进程间通信

课程目标

- 掌握进程间通信的常用手段。
  - 无名管道（匿名管道）
  - 有名管道（命名管道）
  - 信号
  - 消息队列
  - 共享内存
  - 信号灯
  - 套接字
- 会使用进程间通信协同完成资源共享、信息同步。

实验目标

- 利用无名管道，实现父子进程间的通信。
- 利用有名管道，实现进程间通信。
- 利用信号机制，实现进程周期任务的执行。

进程通信的场景

- 为什么要有进程间通信？
  - 1、利用进程相对独立的特点，方便组件开发调试，方便模块的运行和终止，分层开发，增加灵活性，降低耦合性。
  - 2、进程间资源访问的互斥和同步，防止多个进程同时操作临界资源。
  - 3、内核进程和用户进程要进行信息交互。
- 如果进程间不能通信，会怎样？
  - 进程间如果不能进行通信，那么用户需要手动的控制每一个进程，并且管理他们的输入输出。
  - 不能杀进程。

进程间通信

- 思考：回想自己初高中恋爱的时候，如何在老师禁止的情况下传递信息呢？

- 思考：你来设计进程通信，你会设计那些通信手段呢？

进程间通信之

无名管道

无名管道

- 进程间通信最简单的实现思路
  - 1、打开一个文件，。
  - 2、fork一个子进程，此时两个进程都可以操作同一个文件指针。

- 问题：
- 子进程、父进程使用的同一个文件，但是不能使用fopen(有缓存)。
- 两个进程要互斥的使用文件。
- 两个高速内存中的进程要经过低速磁盘传递信息，效率大打折扣

无名管道

- 管道是一种特殊的文件，实现了用文件进行通信的思路，解决了用文件进行通信的弊端。
- 1、用read、write操作。
- 2、文件内容不存在磁盘上，只存在内存中。
- 3、独立设计读、写文件为不同的描述符，封装互斥操作。

无名管道

- 这里所说的管道主要指无名管道，它具有如下特点：
  - 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。

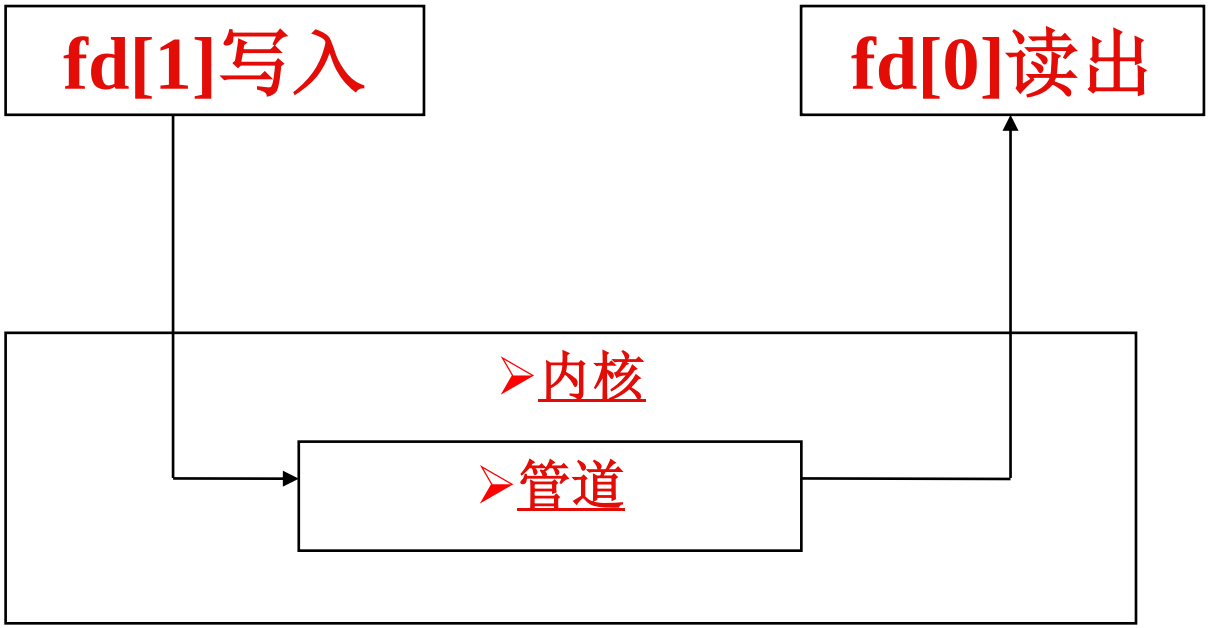
管道创建与关闭

所需头文件	**#include <unistd.h>**
函数原型	**int pipe(int fd[2])**
函数参数	**fd：包含两个元素的整型数组**
函数返回值	**成功：0**
	出错：-1

