

系列课程 —Linux系统编程

第七章

进程通讯

讲师:任继梅

QQ : 59189174

课程目标

✓掌握操作系统的基本原理

- 计算机工作原理、多任务管理、并发与竞态

✓掌握操作系统提供的常用API

- 基本IO、内存映射

✓多进程编程

- 进程管理、进程间通信

✓多线程编程

- 线程管理、线程同步

课程安排

✓ 第一天

上午：计算机系统概述

下午：基本I/O操作

✓ 第二天

上午：Linux进程

下午：信号

✓ 第三天

上午：共享文件

下午：并发和竞争

✓ 第四天

上午：Linux线程

下午：线程同步

✓ 第五天

上午：共享内存、消息队列 下午：管道

课前提问

1. 如何在进程间传递信息？
2. 线程间如何传递信息？
3. 线程间共享数据如何处理竞态问题？
4. 进程间共享资源如何处理竞态问题？

本章目标

✓ 进程间的通讯方式



✓ 管道



✓ IPC对象



✓ 共享内存



✓ 消息队列



✓ 信号灯



✓ 系统编程总结



第一节

进程间的通讯

进程间的通讯关系

UNIX平台进程通信

- IPC: Inter-Process Communications

- 不同进程间传递、共享信息或提供服务的方式

进程间的通讯关系

UNIX平台进程通信方式

- ❖ 早期进程间通信方式
- ❖ AT&T的贝尔实验室，对Unix早期的进程间通信进行了改进和扩充，形成了“system V IPC”，其通信进程主要局限在单个计算机内
- ❖ BSD(加州大学伯克利分校的伯克利软件发布中心)，跳过了该限制，形成了基于套接字(socket)的进程间通信机制
- ❖ Linux继承了上述所有的通信方式

进程间的通讯关系

常用的进程间通信方式

传统的进程间通信方式

无名管道(pipe)、有名管道(fifo)和信号(signal)

System V IPC对象

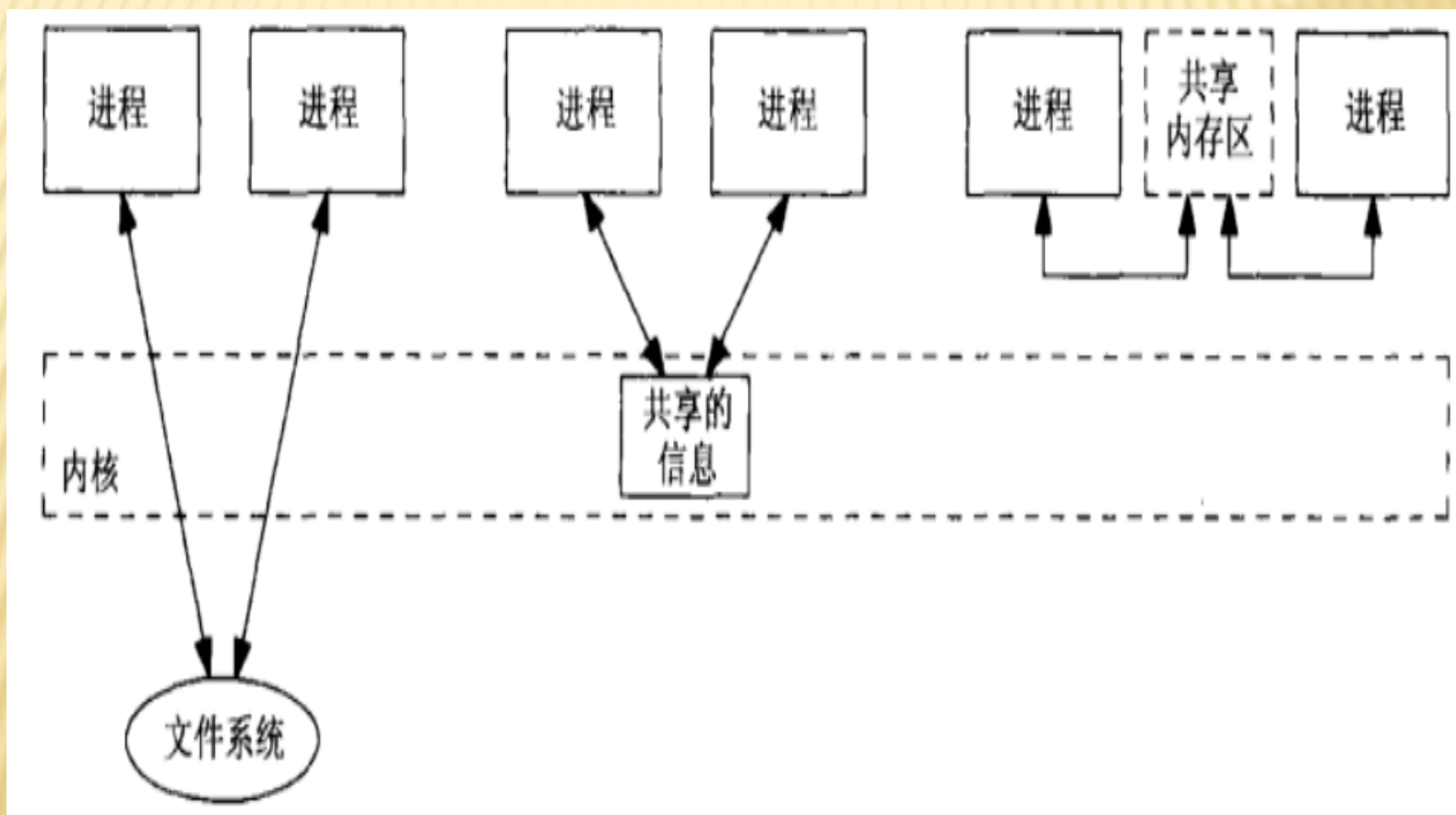
共享内存(share memory)、消息队列(message queue)和信号灯(semaphore)

BSD

套接字(socket)

