Capítulo 3: Processos

Capitulo 3: Processos

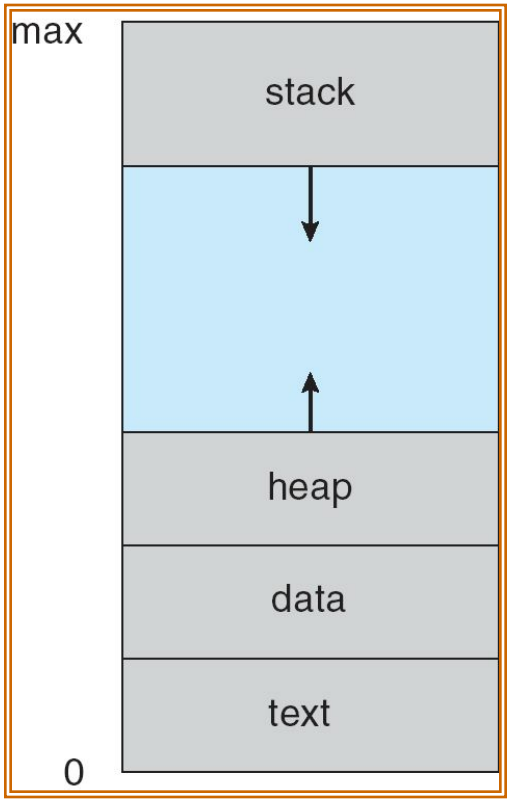
- Conceito de Processos
- Escalonamento de Processos
- Operacoes sobre Processos
- Cooperação entre Processos
- Comunicação Interprocessos
- Comunicação em Sistemas Cliente-Servidor

Conceitos

Conceito de Processos

- Um sistema operacional executa uma variedade de programas:
  - Batch system – jobs
  - Time-shared systems – programas do usuarios
- ***job*** ou ***process (text section)***
- **Processo** – é um programa em execução;
- Um processo inclui:
  - Contador de programa (program counter)
  - Stack (dados temporarios)
  - Heap (memoria alocada dinamicamente)
  - Seçao de dados (variaveis globais)

Processo na Memória



Stack
.acesso muito rapido
.espaço gerenciado pela CPU com uso eficiente
.memoria não ficará fragmentada
.não precisa de desalocação explícita
.armazena variaveis locais
.tamanho limitado (depende do SO)
.tamanho das variáveis nao pode ser redefinido

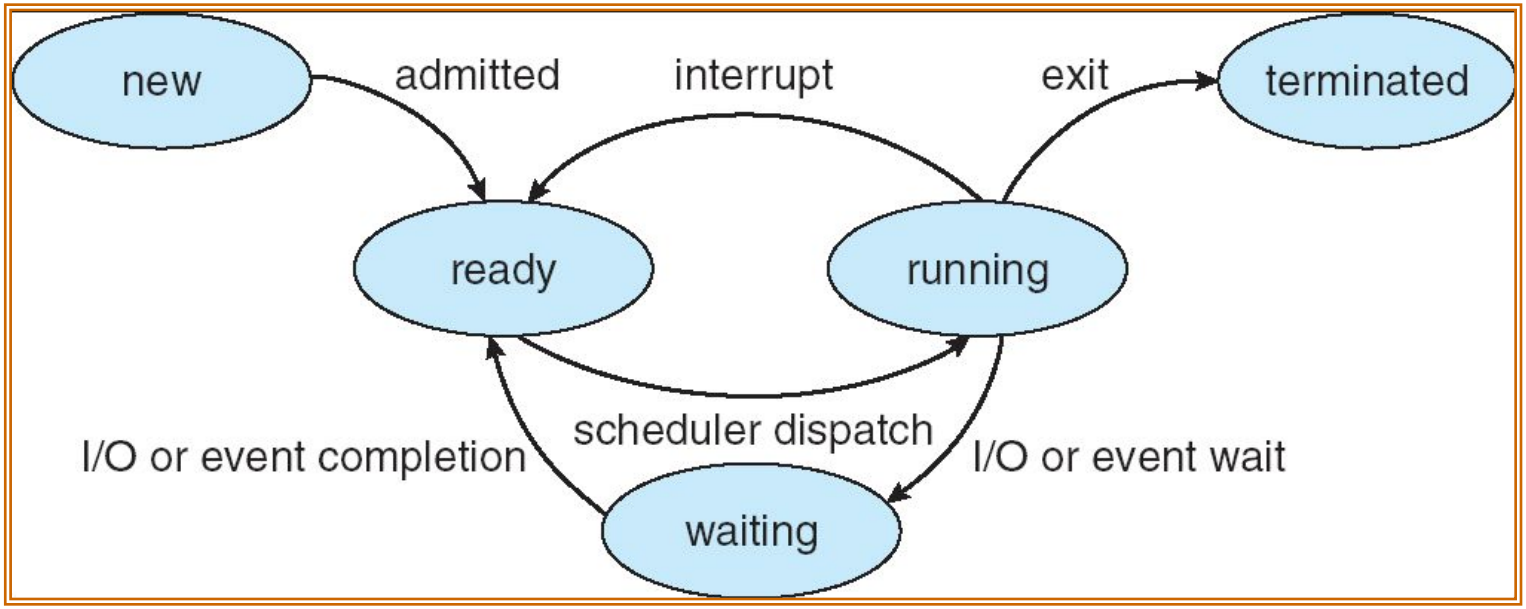
Heap
.variáveis podem ser acessadas globalmente
.sem limites de memoria (limite físico de hardware)
.acesso relativamente lento
.nenhuma garantia de uso eficiente
.pode ficar fragmentada
.gerenciado pelo programador
.variaveis podem ter tamanho alterado via realloc()

Fonte: http://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html

Estado de um processo

- Durante a execução um processo muda de estado
  - **new**: criando processo
  - **running**: instruções sendo executadas
  - **waiting**: processo aguardando algum evento ocorrer
  - **ready**: processo esperando execucao em um processador
  - **terminated**: O processo terminou a execucao

Diagrama de estados de um processo

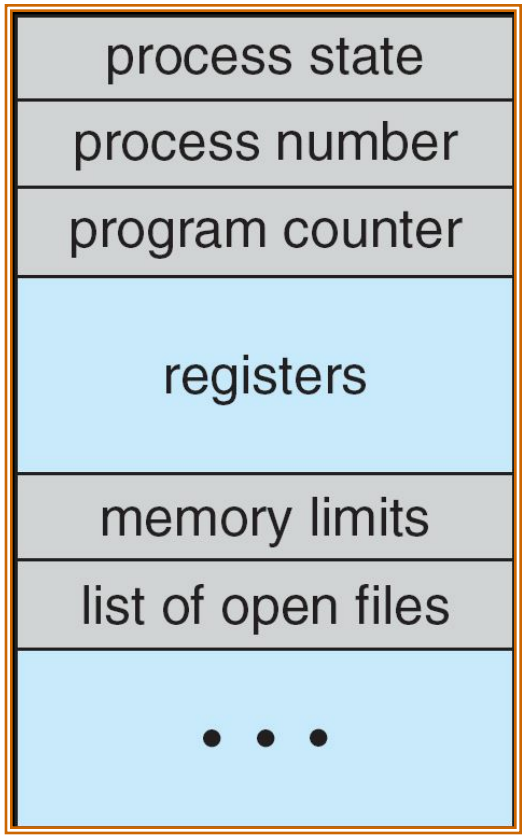


Process Control Block (PCB)