管道创建与关闭

- 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。

无名管道



无名管道

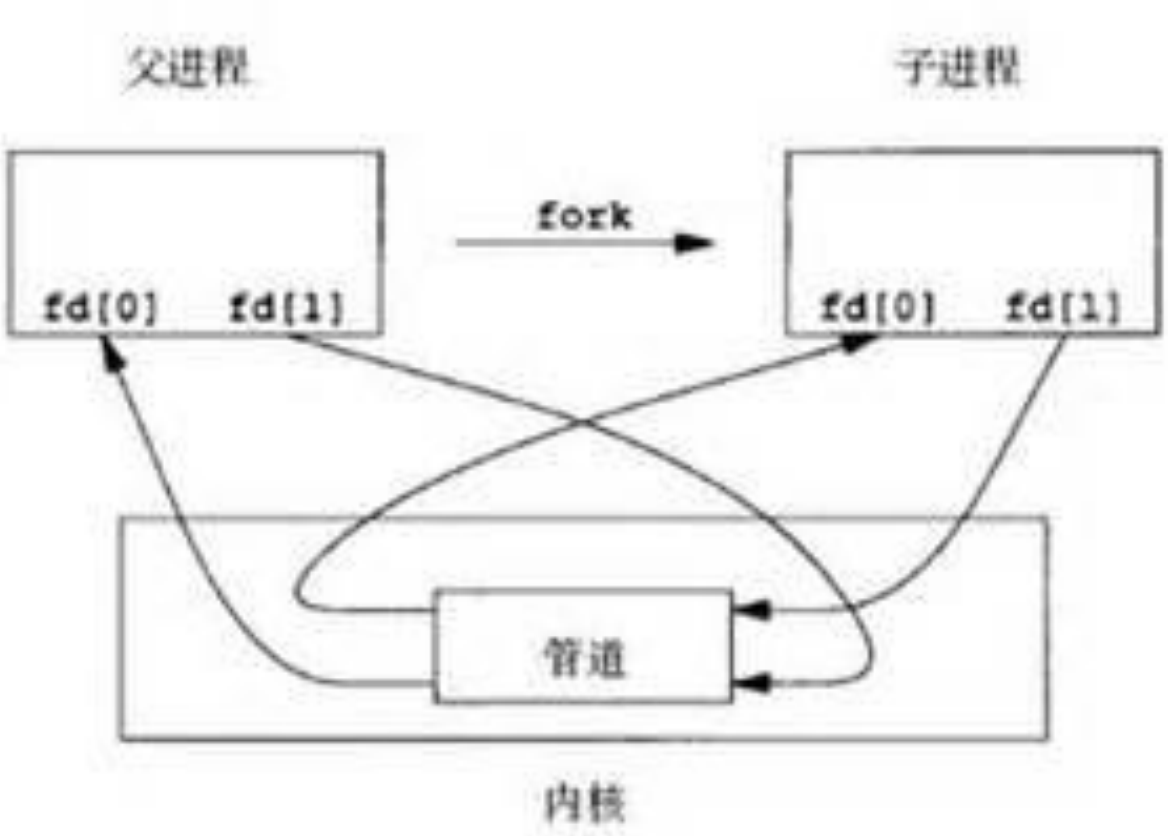


图2 父子进程管道的文件描述符对应关系

管道创建与关闭

华清远见 FARSIGHT 嵌入式培训专家

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

int main()
{
    int  pfd[2];
    if (pipe(pfd) < 0){/*创建无名管道*/
        printf("pipe create error\n");
        return -1;
    }
    else {
        printf("pipe create success\n");
    }
    close(pfd[0]);    /*关闭管道描述符*/
    close(pfd[1]);
}
```

管道读写

ssize_t read(int fd, void *buf, size_t count);

ssize_t write(int fd, void *buf, size_t count);

功能：读文件内容到buf，或将buf内容写入文件。

头文件：#include <unistd.h>

fd：　要读写文件的路径（相对路径、绝对路径）

buf：　要读入或写出的缓冲区，提前申请好的

count：要读入或写出的字节数

成功：

返回：实际读写的字节数

失败：-1

管道读写注意点

- 当管道中无数据时，读操作会阻塞，（读常规文件到达尾部返回EOF）。
- 管道有大小限制，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)。
- 思考：管道作为特殊的文件，其读写操作和普通文件有何不同？

实验1

- 1、创建一个管道，先写10个‘1’，再将其读出来，体会管道读写。
- 2、创建一个管道，不断的写入‘1’，观察写进去多少字符时，写操作阻塞，分析管道默认的大小。
- 3、创建一个管道，再创建一个子进程，父进程每ms写20个字符进管道，子进程每ms读7个字符出管道，体会管道传递速度的木桶效应。

扩展

- dup2(fd, STDOUT_FILENO);　//将标准输出重定向到某个文件描述符上。
- dup复制一个文件描述符 。

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

实验1扩展1（匿名管道实用场景）

- 4、创建子进程ifconfig、将子进程的输出写到管道里，父进程通过读管道，得到子进程的输出，完成父子进程间的通信。

思考：如果运行的不是ifconfig，而是top一些不会退出的函数呢？

