

Capítulo 3: Processos



Capítulo 3: Processos

- ❑ Conceito de processo
- ❑ Escalonamento de processo
- ❑ Operações sobre processos
- ❑ Processos em cooperação
- ❑ Comunicação entre processos
- ❑ Comunicação em sistemas cliente-servidor

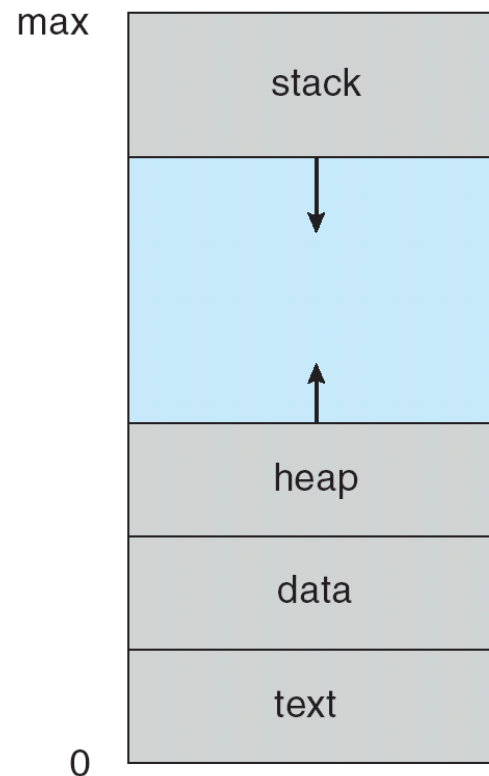


Conceito de processo

- ❑ Um sistema operacional executa diversos programas:
 - Sistemas batch – jobs
 - Sistemas compartilhados no tempo – programas ou tarefas do usuário
- ❑ Livro texto usa os termos *job* e *processo* para indicar quase a mesma coisa (um job contém 1 ou mais processos)
- ❑ **Processo** – um programa em execução; a execução do processo deve progredir de modo seqüencial
- ❑ Um processo inclui:
 - contador de programa
 - pilha
 - seção de dados



Processo na memória

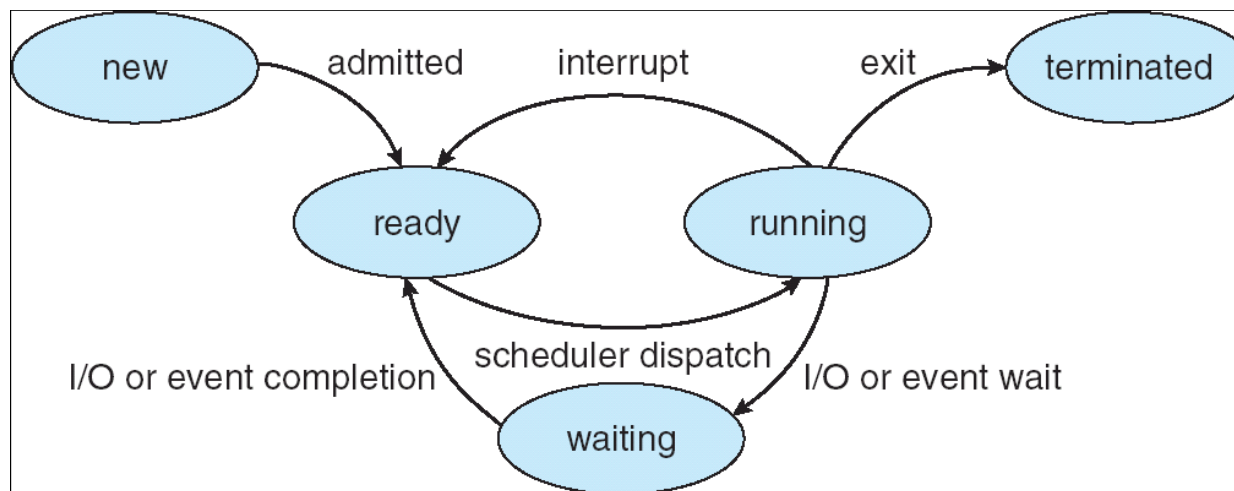


Estado do processo

- Enquanto um processo é executado, ele muda de *estado*:
 - **novo**: O processo está sendo criado
 - **pronto**: O processo foi criado com sucesso e está esperando para ser atribuído a um processador
 - **executando**: Instruções estão sendo executadas
 - **esperando**: O processo está esperando que ocorra algum evento
 - **terminado**: O processo terminou a execução



Diagrama de estado do processo



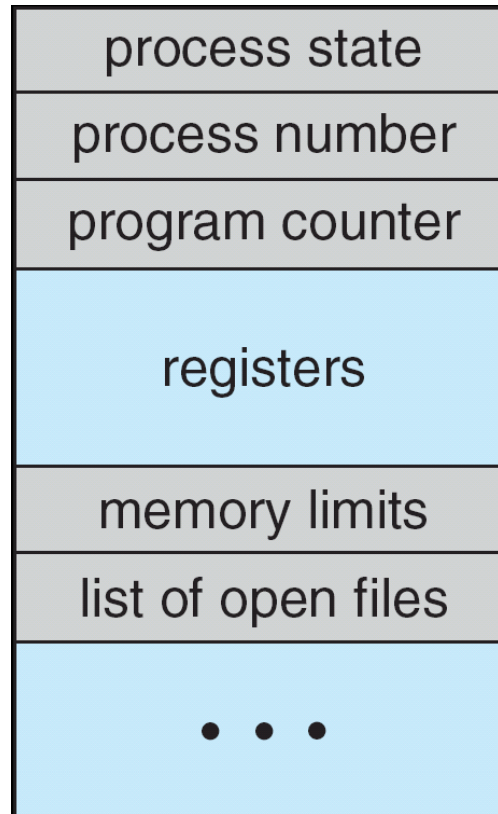
Process Control Block (PCB)

