

高级操作系统

Advanced Operating System

Distributed Systems

Concepts and design

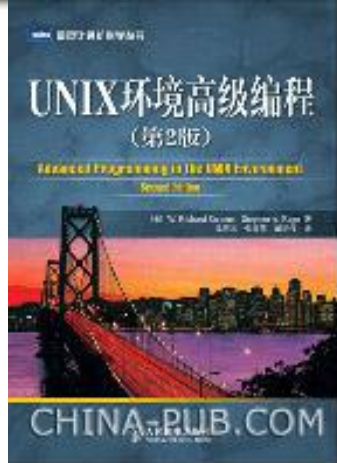
朱青 Qing Zhu,

Department of Computer Science, Renmin University of China

zqruc2012@aliyun.com

Chapter 3

Interprocess Communication 进程间的通信



Teacher: Qing Zhu 朱 青

Department of Computer Science,
Information School,
Renmin University of China
zqruc2012@aliyun.com

Textbook



- ⌘ **Distributed Systems – Concepts and Design, George Coulouris, Jean Dollimore, and Tim Kindberg Edition 3,4, © Pearson Education 2001, 2005**
- ⌘ **The lecture is based on this textbook.**



Chapter 3 进程间的通信

- 3.0 进程及消息传递
- 3.1 进程间的通信
- 3.2 客户/服务器计算模式
- 3.3 Socket编程基础
- 3.4 应用实例

3.0 进程及消息传递

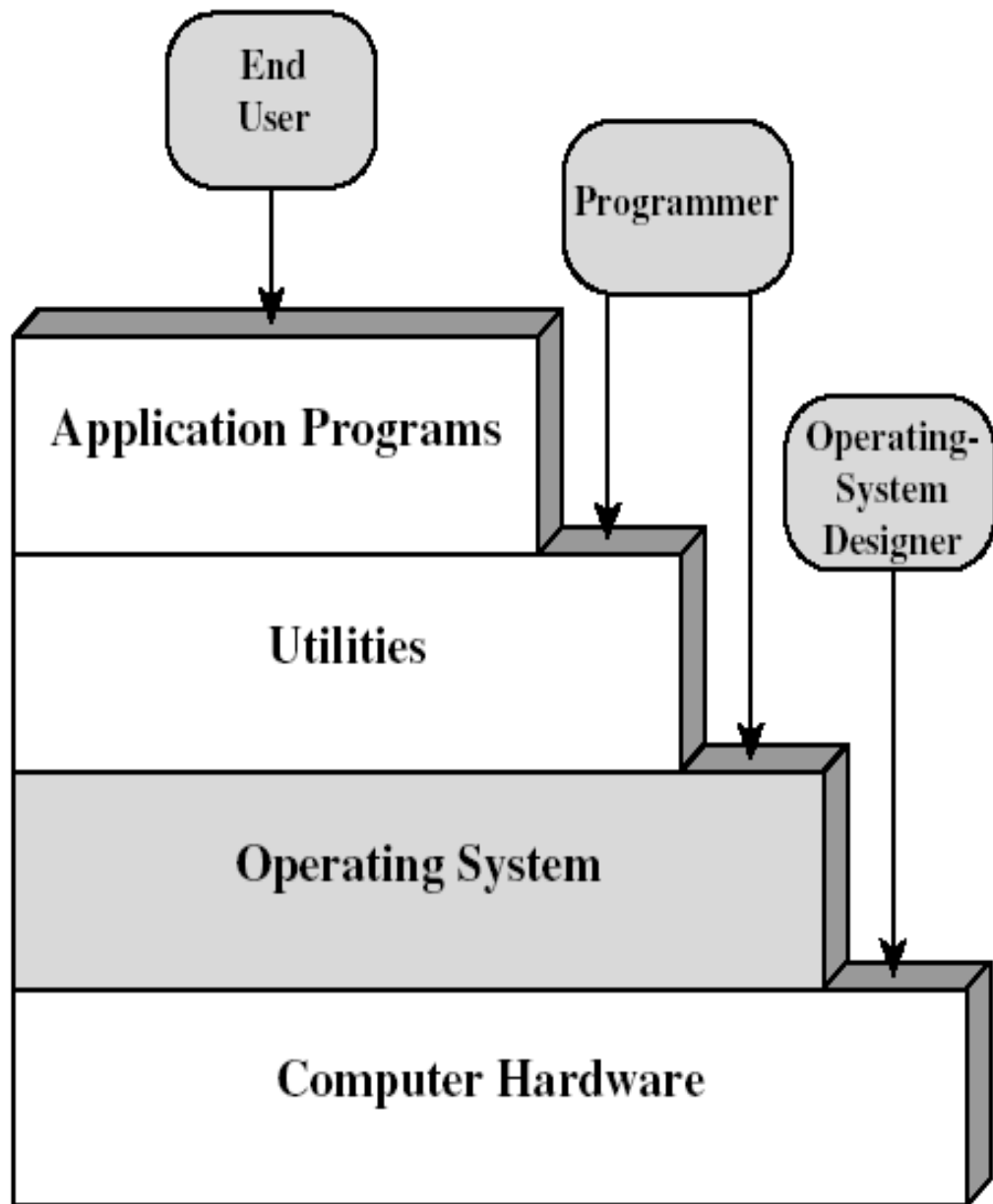
- ⌘ 进程概念
- ⌘ 进程状态转换
- ⌘ 消息传递
- ⌘ 进程通信经典问题

操作系统简介

- ⌘ 什么是操作系统（operating system，——OS）？
- ⌘ 是计算机系统中的—个系统软件，—些程序模块的集合，是配置在计算机上的第—层软件。
- ⌘ 能以有效、合理的方式组织和管理计算机的软硬件资源，合理的组织计算机的工作流程，控制程序的执行并向用户提供各种服务功能，
- ⌘ 使得用户能够灵活、方便、有效的使用计算机，使整个计算机系统能高效地运行。

什么是操作系统

- ⌘ 控制应用程序执行的程序
- ⌘ 充当应用程序和计算机硬件之间接口
- ⌘ 设计目标：
 - ✓ 方便：从用户的观点来看
 - ✓ 有效：从系统管理人员的观点来看
 - ✓ 发展：从发展的观点看



操作系统做什么？

A graphic of a scroll with a light gray background and a dark gray border. The scroll is partially unrolled, showing C code. The code is written in a bold, black, monospaced font. The code is as follows:

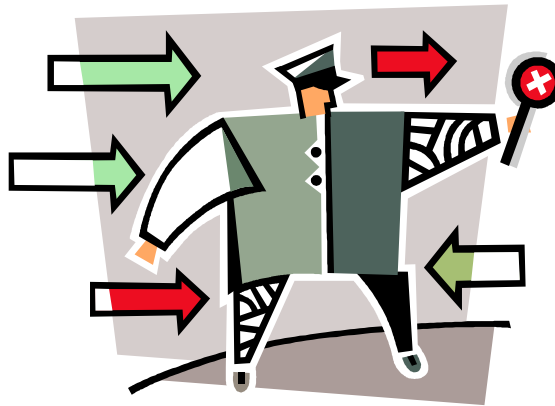
```
#include <stdio.h>
int main(int argc, char *argv[])
{
    puts("hello world");
    return 0;
}
```


操作系统提供一系列服务:

- ⌘ 多任务,
- ⌘ 内存保护, 内存管理,
- ⌘ 文件存取,
- ⌘ 设备控制,
- ⌘ 用户界面
- ⌘ 错误检测及恢复系统
- ⌘ 硬件控制
- ⌘ 安全

Operating System

- ⌘ Exploits the hardware resources of one or more processors to provide a set of services to system users
- ⌘ Manages the processor, secondary memory and I/O devices



Basic Elements

⌘ The processor (CPU)

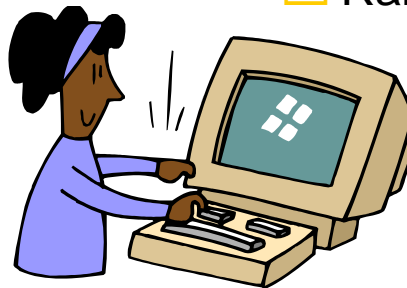
- ☒ Controls the actions of the computer
- ☒ Executes data-processing & logic operations

⌘ The memory

- ☒ Is volatile – contents are lost when powered off
- ☒ Random access

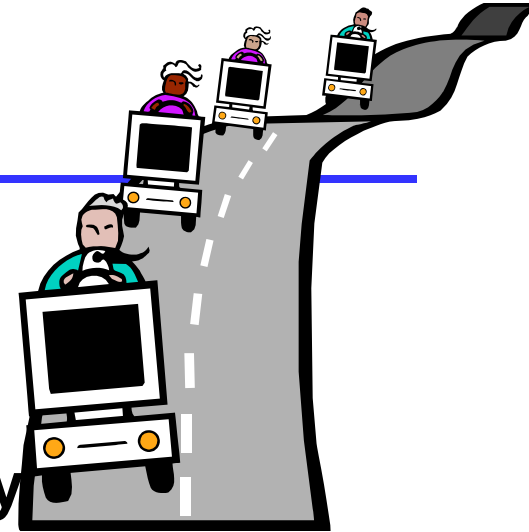
⌘ I/O modules

- ☒ Move data between memory and external devices (storage, communication hardware, terminals, ...)



⌘ The system bus

- ☒ Provides communication among processors, main memory, and I/O modules



进程的概念

- 进程（**Process**）的定义：
- 进程是具有独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的独立单位。

进程的概念

- 进程（**Process**）的定义：
- 进程是具有独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的独立单位。

进程控制块 (Process Control Block)

- 进程控制块 (PCB) 概念：
- 系统为了管理进程设置的一个专门的数据结构，用它来记录进程的外部特征，描述进程的运动变化过程
- 系统利用PCB来控制和管理进程，所以PCB是系统感知进程存在的唯一标志
- 进程与PCB是一一对应的

程序与进程之间的区别：

- 进程更能真实地描述并发，而程序不能
- 进程是由程序和数据+PCB组成的
- 程序是静态的，进程是动态的
- 进程有生命周期，有诞生有消亡，短暂的；而程序是相对长久的
- 一个程序可对应多个进程，反之亦然
- 进程具有创建其他进程的功能，而程序没有