```c
int main(int argc, char ** argv)
{
    int fd[2];
    pid_t pid;
    int ret = 0;
    char buf[1024];
    if(pipe(fd) == -1){
        perror("pipe");
        return -1;
    }
    pid = fork();
    if(pid == 0){
        dup2(fd[1], STDOUT_FILENO);
        execlp("ifconfig", "ifconfig", NULL);
        perror("execlp");
    }else if(pid > 0){
        waitpid(pid, NULL, 0);
        ret = read(fd[0], buf, sizeof(buf));
        printf("read = %d\n%s\n", ret, buf);
    }
    return 0;
}
```

实验1扩展2（匿名管道实用场景）

- 5、设置自己linux主机的IP地址为192.168.8.8。
  - 1、创建一个管道，创建子进程ifconfig，将子进程的标准输出重定向到管道里面。
  - 2、读管道得到ifconfig的输出，分析出主机的网卡名称。
  - 3、依照3的结果，设置ifconfig参数，设置网卡IP地址。
    - ifconfig eth0 192.168.8.8 netmask 255.255.255.0
  - 4、监测设置成功的话打印OK，否则打印failed。

进程间通信之

有名管道

zlongz

FIFO

- 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。如何解决这个问题？
- 系统中的文件都有文件名，可不可以给管道起个名字，放到文件系统里面呢？
- 如果可以，不同进程就可以共享文件路径，而不需要传递文件描述符；通过读写同一个特殊文件，完成信息传递。

- 问题：
- 如何创建这样的特殊文件？
- 这个特殊文件把文件名留在文件系统里，把文件的实体放在内存中，这是文件和无名管道的私生子吗？

FIFO

