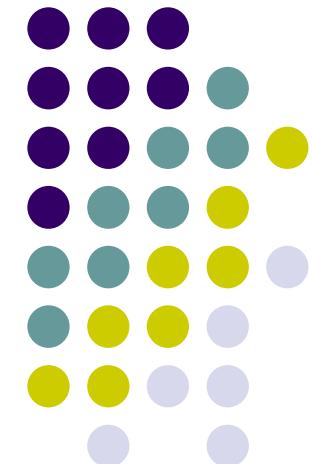
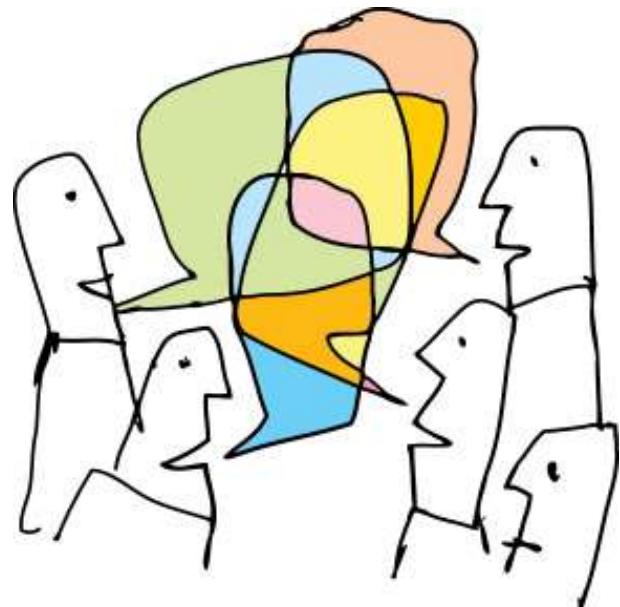
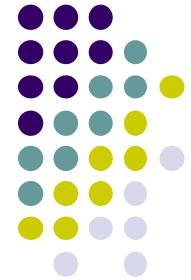


Sistemas de Computação

Comunicação entre Processos



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:

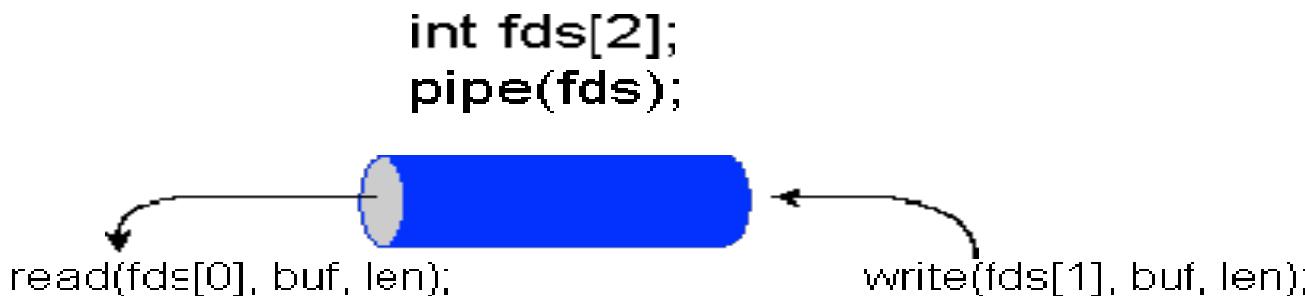
- sincronismo de atividades
- troca de dados

Comunicação Inter-processos (IPC)



Exemplos:

- Processo A e B trocam dados através de um duto (pipe): processo leitor bloqueia até que o outro processo tenha escrito algum dado no pipe;

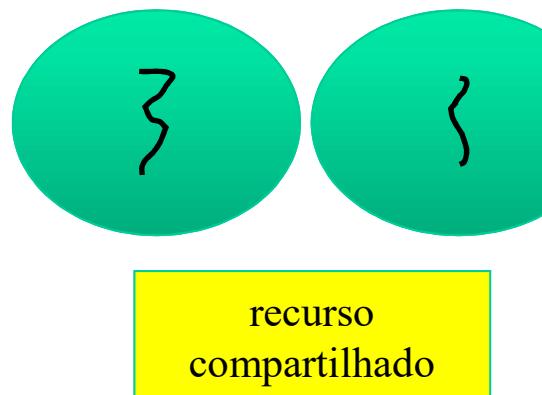


- Dois ou mais processos precisam ler e escrever no mesmo arquivo;
- Jobs de impressão de dois processos devem ser executados de forma atômica, para garantir que as saídas (listagens) não saiam misturadas
- Processos compartilham uma lista (ou vetor) de elementos com escrita: atualização requer escritas combinadas em vários endereços de memória



Comunicação Inter-processos (IPC)

- Processos (e threads) são entidades independentes, que podem ser executados em qualquer ordem.
- 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

Comunicação e Sincronização entre Processos: Duas Abordagens

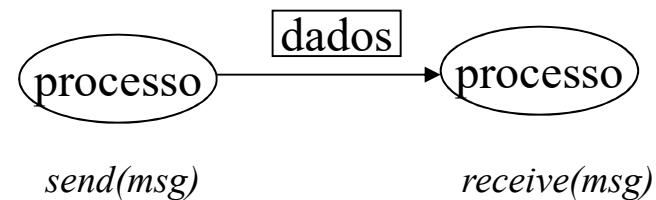
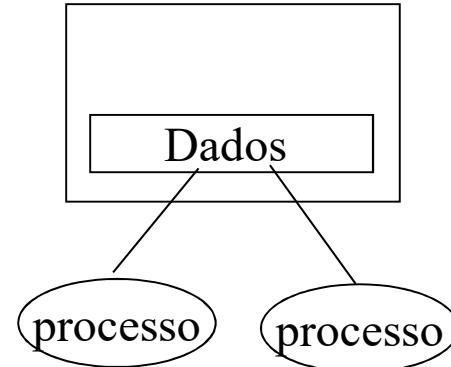


1. Baseada em memória compartilhada

- Assume que processos/threads conseguem escrever & ler em memória compartilhada
- Comunicação é implícita (através do compartilhamento) mas ...
- Sincronização precisa ser explícita

2. Baseada em troca de mensagens

- Comunicação é explícita;
- Sincronização é implícita



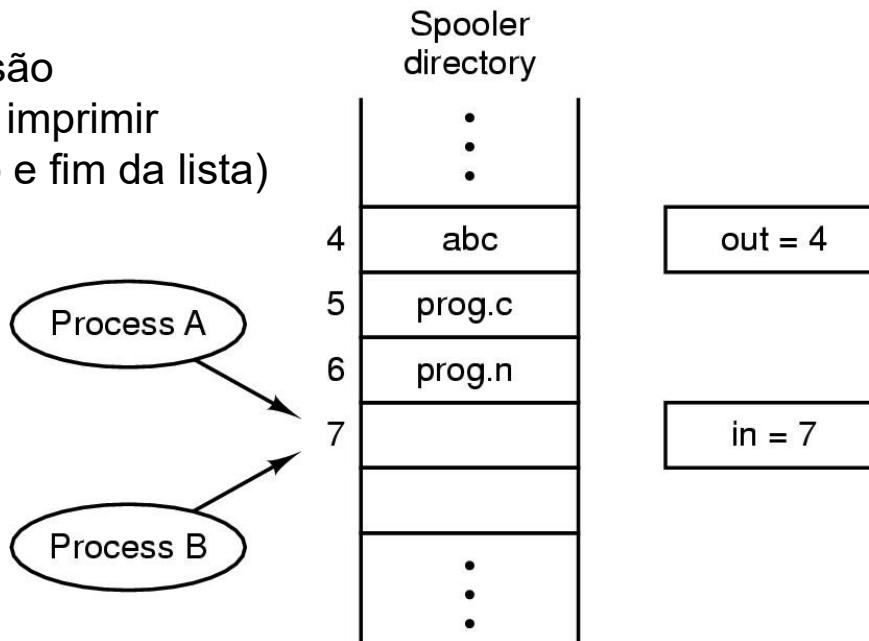
OBS: Na comunicação entre processos (Inter-Process Communication - IPC), o principal problema são as **condições de corrida**.



IPC: Condição de Corrida

Ex: Spool de impressão

Lista de processos a imprimir
(ponteiros para início e fim da lista)



Dois processos querem acessar memória compartilhada “ao mesmo tempo” (e de forma concorrente e imprevisível)

Exemplo:

- processo A lê memória compartilhada “in=7”, e logo depois é interrompido,
- Processo B faz o mesmo e adiciona um novo arquivo no diretório de spool de impressão
- Quando A re-inicia , sobre-escreve o slot 7 com seu arquivo.