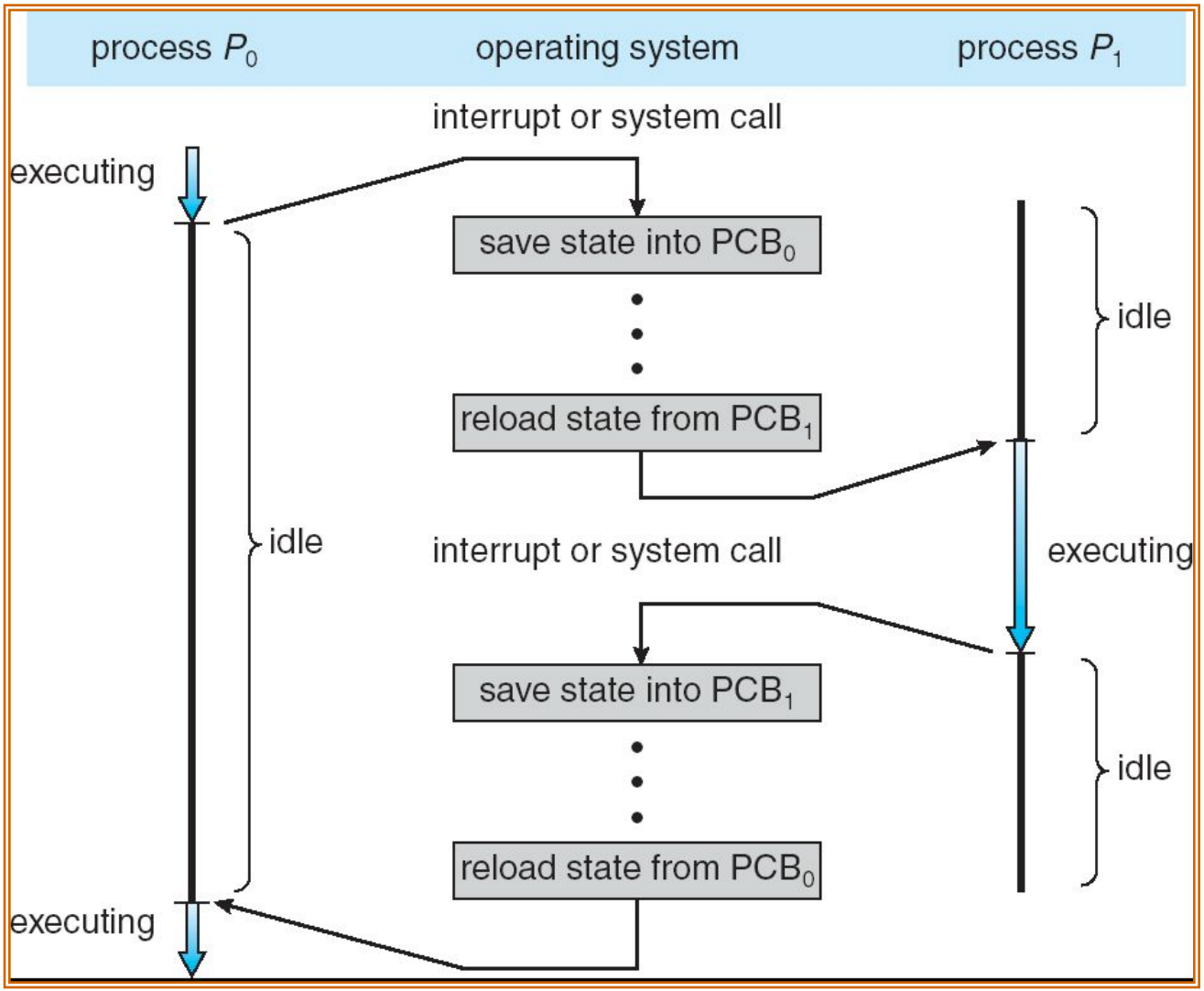
Informacao associada a cada processo

- Estado do processo
- Program counter
- Registradores da CPU
- Informacao sobre escalonamento de CPU
- Informacao sobre gerenciamento de memoria
- Informacao sobre contabilidade
- Informacao sobre E/S

Process Control Block (PCB)



Exemplo de Troca de processos na CPU



Trocando de contexto

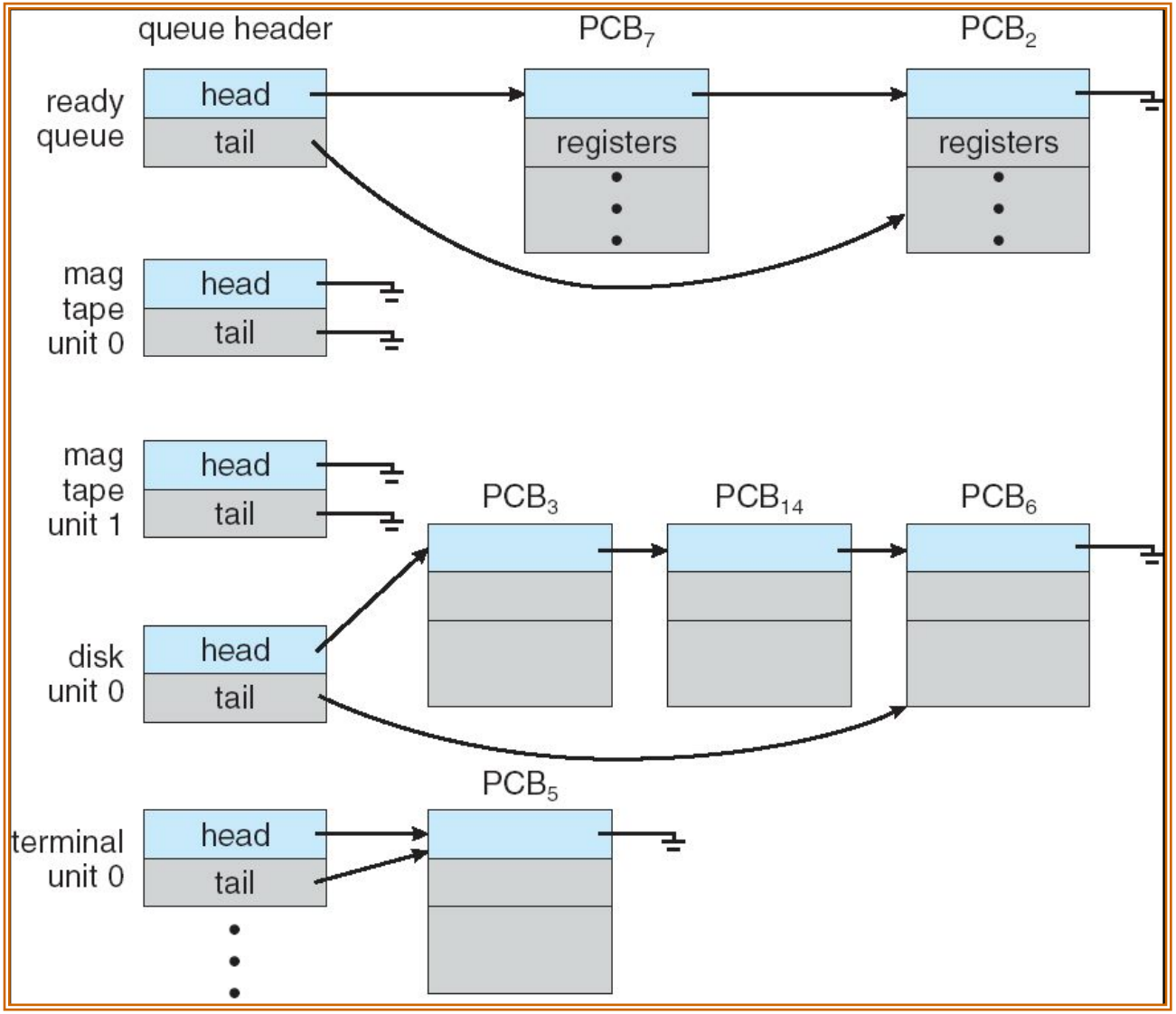
- O sistema deve salvar o estado do processo antigo e carregar o estado salvo do novo processo
- Troca de contexto é um overhead para CPU
- O tempo depende do hardware

