上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

# L2-2. 进程间通信

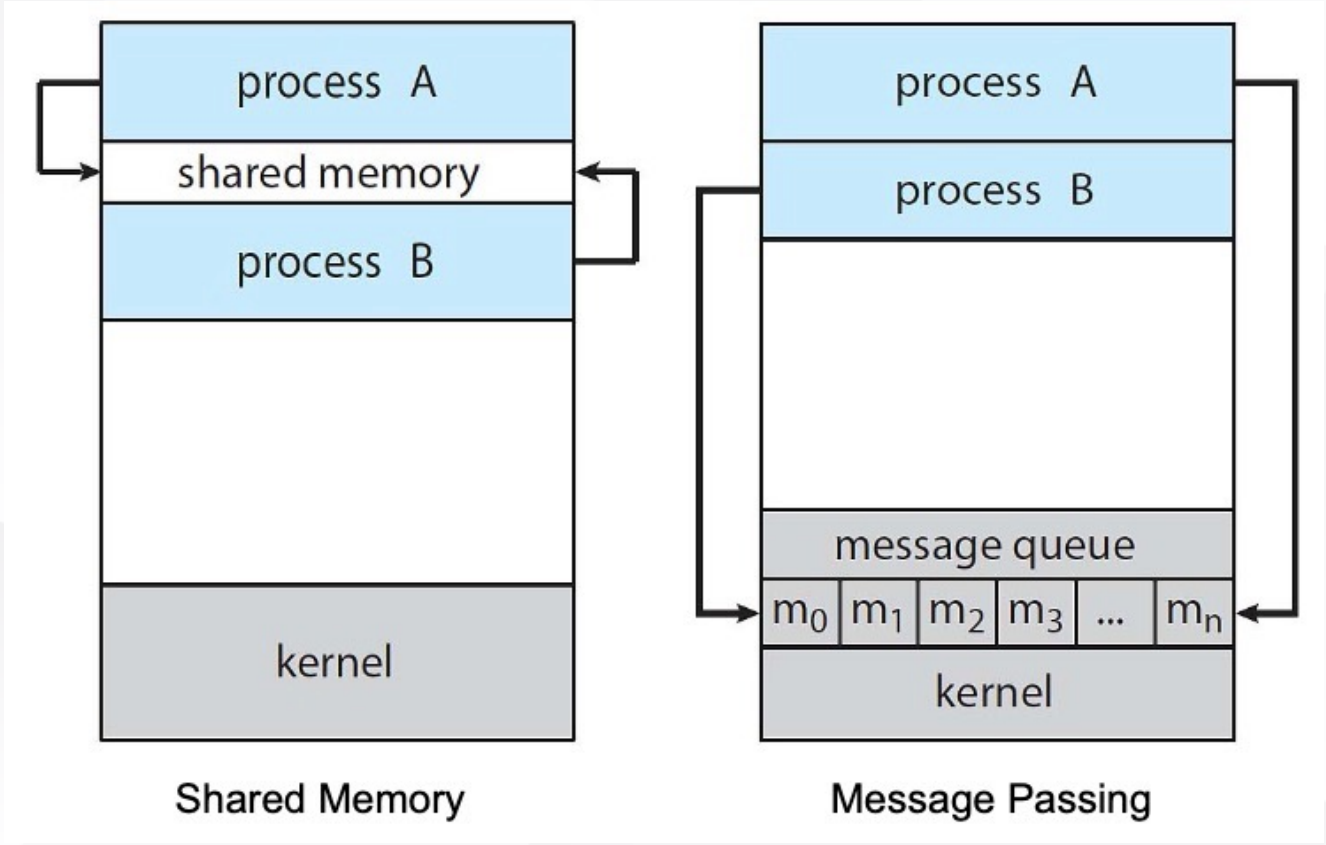

**宋卓然**

**上海交通大学计算机系**

*songzhuoran@sjtu.edu.cn*

# 进程间通信

- 进程间可以独立也可以协作
- 需要让进程协作的原因：
  - 信息共享
  - 计算加速
  - 模块化
  - 方便
- 协作方式：
  - 共享内存
  - 消息传递



