Computação Concorrente (DCC/UFRJ)

Aula 10: Comunicação entre threads via troca de mensagens

Prof. Silvana Rossetto

29 de maio de 2012

1. Sistemas de troca de mensagens
   - Primitivas de troca de mensagens
   - Questões de projeto send/receive
   - Endereçamento e formato das mensagens
   - Canais de comunicação
2. Exclusão mútua entre processos usando mensagens
3. Programas distribuídos
   - Processamento concorrente no paradigma C/S

Interação entre threads/processos

Retomando requisitos básicos...

- **Sincronização**: garantir acesso exclusivo aos recursos compartilhados
- **Comunicação**: permitir a troca de informações entre threads

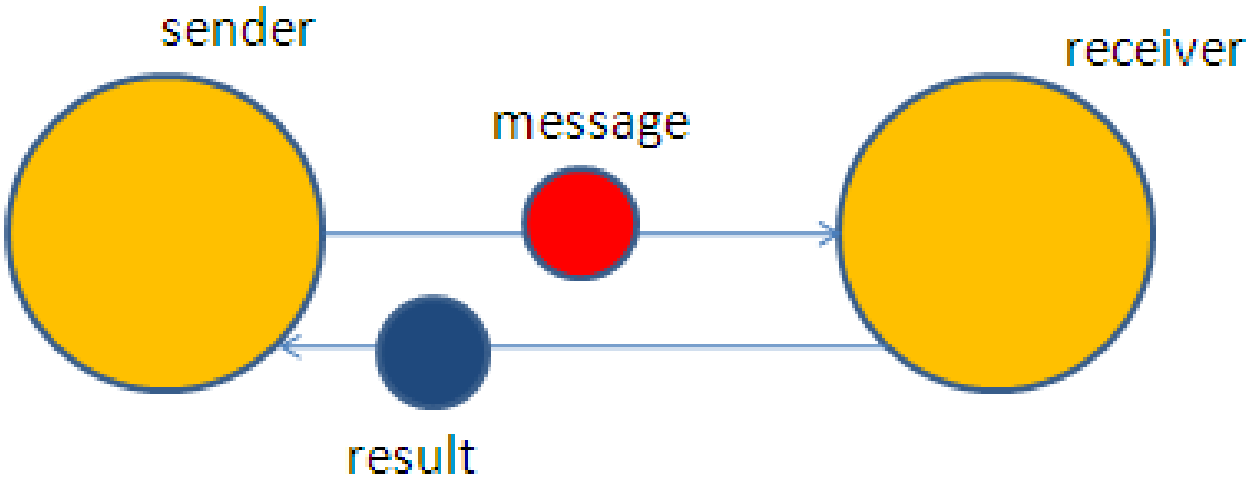
Abordagem com troca de mensagens

Primitivas de troca de mensagens

- **send (dst, msg)**
- **receive (src, msg)**

podem ser usadas em ambientes de memória compartilhada ou máquinas distribuídas

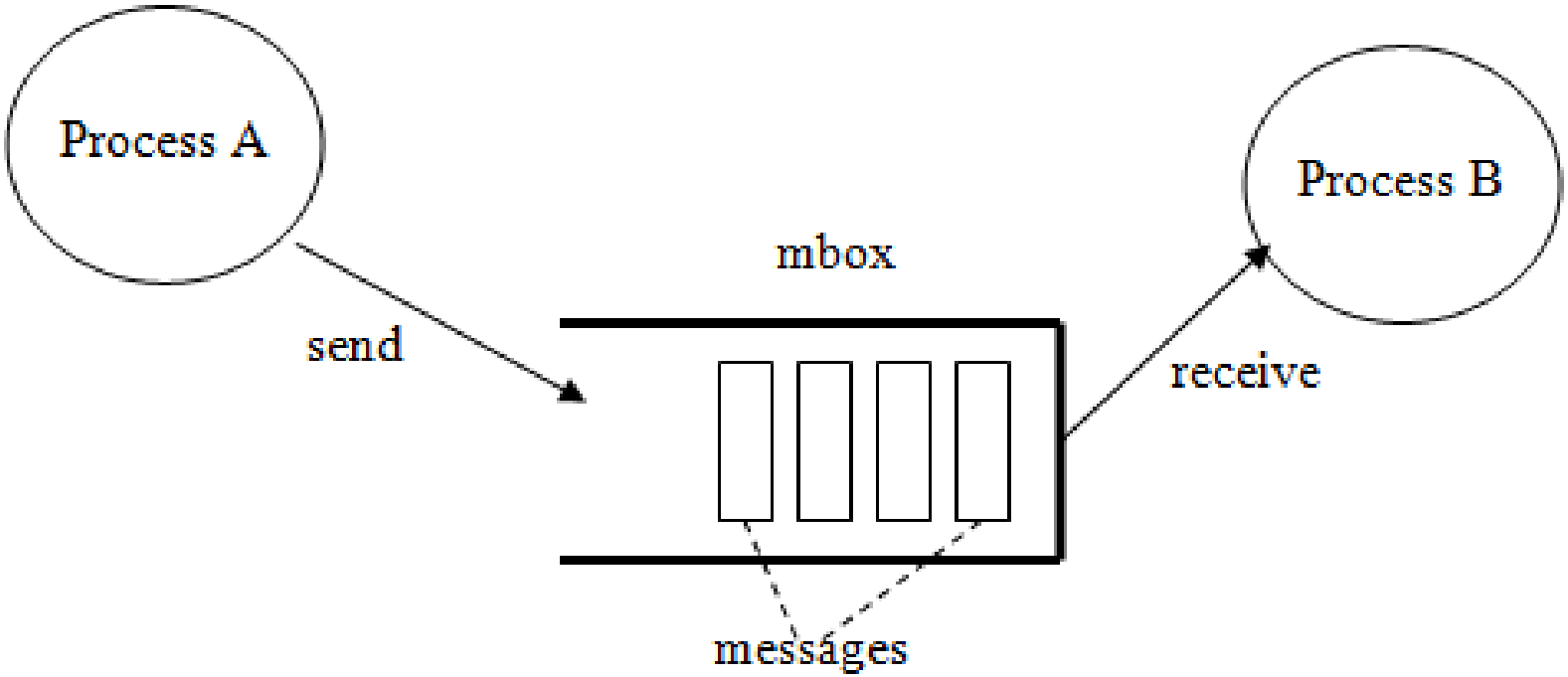
Troca de mensagens com endereçamento direto



1

[1] *Fonte: http://ajlopez.wordpress.com*

Troca de mensagens com endereçamento indireto



[2] *Fonte: http://cnx.org/content/m28019/latest*

Produtor/Consumidor usando mensagens

```
void produtor() {
  message pmsg;
  while(true) {
    receive(pode_produzir, pmsg);
    pmsg = produz_item();
    send(pode_consumir, pmsg);
  }
}

void consumidor() {
  message cmsg, msg_vazia;
  while(true) {
    receive(pode_consumir, cmsg);
    send(pode_produzir, msg_vazia);
    consome(cmsg);
  }
}
```

```
void main() {
  int N //tamanho do buffer
  message msg_vazia;
  createMailbox(pode_produzir);
  createMailbox(pode_consumir);

  for(int i=1; i<=N; i++)
    send(pode_produzir, msg_vazia);
  //inicia os processos
}
```

Questões de projeto

