



第2章 进程与线程

2.8 进程间通信



2.8 进程间通信

- 进程间通信(InterProcess Communication, IPC)
 - 指进程之间的信息交换，是进程之间的一种协作机制
- 进程协作的原因：
 - 信息共享：多用户交换感兴趣的信息
 - 提高运算速度：任务分割，并行执行
 - 系统模块化要求：将系统按照模块化划分
 - 方便性：单用户同时处理多个任务



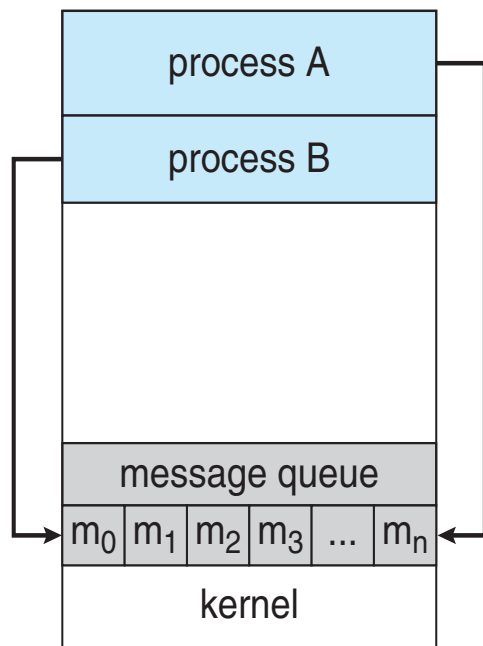
2.8 进程间通信

- 进程间通信的三个问题
 - 一个进程和其他进程协作时，如何把消息传递给其他进程
 - 如何确保两个或多个进程，在关键活动中不出现交叉，即相互之间不会出现互相干扰
 - 如果进程执行过程中对数据的访问存在先后的关联顺序，如何确保访问顺序的正确性

进程间通信的基本形态

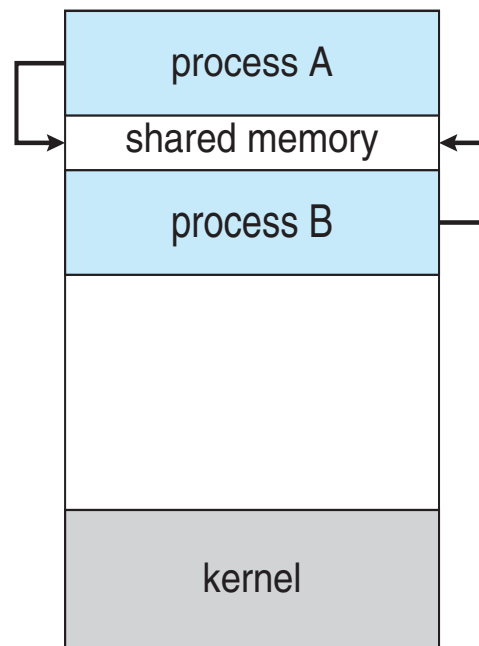
- 两类最基本的IPC模型
 - 共享内存模型，Shared memory
 - 消息传递模型，Message passing

消息传递模型



(a)

共享内存模型



(b)



进程间通信方式的发展(1)

- UNIX发展历史中，AT&T的Bell UNIX与加州大学伯克利分校的BSD是两大主力。
 - Bell致力于改进传统的进程IPC，形成了SYSTEM V IPC机制。
 - BSD在改进IPC的同时，把网络通信规程(TCP/IP)实现到UNIX内核中，考虑把计算机上的进程间通信纳入更广的网络范围的进程间通信，这种努力结果出现了socket通信机制。



进程间通信方式的发展 (2)

- **STSEM V IPC**进程间通信机制，包括：消息队列、共享内存和信号量。
 - 消息队列：允许一个进程向其他进程发送消息
 - 共享内存：让多个进程可共享它们的部分虚地址空间
 - 信号量：允许若干进程通过它来同步协作地运行



进程间通信的类型

- 消息传递
 - 信号机制(Signal)
 - 共享文件机制(Shared File)
 - 消息传递机制(Message Passing)
 - 信号量机制(Semaphore)
 - 套接字通信(Socket)
- 共享内存
 - 共享内存机制(Shared memory)



2.8.1 信号机制

■ 信号机制Signal

- 软件中断通知事件机制，是一种古老且简单的通信机制
- 它通过发送一个**指定信号**来通知进程某个异常事件发生。
- 每个信号都有一个名字和编号，这些名字都以“SIG”开头
 - 例：SIGKILL, SIGSEGV, SIGFPE, SIGSTOP, SIGINT
 - UNIX SYSTEM V中定义了19个软中断
 - Linux中定义了64个信号
 - kill -l



信号的特点

- ① 信号由特定事件的发生所产生
- ② 产生的信号要发送到进程
- ③ 一旦发送，信号必须加以处理
- ④ 信号利用中断机制实现对当前进程的执行中断，然后会转向信号处理程序，当信号处理完毕后，被中断进程将恢复执行
- ⑤ 同中断/异常相比，多数信号对用户态进程是可见的，可被用户进程捕获



信号的产生

- 信号来自内核，用户、内核和进程都能产生信号请求：
 - 用户
 - 当用户输入`ctrl + c`，或终端驱动程序分配给信号控制字符的其他任何键来请求内核产生信号，**SIGINT**
 - 内核
 - 当进程执行出错时，内核检测到事件并给进程发送信号，例如，非法段存取、浮点数溢出、或非法操作码，内核利用信号通知进程种种特定事件发生，**SIGSEGV**、**SIGILL**、**SIGALARM**
 - 进程
 - 进程可通过系统调用`kill()`给另一个进程发送信号，一个进程可通过信号与另一个进程间通信
 - `int kill(pid_t pid, int sig);`
 - `int sigqueue(pid_t pid, int sig, const union sigval value);`



信号的发送

- 信号的发送

- 指由发送进程把信号发送到指定进程信号域，传递的往往只是某一个位上的信号。

- 发送过程完成

- 如果目标进程正在一个可被中断的优先级上等待，内核就将它唤醒，则发送过程就此完成。

- 一个进程可能在其信号域中有多个位被置位，这意味着该进程同时有多个信号到达，但是对于同一类信号，进程只能记住其中一个。

- 进程用`kill()`向一个进程或一组进程发送一个信号。



信号的接收

- 信号通过模拟中断实现
 - 信号发送后必须处理
- 信号利用了类似中断的异步通信机制。
- 信号的接收可以是“同步”，也可以是“异步”
 - 同步信号：如非法访问内存或者除0错等，如果某个运行程序执行了这类动作，就会产生信号。并且该信号会发送到由于执行操作导致这个信号的同一进程
 - 异步信号：如果信号产生是运行程序意外事件产生的，则该进程就异步接收这一信号
- 查询时间点：
 - 一般在异常处理结束或者时钟中断处理结束返回之前查询有无信号