IPC对象

进程间共享信息的实现方式

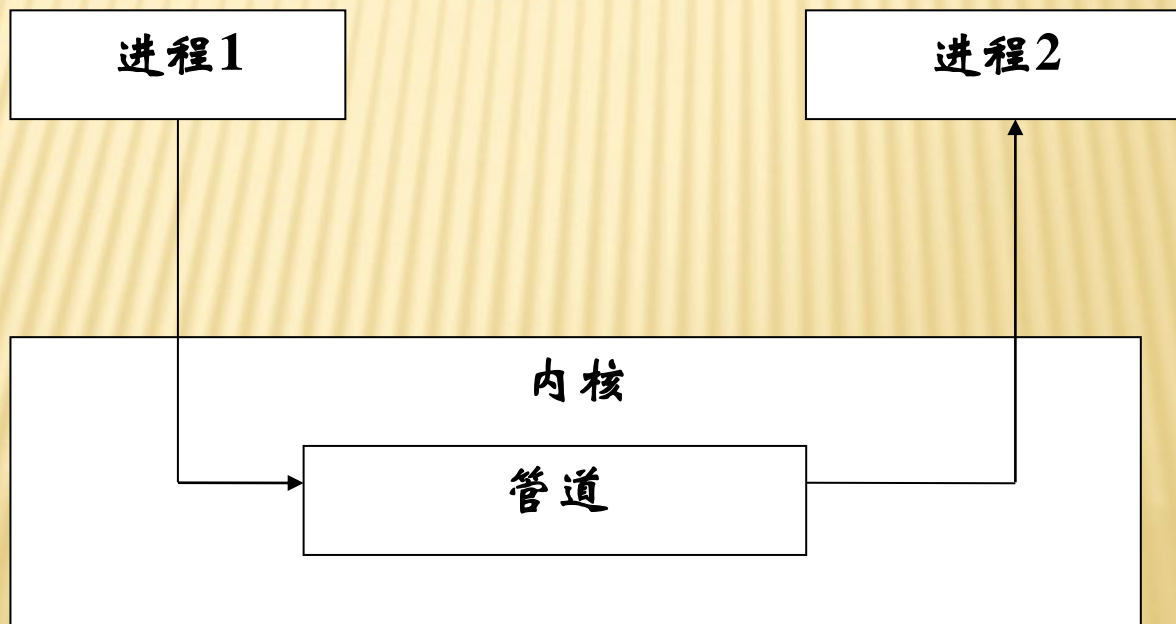


第一节 管道

管道

无名管道

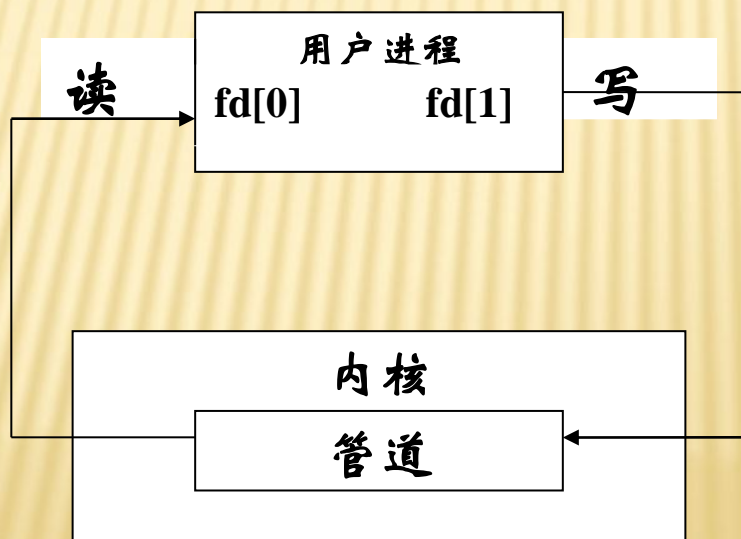
- 只能用于具有亲缘关系的进程之间的通信
- 半双工的通信模式，具有固定的读端和写端
- 管道可以看成是一种特殊的文件，对于它的读写可以使用文件IO如read、write函数。



管道-无名管道的创建和关闭

人 无名管道

- 管道是基于文件描述符的通信方式。当一个管道建立时，它会创建两个文件描述符`fd[0]`和`fd[1]`。其中`fd[0]`固定用于读管道，而`fd[1]`固定用于写管道。
- 构成了一个半双工的通道。



