Sistemas Operacionais I

Profa. Kalinka Regina Lucas Jaquie Castelo Branco
kalinka@icmc.usp.br

Universidade de São Paulo

Outubro de 2020

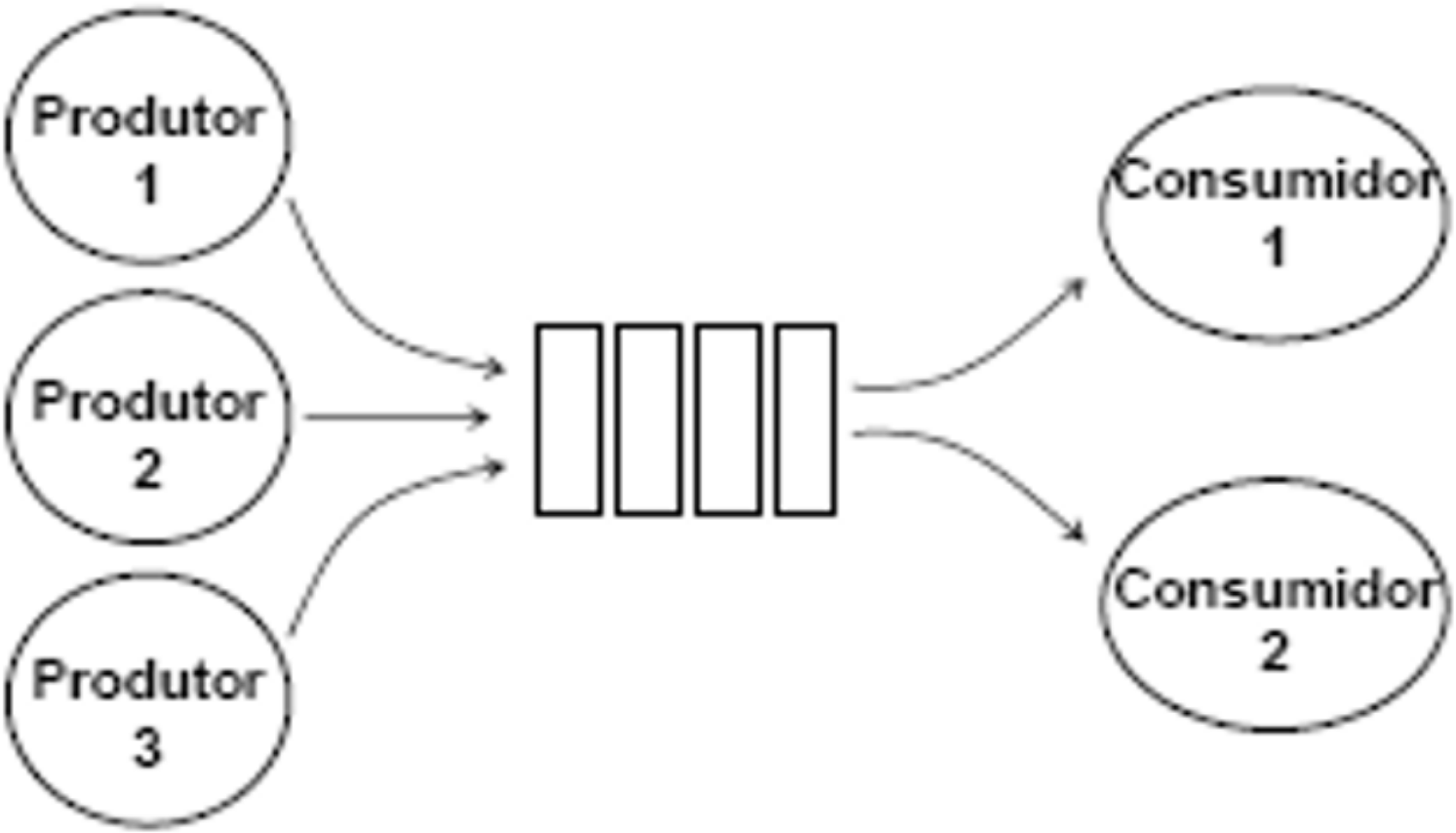
Problemas Clássicos de Comunicação entre Processos

Produtor/Consumidor

- Um sistema é composto por entidade produtoras e entidades consumidoras.
- Entidades Produtoras
  - Responsáveis pela produção de itens que são armazenados em um *buffer* (ou em uma fila).
  - Itens produzidos podem ser consumidos por qualquer consumidor.
- Entidades Consumidoras
  - Consomem os itens armazenados no *buffer* (ou fila).
  - Itens consumidos podem ser de qualquer produtor.

Problemas Clássicos de Comunicação entre Processos

Produtor/Consumidor



Leitores/Escritores

- Um sistema com uma base de dados é acessado simultaneamente por diversas entidades. Estas entidades realizam dois tipos de operações:
  - Leitura
  - Escrita
- Neste sistema é aceitável a existência de diversas entidades lendo a base de dados.
- Porém, se um processo necessita escrever na base, nenhuma outra entidade pode estar realizando acesso à base.

Leitores/Escritores

```
typedef int semaphore;                  /* use sua imaginação */
semaphore mutex = 1;                    /* controla o acesso a 'rc' */
semaphore db = 1;                       /* controla o acesso a base de dados */
int rc = 0;                             /* número de processos lendo ou querendo ler */

void reader(void)
{
     while (TRUE) {                     /* repete para sempre */
          down(&mutex);                 /* obtém acesso exclusivo a 'rc' */
          rc = rc + 1;                  /* um leitor a mais agora */
          if (rc == 1) down(&db);       /* se este for o primeiro leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          read_data_base();             /* acesso aos dados */
          down(&mutex);                 /* obtém acesso exclusivo a 'rc' */
          rc = rc - 1;                  /* um leitor a menos agora */
          if (rc == 0) up(&db);         /* se este for o último leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          use_data_read();              /* região não crítica */
     }
}


void writer(void)
{
     while (TRUE) {                     /* repete para sempre */
          think_up_data();              /* região não crítica */
          down(&db);                    /* obtém acesso exclusivo */
          write_data_base();            /* atualiza os dados */
          up(&db);                      /* libera o acesso exclusivo */
     }
}
```