信号的处理

处理方法分为三类：

① 忽略信号

- 但SIGKILL和SIGSTOP信号不能忽略

② 捕获信号

- 该方法类似中断处理程序，当信号发生时，执行相应的信号处理函数

③ 执行缺省操作

- 内核默认的信号处理程序
- 可由用户自定义的信号处理程序重写



■ 例：用户杀死进程

- ① 用户键入中断组合键`ctrl+c`;
- ② 终端驱动程序收到输入字符，并调用信号系统;
- ③ 信号系统发送**SIGINT**信号给`shell`，`shell`再把它发送给进程;
- ④ 进程收到**SIGINT**信号;
- ⑤ 进程撤销。



信号与中断

- 相似点

- 采用了相同的异步通信方式；
- 当检测出有信号或中断请求时，都暂停正在执行的程序而转去执行相应的处理程序；
- 都在处理完毕后返回到原来的断点；
- 对信号或中断都可进行屏蔽。



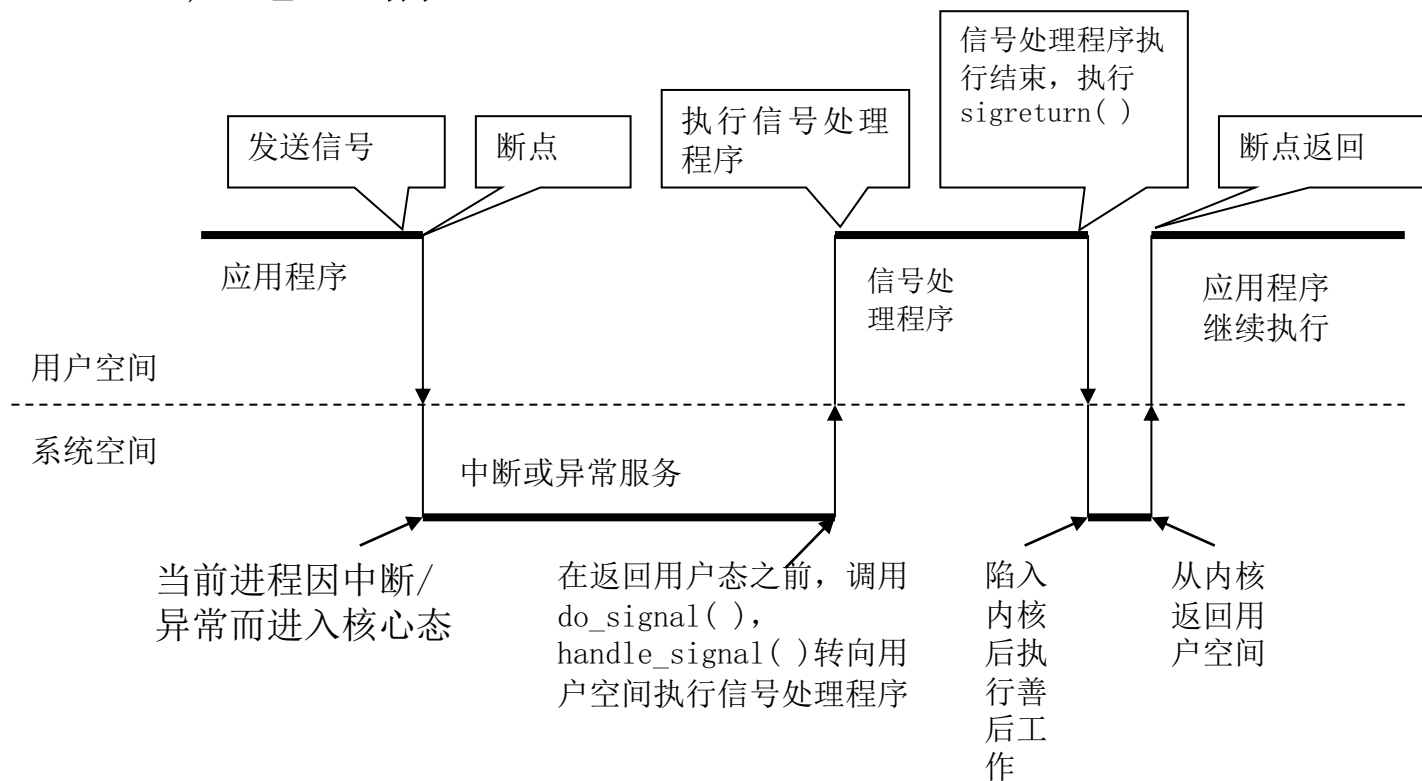
信号与中断

■ 区别

- 中断机制是有硬件和软件相结合来实现，信号机制是软件实现；
- 中断有优先级，而信号没有优先级，所有的信号都是平等的；
- 信号处理程序可由应用程序提供，往往是在用户空间下运行的，而中断处理程序是在核心态下运行；
- 中断响应是及时的，而信号响应通常都有较大的时间延迟。

信号的检测与响应(Linux)

- 信号被置于接收进程的PCB
- 检测与响应发生在系统终端，在进程从核心态返回用户态之前



Linux信号的检测与处理流程



信号机制的实现

- 信号有一个产生、传送、捕获和释放的过程
- 信号接收域**signal**
- 信号屏蔽位**blocked**
- 信号发送工作由系统调用**kill**完成
- 信号响应使用系统调用**sigaction**完成
- 信号的处理过程



Windows信号通信机制

- 分派器对象(dispatcher object)公共框架中，分派器对象可处于两种状态之一：已发信号(signaled)或未发信号(unsignaled)。
- 进程用函数WaitForSingleObject阻塞，等待一个未发信号的对象，当其他进程把该对象的状态改为已发信号时，该进程就会恢复执行。
- WaitForMultipleObjects
 - 对象类型：进程、线程、文件、事件、信号量、定时器、互斥量和队列。
 - 当线程释放互斥锁之后，就会向互斥量对象发信号，一个等待线程被唤醒；当一个定时器到时，就会向定时器对象发信号，可唤醒所有等待线程或仅仅唤醒一个。