- 有名管道兼具了磁盘文件和无名管道的优点，使进程通信更加方便，其有如下特点：

1. 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。
2. 进程通过文件IO来操作有名管道。
3. 有名管道遵循先进先出规则。
4. 不支持如lseek() 操作

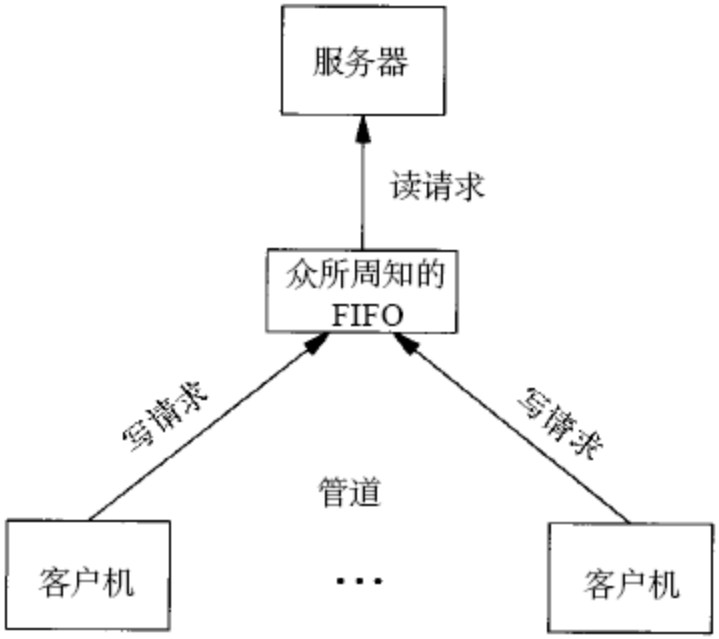
FIFO

所需头文件	#include <unistd.h> #include <fcntl.h> #include <sys/types.h>
函数原型	int mkfifo(const char *filename,mode_t mode);
函数传入值	filename：要创建的管道。
	mode：指定创建的管道的访问权限，一般用8进制数表示 0644
函数返回值	成功：0
	出错：-1

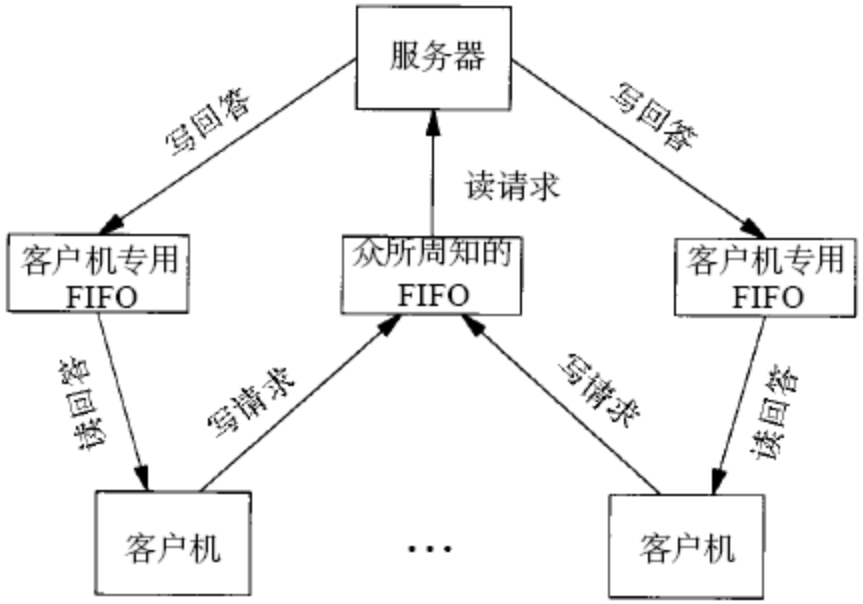
FIFO常见错误码

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**

典型的FIFO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

无名管道、有名管道

- 1、无名管道使用pipe函数，得到一对读写的文件描述符，父子进程通过读写这对文件描述符，完成通信。
- 2、有名管道使用mkfifo函数，创建一个本地的管道文件，所有进程通过打开这个文件，得到内存中管道的文件描述符，读写该文件完成通信。
- 3、有名、无名管道都有大小限制，写满了会阻塞，没东西可读也会阻塞（当然也可以通过设置其不阻塞，或一定时间内阻塞）。

无名管道、有名管道

- 有名管道的读写和文件、无名管道一样，使用read、write进行读写。
- 有名管道在文件系统内有实体的文件存在，读写前要用open打开，匿名管道用pipe打开。
- 有名管道在进程结束、创建的管道文件不会被自动删除，如果有了就不需要创建（ mkfifo前先access下判断是否已存在 ）。
- 如果说有名管道和文件有些类似，那么它更像临时文件，和管道一样，他要同时返回读写的文件描述符，和临时文件一样，关闭文件描述符后，内容就清空了。

实验2

