进程间通信

Star hu

2009/11/30

学习目标

- 了解Linux中进程间通信的概念，和常用的进程间通信方式
- 掌握管道的创建和读写
- 掌握命名管道的创建和读写
- 掌握信号的使用
- 掌握信号量的使用
- 掌握共享内存的使用
- 掌握消息队列的使用
- 掌握套接字的使用

本章的主要内容

- 1 Linux下进程间通信概述
- 2 管道
- 3 信号
- 4 信号量
- 5 共享内存
- 6 消息队列
- 7 套接字

- **Linux下进程间通信概述**
- 管道
- 信号
- 信号量
- 共享内存
- 消息队列
- 套接字

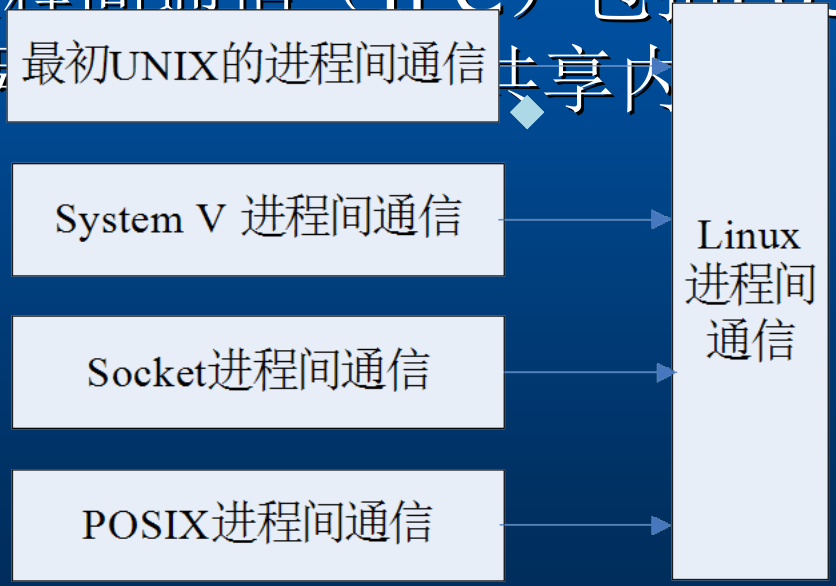
Linux下进程间通信概述（1）

- Linux下的进程通信手段基本上是从UNIX平台上的进程通信手段继承而来的。而对UNIX发展做出重大贡献的两大主力AT&T的贝尔实验室及BSD（加州大学伯克利分校的伯克利软件发布中心）在进程间的通信方面的侧重点有所不同。前者是对UNIX早期的进程间通信手段进行了系统的改进和扩充，形成了“system V IPC”，其通信进程主要局限在单个计算机内；

  后者则跳过了该限制，形成了基于套接口（socket）的进程间通信机制。而Linux则把两者的优势都继承了下来
- UNIX进程间通信（IPC）方式包括管道、FIFO以及信号。

Linux下进程间通信概述（2）

- System V进程间通信（IPC）包括System V消息队列、System V信号量以及System V共享内存区。
- Posix 进程间通信（IPC）包括Posix消息队列、Posix信号量以及Posix共享内存区。



进程间通信方式的种类（1）

（1）管道（Pipe）及命名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，命名管道，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。

（2）信号（Signal）：信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一样的。

（3）消息队列（Messge Queue）：消息队列是消息的链接表，包括Posix消息队列SystemV消息队列。它克服了前两种通信方式中信息量有限的缺点，具有写权限的进程可以按照一定的规则向消息队列中添加新消息；对消息队列有读权限的进程则可以从消息队列中读取消息。

进程间通信方式的种类（2）

（4）共享内存（Shared memory）：可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种通信方式需要依靠某种同步机制，如互斥锁和信号量等。