2.8.2 消息传递机制

- 在消息传递机制中，由操作系统提供，进程间的数据交换以消息为单位
 - 程序员直接利用系统提供的一组通信命令原语来实现通信
 - `send(message), receive(message)`
- 消息传递的特点
 - 任何时刻发送
 - 一个正在执行的进程可在任何时刻向另一个正在执行的进程发送消息
 - 任何时刻请求
 - 一个正在执行的进程可在任何时刻向正在执行的另一个进程请求发出消息。
 - 消息传递机制与进程的同步协作紧密相关



消息传递的核心技术

- 消息传递实现的核心技术是消息队列管理
 - 由系统来帮助维护消息队列
 - 消息的传递就是对消息队列的消息挂载与消息查询
 - 消息传递提供了一种按照消息类型和自定义条件来查询进程间通信数据的方式，而不需要按照某种次序来进行排队接收



通信链路的逻辑特征

- 消息传递机制的通信链路，依据逻辑实现和操作特征可分为：
 - 直接通信&间接通信
 - 同步&异步通信
 - 自动&显示缓冲



1. 直接通信与间接通信

- Q: 如何实现通信进程之间的命名定位?
- 直接通信方式:
 - `Send(P, message)`: 向进程P发送消息
 - `Receive(Q, message)`: 从进程Q接收消息
- 逻辑链路特点
 - 只需要了解对方身份, 自动建立链路
 - 每个链路只与两个进程相关
 - 每队进程之间只有一个链路

直接通信的相关数据结构

- 实现上直接通信是对消息队列的插入和查询操作
- **消息**是指一组信息，消息缓冲区的数据结构如下：

struct message

{

sender; 发送者进程标识符

size; 消息长度

text; 消息正文

next; 指向下一个消息缓冲区的指针

}

- 在PCB中增加对消息机制的处理

struct PCB

{

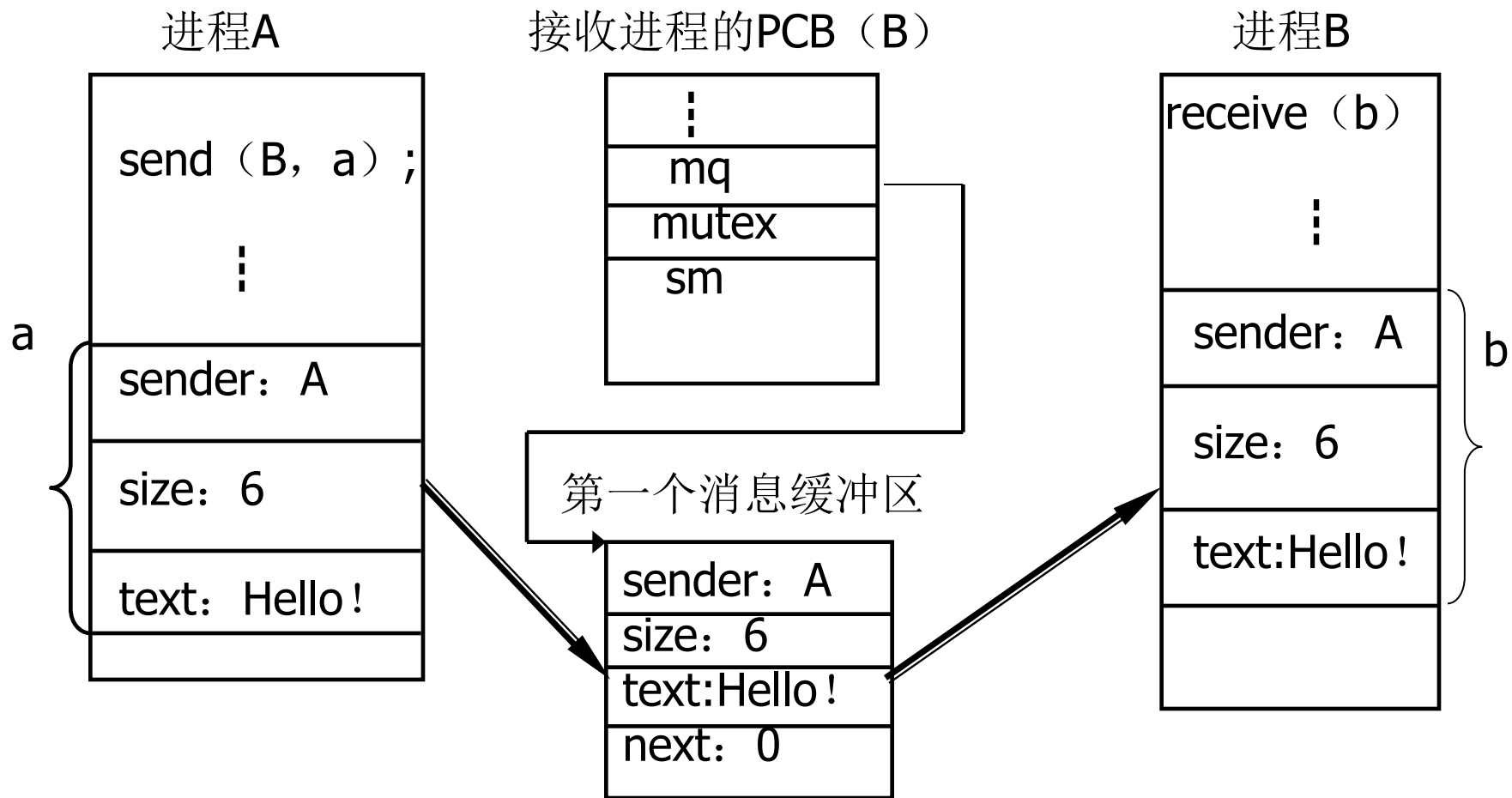
mq;消息队列队首指针

mutex;消息队列互斥信号量

sm;消息队列资源信号量

}

两个进程进行直接通信的过程





直接通信与间接通信

■ 间接通信方式:

- 发送进程将消息发送到信箱(mailbox)或者端口(port), 接收进程从信箱/端口中取消息。

■ 信箱通信机制

- 每一个信箱都有一个唯一的ID
- 进程间通过共享一个信箱实现通信

■ 通信链路的特点

- 当共享公用的邮箱时建立链路
- 链路可以关联多个进程
- 每对进程间可以共享多个通信链路
- 链路可以是单向和双向



信箱通信原语

■ 操作

- 创建一个新的信箱/端口
- 通过信箱发送&接收消息
- 销毁一个信箱

■ 原语定义

- `send(A, message)` - 向信箱A发送一条消息
- `receive(A, message)` - 从信箱A接收一条消息



信件的格式问题

- 单机系统中信件的格式可以分直接信件（又叫定长格式）和间接信件（又叫变长格式）。
-
- 网络环境下的信件格式较为复杂，通常分成消息头和消息体
 - 消息头：包括发送者、接收者、消息长度、消息类型、发送时间等各种控制信息；
 - 消息体：包含消息内容



2. 同步问题

- 消息传递包括阻塞&非阻塞
- 阻塞(Blocking): 同步(synchronous)
 - 阻塞发送: 发送进程阻塞, 直到消息被接收进程或邮箱接收
 - 阻塞接收: 接收进程阻塞, 直到消息可用
- 非阻塞(Non-blocking): 异步(asynchronous)
 - 非阻塞发送: 发送进程发送消息后, 继续其他操作
 - 非阻塞接收: 接收进程收到一个有效消息或者空消息, 继续其他操作

2. 同步问题(Cont.)