管道-无名管道的创建和关闭

创建无名管道

```
#include <unistd.h>
```

```
int pipe(int pipefd[2]);
```

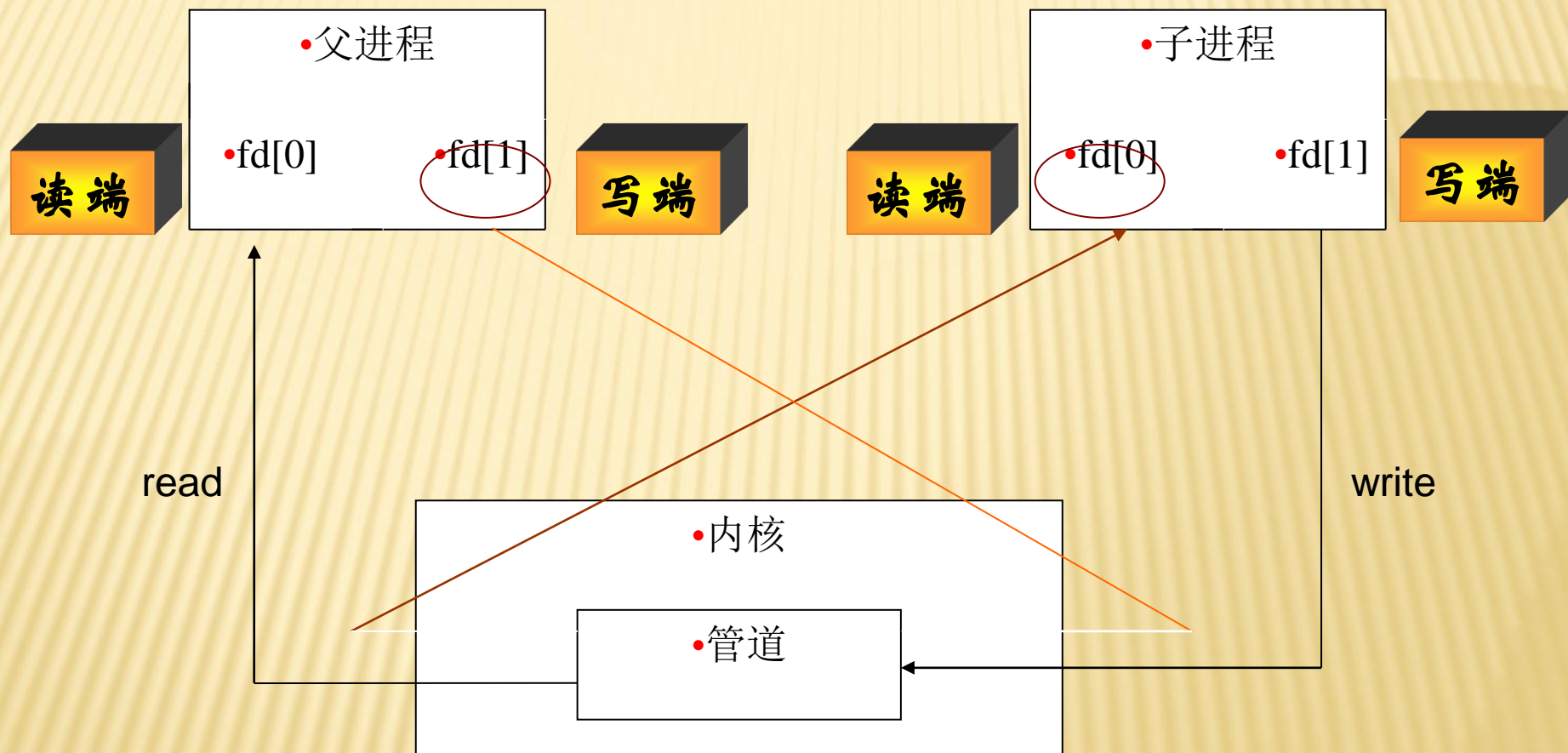
- pipe: 无名管道, 用于父子进程之间的通信

- fd[0] 用于读

- fd[1] 用于写

- 成功返回0, 失败返回-1

管道-无名管道读写



管道-无名管道例子：

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

struct stu
{
    int id;
    char name[32];
    int age;
    char addr[64];
};

int main()
{
    int ret;
    int stutas;
    char buf[32] = {0};
    pid_t pid;

    struct stu s;
```

管道-无名管道例子：

```
//用于存放pipe这个函数返回的文件描述符
int fd[2];

//pipe()函数
//创建一个管道，并返回管道读写的文件描述符到fd中
//fd[0] 用于读
//fd[1] 用于写
ret = pipe(fd);
if(ret < 0)
{
    perror("pipe()");
    return -1;
}

pid = fork();
if(pid == 0)
{
    //write
    s.id = 1;
    strcpy(s.name, "张三");
    s.age = 22;
    strcpy(s.addr, "上海市");
}
```


管道-无名管道例子：

```
    write(fd[1], &s, sizeof(s));
}
else if(pid > 0)
{
    printf("start: stu: id = %d, name = %s, age = %d, \
        addr = %s\n", s.id, s.name, s.age, s.addr);

    ret = read(fd[0], &s, sizeof(s));
    //buf[ret] = '\0';

    printf("end: stu: id = %d, name = %s, age = %d, \
        addr = %s\n", s.id, s.name, s.age, s.addr);

    wait(&stutas);
}
else{
    perror("fork()");
    return -1;
}
close(fd[0]);
close(fd[1]);

return 0;
}
```

管道-无名管道读写

人 无名管道读写

- ❏ 当管道中无数据时，读操作会阻塞
- ❏ 向管道中写入数据时，linux将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读走管道缓冲区中的数据，那么写操作将会一直阻塞。
- ❏ 只有在管道的读端存在时，向管道中写入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的SIGPIPE信号(通常Broken pipe错误)。

管道-有名管道

有名管道

- ❑ 无名管道只能用于具有亲缘关系的进程之间，这就限制了无名管道的使用范围
- ❑ 有名管道可以使互不相关的两个进程互相通信。有名管道可以通过路径名来指出，并且在文件系统中可见
- ❑ 进程通过文件IO来操作有名管道
- ❑ 有名管道遵循先进先出规则
- ❑ 不支持如lseek() 操作

管道-有名管道

创建有名管道

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

```
int mkfifo(const char *pathname, mode_t mode);
```

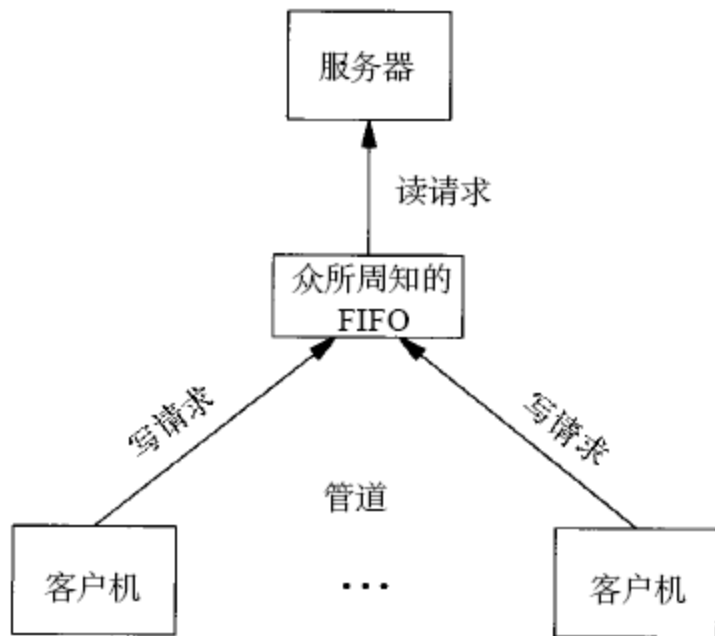
❏ pathname: 路径

❏ mode: 访问权限

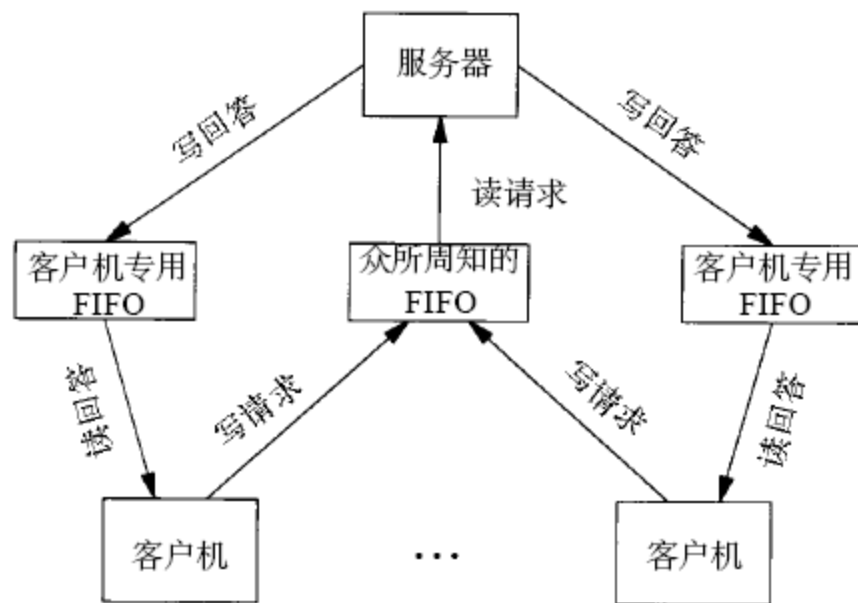
❏ 成功返回0，失败返回负数

- ❏ EACCESS 参数filename所指定的目录路径无可执行的权限
- ❏ EEXIST 参数filename所指定的文件已存在
- ❏ ENAMETOOLONG 参数filename的路径名称太长
- ❏ ENOENT 参数filename包含的目录不存在
- ❏ ENOSPC 文件系统的剩余空间不足
- ❏ EROFS 参数filename指定的文件存在于只读文件系统内

管道-有名管道典型模型



客户机用FIFO向服务器发送请求



客户机-服务器用FIFO进行通信

管道-有名管道例子:

发送例子:

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

#define ABPATH "/tmp/AtoB"
#define BAPATH "/tmp/BtoA"

void* func(void *param)
{
    char buf[32] = {0};
    int fd;
    fd = open(BAPATH, O_RDONLY);
    if(fd < 0)
    {
        perror("open()");
        exit(-1);
    }
}
```