# 进程间通信

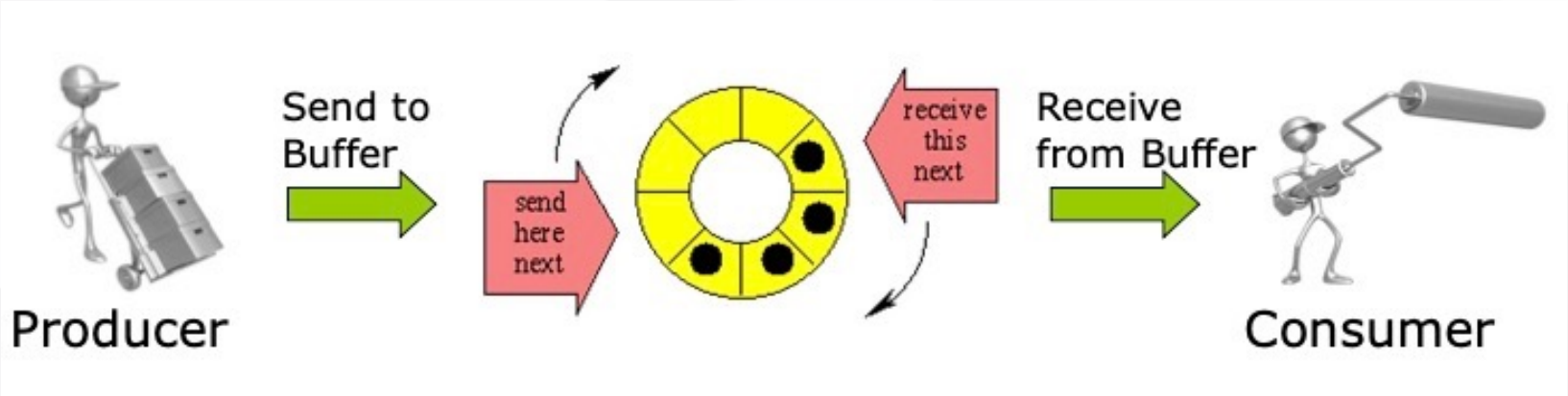
- 共享内存
  - 快于消息传递
- 消息传递
  - 适用于交换较少数量的数据， 因为无需避免冲突
  - 需要采用系统调用，速度较慢
- 对具有多核系统的研究表明：消息传递的性能更好，因为共享内存会存在高速缓存一致性的问题（更改同一位置的数据）

# 共享内存系统

- 采用共享内存的进程间通信，需要通信进程建立共享内存区域
- 一片共享内存区域驻留在创建共享内存段的进程地址空间内，其他希望使用这个共享内存段进行通信的进程应将其附加到自己的地址空间
- 通常操作系统会阻止一个进程访问另一个进程的内存（保证正确性、安全）
- 共享内存需要多个进程同意取消这一限制，通过在共享区域读写数据来交换信息
- 共享内存需确保：
  - 进程不向同一位置同时写入数据

# 共享内存系统 生产者消费者问题

- 生产者生产信息，消费者消费信息
  - 有一个可用的缓冲区，以被生产者填充和被消费者清空
- 缓冲区
  - 无界缓冲区，不限制缓冲区大小
  - 有界缓冲区，假设固定大小的缓冲区



# 共享内存系统 生产者消费者问题

- 有界缓冲区
- in指向缓冲区的下一个空位；out指向缓冲区的第一个满位
  - 当in==out，缓冲区为空
  - 当(in+1)%BUFFER_SIZE==out时，缓冲区为满

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

# 共享内存系统 生产者消费者问题

Producer

```
while (true) {
  /* Produce an item */
 while (((in + 1) % BUFFER_SIZE)  == out)
    ;   /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
while (true) {
  while (in == out)
        ; // do nothing
    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
      return item;
}
```

# 进程间通信-消息传递

- 消息传递提供一种通信机制，以便允许进程不必通过共享地址空间来实现通信和同步
- 通过调用原语进行进程间通信
  - send(message)
  - receive(message)
- 如果进程P和Q需要通信， 那么它们必须互相发送消息和接收消息，它们之间要有通信链路

# 进程间通信-消息传递

- 有几个方法，用于逻辑实现链路和操作send()、receive()
  - 直接或间接的通信
  - 同步或异步的通信
  - 自动或显式的缓冲

# 进程间通信-消息传递

- 直接通信：需要通信的每个进程必须明确指定通信的接收者或发送者
  - send(P, message):向进程P发送message
  - receive(Q, message):从进程Q接收message
- 属性
  - 在需要通信的每对进程之间，自动建立链路。进程仅需知道对方身份就可进行交流
  - 每个链路只与两个进程相关
  - 每对进程之间只有一个链路

# 进程间通信-消息传递

- 间接通信：通过邮箱或端口来发送和接收消息
  - send(A, message):向邮箱A发送message
  - receive(A, message):从邮箱A接收message
- 属性
  - 只有在两个进程共享一个邮箱时，才能建立通信链路
  - 一个链路可以与两个或更多进程相关联
  - 两个通信进程之间可有多个不同链路，每个链路对应于一个邮箱