■ 不同组合

- 阻塞发送，阻塞接收：发送者和接收者间形成一个交会（rendezvous）
- 非阻塞发送，阻塞接收
- 非阻塞发送，非阻塞接收
- 注：没有阻塞发送，非阻塞接收，会导致发送进程死等

严格意义的顺序执行：生产者—消费者问题

```
message next_produced;  
while (true) {  
    /* produce an item in next produced */  
    send(next_produced);  
}
```

```
message next_consumed;  
while (true) {  
    receive(next_consumed);  
  
    /* consume the item in next consumed */  
}
```



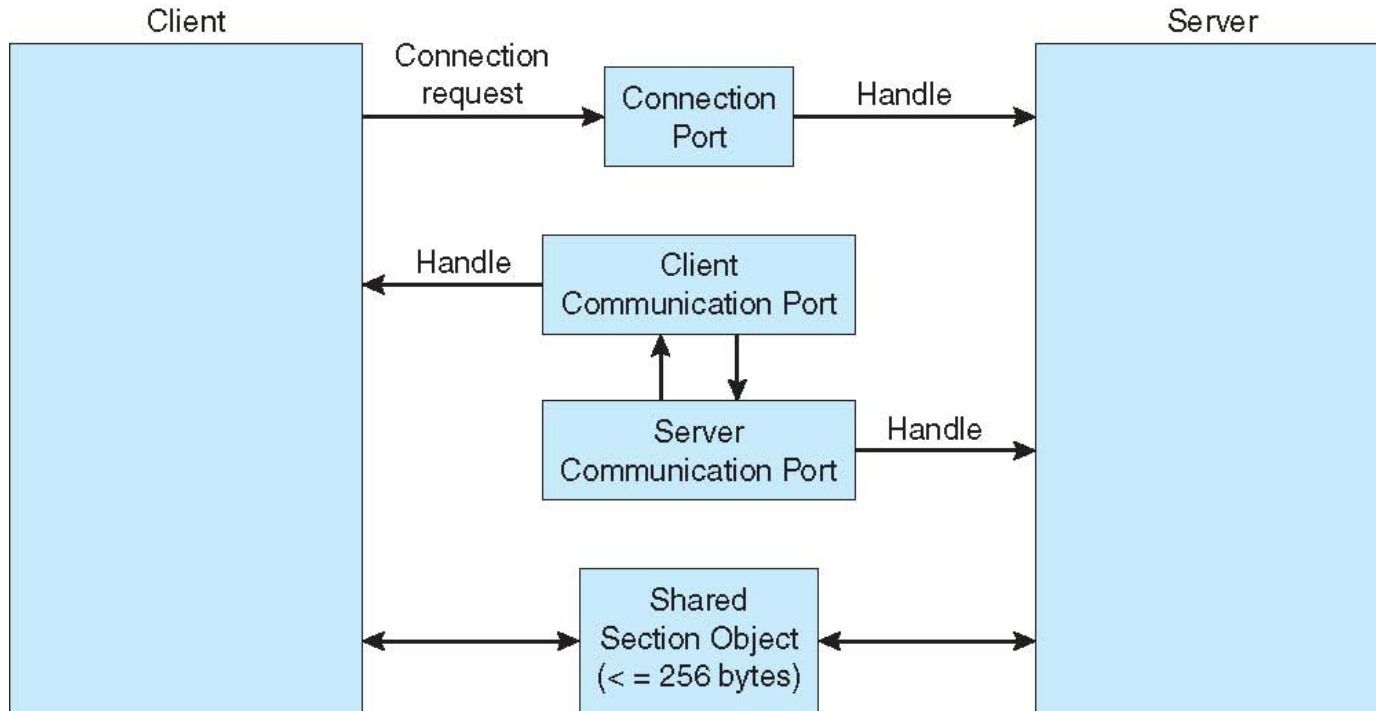


3. 缓存

- 采用消息传递方式的进程间通信，消息总是驻留在临时的队列中
- 实现方法：
 - 零容量：无缓冲系统
 - 队列最大长度为0，链路上没有消息处于等待队列上。即发送者是阻塞发送，直到消息被接收，
 - 自动缓冲系统
 - 有限容量：队列长度为有限的 n ，当链路满了之后，发送者要等待
 - 无限容量：队列长度无限，发送者从不等待

Local Procedure Calls in Windows

- 仅工作于同一个系统的两个进程间
- 使用端口ports(类似邮箱)建立和维护通信信道：连接端口+通信端口



1. 服务器进程发布连接端口对象，方便所有进程访问
2. 客户机打开一个系统的连接端口对象的句柄，并发送一个连接请求
3. 服务器建立两个私有的通信端口，形成一个通道，并把其中一个句柄返回给客户端
4. 客户端和服务端使用对应的端口句柄发送消息，或回调(callback)用于等待应答时，可以完成额外的接收请求。

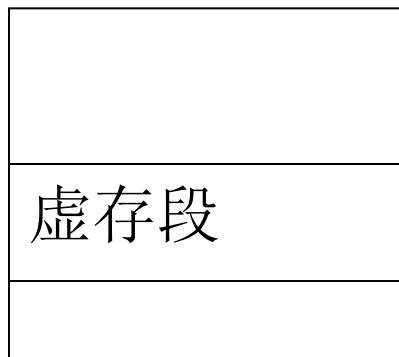


2.8.3 共享内存机制

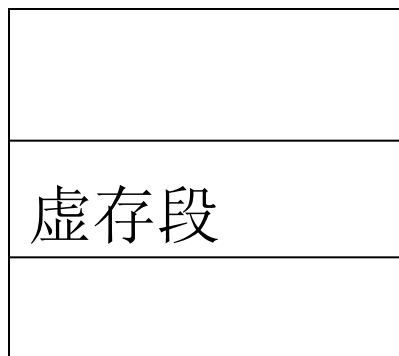
- 相互通信的进程共享某些数据结构或存储区。
 - 将通过一个物理内存空间映射到不同进程的内存空间
- 基于共享数据结构的通信方式
 - 诸进程通过共用某些数据结构交换信息。如生产者-消费者问题，共享了一个缓冲池
- 基于共享内存的通信方式
 - 在存储器中划出一块共享内存，诸进程可通过对共享内存进行读或写来实现通信。包括建立共享内存、附接及断接。