管道-有名管道例子：

```
//重复的往管道里面写入
while(1)
{
    read(fd, buf, sizeof(buf));
    printf("对方说:%s\n", buf);
}

close(fd);
}

int main()
{
    int fd;
    int ret;
    char buf[32];

    if(access(ABPATH, F_OK) == -1)
    {
        //创建管道文件
        ret = mkfifo(ABPATH, 0777);
        if(ret < 0)
        {
            perror("mkfifo()");
        }
    }
}
```

管道-有名管道例子：

```
if(access(BAPATH, F_OK) == -1)
{
    //创建管道文件
    ret = mkfifo(BAPATH, 0777);
    if(ret < 0)
    {
        perror("mkfifo()");
    }
}

pthread_t tid;
pthread_create(&tid, NULL, func, NULL);

//open相当于两个进程连接起来，如果只有一个进程打开此管道文件
// 则在此阻塞等待另一个进程打开文件
fd = open(ABPATH, O_WRONLY);
if(fd < 0)
{
    perror("open()");
    return -1;
}
```

管道-有名管道例子：

```
//重复的往管道里面写入
while(1)
{
    printf(">");
    fflush(stdout);
    fgets(buf, sizeof(buf), stdin);
    write(fd, buf, sizeof(buf));
}

close(fd);

return 0;
}
```

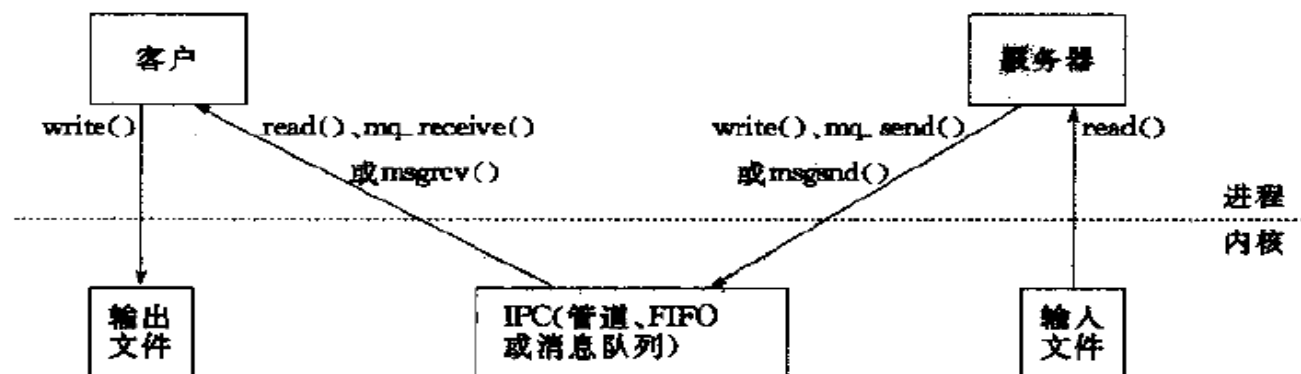

第三节

内存共享

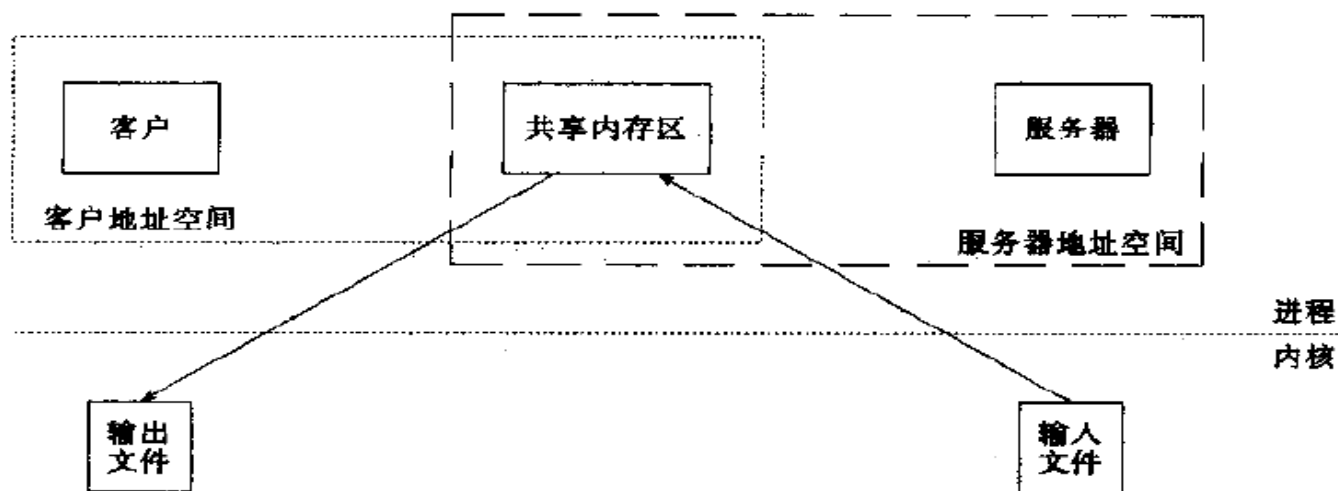
内存共享

- 共享内存是一种最为高效的进程间通信方式，进程可以直接读写内存，而不需要任何数据的拷贝
- 为了在多个进程间交换信息，内核专门留出了一块内存区，可以由需要访问的进程将其映射到自己的私有地址空间
- 进程就可以直接读写这一内存区而不需要进行数据的拷贝，从而大大提高的效率。
- 由于多个进程共享一段内存，因此也需要依靠某种同步机制，如互斥锁和信号量等