Uma solução para o problema dos leitores e escritores

Problemas Clássicos de Comunicação entre Processos

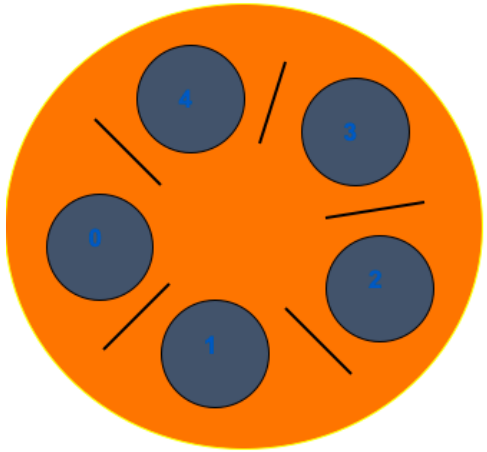
Jantar dos Filósofos

- Cinco filósofos estão sentados ao redor de uma mesa circular para o jantar.
  - Cada filósofo possui um prato para comer espaguete.
  - Como o espaguete é escorregadio, é necessário a utilização de dois garfos.
  - Entre cada par de pratos existe um garfo.

Problemas Clássicos de Comunicação entre Processos

Jantar dos Filósofos

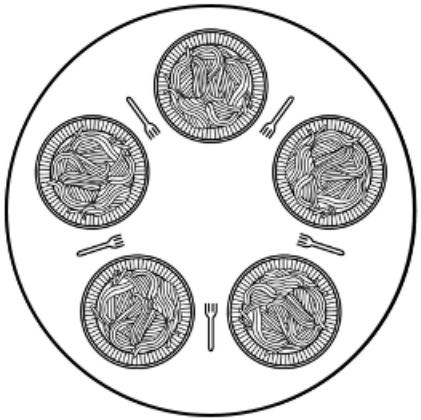
- Cinco filósofos desejam comer espaguete; No entanto, para poder comer, cada filósofo precisa utilizar dois garfo e não apenas um. Portanto, os filósofos precisam compartilhar o uso do garfo de forma sincronizada.
- Os filósofos comem e pensam.



Problemas Clássicos de Comunicação entre Processos

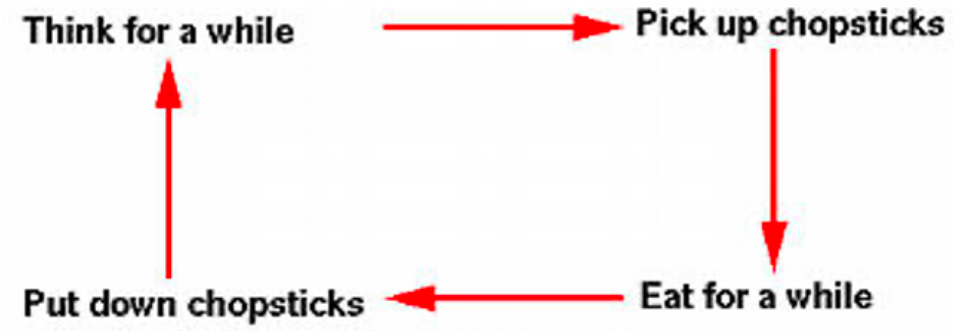
Jantar dos Filósofos

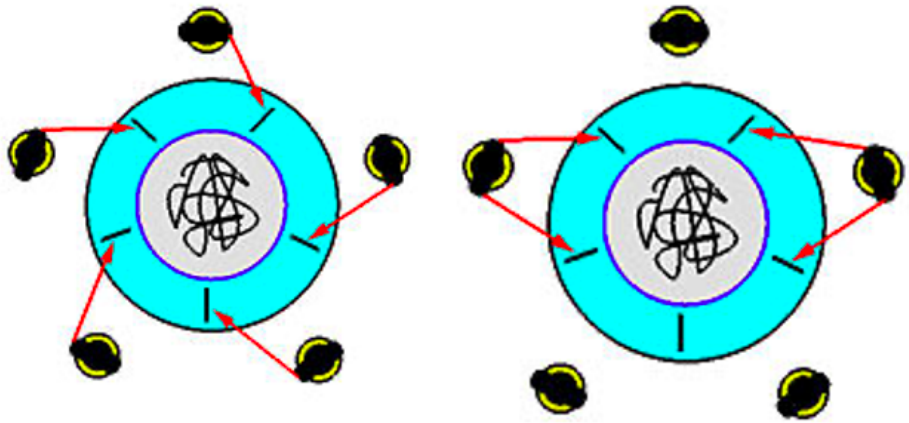
- A vida do filósofo consiste na alternância de períodos de alimentação e reflexão.
  - Quando um filósofo fica com fome, ele tenta pegar os garfos a sua volta (garfos a sua esquerda e direita), em qualquer ordem, um de cada vez.
  - Se o filósofo conseguir pegar os dois garfos ele inicia seu período de alimentação. Após algum tempo ele devolve os garfos a sua posição original e retorna ao período de reflexão.

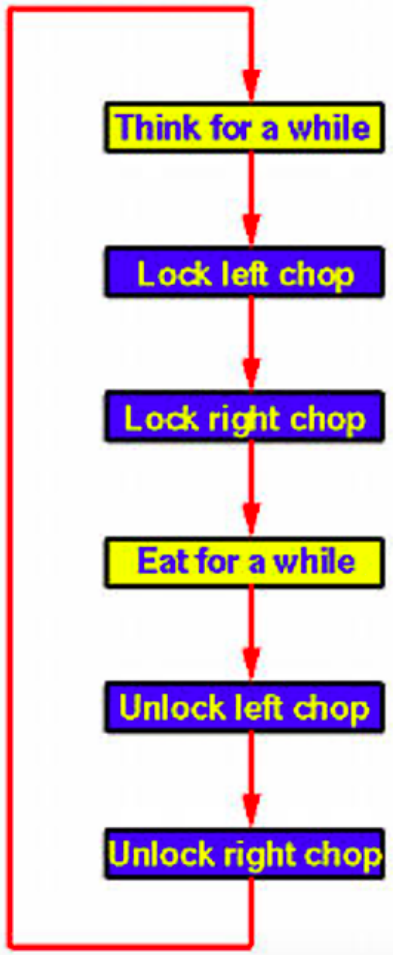


Problemas Clássicos de Comunicação entre Processos

Jantar dos Filósofos







Problemas Clássicos de Comunicação entre Processos

Jantar dos Filósofos

- Problemas que devem ser evitados:
  - *Deadlock* - todos os filósofos pegam **um garfo** ao mesmo tempo;
  - *Starvation* - os filósofos **ficam indefinidamente pegando garfos simultaneamente**.