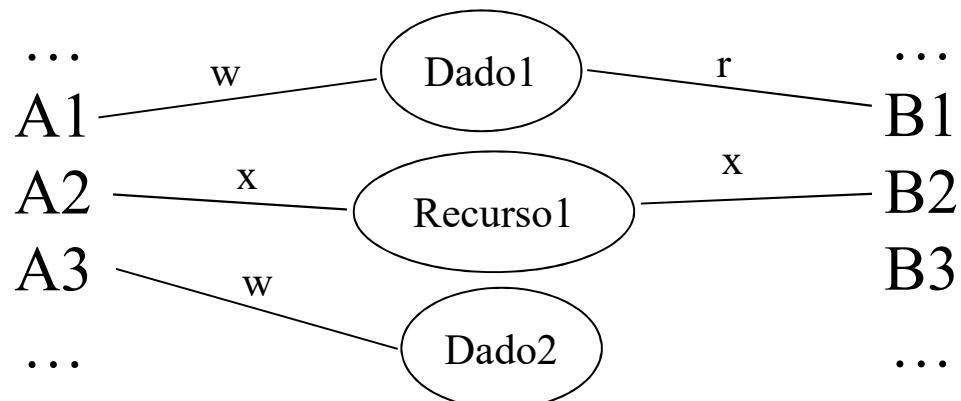


Condição de Corrida

Ocorre sempre que:

- Existem dois ou mais processos concorrentes;
- Cada processo precisa executar um conjunto de ações (a_1, \dots, a_N) que envolvem mais de um dado / recurso compartilhado;
- os dados/recursos precisam manter um estado consistente entre si;
- antes que complete a execução de todo o conjunto de ações, um dos processos é interrompido pelo outro.

Processo1:



Processo2:

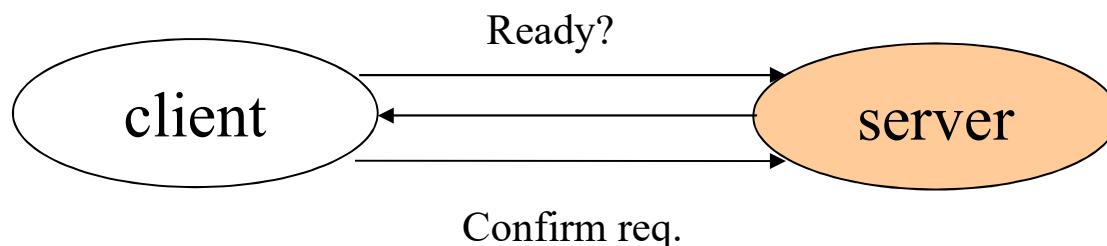


Condição de Corrida

Comportamento análogo vale para troca de mensagens
(entre processos clientes e processos servidores)

Exemplos:

1. Para requisitar um serviço, cliente precisa enviar duas mensagens (1.consulta ao estado, e 2.confirmar requisição do serviço)
2. Só faz sentido confirmar a requisição, se outro serviço (complementar ou anterior) já tiver sido completado.



Condição de Corrida

exemplo do dia-a-dia: atualização de conta bancária conjunta



1º. Titular da conta:

Consulta saldo

R\$ 100,00

Faz retirada

R\$ 50,00

Consulta saldo

R\$ -30,00 ???

2º. Titular da conta:

Consulta saldo

R\$ 100,00

Faz retirada

R\$ 80,00

Consulta saldo:

R\$ -30,00 ???

Tempo

O que aconteceu? Como evitar que a conta fique negativa?

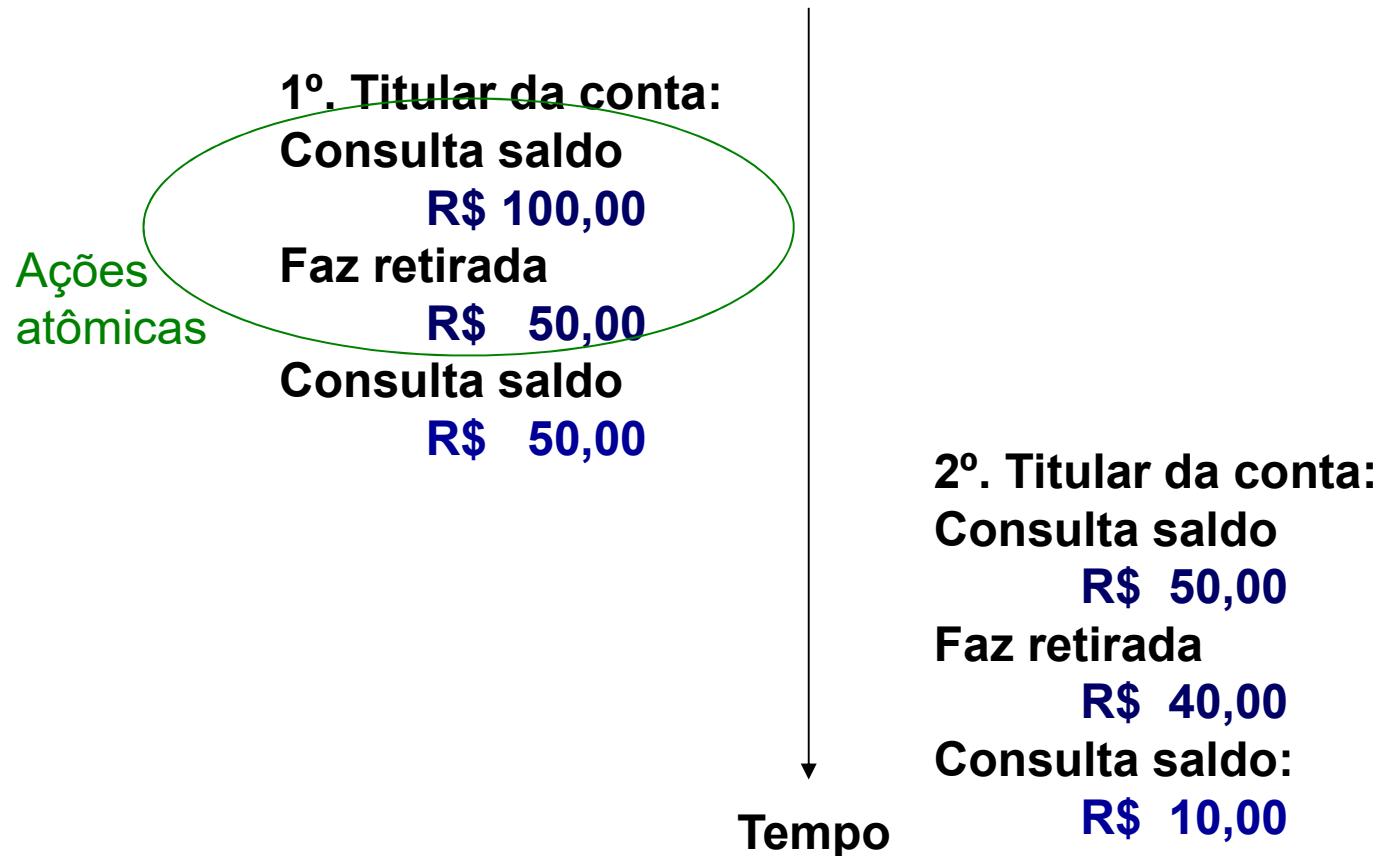
Condição de Corrida

exemplo do dia-a-dia: atualização de conta bancária conjunta



Condição de Corrida

exemplo do dia-a-dia: atualização de conta bancária conjunta



Condição de Corrida

Problemas associados



1. Ausência de atomicidade das ações feitas em dados compartilhados (requer exclusão mútua ou bloqueio)
2. Processos tentam acessar dados compartilhados que ainda *não estão prontos* para serem acessados
3. Operações simultâneas (não previstas) se bloqueiam mutuamente

→ Para permitir uma cooperação correta entre processos é preciso:

- Estabelecer um controle na ordem de execução
- Garantir que algumas execuções ocorram de forma atômica

Região Crítica



Memória/Recursos compartilhados deveriam ser acessados em regime de exclusão mútua (um processo de cada vez)

Região crítica (ou Sessão crítica) é a parte do programa em que estão as ações que manipulam os recursos/dados compartilhados.



Região Crítica

Quatro condições para garantir exclusão mútua:

1. Nunca, dois ou mais processos executam simultaneamente em suas sessões críticas
2. Não deve haver qualquer suposição sobre velocidades e/ou número de processos
3. Quando executa código fora de uma sessão crítica, um processo nunca bloqueia outro processo
4. Qualquer processo que entrou em sua sessão crítica, em algum momento deixa a mesma.