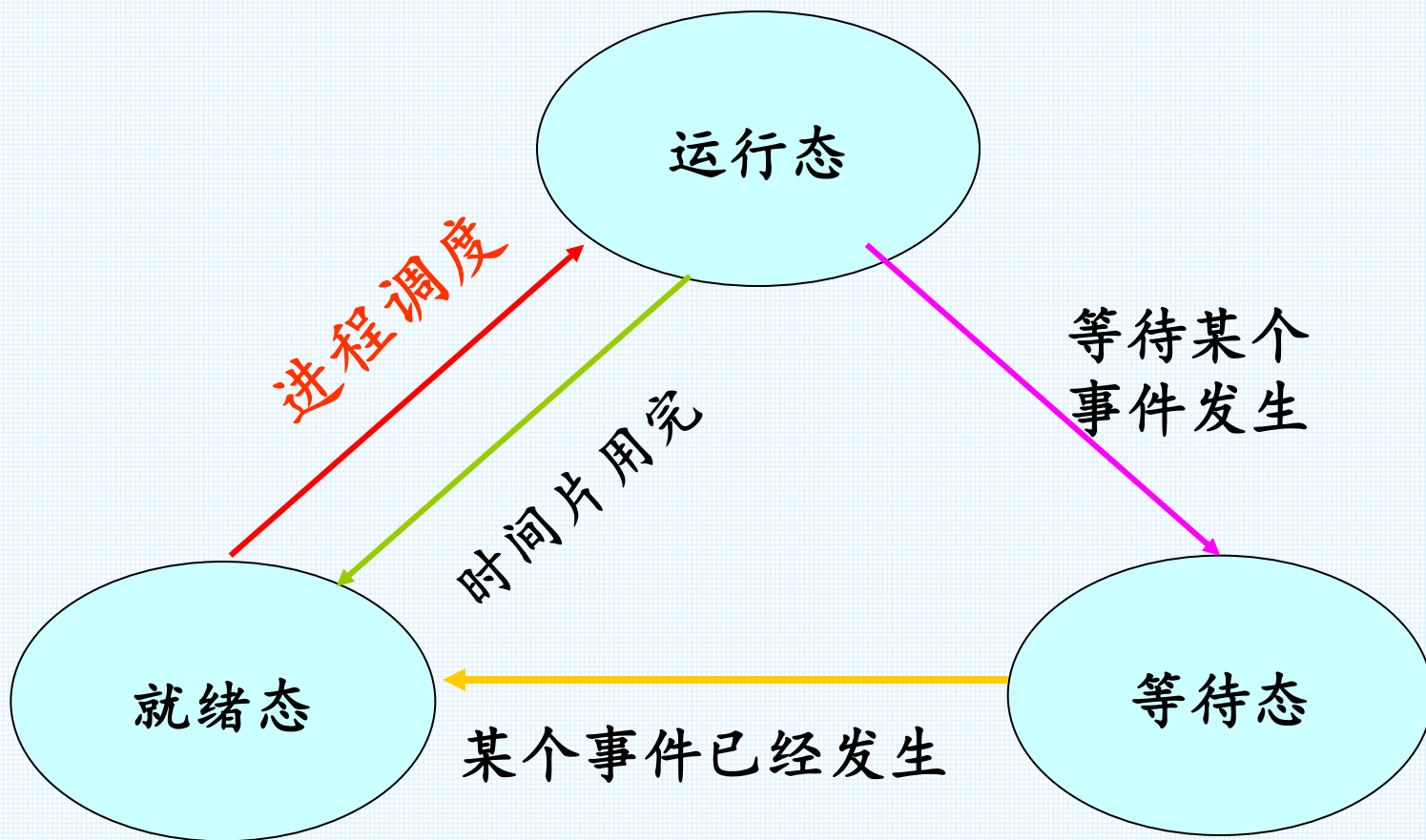
进程的基本状态及其转换

- 进程的三种基本状态：
- 进程在生命消亡前处于且仅处于三种基本状态之一
- 不同系统设置的进程状态数目不同

进程三个基本状态转换图：

就绪态与等待态的区别？

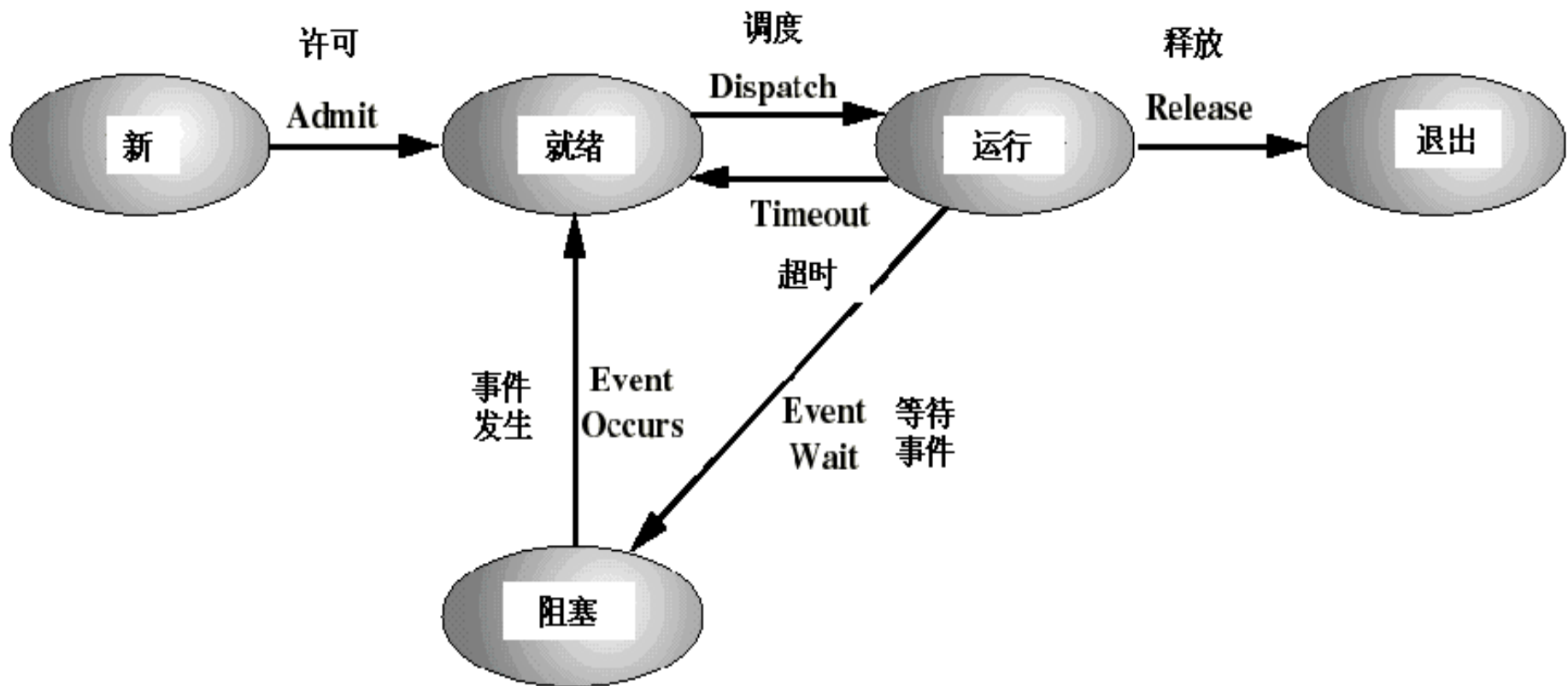
就绪状态：进程已获得除处理机外的所需资源，等待分配处理机资源；只要分配到CPU就可执行。在某一时刻，可能有若干个进程处于该状态



进程三个基本状态

- 运行态（Running）：
 - 进程占有CPU，并在CPU上运行
- 就绪态（Ready）：
 - 一个进程已经具备运行条件，但由于无CPU暂时不能运行的状态（当调度给其CPU时，立即可以运行）
- 等待态（Blocked）：阻塞态、挂起态、封锁态
 - 冻结态、睡眠态
- 指进程因等待某种事件的发生而暂时不能运行的状态（即使CPU空闲，该进程也不可运行）

五状态进程模型



⌘ 准备退出：父进程可中止子进程

消息传递

⌘ 进程交互的两个基本要求

- ☑ 同步：互斥进程间需同步。同步指对进程执行时序的约束，包括互斥和时序的先后限制
- ☑ 通信：合作进程间交换信息

⌘ 消息传递

- ☑ 进程通信的一种常用方法
- ☑ 适用范围广，可用于多核、**SMP**和分布式系统
- ☑ 由原语对“**send**（发送）和**receive**（接收）”提供主要功能：
 - ☒ **send**(目的, 信息)
 - ☒ **receive**(源, 信息)
- ☑ 实现形式有多种

Message Passing

Synchronization	Format
Send	Content
blocking	Length
nonblocking	fixed
Receive	variable
blocking	Queuing Discipline
nonblocking	FIFO
test for arrival	Priority
Addressing	
Direct	
send	
receive	
explicit	
implicit	
Indirect	
static	
dynamic	
ownership	

Table 5.5 Design Characteristics of Message Systems for Interprocess Communication and Synchronization

消息传递的同步

- 消息传递自然地隐含了同步
 - 只有当一个进程发送消息之后，接收者才能收到消息
- 调用send原语的两种可能结果
 - 发送者进程被阻塞，直到这个消息被接收
 - 发送者进程不被阻塞
- 调用receive原语的两种可能结果
 - 接收者进程接收消息时，消息已发出，接收者不阻塞
 - 接收者进程接收消息时，消息未发出，接收者被阻塞，直到发送者进程发出此消息

三种组合方式

- 消息传递实现的三种常用组合方式
 - “阻塞send, 阻塞receive”方式, 即“会合”原则, 适用于进程间的紧密同步 (如打电话)
 - “无阻塞send, 阻塞receive”方式, 最有用的组合 (如服务器提供资源, 收发短信)
 - “无阻塞send, 无阻塞receive”方式, 不要求任何一方等待。危险: 信息可能会丢失 (如贴/看小广告)

三种组合方式

- “无阻塞send”是最自然的选择。但错误可能会导致进程重复传递消息，消耗系统资源。且必须使用应答消息以证实收到消息
- “阻塞receive”是常用的选择。但若消息丢失或发送者进程失效，会导致接收者进程被长期阻塞

消息传递的寻址

- send原语中指明接收者是必要的
- receive原语有时也指明发送者
- 直接寻址方案
 - 发送者在发送时给出了接收者进程的具体标识号，如进程ID



消息传递的寻址

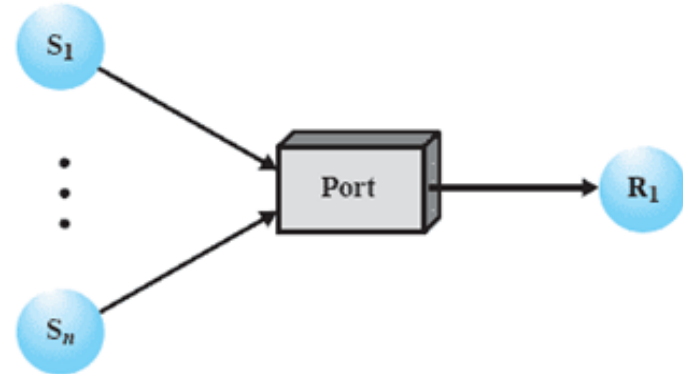
■ 间接寻址方案

- 发送者将消息发送到共享的信箱中临时保管，接收者从信箱中获得这些消息
- 耦合方式：“一对一”、“多对一”、“一对多”或“多对多”
- 进程与信箱的关联：静态方式（端口）和动态方式（使用连接原语connect和disconnect）

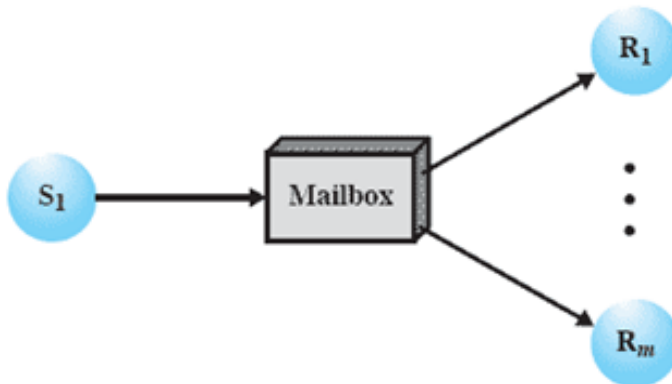
Indirect Process Communication



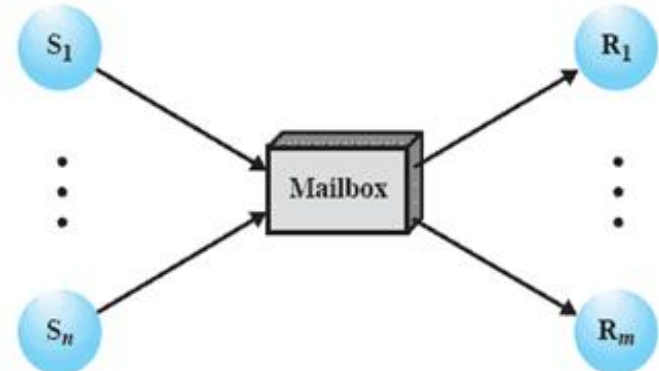
(a) One to one



(b) Many to one



(c) One to many

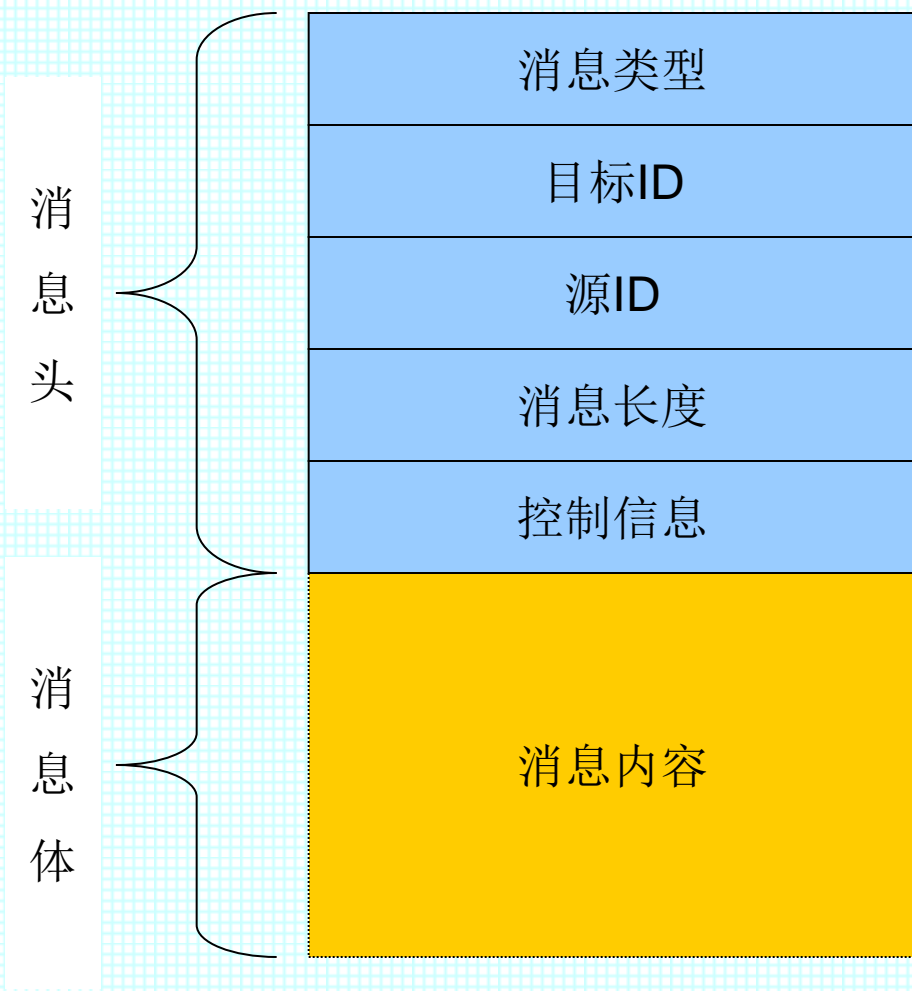


(d) Many to many

消息格式

- 取决于运行环境
 - 单机系统
 - 分布式系统
- 两类格式
 - 定长格式
 - 变长格式
- 消息格式
 - 消息头——消息类型、目标ID、源ID、消息长度、控制信息
 - 消息体——消息内容

