

Comunicação & Sincronização (entre processos/ threads)



Roteiro:

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

Interação entre processos concorrentes

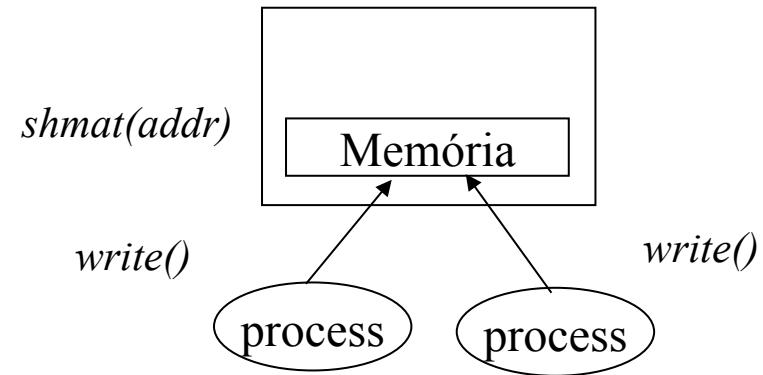
- Em uma aplicação concorrente, ou na própria execução do sistema, muitas vezes é necessário que processos/threads troquem dados entre si .
- Tal comunicação pode ser implementada de diversas formas, por exemplo, usando um arquivo compartilhado, memória compartilhada, ou por troca de mensagens.
- Nesses casos, é necessário que os processos tenham suas execuções sincronizadas pelo sistema operacional.
- Exemplo: 2 processos compartilhando um buffer p/ troca de dados.
 - Leitor precisa esperar até que exista dado no buffer.
 - Escritor precisa esperar para que haja espaço no buffer.

Obs: Interação através de um arquivo compartilhado também envolve o custo elevado de acesso ao disco.

Comunicação e Sincronização entre Processos: duas Abordagens

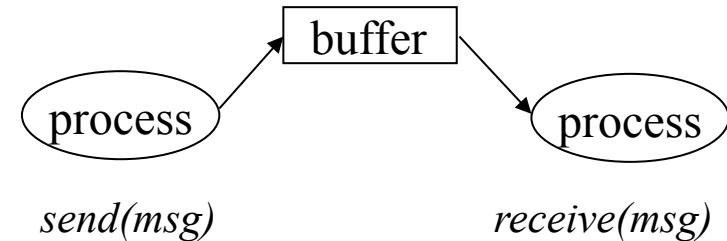
1. Baseada em memória compartilhada

- Comunicação é **implícita** (por dados compartilhados, sem canal de comunicação) mas
- Sincronização para acesso precisa ser feita **explicitamente**



2. Baseada em troca de mensagens

- Canal de comunicação é **explicito**;
- Sincronização é **implícita** (processos bloqueiam nas primitivas)



Processos concorrentes que interagem sofrem do problema de condição de corrida.

Mecanismos de Comunicação e Sincronização em UNIX

- Shared Memory – compartilhamento sem sincronização
- Signals – notificações assíncronas p/ controle e sincronização
- Pipes – para comunicação fifo entre processo pai e filhos
- FIFO (Named pipes) – para comunicação fifo entre qq processos, e com persistência de dados
- Message Queues – para comunicação fifo, com mensagens rotuladas com um tipo
- Semaphore e Mutex – p/ sincronização
- Sockets – para comunicação fifo cliente-servidor

Condição de Corrida

Problema decorrente do **indeterminismo na ordem de acesso concorrente** (simultaneidade) a um **recurso/dados compartilhado**.

- Em Sistemas Operacionais: **o escalonamento gera indeterminismo** na execução dos processos/threads concorrentes (a multiplexação no uso da CPU)

Exemplo: Comandos de atribuição em linguagem de alto nível são traduzidos para instruções de máquina mais elementares.

Seja variável compartilhada x com valor inicial 2.

Processo A:

`x = x+1;`

`LOAD x,%eax
ADD 1,%eax
STO %eax,x`

ProcessoB:

`x = x -1;`

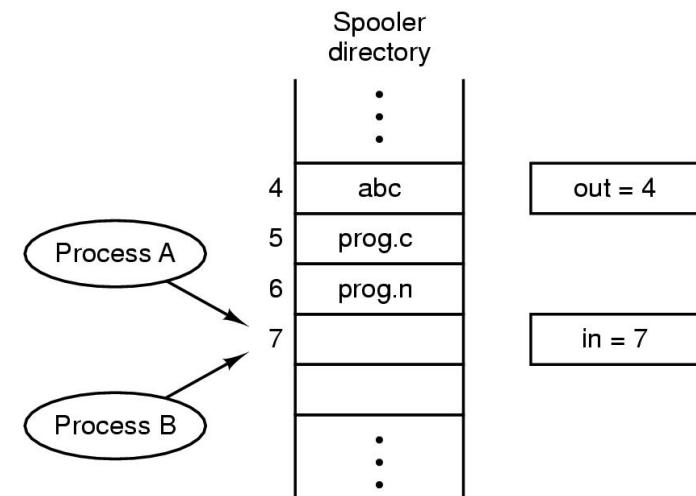
`LOAD x,%ebx
SUB 1,%ebx
STO %ebx,x`

- Proc. A é interrompido após ADD; Proc B executa até final e Proc A retoma execução. Valor de x no final será 3 (inconsistente), ou
- Se Proc. B também for interrompido antes de SUB, valor pode ser 1 (inconsistente).

Condição de Corrida

Exemplo 1: Dois processos incluem job de impressão a fila de spool:

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



Exemplo 2: Dois ou mais processos precisam ler e escrever vários dados no mesmo arquivo; Mas cada escrita têm uma certa coerência (posições consecutivas com conteúdo similar)

Exemplo 3: Processos compartilham uma lista (ou vetor) de elementos com escrita: atualização requer escritas combinadas em vários endereços de memória (para manipulação correta dos ponteiros)

Condição de Corrida

Exemplo 4: 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

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

Condição de Corrida

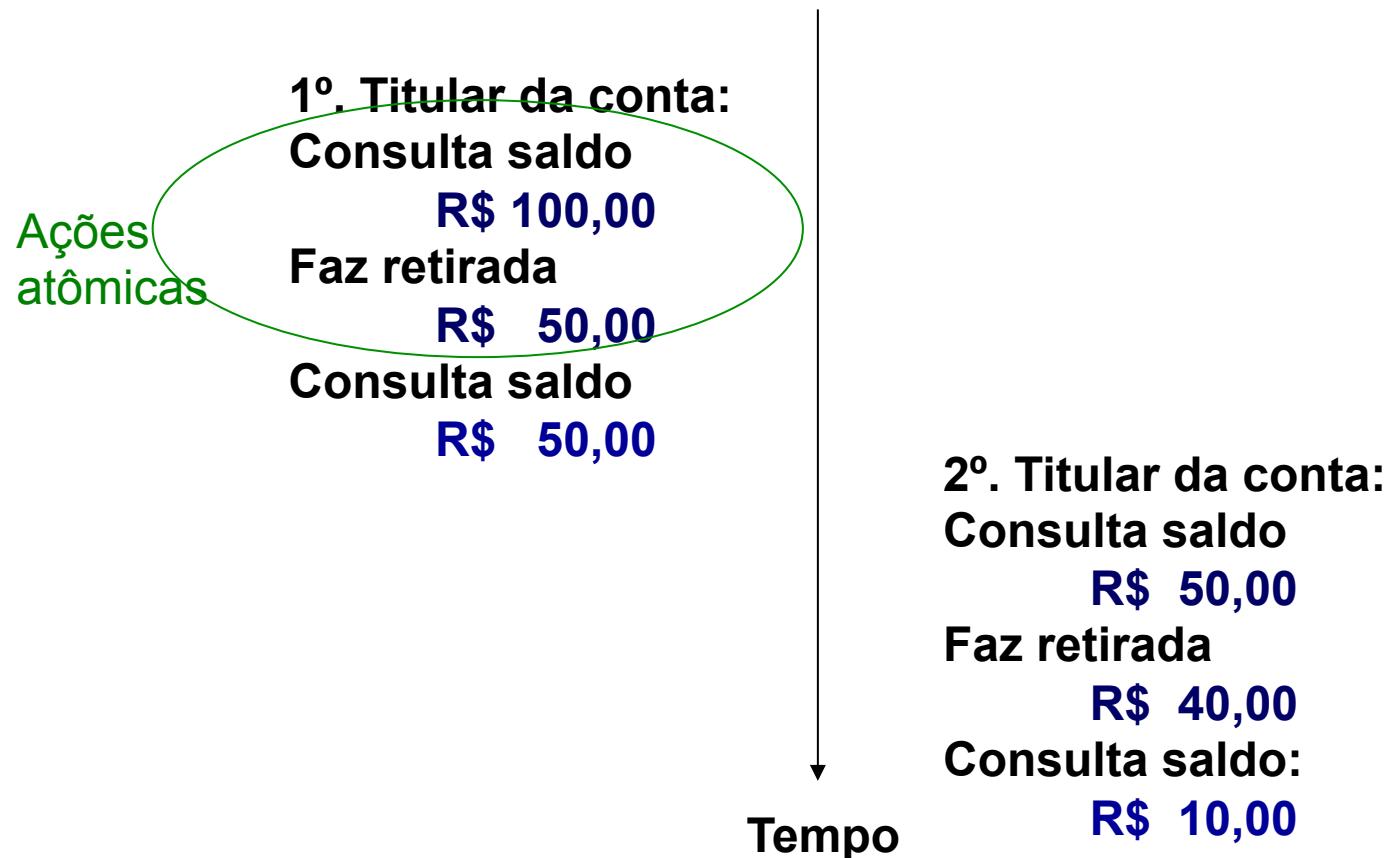
Exemplo 4: atualização de conta bancária conjunta



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

Condição de Corrida

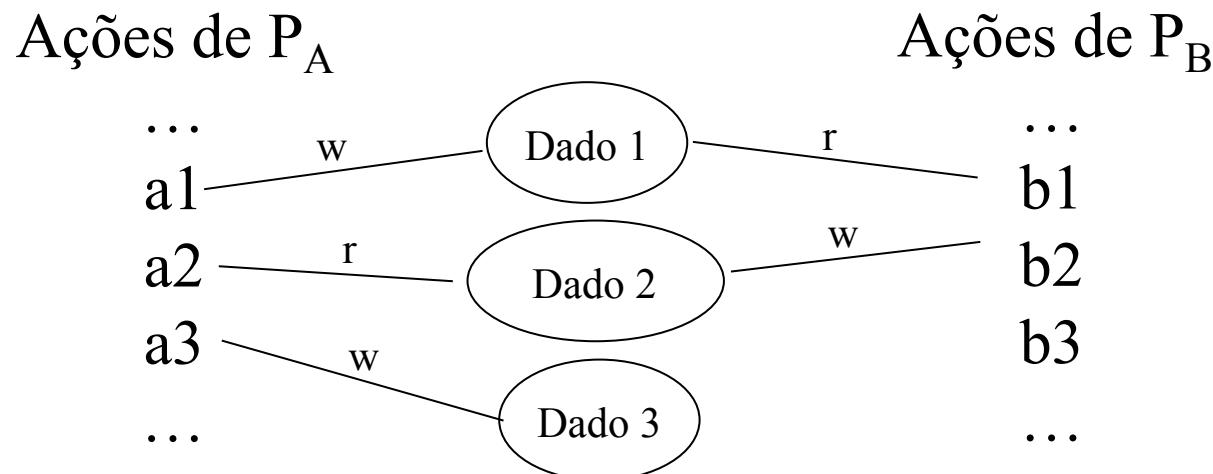
Exemplo 4: atualização de conta bancária conjunta



Condição de Corrida

Ocorre sempre que...

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

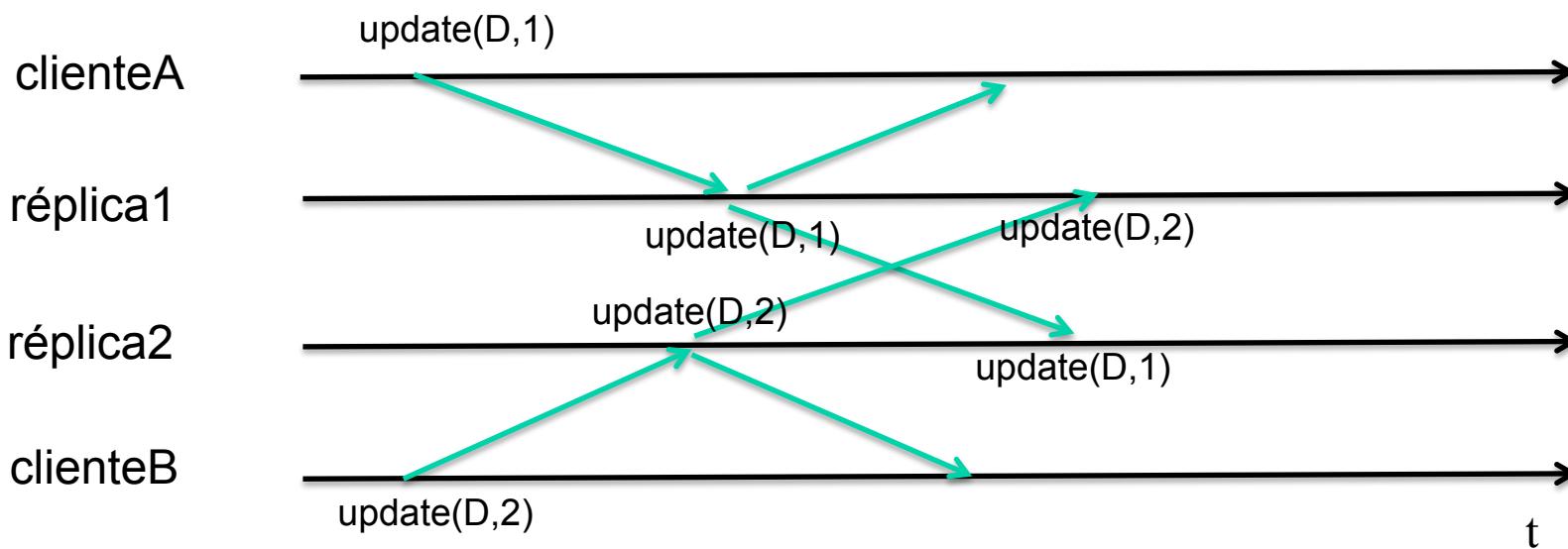


Condição de Corrida

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

Exemplo:

1. Suponha um serviço tolerante a falhas que seja implementado com replicação (distribuída) de dados.
2. Assim que recebe uma requisição, um servidor responde ao cliente e envia a atualização seu dados para o outro servidor, que atualiza seu dado correspondente



Os servidores precisam entrar em um consenso sobre a ordem final das operações.

Condição de Corrida

Para evitar os problemas de condição de corrida os processos/threads que compartilham algum dado/recurso:

1. precisam “bloquear” temporariamente o recurso, impedindo que os outros processos interfiram no seu acesso ao recurso. (ou seja, garantir a **Exclusão mútua**)
2. precisam ser capazes de “notificar” os demais processos quando o recurso foi “liberado” ou quando está no estado que permita uma determinada operação (por exemplo, consumo de dados quando um buffer deixou de estar vazio)

Esse “bloquear”, “liberar” e “notificar” precisa ser atômico.

Os locais no código em que há acesso a dados/recursos compartilhados precisa estar protegido.