（5）信号量（Semaphore）：主要作为进程之间以及同一进程的不同线程之间的同步和互斥手段。

（6）套接字（Socket）：这是一种更为一般的进程间通信机制，它可用于本机或网络中不同机器之间的进程间通信，应用非常广泛。

- Linux下进程间通信概述
- **管道**
- 信号
- 信号量
- 共享内存
- 消息队列
- 套接字

管道概述

- 无名管道是Linux中进程间通信的一种方式。
  - 它只能用于具有亲缘关系的进程之间的通信（也就是父子进程或者兄弟进程之间）。
  - 它是一个半双工的通信模式，具有固定的读端和写端。
  - 管道也可以看成是一种特殊的文件，对于它的读写也可以使用普通的read()和write()等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内核的内存空间中。
- 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fds[0]和fds[1]，其中fds[0]固定用于读管道，而fd[1]固定用于写管道，这样就构成了一个半双工的通道。

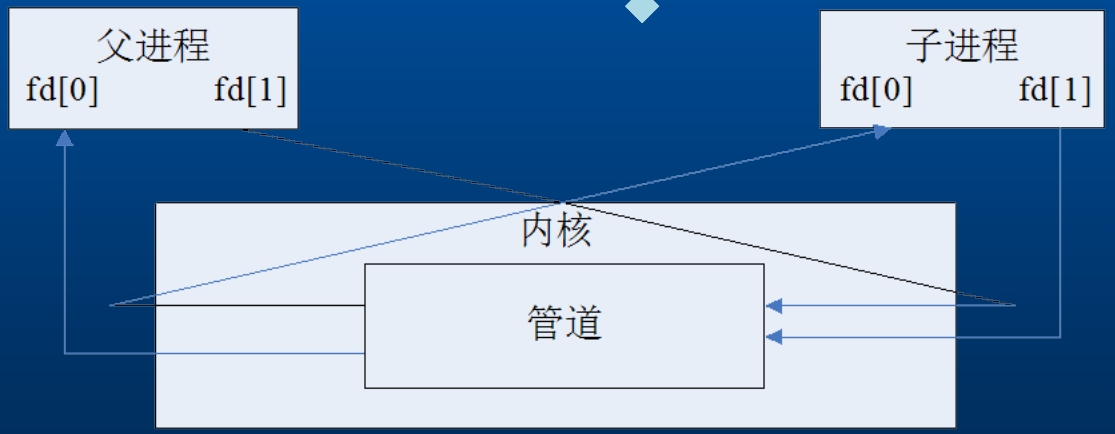
管道的创建和关闭

- 创建管道可以通过调用pipe()来实现。
- 管道关闭时只需使用普通的close()函数逐个关闭各个文件描述符。

所需头文件	#include <unistd.h>
函数原型	int pipe(int fd[2])
函数传入值	fd[2]：管道的两个文件描述符，之后就可以直接操作这两个文件描述符
函数返回值	成功：0
	出错：−1

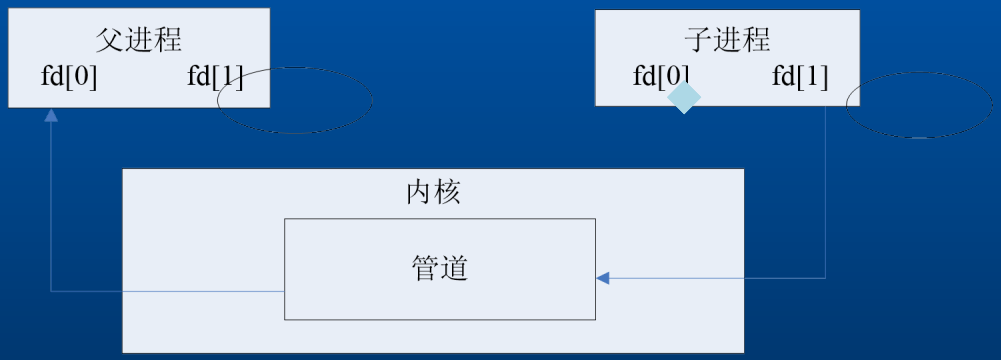
父子进程之间的管道通信（1）

- 用pipe()函数创建的管道两端处于一个进程中，由于管道是主要用于在不同进程间通信的，因此这在实际应用中没有太大意义。实际上，通常先是创建一个管道，再通过fork()函数创建一子进程，该子进程会继承父进程所创建的管道 。



