国嵌

www.enjoylinux.cn

LINUX
进程间通信程序设计-1

Contents

Contents

目的

为什么进程间需要通信？

1、数据传输

一个进程需要将它的数据发送给另一个进程。

2、资源共享

多个进程之间共享同样的资源。

目的

3、通知事件

一个进程需要向另一个或一组进程发送消息，通知它们发生了某种事件。

4、进程控制

有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有操作，并能够及时知道它的状态改变。

发展

Linux进程间通信（IPC）由以下几部分发展而来:

1、UNIX进程间通信

2、基于System V进程间通信

3、POSIX进程间通信

POSIX

POSIX(Portable Operating System Interface)表示可移植操作系统接口。电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）最初开发 POSIX 标准，是为了提高 UNIX 环境下应用程序的可移植性。然而，POSIX 并不局限于 UNIX，许多其它的操作系统，例如 DEC OpenVMS 和 Microsoft Windows，都支持 POSIX 标准。

System V

System V，也被称为 AT&T System V，是Unix操作系统众多版本中的一支。

分类

现在Linux使用的进程间通信方式包括:

1、管道（pipe）和有名管道（FIFO）

2、信号（signal）

3、消息队列

4、共享内存

5、信号量

6、套接字（socket）

Contents

管道通信

什么是管道?

管道是单向的、先进先出的，它把一个进程的输出和另一个进程的输入连接在一起。一个进程（写进程）在管道的尾部写入数据，另一个进程（读进程）从管道的头部读出数据。

管道通信

数据被一个进程读出后，将被从管道中删除，其它读进程将不能再读到这些数据。管道提供了简单的流控制机制，进程试图读空管道时，进程将阻塞。同样，管道已经满时，进程再试图向管道写入数据，进程将阻塞。

管道创建

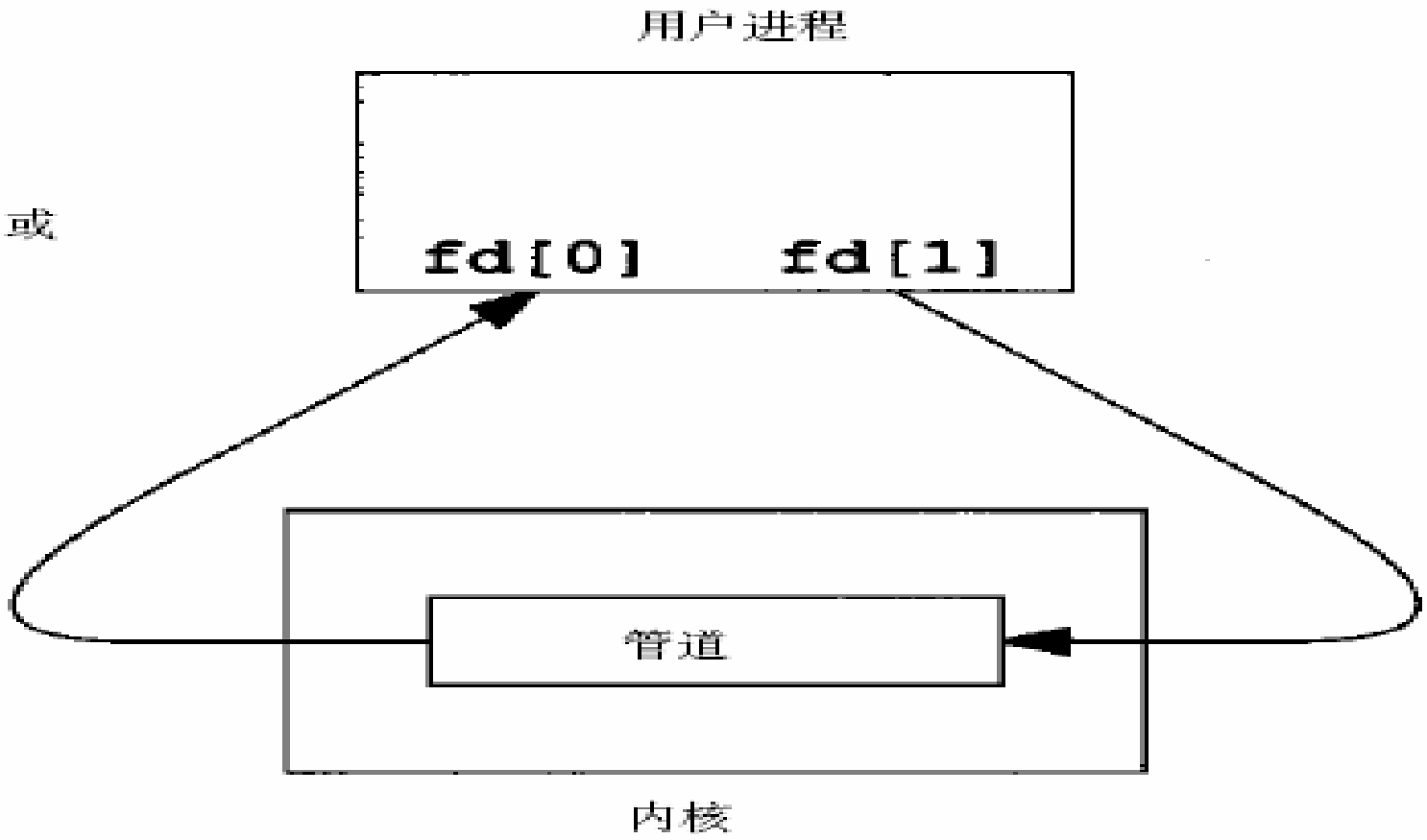
管道包括无名管道和有名管道两种，前者用于父进程和子进程间的通信，后者可用于运行于同一系统中的任意两个进程间的通信。