Região Crítica

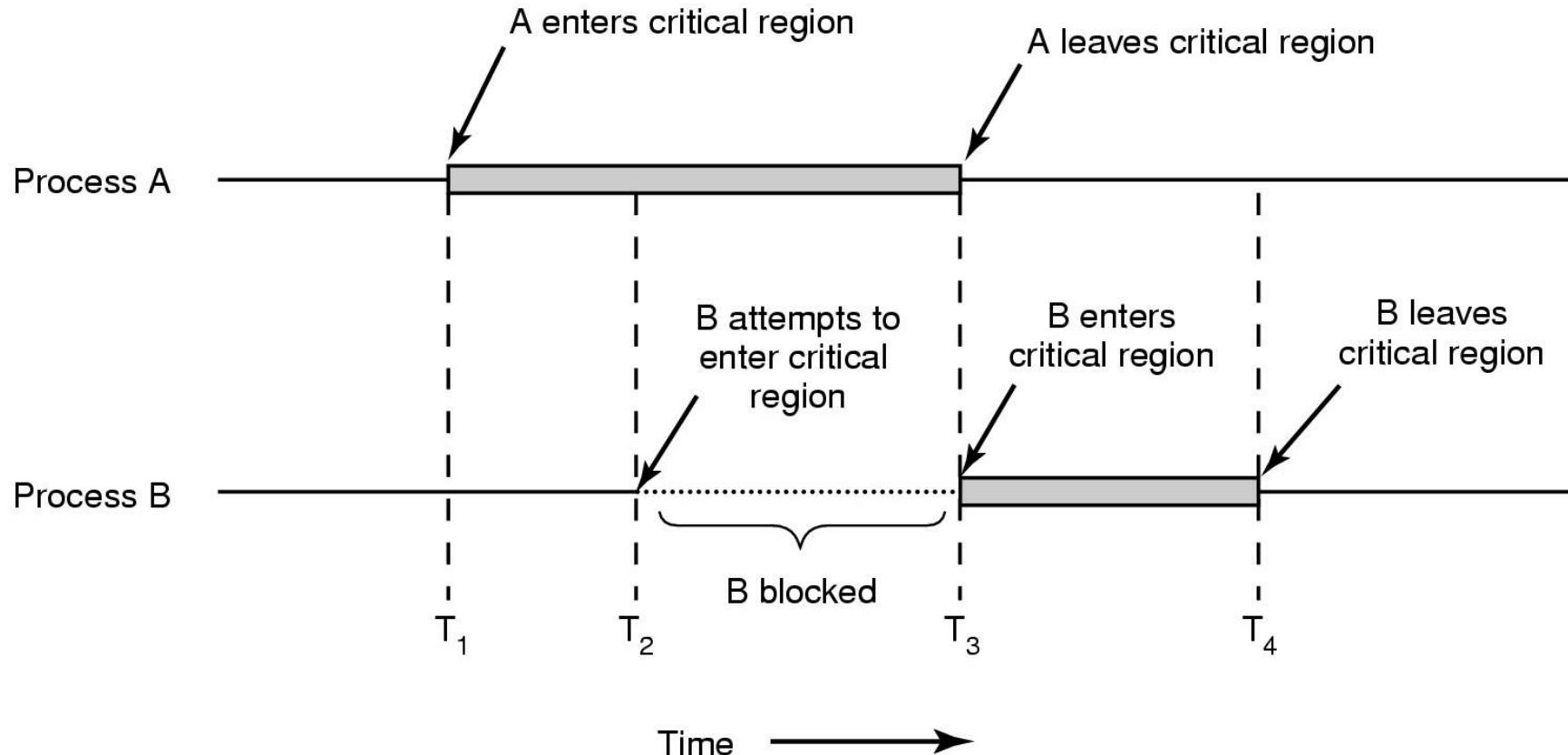
Para implementar uma região crítica deve haver um mecanismo/protocolo para garantir a entrada e saída segura (sincronizada, coordenada) desta parte do código.

Código em um processo:

```
...
Enter_region;    // fica bloqueado se outro processo
                    // estiver dentro da região crítica
A1;
A2;
A3;
Exit_region;    // sai da região, e libera recursos para outros
                    // processos que estão esperando
...
```



Região Crítica



Exclusão mútua em Regiões Críticas

Veremos a seguir algumas possíveis abordagens e mecanismos para solucionar o problema

Exclusão Mútua com Espera ocupada (CPU fica trabalhando - Busy Waiting)



Possibilidades:

- Desabilitar/habilitar interrupções:
 - ➔ Pode ser usado em modo supervisor, mas não em modo usuário
- Usar uma flag “lock” compartilhada:
 - Se lock=0, trocar valor para 1 e processo entra na Região Crítica, senão processo espera
 - ➔ Se leitura & atribuição do lock não for atômica, então o problema permanece
- Alternância regular de acesso por dois processos (PID= 0; PID= 1)
 - ➔ É um problema, se os processos alternantes requisitam o recurso com alta frequência
 - ➔ E se tem velocidades diferentes? Que efeito provoca?

Exclusão Mútua com Espera ocupada (CPU fica trabalhando - Busy Waiting)



- Exemplo: algoritmo de Dekker (1^a abordagem)

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

(a)

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

(b)

Garante a exclusão mútua, porém gera dois problemas:

- **Processos se alternam no uso de suas respectivas Regiões Críticas, o tempo de execução será ditado pelo processo mais lento.**
- **Se um dos processos falhar (abortar por exemplo) o outro jamais poderá entrar em sua Região Crítica novamente.**

Exclusão Mútua com Espera ocupada (CPU fica trabalhando - Busy Waiting)



Algoritmo de Dekker – 2a abordagem

```
int flag[2]; /*variável global inicializada com {0,0}*/
```

Processo 0

```
. . .  
while flag[1] do { };  
/**/  
flag[0] = TRUE;  
<RC>  
flag[0] = FALSE;
```

Processo 1

```
. . .  
while flag[0] do { };  
/**/  
flag[1] = TRUE;  
<RC>  
flag[1] = FALSE;
```

Se os processos forem interrompidos antes da atribuição de flag como TRUE, poderá ocorrer algum problema de execução?

Sim, se interromper antes da atribuição do flag... // não é garantida a EXCLUSÃO MÚTUA!**

Exclusão Mútua com Espera ocupada (CPU fica trabalhando - Busy Waiting)



Algoritmo de Dekker – 3a abordagem

```
int flag[2]; /*variável global */
```

Processo 0

```
flag[0] = TRUE;  
/***/  
while flag[1] do { };  
<RC>  
flag[0] = FALSE;
```

Processo 1

```
flag[1] = TRUE;  
/***/  
while flag[0] do { };  
<RC>  
flag[1] = FALSE;
```

Se os processos forem interrompidos após a atribuição de flag como TRUE, poderá ocorrer algum problema de execução?

Sim, se interromper após a atribuição do flag...

//
pode causar DEADLOCK, ninguém entra na região crítica!**

Exclusão Mútua com Espera ocupada (CPU fica trabalhando - Busy Waiting)



Seja o Algoritmo de Dekker – 3a abordagem – mostrado abaixo:

```
int flag[2]; /*variável global */
```

Processo 0

```
.  
flag[0] = TRUE;  
/**/  
while flag[1] do { };
```

```
<RC>  
flag[0] = FALSE;
```

Processo 1

```
.  
flag[1] = TRUE;  
/**/  
while flag[0] do { };
```

```
<RC>  
flag[1] = FALSE;
```

Se os processos forem interrompidos após a atribuição de flag como TRUE, poderá ocorrer algum problema de execução? Explique.