Região Crítica

Região crítica (ou Sessão crítica) é uma parte do programa que contém as operações sobre dados compartilhados, a serem executadas em **exclusão mútua**.

Para solucionar o problema das regiões críticas **alguns requisitos básicos precisam ser satisfeitos**:

Exclusão Mútua: Se um processo P está executando sua região crítica nenhum outro poderá executar a sua região crítica (**segurança = safety**)

Progresso: Um processo executando dentro da região crítica não pode ser bloqueado/terminado por outro processo (**progresso = liveness**).

Espera Limitada: Um processo não deve esperar indefinidamente para entrar em sua região crítica (**justiça**)

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

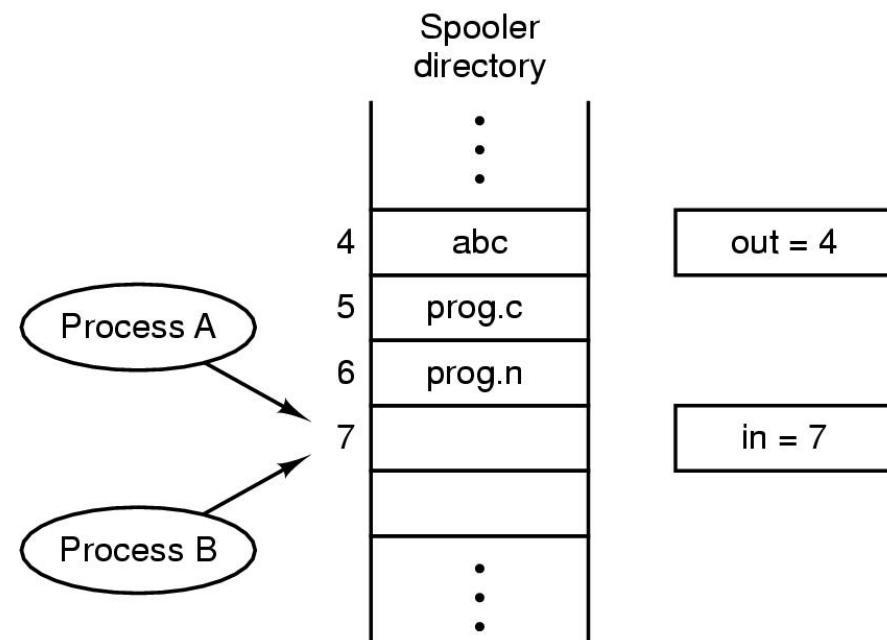
} Região crítica

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

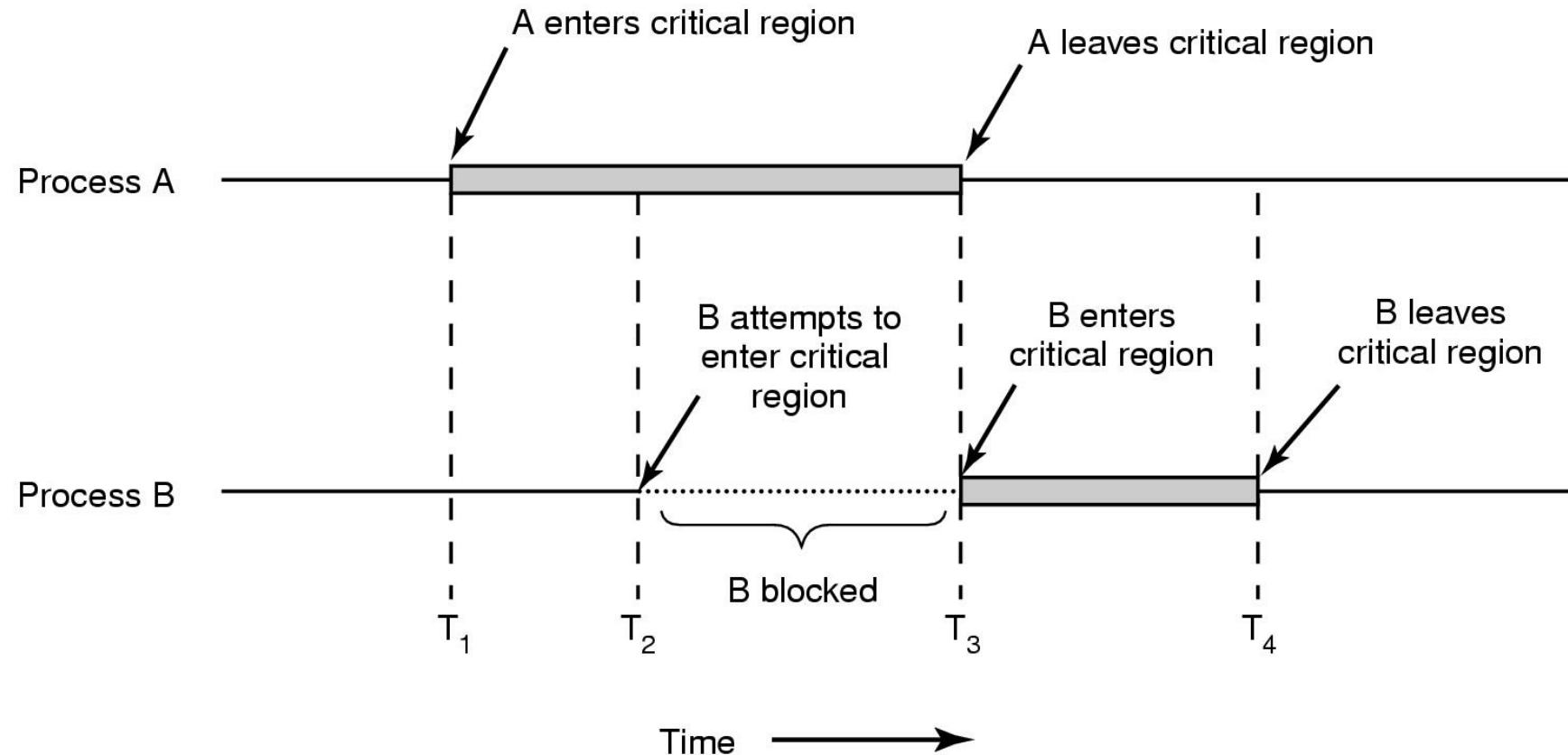
Usando Regiões Críticas

```
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

Possíveis soluções de Exclusão Mútua

Alternativas:

1. Desabilitar interrupções:
 - ➔ 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 & atribuição do lock não for atômica, então exclusão mútua não é garantida
3. Espera ocupada: 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 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

Instruções de máquina: **Test-and-Set-Lock (TSL)**:

Algumas arquiteturas possuem uma instrução de máquina TSL para leitura de um lock para um registrador e atribuição de um valor diferente de zero ($\neq 0$) **de forma atômica!**

Processos que desejam entrar RC executam TSL:

- se registrador (e lock=0), então 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	

Espera Ocupada vs. Bloqueio

- Solução de Peterson e Test-and-Set-Lock apresentam o problema da espera ocupada que consome ciclos de processamento.
- Outro problema da espera Ocupada: Inversão de prioridades

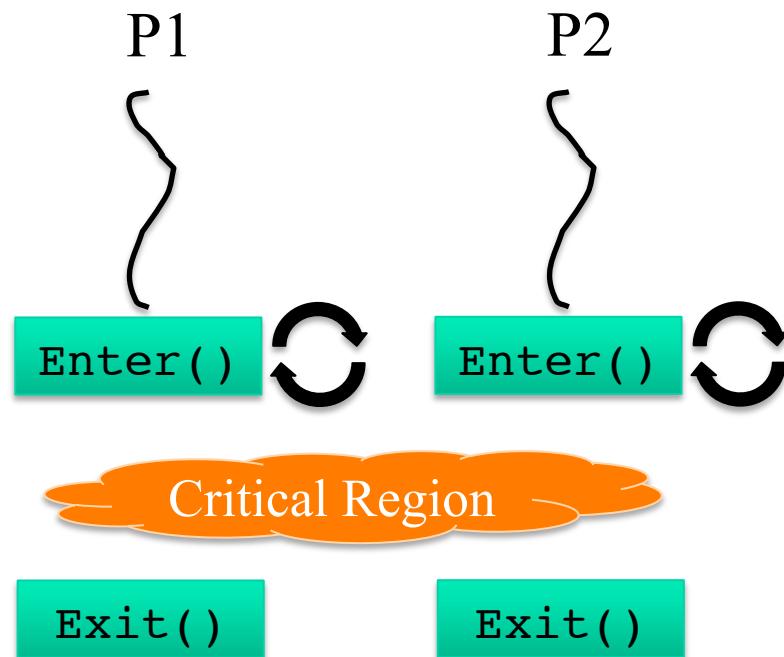
Se um processo com baixa prioridade estiver na RC, demorará mais a ser escalonado (e a sair da RC), pois processos de maior prioridade estarão executando em espera ocupada (para a RC).

A alternativa: Chamadas ao núcleo que bloqueiam o processo e o fazem esperar por um sinal de outro processo

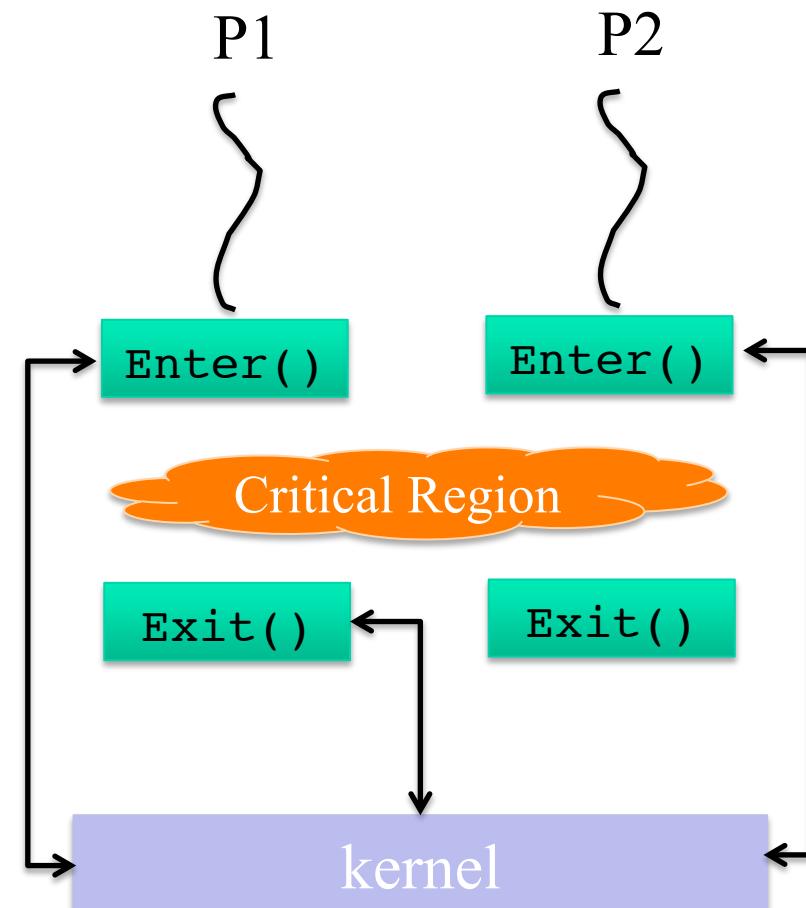
Por exemplo:

- *lock/unlock (mutex)*
- *wait/signal*
- Semáforos

Espera Ocupada *versus* Bloqueio



Espera Ocupada: consome ciclos de processamento. Pode ser usado em multi-core



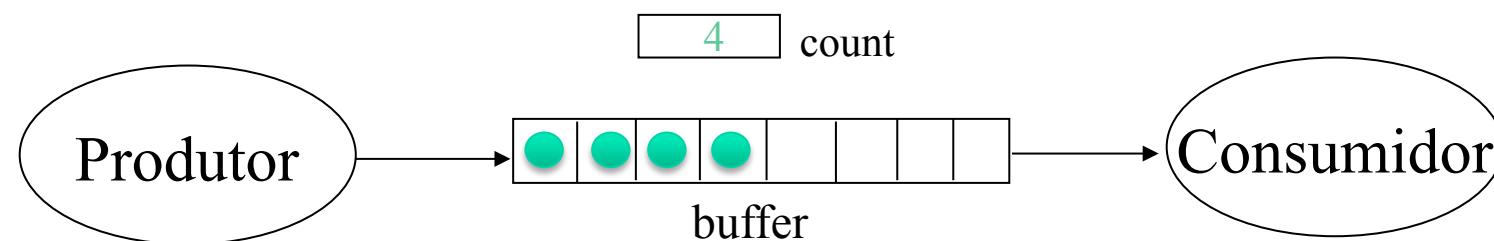
Bloqueio:.o núcleo garante atomicidade

Problema do Produtor e Consumidor

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

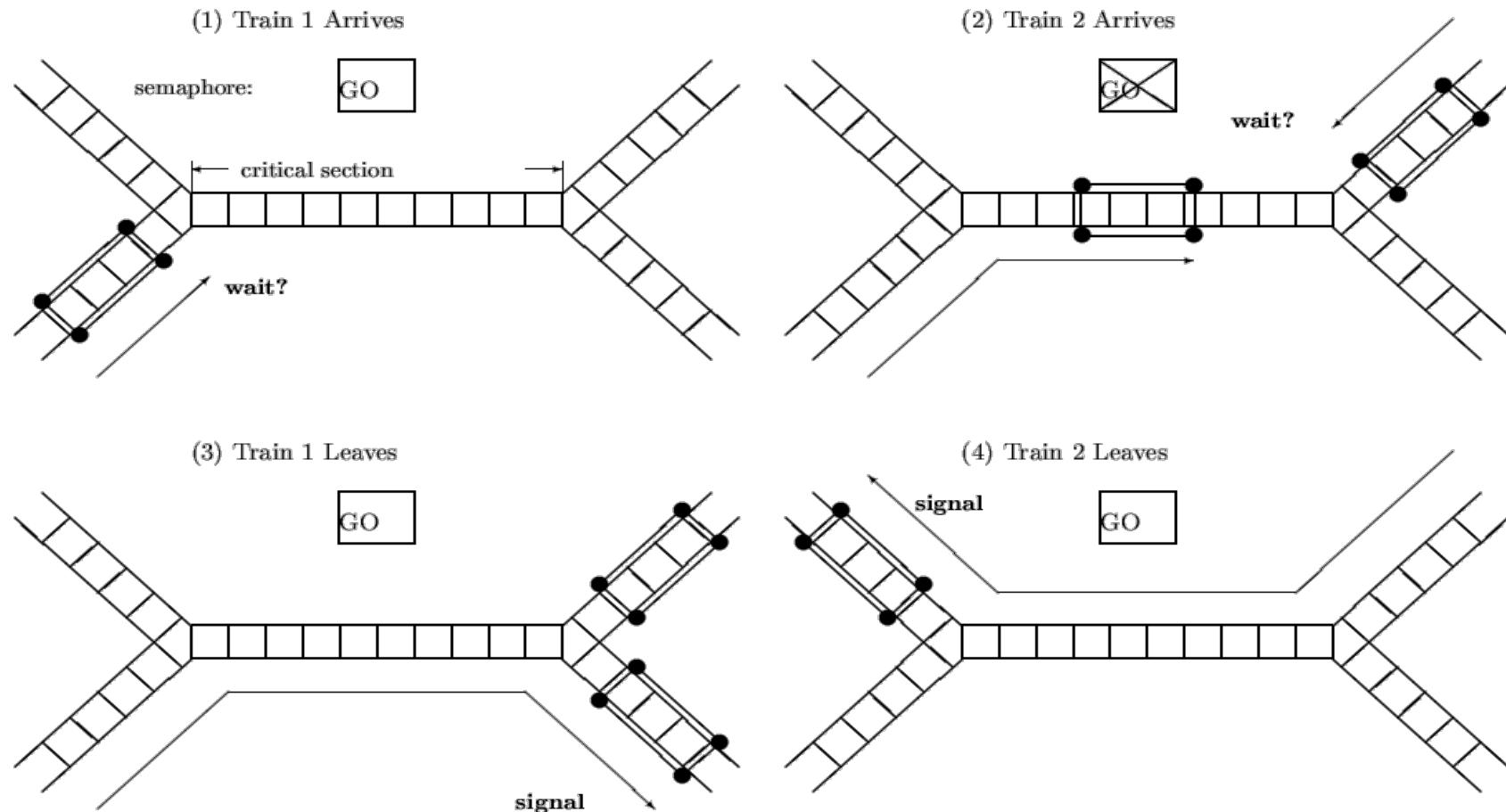
Cada processo deve “acordar” o outro processo quando o conteúdo do buffer mudar a ponto de permitir o prosseguimento do processamento



Esse tipo de sincronização está relacionada ao estado do recurso -> Sincronização de condição

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 consiste de um contador (inteiro não negativo) que pode ser manipulado por duas operações $P()$ e $V()$ (ou $down()$ e $up()$)

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

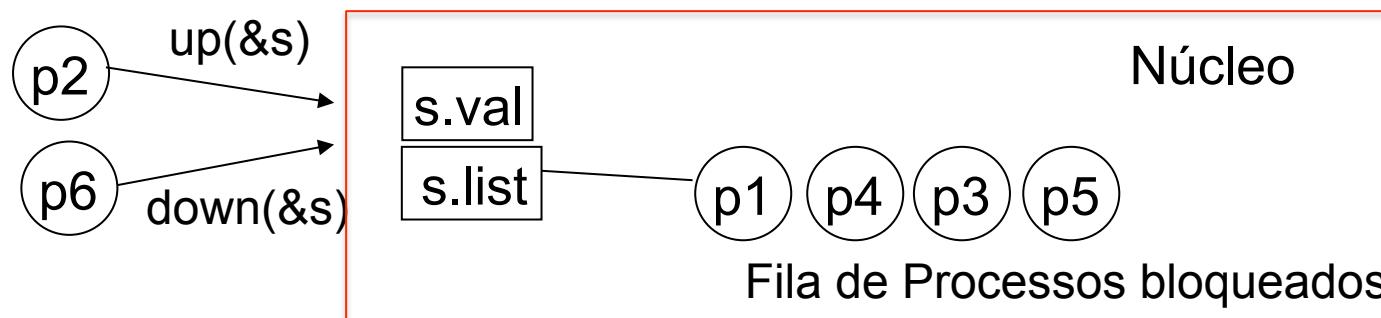
Semáforo - Implementação

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. Se s < 0 processo invocador bloqueia nesta chamada e é colocado no final da fila. Senão, continua execução
- **up(&s)** :: Incrementa s, 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**.

Semáforos: Exemplo de uso