排队原则



- 先进先出 (FIFO) 原则
- 优先级原则

消息传递实现通信/同步

■ 同步

- **send**——阻塞、无阻塞
- **receive**——阻塞、无阻塞、测试是否到达

■ 寻址

- **直接: send/receive**——显式、隐式
- **间接**
- **静态**
- **动态**
- **所有权**

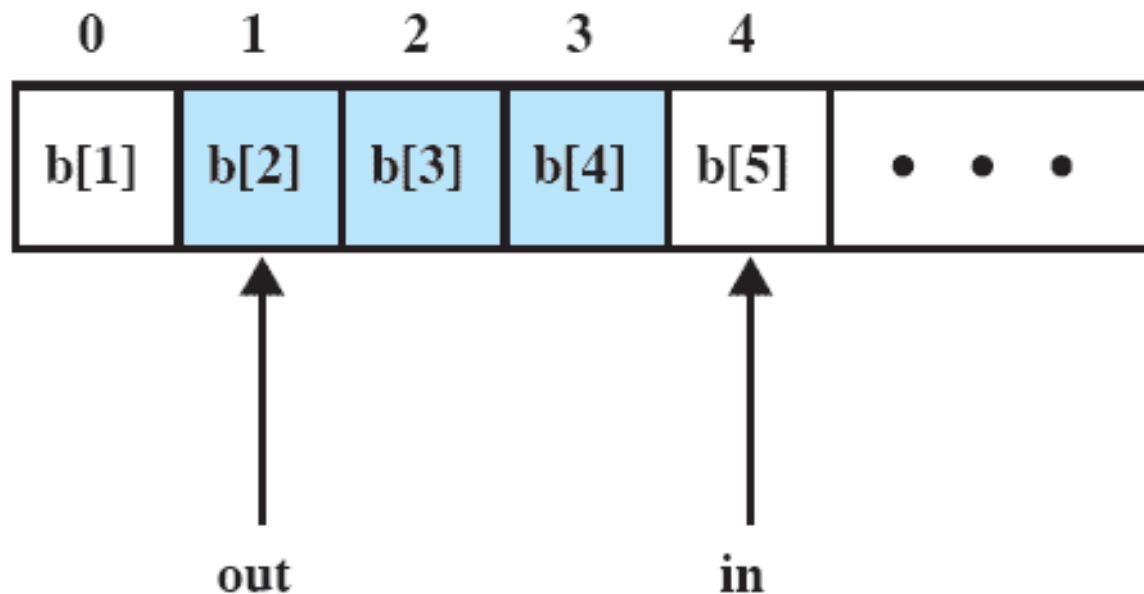
消息传递实现通信/同步

- 格式
 - 内容
 - 长度——固定、可变
- 排队规则
 - **FIFO**
 - 优先级

消息传递实现互斥

- 使用“无阻塞send，阻塞receive”组合
- 一组进程共享一个信箱box，信箱被初始化为一条无内容的空消息（代表进入临界区的钥匙）
- 每个进程在进入临界区前，首先尝试接收消息，离开临界区时将接收到的消息放回信箱
- 每次只有接收到消息的那个进程才可以进入临界区（互斥）
- 消息函数似在进程间传递一个可使用临界区的令牌

Buffer Structure



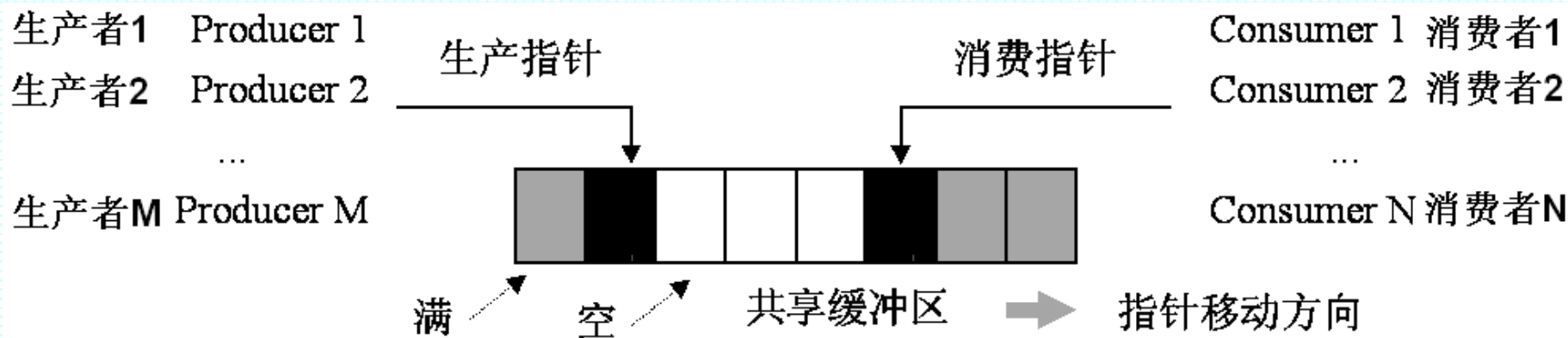
Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

进程通信经典问题

■ 问题描述（生产者/消费者问题）

- 若干进程通过无限/**有限的共享缓冲区**交换数据
 - 一组“**生产者**”进程不断写入
 - 另一组“**消费者**”进程不断读出
 - 共享缓冲区无限/**共有N个**
 - **任何时刻只能有一个进程可对共享缓冲区进行操作**



生产者/消费者问题

```
semaphore n=0; /*缓冲区中  
的产品数*/
```

```
semaphore s=1; /*互斥*/
```

```
void producer() {
```

```
    while (true) {
```

```
        produce();
```

```
        semWait(s);
```

```
        append();
```

```
        semSignal(s);
```

```
        semSignal(n);
```

```
    }
```

```
}
```

```
void consumer() {
```

```
    while (true) {
```

```
        semWait(n);
```

```
        semWait(s);
```

```
        take();
```

```
        semSignal(s);
```

```
        consume();
```

```
    }
```

```
}
```

```
void main() {
```

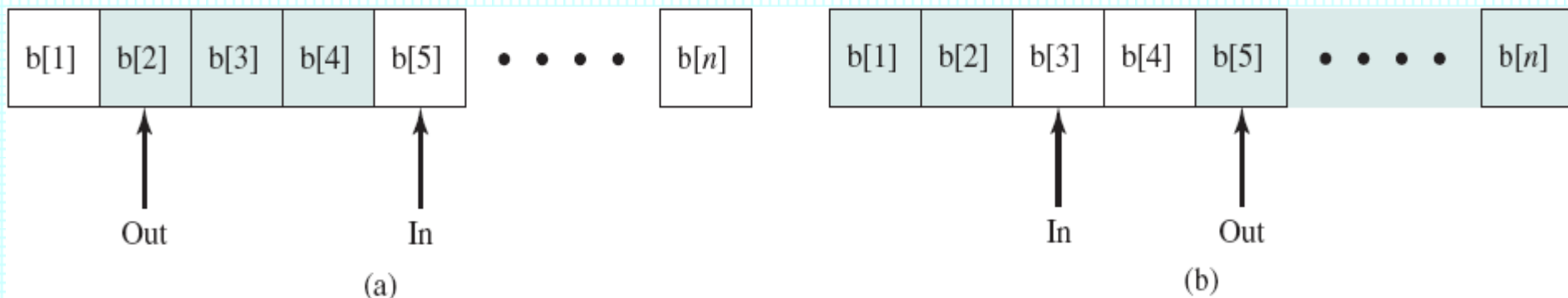
```
    parbegin(producer,
```

```
            consumer);
```

```
}
```

有限缓冲区的生产者/消费者问题

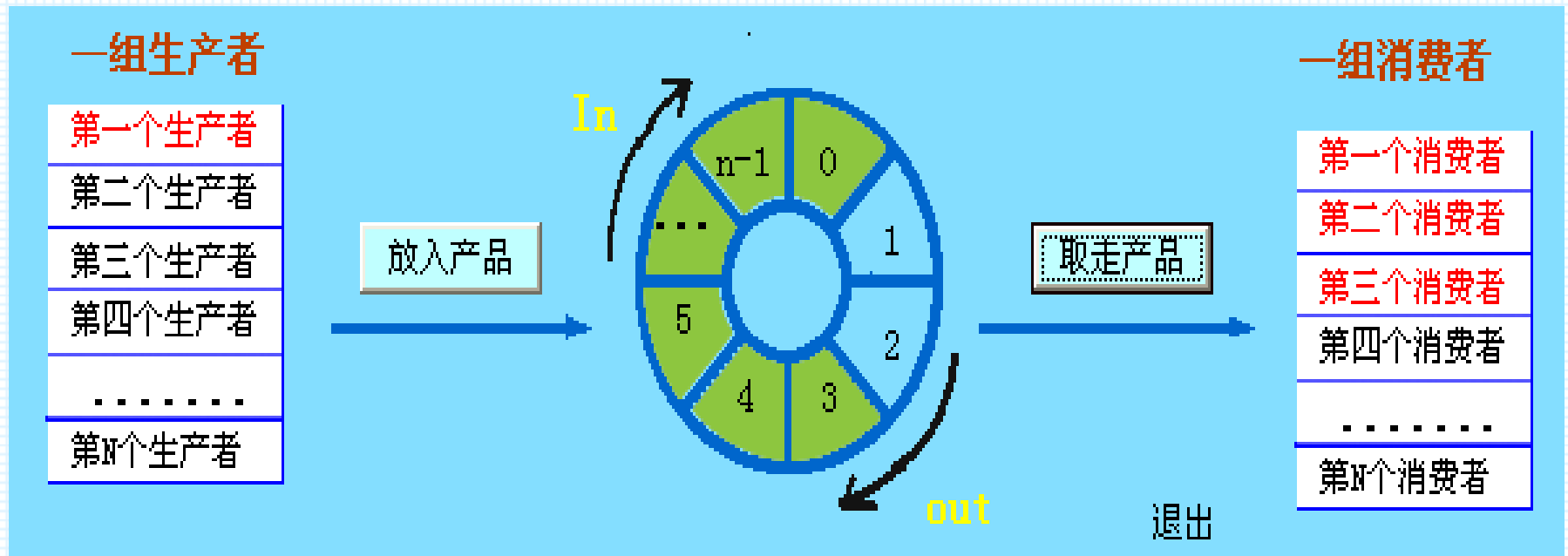
对象	被阻塞事件	解除阻塞事件
生产者	插入满缓冲区	消费者移出一项
消费者	从空缓冲区移出	生产者插入一项



Finite Circular Buffer for the Producer/Consumer Problem

生产者/消费者问题的有限循环缓冲区

生产者—消费者问题



N个循环缓冲区

消息传递实现互斥

```
/* program mutualexclusion */
```

```
const int n=/*进程数*/;
```

```
void P(int i) { /*进程i*/
```

```
    message msg;
```

```
    while(true) {
```

```
        receive(box, msg); /*信箱空时被阻塞*/
```

```
        /*临界区*/;
```

```
        send(box, msg);
```

```
        /*其余部分*/; }
```

```
}
```

```
void main() {  
    create_mailbox(box);  
    send(box, null);  
    parbegin(P(1), ..., P(n));  
}
```



