第四章：信号

4.1 进程间通信概述

进程间通信(IPC:Inter Processes Communication)

进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源（例如打开的文件描述符）。

进程不是孤立的，不同的进程需要进行信息的交互和状态的传递等，因此需要进程间通信。

进程间通信功能：

数据传输：一个进程需要将它的数据发送给另一个进程。

资源共享：多个进程之间共享同样的资源。

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。

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。

linux 进程间通信(IPC)由以下几个部分发展而来

最初的 UNIX 进程间通信

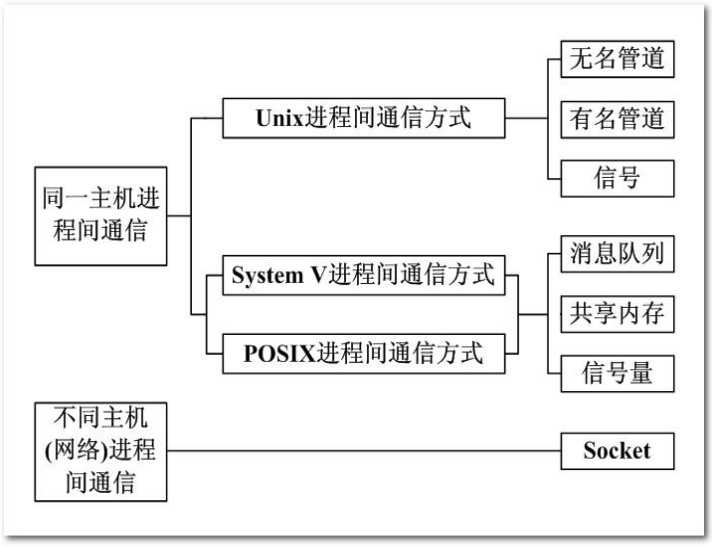
SYSTEM V 进程间通信

POSIX 进程间通信(POSIX:Portable Operating System interface 可移植操作系统接口）

Socket 进程间通信

Linux 把优势都继承了下来并形成了自己的 IPC

Linux 操作系统支持的主要进程间通信的通信机制



4.2 信号

4.2.1 概述

信号是软件中断，它是在软件层次上对中断机制的一种模拟。

信号可以导致一个正在运行的进程被另一个正在运行的异步进程中断，转而处理某一个突发事件。

信号是一种异步通信方式。

进程不必等待信号的到达，进程也不知道信号什么时候到达。

信号可以直接进行用户空间进程和内核空间进程的交互，内核进程可以利用它来通知用户空间进程发生了哪些系统事件。

每个信号的名字都以字符 SIG 开头。

每个信号和一个数字编码相对应，在头文件 signum.h 中，这些信号都被定义为正整数。

信号名定义路径：

/usr/include/i386-linux-gnu/bits/signum.h

在 Linux 下，要想查看这些信号和编码的对应关系，可使用命令：kill -l

```
edu@edu:~/share$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   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
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
edu@edu:~/share$
```

以下条件可以产生一个信号

1、当用户按某些终端键时，将产生信号。

例如：

终端上按“Ctrl+c”组合键通常产生中断信号 SIGINT、终端上按"Ctrl+\"键通常产生中断信号 SIGQUIT、终端上按"Ctrl+z"键通常产生中断信号 SIGSTOP。

2、硬件异常将产生信号。

除数为 0，无效的内存访问等。这些情况通常由硬件检测到，并通知内核,然后内核产生适当的信号发送给相应的进程。

3、软件异常将产生信号。

当检测到某种软件条件已发生，并将其通知有关进程时，产生信号。

4、调用 kill 函数将发送信号。

注意：接收信号进程和发送信号进程的所有者必须相同，或发送信号进程的所有者必须是超级用户。

5、运行 kill 命令将发送信号。

此程序实际上是使用 kill 函数来发送信号。也常用此命令终止一个失控的后台进程。

一个进程收到一个信号的时候，可以用如下方法进行处理：

1、执行系统默认动作

对大多数信号来说，系统默认动作是用来终止该进程。

2、忽略此信号

接收到此信号后没有任何动作。

3、执行自定义信号处理函数

用用户定义的信号处理函数处理该信号。

注意：

SIGKILL 和 SIGSTOP 不能更改信号的处理方式，因为它们向用户提供了一种使进程终止的可靠方法。

4.2.2 信号的基本操作

4.2.2.1 kill 函数

#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signum);