内存共享



从服务器到客户的文件数据流



使用共享内存区从服务器拷贝文件数据到客户

内存共享的实现

 共享内存的使用包括如下步骤：

- ❏ 创建/打开共享内存
- ❏ 映射共享内存，即把指定的共享内存映射到进程的地址空间用于访问
- ❏ 撤销共享内存映射
- ❏ 删除共享内存对象

内存共享的实现

获取共享内存

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

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

- key: IPC_PRIVATE 或 ftok的返回值
- size: 共享内存区大小
- shmflg: 同open函数的权限位，也可以用8进制表示法
- 成功: 共享内存段标识符，出错: -1

内存共享的实现

绑定与分离共享内存

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

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

```
int shmdt(const void *shmaddr);
```

- ❏ shmid: 要映射的共享内存区标识符
- ❏ shmaddr: 将共享内存映射到指定地址(若为NULL, 则表示由系统自动完成映射)
- ❏ shmflg :SHM_RDONLY: 共享内存只读
 - ❏ 默认0: 共享内存可读写
- ❏ 成功: 映射后的地址, 出错: -1

内存共享的实现

控制共享内存

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

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- shmid: 要操作的共享内存标识符
- cmd: IPC_STAT (获取对象属性)
IPC_SET (设置对象属性)
IPC_RMID (删除对象)
- buf: 指定IPC_STAT/IPC_SET时用以保存/设置属性
- 成功: 0, 出错: -1

内存共享的例1：

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

int main()
{
    int segment_id;
    char* shared_memory;
    struct shmid_ds shmbuffer;
    key_t key = 0x1234;

    /* 分配一个共享内存块 */
    const int shared_size = 0x6400;

    /* 绑定到共享内存块 */
    segment_id = shmget(key, shared_size, IPC_CREAT |
                                                                IPC_EXCL | S_IRUSR | S_IWUSR );

    /* 确定共享内存的大小 */
    shared_memory = (char*)shmat(segment_id, 0, 0);
    printf("shared memory attached at address %p\n", shared_memory);
}
```

内存共享的例1：

```
shmctl(segment_id, IPC_STAT, &shmbuffer);

printf("segment size: %ld\n", shmbuffer.shm_segsz);

/* 在共享内存中写入一个字符串 */
sprintf(shared_memory, "Hello, world.");

/* 脱离该共享内存块 */
shmdt(shared_memory);

/* 重新绑定该内存块 */
shared_memory = (char*)shmat(segment_id, 0, 0);

/* 输出共享内存中的字符串 */
printf("shared memory at address %p\n", shared_memory);
printf("%s\n", shared_memory);

/* 脱离该共享内存块 */
shmdt(shared_memory);
/* 释放这个共享内存块 */
shmctl(segment_id, IPC_RMID, 0);

return 0;
}
```


内存共享的例2:

```
int main()
{
    int segment_id;
    char* shared_memory;
    key_t key = 0x1234;
    /* 分配一个共享内存块 */
    const int shared_segment_size = 0x6400;

    /* 绑定到共享内存块 */
    segment_id = shmget(key, shared_segment_size, IPC_CREAT|
        IPC_EXCL|S_IRUSR|S_IWUSR );

    /* 确定共享内存的大小 */
    shared_memory = (char*)shmat(segment_id, 0, 0);
    printf("shared memory attached at address %p\n", shared_memory);

    /* 在共享内存中写入一个字符串 */
    sprintf(shared_memory, "Hello, world.");

    /* 脱离该共享内存块 */
    shmdt(shared_memory);

    return 0;
}
```

内存共享的例2：

```
int main()
{
    int shmid;
    char* shared_memory;
    key_t key = 0x1234;
    /* 分配一个共享内存块 */
    const int shm_size = 0x6400;

    shmid = shmget(key, shm_size, IPC_CREAT | S_IRUSR | S_IWUSR);

    /* 重新绑定该内存块 */
    shared_memory = (char*)shmat(shmid, 0, 0);
    printf("shared memory at address %p\n", shared_memory);

    /* 输出共享内存中的字符串 */
    printf("%s\n", shared_memory);

    /* 脱离该共享内存块 */
    shmdt(shared_memory);
    /* 释放这个共享内存块 */
    shmctl(shmid, IPC_RMID, 0);

    return 0;
}
```

```
struct shmid_ds{
    struct ipc_perm shm_perm; /* 操作权限*/
    int shm_segsz;           /*段的大小（以字节为单位）*/
    time_t shm_atime;        /*最后一个进程附加到该段的时间*/
    time_t shm_dtime;        /*最后一个进程离开该段的时间*/
    time_t shm_ctime;        /*最后一个进程修改该段的时间*/
    unsigned short shm_cpid; /*创建该段进程的pid*/
    unsigned short shm_lpid; /*在该段上操作的最后1个进程的
pid*/
    short shm_nattch;        /*当前附加到该段的进程的个数*/
/*下面是私有的*/
    unsigned short shm_npages; /*段的大小（以页为单位）*/
    unsigned long *shm_pages; /*指向frames->SHMMAX的指针数
组*/
    struct vm_area_struct *attaches; /*对共享段的描述*/
};
```


第三节

消息队列

消息队列

- 消息队列是IPC对象的一种
- 消息队列由消息队列ID来唯一标识
- 消息队列就是一个消息的列表。用户可以在消息队列中添加消息、读取消息等。
- 消息队列可以按照类型来发送/接收消息

消息队列

- 消息队列的操作包括创建或打开消息队列、添加消息、读取消息和控制消息队列
- 创建或打开消息队列使用的函数是`msgget`，这里创建的消息队列的数量会受到系统消息队列数量的限制
- 添加消息使用的函数是`msgsnd`，按照类型把消息添加到已打开的消息队列末尾
- 读取消息使用的函数是`msgrcv`，可以按照类型把消息从消息队列中取走
- 控制消息队列使用的函数是`msgctl`，它可以完成多项功能。

消息队列

👤 控制共享内存

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