Escalonamento de Processos

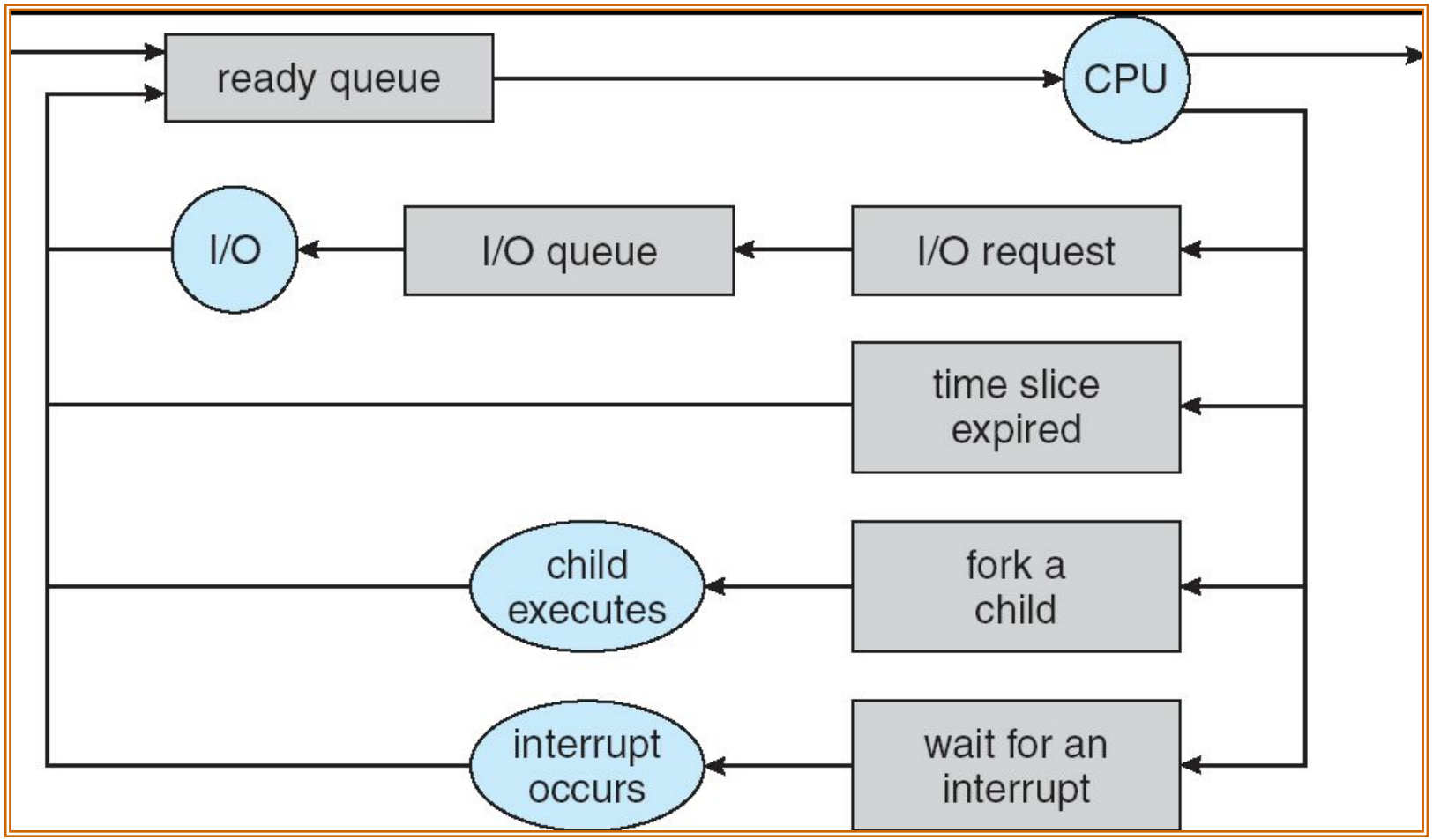
Filas de Escalonamento de Processos

- **Job queue** – todos os processos do sistema
- **Ready queue** – processos residentes em memoria, prontos e aguardando a execucao
- **Device queues** – processos aguardando um dispositivo de E/S
- Processos migrando de filas

Ready Queue varias filas dispositivos de E/S



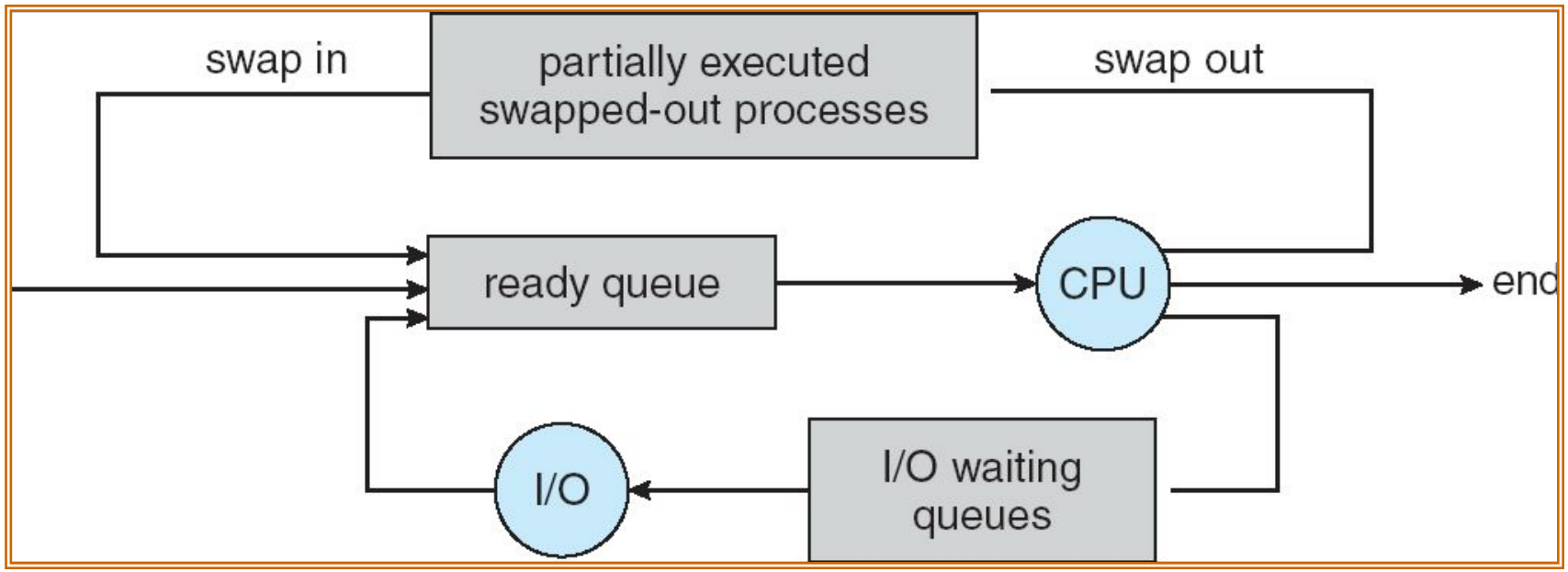
Representação de Escalonamento de Processos