Jantar dos Filósofos - Solução 1

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

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

Problemas Clássicos de Comunicação entre Processos

Jantar dos Filósofos - Solução 1

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

Jantar dos Filósofos - Solução 1 (modificação 2)

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

Somente um filósofo come!

Jantar dos Filósofos - Solução 2 - Usando Semáforos

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

Jantar dos Filósofos - Solução 2 - Usando Semáforos (Parte1)

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

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

Jantar dos Filósofos - Solução 2 - Usando Semáforos (Parte2)

```
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]);
        }
}
```

Problemas Clássicos de Comunicação entre Processos

Barbeiro Sonolento

- Uma barbearia possui:
  - 1 barbeiro
  - 1 cadeira de barbeiro
  - N cadeiras para espera de clientes
- Se, em um determinando momento, não houverem clientes para serem atendidos, o barbeiro dorme
  - Quando um cliente chega, ele acorda e atende o cliente.
  - Quando um cliente chega e o barbeiro estiver atendendo um cliente, ele aguarda sua vez sentado na cadeira e espera.
  - Quando um cliente chega e não existem cadeiras de espera disponíveis, o cliente vai embora.

Barbeiro Sonolento

Barbeiro Sonolento

```
#define CHAIRS 5                    /* número de cadeiras para os clientes à espera */

typedef int semaphore;              /* use sua imaginação */

semaphore customers = 0;            /* número de clientes à espera de atendimento*/
semaphore barbers = 0;              /* número de barbeiros à espera de clientes */
semaphore mutex = 1;                /* para exclusão mútua */
int waiting = 0;                    /* clientes estão esperando (não estão cortando) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* vai dormir se o número de clientes for 0 */
        down(&mutex);               /* obtém acesso a 'waiting' */
        waiting = waiting – 1;      /* decresce de um o contador de clientes à espera */
        up(&barbers);               /* um barbeiro está agora pronto para cortar cabelo */
        up(&mutex);                 /* libera 'waiting' */
        cut_hair( );                /* corta o cabelo (fora da região crítica) */
    }
}

void customer(void)
{
    down(&mutex);                   /* entra na região crítica */
    if (waiting < CHAIRS) {         /* se não houver cadeiras livres, saia */
        waiting = waiting + 1;      /* incrementa o contador de clientes à espera */
        up(&customers);             /* acorda o barbeiro se necessário */
        up(&mutex);                 /* libera o acesso a 'waiting' */
        down(&barbers);             /* vai dormir se o número de barbeiros livres for 0 */
        get_haircut( );             /* sentado e sendo servido */
    } else {
        up(&mutex);                 /* a barbearia está cheia; não espere */
    }
}
```

Solução para o problema do barbeiro sonolento

Problemas Clássicos de Comunicação entre Processos

Links interessantes

- `http://users.erols.com/ziring/diningAppletDemo.html`
- `http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/deadlock.html`
- `http://www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html`

Problemas Clássicos de Comunicação entre Processos

Sugestão de Exercícios

- Entender a solução para o problema dos Filósofos utilizando semáforos.
  - Identificando a(s) **região(ões) crítica(s)**
  - Descrevendo **exatamente** como a solução funciona.
- Entender a solução para o problema dos Produtores/Consumidores utilizando Monitor
  - Identificando a(s) **região(ões) crítica(s)**
  - Descrevendo **exatamente** como a solução funciona.

Comunicação de Processos

Soluções para Exclusão Mútua

- Espera Ocupada.
- Primitivas *Sleep/Wakeup*.
- Semáforos.
- Monitores.
- **Passagem de Mensagem.**

Comunicação de Processos

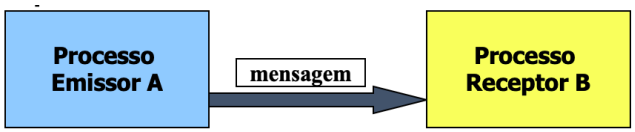
Mecanismos mais elaborados de Comunicação e Sincronização de Processos

- A troca de mensagens é um mecanismo de comunicação e sincronização que exige do S.O., tanto a sincronização quanto a comunicação entre os processos.
- Os mecanismos já considerados exigem do S.O. somente a sincronização, deixando para o programador a comunicação de mensagens por meio da memória compartilhada.
- Os mecanismos estudados até agora asseguram a exclusão mútua, mas não garantem um controle sobre as operações desempenhadas sobre o recurso.

Comunicação de Processos

Mecanismos mais elaborados de Comunicação e Sincronização de Processos

- O uso de troca de mensagens para a manipulação de um recurso compartilhado assegura a exclusão mútua, e impõe restrições nas operações a serem desempenhadas sobre ele.
- Os mecanismos já considerados exigem do S.O. somente a sincronização, deixando para o programador a comunicação de mensagens através da memória compartilhada.
- Esquema de troca de mensagens: os processos enviam e recebem mensagens, em vez de ler e escrever em variáveis compartilhadas.