消息传递实现生产者-消费者问题

/* 互斥程序 */

const int capacity=/*缓冲区容量*/;

message null=/*空消息*/;

void producer() {

message pmsg;

while(true) {

receive(mayproduce, pmsg);

pmsg=produce();

send(mayconsume, pmsg);

}

}



消息传递实现生产者-消费者问题

```
/* 互斥程序 */  
const int capacity=/*缓冲区容量*/;  
message null=/*空消息*/;  
void comsumer() {  
    message cmsg;  
    while(true) {  
        receive(mayconsume, cmsg);  
        consume(cmsg);  
        send(mayproduce, null);  
    }  
}
```


消息传递实现生产者-消费者问题

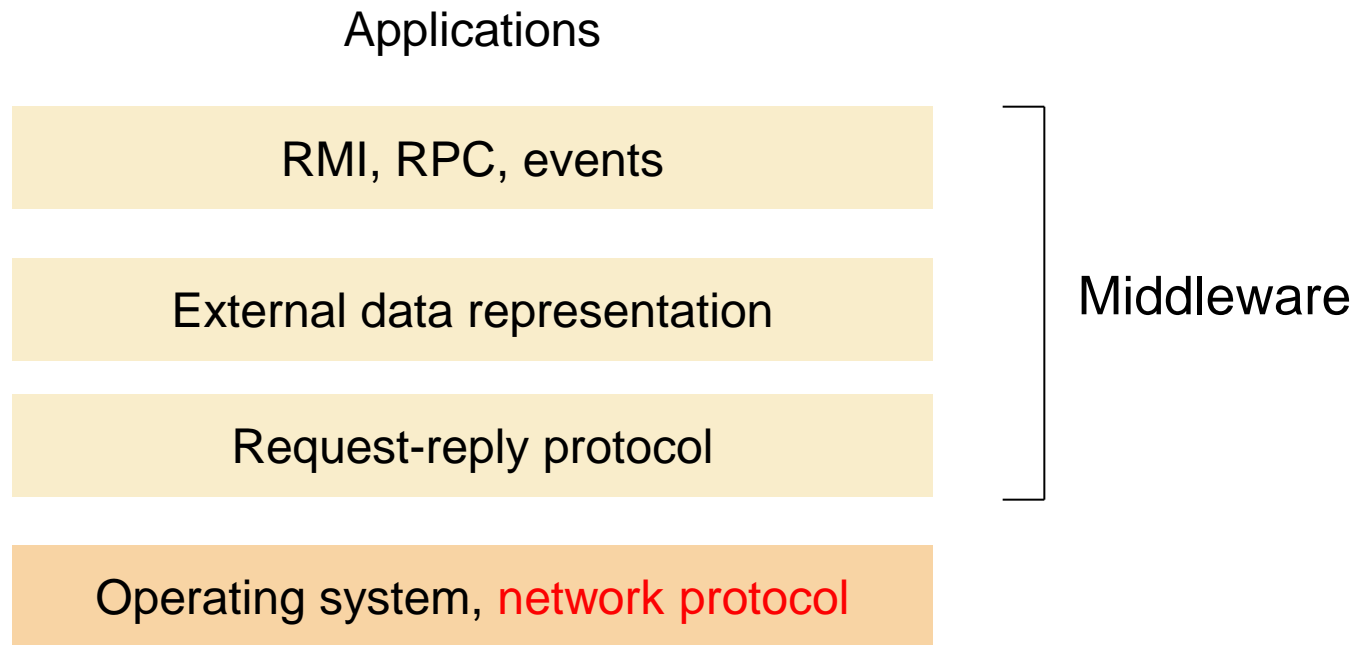
```
void main( )
{
    create_mailbox(mayproduce);
    create_mailbox(mayconsume);
    for(int i=1; i<capacity; i++)
        send(mayproduce, null);
    parbegin(producer(), consumer());
} /*初始时: mayconsume信箱为空,
    mayproduce信箱则被null信号填满*/
```



Chapter 3 进程间的通信

- 3.0 进程及其状态转换
- 3.1 进程间的通信
- 3.2 客户/服务器计算模式
- 3.3 Socket编程基础
- 3.4 应用实例

3.1 进程间的通信



RMI(remote method invocation), RPC (remote procedure call)

Interprocess Communication(进程间的通信)

Point-to-point message passing is typically based on two message communication operations (subroutines).在点对点进程间进行消息传递需要两个消息通信操作支持

⌘ **Send:** A process copies a message (a sequence of bytes) into the (incoming) buffer of a remote system.

⌘ 送：进程复制消息到远程系统的缓冲区

⌘ **Receive:** A process copies a message from the (incoming) buffer of its system into its local memory.

⌘ 接收：进程复制消息从远程系统的缓冲区到本地内存。

Typically, **Receive** must be called before the corresponding **Send**. 在发送之前，相应的接收需被调用

☒ Unbuffered communication: If a receive has not been called before the message arrives it is lost!

☒ 非缓冲通信：在消息到达前接收者没有被调用，消息将丢失。

Alternatively, some incoming messages are kept temporarily in a buffer (mailbox) assuming that a suitable receive will be called „soon“. 到达的消息 被临时保存在缓冲区

☒ Otherwise there is a time-out or a buffer overflow. 另外，超时或缓冲区溢出，

☒ Failure model 错误模型

Communication Types

- ⌘ **Synchronous Communication:** The sending and receiving processes synchronize each other at every message.
- ⌘ **同步通信:** 发送进程和接收进程 在每个消息上同步。
 - ☑ The Send and the Receive operations are blocking.
 - ☑ 发送和接收操作被阻塞。
 - ☑ The Send process is blocked until the corresponding Receive is issued.
 - ☑ 发送进程阻塞直到发生了相应的接收事件。
 - ☑ The Receive process is blocked until a message arrives.
 - 接收进程阻塞直到有消息到达。

⌘ Asynchronous Communication: 异步通信

☑ The sending process is allowed to proceed as soon as the message has been sent to the outgoing (local) buffer.

☑ 只要消息被送到本地缓冲区，发送消息允许执行。

☑ The Send operation is **non-blocking**.

☑ 发送操作是非阻塞的。

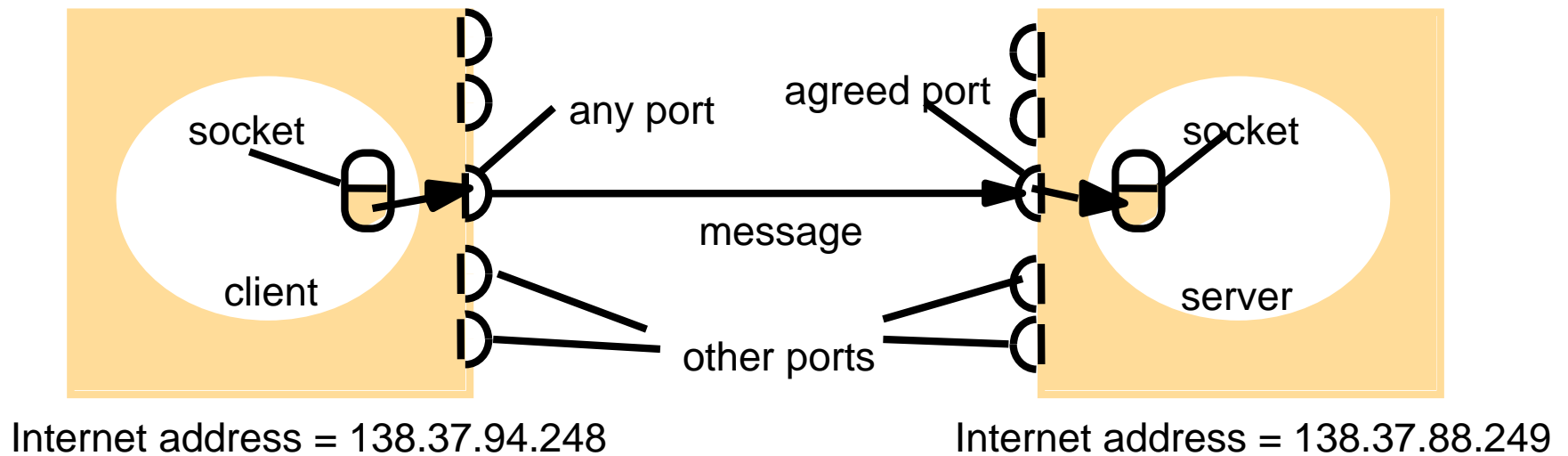
☑ The Receive operation may be either **blocking** or **non-blocking**.

接收操作可以有阻塞和非阻塞两种。

Interprocess Communication

- ⌘ Message Destination: (Internet address, local port)
- ⌘ 消息目的地：（因特网地址，本地端口）。
- ⌘ Reliability: reliable communication in terms of validity and integrity.
- ⌘ 可靠性：以有效性和完整性定义了可靠通信。
- ⌘ Ordering: Some applications require that messages be delivered in sender order. The delivery of messages out of sender order in which is regarded as a failure by such application.
- ⌘ 排序：有些应用要求消息按发送方顺序发送，否则，被认为发送失败。

Figure 4.2
Sockets and ports



Sockets

- ⌘ Sockets are endpoints for communication between process.
- ⌘ Sockets是进程间通信的端点。
 - ☑ They originate from UNIX but are also found in other OS.
 - ☑ 套接字源于UNIX，也在其他OS中出现。
 - ☑ They are used by UDP and TCP。
 - ☑ 它被用于UDP and TCP
 - ☒ Each socket is associated with a particular protocol.
 - ☒ 每个套接字对应特殊的协议。

-
- ⌘ Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process.
 - ⌘ 进程间通信是在一个进程的一个套接字与另一个进程的一个套接字之间传递一个消息。
 - ☑ A socket must be bound to a local port and an Internet address of the computer on which it runs.
 - ☑ 套接字必须绑定到计算机的一个Internet地址和端口号
 - ⌘ A process may use multiple port numbers for receiving messages.
 - ⌘ 进程使用多端口接收消息。
 - ⌘ Ports cannot be shared by different processes.不同进程不能共享端口号。
 - ☑ Exception: IP multicasts (UDP) IP组播的进程除外。

UDP Datagram Communication UDP数据报通信

- ⌘ UDP is **unreliable** and does not support acknowledgements and retries.
- ⌘ UDP是不具有可靠性，不支持确认和重发
 - ☒ If a message does not arrive, a failure occurs.
 - ☒ 如果消息无法到达，故障发生。
- ⌘ Most environments support a message size of 8kB.
- ⌘ 大多数环境支持的消息包大小为8kB
 - ☒ Larger messages must be fragmented.
 - ☒ 大消息必须被分段
- ⌘ The receiving process needs to specify an array for the message.
- ⌘ 接收消息需要列入消息数组
 - ☒ The message is truncated it is too long.
 - ☒ 太长的消息需要被截断。

⌘ UDP uses non-blocking sends and blocking receives.

⌘ UDP数据报使用非阻塞send和阻塞receive

☒ A message is placed in a queue for the socket and can be collected by a future receive.

☒ 消息被放置在socket队列中，以将来被接收。

☒ Time-outs can be set on sockets.

☒ 套接字上设置超时标志。

☒ The message is discarded if the no socket is bound to the receiving port. 如果套接字不被绑定到接收端消息会丢失。

⌘ A receive gets any message addressed to this socket.