```
#define N 100                                /* number of slots in the buffer */
typedef int semaphore;                      /* semaphores are a special kind of int */
semaphore mutex = 1;                        /* controls access to critical region */
semaphore empty = N;                         /* counts empty buffer slots */
semaphore full = 0;                          /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);                      /* TRUE is the constant 1 */
        down(&mutex);                      /* generate something to put in buffer */
        insert_item(item);                 /* decrement empty count */
        up(&mutex);                       /* enter critical region */
        up(&full);                        /* put new item in buffer */
        up(&full);                        /* leave critical region */
        up(&full);                        /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);                     /* infinite loop */
        down(&mutex);                   /* decrement full count */
        item = remove_item();            /* enter critical region */
        up(&mutex);                     /* take item from buffer */
        up(&empty);                     /* leave critical region */
        up(&empty);                     /* increment count of empty slots */
        consume_item(item);             /* do something with the item */
    }
}
```

O problema } Produtor-Consumidor usando 1 mutex e 2 semáforos

Variável de Condição

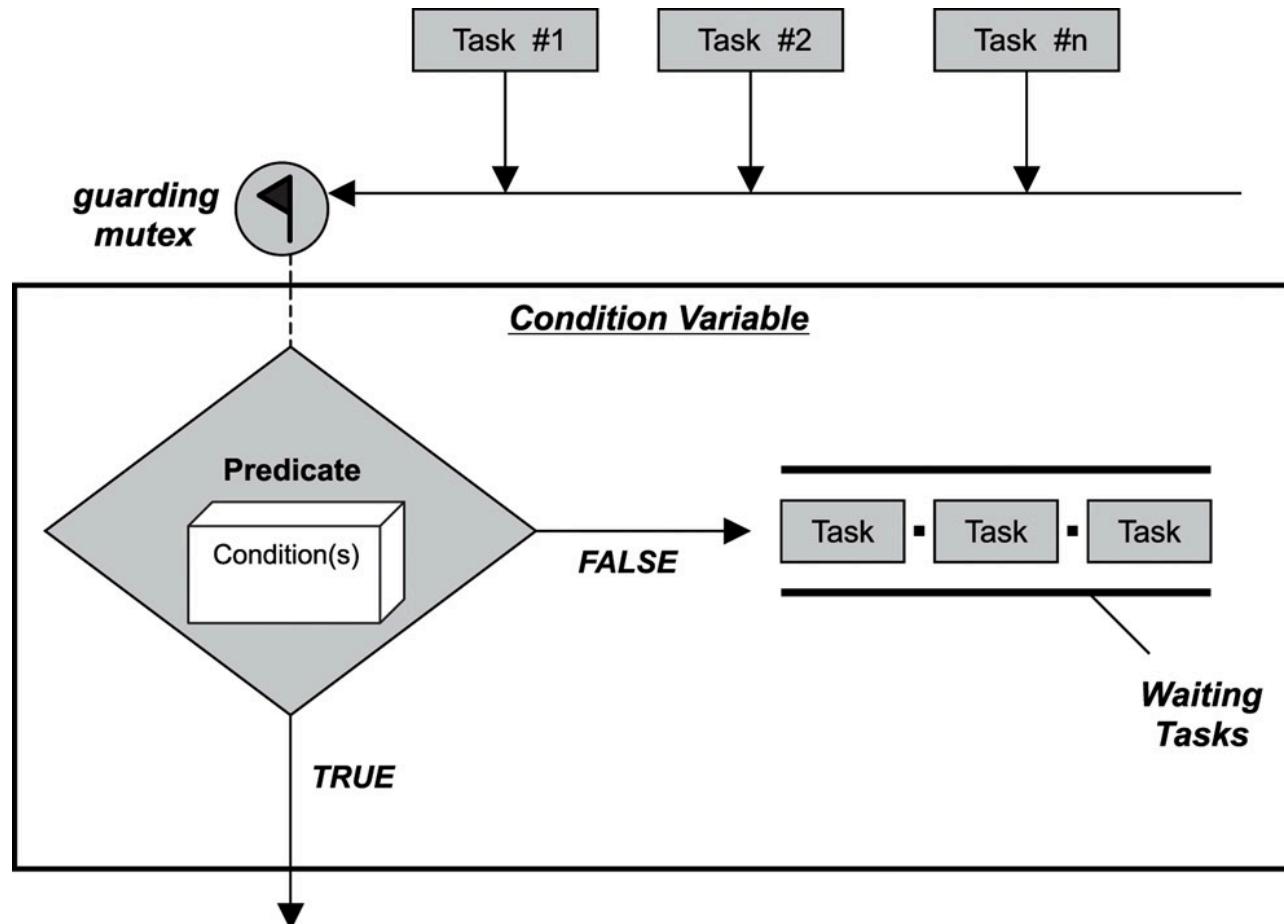
É um mecanismo de sincronização que faz o processo/thread bloquear sua execução até que uma condição esteja satisfeita.

- Threads/processos que estejam na região crítica abrem mão temporariamente da exclusão mútua deixando um outro processo entrar (que irá fazer a condição ficar satisfeita)
- Operações: **wait**, **signal** (ou **notify**) e **notifyAll** (ou **broadcast**)
- Variáveis de condição geralmente estão associadas a um mutex.

Exemplo Produtor_Consumidor:

- Quando buffer está cheio, produtor deve esperar
`pthread_cond_wait(&condp, &the_mutex);`
- Quando buffer está vazio, consumidor deve esperar
`pthread_cond_wait(&condc, &the_mutex);`
- Assim que buffer tiver um espaço, desbloqueia o produtor:
`pthread_cond_signal(&condp);`

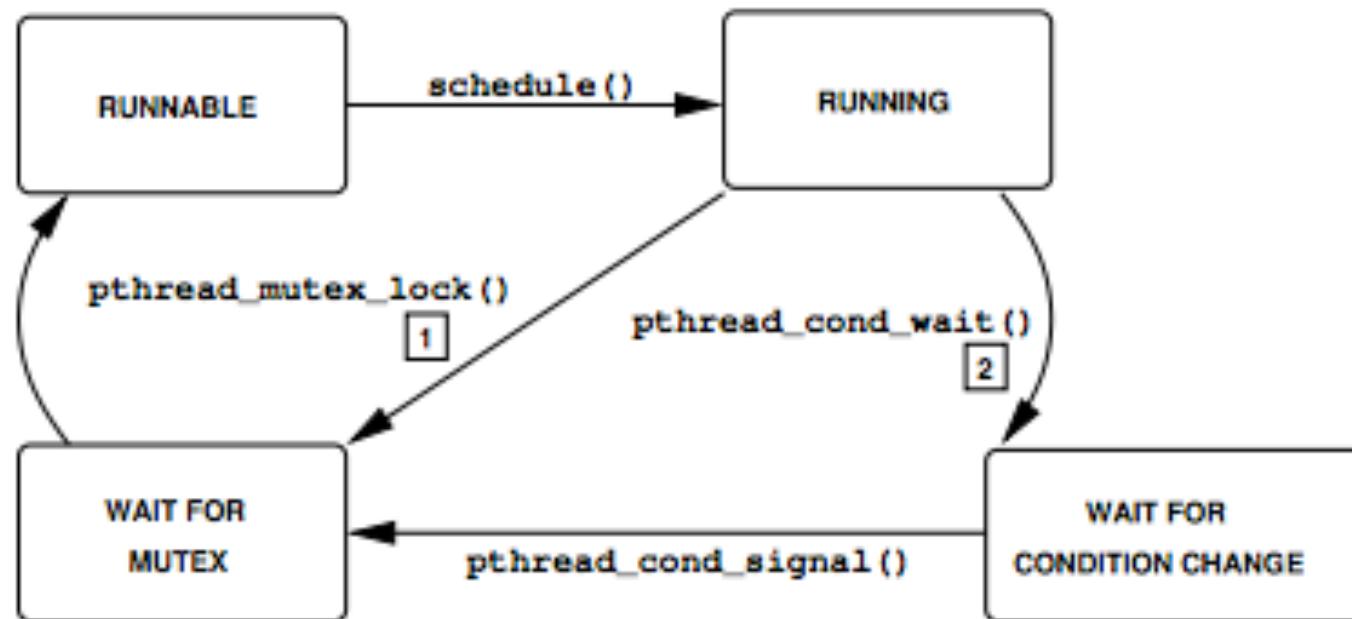
Variável de Condição



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

Espera por Mutex e por Variável de Condição

Diagrama de estados:



- Primeiro, adquire o lock do mutex e depois espera o sinal
- Nessa espera, o lock é mantido, mas outro processo pode entrar na Região Critica

Variáveis de Condição em outras linguagens

Java:

```
private Lock aLock = new ReentrantLock();
private Condition condVar =
aLock.newCondition();

public int get(int who) {
    aLock.lock();
    try {
        while (available == false) {
            try {
                condVar.await();
            } catch (InterruptedException e)
        }
        available = false;
        condVar.signalAll();
    } finally {
        aLock.unlock();
        return contents;
    }
}
```

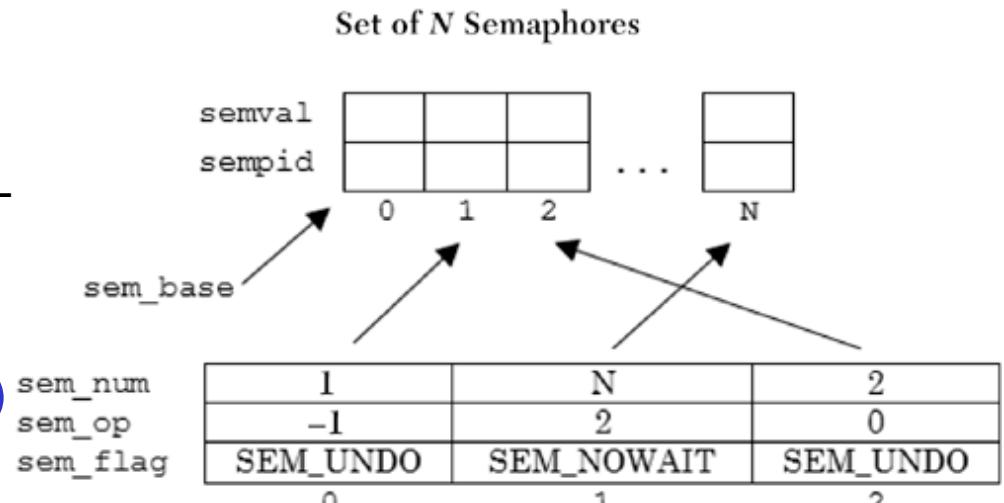
Existe a versão bloqueante e com timeout:

- `await()`
- `await(long timeout)` – se notificação não ocorre até timeout, retorna

Semáforos em UNIX

Aloca-se um vetor de semáforos

- `int semget(key, nsems, semflg)`
 - key: chave do set
 - nsems: número de semáforos
 - semflg: IPC_CREAT | IPC_EXCL
 - Retorna semID
- `int semop(semID, &sops, nsops)`
 - semID
 - sops: ponteiro para vetor de operações sobre os semáforos
 - nsops: número de operações
- `int semctl (semID, int semnum, cmd, arg);`
 - Operações de controle sobre o conjunto de semáforos.
 - Destrução de um vetor de semáforos:



Núcleo garante atomicidade de todas as operações sobre o vetor.