Escalonadores

- **Long-term scheduler** (ou JOB scheduler) – seleciona quais processos devem ir para ready queue
- **Short-term scheduler** (ou CPU scheduler) – seleciona qual próximo processo deve ser executado

Adição de um Escalonador de Médio Termo

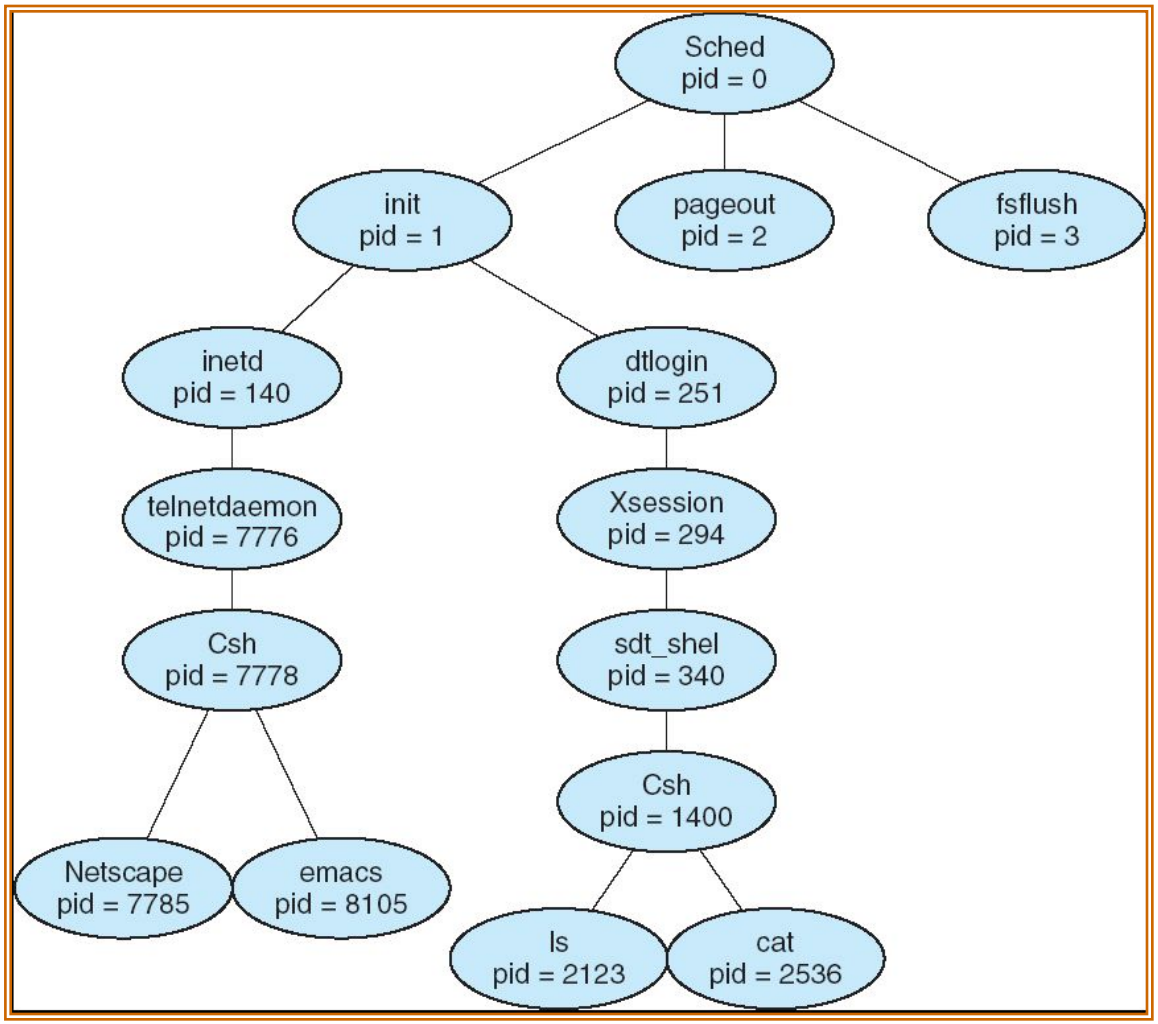


Escalonadores (Cont.)

- Short-term scheduler é invocado muito frequentemente (milliseconds) ⇒ (deve ser rapido)
- Long-term scheduler é pouco invocado (seconds, minutes) ⇒ (pode ser lento)
- O long-term scheduler controla o grau de multiprogramação
- Processos podem ser descritos como:
  - **I/O-bound** – gasta mais tempo fazendo E/S do que computacoes, periodos curtos de uso de CPU
  - **CPU-bound** – gasta mais tempo fazendo computacoes, uso prolongado de CPU

Criação de Processos

Uma árvore típica de processos do Solaris



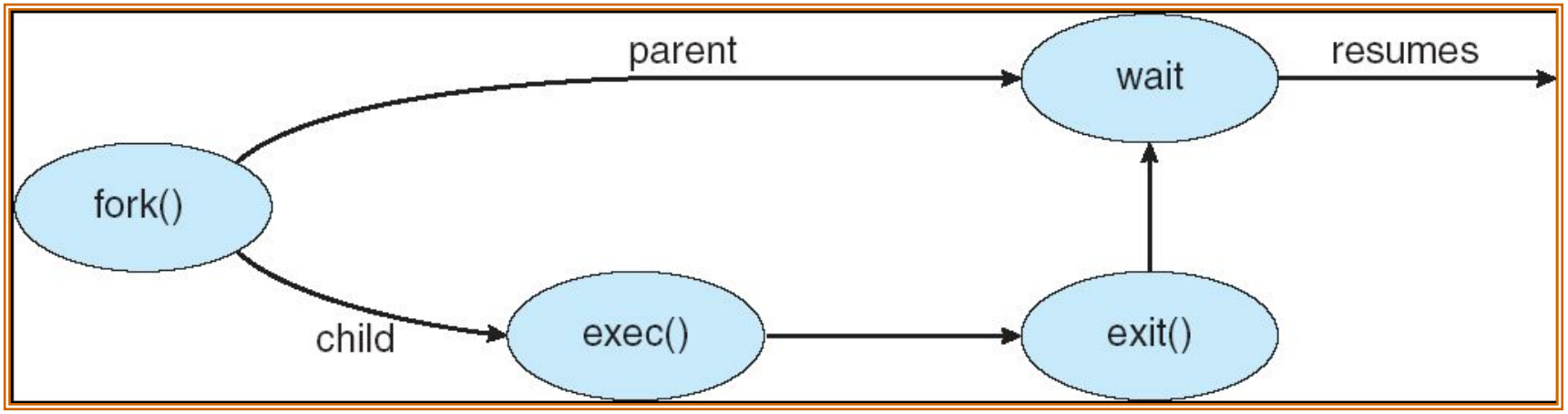
Criacao de Processos

- Processo pai cria processo filho, formando uma arvore
- Formas e compartilhamento de recursos
  - Pais e filhos compartilham todos recursos
  - Filhos compartilham parte do processo dos pais
  - Pais e filhos nao compartilham recursos
- Formas de execução
  - Pais e filhos executam concorrentemente
  - Pais aguardam até filhos terminarem