Comunicação de Processos

Mecanismos mais elaborados de Comunicação e Sincronização de Processos

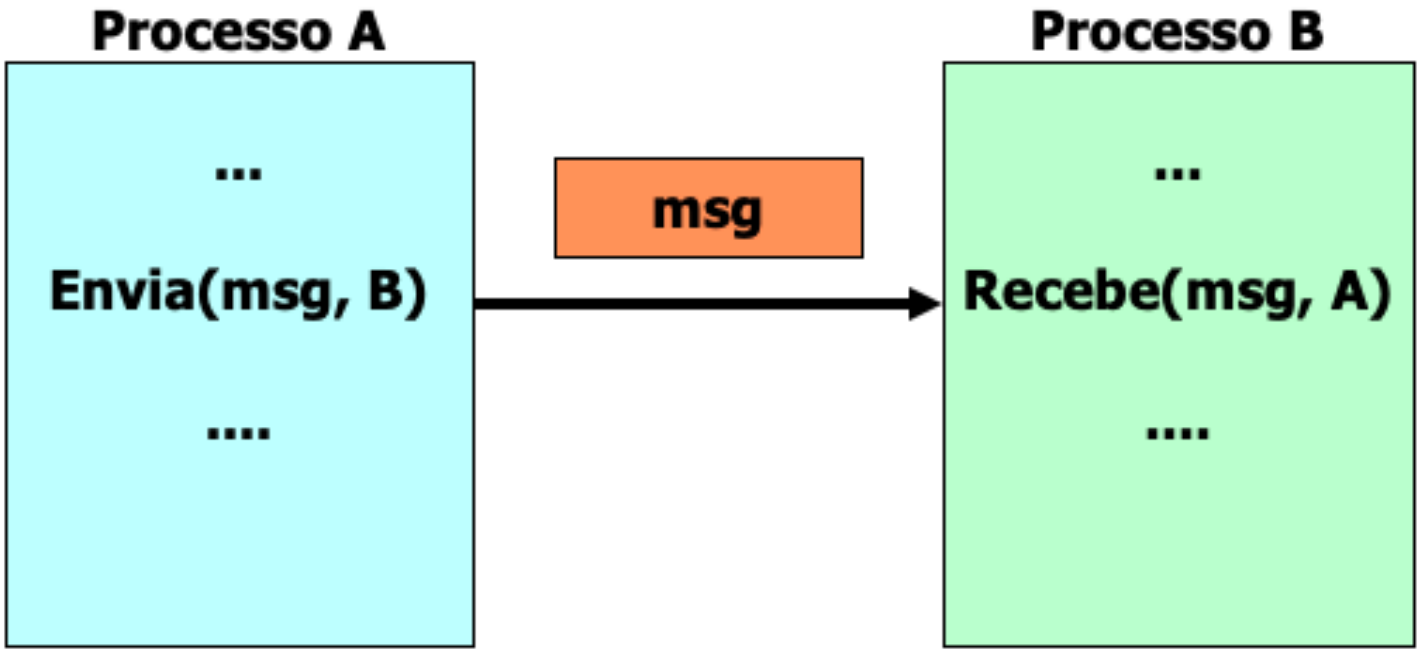
- A sincronização entre processos: é garantida pela restrição de que uma mensagem só poderá ser recebida depois de ter sido enviada.
- A transferência de dados de um processo para outro, após ter sido realizada a sincronização, estabelece a comunicação.

Comunicação de Processos

Primitivas de Troca de Mensagens

- De forma genérica, uma mensagem será enviada quando um processo executar o seguinte comando:
  - Envia (mensagem, $processo_receptor$) ou
  - Send (message, receiver)
- Uma mensagem será recebida quando um processo executar o seguinte comando:
  - Recebe (mensagem, $processo_emissor$) ou
  - Receive (message, sender)

Primitivas de Troca de Mensagens



Comunicação de Processos

Primitivas de Troca de Mensagens

- As primitivas podem ser de dois tipos:
  - **Bloqueantes**: quando o processo que a executar ficar *bloqueado* até que a *operação seja bem sucedida* (ou seja, quando ocorrer a entrega efetiva da mensagem ao processo destino, no caso da emissão, ou o recebimento da mensagem pelo processo destino, no caso de recepção).
  - **Não Bloqueantes**: quando o processo que executar a primitiva, *continuar sua execução normal*, independentemente da entrega ou do recebimento efetivo da mensagem pelo processo destino.

Primitivas de Troca de Mensagens Exemplo

```
Program emissor_receptor;
Type msg=...;
Var mensagem : msg;
Begin  /* inicio do programa principal */
  Cobegin /* inicio dos processos concorrentes */
    Begin /* processo emissor – E */
     repeat
       ...;
       produz uma mensagem;
       Send(mensagem, R);
     until  false
    End;

    Begin /* processo receptor – R */
     repeat
       Receive(mensagem, E);
       Consome a  mensagem;
        ...;
       until  false
    End
  Coend
End.
```

Comunicação de Processos

Primitivas de Troca de Mensagens

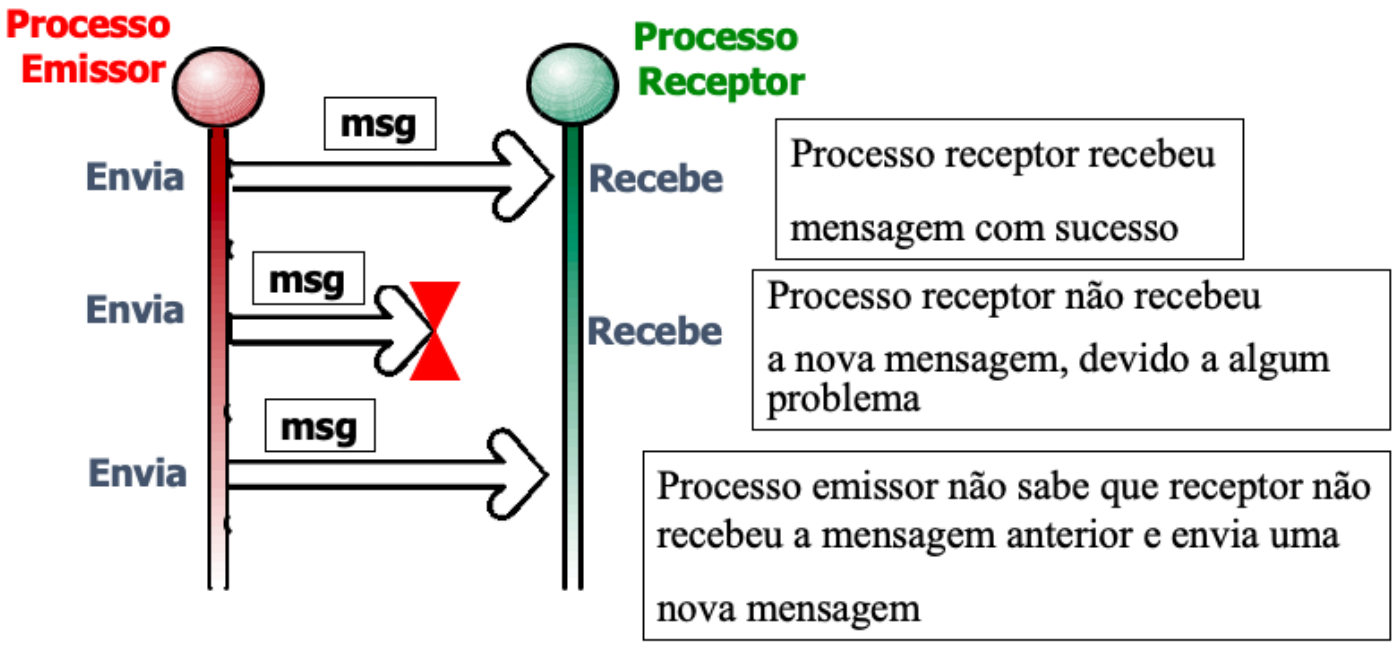
- Os sistemas de troca de mensagens possuem alguns problemas e estudos de projetos interessantes, principalmente quando os processos comunicantes estão em máquinas diferentes, conectadas por uma rede de comunicação. Os principais são:
  - Perda de mensagem.
  - Perda de reconhecimento.
  - Nomeação de processos.
  - Autenticação.
  - Estudos de projeto para quando emissor e receptor estiverem na mesma máquina.