Informações associadas a cada processo

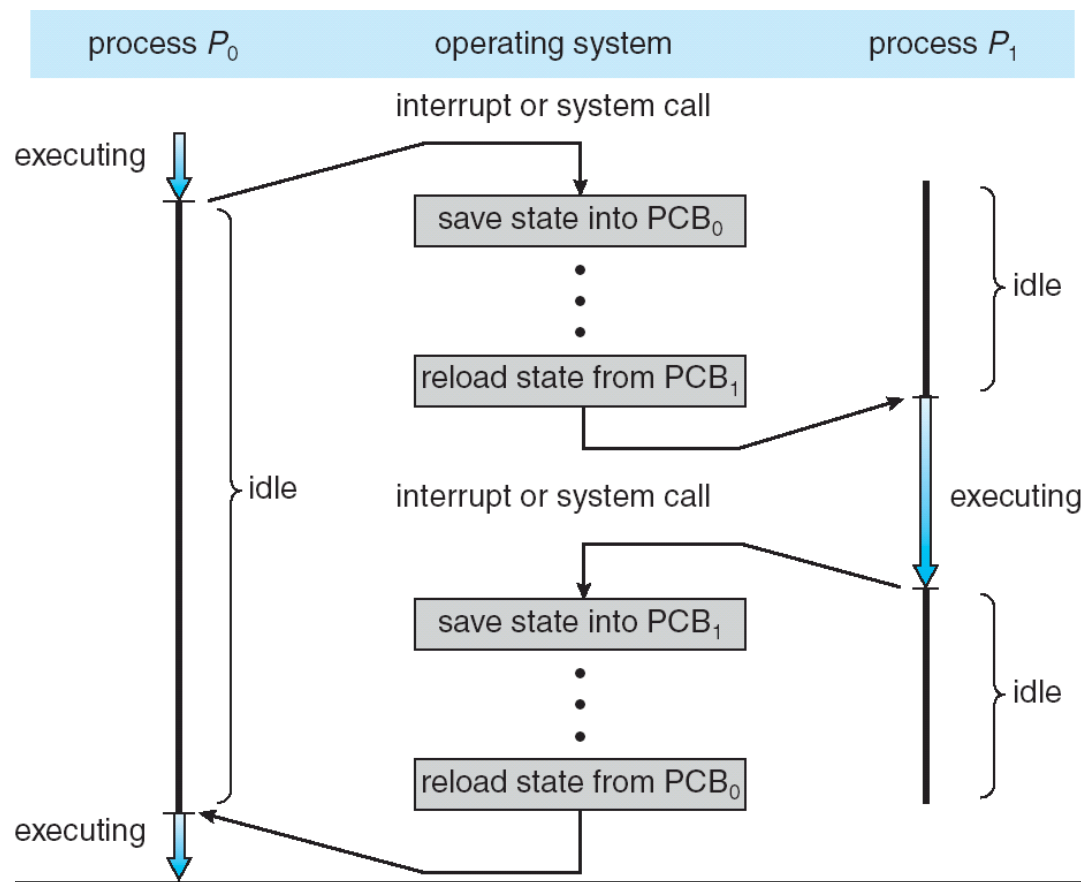
- ❑ Estado do processo
- ❑ Contador de programa
- ❑ Registradores da CPU
- ❑ Informação de escalonamento da CPU
- ❑ Informação de gerenciamento de memória
- ❑ Informação de contabilidade
- ❑ Informação de status de E/S



Process Control Block (PCB)