```
semctl(semid, 0, IPC_RMID, 0);
```

Sincronização:

Não existe algo mais simples?

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”
- Monitor é provido em algumas linguagens de programação: Modula2, Java, etc.

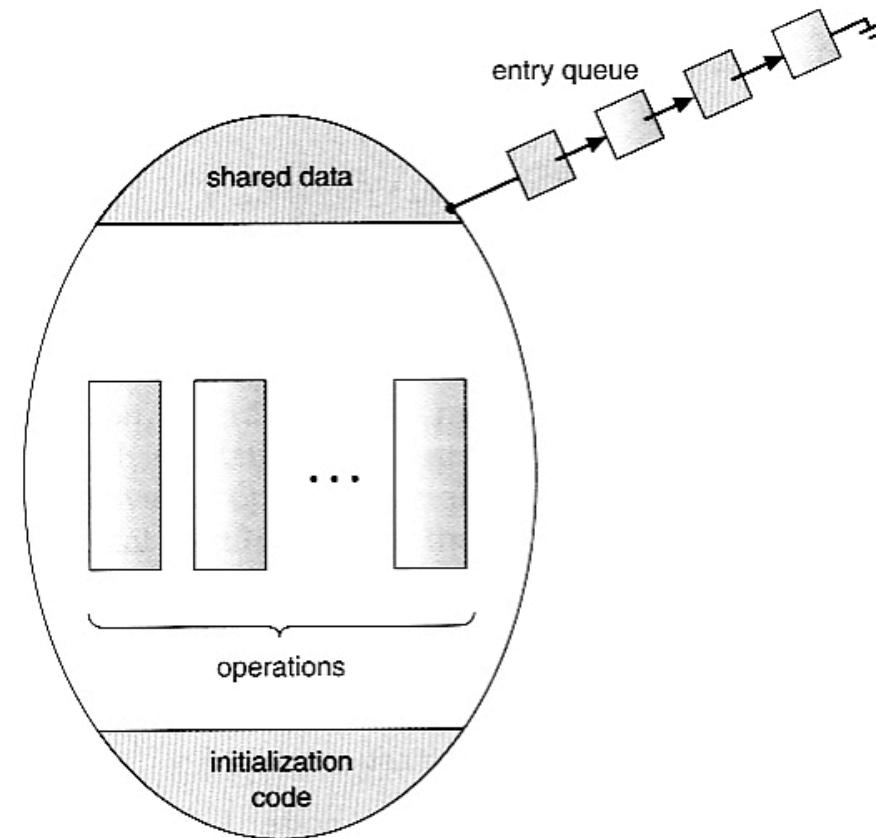


Figure 6.17 Schematic view of a monitor.

Monitores

monitor *monitor-name*

{

 declaração de variáveis compartilhadas

procedure *P1* (...) {

 ...

 }

procedure *P2* (...) {

 ...

 }

procedure *Pn* (...) {

 ...

 }

{

 código de inicialização

}

}

A cada Monitor está associado a um **mutex implícito**. A cada chamada de procedimento é executado um Down, e a cada retorno um Up.

Monitor

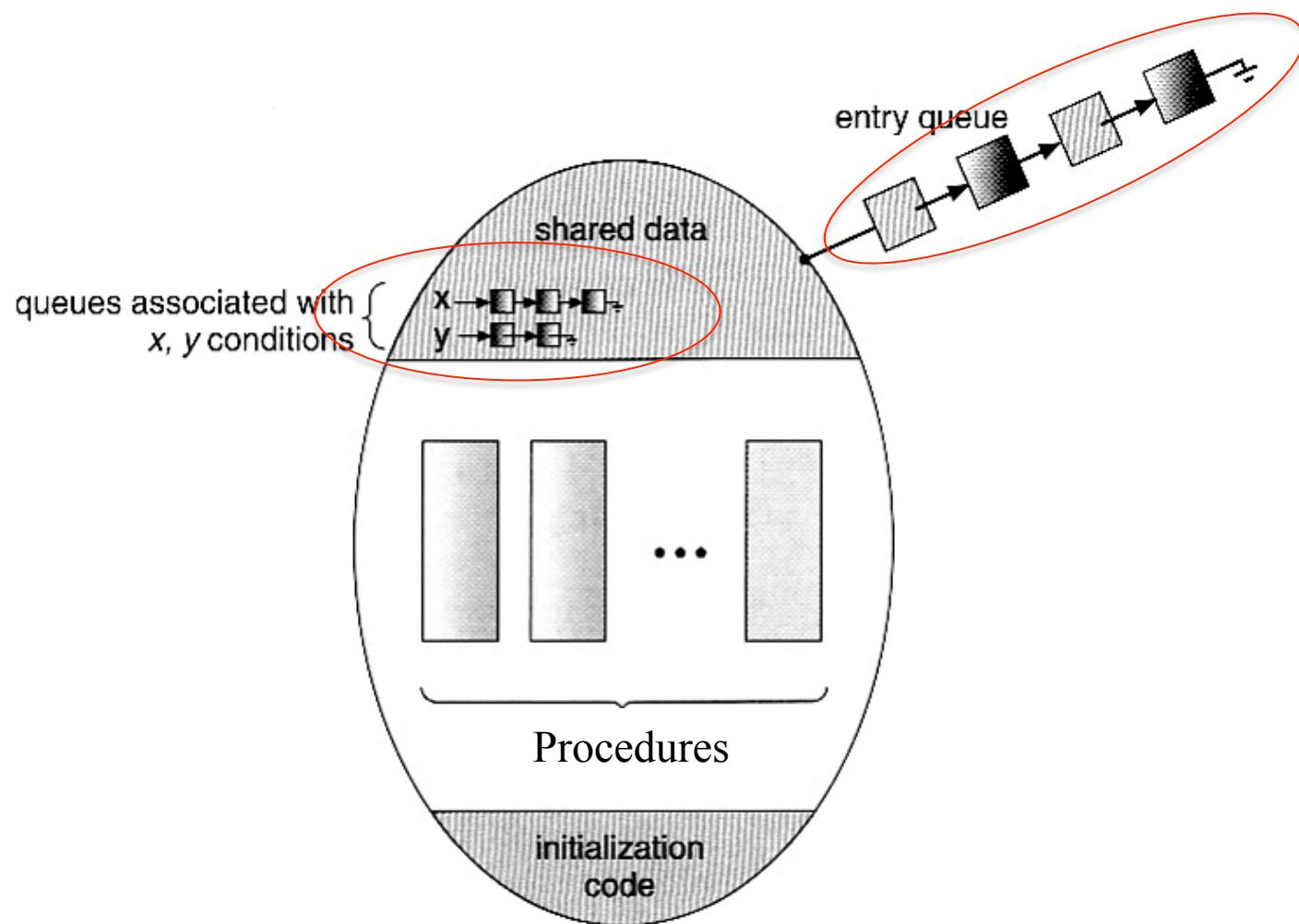


Figure 6.18 Monitor with condition variables.

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

Para implementar a sincronização em Monitores é necessário utilizar variáveis de condição:

- *wait (Cond)*: suspende a execução da thread, fazendo ela liberar o mutex, e colocando-a em estado de espera associado a condição C
- *signal (Cond)*: permite que uma thread bloqueada por *wait(C)* continue a sua execução.
Se existirem mais de uma thread bloqueada, apenas uma é liberada. Senão, não tem efeito.
- *broadcast(Cond)* acorde todas as threads esperando na condição.

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

Exemplo: Monitor em Java

```
class Buffer {  
    private char [] buffer;  
    private int count = 0, in = 0, out = 0;  
    Buffer(int size) {  
        buffer = new char[size];  
    }  
    public synchronized void Put(char c) {  
        while(count == buffer.length)  
        {  
            try { wait(); }  
            catch (InterruptedException e) { }  
            finally { }  
        }  
        System.out.println("Producing "+c+" ...");  
        buffer[in] = c;  
        in = (in + 1) % buffer.length;  
        count++;  
        notify();  
    }  
    ...  
}  
  
    public synchronized char Get() {  
        while (count == 0)  
        {  
            try { wait(); }  
            catch (InterruptedException e) { }  
            finally { }  
        }  
        char c = buffer[out];  
        out = (out + 1) % buffer.length;  
        count--;  
        System.out.println("Consuming " + c + " ...");  
        notify();  
        return c;  
    }  
}
```

Principal Diferença entre Monitores e Semáforos

Monitor:

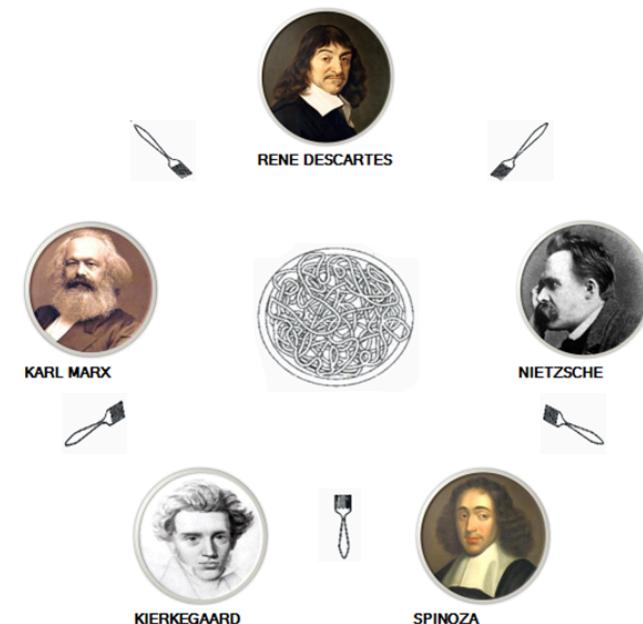
- só serve para threads, já que processos não têm acesso a dados compartilhados (as instâncias de monitor)
- Monitor é um mecanismo mais abstrato, que estende o encapsulamento para a exclusão mútua

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

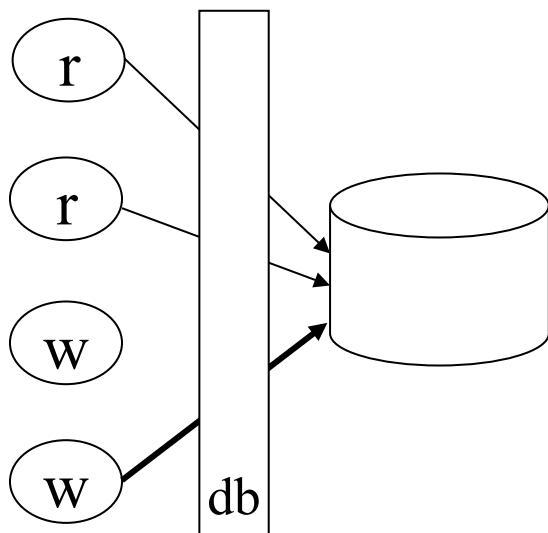
Problemas Clássicos de Concorrência

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



Problema dos Leitores e Escritores Excludentes: solução com semáforos

Vários leitores podem entrar a 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();
    }
}

/* 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);
    }
}
```

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 soneca, até que chega um novo cliente



O Barbeiro Dorminhoco: usando Semáforos

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

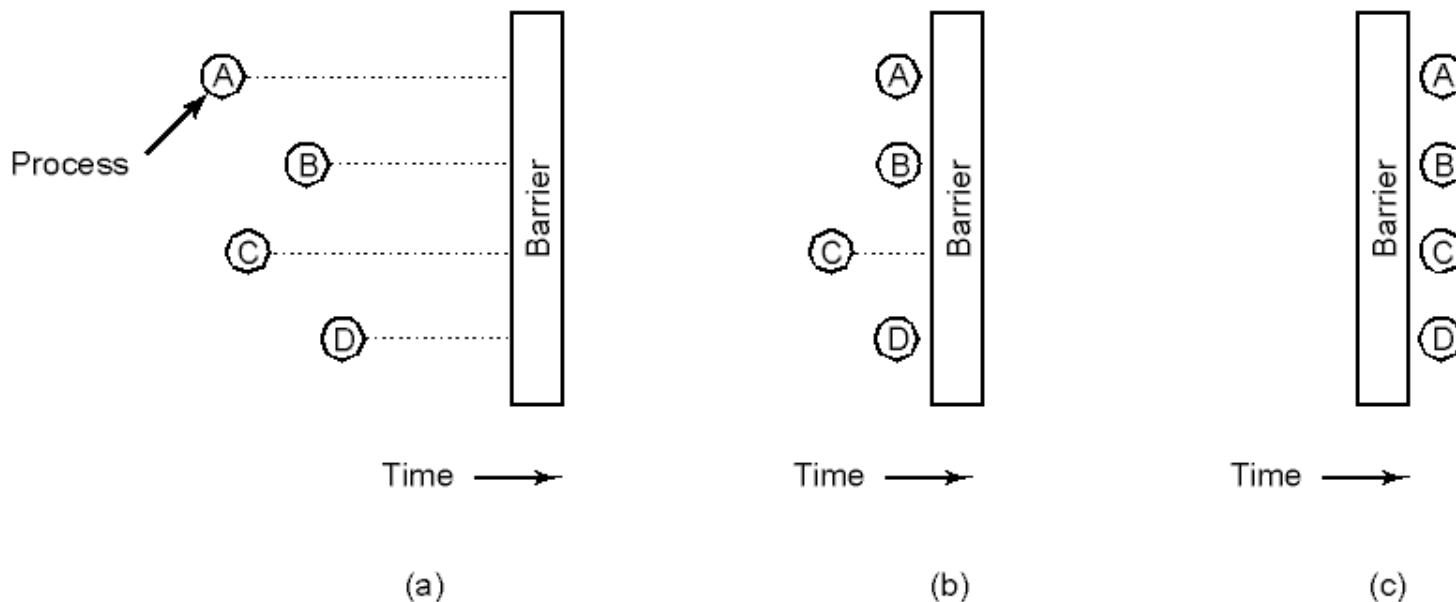
typedef int semaphore;    /* use your imagination */

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 {                /* shop is full; do not wait */
        up(&mutex);
    }
}
```

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

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

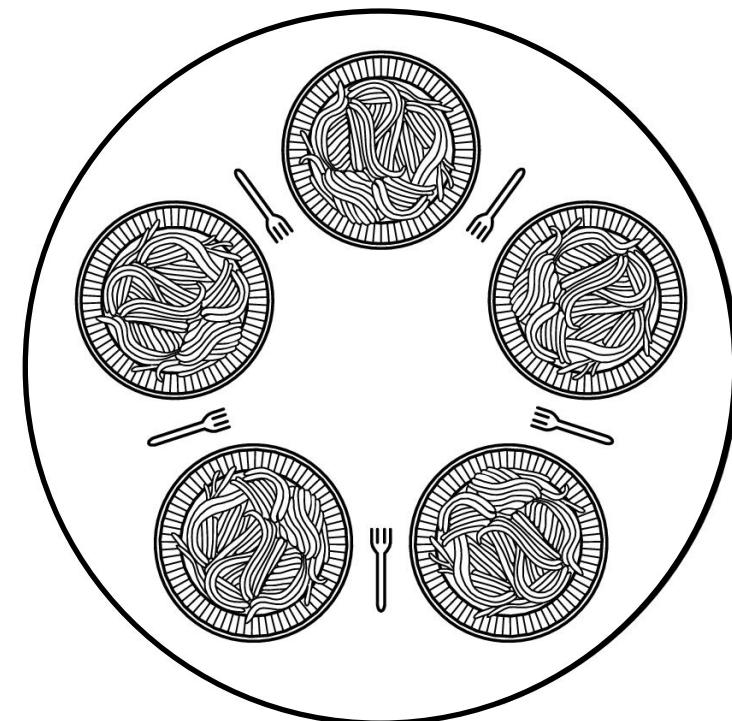
    down(&m)
        count--;
        if (count == 0) last= true;
    up(&m)
    if (NOT last) down (&barrier); // espera pelos demais processos
    else for (i=0; i< N; i++) up (&barrier); // desbloqueia todos
    ...
}
```

O Jantar dos Filósofos

Sincronização para compartilhamento de recursos 2 a 2

Definição do Problema:

- Filósofos só tem 2 estados: comendo e pensando;
- Para comer, precisam de dois garfos, cada qual compartilhado com os seus vizinho;
- Só é possível pegar um garfo por vez;



Questões globais:

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

Relação entre o Jantar dos Filósofos e Sistemas Operacionais?

Isso é um problema que pode ocorrer em Sistemas Operacionais?

Considere a situação:

Cada processo precisa Modificar 2 arquivos ao mesmo tempo, e cada um desses esses arquivos é compartilhado com outros processos.