Sim, se interromper após a atribuição do flag... // pode causar DEADLOCK, ninguém entra na região crítica!**

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 */
}
```

Loop!!!

Solução de Peterson (outra abordagem por software):

- *turn* e 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 (abordagem por hardware)



TSL (Test-and-Set-Lock) = instrução de máquina atômica para leitura de um lock e armazenamento de um valor $\neq 0$

Processos que desejam entrar na Região Crítica executam TSL:

- se lock=0. Entram na RC, senão esperam em loop

enter_region:

TSL REGISTER,LOCK	copy lock to register and set lock to 1
CMP REGISTER,#0	was lock zero?
JNE enter_region	if it was non zero, lock was set, so loop
RET return to caller; critical region entered	

leave_region:

MOVE LOCK,#0	store a 0 in lock
RET return to caller	

É possível executar o processo A duas vezes?

Exclusão Mútua com Espera Ocupada (abordagem por hardware)



TSL (Test-and-Set-Lock) = instrução de máquina atômica para leitura de um lock e armazenamento de um valor $\neq 0$

Processos que desejam entrar na Região Crítica executam TSL:

- se lock=0. Entram na RC, senão esperam em loop

enter_region:

TSL REGISTER,LOCK	copy lock to register and set lock to 1
CMP REGISTER,#0	was lock zero?
JNE enter_region	if it was non zero, lock was set, so loop
RET return to caller; critical region entered	

leave_region:

MOVE LOCK,#0	store a 0 in lock
RET return to caller	

É possível executar o processo A duas vezes? SIM



Espera Ocupada vs. Bloqueio

- Solução de Peterson e TSL apresentam um problema: o loop de espera consome ciclos de processamento.
- Outro possível Problema: Inversão de prioridades
Se um processo com baixa prioridade estiver na RC, demorará mais a ser escalonado (e a sair da RC), pois os processos de alta prioridade que esperam pela RC estarão em espera ocupada.

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

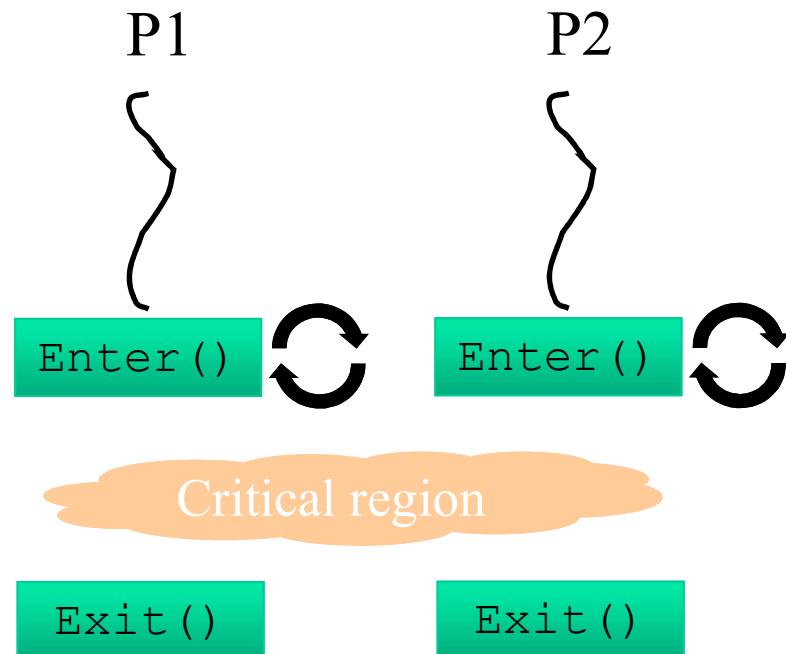
Por exemplo:

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

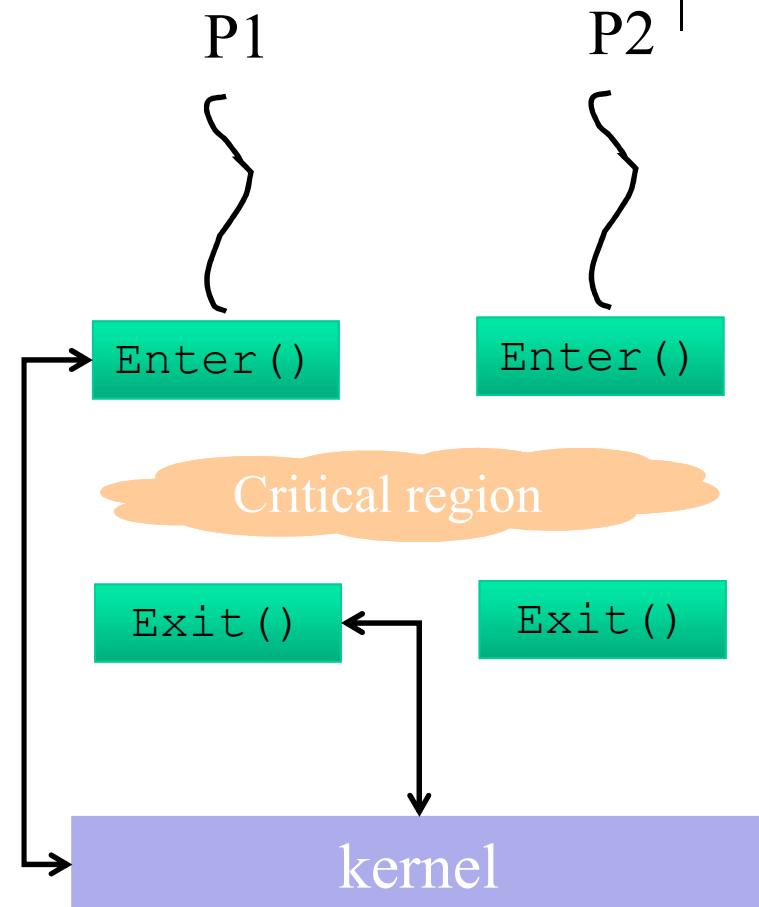
Espera Ocupada vs. Bloqueio



Como garantir que um processo de mais baixa prioridade execute?



Espera Ocupada: para arquitetura multi-core



Bloqueio:.o núcleo garante
atomicidade

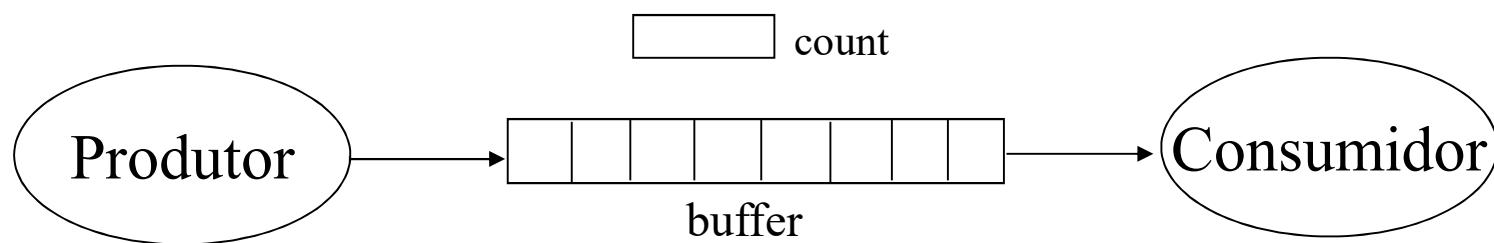


Problema do Produtor e Consumidor

Sincronismo de 2 processos que compartilham um buffer (um produz itens, o outro consome itens do buffer), e que usam uma variável compartilhada *count* para controlar o fluxo de controle.

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

Qualquer processo deve acordar o outro quando o estado do buffer permitir prosseguir o processamento



Esse tipo de sincronização está relacionada ao estado do recurso → Sincronismo de condição



Problema do Produtor e Consumidor