1. Uma thread não pode receber uma mensagem que ainda não foi enviada
2. É necessário especificar o que acontece com um processo/thread depois que ela invoca as primitivas **send/receive**

Envio de mensagem: assíncrona e síncrona

O **envio** de mensagem pode ser **assíncrona ou síncrona**:

- **Assíncrona**: as mensagens ainda não recebidas são armazenadas em uma fila ou canal de mensagens (o processo que enviou a mensagem pode proceder assincronamente com relação ao processo que irá receber a mensagem)
- **Síncrona**: o processo que envia a mensagem fica bloqueado até que a mensagem seja recebida

Vantagens do envio síncrono

- **Não é preciso dimensionar o tamanho das filas** ou canais de mensagens (cada processo terá no máximo uma mensagem no seu canal de comunicação)
- Quando um processo termina de enviar uma mensagem ele **sabe que o receptor foi notificado** de alguma forma
- É mais **fácil descobrir quando um processo não está mais respondendo**

Desvantagens do envio síncrono

- **Redução da concorrência** (o processo que envia a mensagem fica bloqueado até que o processo destino esteja pronto para recebê-la)
- **Possibilidade maior de ocorrência de deadlock** (garantir que sempre que um processo envia uma mensagem, outro processo esteja esperando uma mensagem desse processo e não tentando enviar)

Bloqueio no recebimento

Comportamento do receptor

Quando a **primitiva receive é executada** há também duas possibilidades:

- se a mensagem já foi enviada, a mensagem é recebida e a execução segue
- se a mensagem ainda não foi enviada:
  1. a **thread pode ser bloqueada até a mensagem chegar** (MAIS USUAL), ou
  2. a **thread continua executando e a tentativa de recebimento é abandonada** (MENOS USUAL)