父子进程之间的管道通信（2）

- 父子进程分别拥有自己的读写通道，为了实现父子进程之间的读写，只需把无关的读端或写端的文件描述符关闭即可。此时，父子进程之间就建立起了一条"子进程写入父进程读取"的通道。



标准流管道（1）

- 与Linux的文件操作中有基于文件流的标准I/O操作一样，管道的操作也支持基于文件流的模式。这种基于文件流的管道主要是用来创建一个连接到另一个进程的管道，这里的"另一个进程"也就是一个可以进行一定操作的可执行文件，例如，用户执行"ls -l"或者自己编写的程序"./pipe"等。由于这一类操作很常用，因此标准流管道就将一系列的创建过程合并到一个函数popen()中完成。它所完成的工作有以下几步。
  - 创建一个管道。
  - fork()一个子进程。
  - 在父子进程中关闭不需要的文件描述符。
  - 执行exec函数族调用。
  - 执行函数中所指定的命令。

标准流管道（2）

- 标准流管道的使用可以大大减少代码的编写量，但同时也有一些不利之处，例如，它不如前面管道创建的函数那样灵活多样，并且用popen()创建的管道必须使用标准I/O函数进行操作，但不能使用前面的read()、write()一类不带缓冲的I/O函数。
- 与之相对应，关闭用popen()创建的流管道必须使用函数pclose()来关闭该管道流。该函数关闭标准I/O流，并等待命令执行结束。

标准流管道（3）

所需头文件	#include <stdio.h>	
函数原型	FILE *popen(const char *command, const char *type)	
函数传入值	command：指向的是一个以 null 结束符结尾的字符串，这个字符串包含一个 shell 命令，并被送到/bin/sh 以-c 参数执行，即由 shell 来执行	
	type：	“r”：文件指针连接到 command 的标准输出，即该命令的结果产生输出 “w”：文件指针连接到 command 的标准输入，即该命令的结果产生输入
函数返回值	成功：文件流指针	
	出错：–1	

所需头文件	#include <stdio.h>
函数原型	int pclose(FILE *stream)
函数传入值	stream：要关闭的文件流
函数返回值	成功：返回由 popen()所执行的进程的退出码
	出错：–1

命名管道（1）

- 前面介绍的管道是无名管道，它只能用于具有亲缘关系的进程之间，这就大大地限制了管道的使用。 FIFO(命名管道)的出现突破了这种限制，它可以使互不相关的两个进程实现彼此通信。该管道可以通过路径名来指出，并且在文件系统中是可见的。在建立了管道之后，两个进程就可以把它当作普通文件一样进行读写操作，使用非常方便。不过值得注意的是， FIFO是严格地遵循先进先出规则的，对管道及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，它们不支持如lseek()等文件定位操作。
- FIFO的创建可以使用函数mkfifo()，该函数类似文件中的open()操作，可以指定FIFO的路径和打开的模式。

命名管道（2）

- 在创建管道成功之后，就可以使用open()、read()和write()这些函数了。与普通文件的开发设置一样，对于为读而打开的管道可在open()中设置O_RDONLY，对于为写而打开的管道可在open()中设置O_WRONLY，在这里与普通文件不同的是阻塞问题。
- 由于普通文件的读写时不会出现阻塞问题，而在管道的读写中却有阻塞的可能，这里的非阻塞标志可以在open()函数中设定为O_NONBLOCK。
- 对于读进程
  - 若该管道是阻塞打开，且当前FIFO内没有数据，则对读进程而言将一直阻塞到有数据写入。
  - 若该管道是非阻塞打开，则不论FIFO内是否有数据，读进程都会立即执行读操作。即如果FIFO内没有数据，则读函数将立刻返回0。

命名管道（3）

- 对于写进程
  - 若该管道是阻塞打开，则写操作将一直阻塞到数据可以被写入。
  - 若该管道是非阻塞打开而不能写入全部数据，则读操作进行部分写入或者调用失败。