```
#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); /* was buffer full? */
        consume_item(item);
    }
}
```

/* number of slots in the buffer */
/* number of items in the buffer */

/* repeat forever */
/* generate next item */
/* if buffer is full, go to sleep */
/* put item in buffer */
/* increment count of items in buffer */
/* was buffer empty? */

/* repeat forever */
/* if buffer is empty, got to sleep */
/* take item out of buffer */
/* decrement count of items in buffer */
/* print item */

Por quê testa se
count == 1?

Por quê testa se
count == N - 1?



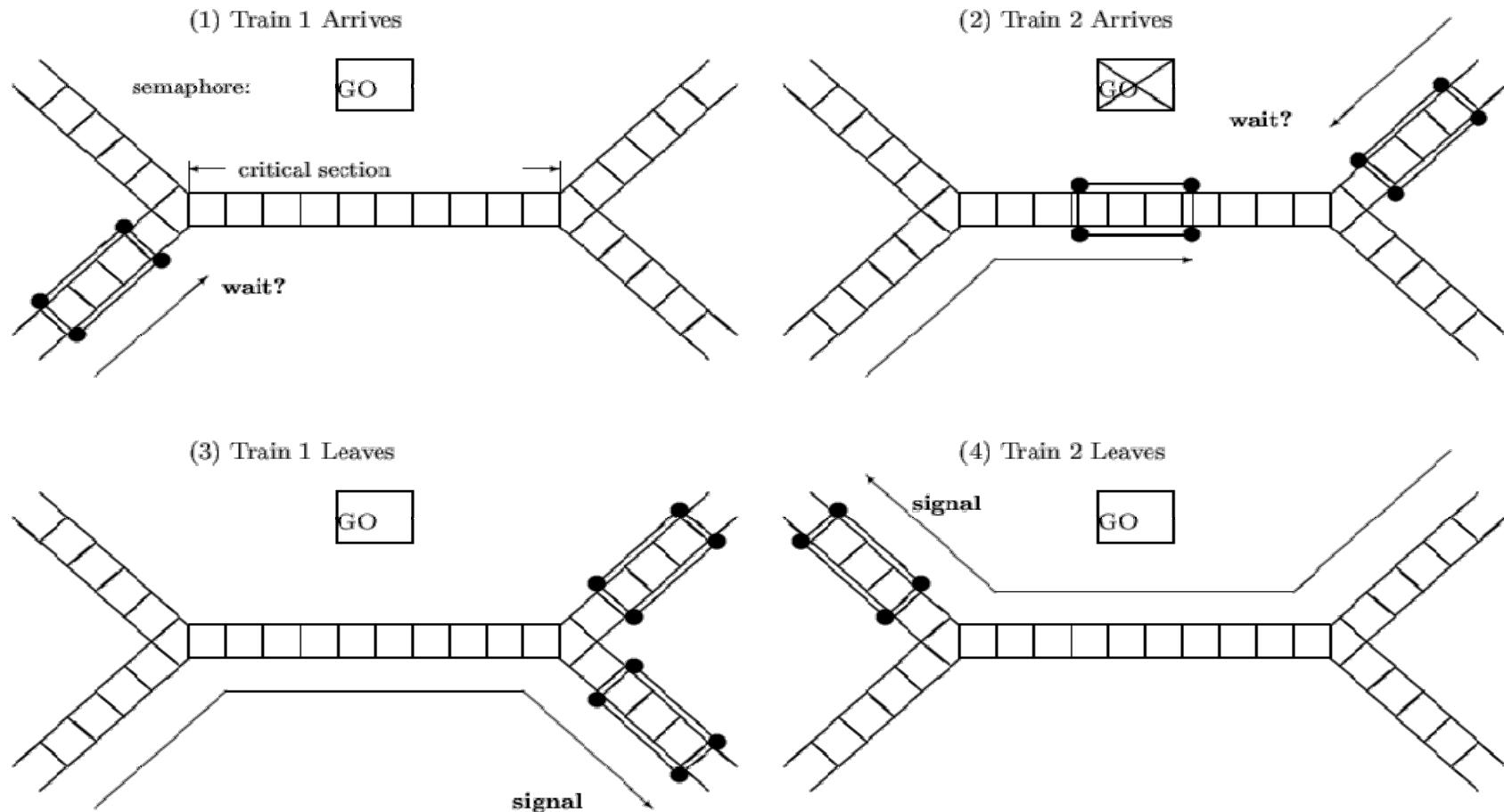
Problema do Produtor e Consumidor

Condição de corrida:

consumidor verifica que `count=0`, mas antes que execute *sleep*, o produtor é escalonado pelo SO, acrescenta item na lista e executa *wakeup*. Mas como consumidor ainda não executou *sleep*, consumidor ficará bloqueado e sistema entrará em um **impasse**.

Semáforos

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





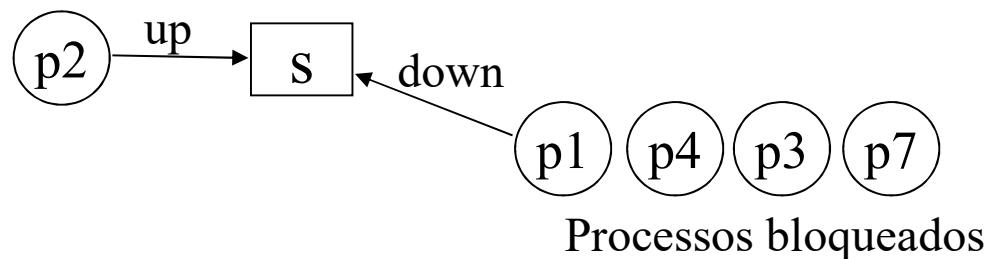
Semáforos

Trata-se de um contador que representa o número de processos que podem entrar em uma Região Crítica.

A cada semáforo está associado uma lista de processos bloqueados.

Para um semáforo **s** existem duas operações atômicas (P/V ou Down/Up):

- Down(&s) :: Se $s=0$, processo invocador bloqueia nesta chamada.
Se $s \neq 0$, decrementa s e continua execução
- Up(&s) :: Incrementa s , desbloqueia um dos processos bloqueados (se houver) e continua execução



Operações Down e Up geralmente são implementadas como chamadas ao sistema operacional, e durante a sua execução o núcleo desabilita temporariamente as interrupções (para garantir a atomicidade)



Semáforos: implementação

Semaphore Structure:

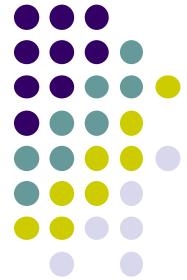
```
Typedef struct {
    int value;
    struct process *list;
} semaphore
```

down Wait Operation:

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

up Signal Operation:

```
Signal (semaphore *S) {
    S->value++;
    if (S->list != NULL) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```



Mutex: semáforos binários

Um **Mutex**, é um semáforo s que pode somente ter dois estados:
Livre e *Ocupado* ($s=1$ e $s=0$, respectivamente)

E as operações recebem outro nome:

- `Mutex_lock`
- `Mutex_unlock`

Os mutex são usados para implementar exclusão mútua simples,
isto é, onde apenas 1 processo pode estar na região crítica.



Semáforos: Exemplo de Uso

O Problema Produtor-Consumidor
(usando 1 mutex e 2 semáforos)

Semáforos: Exemplo de Uso



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

/* 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 */

void producer(void)
{
    int item;

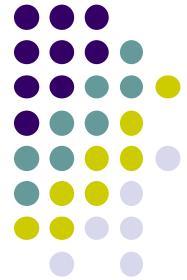
    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

/* 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 */

void consumer(void)
{
    int item;

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

/* 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 */
```



Mutex: Semáforo para Exclusão Mútua

mutex_lock:

```
TSL REGISTER,MUTEX  
CMP REGISTER,#0  
JZE ok  
CALL thread_yield  
JMP mutex_lock
```

ok: RET | return to caller; critical region entered

| copy mutex to register and set mutex to 1
| was mutex zero?
| if it was zero, mutex was unlocked, so return
| mutex is busy; schedule another thread
| try again later

mutex_unlock:

```
MOVE MUTEX,#0  
RET | return to caller
```

| store a 0 in mutex

Implementação de *mutex_lock* e *mutex_unlock* usando TSL



Monitor

Idéia básica:

Usar o princípio de encapsulamento de dados também para a sincronização:

- Várias threads podem estar executando o mesmo monitor;
- A cada momento, apenas um procedimento do monitor pode estar sendo executado;

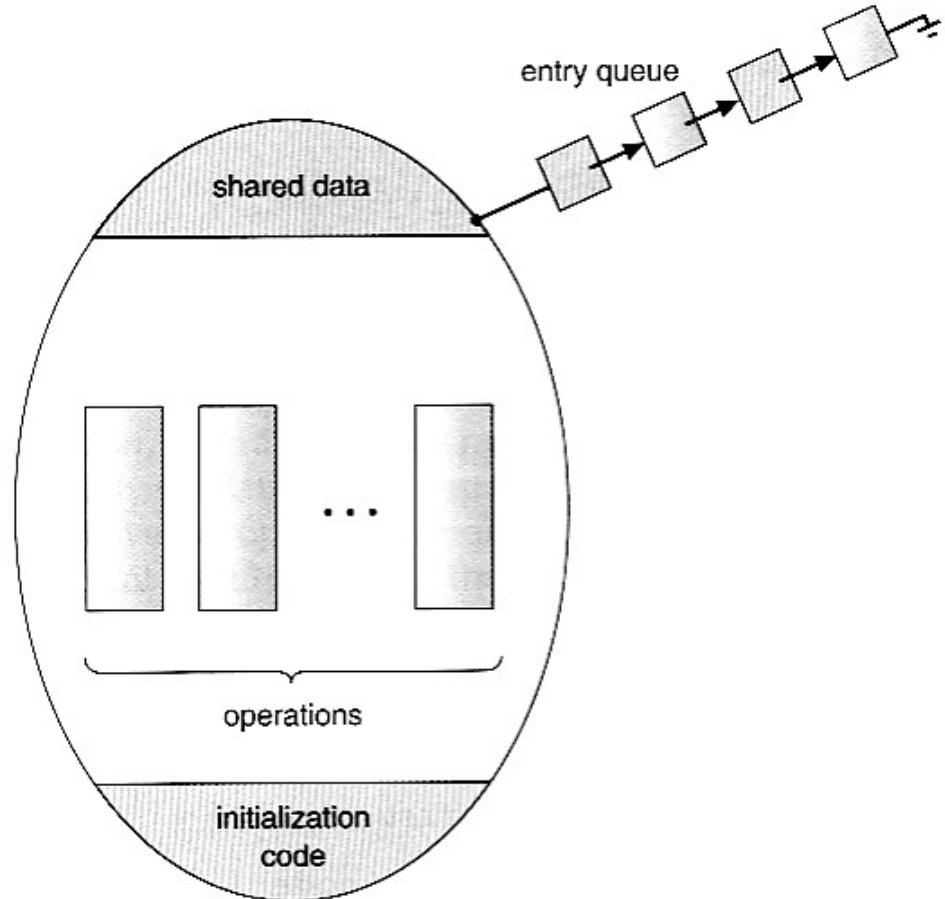
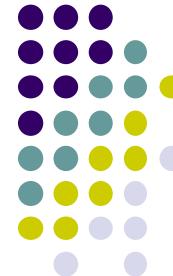


Figure 6.17 Schematic view of a monitor.

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;
- Em Java, tem-se algo similar: classe com métodos **synchronized**

Monitor

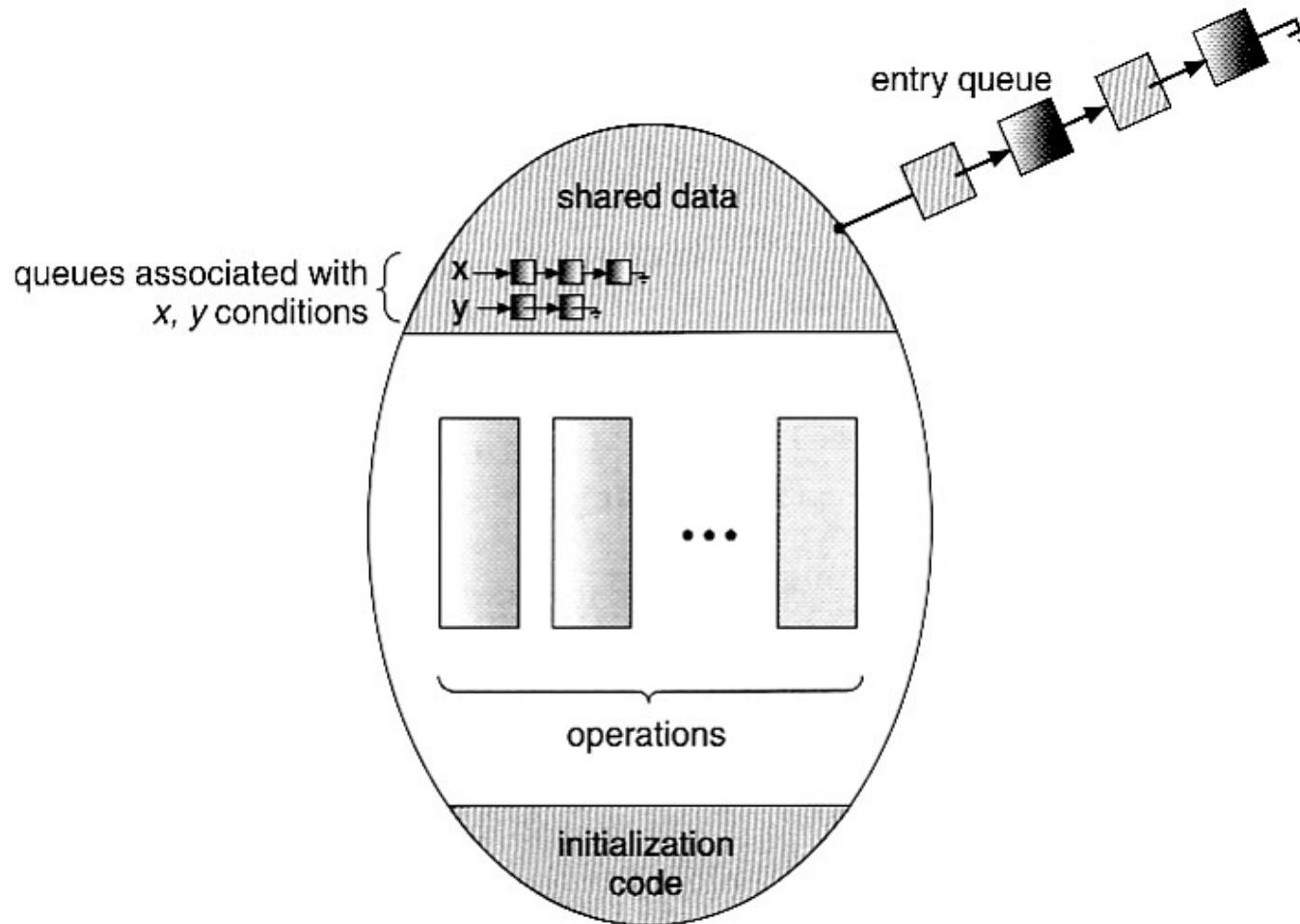


Figure 6.18 Monitor with condition variables.



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;
```

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



Monitores só servem para threads:

- Processos não têm acesso a dados globais (as instâncias de monitor)

Semáforos podem ser usados por processos e threads

- Semáforos são elementos do núcleo

O Jantar dos Filósofos



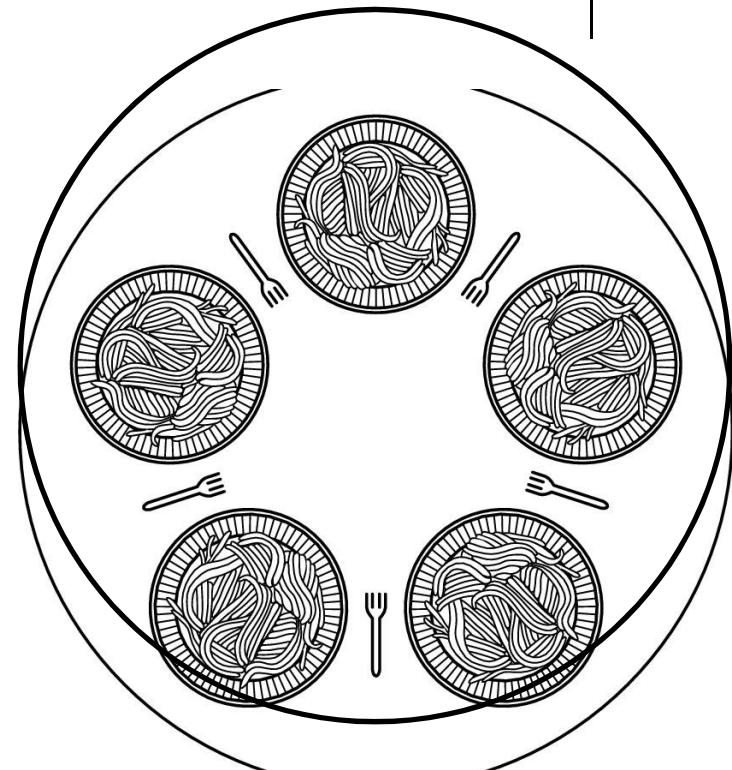
Sincronização para compartilhamento de recursos 2 a 2

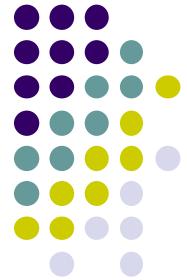
Definição do Problema:

- Filósofos só tem 2 estados: comem ou pensam;
- Para comer, precisam de dois garfos, cada qual compartilhado com os seus vizinhos;
- Só conseguem pegar um garfo por vez;

Questões globais:

- Como garantir que nenhum filósofo morre de fome?
- Como evitar o **impasse**?





Relação entre Jantar dos Filósofos e S.O.?

Isso é um problema que pode ocorrer em Sistemas Operacionais?

Considere a situação:

Um processo precisa escrever dados em 2 arquivos ao mesmo tempo, e cada um desses arquivos é compartilhado com outros processos.