Em um processamento em “pipeline circular” (apagando registro de ARQ_i e inserindo novo registro modificado em ARQ_{i+1})

Possível solução(?):

Lock fileA

Lock fileB

Write/Update information to fileA and fileB

Release the locks

O Jantar dos Filósofos

Solução trivial:

- Se há N recursos (garfos), deixar que no máximo $N-1$ filósofos sentem à mesa.
- para isso, usar um semáforo contador, “mesa”, e inicializá-lo com o valor $N-1$.

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 o garfo esquerdo, e se conseguir, espera pela devolução do garfo direito.

Problema: 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 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)           /* i: philosopher number, from 0 to N-1 */
{
    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);
    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

Solução de Dijkstra:

- Estabelecer uma sequência de garfos ($G_1, G_2, G_3, \dots, G_{i \bmod N}$) e estabelecer que sempre deve-se pegar primeiro o garfo com menor ID – assim quebra-se a dependencia circular!

Solução de Chandy/Misra:

- Se dois filósofos estão competindo por um mesmo garfo, dê prioridade para o filósofo de menor ID
- Garfos podem estar sujos ou limpos
- Se um filósofo pegar um garfo limpo, o segura, mas se estiver sujo, devolve-o.
- Antes de passar o garfo para outro filósofo, o garfo é limpo.

Exercício: Escreva pseudo-códigos para essas soluções e mostre que elas evitam impasse.

Envio de Mensagens

- É uma forma natural de interação entre processos
- Duas primitivas:
 - **send**(canal, &message) – pode ser bloqueante ou não-bloqueante
 - **receive**(canal, &message) – pode ser bloqueante ou não-bloqueante
- Processos não precisam indicar o processo da outra ponta
 - definem um canal lógico de comunicação
 - mensagens podem ser tipadas ou não
 - possuem um cabeçalho e um *payload*
- O canal pode ser:
 - Entre processos co-localizados ou distribuídos
 - confiável ou não-confiável
 - Ponto-a-ponto ou Ponto-a-multiponto
 - ter uma capacidade máxima ou ser um *stream*

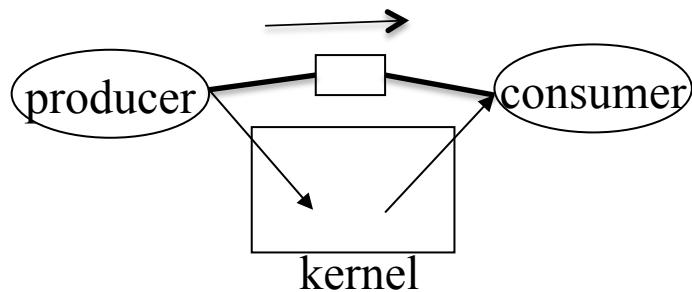
Envio de Mensagens

- 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()` geralmente bloqueia se o outro processo não tiver dado o `send()`
 - Mas `send()` pode ser **assíncrono** (com buffer de mensagens) ou **síncrono (Rendezvous)**
 - Em comunicação do tipo *Rendezvous* podem ocorrer impasses
 - Comunicação confiável: quando a comunicação é remota (pela rede) mensagens podem ser perdidas: → são necessárias confirmações, timeouts e re-transmissões

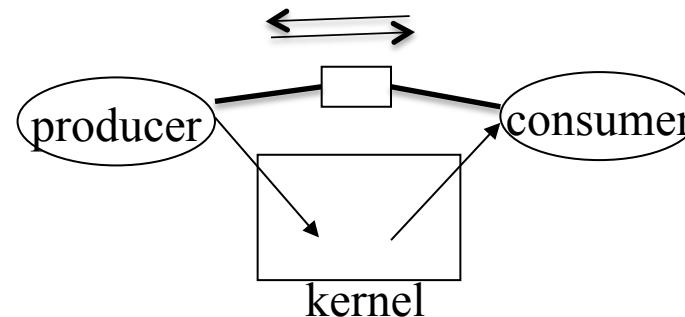
Tipos de Envios de Mensagem

Entre processos co-localizados

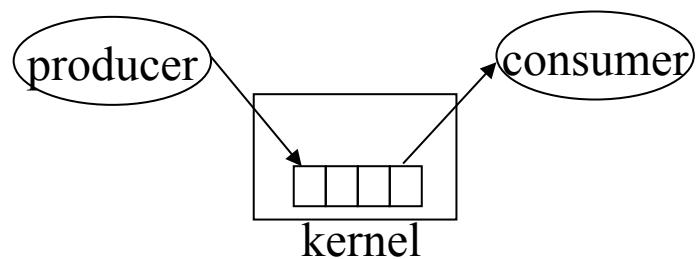
Rendezvous, unidirecional



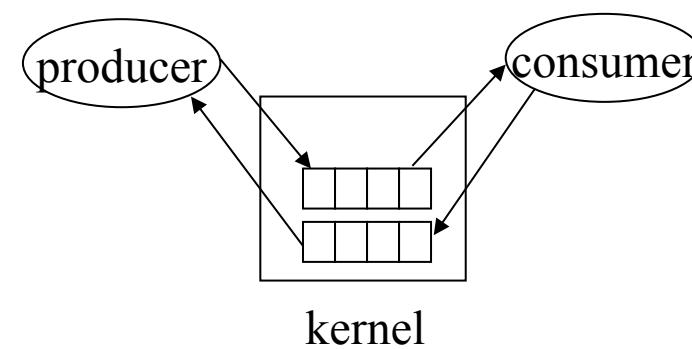
Request-Reply síncrono



Comunicação assíncrona



Request-Reply assíncrono



Mecanismos de Comunicação em Unix

- Pipes/ FIFO
- Message Queues
- Sockets

FIFO (Named pipe)

- Similar ao pipe, só que possui um nome e cria uma entrada no diretório corrente.
- Pode ser usado por quaisquer dois processos
- Para criação:
 - mknod,
 - `mknod("/tmp/mypipe", S_IFIFO, 0)`
- Abre-se e fecha-se como se fosse um arquivo para leitura e/ou escrita
 - `fd = open ("/tmp/mypipe", 0)`
- O controle de acesso é usando as permissões rwx de Usu unlink para remove-lo

Message Queue

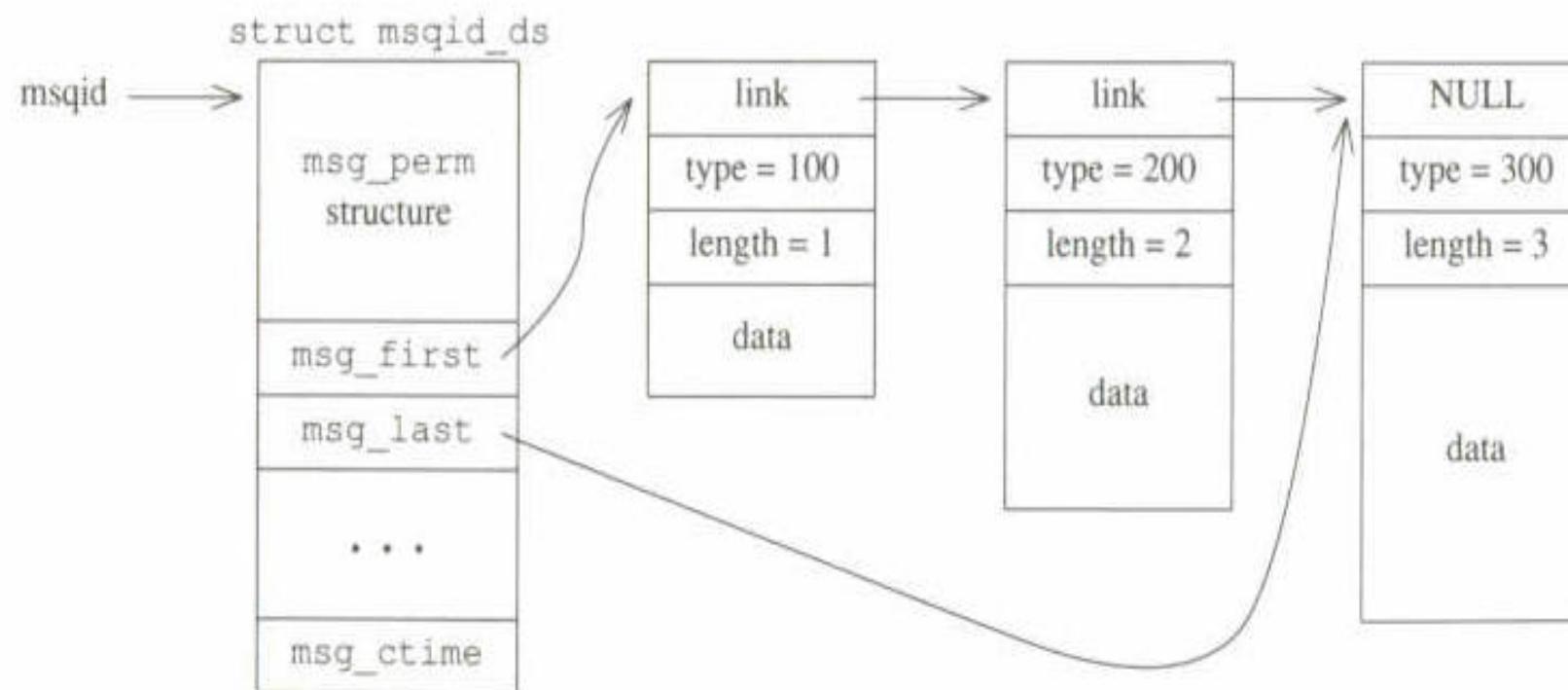
- Adota o paradigma de mailbox: mensagens podem ter qualquer tamanho.
 - msgqID (chave que identifica a message queue)
 - Cada mensagem tem:
 - Type
 - Length
 - Data
 - msgget() – para criar uma msg queue
 - msgsnd(), msgrcv() – para enviar e receber uma msg
 - msgctl() – para remover uma message queue

Message Queue: Chamadas de Sistema

- System calls:
 - `int msgget (key_t key, int flag)` Creates or accesses a message queue, returns message queue identifier
 - `int msgsnd(int msqid, const void *msgp, size_t msgsz, int flag)` Puts a message msgp in the queue
 - `int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg)` Receives a message and stores it to msgp
 - `msgtype`: Messages can be typed and each type defines a communication channel
 - `int msgctl(int msqid, int cmd, struct msqid_ds *buf)` provides a variety of control operations on a message queue (e.g. remove)
- Processos são bloqueados quando:
 - tentam ler de uma queue vazia
 - Tentam escrever em uma queue cheia
- Processos podem se restringir a receber mensagens apenas de certo tipo (portanto, pode não ser FIFO)

Estrutura da Message Queue

Estrutura mantida no núcleo:



Exemplo de Message Queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define BUFFSIZE 128
#define PERMS          0666
#define KEY ((key_t) 7777)
main() {
    int i, msqid;
    struct {
        long      m_type;
        char     m_text[BUFFSIZE];
    } msgbuffs, msgbuffr;

    if ((msqid = msgget(KEY, PERMS | IPC_CREAT)) < 0) error("msgget perror");
    msgbuffs.m_type = 1L;
    strcpy(msgbuffs.m_text, "a REALLY boring message");
    if (msgsnd(msqid, &msgbuffs, BUFFSIZE, 0) < 0) perror("msgsnd error");
    printf("the message sent is: %s \n", msgbuffs.m_text);
    if (msgrcv(msqid, &msgbuffr, BUFFSIZE, 0L, 0) != BUFFSIZE) perror("msgrcv err");
    printf("the message received is: %s \n", msgbuffr.m_text);
    // remove msg
    if (msgctl(msqid, IPC_RMID, (struct msqid_ds *)0)< 0) perror("IPC_RMID err");
    exit(0);
}
```

Sockets

O que é um socket?

Socket é uma abstração de um *ponto final de comunicação* que pode ser manipulado como um descritor de arquivo.

- Através dele, mensagens podem ser recebidas e enviadas
- São criados para um domínio de comunicação,
- **Domínio de comunicação** é a especificação de propriedades comuns da comunicação entre processos (p.ex. o formato dos endereços)
- Exemplos de domínio:
 - UNIX domain (para comunicação local),
 - Internet domain.

Sockets

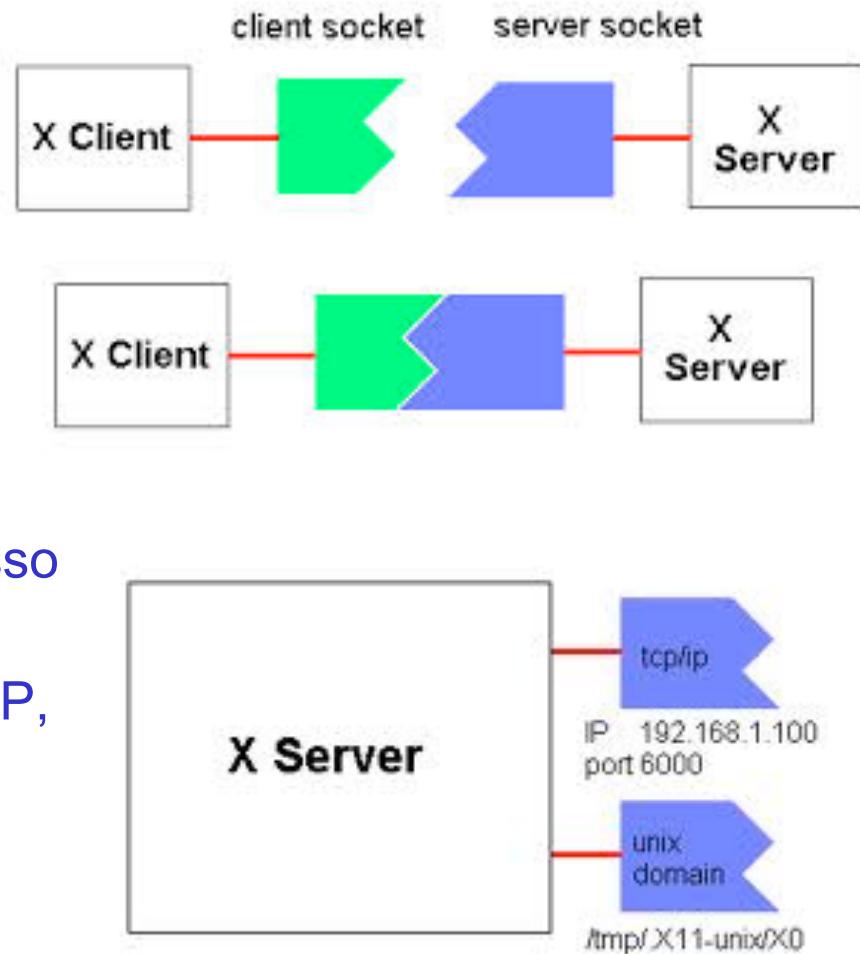
Um socket é um *ponto de comunicação direcional* de um processo:

É um objeto lógico através do qual mensagens podem ser enviadas e recebidas.

Processso recebe um descritor de socket (similar a um file descriptor)

Principal Vantagem:

- não precisar conhecer o outro processo
- É genérico para diferentes domínios/protocolos de comunicação (TCP/UDP, Unix domain),



Tipos de Internet Sockets

- Stream socket
 - Orientado a conexão
 - Comunicação bi-direcional
 - Confiável (sem perdas), entrega de mensagens na ordem
 - Tipicamente usa o Transmission Control Protocol (TCP)
 - e.g. ftp, telnet, ssh, http
- Datagram socket
 - Orientado a pacotes, não mantém uma conexão aberta,
 - cada pacote (data-gramma) é transmitido isoladamente
 - Tipicamente usa User Datagram Protocol (UDP)
 - e.g. telefonía IP (skype, etc.)

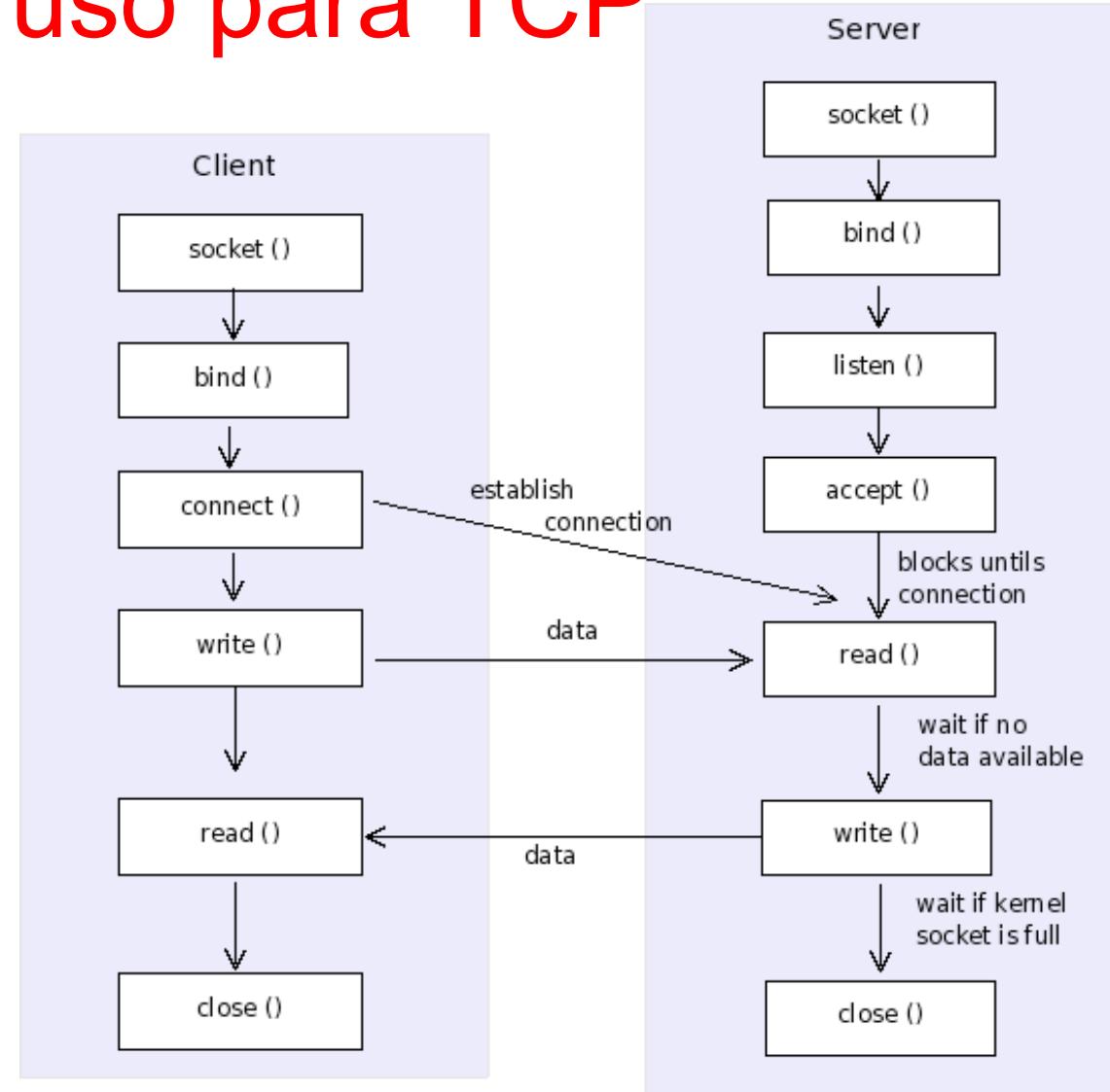
Sockets – uso para TCP

bind() atribui um endereço IP ao socket

- Nome depende do tipo de protocolo (address families):
- AF_INET
- AF_INET6
- AF_APPLETALK
- ...

connect() faz uma ligação com outro endereço (host_ID: porta)

listen() informa que o socket passado como argumento é passivo, ou seja, apenas para aceitar novas conexões



accept() processa o próximo pedido de conexão e cria um novo socket conectado e retorna o descriptor desse socket criado

Cliente TCP

- Obtendo o descriptor do socket