Criacao de Processos (Cont.)

- Espaco em memoria
  - Filho duplica o processo do pai
  - Filho possui um programa para carregar no espaco do pai
- UNIX
  - **fork** system call cria novo processo
  - **exec** system call usada apos o fork para substituir o novo espaco em memoria pelo novo programa

Criacao de Processos



Process Creation in 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);
    }
}
```

Process Creation in 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);
}
```

Process Creation in 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();
  }
}
```

Finalização de Processos

- Processo solicita o SO para elimina-lo (**exit**)
  - Retorno de dados do processo filho para o pai via **wait**
  - Recursos sao desalocados pelo SO

- Pais podem terminar execução de processos filhos (**abort**)
  - Filho pode exceder o recurso alocado
  - Tarefa associada ao processo filho nao é mais necessaria
  - Se pai esta terminando
    - Alguns S.O.'s nao permitem filho
      - Todos os filhos sao teminados –finalizacao em cascata (*cascading termination)*

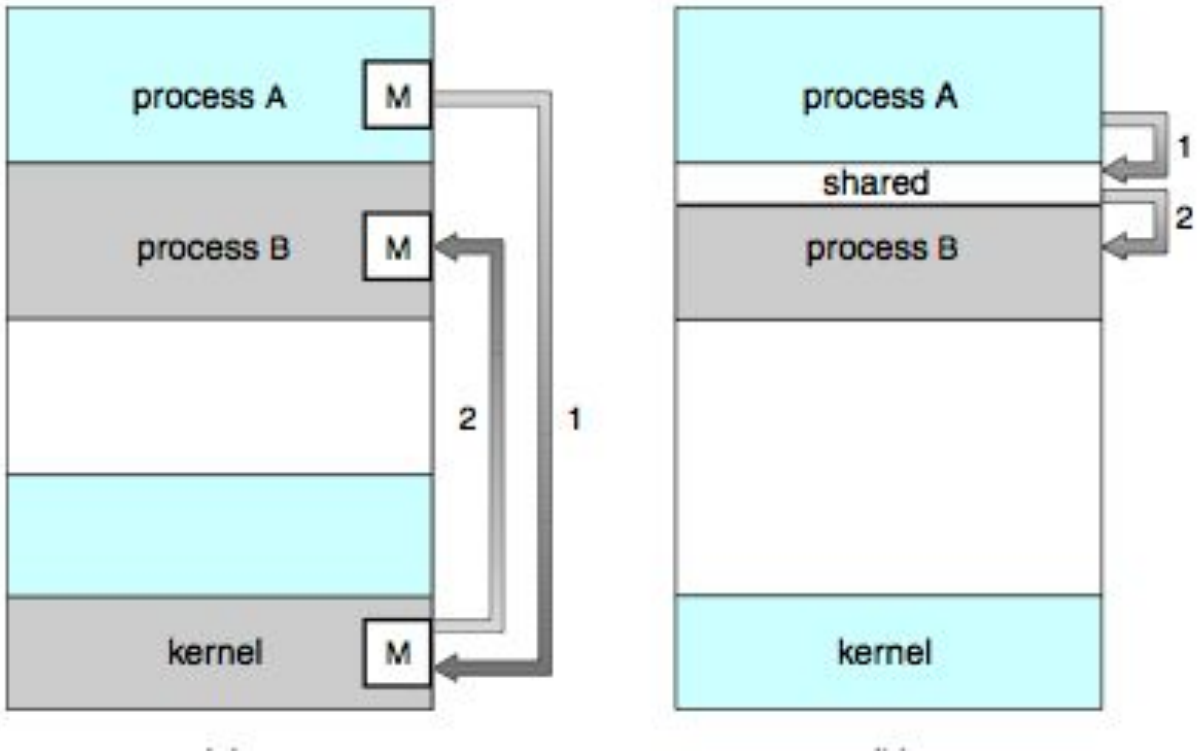
Comunicação entre Processos

Cooperação entre processos

- Todo processo é independente de outro processo;
- No entanto, processos podem cooperar, ou, trocar informações
- Qual a utilidade da Cooperação entre processos ?
  - Compartilhamento de informações
  - Acelerar Computação
  - Modularidade
  - Conveniência
- Cooperação entre processos requer:
  - Comunicação Interprocessos (em inglês IPC)

Comunicacao Interprocesso

Message Passing **Shared Memory**



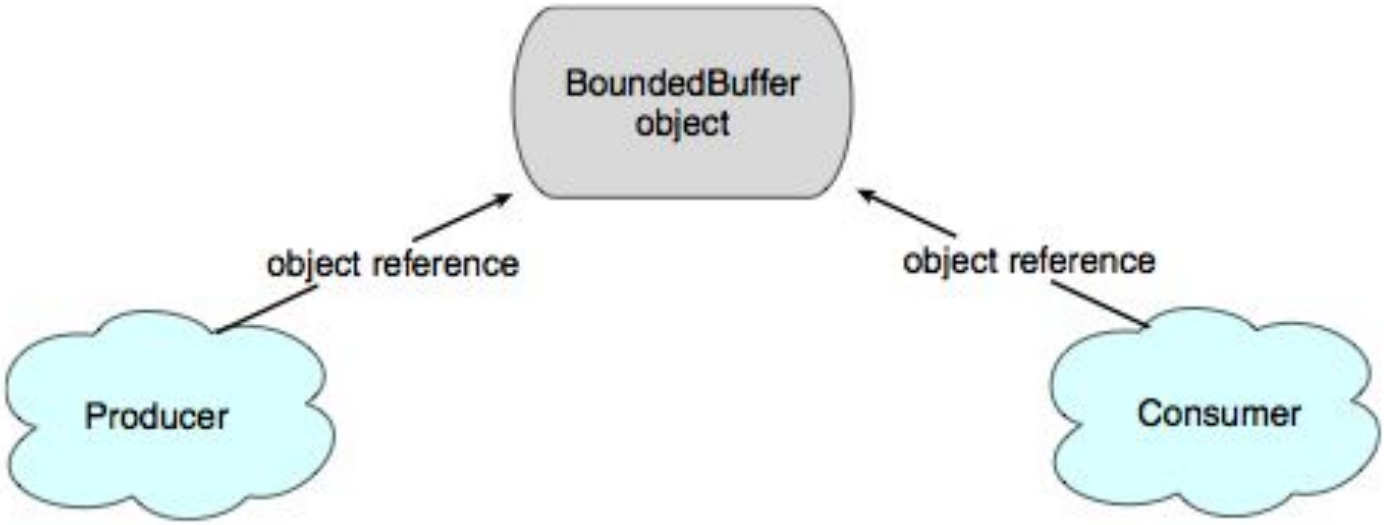
Problema Produtor-Consumidor

- Paradigma para cooperacao de processos,
  - Processo produtor produz informacao que é consumida pelo processo consumidor
  - *unbounded-buffer* nao limita o tamanho do buffer
  - *bounded-buffer* assume que existe um buffer de tamanho fixo

Sistemas de Memória Compartilhada

- Processos devem acordar em remover restrições de acesso à memoria compartilhada;
- Podem escrever ou ler da memória compartilhada;
- Tipicamente, a área de memória compartilhada reside no espaço de endereçamento do processo que cria o compartilhamento;

Simulando Memoria Compartilhada em Java



Bounded-Buffer – Solucao de memoria compartilhada

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

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

Bounded-Buffer – Solucao de memoria 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
   }
```