Comunicação de Processos

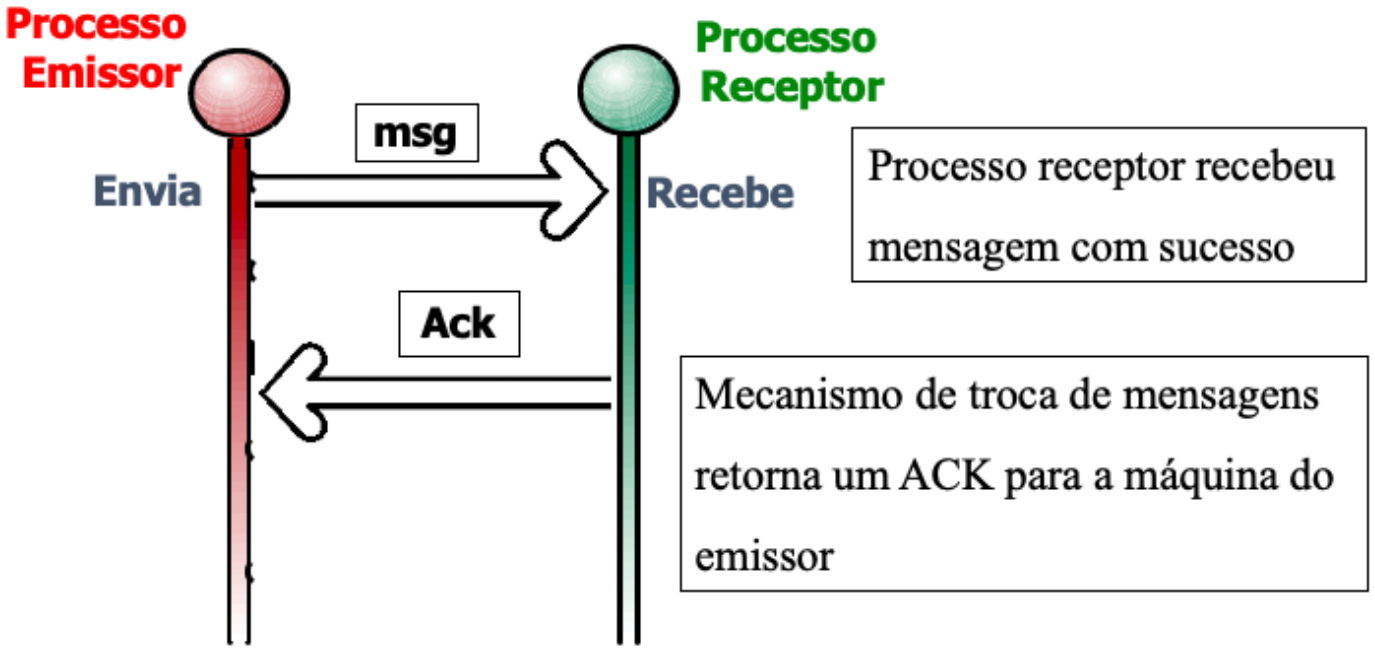
Perda de Mensagens

- Uma solução é que o receptor, ao receber uma nova mensagem, envie uma mensagem especial de reconhecimento (ACK). Se o emissor não receber um ACK a tempo, deve retransmitir a mensagem.