无名管道由pipe（ ）函数创建:

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

当一个管道建立时，它会创建两个文件描述符：filedis[0] 用于读管道， filedis[1] 用于写管道。

管道通信



管道关闭

关闭管道只需将这两个文件描述符关闭即可，可以使用普通的close函数逐个关闭。

管道通信

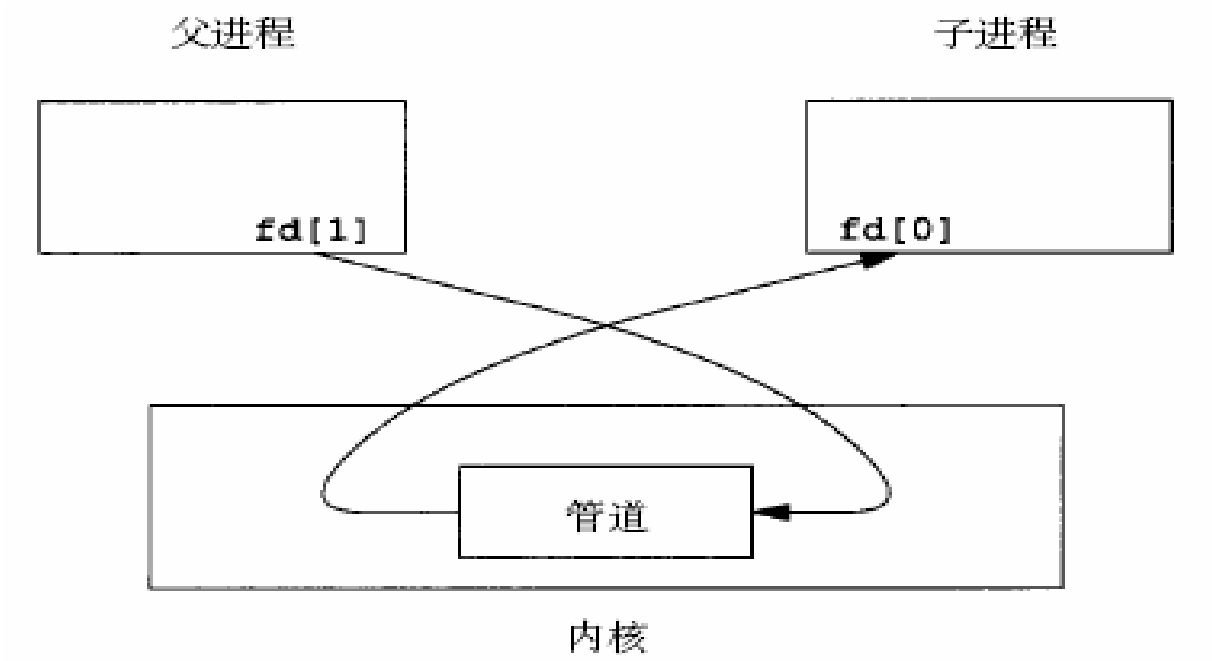
www.enjoylinux.cn

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int pipe_fd[2];
        if(pipe(pipe_fd)<0)
        {
            printf("pipe create error\n");
            return -1;
        }
        else
                printf("pipe create success\n");
        close(pipe_fd[0]);
        close(pipe_fd[1]);
}
```


管道读写

管道用于不同进程间通信。通常先创建一个管道，再通过fork函数创建一个子进程，该子进程会继承父进程所创建的管道。



注意事项

必须在系统调用fork( )前调用pipe( )，否则子进程将不会继承文件描述符。

实例分析（演示）

pipe_rw.c

命名管道（FIFO）

命名管道和无名管道基本相同，但也有不同点：无名管道只能由父子进程使用；但是通过命名管道，不相关的进程也能交换数据。

创建