命名管道（4）

所需头文件	#include <sys/types.h> #include <sys/state.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	
函数传入值	mode：	O_RDONLY：读管道
		O_WRONLY：写管道
		O_RDWR：读写管道
		O_NONBLOCK：非阻塞
		O_CREAT：如果该文件不存在，那么就创建一个新的文件，并用第三个参数为其设置权限
		O_EXCL：如果使用 O_CREAT 时文件存在，那么可返回错误消息。这一参数可测试文件是否存在
函数返回值	成功：0	
	出错：-1	

- Linux下进程间通信概述
- 管道
- **信号**
- 信号量
- 共享内存
- 消息队列
- 套接字

信号概述（1）

- 信号是UNIX中所使用的进程通信的一种最古老的方法。它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。它可以在任何时候发给某一进程，而无需知道该进程的状态。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为止；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。
- 不可靠信号和可靠信号

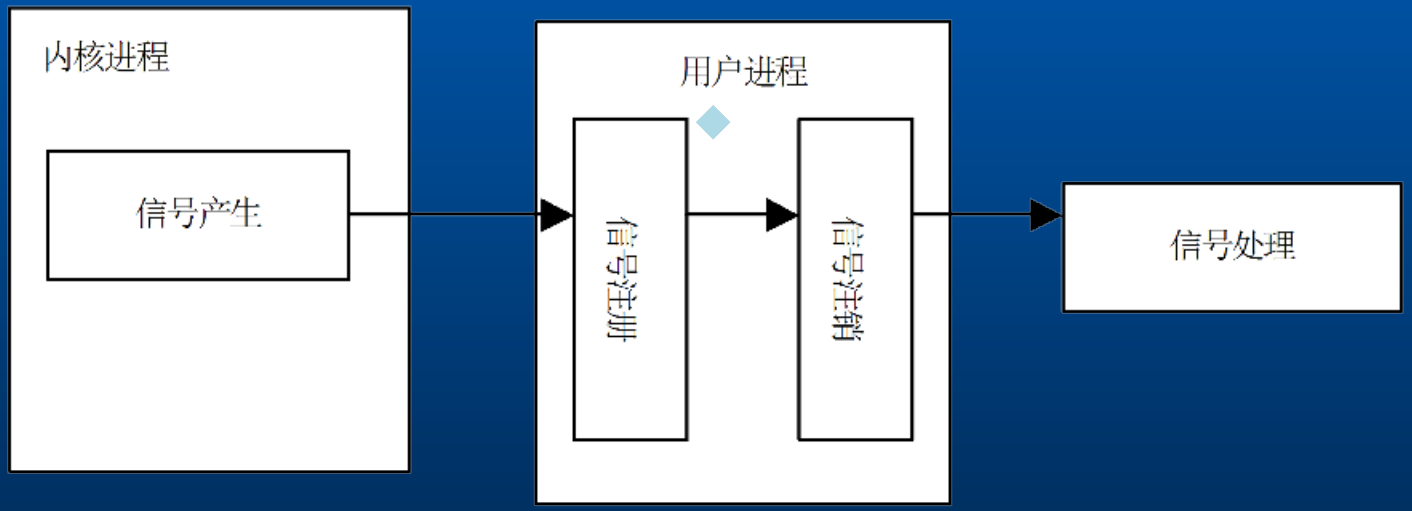
  一个不可靠信号的处理过程是这样的：如果发现该信号已经在进程中注册，那么就忽略该信号。因此，若前一个

信号概述（2）

- 信号还未注销又产生了相同的信号就会产生信号丢失。
  而当可靠信号发送给一个进程时，不管该信号是否已经在进程中注册，都会被再注册一次，因此信号就不会丢失。所有可靠信号都支持排队，而所有不可靠信号都不支持排队。
- 一个完整的信号生命周期可以分为3个重要阶段，这3个阶段由4个重要事件来刻画的：信号产生、信号在进程中注册、信号在进程中注销、执行信号处理函数
- 用户进程对信号的响应可以有3种方式。
  - 忽略信号，即对信号不做任何处理，但是有两个信号不能忽略，即SIGKILL及SIGSTOP。

信号概述（3）

- 捕捉信号，定义信号处理函数，当信号发生时，执行相应的自定义处理函数。
- 执行缺省操作，Linux对每种信号都规定了默认操作。



信号发送与捕捉（1）

- 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）。这里需要注意的是，它不仅可以中止进程（实际上发出SIGKILL信号），也可以向进程发送其他信号。
- 与kill()函数所不同的是，raise()函数允许进程向自身发送信号。