Combinações mais comuns de comportamentos

1. **envio bloqueante, recebimento bloqueante**: emissor e receptor são bloqueados até a mensagem ser entregue (*rendezvous*)
   - caracteriza o tipo mais forte de sincronização
2. **envio não-bloqueante, recebimento bloqueante**: o receptor é bloqueado até a mensagem chegar
   - é a combinação mais usual e permite que uma thread envie mensagens para vários destinos em sequência
3. **envio não-bloqueante, recebimento não-bloqueante**: nenhuma parte fica bloqueada

Propósitos do bloqueio do emissor

1. Gerência de recursos: ex., a thread emissora não pode modificar dados de saída até que o sistema tenha copiado os valores antigos para uma localização segura
2. Semântica de falhas: como a comunicação em rede é susceptível a falhas, o bloqueio do emissor garante que ele irá esperar até a mensagem ser entregue com sucesso
3. Parâmetros de retorno: nos casos em que uma mensagem constitui uma requisição para a qual uma resposta é esperada, o emissor fica bloqueado até receber a resposta

Tempo de bloqueio do emissor

Requisitos de bufferização

Particularmente no caso do "envio sem espera", haverá um **limite quando o espaço de buffer for preenchido**, então o processo deverá bloquear de fato

Endereçamento direto, emissor explícito

Destino identificado

O **receptor define explicitamente o emissor da mensagem** (nem sempre é viável, um servidor de impressão pode receber requisições de vários clientes)

Thread P
(emissor)

```
.
.
send(Q, msg);
.
.
```

Thread Q
(receptor)

```
.
.
receive(P, &msg);
.
.
```

Endereçamento direto, emissor implícito

Destino identificado

O **receptor usa endereçamento implícito**, nesse caso o valor do parâmetro fonte é preenchido quando a operação é concluída

Thread P
(emissor)

```
.
.
send(Q, msg);
.
.
```

Thread Q
(receptor)

```
.
.
receive(&msg);
.
.
```

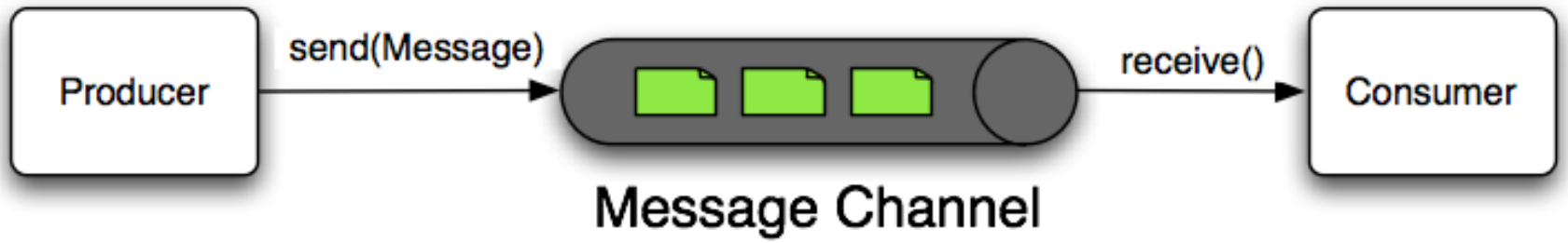
Endereçamento indireto

Canais de mensagens

Mensagens enviadas para **canais de mensagens** (vantagem: desacoplamento de emissor e receptor)

Implementação de canais

Threads dentro de um mesmo processo podem acessar **objetos do tipo canal** implementados na memória compartilhada



Implementação de canais

Podemos distinguir três tipos de canais, baseado no número de threads emissoras e receptoras que podem acessá-lo:

1. **Caixa de mensagem** (*mailbox*): vários emissores e vários receptores podem acessar o canal;
2. **Porta** (*port*): vários emissores e apenas um receptor podem acessar o canal;
3. **Enlace** (*link*): um par emissor e receptor pode acessar o canal.

Sincronização entre processos usando mensagens

Quando processos de uma aplicação interagem via troca de mensagens, **as próprias primitivas de comunicação oferecem suporte para sincronizar** ou coordenar as várias atividades da aplicação

Exclusão mútua com troca de mensagens

```
const int N //número de processos/threads da aplicação
void P(mailbox mb) {
   message msg;
   while(true) {
      receive(mb, msg);
      //executa a seção crítica
      send(mb, msg);
      //executa fora da seção crítica
   }
}
void main() {
   createMailbox(mb);
   send(mb, null);
   //inicia os N processos/threads e passa mailbox
}
```

Qual deve ser a semântica das primitivas **send** e **receive**?