```
int msgget(key_t key, int msgflg);
```

- ❏ key: 和消息队列关联的key值
- ❏ flag: 消息队列的访问权限
- ❏ 成功: 消息队列ID, 出错: -1

消息队列

控制共享内存

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

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

- ❖ msqid: 消息队列的ID
- ❖ msgp: 指向消息的指针。
- ❖ size: 发送的消息正文的字节数
- ❖ flag: IPC_NOWAIT 消息没有发送完成函数也会立即返回。
 - ❖ 0: 直到发送完成函数才返回
 - ❖ 成功: 消息队列ID, 出错: -1

消息队列

控制共享内存

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

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
               int msgflg);
```

- ❑ msqid: 消息队列的ID
- ❑ msgp: 指向消息的指针。
- ❑ size: 要接收的消息的字节数
- ❑ msgtype: 0: 接收消息队列中第一个消息。
 - ❑ 大于0: 接收消息队列中第一个类型为msgtyp的消息。
 - ❑ 小于0: 接收消息队列中类型值不小于msgtyp的绝对值且类型值又最小的消息。
- ❑ flag: 0: 若无消息函数会一直阻塞
 - ❑ IPC_NOWAIT: 若没有消息, 进程会立即返回ENOMSG。
- ❑ 成功: 接收到的消息的长度, 出错: -1

消息队列

👤 控制共享内存

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

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- ❏ msqid: 消息队列的队列ID
- ❏ cmd:
 - ❏ IPC_STAT: 读取消息队列的属性，并将其保存在buf指向的缓冲区中。
 - ❏ IPC_SET: 设置消息队列的属性。这个值取自buf参数。
 - ❏ IPC_RMID: 从系统中删除消息队列。
- ❏ buf: 消息队列缓冲区
- ❏ 成功: 0, 出错: -1

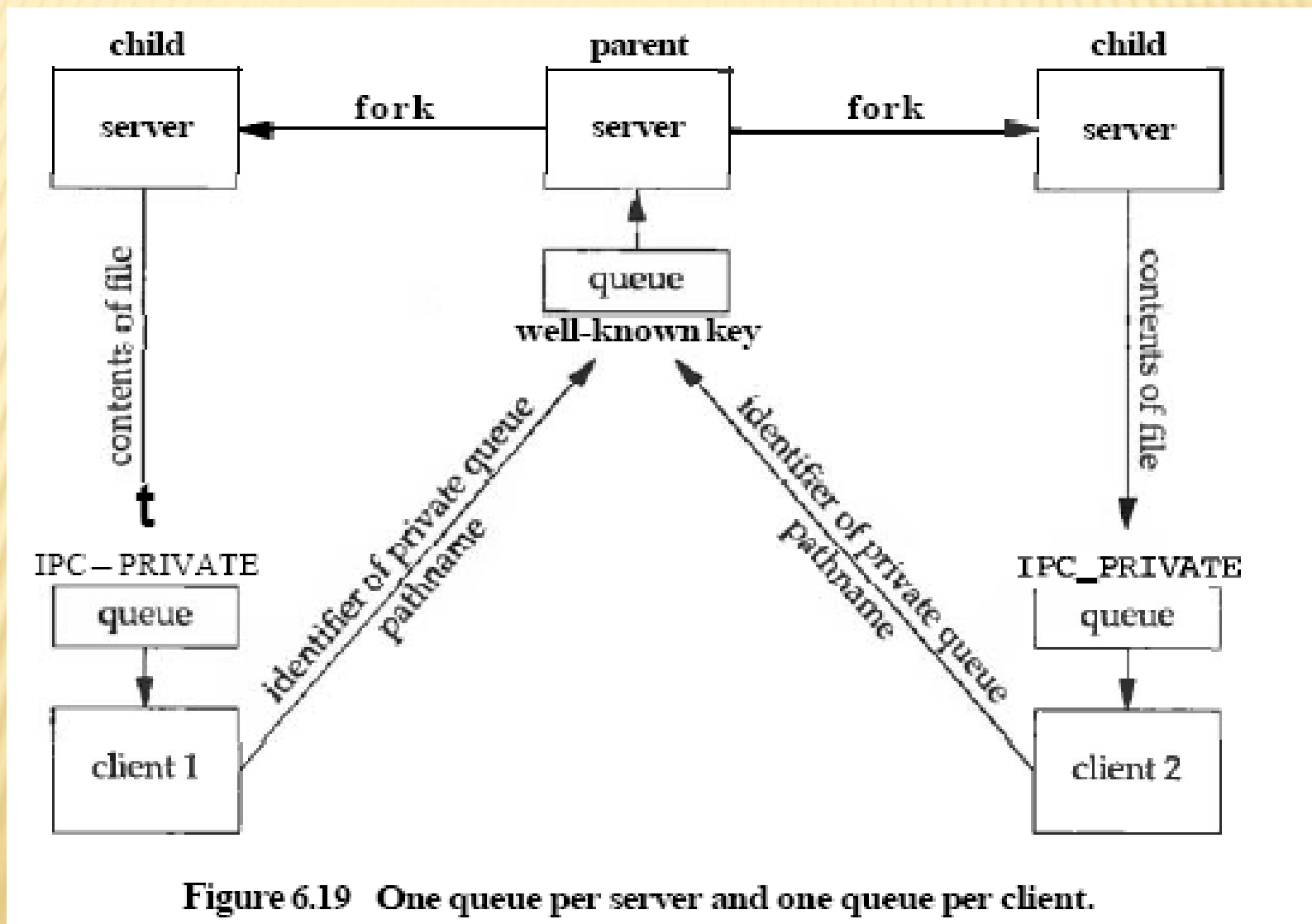
消息队列

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

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

```
struct msqid_ds {
    struct ipc_perm msg_perm;      /* Ownership and permissions */
    time_t          msg_stime;     /* Time of last msgsnd(2) */
    time_t          msg_rtime;     /* Time of last msgrcv(2) */
    time_t          msg_ctime;     /* Time of last change */
    unsigned long   __msg_cbytes;  /* Current number of bytes in
                                   queue (non-standard) */
    msgqnum_t       msg_qnum;      /* Current number of messages
                                   in queue */
    msglen_t        msg_qbytes;    /* Maximum number of bytes
                                   allowed in queue */
    pid_t           msg_lspid;     /* PID of last msgsnd(2) */
    pid_t           msg_lrpid;    /* PID of last msgrcv(2) */
};
```

典型的基于MSQ的CS模型



消息队列举例

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

```
struct stu
{
    int id;
    char name[32];
};
```

//这个结构体为消息类型，必须定义为第一个成员为long mtype的类型

```
struct msgbuf {
    long mtype;           /* message type, must be > 0 */
    struct stu s;         /* message data */
};
```