2.8.3 共享内存通信机制

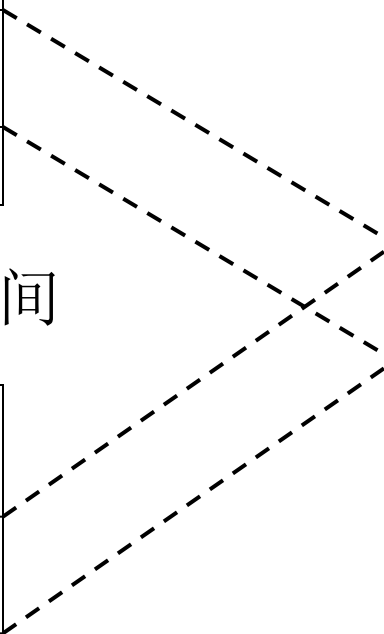
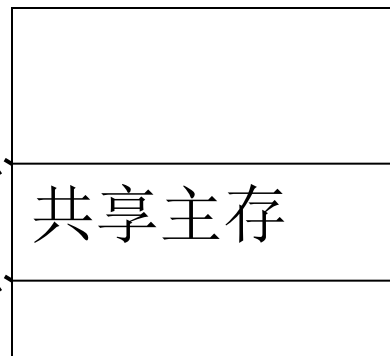
进程1的虚存空间



进程2的虚存空间



物理主存





共享内存通信例子- POSIX

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

```
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
```

- **Shmget:** 创建共享内存段

- `int shmget(key_t key, size_t size, int shmflg)`

- **key:** 所创建共享内存段的关键字

- **size:** 共享内存段大小

- **shmflag:** 访问模式

- 返回值为共享内存的标识符

- **Shmat:** 将共享内存段映射到调用进程的地址空间

- `void *shmat(int shmid, const void *shmaddr, int shmflg)`

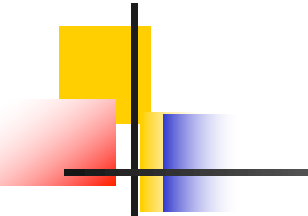
- **Shmid:** 共享内存的标识符

- **Shmaddr:** 共享内存映射后的地址指针，若为NULL，则系统来设置

- **Shmflag:** 映射模式

- **Shmdt:** 取消映射

- **Shmctl:** 访问与修改



```
16 int main(int argc, const char * argv[]){
17     int shm_id;
18     char * shared_memory;
19     const int size=4096;
20     pid_t pid ;
21     printf("Hello, World!\n");
22     shm_id = shmget(IPC_PRIVATE, size, IPC_CREAT|S_IRUSR|S_IWUSR);
23     if (shm_id < 0 ){
24         perror("get shm ipc_id error") ;
25         return -1 ;
26     }
27     pid = fork() ;
28     if (pid == 0){
29         printf("Child Process: %d\n",getpid());
30         shared_memory = (char *) shmat(shm_id, NULL,0);
31         sprintf(shared_memory,"Hi I'm Child Process!\n");
32         shmdt(shared_memory);
33         return 0;
34     }else if (pid > 0){
35         sleep(10) ;
36         printf("Parent Process: %d\n",getpid());
37         shared_memory = (char *) shmat(shm_id, NULL, 0 ) ;
38         printf("Parent Process: %s", shared_memory ) ;
39         shmdt(shared_memory);
40         shmctl(shm_id, IPC_RMID, NULL) ;
41         wait(NULL);
42         printf("Parent Process Exits: %d\n",getpid());
43         return 0;
44     } else{
45         perror("fork error") ;
46         shmctl(shm_id, IPC_RMID, NULL) ;
47         return -1;
48     }
49     return 0;
50 }
```



2.8.4 共享文件通信机制

- 管道(pipeline)
 - 是连接读写进程的一个特殊文件
 - 允许进程按先进先出方式传送数据，也能使进程同步执行操作。
 - 发送进程以字符流形式把大量数据送入管道，接收进程从管道中接收数据，所以叫**管道通信**。
- 管道的实质是一个共享文件，基本上可借助于文件系统的机制实现，包括（管道）文件的创建、打开、关闭和读写。



共享文件通信机制(2)

- 读写进程相互协调，必须做到：
 - 进程对通信机构的使用应该互斥，一个进程正在使用某个管道写入或读出数据时，另一个进程就必须等待。
 - write阻塞：管道写满了，写进程要等待
 - read阻塞：管道为空了，读进程要等待
 - 发送者和接收者双方必须能够知道对方是否存在，如果对方已经不存在，就没有必要再发送信息，这时系统会发出**SIGPIPE**信号通知进程



管道的类型

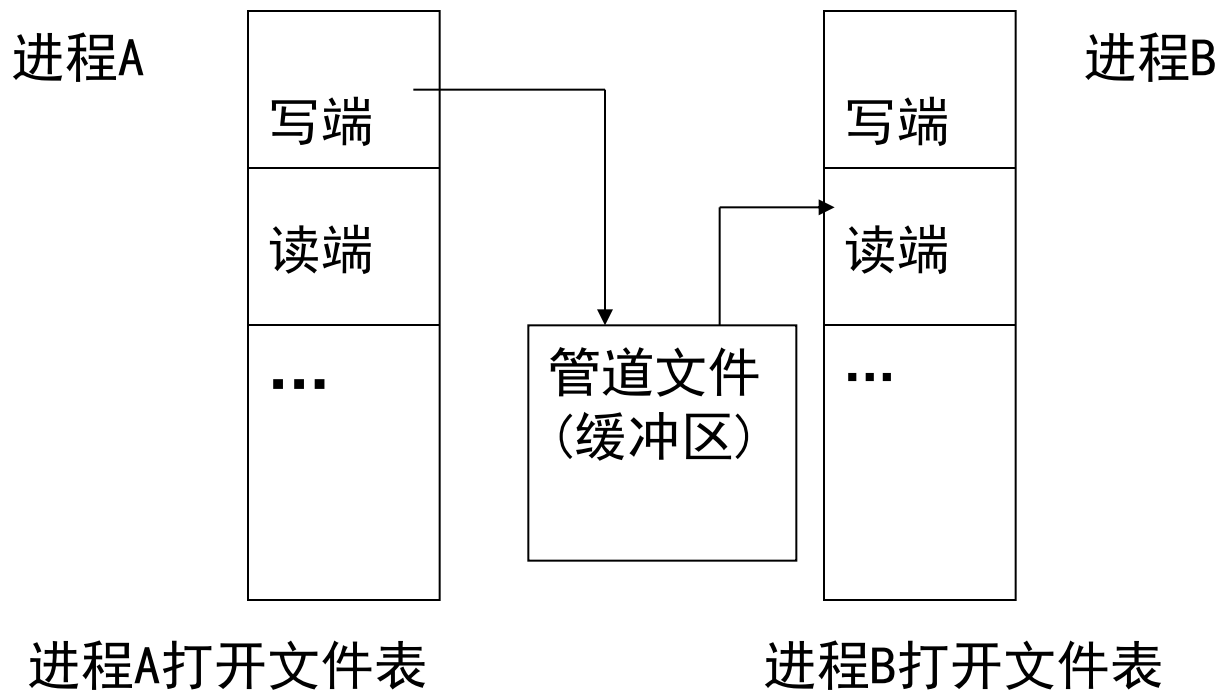
- 管道类型
 - 匿名管道
 - 有名管道
- 管道机制是**UNIX**设计的一大特点