Perda de Mensagens - O Problema



Perda de Mensagens - Uma solução

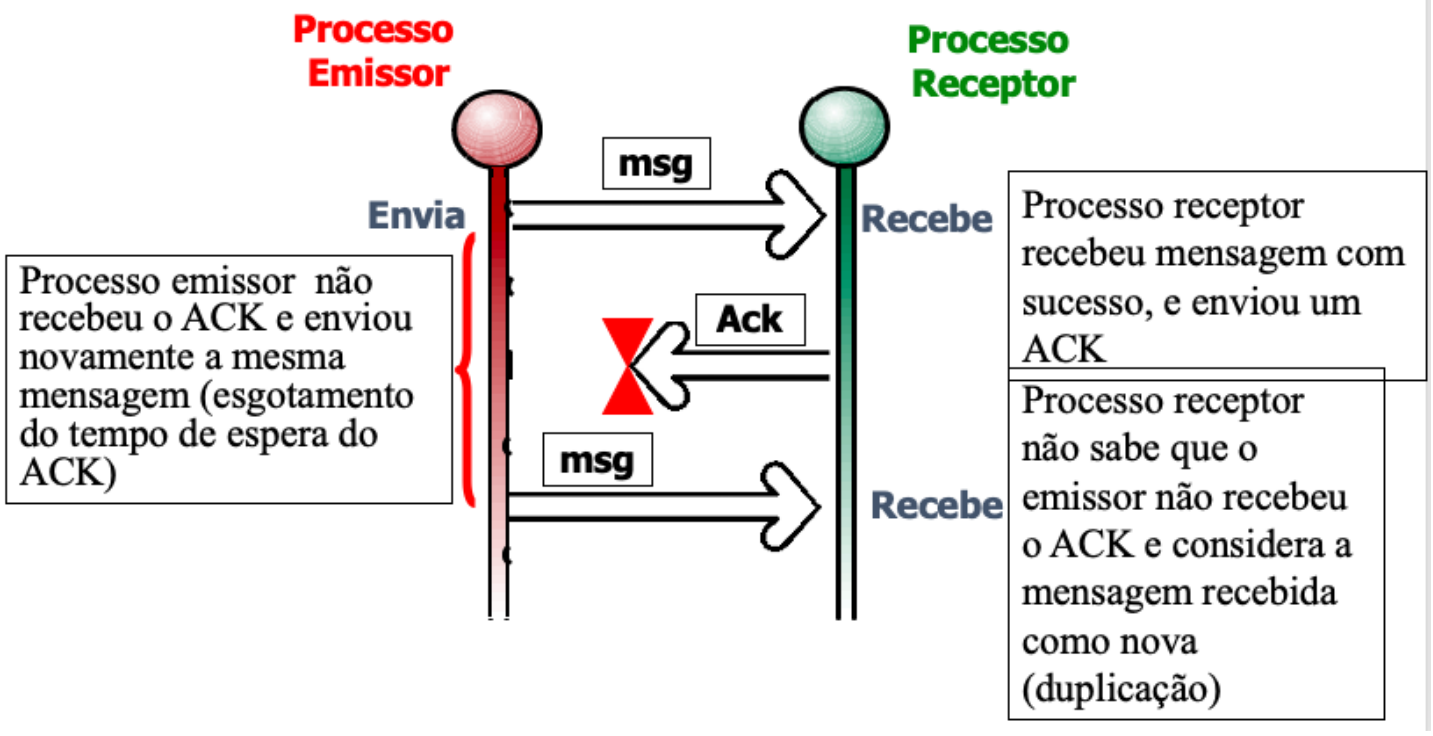


Comunicação de Processos

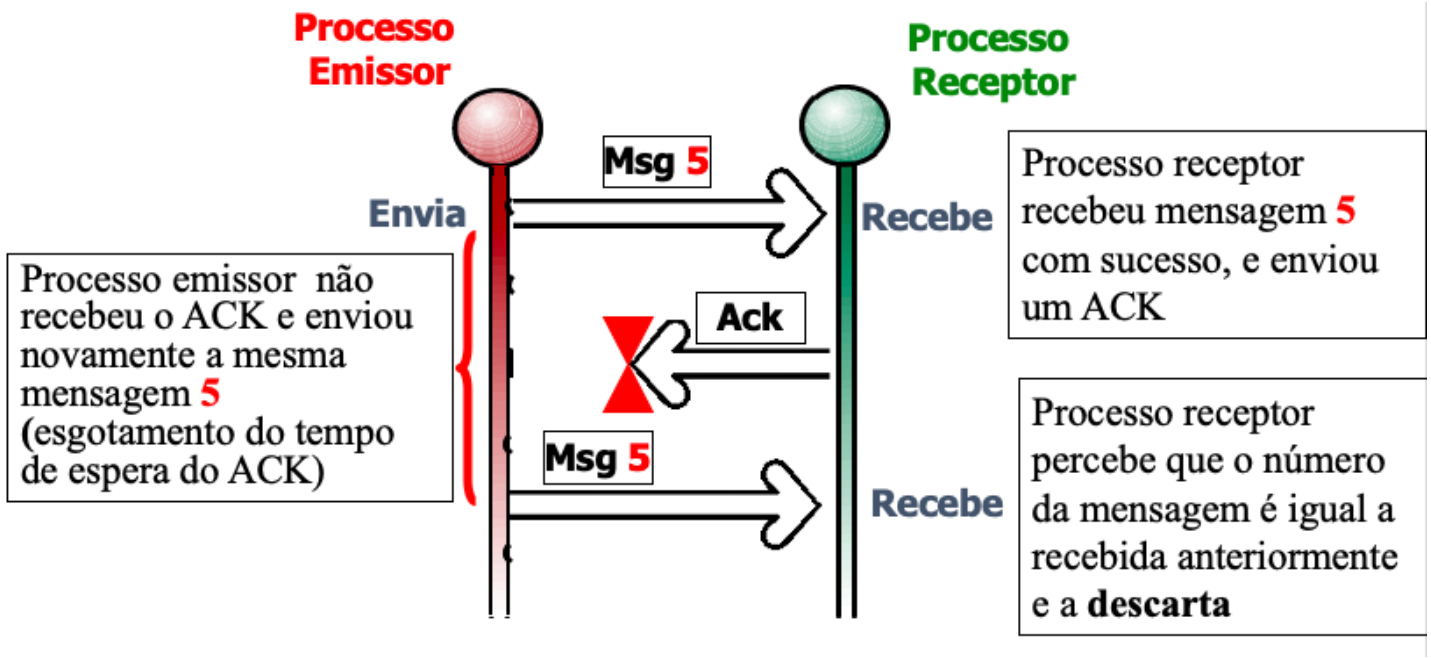
Perda de Reconhecimento

- causa o problema de se receber mensagens idênticas.
- Uma das soluções é a numeração de mensagens.

Perda de Reconhecimento - O Problema



Perda de Reconhecimento - Uma solução é Numerar as mensagens



Comunicação de Processos

Nomeação de Processos

- os processos devem ser nomeados de maneira única, para que o nome do processo especificado no *Send* ou *Receive* não seja ambíguo.
- Ex. processo@máquina (normalmente existe uma autoridade central que nomeia as máquinas).
- Quando o número de máquinas é muito grande: processo@máquina.domínio

Comunicação de Processos

Autenticação

- permitir a comunicação e acessos apenas dos usuários autorizados. Uma solução é criptografar as mensagens com uma chave conhecida apenas por usuários autorizados.

Comunicação de Processos

Estudo de projeto para quando emissor e receptor estão na mesma máquina

- Para aumento do desempenho, pensa-se em registradores especializados para a troca de mensagens.

Comunicação de Processos

Combinação de Primitivas

- Existem quatro maneiras de se combinar as primitivas de troca de mensagens:

Envia Bloqueante – Recebe Bloqueante → **síncrono**

Envia Bloqueante – Recebe Não Bloqueante

Envia Não Bloqueante – Recebe Bloqueante

} **Semi síncrono**

Envia Não Bloqueante – Recebe Não Bloqueante → **Assíncrono**

Comunicação de Processos

- Podem ser implementadas como procedimentos:
  - *send (destination, &message)*
  - *receive (source, &message)*
- O procedimento *send* envia para um determinado destino uma mensagem, enquanto que o procedimento *receive* recebe essa mensagem em uma determinada fonte; Se nenhuma mensagem está disponível, o procedimento *receive* é bloqueado até que uma mensagem chegue.