消息队列举例

```
int main()
{
    int msgid;
    key_t key = 1234;
    //1.创建消息队列
    msgid = msgget(key, IPC_CREAT | 0644);

    struct msgbuf msg;
    msg.mtype = 1; //设置消息类型, 必须 > 0
    msg.s.id = 1;
    strcpy(msg.s.name, "张三");
    //2.往消息队列发送消息
    msgsnd(msgid, &msg, sizeof(struct stu), IPC_NOWAIT);

    //3.从消息队列里面接收消息
    memset(&msg, 0, sizeof(msg));
    //1 为消息类型
    msgrcv(msgid, &msg, sizeof(struct stu), 1, IPC_NOWAIT);
    printf("msg:[id = %d,name = %s\n", msg.s.id, msg.s.name);

    //4.删除消息队列
    msgctl(msgid, IPC_RMID, NULL);
    return 0;
}
```

消息队列举例

```
#define BUFSZ 512
#define TYPE 100
struct msgbuf{
    long mtype;
    char mtext[BUFSZ];
};
int main()
{
    int qid, len;
    key_t key;
    struct msgbuf msg;

    /*根据不同的路径和关键字表示产生标准的key*/
    if ((key = ftok(".", 'a')) == -1){
        perror("ftok");
        exit(1);
    }
```


消息队列举例

```
/*创建消息队列*/
```

```
if ((qid = msgget(key, IPC_CREAT|0666)) == -1){  
    perror("msgget");  
    exit(-1);  
}
```

```
printf("opened queue %d\n",qid);  
puts("Please enter the message to queue:");
```

```
if ((fgets((&msg)->msg_text, BUFSZ, stdin))= =NULL){  
    puts("no message");  
    exit(-1);  
}
```

```
msg.mtype = TYPE;  
len = strlen(msg.mtext) + 1;
```

消息队列举例

```
/*添加消息到消息队列*/
```

```
    if (msgsnd(qid, &msg, len, 0) < 0){  
        perror("msgsnd");  
        exit(-1);  
    }
```

```
/*从消息队列读取消息*/
```

```
    if (msgrcv(qid, &msg, BUFSZ, 0, 0) < 0){  
        perror("msgrcv");  
        exit(-1);  
    }  
    printf("message is:%s\n", (&msg)->mtext);
```

```
/*从系统中删除消息队列。*/
```

消息队列举例

```
if (msgctl(qid, IPC_RMID, NULL) < 0){  
    perror("msgctl");  
    exit(1);  
}  
return 0;  
}
```

练习：创建一个消息队列，结构体如下：

```
struct msgstru  
{ long msgtype;  
  Char msgtext[2048]; };
```

实现发送进程和接受进程（要求终端输入数据）

知识点6-信号

● 什么是信号？

- ◆ 生活中经商定作为采取一致行动的暗号
- ◆ Unix家族操作系统中：是一种进程间通讯的有限制的方式
- ◆ 是一种异步的通知机制，用来提醒进程一个事件已经发生
- ◆ 信号只是用来通知某进程发生了什么事，并不给该进程传递任何数据
- ◆ 当一个信号发送给一个进程，操作系统中断了进程正常的控制流程，此时，任何非原子操作都将被中断
- ◆ 如果进程定义了信号的处理函数，那么它将被执行，否则就执行默认的处理函数
- ◆ 一个进程可以给其它进程和自己发送信号
- ◆ 内核也可以给所有进程发送信号
- ◆ 进程PCB结构里有一个32位信号，每一位对应一个信号，因此系统支持的信号最多32个，每个信号都有自己的序号，其中序号为0的叫空信号，任何进程都无视空信号

信号

● 进程对收到的信号处理方法：

- ◆ 第一种是类似中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处理。
- ◆ 忽略某个信号，对该信号不做任何处理，就象未发生过一样，这也叫信号屏蔽，这由PCB32位的信号屏蔽成员控制
- ◆ 对该信号的处理保留系统的默认值，这种缺省操作，对大部分的信号的缺省操作是使得进程终止

信号

● 常见的信号1:

收到这些信号的进程采取的默认动作是：异常终止进程

SIGALRM 14	由alarm函数设置的定时器超时时产生
SIGHUP 1	终端断开连接时发送给控制进程或控制进程退出时发给每个前台进程
SIGINT 2	Ctrl+c组合键产生
SIGKILL 9	不能被捕获或忽略，系统管理员一种杀死任何进程的可靠方法
SIGPIPE 13	管道数据没有读进程时产生
SIGTERM 15	Kill命令发送的默认信号，要求进程结束运行
SIGUSR1 10	进程之间可以用来通信，含义进程之间自定义
SIGUSR2 12	进程之间可以用来通信，含义进程之间自定义

信号

● 常见的信号2：

收到这些信号的进程采取的默认动作是：除了异常终止进程，可能有些额外处理，如产生core文件

SIGFPE 8	浮点运算异常时产生
SIGILL 4	处理器执行了一条非法的指令
SIGQUIT 3	Ctrl+\组合键产生
SIGSEGV 11	段冲突。内存非法操作时产生

信号

● 常见的信号3:

收到这些信号的进程采取的默认动作是：进程挂起，进程暂停执行

SIGSTOP 19	暂停执行进程，不能被捕获或忽略
SIGTSTP 20	Ctrl+z组合键产生，导致终端挂起

收到这些信号的进程采取的默认动作是：忽略

SIGCONT 18	如果进程被暂停，就继续执行，否则忽略
SIGCHLD 17	进程暂停或退出时产生，发送给其父进程

信号

● 通过命令发信号给指定进程

给指定的进程发信号

`kill -num pid`

`kill -XXX pid` //XXX, 不带SIG信号名称如KILL、ALRM等等

给所有可执行文件名叫指定名称的进程发信号

`killall -num 程序名`

`killall -XXX 程序名`

信号

● 在代码给其它进程发信号

功能：给指定的进程发信号

原型：int kill(pid_t pid, int signo);

返回值：成功返回0，否则< 0

参数：

signo参数可以直接使用序号，或预定义好的常量宏如SIGINT
(推荐)

pid > 0 表示接受信号的进程的PID

pid == 0 给同组（进程组）进程发送信号

pid < 0 给进程组ID等于pid绝对值的进程发送信号

pid == -1 发给本进程有权限给对方发信号的所有进程

给进程自身发信号

int raise(int signo) 完全等价于kill(getpid(),signo)