功能：

给指定进程发送信号。

参数：

pid：详见下页

signum：信号的编号

返回值：

成功返回 0，失败返回 -1。

pid 的取值有 4 种情况：

pid>0：将信号传送给进程 ID 为 pid 的进程。
pid=0：将信号传送给当前进程所在进程组中的所有进程。
pid=-1：将信号传送给系统内所有的进程。
pid<-1：将信号传给指定进程组的所有进程。这个进程组号等于 pid 的绝对值。

例：01_kill.c　父进程给子进程发送信号

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

```
#include <signal.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
    pid_t pid;
    pid = fork();
    if(pid<0)
        perror("fork");
    if(pid==0)
    {
        int i = 0;
        for(i=0; i<5; i++)
        {
            printf("in son process\n");
            sleep(1);
        }
    }
    else
    {
        printf("in father process\n");
        sleep(2);
        printf("kill sub process now \n");
        kill(pid, SIGINT);
    }
    return 0;
}
```

注意：

使用 kill 函数发送信号，接收信号进程和发送信号进程的所有者必须相同，或者发送信号进程的所有者是超级用户。

4.2.2.2 alarm 函数

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

功能：

在 seconds 秒后，向调用进程发送一个 SIGALRM 信号，SIGALRM 信号的默认动作是终止调用 alarm 函数的进程。

返回值：

若以前没有设置过定时器，或设置的定时器已超时，返回 0；否则返回定时器剩余的秒数，并重新设定定时器。

例：02_alarm.c

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int seconds = 0;
    seconds = alarm(5);
    printf("seconds = %d\n", seconds);
    sleep(2);
    seconds = alarm(5);
    printf("seconds = %d\n", seconds);
    while(1);
    return 0;
}
```

4.2.2.3 raise 函数

#include <signal.h>

int raise(int signum);

功能：

给调用进程本身发送一个信号。

参数：

signum：信号的编号。

返回值：

成功返回 0，失败返回 -1。

例：03_raise.c

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

int main(int argc, char *argv[])
{
    printf("in raise function\n");
    sleep(2);
    raise(SIGALRM);
    sleep(10);
    return 0;
}
```

4.2.2.4 abort 函数

#include <stdlib.h>
void abort(void);
功能：
向进程发送一个 SIGABRT 信号，默认情况下进程会退出。
注意：
即使 SIGABRT 信号被加入阻塞集，一旦进程调用了 abort 函数，进程也还是会被终止，且在终止前会刷新缓冲区，关文件描述符。

4.2.2.5 pause 函数

#include <unistd.h>
int pause(void);
功能：
将调用进程挂起直至捕捉到信号为止。这个函数通常用于判断信号是否已到。
返回值：
直到捕获到信号，pause 函数才返回-1，且 errno 被设置成 EINTR。
例：04_pause.c

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("in pause function\n");
    pause();
    return 0;
}
```

4.2.2.6 signal 函数

进程接收到信号后的处理方式
1、执行系统默认动作
2、忽略此信号
3、执行自定义信号处理函数
程序中可用函数 signal()改变信号的处理方式。

#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum,sighandler_t handler);

功能：

注册信号处理函数（不可用于 SIGKILL、SIGSTOP 信号），即确定收到信号后处理函数的入口地址。

参数：

signum：信号编号

handler 的取值：

忽略该信号：SIG_IGN

执行系统默认动作：SIG_DFL

自定义信号处理函数：信号处理函数名

返回值：

成功：返回函数地址，该地址为此信号上一次注册的信号处理函数的地址。

失败：返回 SIG_ERR

例：05_signal_1.c 注册信号处理函数

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

void signal_handler(int signo)
{
    if(signo==SIGINT)
        printf("recv SIGINT\n");
    if(signo==SIGQUIT)
        printf("recv SIGQUIT\n");
}

int main(int argc, char *argv[])
{
    printf("wait for SIGINT OR SIGQUIT\n");
    signal(SIGINT, signal_handler);
    signal(SIGQUIT, signal_handler);

    pause();
    pause();
    return 0;
}
```