Troca de processos pela CPU

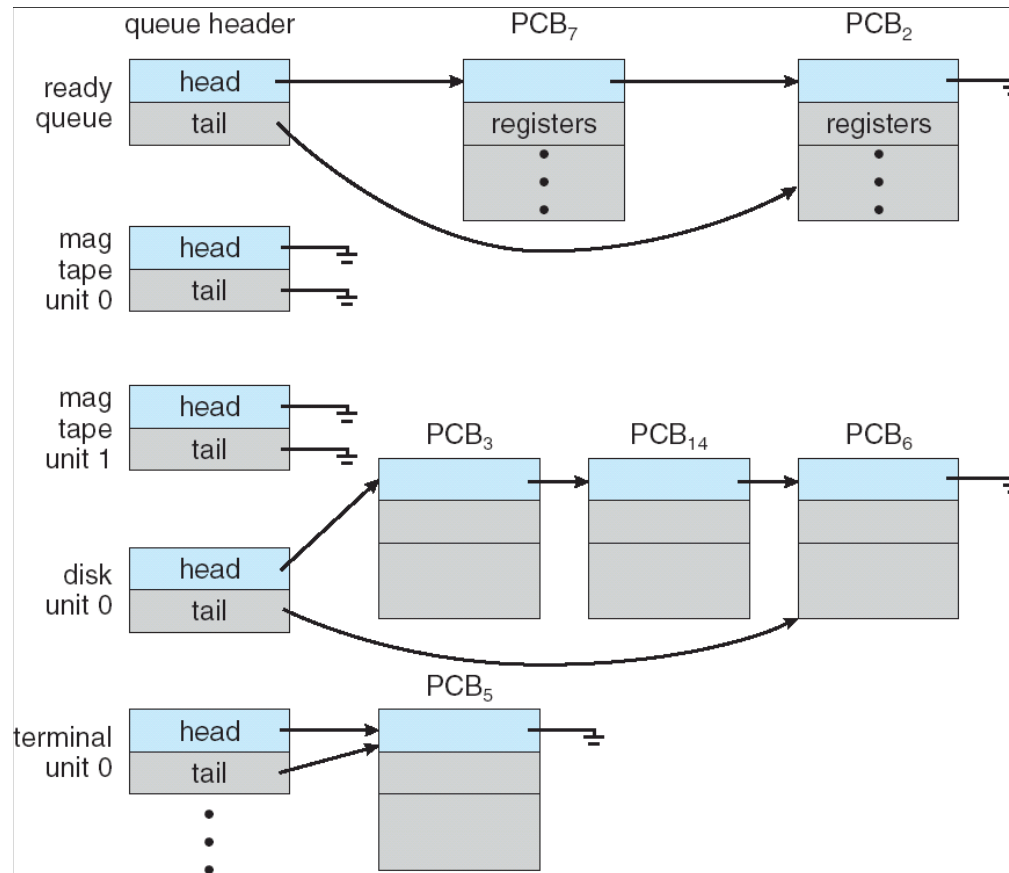


Filas de escalonamento de processo

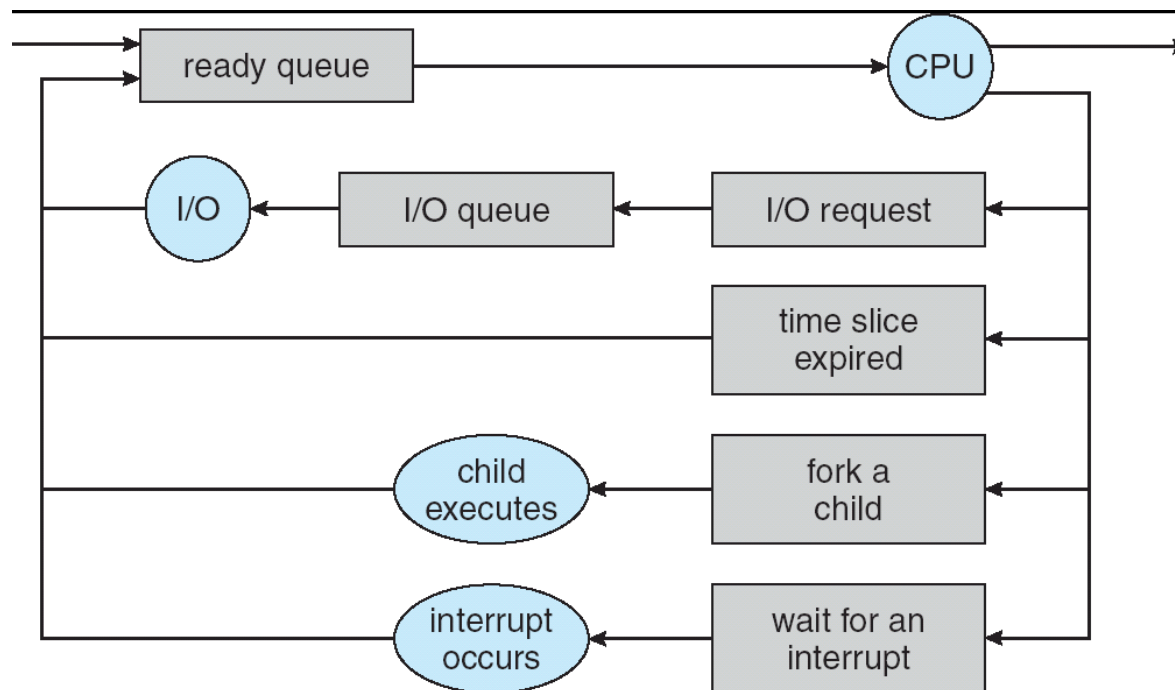
- ❑ **Fila de tarefas** – conjunto de todos os processos no sistema
- ❑ **Fila de pronto** – conjunto de todos os processos residindo na memória principal, prontos e esperando para execução
- ❑ **Filas de dispositivo** – conjunto de processos esperando por um dispositivo de E/S
- ❑ Processos migram entre as diversas filas



Fila de pronto e diversas filas de dispositivo de E/S



Representação do escalonamento de processos



Escalonadores

- ❑ **Escalonador a longo prazo** (ou escalonador de job) – seleciona quais processos devem ser trazidos para a fila de pronto. Geralmente, utilizado em sistemas batch.
- ❑ **Escalonador a curto prazo** (ou escalonador de CPU) – seleciona qual processo deve ser executado em seguida e aloca CPU

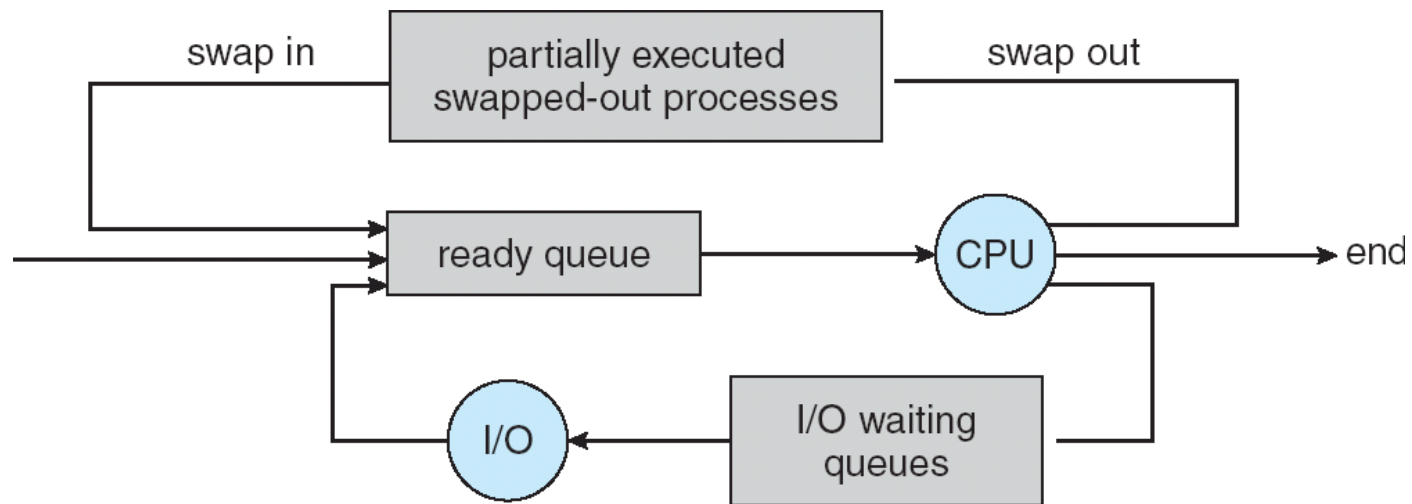


Escalonadores (cont.)

- ❑ O escalonador a curto prazo é invocado muito freqüentemente (milissegundos) ! (deve ser rápido)
- ❑ O escalonador a longo prazo é invocado com pouca freqüência (segundos, minutos) ! (pode ser lento)
- ❑ O escalonador a longo prazo controla o *grau de multiprogramação*
- ❑ Os processos podem ser descritos como:
 - **Processos voltados para E/S** – gasta mais tempo realizando E/S do que cálculos, com *bursts* de CPU muito curtos
 - **Processos voltados para CPU** – gasta mais tempo realizando cálculos; poucos *bursts* de CPU muito longos



Escalonador de médio prazo (swapping)



Troca de contexto

- ❑ Quando a CPU passa para outro processo, o sistema deve salvar o estado do processo antigo e carregar o estado salvo para o novo processo
- ❑ *Overhead* é o tempo de troca de contexto; o sistema não realiza trabalho útil enquanto faz a troca
- ❑ Tempo dependente do suporte do hardware