信号发送与捕捉（2）

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

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：要发送信号的进程号
		0：信号被发送到所有和当前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）
		<-1：信号发送给进程组号为-pid 的每一个进程
	sig：信号	
函数返回值	成功：0	
	出错：−1	

信号发送与捕捉（3）

- 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，它就向进程发送SIGALARM信号。要注意的是，一个进程只能有一个闹钟时间，如果在调用alarm()之前已设置过闹钟时间，则任何以前的闹钟时间都被新值所代替。
- pause()函数是用于将调用进程挂起直至捕捉到信号为止。这个函数很常用，通常可以用于判断信号是否已到。

信号发送与捕捉（4）

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，系统经过 seconds 秒之后向该进程发送 SIGALRM 信号
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

signal()函数 （1）

- 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。

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：指定信号代码	
	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：以前的信号处理配置	
	出错：-1	

signal()函数（2）

所需头文件	#include <signal.h>
函数原型	int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
函数传入值	signum：信号代码，可以为除 SIGKILL 及 SIGSTOP 外的任何一个特定有效的信号
	act：指向结构 sigaction 的一个实例的指针，指定对特定信号的处理
	oldact：保存原来对相应信号的处理
函数返回值	成功：0
	出错：-1

```
struct sigaction
{
        void (*sa_handler)(int signo);
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_restore)(void);
}
```

sa_handler是一个函数指针，指定信号处理函数，这里除可以是用户自定义的处理函数外，还可以为SIG_DFL（采用缺省的处理方式）或SIG_IGN（忽略信号）。它的处理函数只有一个参数，即信号值。
sa_mask是一个信号集，它可以指定在信号处理程序执行过程中哪些信号应当被屏蔽，在调用信号捕获函数之前，该信号集要加入到信号的信号屏蔽字中。
sa_flags中包含了许多标志位，是对信号进行处理的各个选择项。

信号集函数组（1）

- 使用信号集函数组处理信号时涉及一系列的函数，这些函数按照调用的先后次序可分为以下几大功能模块：创建信号集合、注册信号处理函数以及检测信号。
- 其中，创建信号集合主要用于处理用户感兴趣的一些信号，其函数包括以下几个。
  - sigemptyset()：将信号集合初始化为空。
  - sigfillset()：将信号集合初始化为包含所有已定义的信号的集合。
  - sigaddset()：将指定信号加入到信号集合中去。
  - sigdelset()：将指定信号从信号集合中删去。
  - sigismember()：查询指定信号是否在信号

信号集函数组（2）

- 注册信号处理函数主要用于决定进程如何处理信号。这里要注意的是，信号集里的信号并不是真正可以处理的信号，只有当信号的状态处于非阻塞状态时才会真正起作用。因此，首先使用sigprocmask()函数检测并更改信号屏蔽字（信号屏蔽字是用来指定当前被阻塞的一组信号，它们不会被进程接收），然后使用sigaction()函数来定义进程接收到特定信号之后的行为。检测信号是信号处理的后续步骤，因为被阻塞的信号不会传递给进程，所以这些信号就处于"未处理"状态（也就是进程不清楚它的存在）。sigpen()函数允许进程检测"未处理"信号，并进一步决定对它们作何处理。

信号集函数组 （3）

定义信号集合
sigemptyset()
sigfillset()
sigaddset() ,……

→

设置信号屏蔽位
sigprocmask()

→

定义信号处理函数
sa_mask
sa_handler
sigaction

→

测试信号
sigpending()

所需头文件	#include <signal.h>
函数原型	int sigemptyset(sigset_t *set)
	int sigfillset(sigset_t *set)
	int sigaddset(sigset_t *set, int signum)
	int sigdelset(sigset_t *set, int signum)
	int sigismember(sigset_t *set, int signum)
函数传入值	set：信号集
	signum：指定信号代码
函数返回值	成功：0（sigismember 成功返回 1，失败返回 0）
	出错：-1