```
soc= socket(AF_INET, SOCK_STREAM, 0);
```

- Setando o endereço

```
struct sockaddr_in sin;
struct hostent *host = gethostbyname (argv[1]);
unsigned int server_address = *(unsigned long *) host->h_addr_list[0];
unsigned short server_port = atoi (argv[2]);

memset (&sin, 0, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = server_address;
sin.sin_port = htons (server_port);
```

No Cliente TCP (cont.)

- Fazendo a conexão

```
if (connect(chat_sock, (struct sockaddr *) &sin, sizeof (sin)) < 0) {  
    perror("connect");  
    printf("Cannot connect to server\n");  
    abort();  
}
```

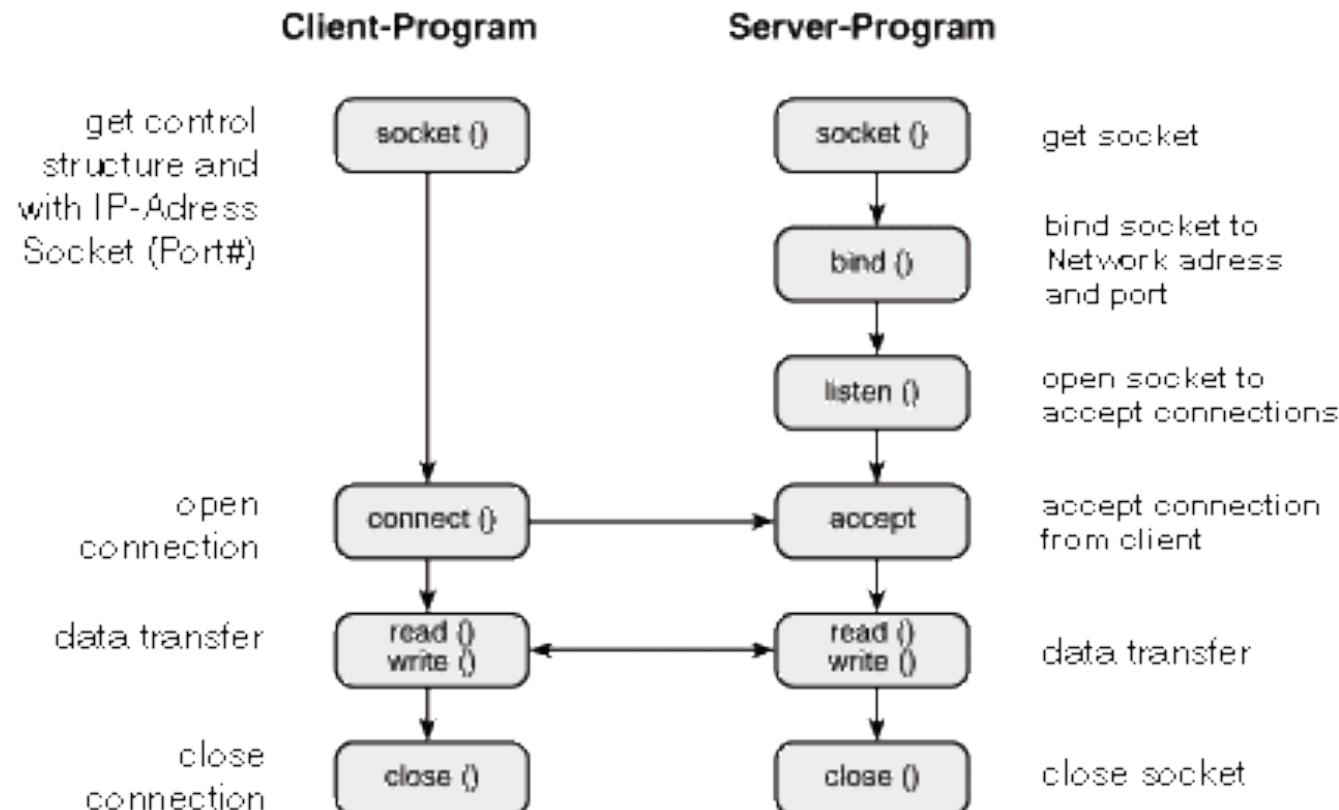
- Enviando os dados pelo descriptor