```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char * pathname, mode_t mode)
```

- pathname： FIFO文件名
- mode： 属性（见文件操作章节）

一旦创建了一个FIFO，就可用open打开它，一般的文件访问函数（close、read、write等）都可用于FIFO。

操作

当打开FIFO时，非阻塞标志（O_NONBLOCK）

将对以后的读写产生如下影响:

1、没有使用O_NONBLOCK：访问要求无法满足时进程将阻塞。如试图读取空的FIFO，将导致进程阻塞。

2、使用O_NONBLOCK：访问要求无法满足时不阻塞，立刻出错返回，errno是ENXIO。

实例分析（演示）

fifo_write.c

fifo_read.c

Contents

信号通信

信号(signal)机制是Unix系统中最为古老的进程间通信机制，很多条件可以产生一个信号：

1、当用户按某些按键时，产生信号。

2、硬件异常产生信号：除数为0、无效的存储访问等等。这些情况通常由硬件检测到，将其通知内核，然后内核产生适当的信号通知进程，例如，内核对正访问一个无效存储区的进程产生一个SIGSEGV信号。

信号通信

3、进程用kill函数将信号发送给另一个进程。

4、用户可用kill命令将信号发送给其他进程。

信号类型

www.enjoylinux.cn

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGIOT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	
17) SIGCHLD	18) SIGCONT	19) SIGSTOP	
20) SIGTSTP	21) SIGTTIN	22) SIGTTOU	
23) SIGURG	24) SIGXCPU	25) SIGXFSZ	
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	
29) SIGIO	30) SIGPWR		

信号类型

下面是几种常见的信号:

- SIGHUP: 从终端上发出的结束信号
- SIGINT: 来自键盘的中断信号（Ctrl-C）
- SIGKILL: 该信号结束接收信号的进程
- SIGTERM: kill 命令发出的信号
- SIGCHLD: 标识子进程停止或结束的信号
- SIGSTOP: 来自键盘（Ctrl-Z）或调试程序的停止执行信号

信号处理

当某信号出现时，将按照下列三种方式中的一种进行处理:

1、忽略此信号

大多数信号都按照这种方式进行处理，但有两种信号却决不能被忽略。它们是: **SIGKILL和SIGSTOP**。这两种信号不能被忽略的原因是: 它们向超级用户提供了一种终止或停止进程的方法。

信号处理

2、执行用户希望的动作

通知内核在某种信号发生时，调用一个用户函数。在用户函数中，执行用户希望的处理。

3、执行系统默认动作

对大多数信号的系统默认动作是终止该进程。

信号发送

发送信号的主要函数有 kill和raise。

区别:

Kill既可以向自身发送信号，也可以向其他进程发送信号。与kill函数不同的是，raise函数是向进程自身发送信号。

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo)
int raise(int signo)
```

信号发送

kill的pid参数有四种不同的情况:

1、pid>0

将信号发送给进程ID为pid的进程。

2、pid == 0

将信号发送给同组的进程。

3、pid < 0

将信号发送给其进程组ID等于pid绝对值的进程。

4、pid == -1

将信号发送给所有进程。

Alarm

使用alarm函数可以设置一个时间值(闹钟时间)，当所设置的时间到了时，产生SIGALRM信号。如果不捕捉此信号，则默认动作是终止该进程。

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds)
```

- **Seconds:**

  经过了指定的seconds秒后会产生信号SIGALRM。

Alarm

- 每个进程只能有一个闹钟时间。如果在调用alarm时，以前已为该进程设置过闹钟时间，而且它还没有超时，以前登记的闹钟时间则被新值代换。
- 如果有以前登记的尚未超过的闹钟时间，而这次seconds值是0，则表示取消以前的闹钟。

Pause

pause函数使调用进程挂起直至捕捉到一个信号。