Programas distribuídos

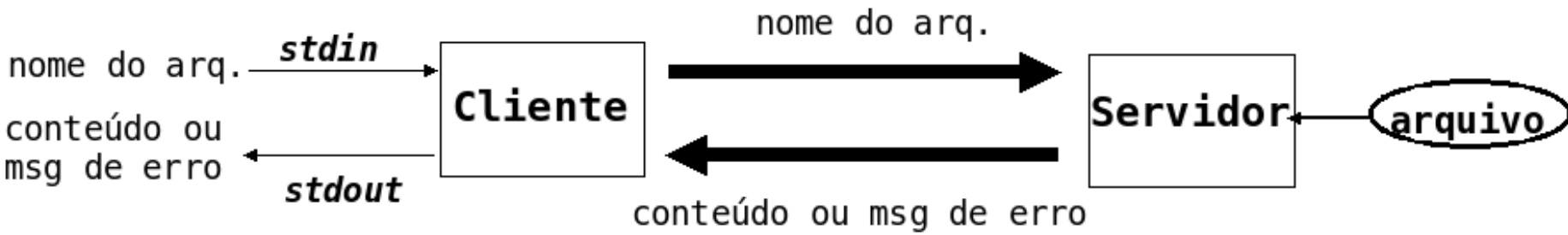
- **Programa distribuído**: conjunto de processos concorrentes que executam em uma rede de computadores
- Normalmente cada processo é um **programa multithreaded** que executa em um único computador

- Os processos em computadores distintos se **comunicam e sincronizam suas ações** trocando mensagens através da rede

Modelos de programação

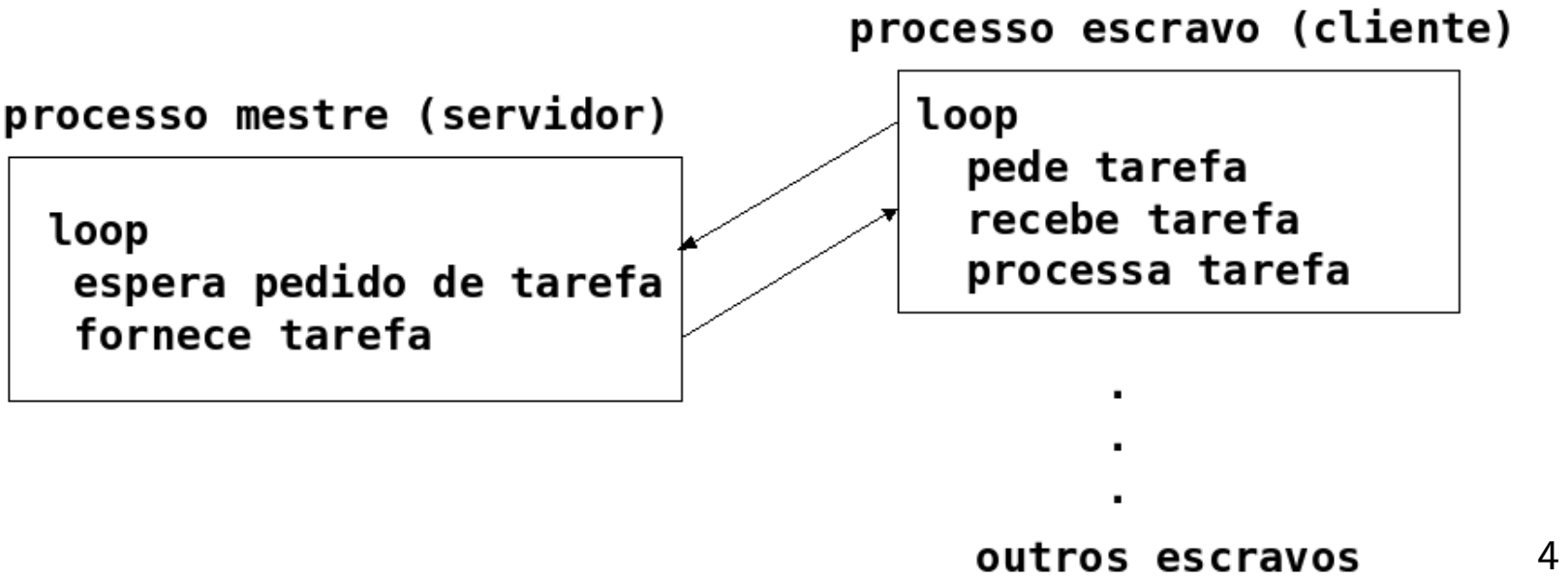
- Necessidade de organizar a comunicação para entender os programas distribuídos
  - (paradigmas, padrões, etc)
- Modelo mais comum: **cliente-servidor**
  - um processo servidor está sempre a espera de comunicação
  - o processo cliente tem a iniciativa de começar a comunicação quando deseja algum serviço

Exemplo de interação cliente/servidor



[3] *Fonte: Notas de aula, profa. Noemi Rodriguez*

Exemplo de interação cliente/servidor



[4]