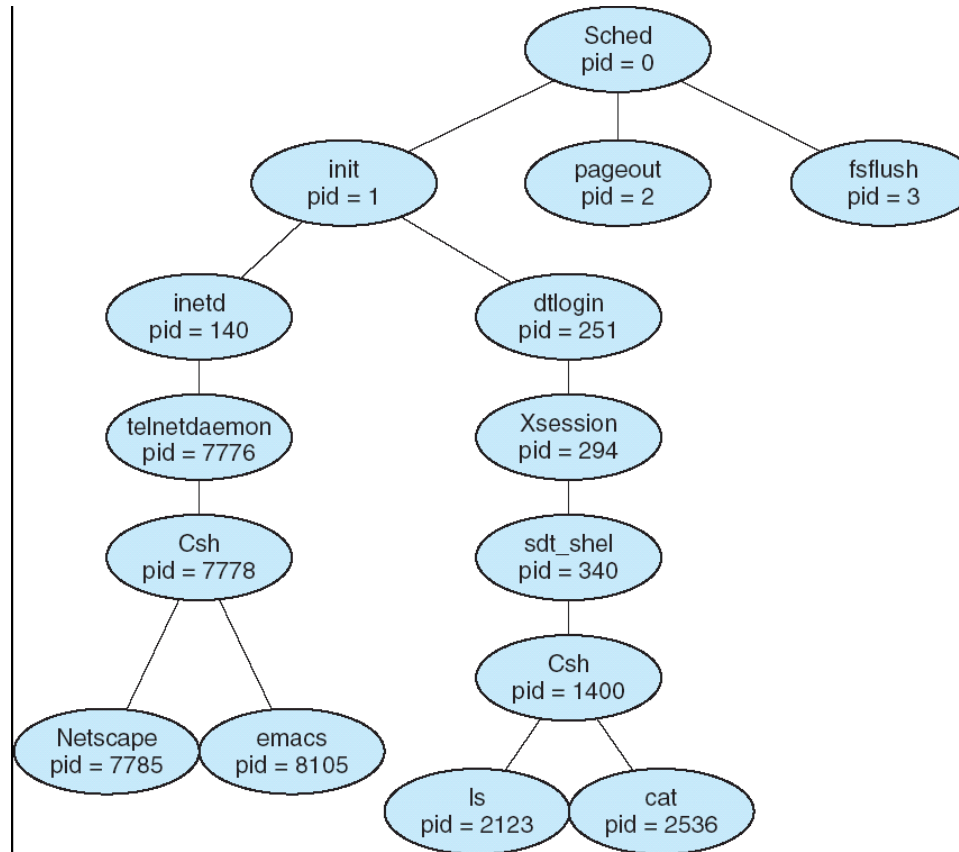


Criação de processo

- ❑ Processo pai cria processos filho que, por sua vez, criam outros processos, formando uma árvore de processos
- ❑ Compartilhamento de recursos (3 tipos)
 - Pai e filhos compartilham todos os recursos
 - Filhos compartilham subconjunto dos recursos do pai
 - Pai e filho não compartilham recursos
- ❑ Execução (2 tipos)
 - Pai e filhos executam simultaneamente
 - Pai espera até que filhos terminem



Árvores de processos

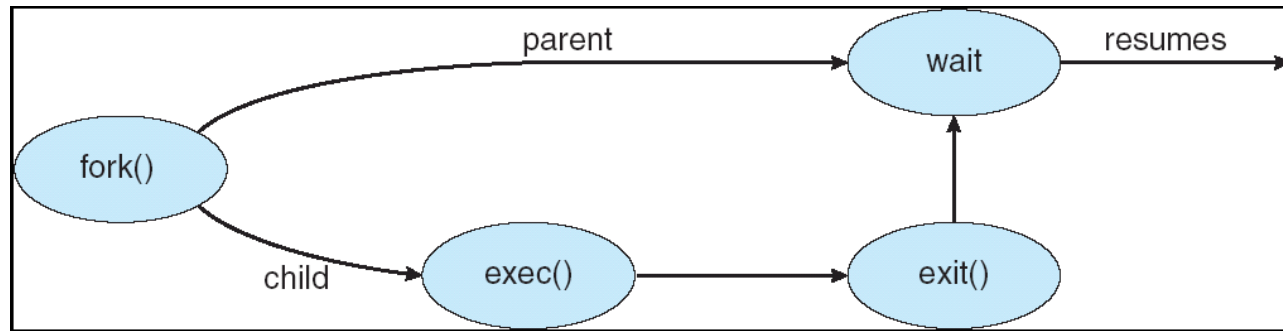


Criação de processo (cont.)

- Espaço de endereços (2 tipos)
 - Filho duplicata do pai
 - Filho tem um programa carregado
- Exemplos do UNIX
 - Chamada do sistema **fork** cria novo processo
 - Chamada do sistema **exec** usada após um **fork** para substituir o espaço de memória do processo por um novo programa



Criação de processo



Criação de processo no POSIX

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```



Criação de processo no Win32

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
        "C:\\WINDOWS\\system32\\mspaint.exe", // command line
        NULL, // don't inherit process handle
        NULL, // don't inherit thread handle
        FALSE, // disable handle inheritance
        0, // no creation flags
        NULL, // use parent's environment block
        NULL, // use parent's existing directory
        &si,
        &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```



Criação de processo em Java

```
import java.io.*;

public class OSProcess
{
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("Usage: java OSProcess <command>");
            System.exit(0);
        }

        // args[0] is the command
        ProcessBuilder pb = new ProcessBuilder(args[0]);
        Process proc = pb.start();

        // obtain the input stream
        InputStream is = proc.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);

        // read what is returned by the command
        String line;
        while ( (line = br.readLine()) != null)
            System.out.println(line);

        br.close();
    }
}
```



Término de processo

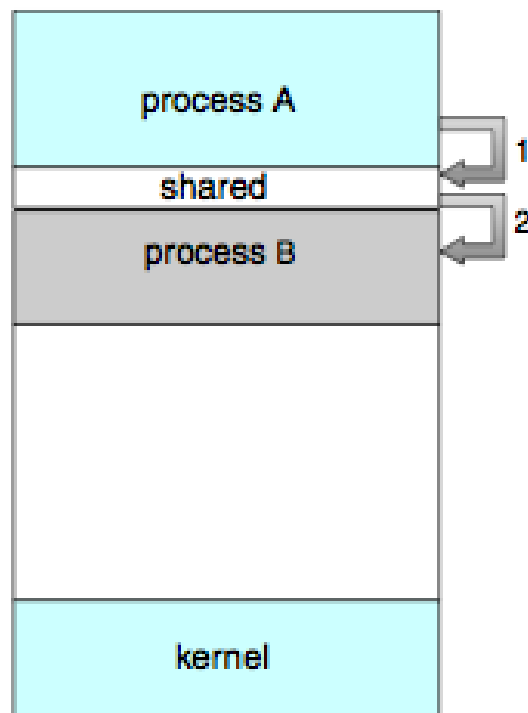
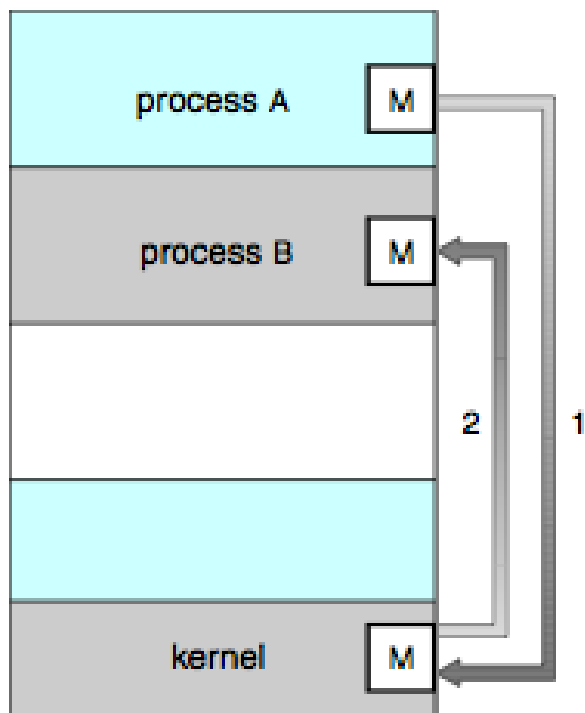
- Processo executa última instrução e pede ao sistema operacional para excluí-la (**exit**)
 - Dados de saída do filho vão para o pai (via **wait**)
 - Recursos do processo são liberados pelo sistema operacional
- Pai pode terminar a execução dos processos dos filhos (**abort**)
 - Filho excedeu recursos alocados
 - Tarefa atribuída ao filho não é mais exigida
 - Se o pai estiver saindo
 - Alguns sistemas operacionais não permitem que o filho continue se o pai terminar (*término em cascata*)



