/  
/\* yum-yum, spaghetti \*/  
/\* put both forks back on table \*/

Solução (parte 1)

# Jantar dos Filósofos

Solução (parte 2): usar um vetor state[] para verificar o estado dos vizinhos e dos vizinhos dos vizinhos.

```
void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)                                     /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

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]);
    }
}
```

# Jantar dos Filósofos: outras soluções baseadas em prioridades

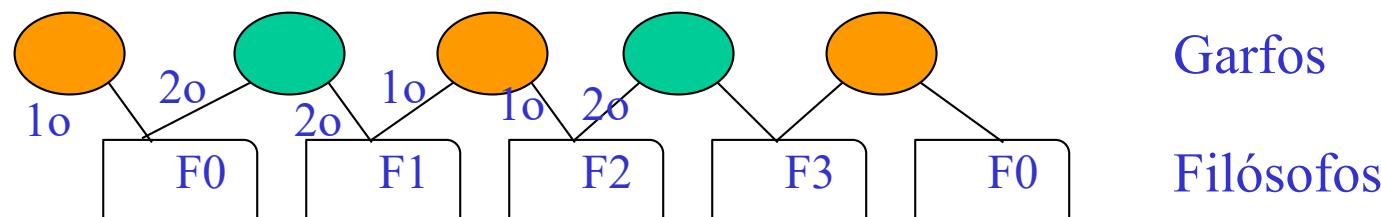
## Solução de Dijkstra:

- Estabelecer uma ordem de prioridade dos garfos ( $G_1, G_2, G_3, \dots, G_N$ ) e impor que os filósofos sempre peguem primeiro o garfo com menor ID – assim quebra-se a dependencia circular! E garante-se que pelo menos um filósofo será capaz de comer.

# Jantar dos Filósofos: Outra Solução

Ideia Central: Diferenciar o comportamento de filósofos pares e ímpares:

- os filósofos pares tentam pegar primeiro o garfo esquerdo, e depois o direito
- os filósofos ímpares tentam pegar primeiro o garfo direito e depois o esquerdo
- assim, os garfos (à direita do filósofo par) serão sempre os primeiros a serem tentados e nunca dois filósofos vizinhos diretos entrariam em um impasse



**Exemplo:**

F0 par obtém garfo à sua esquerda  
F1 ímpar obtém garfo à sua direita  
ou F0 ou F1 obtém o seu segundo garfo  
.... (isso vale para todos os vizinhos)

# Jantar dos Filósofos: Outras Soluções

A **solução do garçom** oferece uma maneira simples de resolver o problema do Jantar dos Filósofos, pressupondo uma entidade externa chamada *garçom*.

- Estratégia:
  - Cada filósofo deve solicitar cada um dos 2 garfos (compartilhados) a um garçom, que pode recusar a solicitação para evitar um impasse.
  - Por conveniência, presumimos que todos os filósofos solicitem primeiro o garfo esquerdo e depois o garfo direito.
  - O garçom sempre fornece os garfos quando solicitado , *a menos que* apenas um garfo não seja utilizado. Nesse caso, o garçom atende ao pedido somente se for solicitado um garfo *direito*; os pedidos de garfo esquerdo são *adiados* até que outro filósofo termine de comer.
- Não haverá um *impasse*: O último garfo só será atribuído se o garçom tiver certeza de que pelo menos um filósofo pode terminar de comer (liberando os seus 2 garfos). Portanto, a "espera circular" necessária para o deadlock não pode ocorrer.
- Desvantagens: Escalabilidade (o garçom pode se tornar um gargalo se o número de processadores for grande).