# 进程间通信-消息传递

- 消息传递可以是阻塞或非阻塞的，也称为同步或异步的
  - 阻塞发送：发送进程阻塞，直到消息由接收进程或邮箱所接收
  - 非阻塞发送：发送进程发送消息，并且恢复操作
  - 阻塞接收：接收进程阻塞，直到有消息可用
  - 非阻塞接收：接收进程收到一个有效消息或空消息

# 进程间通信 实例 POSIX系统使用共享内存

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;
```

```
   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

采用POSIX共享内存API的生产者进程

# 进程间通信 实例 POSIX共享内存

```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

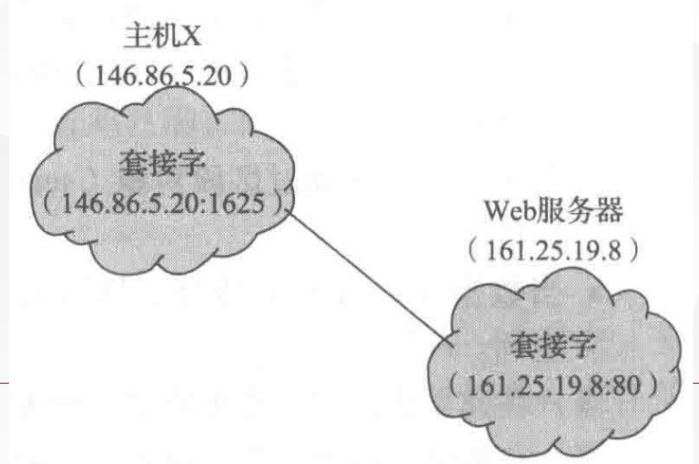
采用POSIX共享内存API的消费者进程

# 客户机/服务器通信

- 客户机与服务器之间通信的三种策略
  - 套接字（socket）
  - 远程程序调用（RPC）
  - 管道

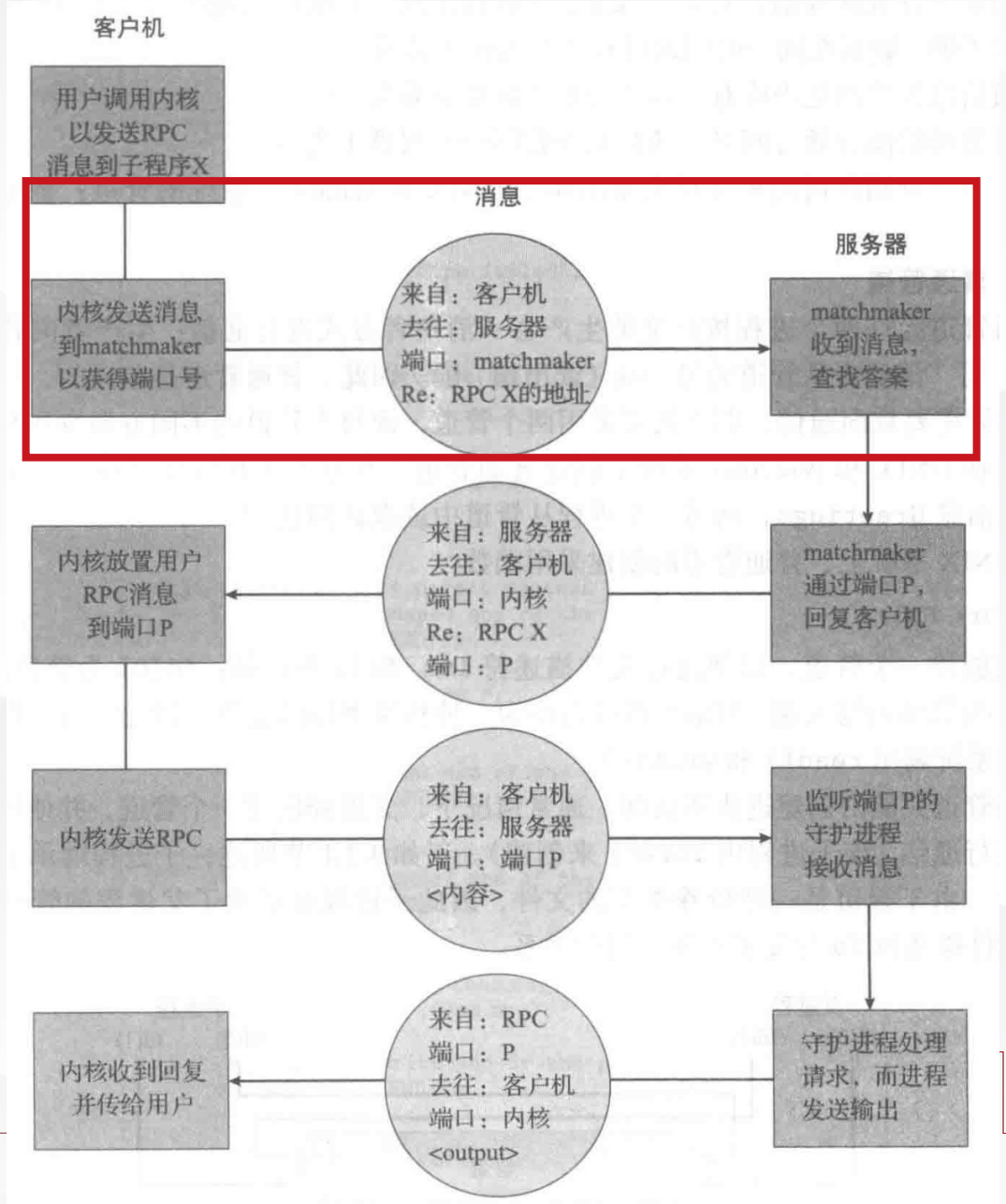
# 套接字

- 通过网络通信的每对进程需要使用一对套接字，即每个进程有一个，每个套接字具有一个IP地址和一个端口
- 套接字采用客户端-服务器架构，服务器通过监听指定端口，来等待客户请求。服务器在收到请求后，接受来自客户套接字的连接，从而完成连接