Possível solução(?):

Lock fileA

Lock fileB

Write information to fileA and fileB

Release the locks



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);  
    }  
}
```

Tentativa 1: Cada filósofo tenta pegar o garfo esquerdo, e se conseguir, espera pela devolução do garfo direito.

E se todos pegarem o esquerdo ao mesmo tempo?

Tentativa 2: Aguarde até obter garfo esquerdo; Se garfo direito estiver disponível, ok, senão devolve também o garfo esquerdo e espera.

Qual é o problema agora?



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);
    }
}
```

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);                                     /* 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 */
}

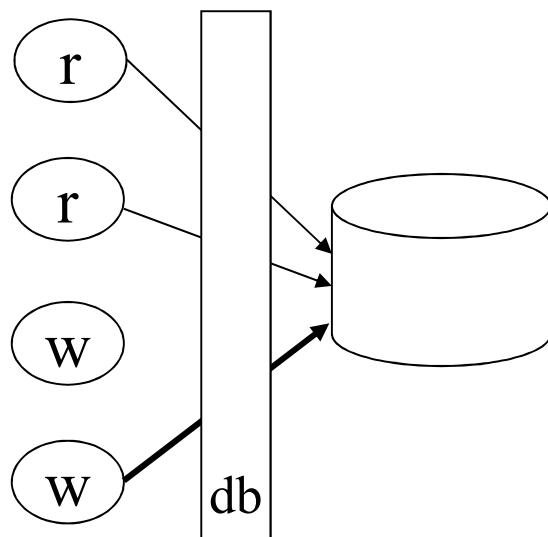
void put_forks(i)                                    /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                     /* enter critical region */
    state[i] = THINKING;                            /* philosopher has finished eating */
    test(LEFT);                                       /* 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]);
    }
}
```

O Problema dos Leitores e Escritores



Vários leitores podem entrar na RC ao mesmo tempo, mas escritores precisam executar em exclusão mútua.



```
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();  
    }  
}  
  
void writer(void)  
{  
    while (TRUE) {  
        /* repeat forever */  
        /* noncritical region */  
        think_up_data();  
        down(&db);  
        write_data_base();  
        up(&db);  
    }  
}
```

O Problema do Barbeiro Dorminhoco



- Pode haver até 5 clientes esperando serviço.
- Se todas cadeiras ocupadas, cliente aguarda fora
- Se não há clientes, barbeiro tira soneca, até que chega um novo cliente





O Barbeiro Dorminhoco: Semáforos

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

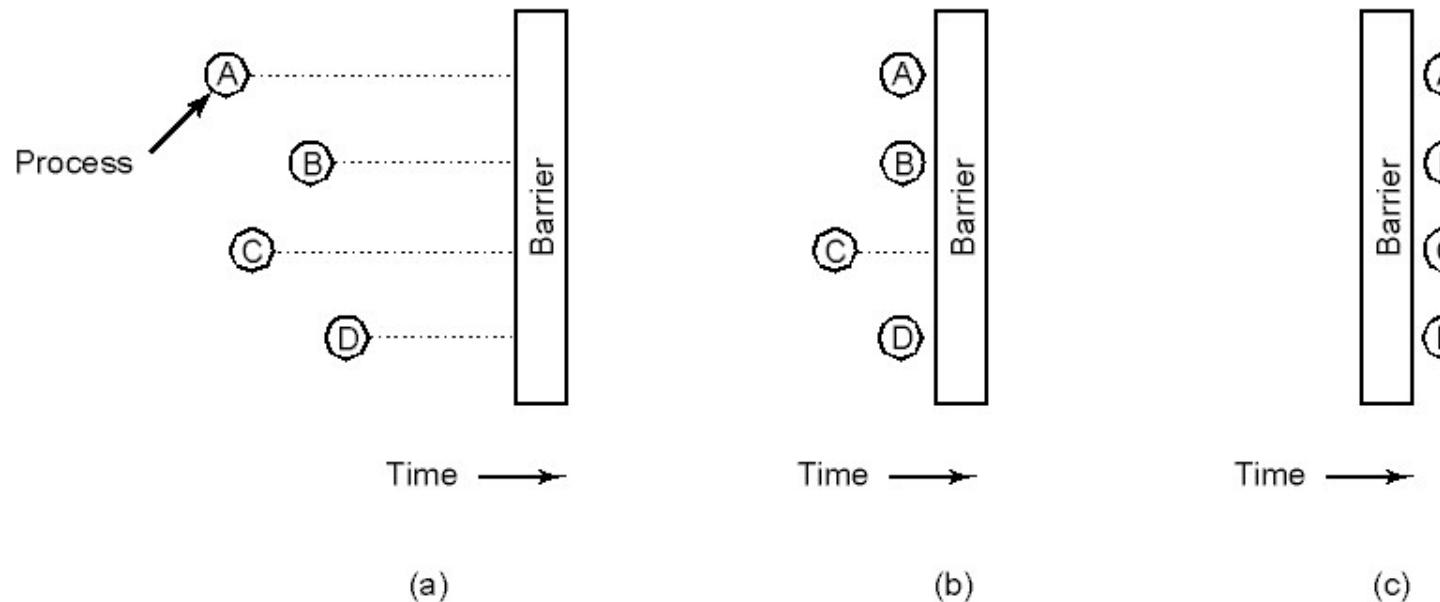
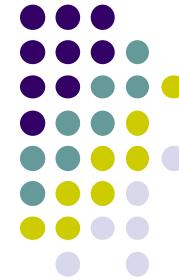
typedef int semaphore;    /* use your imagination */

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) */

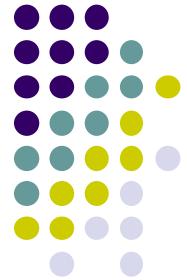
void barber(void)
{
    while (TRUE) {
        down(&customers);   /* go to sleep if # of customers is 0 */
        down(&mutex);       /* acquire access to 'waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&barbers);       /* one barber is now ready to 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 */
    }
}
```

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 à barreira, são bloqueados para esperar pelos demais
 - Quando o retardatário chega, todos são desbloqueados e podem prosseguir