- Programas paralelos, seguindo o modelo mestre-escravo, onde o mestre detém um *pool* de tarefas, se encaixa bem neste paradigma

[4] *Fonte: Notas de aula, profa. Noemi Rodriguez*

Interface de *sockets*

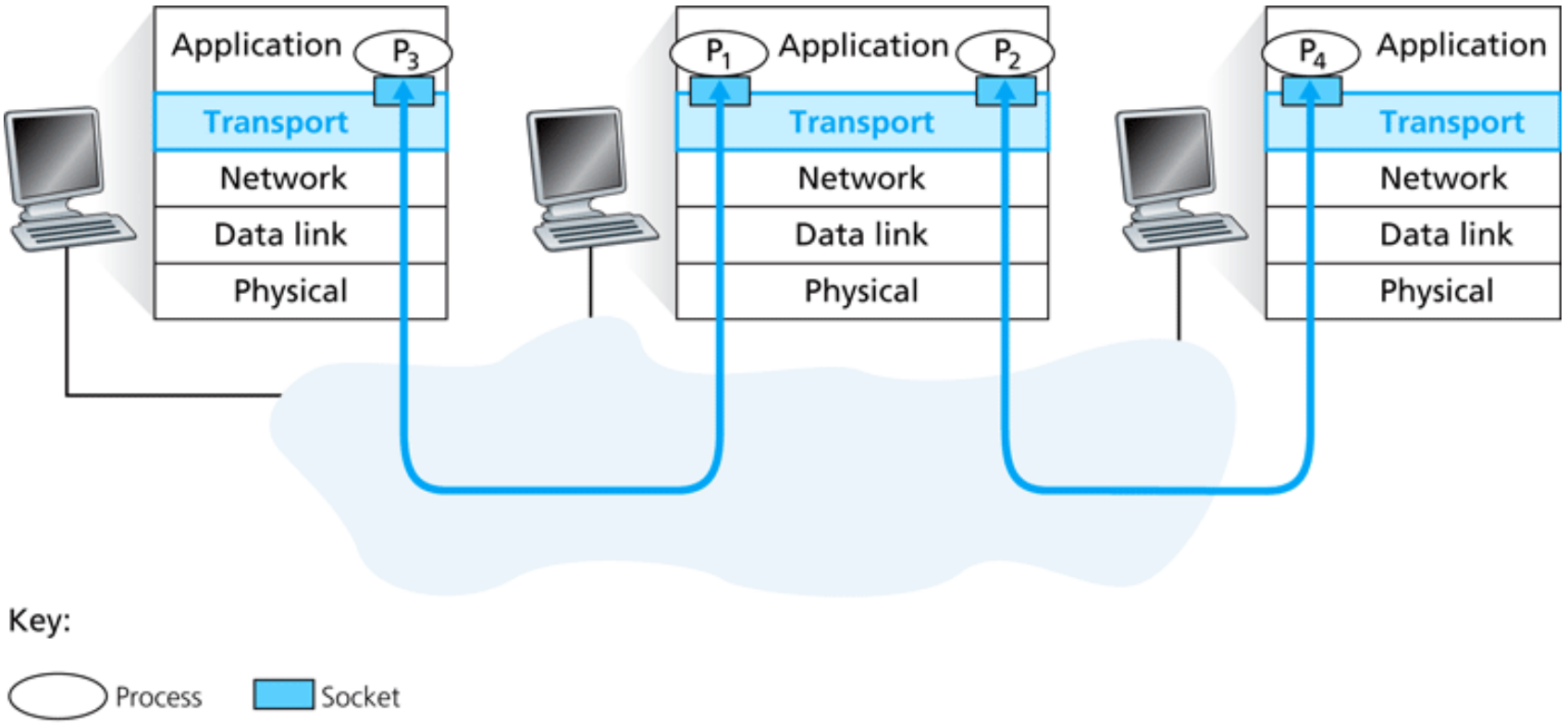


Figure 3.2 ♦ Transport-layer multiplexing and demultiplexing

5

[5] *Fonte: http://www.aw-bc.com/kurose-ross/*

Processamento concorrente no paradigma cliente/servidor

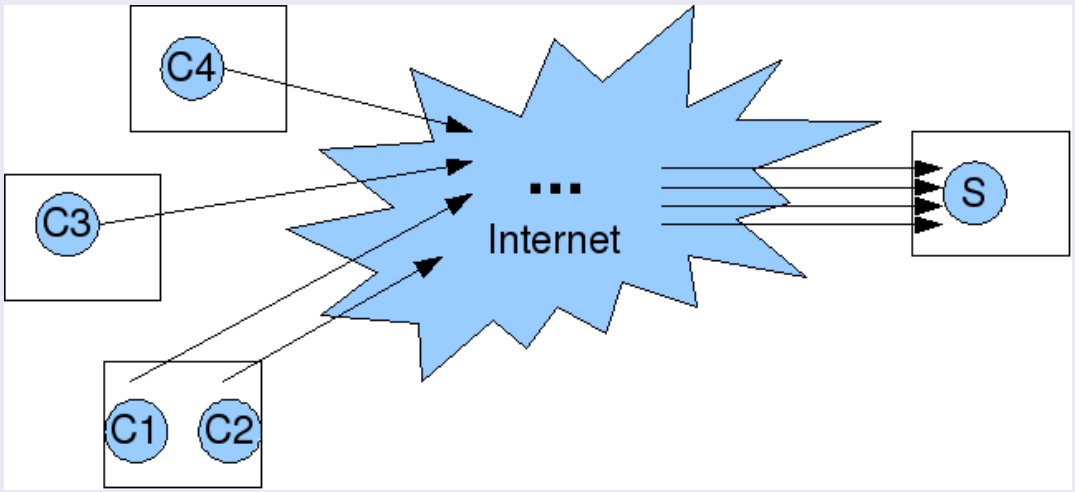