Comunicação de Processos

Problemas desta solução

- Mensagens são enviadas para/por máquinas conectadas em rede; assim mensagens podem se perder ao longo da transmissão;
- Mensagem especial chamada **acknowledgement** - o procedimento *receive* envia um **acknowledgement** para o procedimento *send*. Se esse **acknowledgement** não chega no procedimento *send*, esse procedimento retransmite a mensagem já enviada.

Comunicação de Processos

Problemas desta solução

- A mensagem é recebida corretamente, mas o **acknowledgement** se perde.
- Então o *receive* deve ter uma maneira de saber se uma mensagem recebida é uma retransmissão - cada mensagem enviada pelo *send* possui uma identificação – sequência de números; Assim, ao receber uma nova mensagem, o *receive* verifica essa identificação, se ela for semelhante a de alguma mensagem já recebida, o *receive* descarta a mensagem!

Comunicação de Processos

Problemas desta solução

- Desempenho: copiar mensagens de um processo para o outro é mais lento do que operações com semáforos e monitores.
- Autenticação - Segurança.

Passagem de Mensagem

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Comunicação de Processos

Mecanismos de Comunicação Síncronos

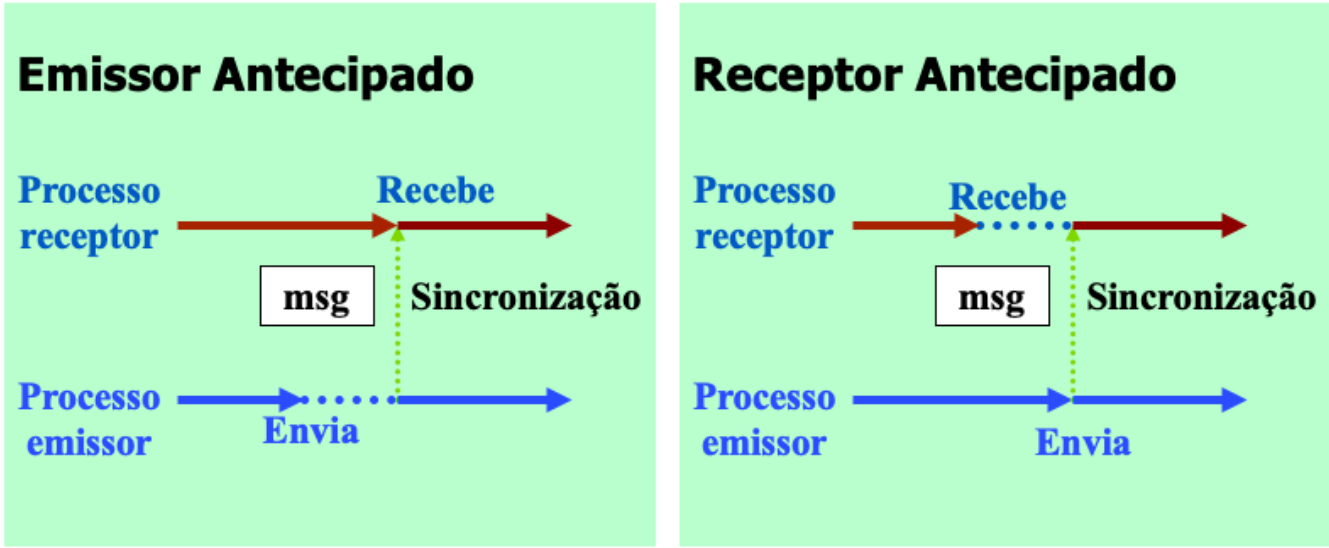
- Existem três mecanismos de comunicação síncronos mais importantes:
  - Rendez-vous
  - Rendez-vous Estendido
  - Chamada Remota de Procedimento

Comunicação de Processos

Mecanismos de Comunicação Síncronos - Rendez-vous

- É obtido por meio de primitivas Envia e Recebe bloqueantes colocadas em processos distintos; a execução destas primitivas em tempos diferentes, fará com que o processo que executar a primitiva antes do outro fique bloqueado até que ocorra a sincronização entre os dois processos, e a consecutiva transferência da mensagem; em seguida, ambos os processos continuarão seu andamento em paralelo.
- Ex.: linguagem CSP.