- 当客户进程发出连接请求时，它的主机为它分配一个端口，**端口唯一，连接唯一**
  - 一般情况下，客户端会随机选择一个未被使用的本地端口号，作为与服务器建立连接的端口。这样可以避免端口冲突，并提高系统的安全性和可靠性



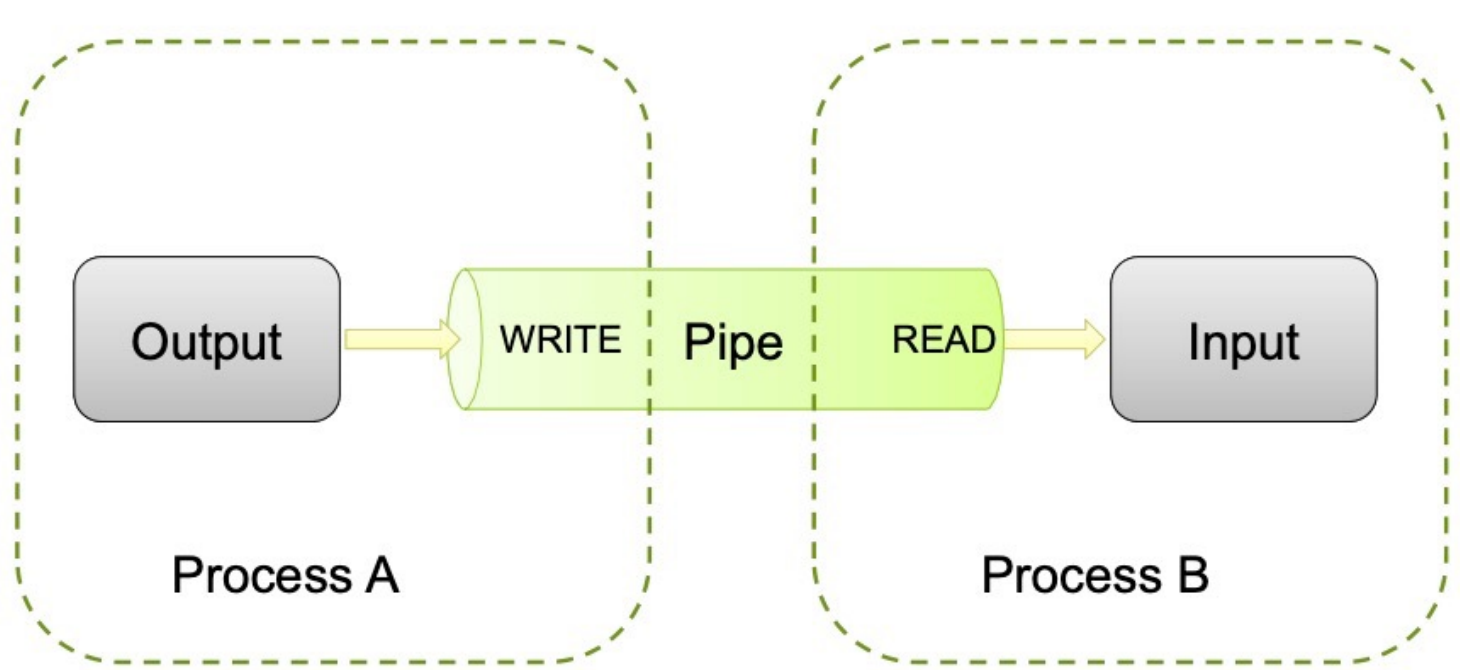
# 远程程序调用（RPC）

- RPC是一种常见的远程服务
- 服务器使用端口（port）接收消息，提供服务
  - 端口预先固定
  - 操作系统通过提供交会服务程序或月老（matchmaker）提供端口号



# 普通管道

- 普通管道允许两个进程进行单向通信
  - 早期UNIX系统最早使用的一种通信机制
  - 一端写
  - 一端读

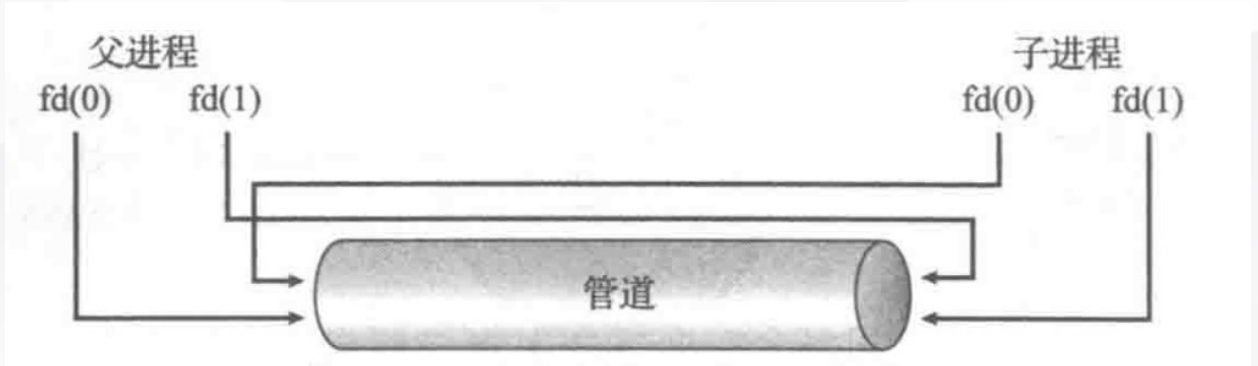


# 普通管道 UNIX实例

- 在UNIX 系统上，普通管道的创建采用函数

  pipe(int fd[])

- fd[0]为管道的读出端， 而fd[1]为管道的写入端
- 访问管道采用系统调用read()和write()
- 通常情况下，父进程创建一个管道，并使用它来与其子进程进行通信(该子进程由fork()来创建)
  - 父进程需关闭读端口，子进程关闭写端口



# 普通管道 UNIX实例

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

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

    /* Program continues in 3.26 */
```