Do lado do **cliente**, a operação concorrente é alcançada facilmente:

- o SO permite que vários usuários executem o programa cliente concorrentemente na mesma máquina; ou
- usuários em diferentes máquinas podem executar o programa cliente simultaneamente

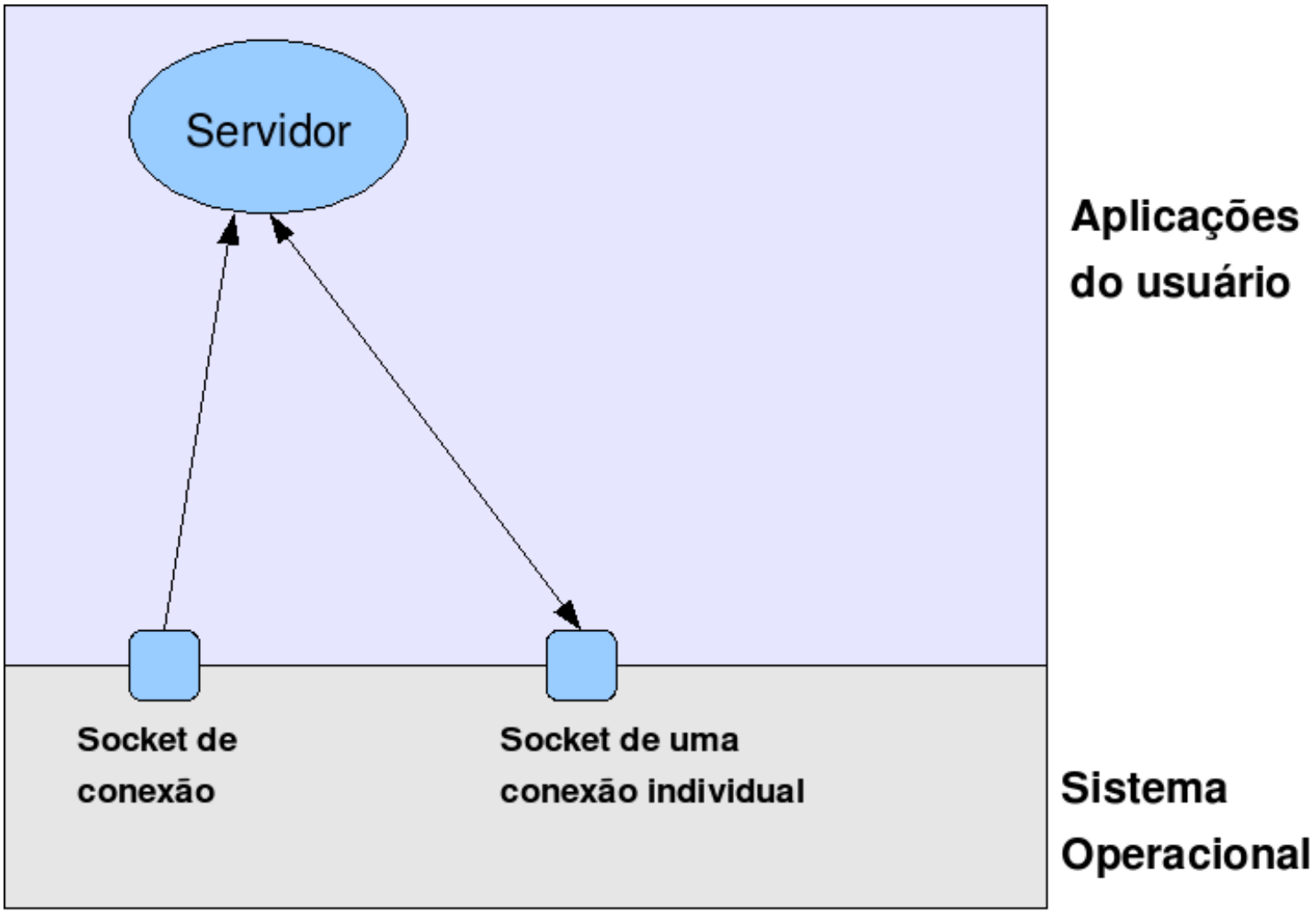
Um programa cliente individual opera como um programa convencional, ele não precisa gerenciar concorrência explicitamente