⌘ 接收端接收定位在套接字上的消息。

TCP Stream Communication TCP流通信

- ⌘ TCP is reliable and uses an acknowledgement scheme.
- ⌘ TCP是可靠的并且使用确认机制
 - ☑ Sliding window 滑行窗口
- ⌘ The TCP stream is transmitted as one or more IP packets (implementation dependent).
- ⌘ TCP流作为一个或多个IP包被传输（随执行而定）
- ⌘ TCP tries to match the speed of the processes writing to or reading from a stream: **Flow control.**
- ⌘ TCP协议试图匹配读写流的进程控制：流控制
- ⌘ A sender may be blocked. 送者可能被阻塞

-
- ⌘ Due to message identifiers the recipient can detect and reject duplicates and reorder message if necessary.
 - ⌘ 由于消息标示接收者能探测和拒绝复写如果需要可以重排消息
 - ⌘ A pair of communicating process must establish a connection before using a stream.
 - ⌘ 使用流之前，一对消息进程必须建立连接
 - ☒ Once the connection exists no Internet addresses and ports are necessary to write to or read from the stream.
 - ☒ 一旦连接建立，对于流的读写，Internet 地址和端口可以不需要
 - ☒ The processes must agree on the content of the message.
 - ☒ 进程必须确认消息的内容。

Failure Models 故障模型 for UDP and TCP

- ⌘ UDP uses a checksum to ensure that a message is not corrupted.
- ⌘ UDP使用校验确保消息不再破坏
 - ☑ Some messages may get lost (omission failures).
 - ☑ 一些消息可能被丢失（冗长失效）
 - ☑ Messages may arrive out of sender order.
 - ☑ 消息可以不按发送的顺序
- ⌘ TCP uses checksums to detect and reject corrupt packets.
- ⌘ TCP 使用校验诊断或拒绝被破坏的包。
- ⌘ TCP uses sequence numbers to detect and reject duplicate packets. TCP 使用序列号诊断或拒绝重复的包。

⌘ TCP uses timeouts and retransmissions to deal with lost packets. TCP使用超时设定或转发去处理丢包

☒ After a long time without acknowledgement TCP will declare the connection to be broken.

☒ 长时间没有被确认的TCP将宣布连接中断

☒ The processes cannot distinguish between network failure and failure of the process at the other end of the connection.

☒ 进程不能区分网络故障或连接端的进程故障

☒ In this case a process cannot tell whether its most recent messages have been received.

☒ 在这种情况下进程无法告诉是否最近的消息被接收。

Figure 4.1
Middleware layers

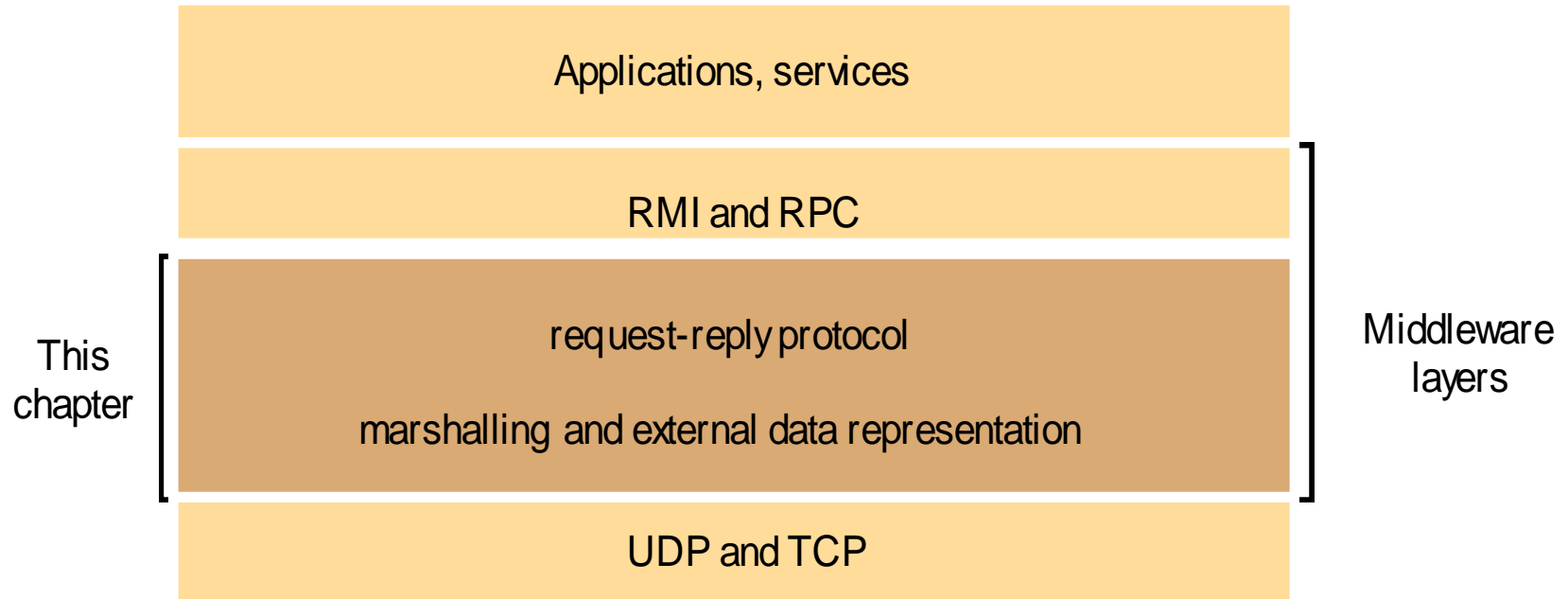


Figure 4.3

UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        // args give message contents and server hostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request = new DatagramPacket
                (m, args[0].length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        }catch (IOException e){System.out.println("IO: " + e.getMessage());}
    }finally {if(aSocket != null) aSocket.close();}
}
```

Figure 4.4

UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
    }finally {if(aSocket != null) aSocket.close();}
}
```

Figure 4.5

TCP client makes connection to server, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =
                new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);           // UTF is a string encoding see Sn 4.3
            String data = in.readUTF();
            System.out.println("Received: "+ data) ;
        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());}
        }catch (IOException e){System.out.println("IO:"+e.getMessage());}
    }finally {if(s!=null) try {s.close();}catch (IOException
e){System.out.println("close:"+e.getMessage());}}
    }
}
```

Figure 4.6 TCP server makes a connection for each client and then echoes the client's request

```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
}
```

// this figure continues on the next slide

Figure 4.6 continued


```
class Connection extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public Connection (Socket aClientSocket) {
                try {
                        clientSocket = aClientSocket;
                        in = new DataInputStream( clientSocket.getInputStream());
                        out =new DataOutputStream( clientSocket.getOutputStream());
                        this.start();
                    } catch(IOException e) {System.out.println("Connection:"+e.getMessage());}
            }
        public void run(){
                try {
                        // an echo server
                        String data = in.readUTF();
                        out.writeUTF(data);
                    } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());}
                    } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
                    } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
            }
    }
```

Figure 4.21

Sockets used for datagrams 用于数据包的套接字

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress)
•
•
sendto(s, "message", ServerAddress)
```



Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress)
•
•
amount = recvfrom(s, buffer, from)
```

ServerAddress and *ClientAddress* are socket addresses

Figure 4.22

Sockets used for streams 使用流的套接字

Requesting a connection

```
s = socket(AF_INET, SOCK_STREAM, 0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM, 0)
•
bind(s, ServerAddress);
listen(s, 5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

ServerAddress and *ClientAddress* are socket addresses



Chapter 3 进程间的通信

- 3.0 进程及其状态转换
- 3.1 进程间的通信
- 3.2 客户/服务器计算模式
- 3.3 Socket编程基础
- 3.4 应用实例

3.2 客户/服务器计算模式

- 1. 网络计算模式
- (1) 以大型机为中心的计算模式
- (分时共享模式(time-sharing))
- 形式：系统提供专用的用户界面；所有的用户击键和光标位置都被传入主机；
- 通过硬连线把简单的终端连接到主机或一个终端控制器上；
- 从主机返回的所有结果，都显示在屏幕的特定位置；
- 技术特点：利用主机的能力来运行应用，采用无智能的终端来对应用进行控制。



网络计算模式

- (2)以服务器为中心的计算模式
- （资源共享模式 resource-sharing）
- 形式：共享共同的应用、数据，以及打印机；
- 每个应用提供自己的用户界面，并对界面给与全面的控制；
- 所有的用户查询或命令处理都在工作站方完成。
- 技术特点：利用工作站的能力来运行所有的应用，用服务器的能力来作为外设的延伸，如硬盘、打印机等。

网络计算模式

- (3)小型化和客户机/服务器计算模式
- 形式：应用被分为前端 (front-end)客户部分，后端 (back-end)服务器部分；
- 客户运行在微机或工作在上；服务器部分运行在从微机到大型机等各种计算机上。
- 客户机和服务器工作在不同的逻辑实体中，协调工作。
- 通信时，客户即可索取信息，它依赖于服务器执行客户方不能完成或不能有效完成的工作。
- 服务器方随时等待客户机提交申请的信息，进行信息交互。
- 技术特点：系统使用客户集合服务器两方面的智能、资源和计算机能力来执行一个特定的任务。



C/S模式

- (1) 客户机的特点：
 - A) 客户机提供用户界面 (UI或GUI), 完成用户的输入输出。
 - B) 一个C/S模式下, 可包括多个客户机。
 - C) 客户机使用标准语言进行信息传递。
 - D) 客户机可使用缓冲或优化技术。
 - E) 客户机通过一个进程间通讯机制完成通信。
 - F) 客户机对服务器送回的信息进行分析, 提交用户。



C/S模式

- (2) 服务器的特点:
- A)服务器向客户机提供一种服务，服务的类型有系统定；
- B)服务器只负责响应来自客户机的查询或命令。不主动提供联系。
- C)在多服务器的环境下，服务器之间可以协调工作，共同向客户机提供服务。



C/S模式

- 客户/ 服务器的特点：
 - A) 桌面上的智能
 - B) 优化地共享服务器资源
 - C) 优化网络利用率
 - D) 底层操作系统和通信系统提供一个抽象层。易移植和维护。



Chapter 3 进程间的通信

- 3.0 进程及其状态转换
- 3.1 进程间的通信
- 3.2 客户/服务器计算模式
- 3.3 Socket编程基础(C)
- 3.4 应用实例(C)



3.3 Socket 编程基础

- 网络概述
- TCP/IP基础知识
- 使用TCP的 Client/Server
- Socket中的系统调用

网络基础

- 1. 终端登录与网络登录
 - 终端登录: **login: root**
 - **password: XXXX**
 - 网络登录: **telnet hostname**
 - （该客户进程打开一个到名hostname的主机的TCP连接。在hostname主机上启动的程序被称为TELNET服务器,客户进程和服务进程之间使用TELNET应用协议通过TCP连接交换数据）。

文件I/O和网络I/O

文件I/O

- 六种输入/输出服务：
`open, creat, close, read, write`和`lseek`

■ 网络I/O：（比文件I/O复杂）

- 1) 客户/服务器关系是非对称的，在初始化一条网络连接时应该让程序知道它扮演的角色（客户/服务器）。
- 2) 网络I/O有面向连接与非连接的。前者很类似文件I/O，顾客可以采用`fd`格式；而后者没有`open`的必要。



网络I/O

- 3) 网络中名字比标识更重要。文件继承了fd后，无需知道对应的名称，而网络需要名称进行权限检查。
- 4) UNIX的文件I/O是面向流的，而网络I/O有面向流与消息的。
- 5) 网络I/O要支持多种协议，文件I/O需考虑的参数少。