# 普通管道 UNIX实例

```
/* create the pipe */
if (pipe(fd) == -1) {
   fprintf(stderr,"Pipe failed");
   return 1;
}

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

if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
}

if (pid > 0) { /* parent process */
   /* close the unused end of the pipe */
   close(fd[READ_END]);

   /* write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

   /* close the read end of the pipe */
   close(fd[WRITE_END]);
}
else { /* child process */
   /* close the unused end of the pipe */
   close(fd[WRITE_END]);

   /* read from the pipe */
   read(fd[READ_END], read_msg, BUFFER_SIZE);
   printf("read %s",read_msg);

   /* close the write end of the pipe */
   close(fd[READ_END]);
}

return 0;
}
```

# 普通管道 Windows实例

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;

        /* Program continues in 3.28 */
```

Windows系统管道的父进程

# 普通管道 Windows实例

```
/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
   fprintf(stderr, "Create Pipe Failed");
   return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);
```

```
/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
  TRUE, /* inherit handles */
  0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
   fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Windows系统管道的父进程

# 普通管道 Windows实例

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

   /* get the read handle of the pipe */
   ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

   /* the child reads from the pipe */
   if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
      printf("child read %s",buffer);
   else
      fprintf(stderr, "Error reading from pipe");

   return 0;
}
```

Windows系统管道的子进程

# 命名管道

- 命名管道更为强大，允许双向通信，父子进程非必需
- 对于UNIX系统，命名管道为FIFO。一旦创建，它们表现为文件系统的典型文件。通过系统调用mkfifo()，可以创建FIFO
- 对于UNIX系统，通过系统调用open()、read()、write()和close()，可以操作FIFO
- 对于Windows系统，命名管道通信机制更丰富，允许全双工通信，且通信进程可以位于同一或不同机器