例：05_signal_2.c 验证 signal 函数的返回值

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

```
typedef void (*sighandler_t)(int);

void fun1(int signo)
{
    printf("in fun1\n");
}

void fun2(int signo)
{
    printf("in fun2\n");
}

int main(int argc, char *argv[])
{
    sighandler_t previous = NULL;

    previous = signal(SIGINT,fun1);
    if(previous == NULL)
    {
        printf("return fun addr is NULL\n");
    }
    previous = signal(SIGINT,fun2);
    if(previous == fun1)
    {
        printf("return fun addr is fun1\n");
    }
    previous = signal(SIGQUIT,fun1);
    if(previous == NULL)
    {
        printf("return fun addr is NULL\n");
    }
    return 0;
}
```

4.2.3 可重入函数

可重入函数是指函数可以由多个任务并发使用，而不必担心数据错误。

编写可重入函数：

1、不使用（返回）静态的数据、全局变量（除非用信号量互斥）。

2、不调用动态内存分配、释放的函数。

3、不调用任何不可重入的函数（如标准 I/O 函数）。

注：

即使信号处理函数使用的都是可重入函数(常见的可重入函数)，也要注意进入处理函数时，首先要保存 errno 的值，结束时，再恢复原值。因为，信号处理过程中，errno 值随时可能被改变。

常见的可重入函数列表：

accept	fchmod	lseek	sendto	stat
access	fchown	lstat	setgid	symlink
aio_error	fcntl	mkdir	setpgid	sysconf
aio_return	fdatasync	mkfifo	setsid	tcdrain
aio_suspend	fork	open	setsockopt	tcflow
alarm	fpathconf	pathconf	setuid	tcflush
bind	fstat	pause	shutdown	tcgetattr
cfgetispeed	fsync	pipe	sigaction	tcgetpgrp
cfgetospeed	ftruncate	poll	sigaddset	tcsendbreak
cfsetispeed	getegid	posix_trace_event	sigdelset	tcsetattr
cfsetospeed	geteuid	pselect	sigemptyset	tcsetpgrp
chdir	getgid	raise	sigfillset	time
chmod	getgroups	read	sigismember	timer_getoverrun
chown	getpeername	readlink	signal	timer_gettime
clock_gettime	getpgrp	recv	sigpause	timer_settime
close	getpid	recvfrom	sigpending	times
connect	getppid	recvmsg	sigprocmask	umask
creat	getsockname	rename	sigqueue	uname
dup	getsockopt	rmdir	sigset	unlink
dup2	getuid	select	sigsuspend	utime
execle	kill	sem_post	sleep	wait
execve	link	send	socket	waitpid
_Exit & _exit	listen	sendmsg	socketpair	write