Processamento concorrente no paradigma cliente/servidor

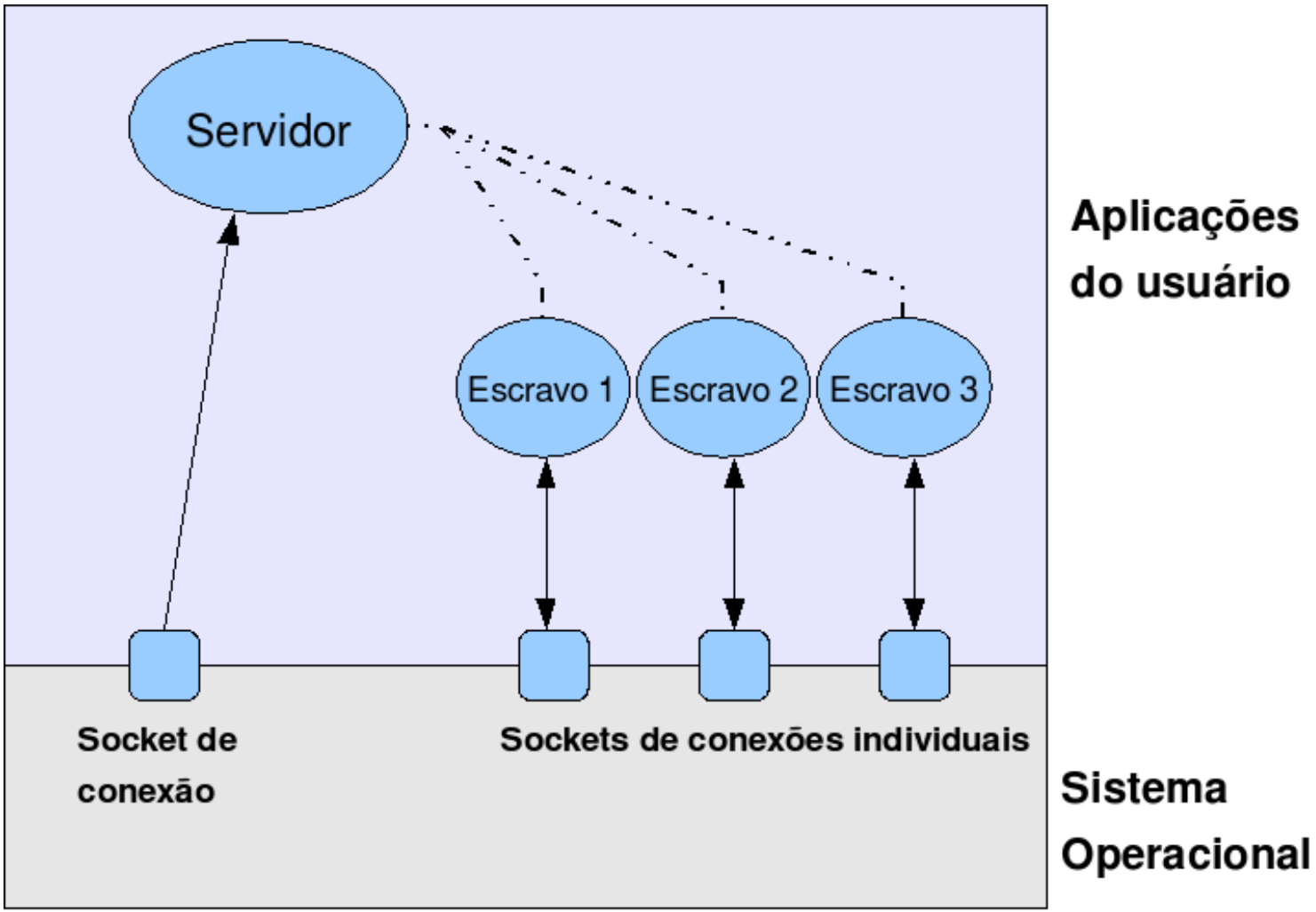
- Do lado do **servidor é preciso tratar as requisições recebidas concorrentemente**
- O software do processo servidor deve ser explicitamente programado para tratar requisições concorrentes, uma vez que vários clientes podem contactar o servidor