通信协议

通信协议：用于协调不同网络设备之间的信息交换，建立设备之间相互识别的有价值的信息机制。如：**XNS** , **IPX/SPX** , **TCP/IP**。

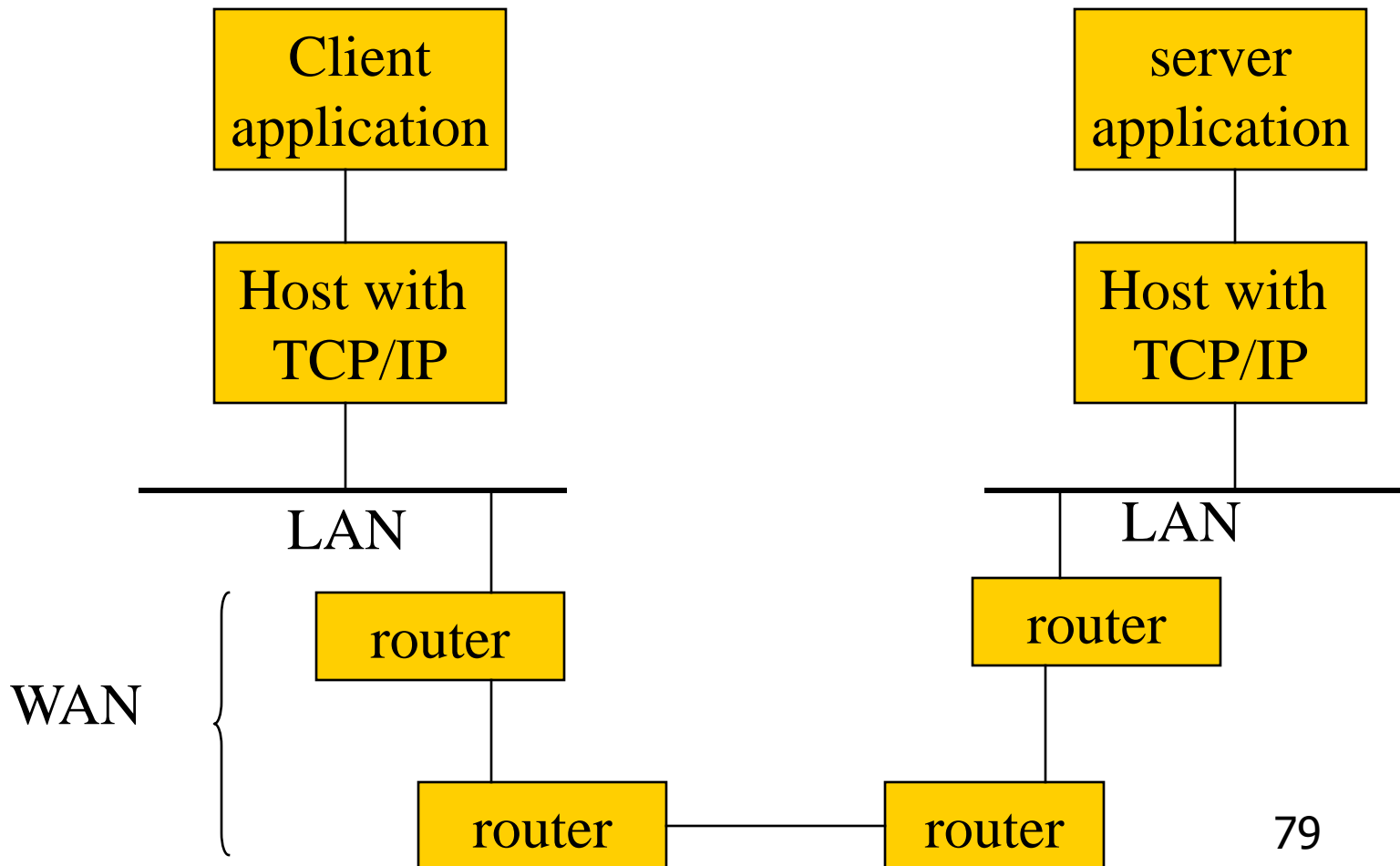
- 广域网 **WAN**

- 局域网 **LAN**

- 集中模式（信息的处理和控制是集中的）
- 客户/服务器模式（信息处理是分布的，系统控制是集中的）
- 对等模式（信息的处理和控制都是分布的）

TCP/IP概述

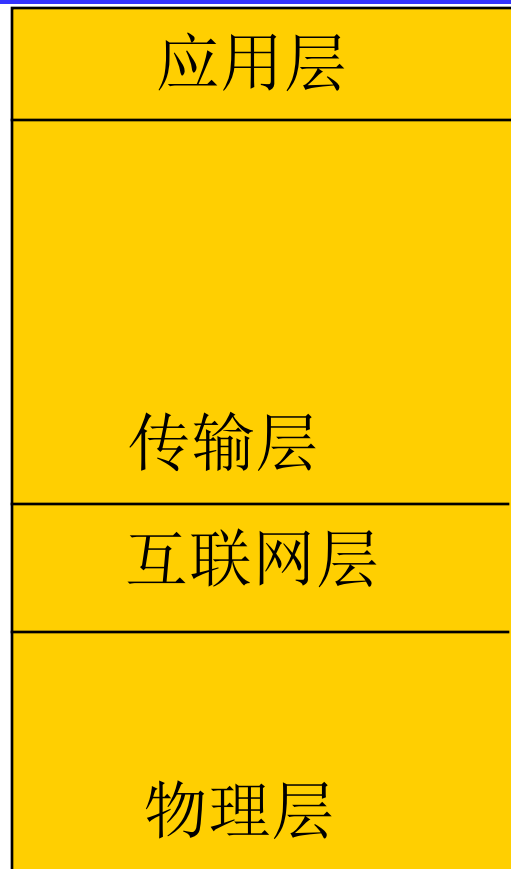
1. 网络



2.网络模型



OSI



TCP/IP

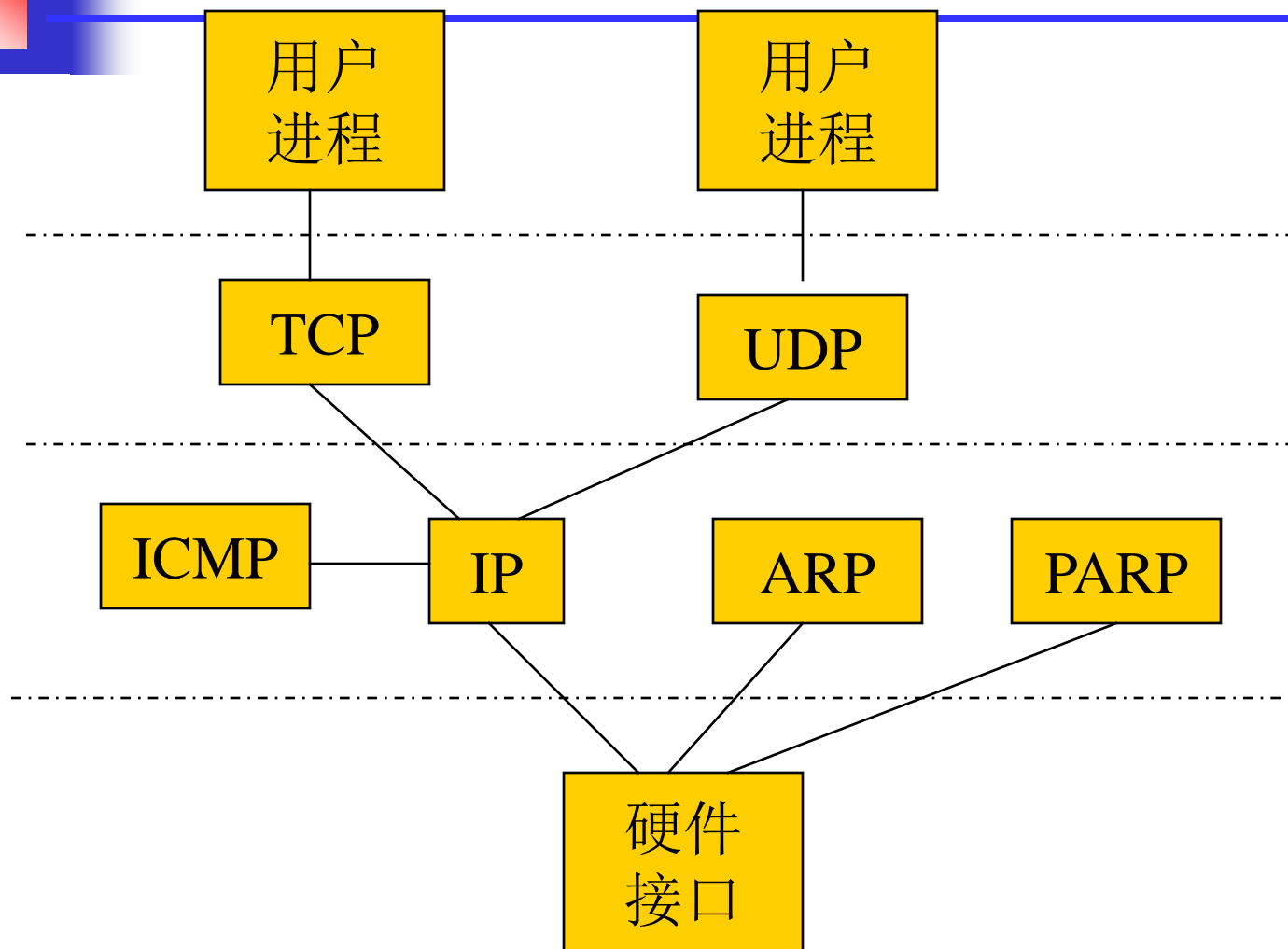
网络层次

- 应用层:包含所有的高层协议，如远程虚拟终端协议TELNET、文件传输协议FTP、简单邮件传输协议SMTP等。
 - TELNET协议：允许用户远程登录到另一台UNIX机器；
 - FTP协议：用于传输文件，
 - SMTP协议：用来传送email，常见的服务器端程序有netscape等公司制作的程序，也有免费使用的sendmail程序；
 - 还有域名系统服务DNS协议，新闻组传送协议NNTP，用于WWW的超文本传输协议HTTP等。

网络层次

- 传输层: 主要功能使源和目的主机之间可以进行会话。该层定义了两个端到端的协议, 一个是面向连接的传输控制协议TCP, 另一个是无连接的用户数据报协议UDP。
- 互联网层: 是基于无连接互联网络层的分组交换网络。在这一层中主机可以把报文 (**Packet**) 发往任何网络, 报文独立地传向目标。互联网层定义了报文的格式和协议, 这就是IP协议族 (**Internet Protocol**)。互联网层的功能是将报文发送到目的地, 主要的设计问题是报文路由和避免阻塞。
- 物理层:

TCP/IP协议框架





TCP/IP协议框架

- **TCP**（传输控制协议）：是面向连接的协议。为用户提供可靠的、全双工的字节流服务。
- **UDP**（用户数据报协议）：一种无连接的协议。不保证数据报一定能到达目的地。
- **ICMP**（网间控制报文协议）：用来处理网间连接器和主机之间的差错信息和控制信息。



Socket描述与使用

- **Socket**（进程间通信手段）支持下列通信协议
 - **UNIX域**
 - **Internet域**
 - **XNS域**



1. socket 描述符

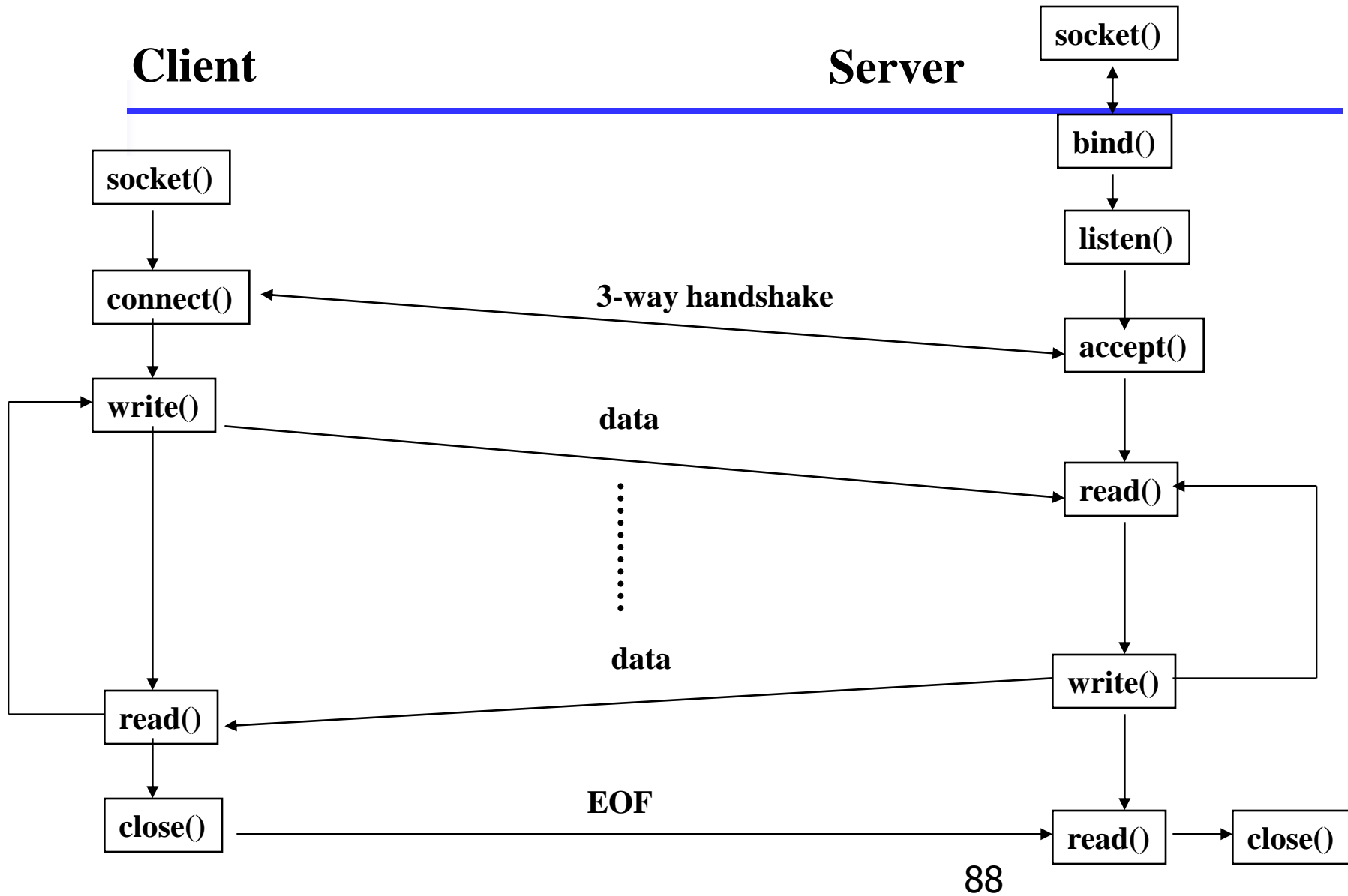
- **socket**接口增加了网络通信操作的抽象定义，与文件操作一样，每个打开的**socket**都对应一个整数，我们称它为**socket**描述符。
- 该整数也是**socket**描述符在文件描述符表中的索引值。**socket**描述符 指向一个与该**socket**有关的数据结构。
- **BSD UNIX**中**socket**调用，可以新建一个**socket**描述符。并建立通信的部分工作，一旦建立了一个**socket**，应用程序可以使用其他特定的调用，完成建立通信的过程。