Comunicação entre processos

Passagem de mensagem

Memória compartilhada

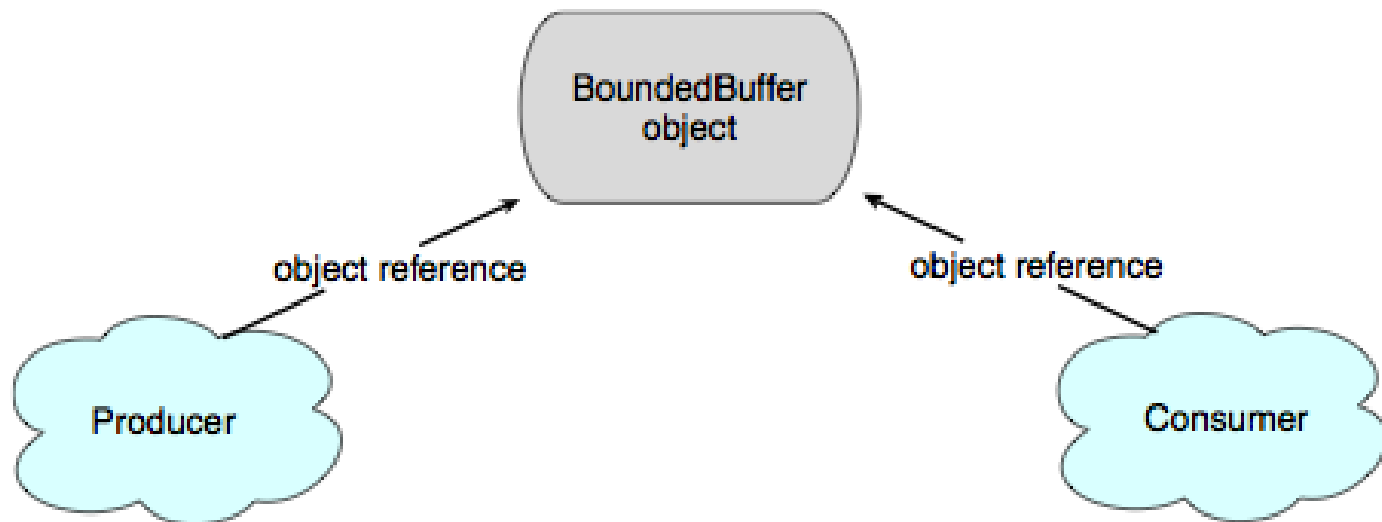


Problema do produtor-consumidor

- Paradigma para processos em cooperação, processo *produtor* produz informações que são consumidas por um processo *consumidor*
 - *Buffer ilimitado* não impõe limite prático sobre o tamanho do buffer
 - *Buffer limitado* assume que existe um tamanho de buffer fixo



Simulando memória compartilhada em Java



Buffer vinculado – solução de memória compartilhada

```
public interface Buffer
{
    // producers call this method
    public abstract void insert(Object item);

    // consumers call this method
    public abstract Object remove();
}
```



Buffer vinculado – solução de memória compartilhada

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private int count; // number of items in the buffer
    private int in; // points to the next free position
    private int out; // points to the next full position
    private Object[] buffer;

    public BoundedBuffer() {
        // buffer is initially empty
        count = 0;
        in = 0;
        out = 0;

        buffer = new Object[BUFFER_SIZE];
    }

    // producers calls this method
    public void insert(Object item) {
        // Figure 3.16
    }

    // consumers calls this method
    public Object remove() {
        // Figure 3.17
    }
}
```



Buffer vinculado –Figura 3.16 – método insert()

```
public void insert(Object item) {  
    while (count == BUFFER_SIZE)  
        ; // do nothing -- no free buffers  
  
    // add an item to the buffer  
    ++count;  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
}
```



Buffer vinculado –Figura 3.17 – método remove()

```
public Object remove() {  
    Object item;  
  
    while (count == 0)  
        ; // do nothing -- nothing to consume  
  
    // remove an item from the buffer  
    --count;  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
  
    return item;  
}
```



Passagem de mensagens

- Sistema de mensagem – processos se comunicam entre si sem lançar mão de variáveis compartilhadas
- Facilidade de passagem de mensagem oferece duas operações:
 - **send**(mensagem) – tamanho da mensagem fixo ou variável
 - **receive**(mensagem)
- Se P e Q quiserem se comunicar, eles precisam:
 - estabelecer um *link de comunicação* entre eles
 - trocar mensagens por meio de send/receive



Questões de implementação

- ❑ Como os links são estabelecidos?
- ❑ Um link pode estar associado a mais de dois processos?
- ❑ Quantos links pode haver entre cada par de processos em comunicação?
- ❑ Qual é a capacidade de um link?
- ❑ O tamanho de uma mensagem que o link pode acomodar é fixo ou variável?
- ❑ Um link é unidirecional ou bidirecional?



Comunicação direta

- Processos devem nomear um ao outro explicitamente:
 - **send** (P , *mensagem*) – envia uma mensagem ao processo P
 - **receive**(Q , *mensagem*) – recebe uma mensagem do processo Q
- Propriedades do link de comunicação
 - Links são estabelecidos automaticamente
 - Um link é associado a exatamente um par de processos em comunicação
 - Entre cada par existe exatamente um link
 - O link pode ser unidirecional, mas normalmente é bidirecional



Comunicação indireta

- ❑ As mensagens são direcionadas e recebidas de caixas de correio (também conhecidas como portas)
 - Cada caixa de correio tem um id exclusivo
 - Os processos só podem se comunicar se compartilharem uma caixa de correio
- ❑ Propriedades do link de comunicação
 - Link estabelecido somente se os processos compartilharem uma caixa de correio comum
 - Um link pode estar associado a muitos processos
 - Cada par de processos pode compartilhar vários links de comunicação
 - O link pode ser unidirecional ou bidirecional



Comunicação indireta

- Operações
 - cria uma nova caixa de correio
 - envia e recebe mensagens por meio da caixa de correio
 - destrói uma caixa de correio
- Primitivos são definidos como:
 - send**(*A, mensagem*) – envia uma mensagem à caixa de correio *A*
 - receive**(*A, mensagem*) – recebe uma mensagem da caixa de correio *A*



Comunicação indireta

- Compartilhamento de caixa de correio
 - P_1 , P_2 e P_3 compartilham caixa de correio A
 - P_1 envia; P_2 e P_3 recebem
 - Quem recebe a mensagem?
- Soluções
 - Permite que um link seja associado a no máximo dois processos
 - Permite que somente um processo de cada vez execute uma operação de recepção
 - Permite que o sistema selecione arbitrariamente o receptor.
 - Permite que o sistema selecione arbitrariamente o receptor. Emissor é notificado de quem foi o receptor.



Sincronismo

- ❑ A passagem de mensagens pode ser com bloqueio ou sem bloqueio
- ❑ **Com bloqueio** é considerada **síncrona**
 - **Envio com bloqueio** deixa o emissor bloqueado até que a mensagem é recebida
 - **Recepção com bloqueio** deixa o receptor bloqueado até que uma mensagem esteja disponível
- ❑ **Sem bloqueio** é considerada assíncrona
 - **Envio sem bloqueio** faz com que o emissor envie a mensagem e continue
 - **Recepção sem bloqueio** faz com que o receptor receba uma mensagem válida ou nulo



Buffers

- Fila de mensagens conectadas ao link;
implementados de três maneiras
 1. **Capacidade zero** – 0 mensagens
Emissor deve esperar pelo receptor
(*rendezvous*)
 2. **Capacidade limitada** – tamanho finito de n
mensagens
Emissor deve esperar se o link estiver cheio
 3. **Capacidade ilimitada** – tamanho infinito
Emissor nunca espera