Bounded-Buffer -- Figura 3.16 - insert() method

```
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;
}
```

Bounded-Buffer – Figure 3.17 - remove() method

```
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 Mensagem

- Sistema de Mensagem – nao usa variaveis compartilhadas
- Prove duas operacoes:
  - **send**(*message*) – mensagem de tamanho fixo
  - **receive**(*message*)
- Para processos se comunicarem:
  - Estabelecer uma link de comunicacao
  - Trocar mensagens via send/receive
- Implementacao de um link de comunicacao
  - Fisico (e.g., memoria compartilhada, barramento hardware)
  - Logico (e.g., propriedades logicas)

Questoes

- Como os links sao estabelecidos?
- Pode um link ser associado com mais de dois processos?
- Como varios links podem existir entre cada par de processos?
- Qual é a capacidade de um link?
- Uma mensagem deve ser de tamanho fixo ou variavel?
- Um link é unidirecional ou bidirecional?

Comunicação Direta

- Processos devem se referenciar explicitamente:
  - **send** (*P, message*)
  - **receive**(*Q, message*)

- Propriedades de um link
  - Sao estabelecidos automaticamente
  - Um link para cada par de processos
  - Entre cada par existe exatamente um link
  - O link pode ser unidirecional, mas é em geral bidirecional

Comunicação Indireta

- Mensagens são direcionadas e recebidas via mailbox (portas)
  - Cada mailbox tem um único id
  - Processos podem comunicar somente entre mailbox compartilhadas
- Propriedades de um link de comunicação
  - Link estabelecido somente se os processos compartilham um mailbox comum
  - Um link pode ser associado com vários processos
  - Cada par de processos pode compartilhar vários links de comunicação
  - Link pode ser unidirectional ou bidirectional

Comunicacao Indireta

- Operacoes
  - Criar uma nova mailbox
  - Enviar e receber mensagens atraves de mailbox
  - Destruir uma mailbox
- Primitivas sao definidas como:

  send(*A, message*) – enviar mensagem para mailbox A

  receive(*A, message*) – receber mensagem da mailbox A

Comunicacao Indireta

- Compartilhando Mailbox
  - P_1, P_2, e P_3 compartilham mailbox A
  - P_1, envia; P_2 e P_3 recebem
  - Quem pega a mensagem?
- Solucoes
  - Permitir um link ser associado com no maximo 2 processos
  - Permitir que um unico processo por vez execute a operacao receive
  - Permitir o sistema selecionar arbritariamente o receptor. Emissor é notificado sobre quem é o receptor.

Sincronização

- Passagem de mensagem pode ser blocking ou non-blocking

- **Blocking** é considerado **síncrono**
  - **Blocking send** o emissor é bloqueado até que a mensagem seja recebida
  - **Blocking receive** receptor é bloqueado até ter mensagem disponível

- **Non-blocking** é considerado **assíncrono**
  - **Non-blocking send** o processo emissor envia a mensagem e finaliza operação
  - **Non-blocking receive** o receptor recebe uma mensagem válida ou nula

Bufferização

- Fila de mensagens anexada ao link; implementada de tres maneiras
  1. **Zero capacity** – 0 mensages
     Emissor deve esperar pelo receptor (rendezvous)
  2. **Bounded capacity** – tamanho finito do numero de mensagens. Emissor deve aguardar se link estiver cheio
  3. **Unbounded capacity** – tamanho infinito
     Emissor nunca aguarda

Bounded-Buffer – Solucao de Passagem de Mensagem

```
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();
}
```

Bounded-Buffer – Solucao 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);
    }
}
```

Bounded-Buffer – Solucao de Passagem de Mensagem

The Producer

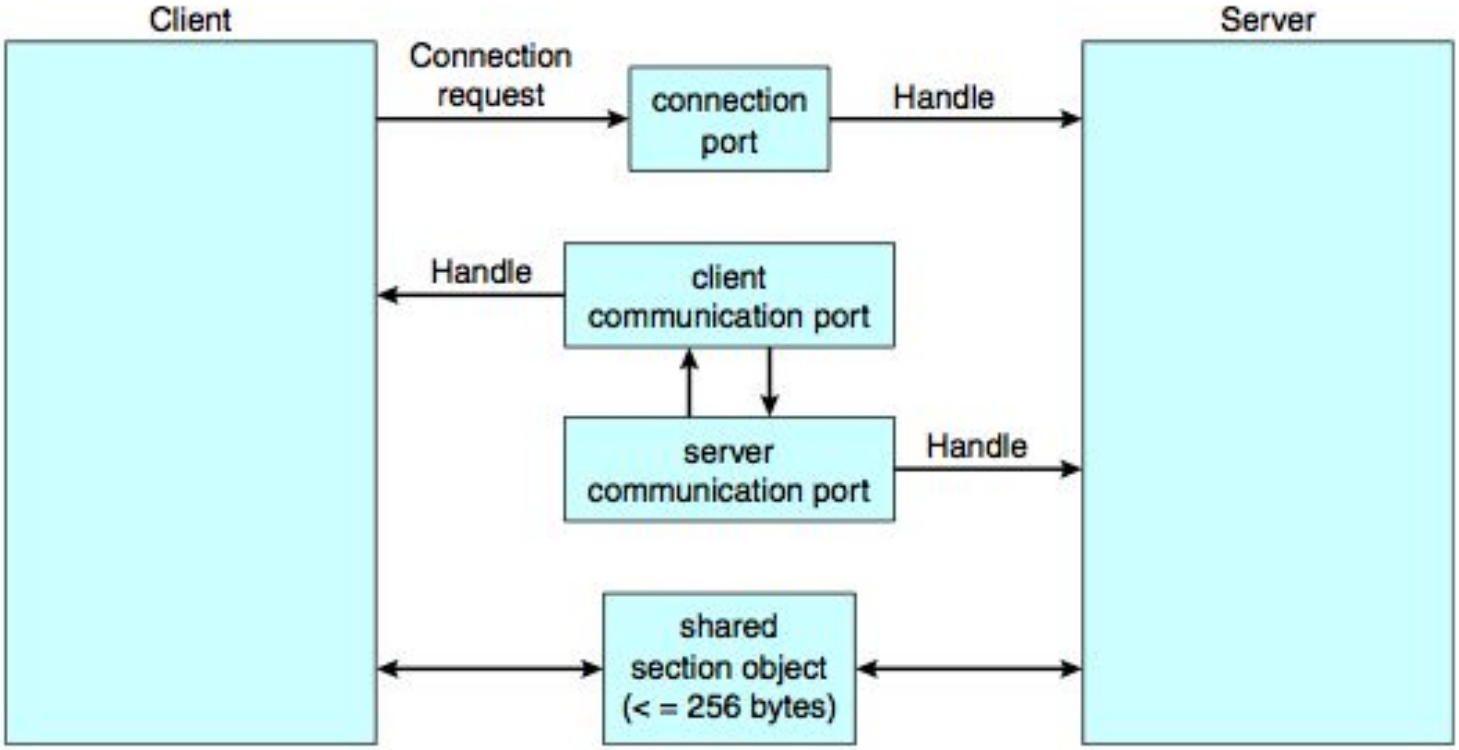
```
Channel mailBox;

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

Bounded-Buffer – Solucao de Passagem de Mensagem

The Consumer