Mecanismos de Comunicação Síncronos - Rendez-vous

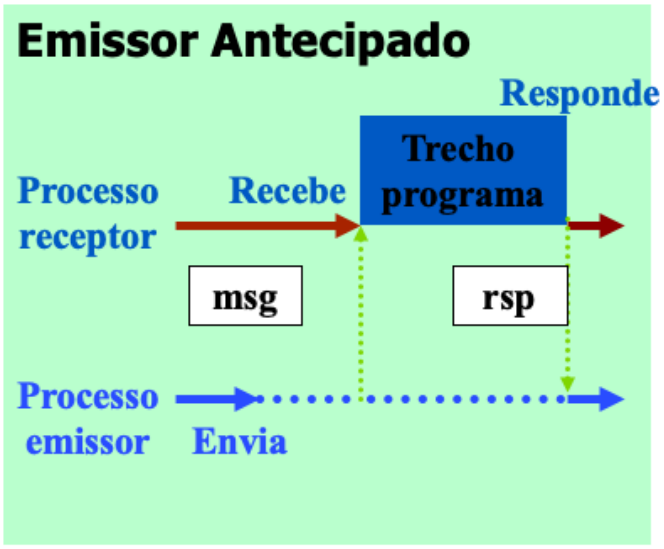


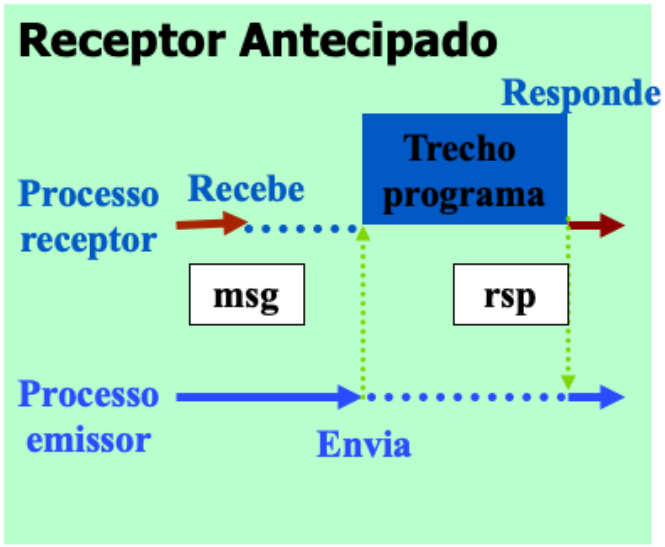
Comunicação de Processos

Mecanismos de Comunicação Síncronos - Rendez-vous Estendido

- Caracteriza-se por apresentar uma estrutura de comunicação onde um processo consegue comandar a execução de um trecho de programa previamente estabelecido, pertencente a outro processo, envolvendo sincronização e, eventualmente, troca de mensagem.
- Ex: linguagem ADA.

Mecanismos de Comunicação Síncronos - Rendez-vous Estendido



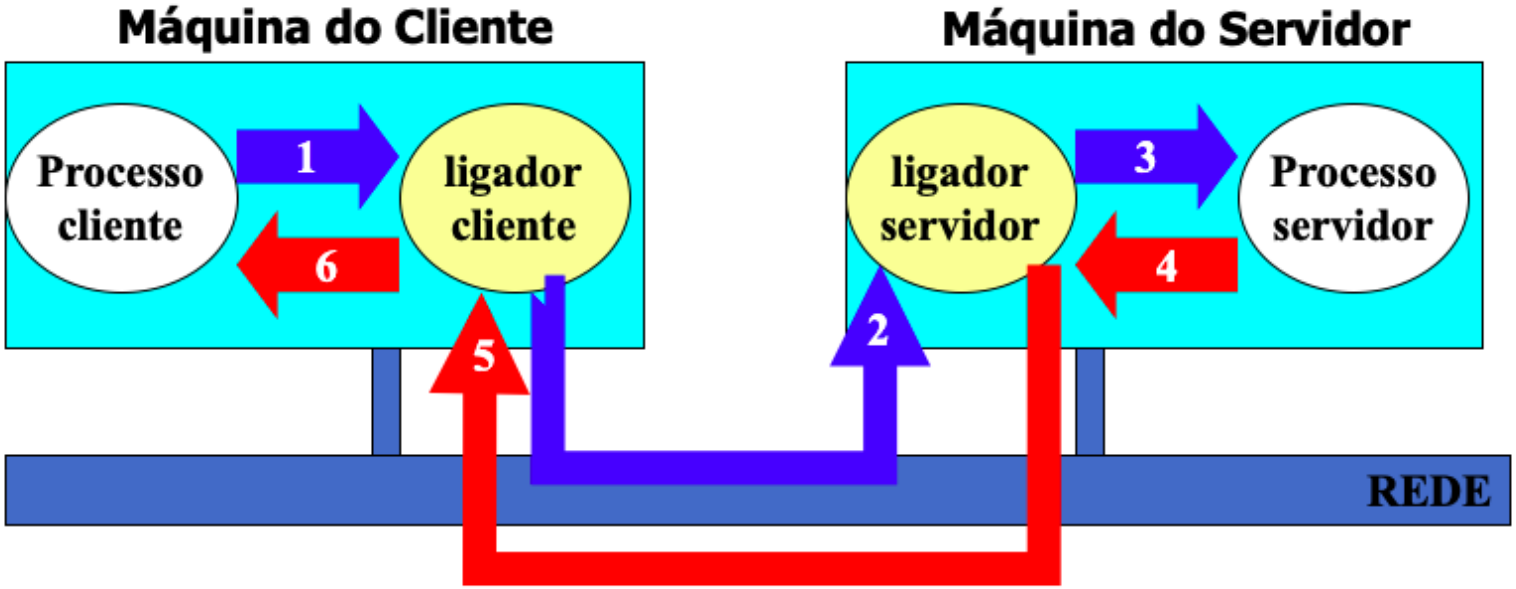


Comunicação de Processos

Mecanismos de Comunicação Síncronos - Chamada Remota de Procedimento (*Remote Procedure Call* - RPC)

- Apresenta uma estrutura de comunicação na qual um processo pode comandar a execução de um procedimento situado em outro processador. O processo chamador deverá ficar bloqueado até que o procedimento chamado termine. Tanto a chamada quanto o retorno podem envolver troca de mensagem, conduzindo parâmetros.
- Ex: linguagem DP.

Mecanismos de Comunicação Síncronos - Chamada Remota de Procedimento (*Remote Procedure Call* - RPC)



(1) e (3) são chamadas de procedimento comuns
(2) e (5) são mensagens
(4) e (6) são retornos de procedimento comuns

Comunicação de Processos

RPC - Vantagens

- Cliente e servidor não precisam saber que as mensagens são utilizadas.
- Eles as veem como chamadas de procedimento locais.

RPC - Problemas

- Dificuldade da passagem de parâmetros por referência: por exemplo, se as máquina servidora e cliente possuem diferentes representações de informação (necessidade de conversão e desconversão).
- Diferenças de arquitetura: por exemplo, as máquinas podem diferir no armazenamento de palavras.
- Falhas semânticas: por exemplo, o servidor pára de funcionar quando executava uma RPC. O que dizer ao cliente? Se disser que houve falha e o servidor terminou a chamada logo antes de falhar, o cliente pode pensar que falhou antes de executar a chamada. Ele pode tentar novamente, o que pode não ser desejável. Principais abordagens: "no mínimo uma vez", "exatamente uma vez" e "talvez uma vez".

Comunicação de Processos

Outros Mecanismos

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

Comunicação de Processos

Outros Mecanismos

- Pipe
  - Permite a criação de filas de processos
  - ps -ef — grep alunos
  - Saída de um processo é a entrada de outro
  - Existe enquanto o processo existir
- Named Pipe
  - Extensão de pipe
  - Continua existindo mesmo depois que o processo terminar
  - Criado com chamadas de sistemas
- Socket
  - Par endereço IP e porta utilizado para comunicação entre processos em máquinas diferentes
  - Host X (192.168.1.1:1065) Server Y (192.168.1.2:80)

Continuemos com **DEADLOCK** ....