```
#include <unistd.h>
int pause(void)
```

只有执行了一个信号处理函数后，挂起才结束。

信号的处理

- 当系统捕捉到某个信号时，可以忽略该信号或是使用指定的处理函数来处理该信号，或者使用系统默认的方式。
- 信号处理的主要方法有两种，一种是使用简单的signal函数，另一种是使用信号集函数组。

signal


```
#include <signal.h>
void (*signal (int signo, void (*func)(int)))(int)
```

如何理解？

```
typedef void (*sighandler_t)(int)
sighandler_t signal(int signum, sighandler_t handler))
```

Func可能的值是:

1、SIG_IGN: 忽略此信号

2、SIG_DFL: 按系统默认方式处理

3、信号处理函数名：使用该函数处理

实例分析（演示）

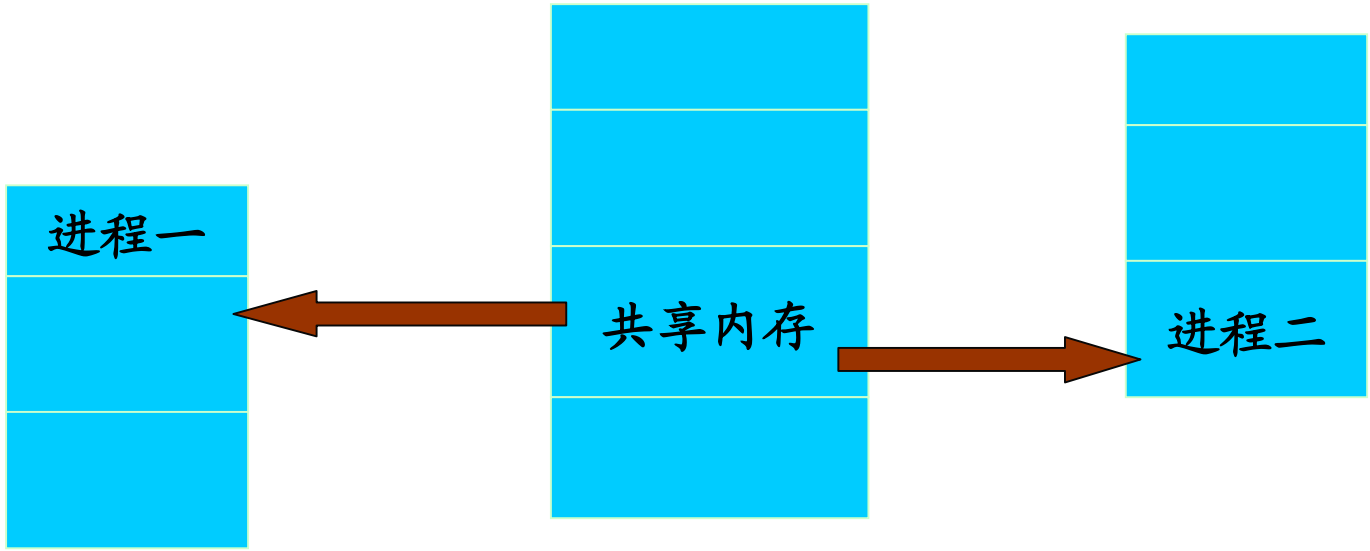
mysignal.c

Contents

共享内存

共享内存是被多个进程共享的一部分物理内存。共享内存是进程间共享数据的一种最快的方法，一个进程向共享内存区域写入了数据，共享这个内存区域的所有进程就可以立刻看到其中的内容。

共享内存



共享内存原理示意图

共享内存

共享内存实现分为两个步骤:

一、创建共享内存，使用**shmget**函数。

二、映射共享内存，将这段创建的共享内存映射到具体的进程空间去，使用**shmat**函数。

创建

int shmget (key_t key, int size, int shmflg)

key标识共享内存的键值: 0/IPC_PRIVATE。 当key的取值为IPC_PRIVATE，则函数shmget()将创建一块新的共享内存；如果key的取值为0，而参数shmflg中又设置IPC_PRIVATE这个标志，则同样会创建一块新的共享内存。

返回值：如果成功，返回共享内存标识符；如果失败，返回-1。

映射

int shmat (int shmid, char *shmaddr, int flag)

参数:

- **shmid**：shmget函数返回的共享存储标识符
- **flag**：决定以什么方式来确定映射的地址（通常为0）

返回值:

如果成功，则返回共享内存映射到进程中的地址；如果失败，则返回- 1。

共享内存

当一个进程不再需要共享内存时，需要把它从进程地址空间中脱离。

int shmdt (char *shmaddr)

实例分析（演示）

shmem.c