```
public interface Channel
{
    // Send a message to the channel
    public abstract void send(Object item);

    // Receive a message from the channel
    public abstract Object receive();
}
```



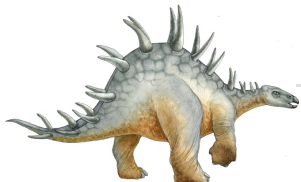
Buffer vinculado – Solução de passagem de mensagem

```
public class MessageQueue implements Channel
{
    private Vector queue;

    public MessageQueue() {
        queue = new Vector();
    }

    // This implements a nonblocking send
    public void send(Object item) {
        queue.addElement(item);
    }

    // This implements a nonblocking receive
    public Object receive() {
        if (queue.size() == 0)
            return null;
        else
            return queue.remove(0);
    }
}
```



O produtor

```
Channel mailBox;  
  
while (true) {  
    Date message = new Date();  
    mailBox.send(message);  
}
```

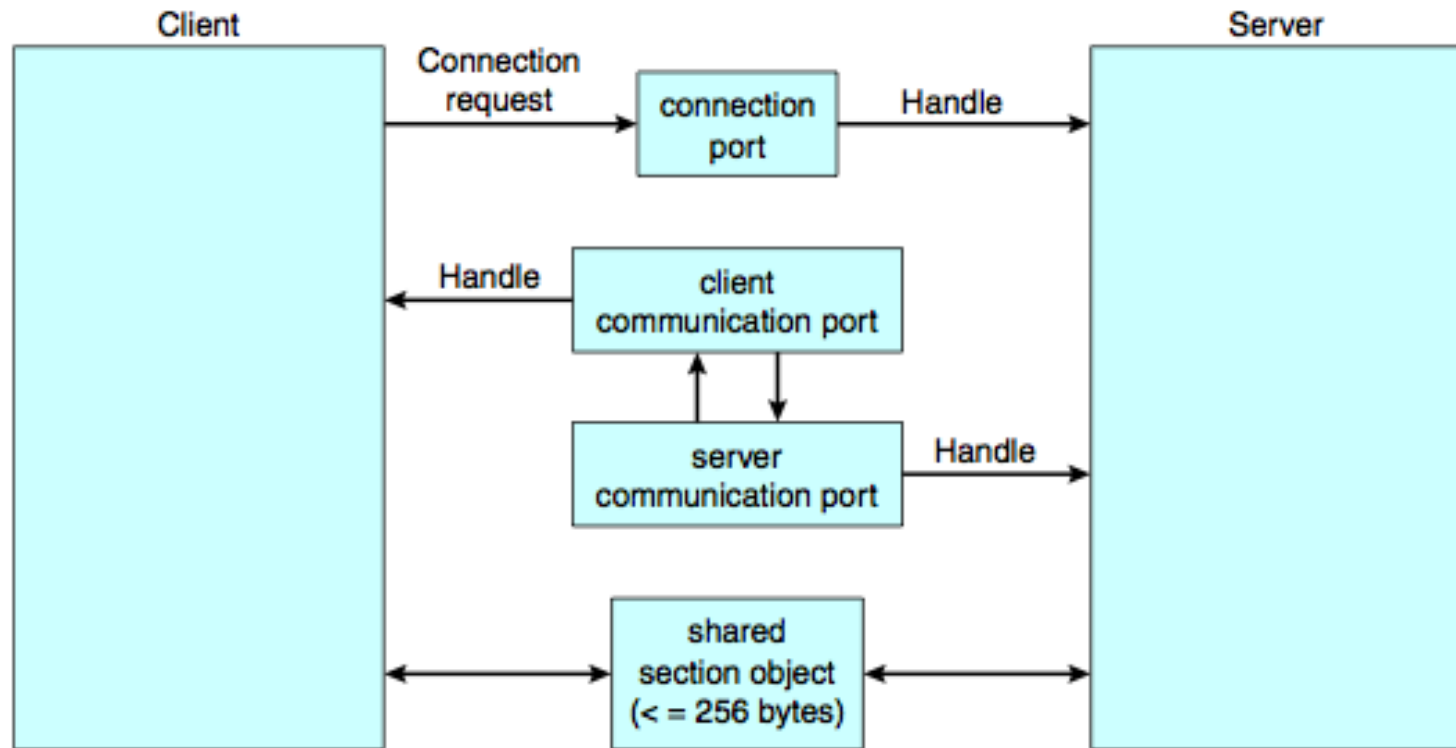


O consumidor