```
int send_packets(char *buffer, int buffer_len) {  
    sent_bytes = send(chat_sock, buffer, buffer_len, 0);  
    if (send_bytes < 0) {  
        perror ("send");  
    }  
    return 0;  
}
```

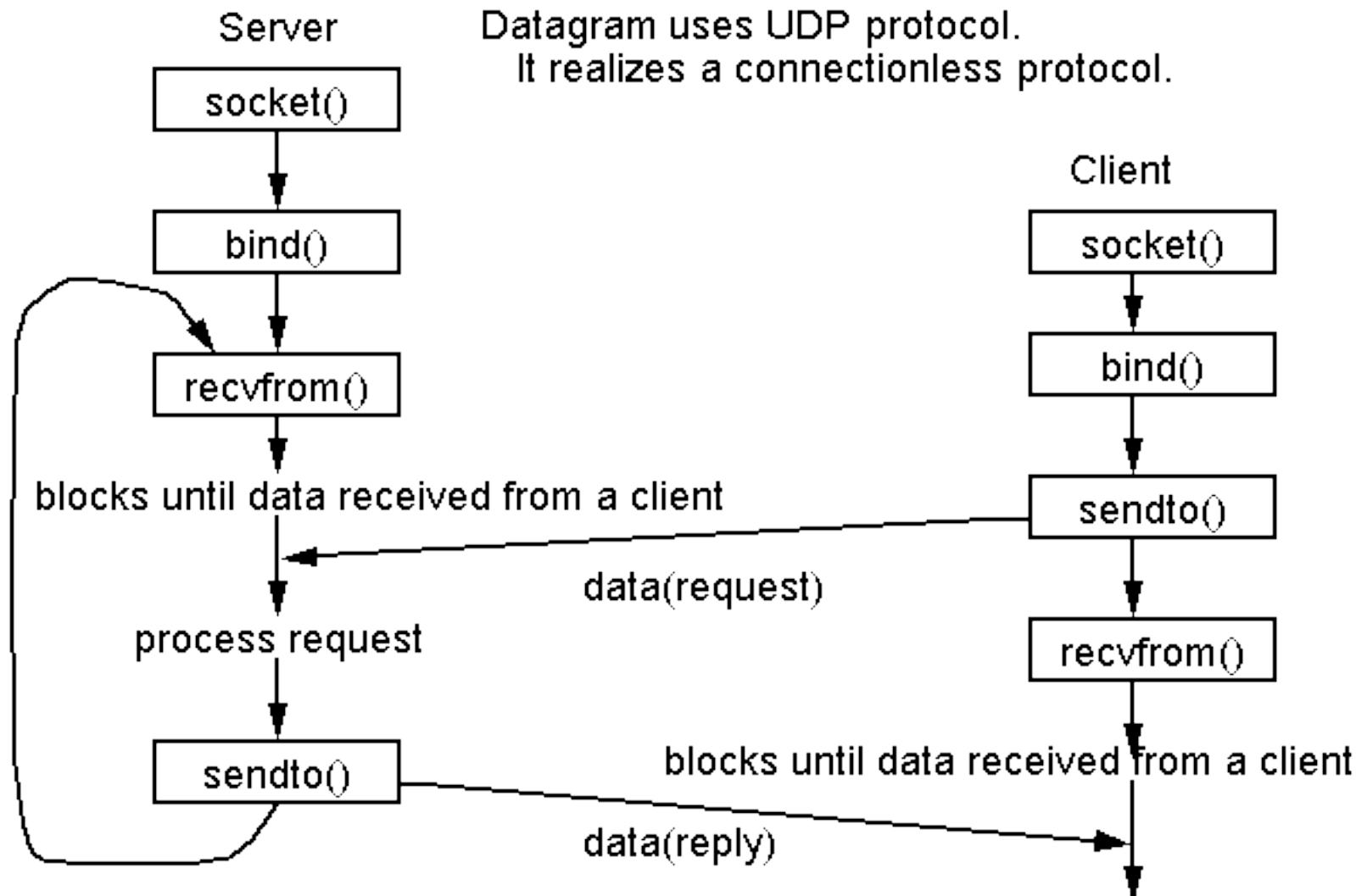
No Servidor TCP

- Cria stream socket: *socket()*
- Atribui porta ao socket: *bind()*
- Espera por conexão de cliente: *listen()*
- `while(1)`
 - Aceita conexão de cliente e cria novo socket Nsock: *accept()*
 - Recebe dado do cliente: *recv(Nsock, ...)*
 - Envia resposta para client: *send(Nsock, ...)*

Sockets com TCP: Interação Cliente-Servidor

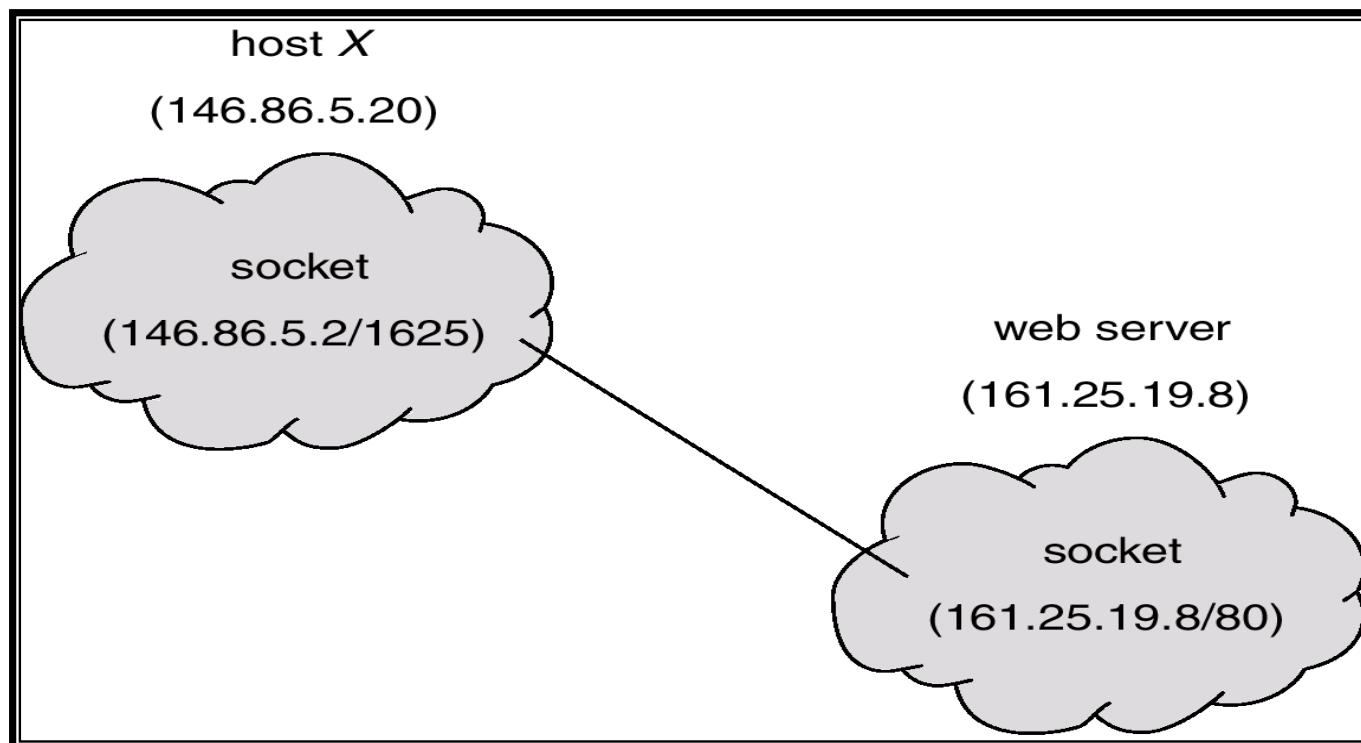


Sockets com UDP



Sockets e IP

- Concatenação de endereço IP address e porta: **161.25.19.8:1625** se refere a porta **1625** no host **161.25.19.8**
- **Localhost:** 127.0.0.1
- Uma conexão TCP consiste de um par de sockets



Portas reservadas para serviços Internet

Keyword	Number	Protocol(s)	Description
tcpmux	1	TCP, UDP	TCP Port Service Multiplexer
echo	7	TCP, UDP	Echo
discard	9	TCP, UDP	Discard
systat	11	TCP	Active Users
daytime	13	TCP, UDP	Daytime (RFC 867)
qotd	17	TCP	Quote of the Day
msp	18	TCP, UDP	message send protocol
chargen	19	TCP, UDP	Character Generator
ftp-data	20	TCP, UDP	File transfer default data
ftp	21	TCP, UDP	File transfer control
ssh	22	TCP, UDP	Remote login protocol
telnet	23	TCP, UDP	Telnet
smtp	25	TCP, UDP	Simple Mail Transfer
time	37	TCP, UDP	Time
rlp	39	TCP, UDP	Resource location protocol
nameserver	42	TCP, UDP	Host name server
whois	43	TCP, UDP	Who is
re-mail-ck	50	TCP, UDP	Remote mail checking protocol
domain	53	TCP, UDP	Domain name server
bootps	67	TCP, UDP	Bootstrap protocol server
bootpc	68	TCP, UDP	Bootstrap protocol client
tftp	69	TCP, UDP	Trivial file transfer protocol
gopher	70	TCP, UDP	Gopher
finger	79	TCP, UDP	Finger
www	80	TCP, UDP	World wide web or HTTP

kerberos	88	TCP, UDP	Kerberos
supdup	95	TCP, UDP	SUPDUP
hostname	101	TCP, UDP	NIC Host Name Server
iso-tsap	102	TCP, UDP	ISO-TSAP Class 0
csnet-ns	105	TCP, UDP	CCSO name server protocol
rlogin	107	TCP, UDP	Remote Login Service
pop-2	109	TCP, UDP	Post Office Protocol - Version 2
pop-3	110	TCP, UDP	Post Office Protocol - Version 3
sunrpc	111	TCP, UDP	SUN Remote Procedure Call
auth	113	TCP, UDP	Authentication Service
sftp	115	TCP, UDP	Simple File Transfer Protocol
uucp-path	117	TCP, UDP	UUCP Path Service
nntp	119	TCP, UDP	Network News Transfer Protocol
nntp	123	TCP, UDP	Network Time Protocol
netbios-ne	137	TCP, UDP	NETBIOS Name Service
netbios-dgram	138	TCP, UDP	NETBIOS Datagram Service
netbios-ssn	139	TCP, UDP	NETBIOS Session Service
imap	143	TCP, UDP	Internet Message Access Protocol
snmp	161	TCP, UDP	SNMP
snmp-trap	162	TCP, UDP	SNMPTRAP
cmip-man	163	TCP, UDP	CMIP/TCP Manager
cmip-agent	164	TCP, UDP	CMIP/TCP Agent
xdmcp	177	TCP, UDP	X Display Manager Control Protocol
nextstep	178	TCP, UDP	NextStep Window Server
bgp	179	TCP, UDP	Border Gateway Protocol
prospero	191	TCP, UDP	Prospero Directory Service
irc	194	TCP, UDP	Internet Relay Chat Protocol
smux	199	TCP, UDP	SMUX

Outros Mecanismos de Comunicação

Chamada Remota de Procedimento/Métodos (RPC, RMI)

Message Passing Interface (MPI)

Publish/Subscribe

Chamada Remota de Procedimento

(Remote Procedure Call, Remote Method Invocation)

Remote Procedure Call (RPC) cria uma abstração de uma chamada de procedimento entre processos executando em máquinas uma rede;

- **Stubs** – proxy no lado do cliente para o procedimento no lado servidor
 - **stub cliente** encontra o servidor, estabelece uma conexão e faz o marshalling dos parâmetros
 - **stub do servidor** (skeleton) recebe a mensagem, desempacota os parâmetros, faz a chamada do procedimento, empacota o resultado e envia de volta ao stub cliente.

Interface Definition Language (IDL)

É a linguagem em que se especifica a interface de um processo (servidor). É independente de linguagem de programação.

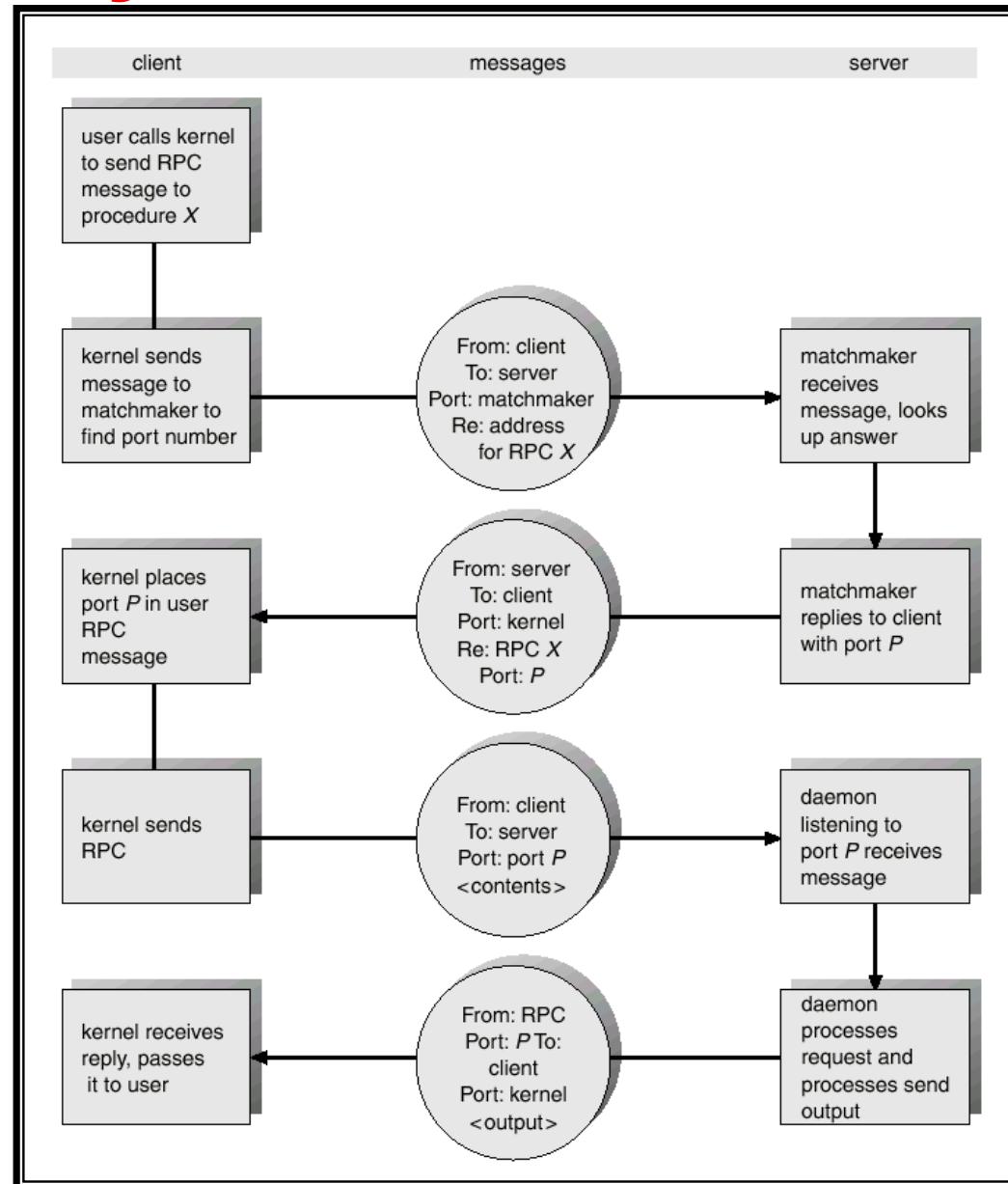
Além dos tipos atômicos (long, double, string) e de structs para estruturas de dados mais complexas, também define os parâmetros de entrada, saída e os tipos de retorno dos procedimentos remotos