匿名管道的特点

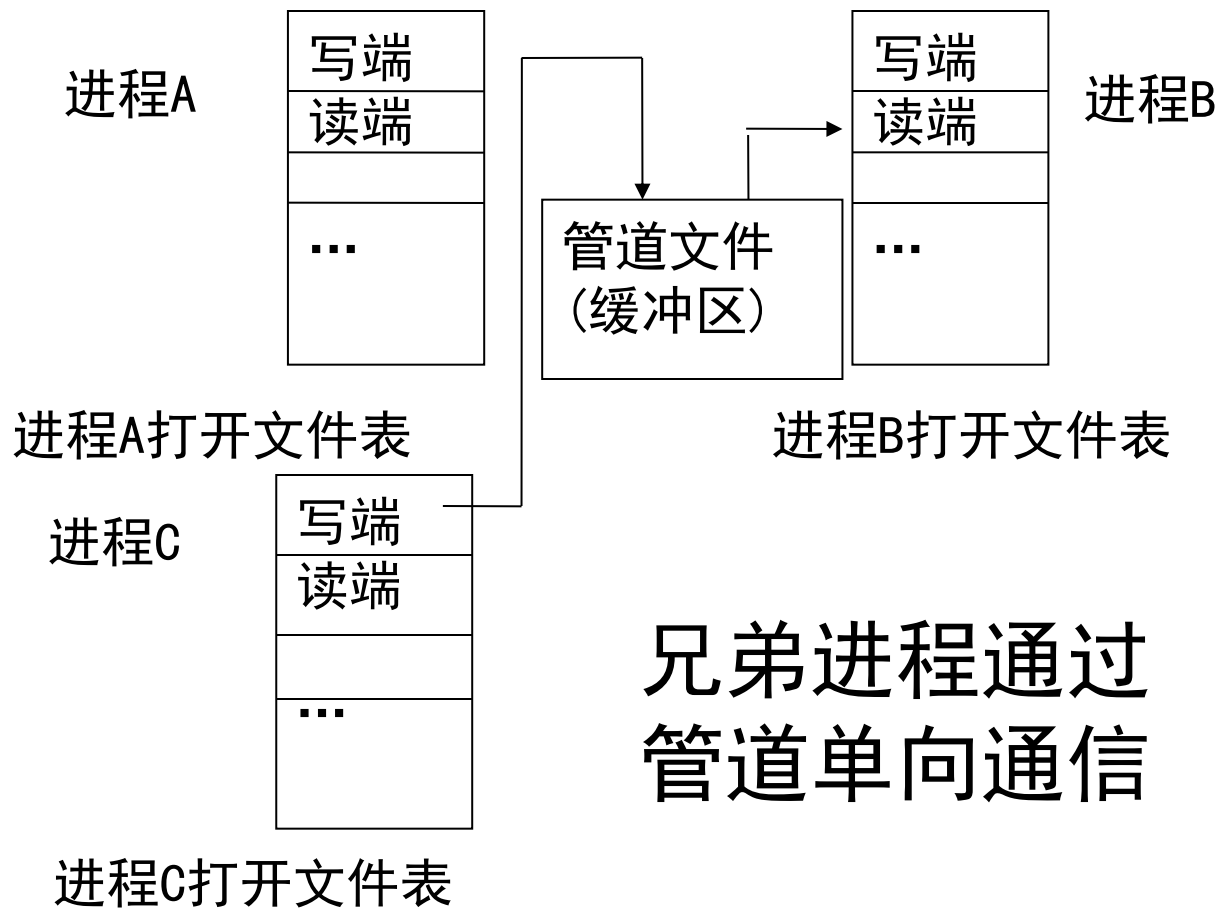
- 匿名管道是半双工的，数据只能向一个方向流动；要求双向通信时，需要建立两个匿名管道。
- 只能用于**具有亲缘关系**的进程间通信，亲缘关系指的是具有共同祖先，如父子进程或者兄弟进程之间。
- 匿名管道对于管道两端的进程而言，就是一个文件，但它不是普通文件，而是一个只存在于主存中的特殊文件。
- 一个进程向管道中写入的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