- 1、进程A创建一个有名管道”/tmp/pipetest”，然后一直读管道内容并打印；进程B可写方式打开这个管道，且每次运行的时候将自己的参数写入管道中。
- 2、改造实验2.1，将进程A运行在root权限，将从管道中取得的内容作为指令运行，进程B可写方式打开这个管道，且每次运行的时候将想要以root权限运行的指令写入管道中。（注意如果进程A要新创建这个有名管道，权限设为你当前用户可写，如mask为0666）

进程间通信之

信号

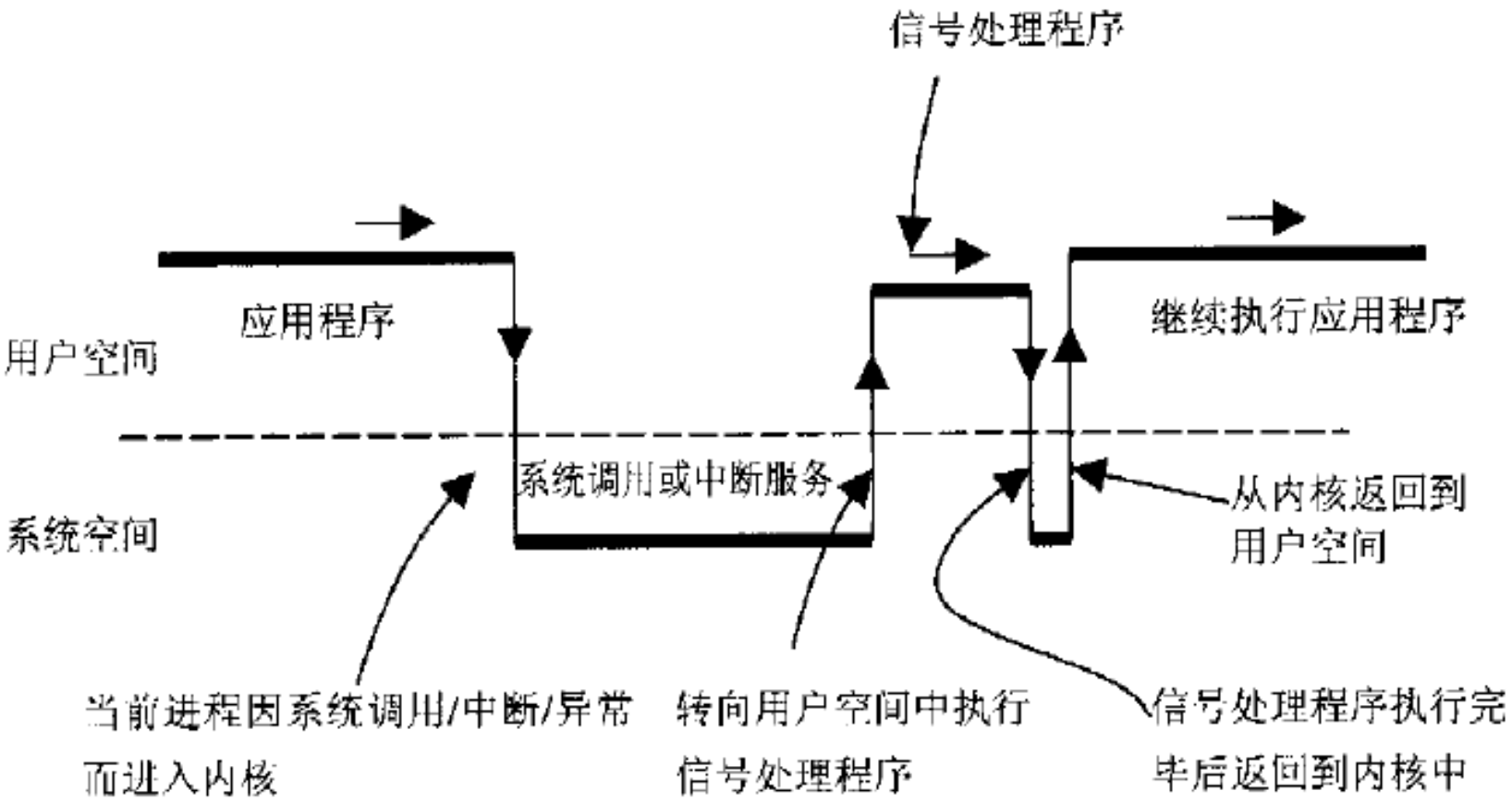
信号

- 信号：顾名思义，通过发送信号完成通信，类似于暗号、手势通信。
- 特点：
  - 暗号、手势等信号只能传递有限的信息。
  - 暗号、手势负责传递特定的信息，需提前约定好信号的含义、处理方式等。
- 问题：
  - 给进程发信号，要指定发给谁？
  - 如何定义收到某种信号后的处理方式？
  - 我现在正忙着上课，会不会错过大脑尿急的信号而失禁？

信号

- 信号是在软件层次上对中断机制的一种模拟，是一种异步通信方式
- 信号可以直接进行用户空间进程和内核进程之间的交互，内核进程也可以利用它来通知用户空间进程发生了哪些系统事件。
- 如果该进程当前并未处于执行态，则该信号就由内核保存起来，直到该进程恢复执行再传递给它；如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程

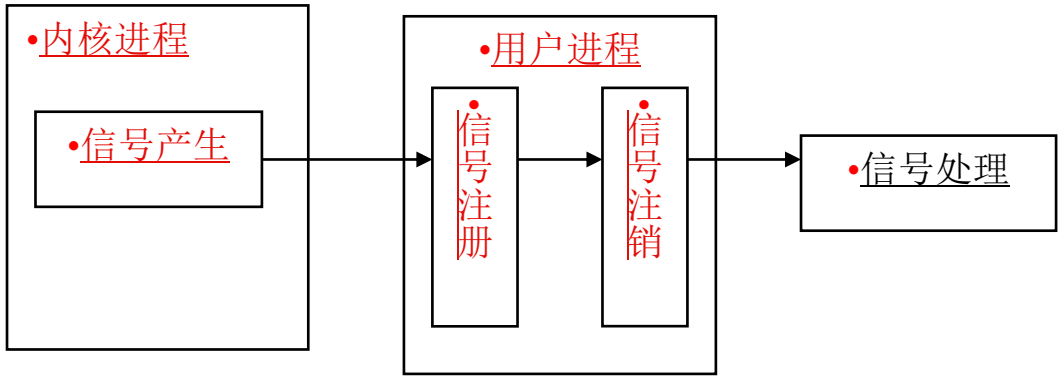
信号处理流程



信号的检测与处理流程图

信号通信

- 信号的生存周期



- 用户进程对信号的响应方式：
  - 忽略信号：对信号不做任何处理，但是有两个信号不能忽略：即SIGKILL及SIGSTOP。
  - 捕捉信号：定义信号处理函数，当信号发生时，执行相应的处理函数。
  - 执行缺省操作：Linux对每种信号都规定了默认操作

信号通信

信号名	含义	默认操作
SIGHUP	该信号在用户终端连接(正常或非正常)结束时发出，通常是在终端的控制进程结束时，通知同一会话内的各个作业与控制终端不再关联。	终止
SIGINT	该信号在用户键入INTR字符(通常是Ctrl-C)时发出，终端驱动程序发送此信号并送到前台进程中的每一个进程。	终止
SIGQUIT	该信号和SIGINT类似，但由QUIT字符(通常是Ctrl-\)来控制。	终止