```
Channel mailBox;

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

Troca de Mensagens no Windows XP



Tipos de Comunicação

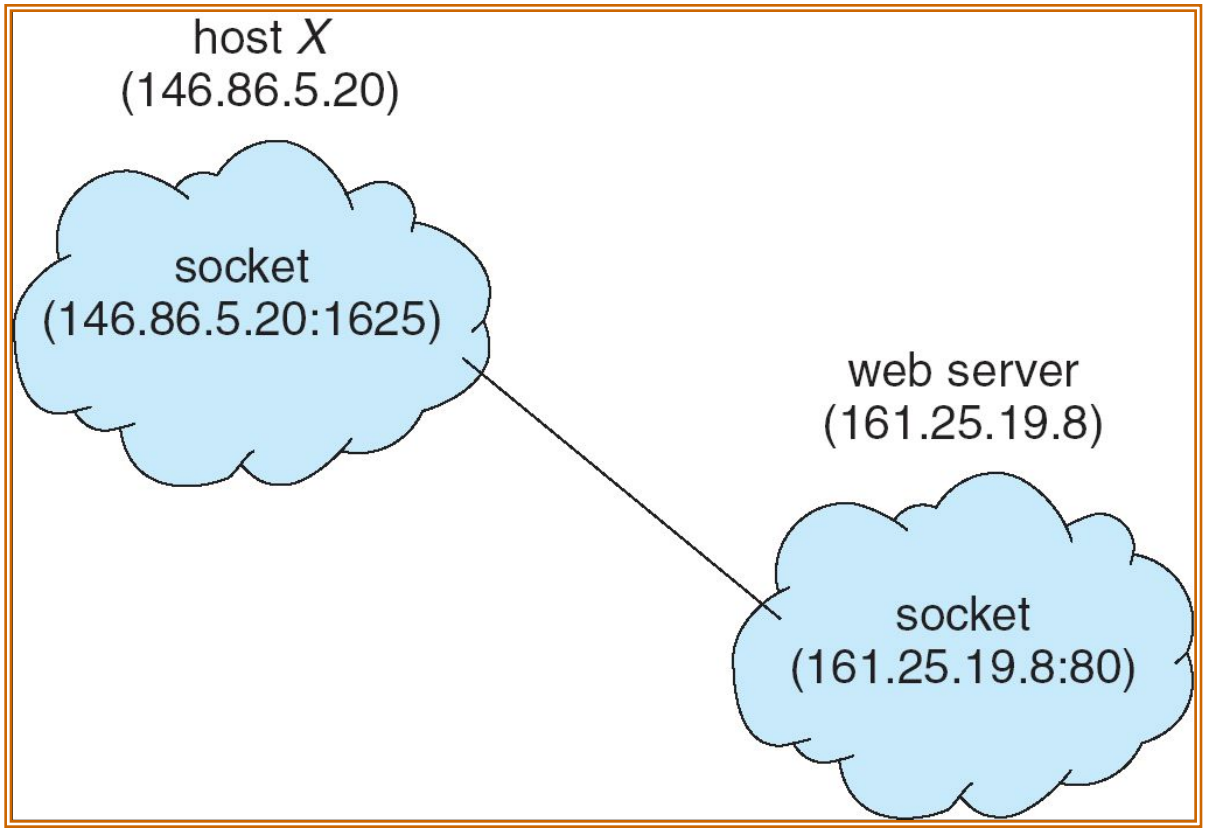
Comunicacao Cliente-Servidor

- Socket
- Remote Procedure Call (RPC)
- Java Remote Method Invocation (RMI)

Sockets

- Definido como um ponto final da comunicacao
- Concatenacao de endereco IP e porta
- O socket **161.25.19.8:1625** refere-se a porta **1625** do computador **161.25.19.8**
- A comunicacao consiste de um par de sockets

Comunicacao de Socket



Comunicacao com 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);
      }
   }
}
```

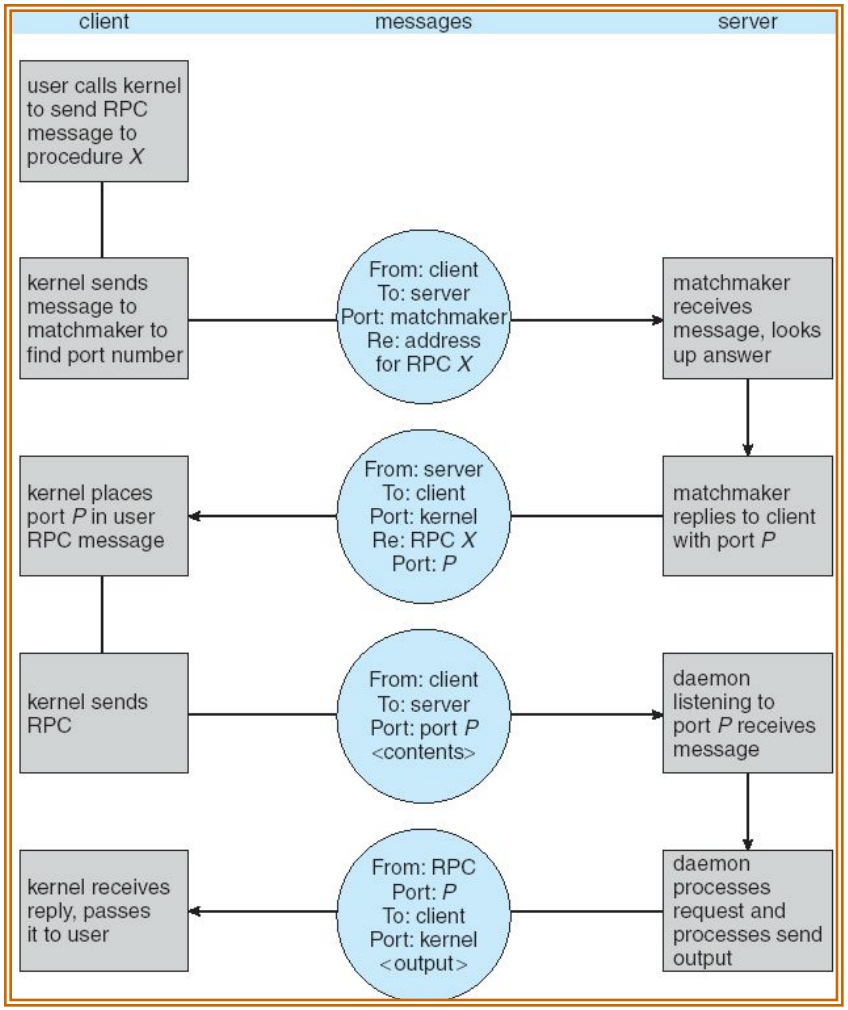
Comunicacao com Socket em Java