信号-发送和捕捉

所需头文件	#include <signal.h> #include <sys/types.h>	
函数原型	int kill(pid_t pid, int sig);	
函数传入值	pid:	正数: 要接收信号的进程的进程号
		0: 信号被发送到所有和pid进程在同一个进程组的进程
		-1: 信号发给所有的进程表中的进程(除了进程号最大的进程外)
	sig: 信号	
函数返回值	成功: 0	
	出错: -1	

信号-发送和捕捉

所需头文件	<code>#include <signal.h></code> <code>#include <sys/types.h></code>
函数原型	<code>int raise(int sig);</code>
函数传入值	sig: 信号
函数返回值	成功: 0
	出错: -1

信号-发送和捕捉

```
int main()
{
    pid_t pid;
    int ret;
    if ((pid = fork()) < 0) /*创建一子进程*/
    {
        perror("fork");
        exit(1);
    }
    if (pid == 0)
    {
        raise(SIGSTOP); /*发出SIGSTOP信号*/
        printf("child process exit...\n");
        exit(0);
    }
}
```

信号-发送和捕捉

```
else /*在父进程中检测子进程的状态，调用kill函数*/
{
    printf("pid = %d\n",pid);
    if((waitpid(pid, NULL, WNOHANG)) == 0)
    {
        kill(pid, SIGKILL);
        printf("kill %d\n", pid);
    }
}
}
```

信号-发送和捕捉

- ✗ alarm()和pause()
- ✗ alarm()也称为闹钟函数，它可以在进程中设置一个定时器。当定时器指定的时间到时，内核就向进程发送SIGALARM信号。
- ✗ pause()函数是用于将调用进程挂起直到收到信号为止。

信号-发送和捕捉

所需头文件	#include <unistd.h>
函数原型	unsigned int alarm(unsigned int seconds)
函数传入值	seconds: 指定秒数
函数返回值	成功: 如果调用此alarm()前, 进程中已经设置了闹钟时间, 则 返回上一个闹钟时间的剩余时间, 否则返回0。
	出错: -1
所需头文件	#include <unistd.h>
函数原型	int pause(void);
函数返回值	-1, 并且把error值设为EINTR。

信号-发送和捕捉

```
int main()
{
    int ret;
    /*调用alarm定时器函数*/
    ret = alarm(5);
    pause();
    printf("I have been waken up.\n", ret);
    return 0;
}
```

[root@(none) tmp]#./alarm

Alarm clock

信号-信号的处理

- ✗ 特定的信号是与相应的事件相联系的
- ✗ 一个进程可以设定对信号的相应方式
- ✗ 信号处理的主要方法有两种
 - + 使用简单的signal()函数
 - + 使用信号集函数组
- ✗ signal()
 - + 使用signal函数处理时，需指定要处理的信号和处理函数
 - + 使用简单、易于理解

信号-信号的处理

所需头文件	#include <signal.h>	
函数原型	void (*signal(int signum, void (*handler)(int)))(int);	
函数传入值	signum: 指定信号	
	handler:	SIG_IGN: 忽略该信号。
		SIG_DFL: 采用系统默认方式处理信号。
		自定义的信号处理函数指针
函数返回值	成功: 设置之前的信号处理方式	
	出错: -1	

信号-信号的处理

```
void my_func(int sign_no)
{
    if (sign_no == SIGINT)
        printf("I have got SIGINT\n");
    else if (sign_no == SIGQUIT)
        printf("I have got SIGQUIT\n");
}

int main()
{
    printf("Waiting for signal SIGINT or SIGQUIT \n ");
    signal(SIGINT, my_func);
    signal(SIGQUIT, my_func);
    pause();
    exit(0);
}
```

信号

● 检查或修改与指定信号关联的处理动作

原型:

```
int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
```

返回值: 成功返回0, 否则< 0

参数:

act: 非空时修改对应信号处理动作

oact: 非空时保存对应信号的上一个动作

```
struct sigaction{  
    void (*sa_handler)(int); /*新的信号处理函数指针*/  
    /*两个特殊值SIG_IGN忽略信号, SIG_DFL恢复默认行为*/  
    sigset_t sa_mask; /*新的进程信号屏蔽字*/  
    int sa_flags; /*一般填零*/  
}
```


信号

● sigaction示例

testsig.c

信号

● 信号集操作

```
int sigaddset(sigset_t *set,int signo);/*向信号集里增加信号*/  
int sigemptyset(sigset_t *set);/*清空信号集*/  
int sigfillset(sigset_t *set); /*信号集里包含所有信号*/  
int sigdelset(sigset_t *set,int signo); /*从信号集里删除信号*/  
成功返回0，失败< 0
```

```
int sigismember(sigset_t *set,int signo);/*判断信号集是否存在指定信号*/  
存在返回1，如果不是返回0，给定的信号无效返回 < 0
```

```
int sigsuspend(const sigset_t *sigmask);  
将进程的信号屏蔽字设置为sigmask。在捕捉到一个信号或发送了一个会终止该进程的信号之前，该进程被挂起，唤醒还原原来的屏蔽字。始终返回-1，errno为EINTR。
```

进程间通讯方式比较

人 进程间通信

- pipe: 具有亲缘关系的进程间，单工，数据在内存中
- fifo: 可用于任意进程间，双工，有文件名，数据在内存
- signal: 唯一的异步通信方式
- msg: 常用于cs模式中，按消息类型访问，可有优先级
- shm: 效率最高(直接访问内存)，需要同步、互斥机制
- sem: 配合共享内存使用，用以实现同步和互斥






本课总结

重点

- 文件系统
- 文件描述符
- 进程，进程控制块
- 进程空间布局
- 信号
- 线程
- 临界区，同步机制（信号量）
- IPC：管道，共享内存，信号量

本课总结

难点

-  文件描述符
-  进程控制
-  线程取消点
-  同步机制的运用
-  信号量集






课总结

重要接口函数




- ❖ I/O操作：open、close、read、write、lseek、stat、ioctl、fcntl、mmap、msync、munmap
- ❖ 进程：fork、execl、execvp、wait、waitpid、exit、atexit
- ❖ 信号：sigaction、kill、alarm、pause、sleep
- ❖ 线程：pthread_create、exit、cancel、join、detach
- ❖ 线程同步：pthread_mutex、pthread_rwlock
- ❖ IPC：管道、共享内存、信号量集、锁文件、区域锁

课程总结

本节课程内容

-  进程间的通讯关系
-  管道
-  IPC对象
-  内存共享
-  消息队列

下节课程

-  网络通讯基本知识
-  网络基本术语
-  五层架构协议

联系方式

QQ: 59189174

E-mail: yumeifly@sohu.com