4.2.3 信号集

信号集概述

一个用户进程常常需要对多个信号做出处理。为了方便对多个信号进行处理，在 Linux 系统中引入了信号集。信号集是用来表示多个信号的数据类型。

信号集数据类型

sigset_t

定义路径：

/usr/include/i386-linux-gnu/bits/sigset.h

信号集相关的操作主要有如下几个函数：

sigemptyset
sigfillset
sigismember
sigaddset
sigdelset

4.2.3.1 sigemptyset 函数

初始化一个空的信号集

```
#include <signal.h>
int sigemptyset(sigset_t *set);
```

功能：

初始化由 set 指向的信号集，清除其中所有的信号即初始化一个空信号集。

参数：

set：信号集标识的地址，以后操作此信号集，对 set 进行操作就可以了。

返回值：

成功返回 0，失败返回 -1。

4.2.3.2 sigfillset 函数

初始化一个满的信号集

```
#include <signal.h>
int sigfillset(sigset_t *set);
```

功能：

初始化信号集合 set，将信号集合设置为所有信号的集合。

参数：

信号集标识的地址，以后操作此信号集，对 set 进行操作就可以了。

返回值：

成功返回 0，失败返回 -1。

4.2.3.3 sigismember 函数

判断某个集合中是否有某个信号

```
#include <signal.h>
int sigismember(const sigset_t *set,int signum);
```

功能：

查询 signum 标识的信号是否在信号集合 set 之中。

参数：

set：信号集标识符号的地址。

signum：信号的编号。

返回值：

在信号集中返回 1，不在信号集中返回 0

错误，返回 -1

4.2.3.4 sigaddset 函数

向某个集合中添加一个信号

```
#include <signal.h>
int sigaddset(sigset_t *set, int signum);
```

功能：

将信号 signum 加入到信号集合 set 之中。

参数：

set：信号集标识的地址。

signum：信号的编号。

返回值：

成功返回 0，失败返回 -1。

4.2.3.5 sigdelset 函数

从某个信号集中删除一个信号

```
#include <signal.h>
int sigdelset(sigset_t *set, int signum);
```

功能：

将 signum 所标识的信号从信号集合 set 中删除。

参数：

set：信号集标识的地址。

signum：信号的编号。

返回值：

成功：返回 0

失败：返回 -1

例：06_signal_set.c 创建一个空的信号集合，向集合中添加信号，判断集合中是否有这个信号

```
#include <signal.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    sigset_t set;
    int ret = 0;
```

```
    sigemptyset(&set);
    ret = sigismember(&set, SIGINT);
    if(ret == 0)
        printf("SIGINT is not a member of sigprocmask \nret = %d\n", ret);

    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGQUIT);

    ret = sigismember(&set, SIGINT);
    if(ret == 1)
        printf("SIGINT is a member of sigprocmask \nret = %d\n", ret);

    return 0;
}
```

4.2.4 信号阻塞集(屏蔽集、掩码)

每个进程都有一个阻塞集，它用来描述哪些信号递送到该进程的时候被阻塞(在信号发生时记住它，直到进程准备好时再将信号通知进程)。

所谓阻塞并不是禁止传送信号，而是暂缓信号的传送。若将被阻塞的信号从信号阻塞集中删除，且对应的信号在被阻塞时发生了，进程将会收到相应的信号。

4.2.4.1 sigprocmask 函数

创建一个阻塞集合

#include <signal.h>

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

功能：

检查或修改信号阻塞集，根据 how 指定的方法对进程的阻塞集合进行修改，新的信号阻塞集由 set 指定，而原先的信号阻塞集合由 oldset 保存。

参数：

how：信号阻塞集合的修改方法。

set：要操作的信号集地址。

oldset：保存原先信号集地址。

how：

SIG_BLOCK：向信号阻塞集合中添加 set 信号集

SIG_UNBLOCK：从信号阻塞集合中删除 set 集合

SIG_SETMASK：将信号阻塞集合设为 set 集合

注：若 set 为 NULL，则不改变信号阻塞集合，函数只把当前信号阻塞集合保存到 oldset 中。

返回值：

成功：返回 0

失败：返回 -1

例：06_sigprocmask.c

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

int main(int argc, char *argv[])
{
    sigset_t set;
    int i=0;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    while(1)
    {
        sigprocmask(SIG_BLOCK, &set, NULL);
        for(i=0; i<5; i++)
        {
            printf("SIGINT signal is blocked\n");
            sleep(1);
        }
        sigprocmask(SIG_UNBLOCK, &set, NULL);
        for(i=0; i<5; i++)
        {
            printf("SIGINT signal unblocked\n");
            sleep(1);
        }
    }
    return 0;
}
```