父子进程通过管道传递信息



父子进程通过
管道单向通信

兄弟进程通过管道传送信息



UNIX中的匿名管道

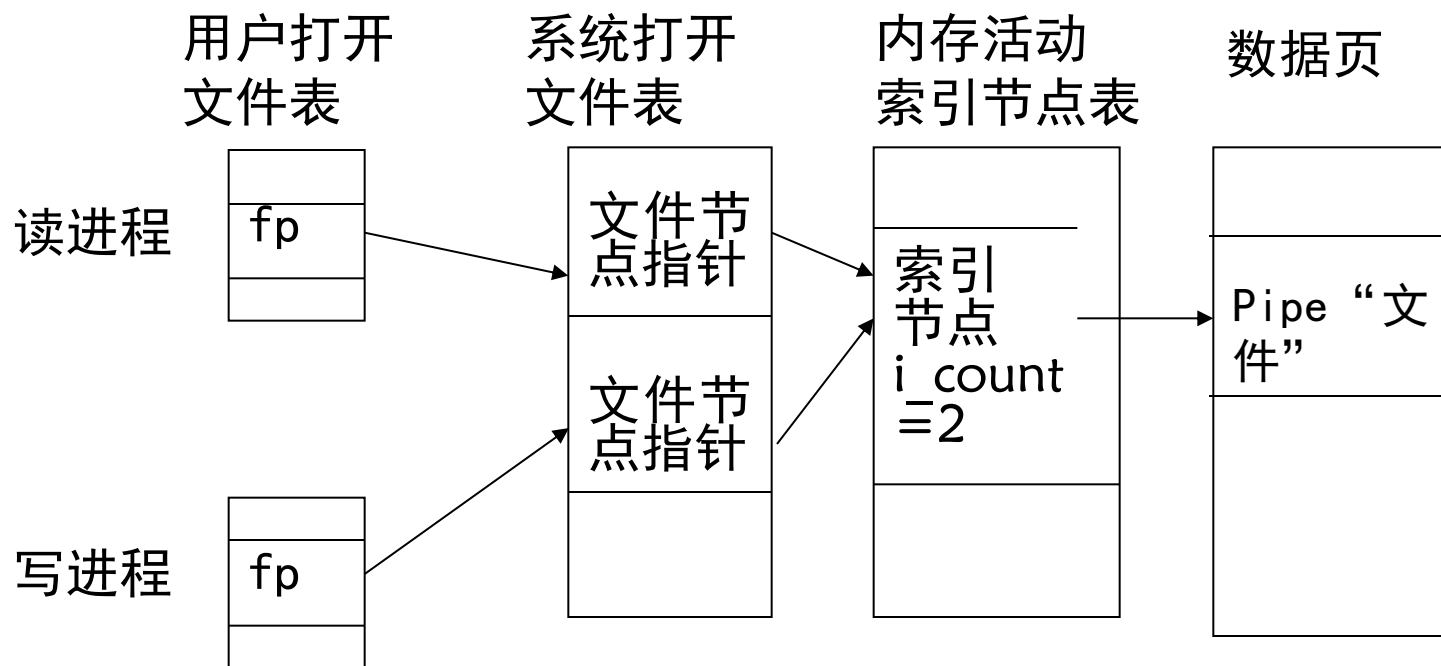
■ 匿名管道编程

- 使用pipe(int fd[])建立管道
- fd[0]为读出端，fd[1]为写入端
- 使用read(),write()对管道实现消息通信

```
17 #define BUFFER_SIZE 256
18 #define READ_END 0
19 #define WRITE_END 1
20 int main(int argc, const char * argv){
21     char write_msg[BUFFER_SIZE]="Greetings";
22     char read_msg[BUFFER_SIZE]={0};
23     int fd[2]={0};
24     pid_t pid;
25
26     if (pipe(fd)==-1) {
27         fprintf(stderr,"Pipe Falied");
28         return 1;
29     }
30     pid = fork();
31     if (pid<0) {
32         fprintf(stderr,"Fork Failed!\n");
33     }
34     if (pid>0) {
35         close(fd[READ_END]);
36         printf("Process %d write: \t%s\n",getpid(),write_msg);
37         write(fd[WRITE_END],write_msg,strlen(write_msg)+1);
38     } else {
39         close(fd[WRITE_END]);
40         read(fd[READ_END],read_msg,BUFFER_SIZE);
41         printf("Process %d read: \t%s\n",getpid(), read_msg);
42         close(fd[READ_END]);
43     }
44     return 0;
45 }
```

Pipe的数据结构

pipe的数据结构





有名管道

- 又称**FIFO**，克服只能用于具有亲缘关系的进程之间通信的限制。
- **FIFO**提供一个与路径名的关联，以**FIFO**的文件形式存在于文件系统中，通过**FIFO**不相关的进程也能交换数据。
- **FIFO**遵循先进先出，对有名管道及**FIFO**的读总是从开始处返回数据，对它们的写则把数据添加到末尾。



UNIX中的有名管道

- 有名管道利用mknod()/mkfifo()来实现，是可以在文件系统中长期存在的具有路径名的特殊文件
- 其他进程可以知道有名管道的存在并能利用路径名来访问该文件
- 对有名管道的访问方式可以像访问文件一样，要用open ()系统调用打开它。