2 socket用于网络编程

- 客户/服务器模式。
- 服务端:有一个进程（或多个进程）在指定的端口等待客户来连接，服务程序等待客户的连接信息，一旦连接上之后，就可以按设计的数据交换方法和格式进行数据传输。
- 客户端:在需要的时刻发出向服务端的连接请求。

TCP Connection Sequence



服务端进程:

- (首先启动服务器, 过一些时间客户启动, 并连接到服务器)
- (1)使用`socket`调用得到一个描述符,
- (2)使用`bind`调用将一个名字与`socket`描述符连接起来, 对于Internet域就是将Internet地址联编到`socket`。
- (3)服务端使用`listen`调用指出等待服务请求队列的长度。



服务端进程:

- (4)使用**accept**调用等待客户端发起连接，一旦有客户端发出连接，**accept**返回客户的地址信息，并返回一个新的**socket**描述符，该描述符与原先的**socket**有相同的特性，
- (5)服务端使用新的**socket**进行读写操作。
- 说明：一般服务端可能在**accept**返回后创建一个新的进程进行与客户的通信，父进程则再到**accept**调用处等待另一个连接。



客户端进程:

- (1)使用socket调用得到一个socket描述符
- (2)使用connect向指定的服务器上的指定端口发起连接，一旦连接成功返回，就说明已经建立了与服务器的连接
- (3)通过socket描述符进行读写操作。



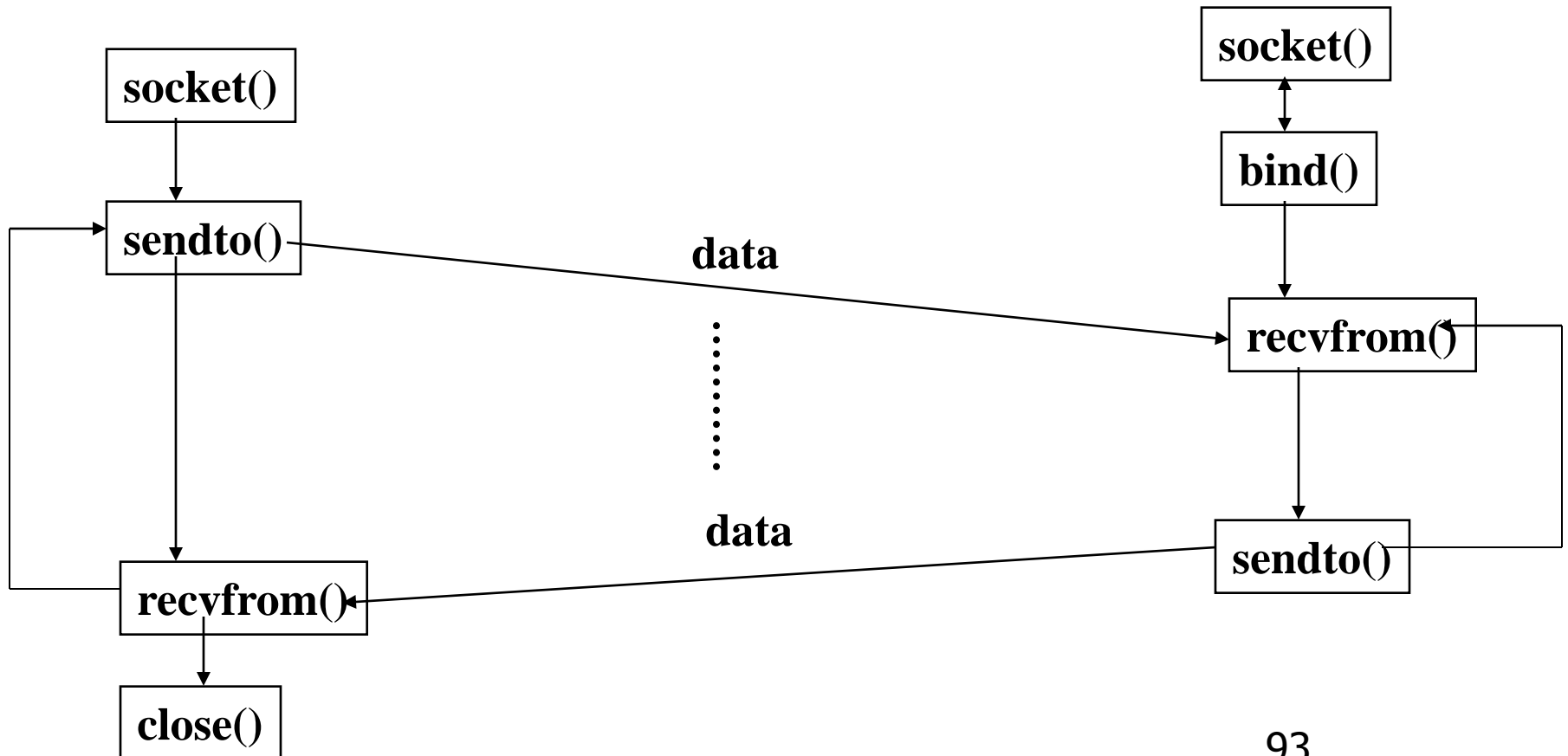
管套调用类型（无连接协议UDP）

- 客户不是和服务器建立连接，而是使用**sendto**系统调用向服务器发一个数据报，作为参数请求目的地址（服务器端）。
- 服务器也不必接收来自客户的连接，而是发送系统调用**recvfrom**，等待来自客户的数据到达。此系统调用返回客户进程的网络地址和数据报，服务器可向对方进程发出响应。

UDP Connection Sequence

Client

Server





服务端进程

- 创建一个socket,
- 调用recvfrom接收客户端的数据报,
- 调用sendto将要返回客户端的消息发送给客户进程。
- 客户端:
- 创建一个socket,
- 再使用sendto向服务端进程发出请求,
- 使用recvfrom得到返回的消息。



进程通信系统调用

- **Socket**
- **connect**（建立连接）
- **write、read及close**
- **send和 sendto**
- **recv 和 recvfrom**
- **bind(将一个名字绑定到一个socket)**



1. Socket的系统调用

- 调用**socket**用来建立一个通信的端点。
- 声明格式如下：
- **int socket(int domain, int type, int protocol);**
- 头文件：**#include <sys/types.h>**
- **#include <sys/socket.h>**
- **socket**调用建立一个通信端点并返回一个**socket**描述符。



Socket的系统调用

- 参数`domain`用来指定发生通信的域，选择要使用的协议族的参数。
- 参数`type` 指通讯的类型。
- 参数`protocol`用来指定在`socket`上使用的特定协议。
- 如：在`PF_INET`域中，类型为`SOCK_STREAM`的`socket`支持TCP协议，
- 类型为`SOCK_DGRAM`的`socket`支持UDP协议。



2. Connect系统调用

- **connect**的作用是在一个指定的**socket**上建立一个连接。
- 声明的格式:
- **int connect(int s, const struct sockaddr *name, int namelen)**
- 头文件: **#include <sys/types.h>**
- **#include <sys/socket.h>**
- 第一个参数**s**是一个打开的**socket**描述符。

Connect系统调用

- 如果该socket的类型是SOCK_DGRAM，该调用指出与该socket绑定的节点，指定的地址是数据报发送到的地址。并不真正建立连接，仅仅是让系统知道写到该socket的数据报将发送到何地，而且只有该地址发来的数据才被该socket接收。在使用UDP时，使用connect的好处是不必为每次发送和接收指定地址，可以使用read、write、send和recv调用。
- 如果该socket的类型是SOCK_STREAM，与另一个socket建立连接。

Connect系统调用

- 第二个参数是一个地址结构，服务端的名字。
- 该结构如下
- `struct sockaddr {`
- `u_char sa_len; /* 总长度 */`
- `u_char sa_family; /* 地址簇 */`
- `char sa_data[14]; /* 实际，地址值 */};`
- 对于internet域的地址（AF_INET）表示，常用sockaddr_in结构，头文件<netinet/in.h>中
- 第三个参数是指出地址的长度。

3 write、read及close 调用

- 在网络程序中使用write、read和close调用的方法与一般文件基本相同。
- Write()
- 在使用TCP协议的程序中，客户端和服务端都可以使用write来发送数据，唯一不同的是第一个参数是socket描述符。
- 第二个参数是指向要发送的数据，
- 第三个参数是发送的数据长度。一般核心将要发送的数据复制到发送数据缓冲，在网络传送的过程中发送数据的进程继续执行。但当缓冲被写满后，write调用会临时被阻塞，直到缓冲中有数据传送完成使得缓冲中有空间可用时为止。

read调用

- 第一个参数也是socket描述符，
- 第二个参数是存放读入数据的用户缓冲地址，第三个参数是缓冲的长度。
- 核心也使用一个接收数据缓冲，当用户进程使用read调用读入数据时，核心将核心缓冲的数据复制到用户的缓冲中。当核心缓冲中数据的数量大于用户缓冲的大小时，核心只复制用户缓冲大小的数据。当客户端或服务端使用UDP协议时，程序也可以使用read来接收UDP数据报，但当消息的长度大于用户缓冲时，核心仅为read调用复制用户缓冲大小的数据，剩余数据将被丢弃。




Close调用


- 当客户端或服务端使用完一个socket时，可以使用close来关闭该socket。close调用的参数为socket描述符。
- 在TCP协议下，发出该调用将断开连接，如果该描述符有多个引用（与打开文件相同），则将该socket的引用数减一，如果引用数减为0，则释放该socket。



4. Send和 sendto

- **send和sendto用来从一个socket发送一个消息。**
- **声明格式:**
- **int send(int s, const void *msg, int len, unsigned int flags);**
- **int sendto(int s, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);**


- 
- 头文件: **#include <sys/types.h>**
 - **#include <sys/socket.h>**
 - 调用**send**和**sendto**用来发送一个消息到另一个**socket**。
 - **send**调用只用于处于连接状态的**socket**,
 - **sendto**不需要**socket**处于连接状态, 在**sendto**调用中要指出消息的目的地。

- 
- 参数s是一个socket描述符，
 - 参数msg指向用户空间的一片缓冲空间，该处存放要发送的消息，消息的长度由参数len指出。在sendto调用中，
 - 参数to为sockaddr_in结构，用于指出在AF_INET域中目的地的地址，地址结构的大小由参数tolen指出。
 - 第四个参数flags可以包括一个或多个常量值（“或”运算）

5. Recv 和 recvfrom

系统调用recv和recvfrom用来从一个socket接收消息

- 声明格式如下:
- `int recv(int s, void *buf, int len, unsigned int flags);`
- `int recvfrom(int s, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);`
- 头文件: `#include <sys/types.h>`
- `#include <sys/socket.h>`

- 
- 参数s为要接收数据的socket描述符。
 - 参数buf指向接收数据后存放的用户缓冲的地址。
 - 参数len指出用户缓冲的大小。
 - 系统调用recvfrom用来从一个socket接收消息，不管这个socket是否面向连接。
 - 参数from非空，并且该socket不是面向连接的，则recvfrom调用返回时，参数from中存放的是消息的源地址。

6. Shutdown


- **shutdown**调用该调用将关闭一个全双工连接的一端。
- 格式如下: **int shutdown(int s, int how);**
- 头文件: **#include <sys/socket.h>**
- **shutdown**调用将导致**socket**描述符**s**所指的全双工**socket**连接被部分或全部关闭。
- 如果参数**how**指定为**0**, 则被该调用关闭的**socket**将禁止接收数据。
- 如果参数**how**为**1**, 则禁止继续发送数据。如果**how**为**2**, 则禁止在该**socket**上接收或发送数据。