SIGILL	该信号在一个进程企图执行一条非法指令时(可执行文件本身出现错误，或者试图执行数据段、堆栈溢出时)发出。	终止
SIGFPE	该信号在发生致命的算术运算错误时发出。这里不仅包括浮点运算错误，还包括溢出及除数为0等其它所有的算术的错误。	终止

信号通信

信号名	含义	默认操作
SIGKILL	该信号用来立即结束程序的运行，并且不能被阻塞、处理和忽略。	终止
SIGALRM	该信号当一个定时器到时的时候发出。	终止
SIGSTOP	该信号用于暂停一个进程，且不能被阻塞、处理或忽略。	暂停进程
SIGTSTP	该信号用于暂停交互进程，用户可键入SUSP字符(通常是Ctrl-Z)发出这个信号。	暂停进程
SIGCHLD	子进程改变状态时，父进程会收到这个信号	忽略
SIGABORT	该信号用于结束进程	终止

信号

- shell中kill指令用来向特定的进程发送指令。
- Kill –l得到系统支持的指令。

- 如果向特定进程发送SIGKILL信号
  - Kill –KILL pid
- 如果向特定进程发送SIGHUP信号
  - Kill –HUP pid
- 依次类推…

信号发送

- kill()和raise()
  - kill函数同读者熟知的kill系统命令一样，可以发送信号给进程或进程组(实际上，kill系统命令只是kill函数的一个用户接口)。
  - raise函数向自己发送信号，像当于kill(getpid(), sig);
- 注意：低权限的进程不能向高权限的用户发信号。

信号发送

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

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

实验3

1、使用终端指令kill -l查看系统支持的信号。
2、向自己发送SIGKILL信号，立即结束自己，观察输出。

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

int main()
{
    printf("hello world\n");
    kill(getpid(), SIGKILL);
    printf("Please don't kill me!\n");
    return 0;
}
```