2.8.5 Clinet-Server系统中的通信

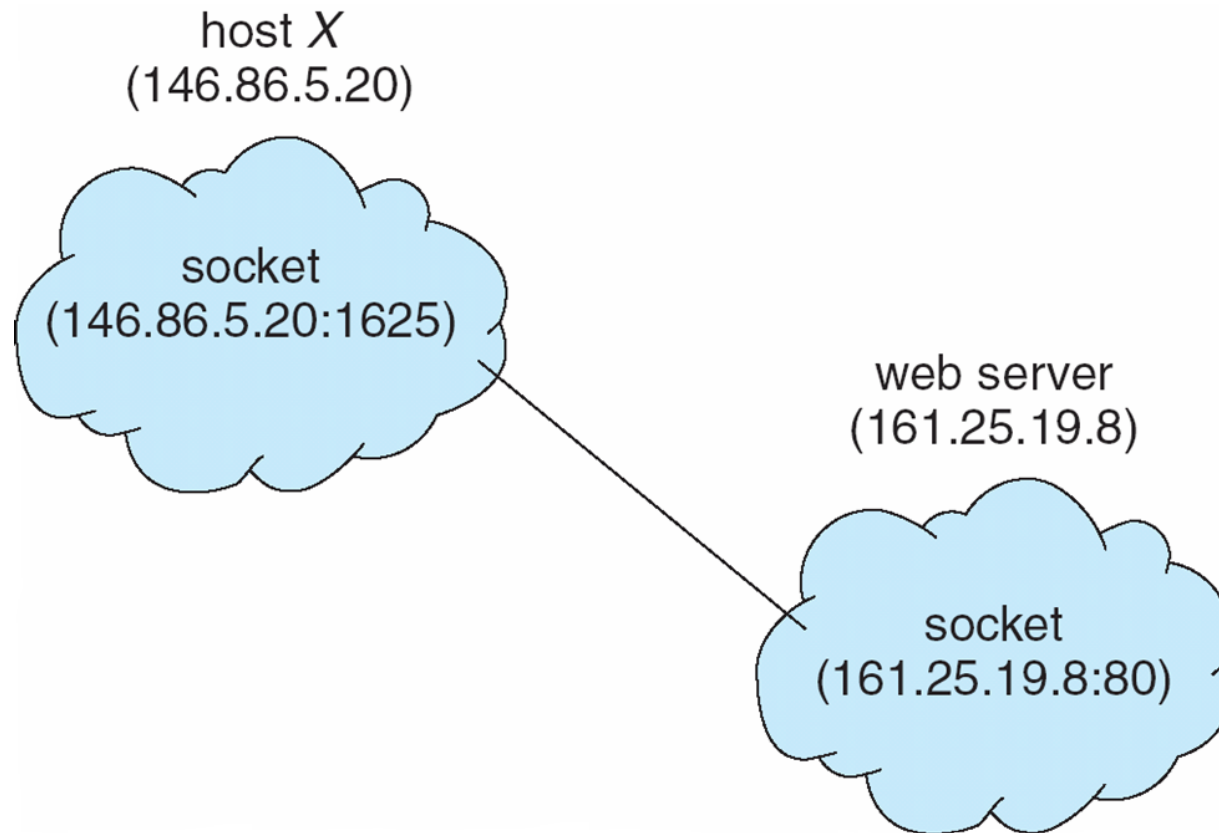
- Socket
- Remote Procedure Calls



Sockets

- **Socket**: 一种通信的端点
 - 通过网络通信的进程需要各持有至少一个socket
- **Socket**由**IP**地址和一个端口号**port**组成
 - 这些信息包含于消息报文中
 - 用于区分在同一个主机上的不同网络服务
 - 例如**socket: 161.25.19.8:1625** 指主机**161.25.19.8**上的**port 1625**
- 低于**1024**的**port**是保留用于标准服务，用户自定义的应该使用高于**1024**的号。
- 一个特殊**IP**: **127.0.0.1**，指向进程所运行的所在主机地址

Socket Communication





Sockets in Java

- 三种不同的Socket sockets
 - 面向有连接, Connection-oriented (TCP)
 - 面向无连接, Connectionless (UDP)
 - 组播, MulticastSocket

```
import java.net.*;
import java.io.*;

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);
        }
    }
}
```



Remote Procedure Calls

■ RPC

- 抽象了在网络通信系统之上的进程之间的过程调用
- 仍然使用了port作为不同服务的代表

■ 存根 Stubs

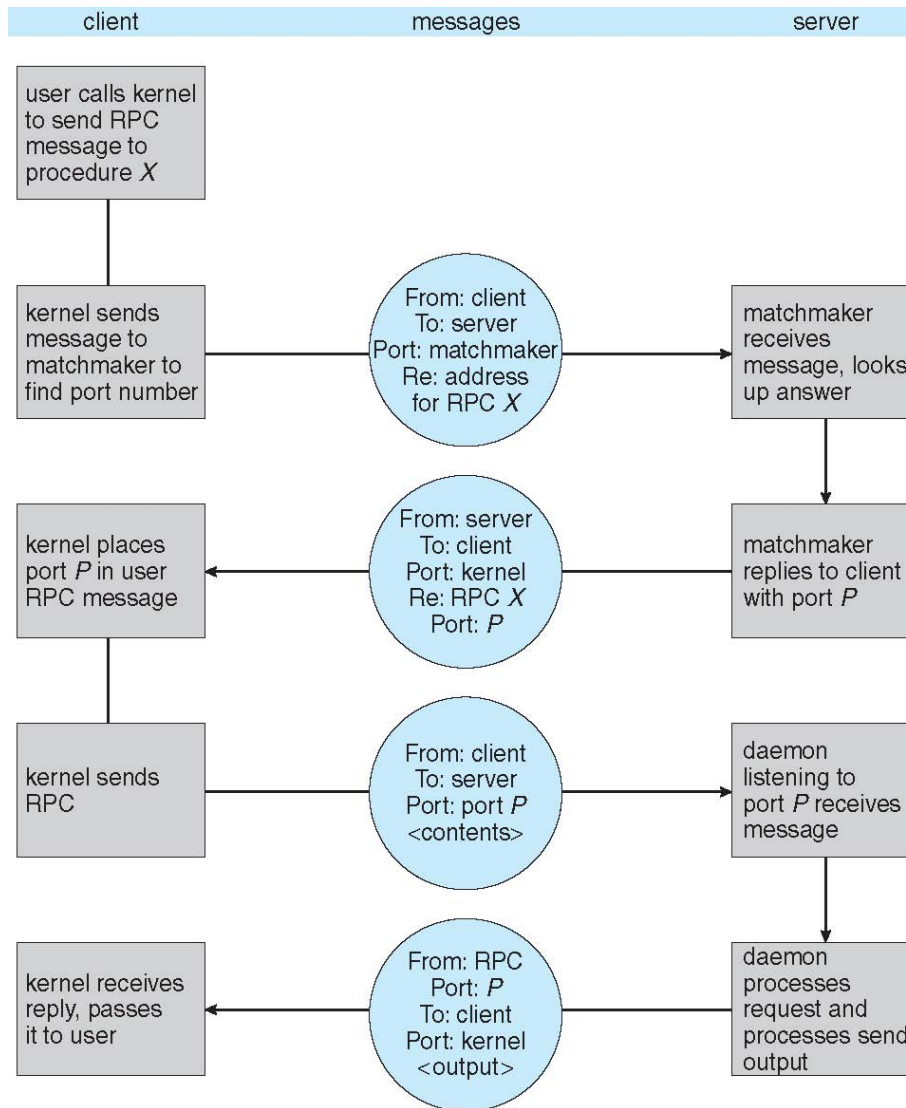
- 在服务器上的客户端的过程代理
- 客户端的stub位于服务器端，并且负责编组参数
- 服务器端的stub则负责接收消息，解包编组后的参数，并在服务器端执行过程
- Windows: stub以Microsoft Interface Definition Language (MIDL)规范编写。



Remote Procedure Calls (Cont.)

- 数据表示
 - 机器无关: External Data Representation (XDL)
 - 如Big-endian and little-endian
- 过程调用的语义问题: 如何提高避错能力?
 - 最多一次: 时间戳协议, 处理重传时重复的消息
 - 刚好一次: 客户端要周期性重发ACK消息, 直到收到了调用的ACK
- 如何让Client知道Server的具体通信端口?
 - 采用集合点服务程序(matchmaker)

RPC执行过程





作业-2

- 小作业-2
 - v7: 3.10
- 大作业-2（2选1）
 - 选择1:v7版本: UNIX Shell and History Feature
 - 完成进度: 3周完成
 - 选择2:分析AFL源码中共享内存和pipe进程通信的实现方法
 - AFL <http://lcamtuf.coredump.cx/afl/>
 - 报告要求: 撰写进程间通信的机理, 画出流程图, 并尝试完成一个真实可执行文件的测试
 - 完成进度: 4周时间完成