Sincronização de 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);
    ...
}
```



Semáforos

- Todas as bibliotecas de threads (ou system calls) provêm operações para semáforos:
 - `sem_t semaphore`
 - `sem_init (&semaphore, 0, some_value);`
 - `sem_down (&semaphore);`
 - `sem_up (&semaphore);`
- Em Pthreads usa-se: `wait` = `down`; `post` = `up`.



Envio de Mensagens

- É uma forma natural de interação entre processos
- Duas primitivas:
 - **Send** (dest, &message) – tamanho da mensagem fixo ou variável
 - **Receive** (fonte, &message) - fonte pode ser ANY
- Requer que processos se conheçam mutuamente
 - definem um enlace lógico de comunicação
 - Mensagens são tipadas e possuem header e dados
- Um enlace pode ser:
 - com processos co-localizados ou remotos
 - confiável ou não-confiável
 - ponto-a-ponto ou ponto-a-multiponto
 - envolve elementos físicos (e.g., memória compartilhada, barramento, rede)

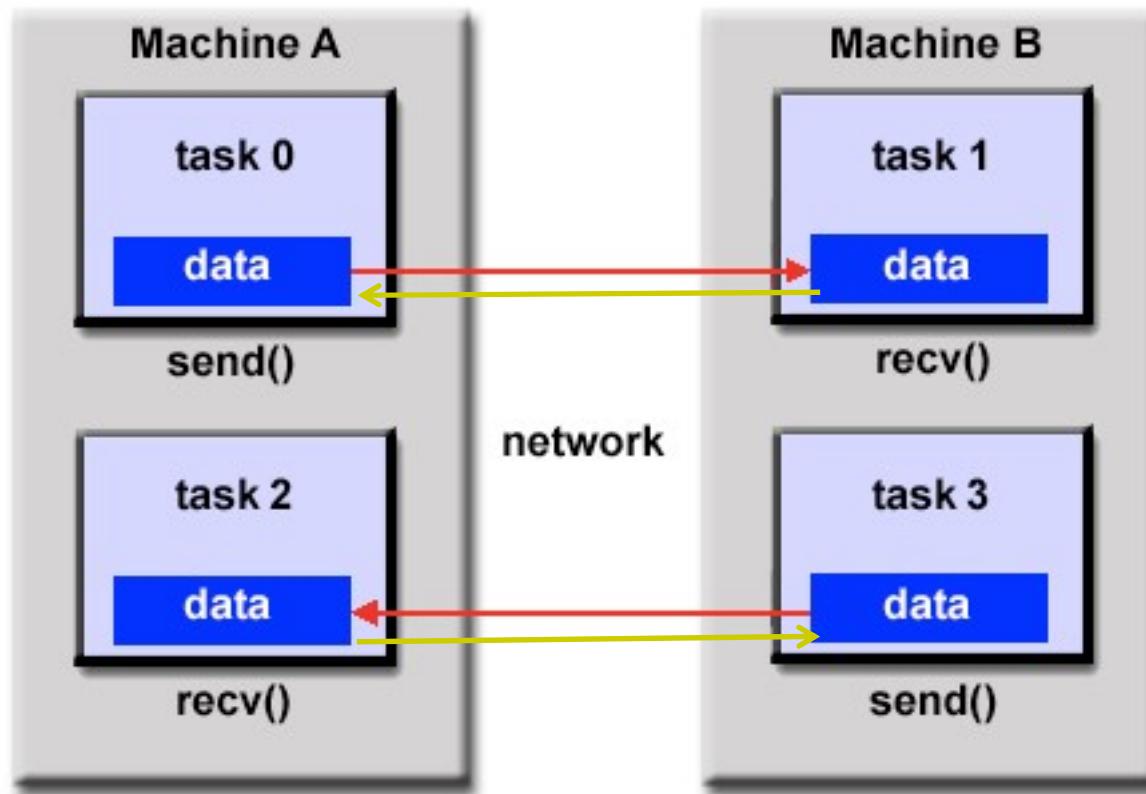


Envio de Mensagens

- Envio de mensagens é um mecanismo de sincronismo mais genérico, porque:
 - Permite também a troca de dados
 - É independente se processos compartilham memória ou não.
 - Qualquer solução baseada em semáforos pode ser resolvida por envio de mensagens (considere: Down \equiv receive, Up \equiv send, valor do semáforo = número de mensagens)
- Decisões de projeto do mecanismo:
 - Com ou sem bufferização (send assíncrono ou *Rendezvous*)
 - Quando a comunicação é remota (pela rede) mensagens podem ser perdidas: → são necessárias confirmações, timeouts e re-transmissões
 - Na versão *Rendezvous* podem ocorrer impasses

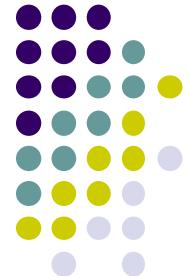


Envio de Mensagens

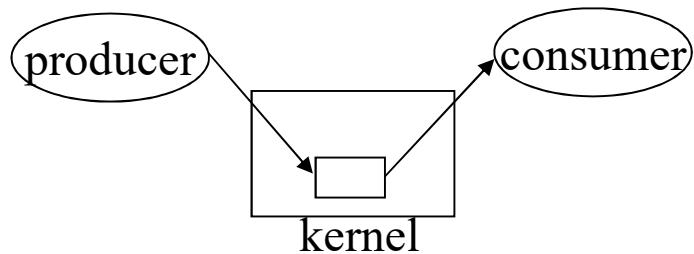


Tipos de Envios de Mensagem

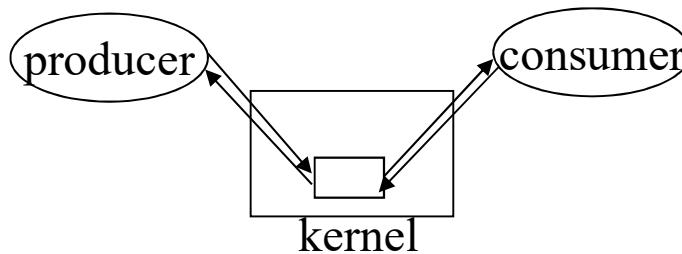
Entre processos co-localizados



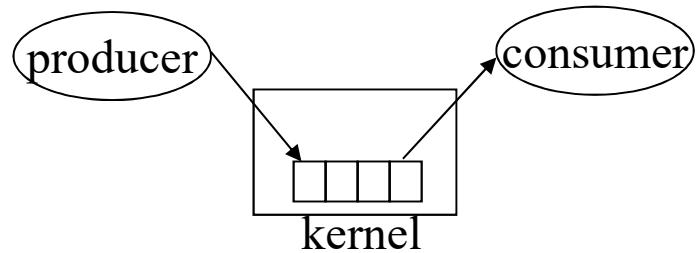
Rendezvous (comunicação síncrona – emissor é bloqueado até que a msg seja recebida)



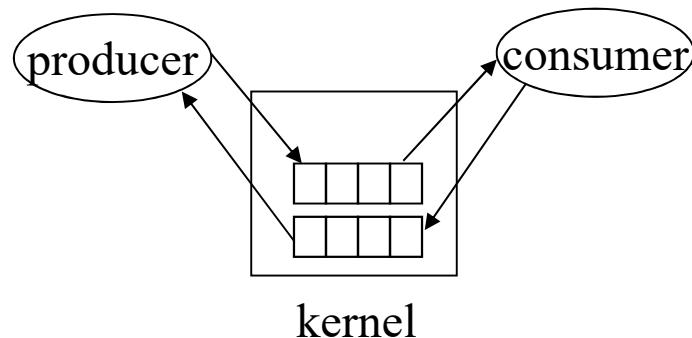
Request-Reply síncrono



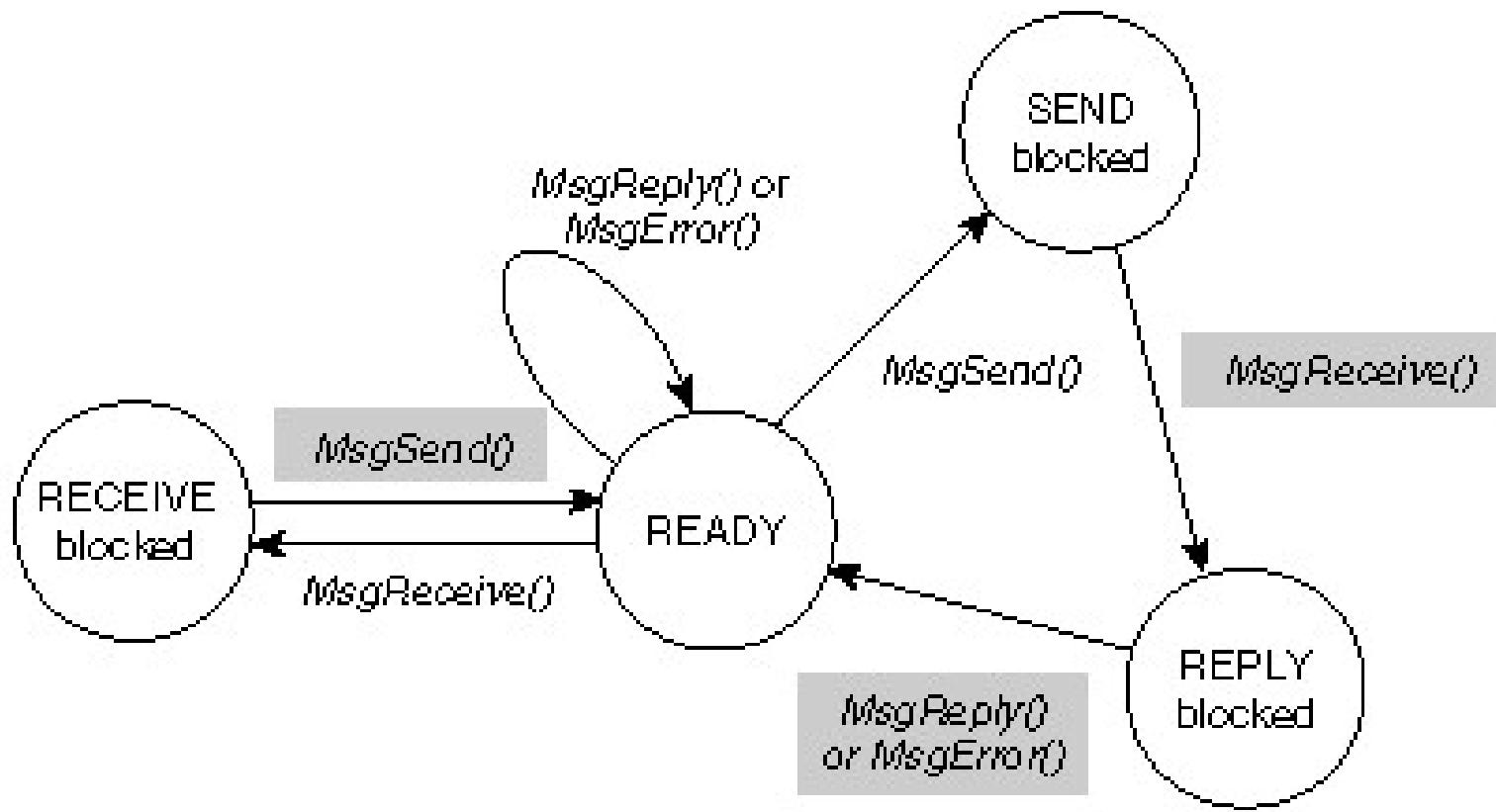
Não-bloqueante



Request-Reply, não-bloqueante



Request-Reply Síncrono: possíveis estados dos processos/threads



This thread

Other thread

Problema do Produtor-Consumidor com envio de N Mensagens



Idéia: 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 */
    }
}
```



Perguntas?