```
module BankExample {
    typedef float MoneyType;
    struct NameType {
        string first;
        string last;
    };
    interface BankAccount {
        MoneyType balance();
        MoneyType deposit(
            in MoneyType amount);
        MoneyType withdraw(
            in MoneyType amount);
    };
    interface CheckingAccount :
        BankAccount {
        exception BadCheck {
            MoneyType fee;
        };
        MoneyType writeCheck(
            in MoneyType amount)
        raises (BadCheck);
    };
    interface BankManager {
        CheckingAccount openAccount(
            in NameType name,
            in MoneyType deposit,
            out MoneyType balance);
    };
};
```

Figure 1: Bank IDL Example

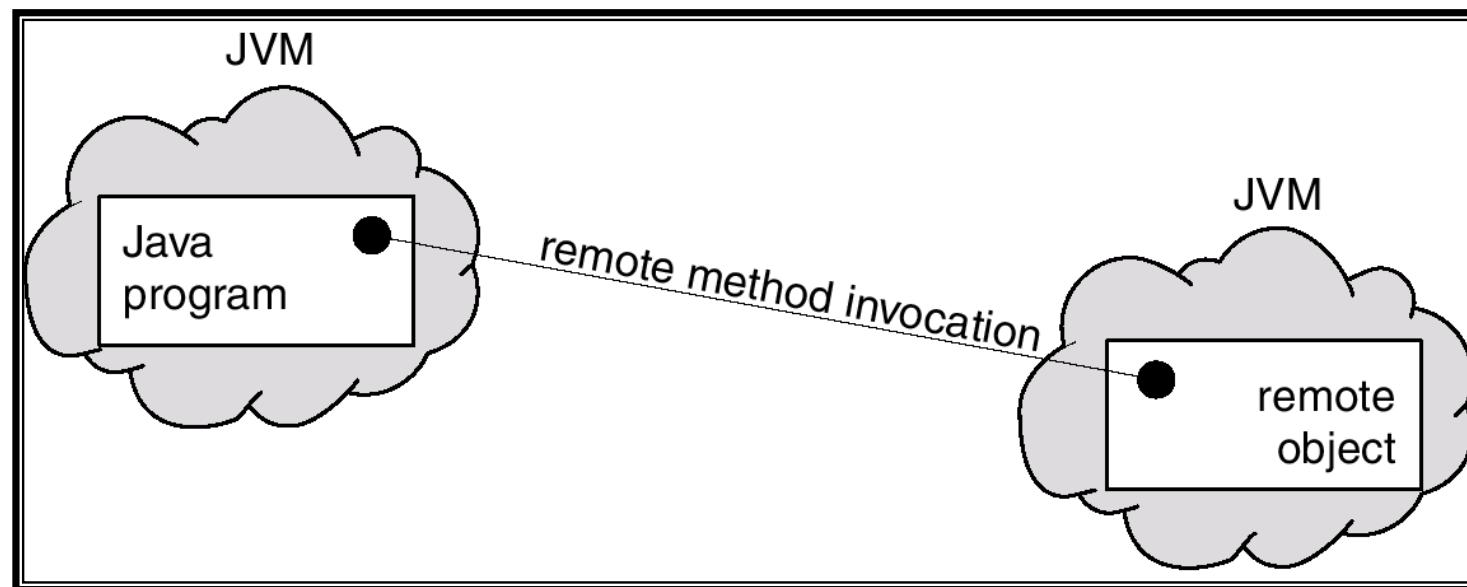
Execução de um RPC

Muitas vezes envolve um broker (Matchmaker), que localiza o IP e a porta do processo que é o “executor” do procedimento procurado.

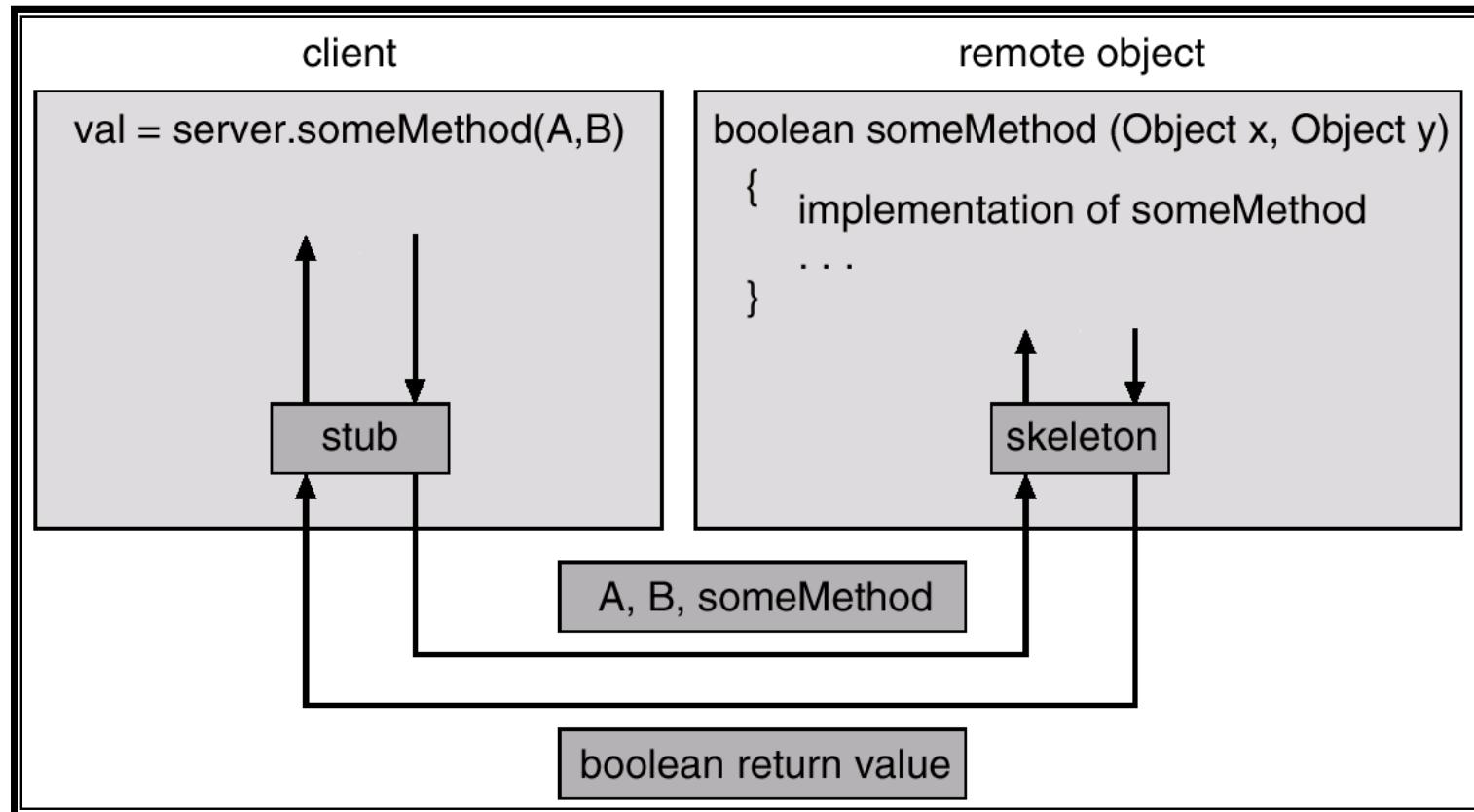


Remote Method Invocation

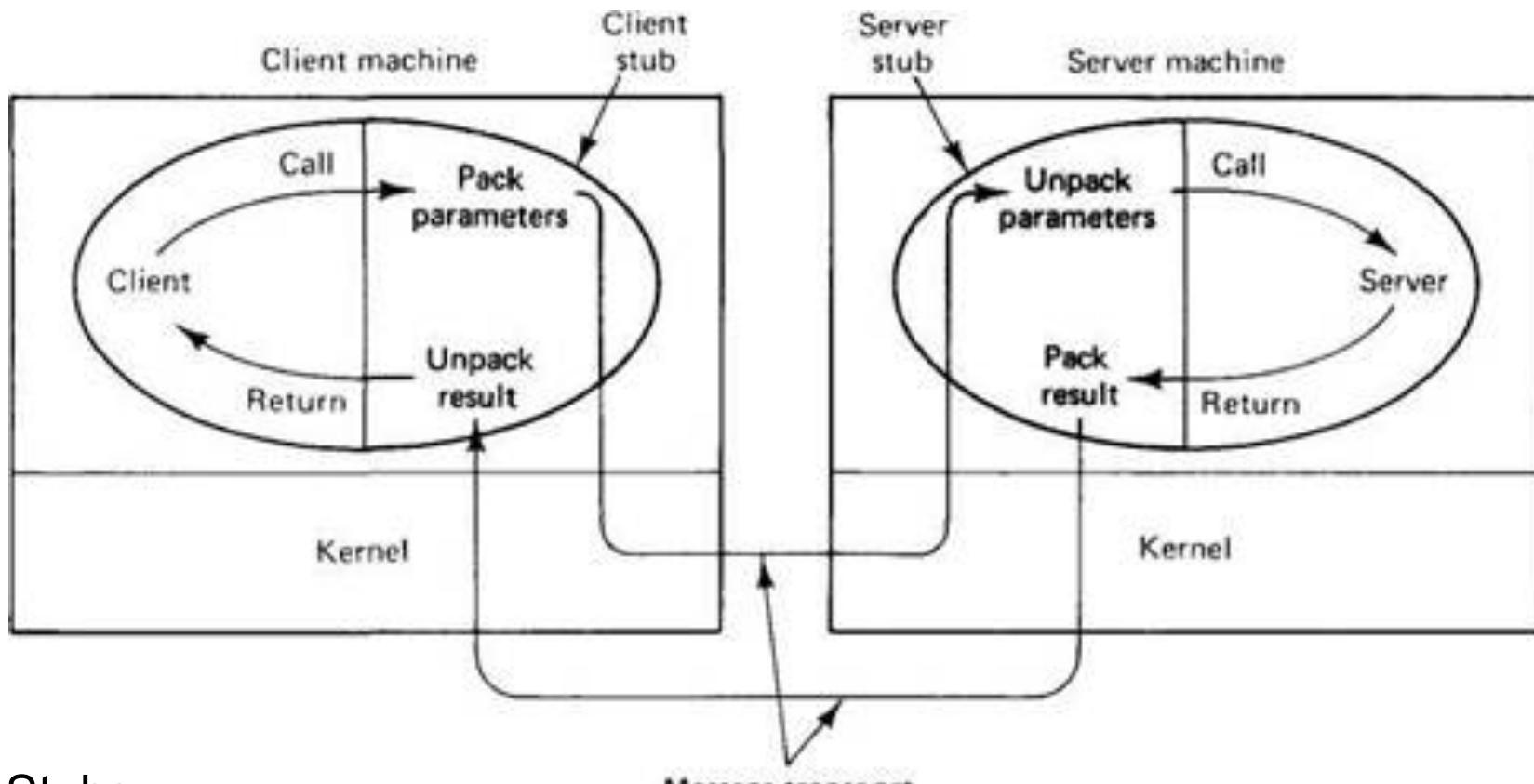
- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.



RPC: Funcionamento



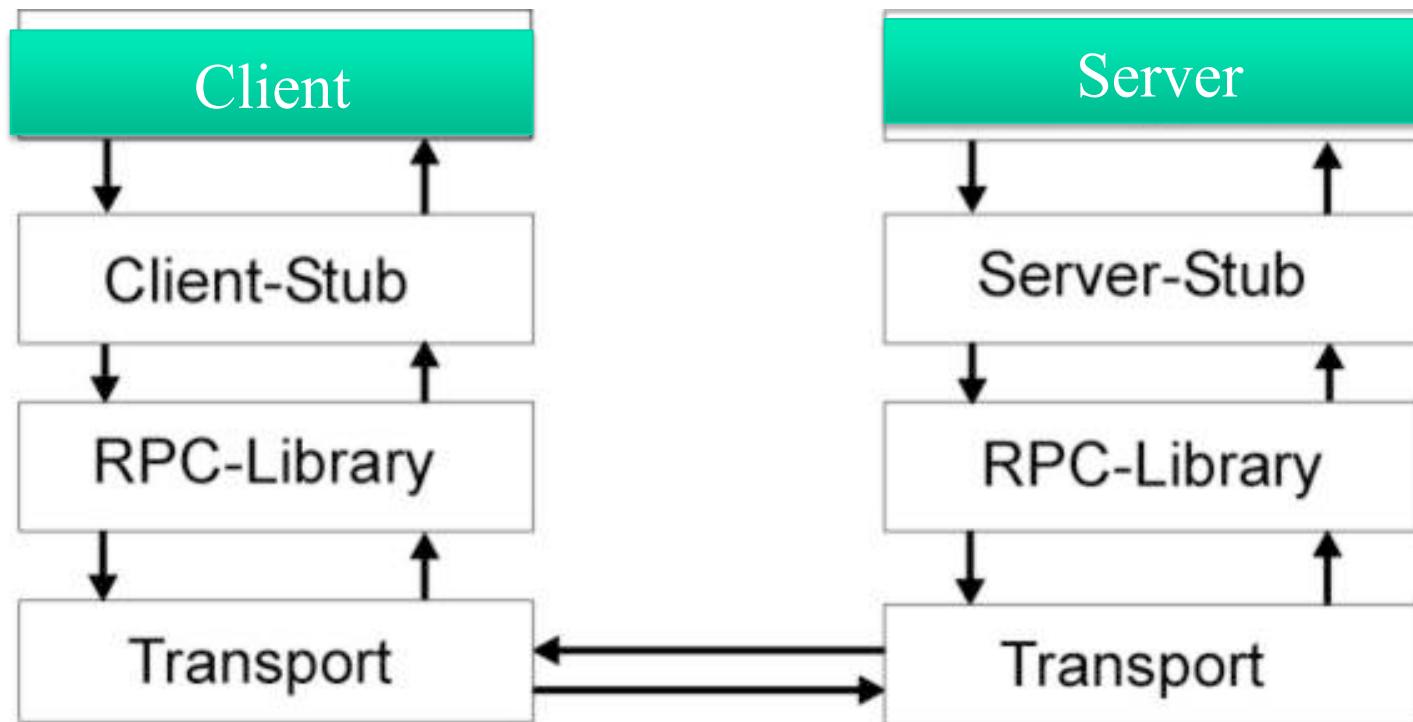
RPC: Funcionamento



Stubs:

- empacotamento dos parâmetros dos procedimentos de/para mensagens;
- conversão de dados da representação específica da lingu. de programacão para uma representação intermediária, e vice-versa

RPC



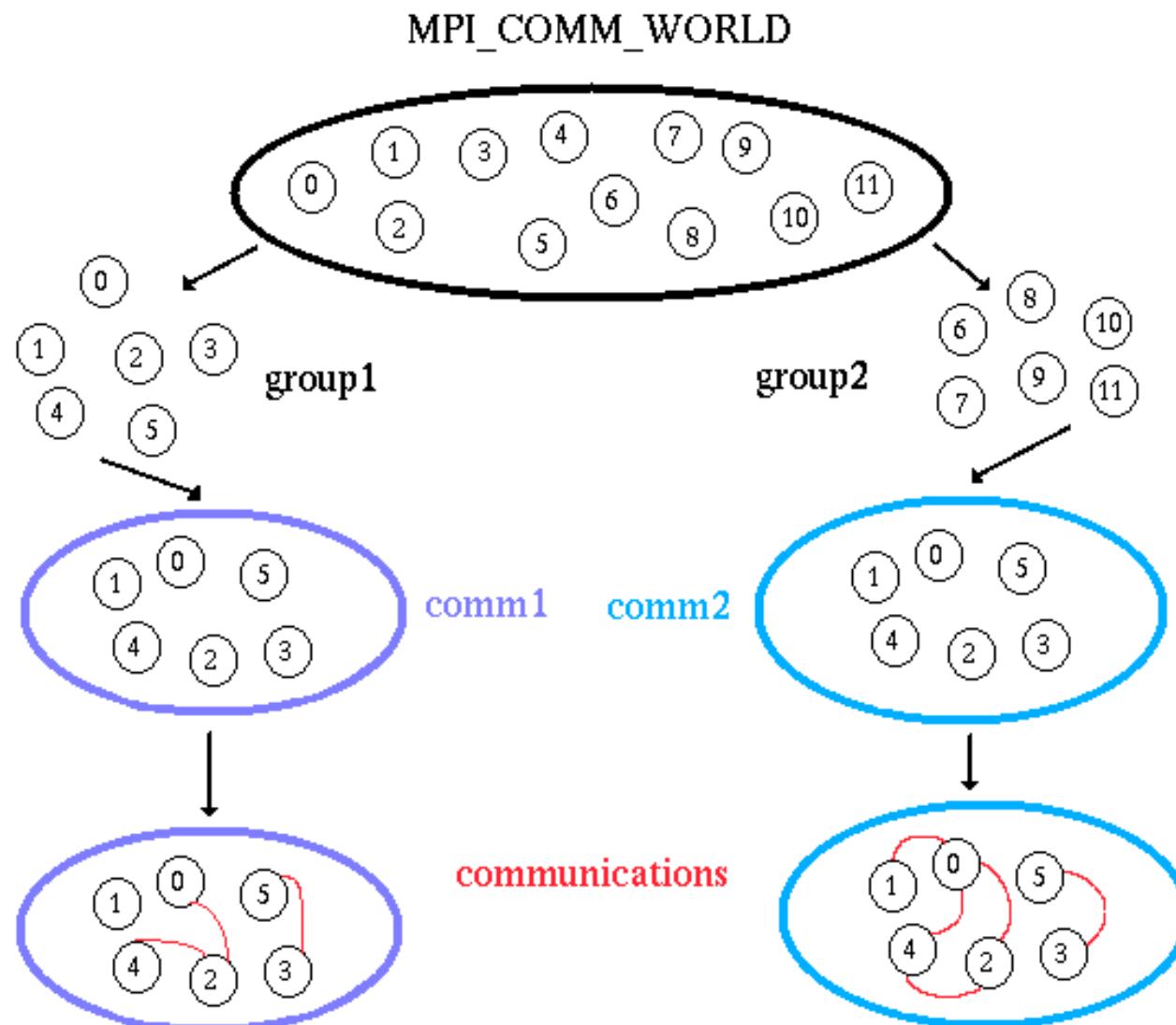
Biblioteca RPC: para registro dos stubs, multiplexação de sockets, e controle do protocolo de transporte;

MPI

- MPI têm a noção de “communicator”
- Consiste de um grupo de processos e um contexto
- Uma mensagem deve ser enviada e recebida no mesmo contexto.
- Comunicator é ma forma de agrupar dinamicamente o conjunto de processos que irão interagir
- Existe um communicator default que engloba inicialmente todos os processos criados.

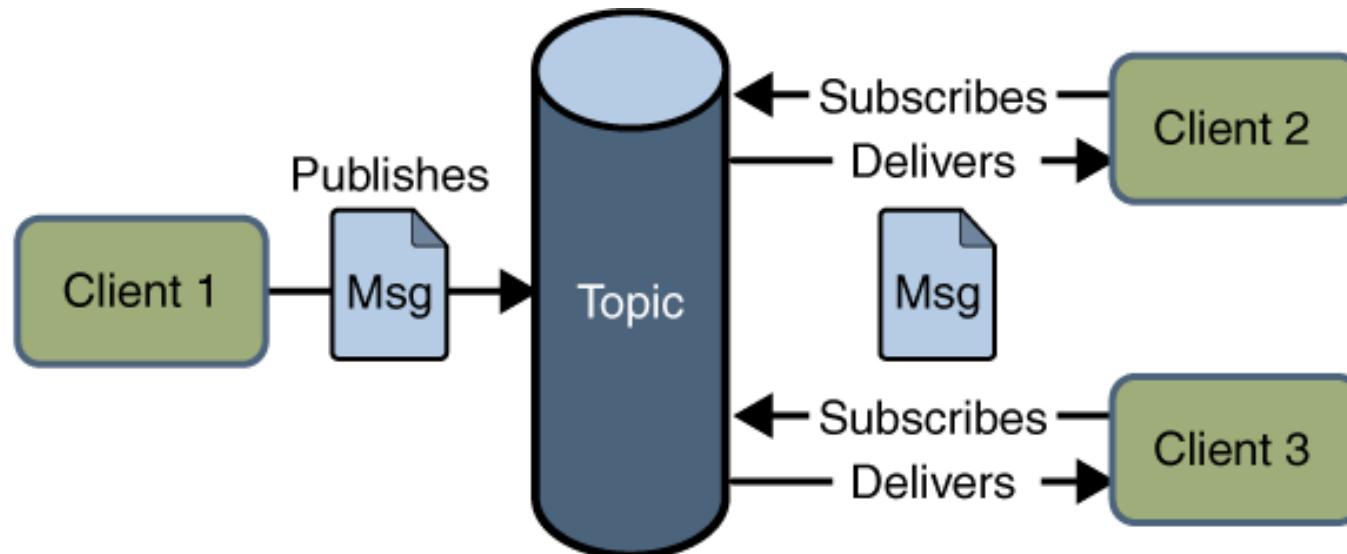
- Cada mensagem é composta de (address, count, datatype)
- Um datatype pode ser tipo básico da linguagem; um vetor de *datatypes*; um vetor indexado de blocos de *datatype*; uma estrutura arbitária de *datatypes*
- MPI provê primitivas para construir datatypes customizados

MPI Communicator



Publish/Subscribe

- Clientes se registram em um tópico para receber mensagens (subscribe), possivelmente com um *filtro*
- Publicadores postam mensagens nos tópicos, que middleware distribui para todos os subscribers
- Comunicação 1-N
- Interação com **desacoplamento referencial e espacial**
- Possivelmente, também com **desacoplamento temporal**



Exemplos: XMPP. DDS. MQTT, Amazon SNS. Azure Service Bus, etc.