信号发送与捕捉

- 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。

信号发送与捕捉

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。**

alarm注意

- 1、alarm函数尽量不要和sleep函数一起使用，某些系统的sleep是用alarm实现的。
- 2、 alarm函数执行时如果上一个定时器没有结束，那么会取消上一个计时器，重置时间。

- 思考1：我同时设置两个闹钟，一个1秒钟以后，一个7秒钟后，可以吗？
- 思考2：我同时设置一个闹钟，设置为1秒钟以后，之后我直接sleep(1000)，会休眠1000秒吗？
- 设计程序验证自己的猜想!

实验4

1、使用alarm实现程序休眠1秒，相对与sleep(1)

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

int main()
{
    int ret;
    ret = alarm(1); //调用alarm定时器函数
    pause();
    printf("I have been waken up.\n");
    return 0;
}
```

◆思考：为什么printf内容没有打印出来？

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
  - 使用信号集函数族
- 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

信号的处理

所需头文件	#include <signal.h>	
函数原型	void *signal(int signum, void *handler);	
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	

信号的处理

- Signal注册的信号处理函数，一般不要设计为长时间执行，如果要长时间执行，设计为线程方式。
- 中断的处理，后面的课程会讲述，一般情况下，中断处理函数执行过程中，不能被再次中断（试想一下printf执行打印中断过程中被另一个printf中断会出现什么情况）。
- 很多的系统调用都是用中断处理的，从而保证操作的原子性。

信号的处理

信号处理函数是一个void类型的无参数的回调函数

思考：回忆一下我们的线程函数原型。

```c
void sighup()
{
    printf("i get SIGHUP signal\n");
}
int main()
{
    pid_t pid;

    pid = fork();
    if(pid == 0){
        signal(SIGHUP, sighup);
        pause();
    }else{
        sleep(1);
        kill(pid, SIGHUP);
        wait(NULL);
    }
    return 0;
}
```

信号的处理实用技巧

- 1、如果父进程fork前设置signal(SIGCLD, SIG_IGN)，即忽略子进程的信号，也即父进程不再为子进程回收资源，新创建的子进程就会自己回收资源。
- 2、如果进程设为signal(SIGHUP, SIG_IGN)，即忽略控制中断关闭信号，则可以脱离终端运行。
- 3、用alarm做定时器，定时做一些操作时，一般将重设定时器的操作（alarm函数）放在时钟信号处理函数里面。

信号使用知识回顾

- 1、信号机制是一种软中断，由内核提供支持。
- 2、通过kill –l可以查看内核所支持的信号。
- 3、*int kill(pid_t pid, int sig)*用来向特定进程发送信号，*int raise(int sig)*向自己发送信号。
- 4、 *int pause(void)*阻塞等待一个信号。
- 5、 *unsigned int alarm(unsigned int seconds)* 用来定时发送一个定时器信号SIGALRM
- 6、 *sighandler_t signal(int signum, sighandler_t handler)* 用来注册一个信号处理函数。

实验5

- 1、用信号机制重新实现多线程的实验2-3
- 多线程的实验2-3：编写一个程序，一面得到用户的输入，一方面倒计时，如果十秒钟用户没有输入“I love you”，就打印“都老了，没戏了！”，否则打印“结婚吧，生娃吧！”，10秒钟后要结束输入线程。

实验5

- 2、用定时器实现每1秒，打印一个字符‘O’。
- 3、将上面的使用改造成一个进程接收并处理信号（打印一个字符‘O’），另一个进程给它发信号。

```c
void fun()
{
    printf("O\n");
}
int main()
{
    signal(SIGALRM, fun);
    while(1){
        alarm(1);
        pause();
    }

    return 0;
}
```

```c
void * fun()
{
    alarm(1);
    printf("O\n");
}
int main()
{
    signal(SIGALRM, fun);
    alarm(1);
    while(1){
        pause();
    }
    return 0;
}
```

纠错

```c
void * fun()
{
    alarm(1);
    printf("O\n");
}
int main()
{
    signal(SIGALRM, fun);
    alarm(1);
    sleep(10000000000000000000000000);
    return 0;
}
```

```c
void * fun()
{
    printf("O\n");
}
int main()
{
    signal(SIGALRM, fun);
    while(1){
        alarm(1);
    }
    return 0;
}
```