```
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);
      }
   }
}
```

Remote Procedure Calls

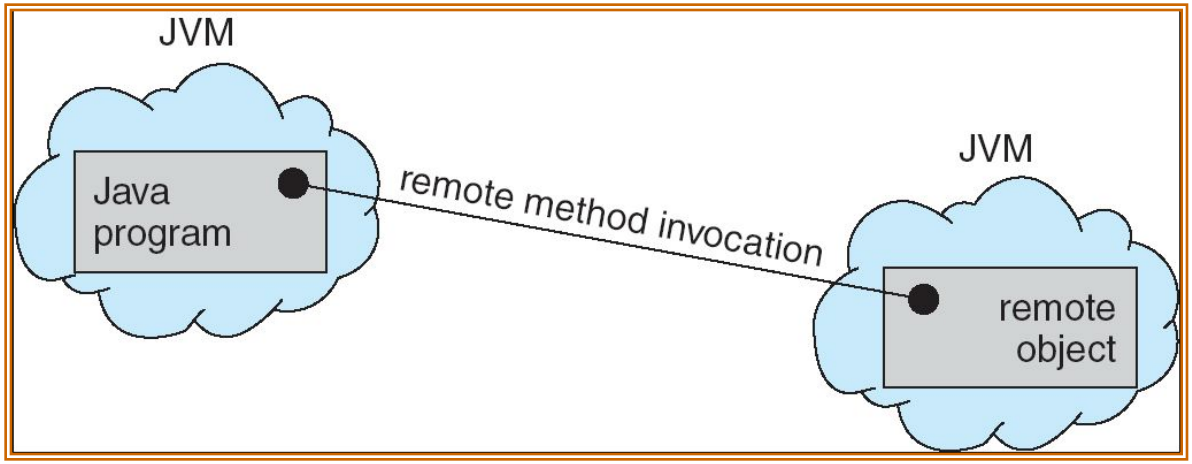
- RPC abstrai chamada a procedimentos entre processos executando em rede.
- **Stubs** – um proxy no lado do cliente representando o procedimento armazenado no servidor.
- O stub do lado do cliente localiza o servidor e prepara a mensagem de envio com os parametros.
- O stub do lado do servidor recebe a mensagem, recupera os parametros, e executa o procedimento no servidor.

Execucao do RPC

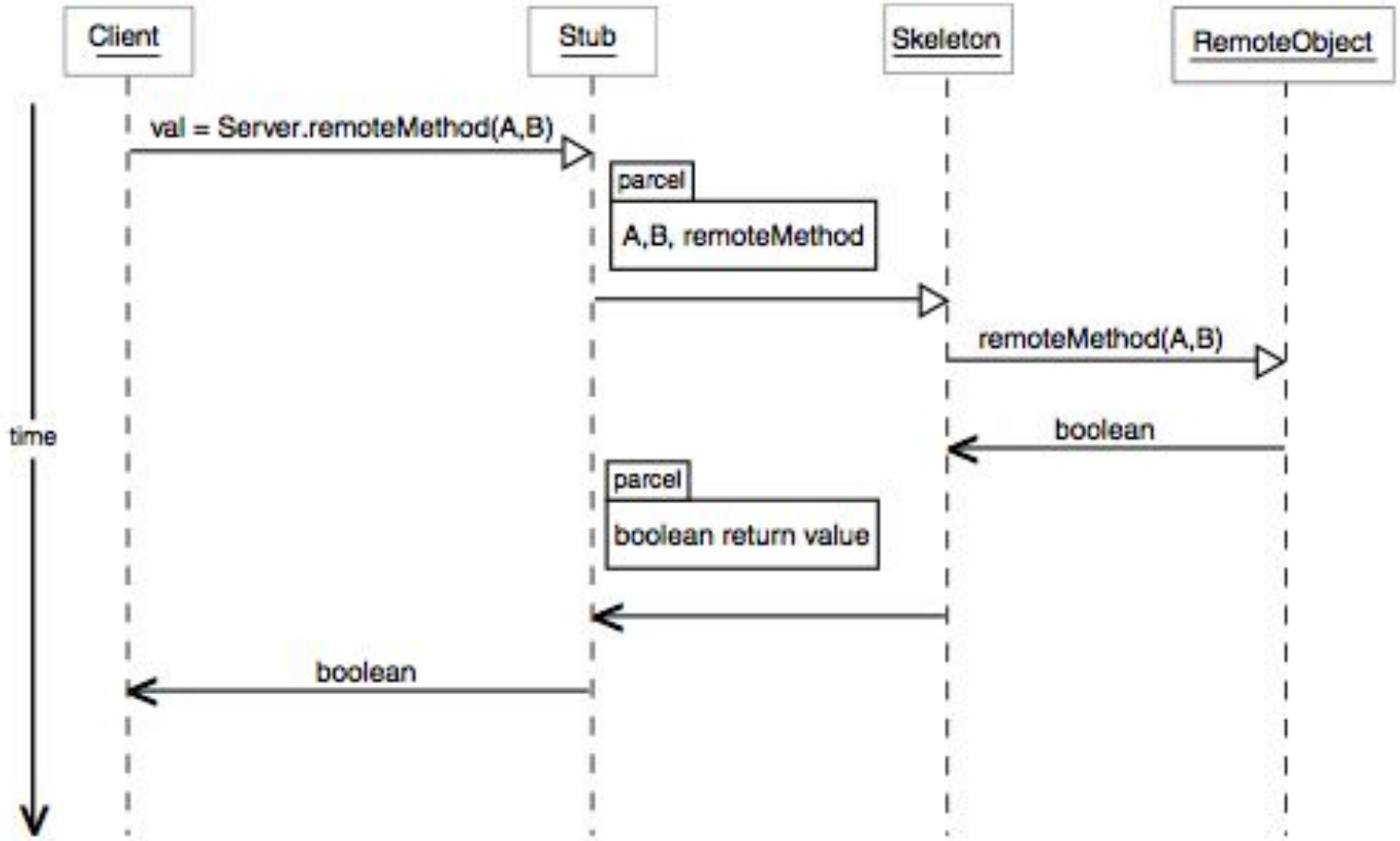


Remote Method Invocation (RMI)

- Mecanismo Java similar aos RPCs.
- RMI permite programa Java em um maquina invocar metodo de um objeto remoto.



Executando a chamada



Exemplo RMI

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

Exemplo RMI – Lado Servidor

```
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 RMI – Lado Cliente

```
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);
      }
   }
}
```

Fim do Capitulo 3