```
Channel mailBox;  
  
while (true) {  
    Date message = (Date) mailBox.receive();  
    if (message != null)  
        // consume the message  
}
```



Local Procedure Call (LPC) no Windows XP



Comunicação cliente-servidor

- ❑ *Sockets*
- ❑ Chamadas de procedimento remoto (RPC)
- ❑ Invocação de método remoto (RMI) - Java

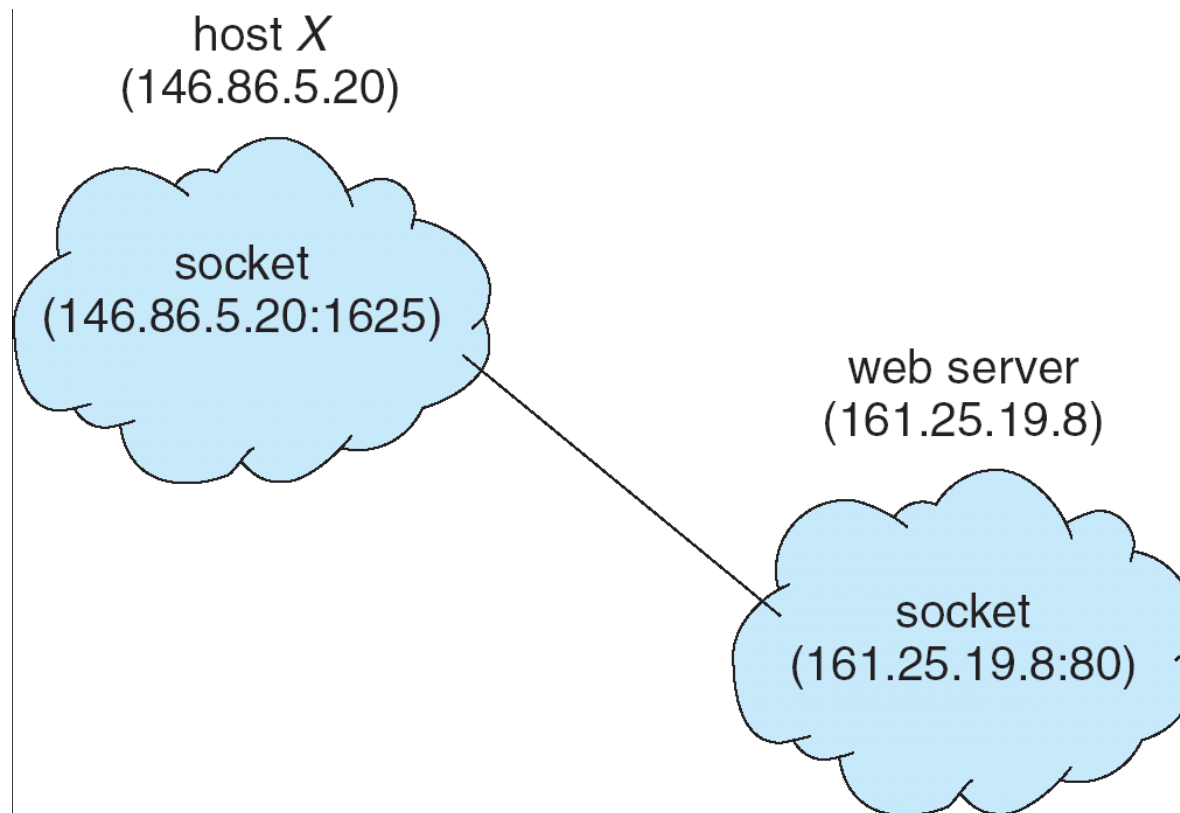


Sockets

- ❑ Um *socket* é definido como uma *extremidade para comunicação*
- ❑ Concatenação de endereço IP e porta
- ❑ O *socket* **161.25.19.8:1625** refere-se à porta **1625** no host **161.25.19.8**
- ❑ A comunicação acontece entre um par de *sockets*



Comunicação por socket



Comunicação por *socket* em Java

```
public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            // now listen for connections
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                // write the Date to the socket
                pout.println(new java.util.Date().toString());

                // close the socket and resume
                // listening for connections
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```