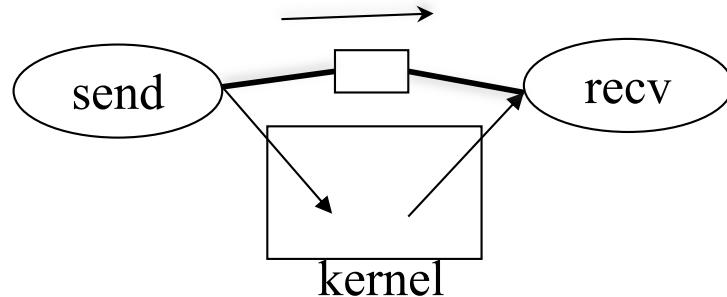
# Envio de Mensagens

- Ao contrário de mecanismos de sincronização, envio de mensagens é um mecanismo de sincronização mais sofisticado, pois:
  - Permite também a troca/transmissão de dados
  - processos podem estar executando na mesma máquina ou em máquinas distintas
  - Qualquer solução de concorrência usando semáforos pode ser resolvida por envio de mensagens
  - considere:  $\text{down}() \cong \text{receive}()$  e  $\text{up}() \cong \text{send}()$ , e o valor do semáforo sendo o número de mensagens
- Decisões de projeto do mecanismo:
  - `receive()` sempre bloqueia se o outro processo não tiver executado o `send()`
  - Mas `send()` pode ser **assíncrono** (com buffer de mensagens) ou **síncrono (Rendezvous)**
  - Em comunicação do tipo *Rendezvous* podem ocorrer impasses
  - Comunicação nem sempre é confiável: quando através da rede, mensagens podem ser perdidas:
    - são necessárias confirmações (acks) e re-transmissões

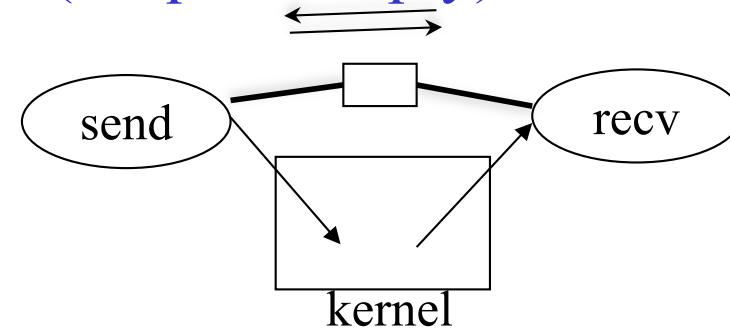
# Tipos de Envios de Mensagem

Entre processos co-localizados

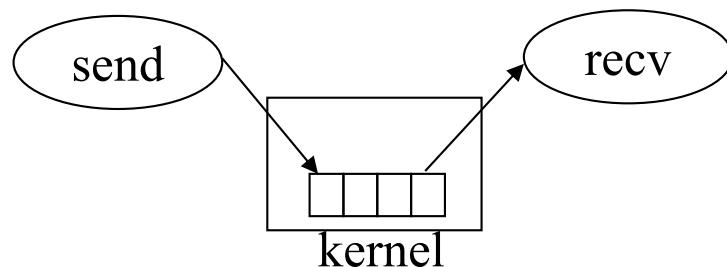
*Rendezvous, unidirecional*



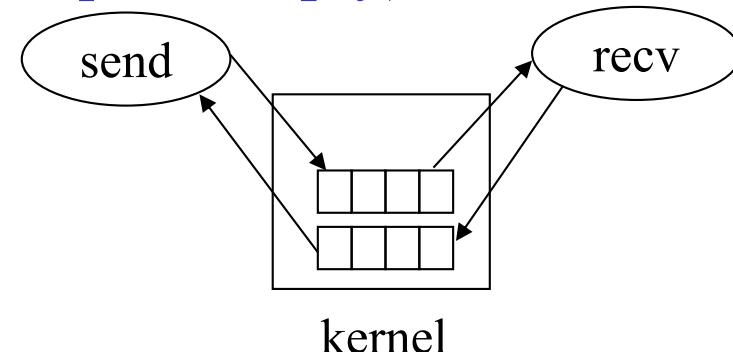
*Rendezvous, bidirecional  
(Request-Reply)*



*Comunicação assíncrona,  
unidirecional*



*Comunicação bidirecional,  
(Request-Reply) assíncrono*



# Problema do Produtor-Consumidor com envio de N mensagens

Idéia: Para a sincronização de condição, o consumidor envia mensagem vazia, e produtor responde com a mensagem preenchida.

```
#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 */
    }
}
```