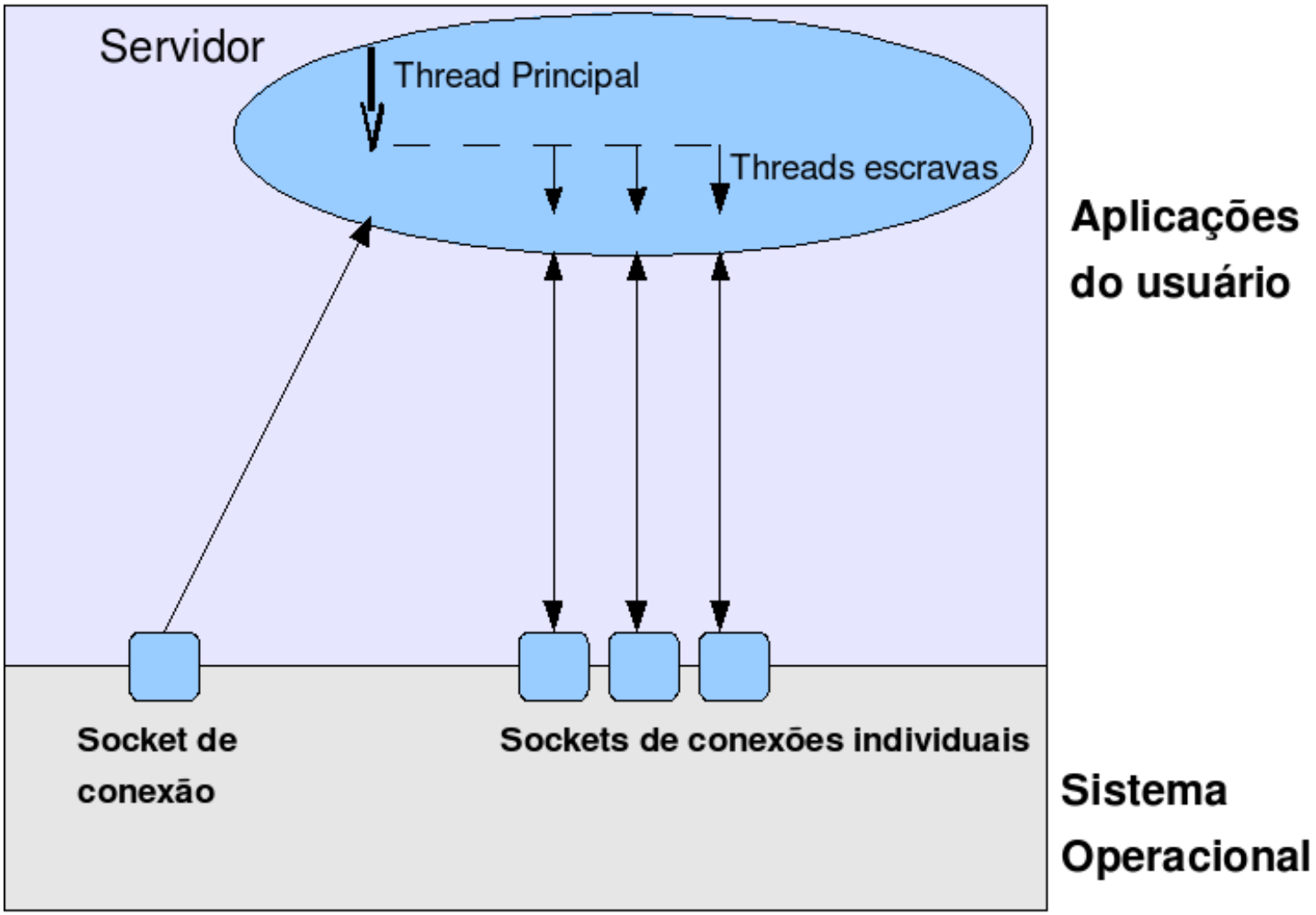
Servidor TCP iterativo



Servidor TCP concorrente multiprocesso



Servidor TCP concorrente multithreading



Referências bibliográficas

1. *Programming Language Pragmatics*, Scott, Morgan-Kaufmann, ed. 2, 2006
2. *Operating Systems – Internals and Design Principles*, Stallings, Pearson, ed. 6, 2009
3. *Modern Multithreading*, Carver e Tai, Wiley, 2006