Comunicação por soquete em Java

```
public class DateClient
{
    public static void main(String[] args) {
        try {
            //make connection to server socket
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            // read the date from the socket
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            // close the socket connection
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

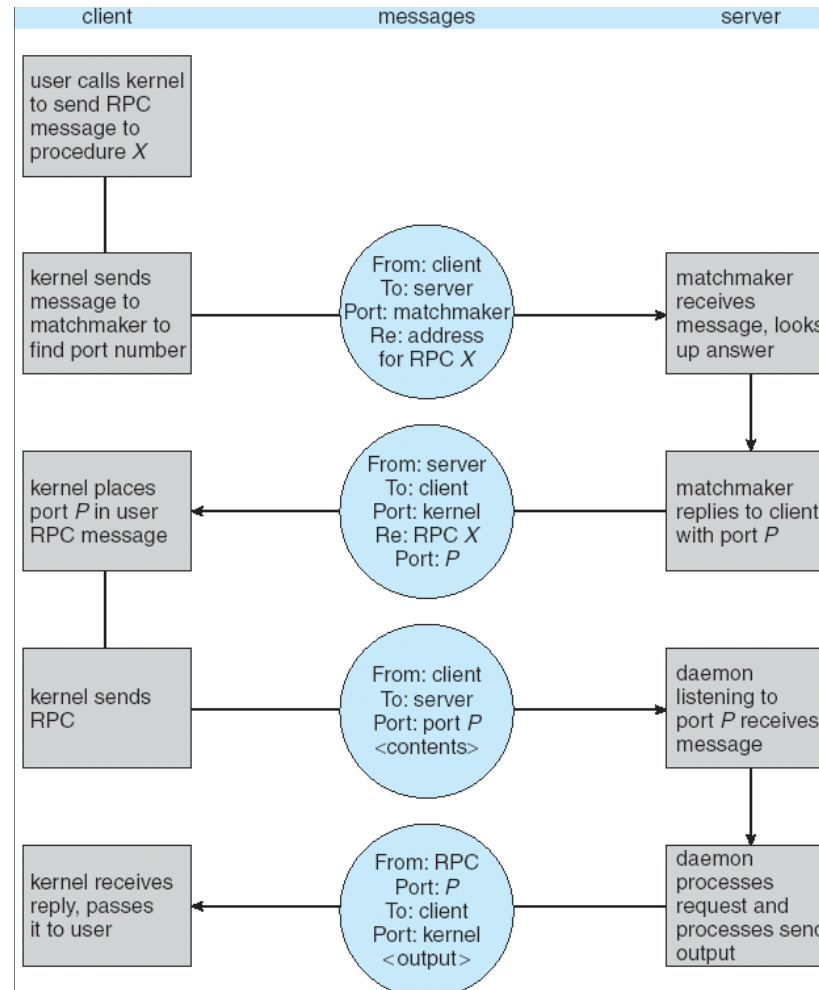


Chamadas de procedimento remoto

- ❑ Chamada de procedimento remoto (RPC) passa chamadas de procedimento entre processos nos sistemas em rede.
- ❑ **Stubs** – proxy no cliente para o procedimento real no servidor.
- ❑ O stub no cliente localiza o servidor e *organiza* os parâmetros.
- ❑ O *skeleton* (um “stub” no servidor) no servidor recebe essa mensagem, desempacota os parâmetros organizados e realiza o procedimento no servidor.

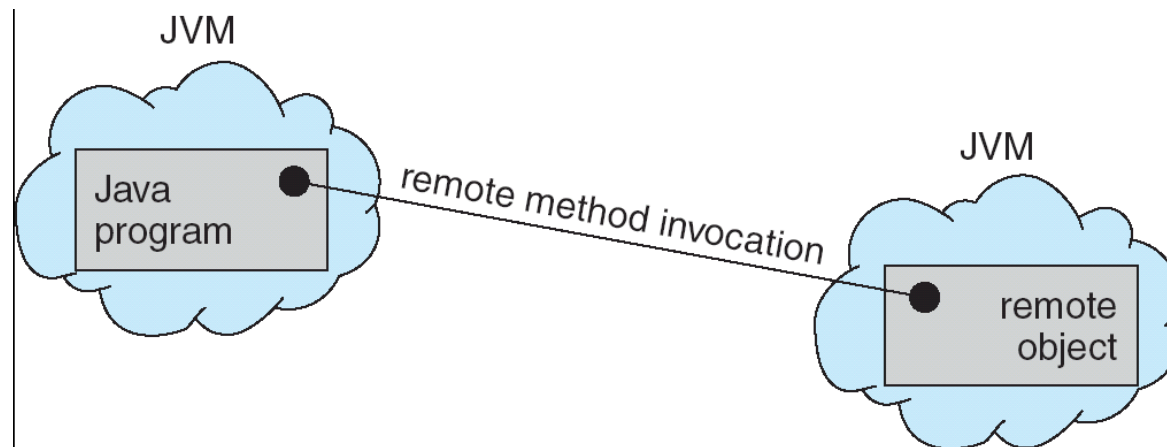


Execução da RPC

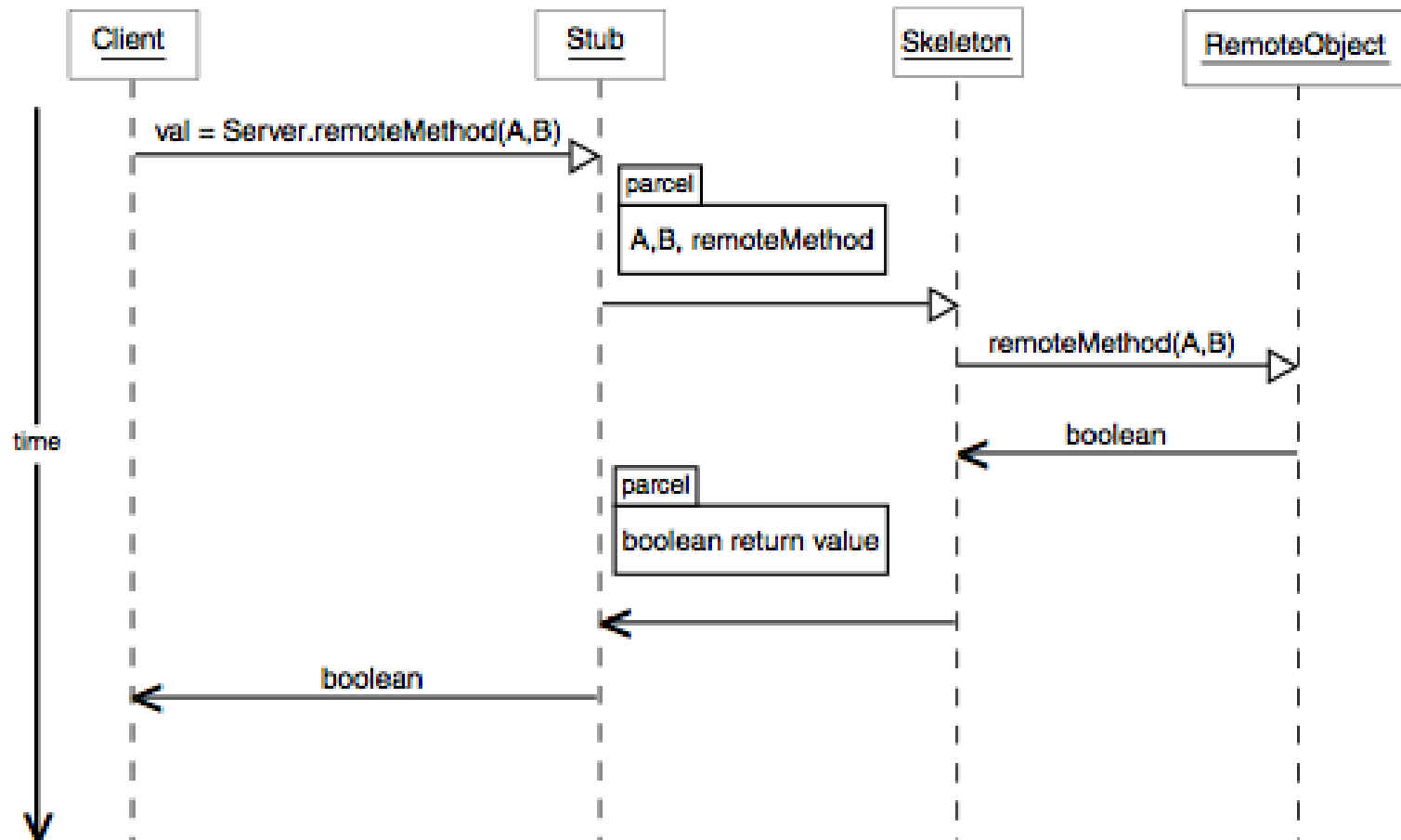


Invocação de método remoto

- ❑ Invocação de método remoto (RMI) é um mecanismo da Java semelhante às RPCs.
- ❑ RMI permite que um programa Java em uma máquina chame um método em um objeto remoto.



Organização de parâmetros



Exemplo de RMI

```
public interface RemoteDate extends Remote
{
    public abstract Date getDate() throws RemoteException;
}
```



Exemplo de RMI

```
public class RemoteDateImpl extends UnicastRemoteObject
    implements RemoteDate
{
    public RemoteDateImpl() throws RemoteException { }

    public Date getDate() throws RemoteException {
        return new Date();
    }

    public static void main(String[] args) {
        try {
            RemoteDate dateServer = new RemoteDateImpl();

            // Bind this object instance to the name "DateServer"
            Naming.rebind("DateServer", dateServer);
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}
```



Exemplo de RMI

```
public class RMIClient
{
    public static void main(String args[]) {
        try {
            String host = "rmi://127.0.0.1/DateServer";

            RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
            System.out.println(dateServer.getDate());
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}
```



Final do Capítulo 3