7. Bind调用

- **bind调用将一个名字绑定到一个socket上。**
- **格式如下： `int bind(int s, struct sockaddr *my_addr, int addrlen);`**
- **头文件： `#include <sys/types.h>`**
- **`#include <sys/socket.h>`**
- **bind调用为一个未命名的socket分配一个名字。**
- **即： bind调用是给套接字s赋予本地的地址 `my_addr`，这个 `my_addr` 地址的长度为 `addrlen`。对于不同通信域的地址联编的规则是不同的，这里还是以internet域为主。**

8. Listen

- **listen** 调用该调用在一个**socket**侦听连接。
- 格式如下：**int listen(int s, int backlog);**
- 头文件：**#include <sys/socket.h>**
- 首先由**socket**调用创建一个**socket**，而**listen**调用指定接受连接的队列长度限制，然后由**accept**调用来接受连接请求。该调用只能用在**SOCK_STREAM**或**SOCK_SEQPACKET**类型的**socket**上。

- 
- 参数backlog指定队列的最大长度，该队列用来放置等待处理的连接请求。
 - 如果服务端的等待队列已满，客户的连接请求将导致客户端收到一个错误，
 - 或者请求被忽略。

9. Accept

- **accept** 调用在指定的socket上接受一个连接请求。
- 格式如下: `int accept(int s, struct sockaddr *addr, int *addrlen)`
- 头文件: `#include <sys/types.h>`
- `#include <sys/socket.h>`
- 参数s是要接受连接请求的socket, 这个socket是由socket调用创建,
- 第二个参数addr是用来存放返回值的, 在使用IP地址时, 使用sockaddr_in结构, accept返回时, 该结构中存放的是连接实体的地址。
- 第三个参数addrlen在调用前用来指出addr的大小, 在调用后返回精确的addr的长度。

10. 其它调用

- **DNS(域名服务)**
 - 通过因特网名字得到名字所代表的**IP地址**。
 - **\$ telnet bbs.tsinghua.edu.cn**
 - **IP为 202.112.58.200**
 - **人大四层UNIX机器:**
 - **IP为 202.112.115.211**



Socket API summary

- **int `socket`(int family, int type, int protocol)**

Creates a network plug point that enables the client/server to communicate

- **int `connect`(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen)**

Enables a client to connect to a server.

- **int `bind`(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen)**

Allows a server to specify the IP address/port_number associated with a socket



Socket API summary...

- **int** **listen**(int sockfd, int backlog)

Allows the server to specify that a socket can be used to accept connections.

- **int** **accept**(int sockfd, struct sockaddr *client_addr, socklen_t *addrlen)

Allows a server to wait till a new connection request arrives.

- **int** **close**(int sockfd)

Terminates any connection associated with a socket and releases the socket descriptor.



Socket API summary...

- **int read(int sockfd, void *buf, int count);**
Read data from a TCP connection.
- **int write(int sockfd, void *buf, int count)**
Write data to a TCP connection.
- **int sendto(int sockfd, void *msg, ...)**
Send a UDP message on a socket to specified destination
- **int recvfrom(int sockfd, void *buf, ...)**
Recv a UDP message on a socket along with address of sending source.
- Similarly check man pages for **sendmsg(), recvmsg()**.

Converting Between Address formats

■ From ASCII to numeric

- “130.245.1.44” → 32-bit network byte ordered value
- `inet_aton(...)` with IPv4
- `inet_pton(...)` with IPv4 and IPv6

■ From numeric to ASCII

- 32-bit value → “130.245.1.44”
- `inet_ntoa(...)` with IPv4
- `inet_ntop(...)` with IPv4 and IPv6

- Note – `inet_addr(...)` obsolete

- cannot handle broadcast address “255.255.255.255”
(0xFFFFFFFF)

Getting IP address/port from socket

- **int `getsockname`(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen)**
 - Get the local IP/port bound to socket
- **int `getpeername`(int sockfd, struct sockaddr *remoteaddr, socklen_t *addrlen)**
 - Get the IP/port of remote endpoint

Domain Name \leftrightarrow IP address



- `struct hostent *gethostbyaddr (void *addr, size_t len, int type);`
 - Converts from IP addr to domain name
- `struct hostent *gethostbyname (char *name);`
 - Converts from domain name to IP address



Chapter 3 进程间的通信

- 3.0 进程及其状态转换
- 3.1 进程间的通信
- 3.2 客户/服务器计算模式
- 3.3 Socket编程基础(C)
- 3.4 应用实例(C)



3.4 应用实例

- **程序功能：服务器向客户端发一个字符串 “hello, world!”.**


简单的流服务器

- (Ex37.c/* server */
- #include <stdio.h>
- #include <stdlib.h>
- #include <errno.h>
- #include <string.h>
- #include <sys/types.h>
- #include <netinet/in.h>
- #include <sys/socket.h>
- #include <sys/wait.h>
- //服务器倾听本地端口
- #define MYPORT 4000
- //能够同时接受多少没有accept的连接
- #define BACKLOG 10

main()

{

- `fd_set readfds;`
- `int sock_fd,new_fd;`
- `//自己的地址信息`
- `struct sockaddr_in my_addr;`
- `//连接者的地址信息`
- `struct sockaddr_in their_addr;`
- `int sin_size;`
- `if((sock_fd=socket(AF_INET,SOCK_STREAM,0))`
`== -1)`
- `{ perror("socket failed");`
- `exit(1);}`

- 
- **//主机字节顺序**
 - **my_addr.sin_family=AF_INET;**
 - **//将运行程序机器的IP填入s_addr**

 - **my_addr.sin_addr.s_addr=INADDR_ANY;**
 - **//网络字节顺序**
 - **my_addr.sin_port=htons(MYPORT);**
 - **//将此结构的其余空间清零**
 - **bzero(&(my_addr.sin_zero),8);**
 - **if(bind(sock_fd,(struct sockaddr**
***)&my_addr,sizeof(struct sockaddr))== -1)**
 - **{ perror("bind failed"); exit(1);}**
 - **if(listen(sock_fd,BACKLOG)== -1)**
 - **{ perror("listen failed");**
 - **exit(1);}**



```
while(1)  
{ //主accept( ) 循环  
  
    sin_size=sizeof(struct sockaddr_in) ;  
  
    if((new_fd=accept(sock_fd,(struct  
sockaddr*)&their_addr,&sin_size))==-1)  
  
        {            perror("accept failed");  
  
            continue; }  
  
    //服务器给出出现连接的信息  
  
    printf("server:got connection from  
%s\n",inet_ntoa(their_addr.sin_addr));
```



```
if(!fork())
```

```
{//子进程
```

```
    if(send(new_fd,"hello,work!\n",14,0)==-1)
```

```
    { perror("send failed");
```

```
      close(new_fd);
```

```
      exit(0);
```

```
    } //关闭new_fd代表的这个套接值的连接
```

```
    close(new_fd);          }      }
```

```
//等待所有的子进程都退出
```


```
while(waitpid(-1,NULL,WNOHANG)>0);
```

```
}
```




简单的流式客户端(ex38.c)

- **`/* client */`**
- **`#include <stdio.h>`**
- **`#include <stdlib.h>`**
- **`#include <string.h>`**
- **`#include <netdb.h>`**
- **`#include <sys/types.h>`**
- **`#include <netinet/in.h>`**
- **`#include <sys/socket.h>`**
- **`#define PORT 4000`**
- **`#define MAXDATASIZE 100`**




- **int main(int argc,char *argv[])**
- **{ int sockfd,numbytes;**

- **char buf[MAXDATASIZE];**
- **struct hostent * he;**
- **struct sockaddr_in their_addr;**
- **if(argc!=2)**
- **{ fprintf(stderr,"usage:client hostname\n") ;**
- **exit(1);}**
- **if((he=gethostbyname(argv[1]))==NULL)**
- **{ perror("gethostbyname failed");**
- **exit(1);**
- **}**

- 
- `printf("host ip %s\n",inet_ntoa(*((struct in_addr *)he->h_addr)));`

 - `if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)`
 - `{ perror("socket failed"); exit(1);}`
 - `their_addr.sin_family=AF_INET;`
 - `their_addr.sin_port=htons(PORT);`
 - `their_addr.sin_addr=*((struct in_addr *)he->h_addr);`
 - `bzero(&(their_addr.sin_zero),8);`
 - `if(connect(sockfd,(struct sockaddr *)&their_addr,sizeof(struct sockaddr))==-1)`
 - `{ perror("connect failed");`
 - `exit(1);}`

- 
- **if((numbytes=recv(sockfd,buf,MAXDATASIZE,0))==-1)**
 - **{ perror("recv failed");**
 - **exit(1);}**
 - **buf[numbytes]='\0';**
 - **printf("Received:%s",buf);**
 - **close(sockfd);return 0;**
 - **}**




数据报套接字例程

- **listener.c :在服务器上运行，监听端口5000。**
- **talker.c : 发送UDP数据报道服务器的5000端口。**



UDP实例(ex39.c)

- **`/* listener */`**
- **`#include <stdio.h>`**
- **`#include <stdlib.h>`**
- **`#include <errno.h>`**
- **`#include <string.h>`**
- **`#include <sys/types.h>`**
- **`#include <netinet/in.h>`**
- **`#include <sys/socket.h>`**
- **`#include <sys/wait.h>`**
- **`#define MYPORT 5000`**
- **`#define MAXBUFLen 100`**

- 
- **main(**
 - **{**
 - **int sockfd;**

 - **struct sockaddr_in my_addr;**
 - **struct sockaddr_in their_addr**
 - **;int addr_len,numbytes;**
 - **char buf[MAXBUFLen];**
 - **if((sockfd=socket(AF_INET,SOCK_DGRAM,0))==-1)**
 - **{ perror("socket failed");**
 - **exit(1); }**
 - **my_addr.sin_family=AF_INET;**
 - **my_addr.sin_port=htons(MYPORT);**
 - **my_addr.sin_addr.s_addr=INADDR_ANY;**
 - **bzero(&(my_addr.sin_zero),8);**

- **if(bind(sockfd,(struct sockaddr *)&my_addr,sizeof(struct sockaddr))== -1)**
 - **{ perror("bind failed");**
 - **exit(1);}**
- **addr_len=sizeof(struct sockaddr);**
- **if((numbytes=recvfrom(sockfd,buf,MAXBUFLen,0,(struct sockaddr *)&their_addr,&addr_len))== -1)**
- **{ perror("recvfrom failed");**
- **exit(1);}**
- **printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));**
- **printf("packet is %d bytes long\n",numbytes);**
- **buf[numbytes]='\0';**
- **printf("packet contains \"%s\"\n",buf);**
- **close(sockfd);**
- **}**




UDP(ex40.c)


- **`/* talker */`**
- **`#include <stdio.h>`**
- **`#include <stdlib.h>`**
- **`#include <errno.h>`**
- **`#include <string.h>`**
- **`#include <sys/types.h>`**
- **`#include <netinet/in.h>`**
- **`#include <netdb.h>`**
- **`#include <sys/socket.h>`**
- **`#include <sys/wait.h>`**
- **`#define MYPORT 5000`**



```
int main(int argc,char *argv[])
```

- **{int sockfd;**
- **struct sockaddr_in their_addr;**
- **struct hostent *he;**
- **int numbytes;**
- **if(argc!=3)**
- **{ fprintf(stderr,"usage:talker hostnamemessage\n");**
- **exit(1);}**
- **if((he=gethostbyname(argv[1]))==NULL)**
- **{ perror("gethostbyname failed");**
- **exit(1);}**

- 
- **if((sockfd=socket(AF_INET,SOCK_DGRAM,0))==-1)**
 - **{ perror("socket failed");**
 - **exit(1);}**
 - **their_addr.sin_family=AF_INET;**
 - **their_addr.sin_port=htons(MYPORT);**
 - **their_addr.sin_addr=*((struct in_addr *)he->h_addr);**
 - **bzero(&(their_addr.sin_zero),8);**

- 
- **if((numbytes=sendto(sockfd,argv[2],strlen(argv[2]),0,(struct sockaddr *)&their_addr,sizeof(struct sockaddr)))==-1)**
 - **{ perror('recvfrom failed');**
 - **exit(1);}**
 - **printf('send %d bytes to %s\n',numbytes,inet_ntoa(their_addr.sin_addr));**
 - **close(sockfd);return 0;**
 - **}**