# Signal

## 基本概念

信号有时也称之为软件中间，和硬件中断(IO准备就绪、键盘键入字符等)的相似之处就在于均会打断原有程序的正常执行

硬件中断与异常

- 中断：中断常常由外部设备产生，如socket接收缓冲区内容可读，DMA将硬盘文件内容复制至内核指定的缓冲区中后发起中断等
- 异常：异常往往发生在程序运行的内部，例如当程序访问的虚拟内存在虚拟内存页表中无对应表项时，则会发生缺页异常，由内核新建页表项，并将所需要的内容载入内存

信号的分类

- 标准信号：由内核定义，且用于内核向进程发送通知事件。**内核不会对标准信号做排队处理: 当进程执行某一信号处理函数时，此时若有多个相同的信号产生，则信号的发送将处于阻塞状态，并且在后续只会传递一次**
- 实时信号：由内核定义，用于弥补标准信号的不足之处，其范围通常为32~65。**内核会对实时信号进行排队处理，且具有优先级的概念: 信号编码越小，优先级越高，就会越先发送给进程**

进程处理信号的默认行为

- 忽略信号：进程就当无事发生过，该干嘛还干嘛
- 终止(杀死)进程：需要注意是，信号终止进程区别与调用exit()正常结束进程
- 产生核心转储文件(Core Dump)
  - 核心转储文件包含对进程虚拟内存的镜像，可将其载入调试器中进行debug
  - 例如C程序引用了一个未初始化的指针
- 暂停进程的执行
- 于之前暂停后再度恢复进程的执行

当进程未对某些信号注册信号处理函数时，进程就会采用默认行为来处理收到的信号

## Linux标准信号(重要)

- SIGALRM：当调用alarm()或者是settimer()设置的定时器到期时，内核将发送此信号至进程。可用于实现带有超时时间的scoket相关调用：该信号默认会终止进程
- SIGCHLD：**当父进程的某一个子进程终止(包括异常退出的情况)时，内核将向父进程发送该信号，表明子进程已终止**
- SIGINT：**当用户键入Ctrl + C时，终端驱动程序则会发送该信号给前台进程组，默认为终止进程**
- SIGKILL：当使用kill -9 pid时，则会发送必杀信号给对应的进程：**该信号无法对其添加用户自己的信号处理函数，所以总是能杀死进程**
- SIGSTOP：当键入Ctrl + Z时，则会产生该必停信号，无法被捕获，总是能够停止进程的执行
- SIGTERM：**当使用kill pid时，则会发送该信号给对应的进程**
- SIGIO：当特定类型(终端或套接字)的打开文件描述符发生I/O事件时产生该信号

## 改变信号处置: signal()

除了SIGKILL以及SIGSTOP两个信号行为无法改变以外，其余信号均可使用signal()系统调用为其添加用户所编写的信号处理函数

*原型: void ( *signal(int sig, void(*handler)(int)) ) (int);*

- signal()调用将返回原有的信号处理函数，错误时将返回SIG_ERR
- 信号处理函数无返回值，接收整型参数，其实就是信号的整型编号

signal()调用的一个局限之处就在于，它没有`GET`方法，也就是说，无法直接获取到某一个信号当前进程的处理函数：只能连续调用两次signal()获得

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

void signalHandler(int sig) {
    printf("Ouch \n");
}

int main() {
    // Ctrl + C
    if (signal(SIGINT, signalHandler) == SIG_ERR) {
        perror("signal terminate");
    }

    // Ctrl + Z, SIGSTOP信号无法被捕获, 所以总是能够停止进程的执行
    if (signal(SIGSTOP, signalHandler) == SIG_ERR) {
        perror("signal stop");
    }

    for (int i = 0; i < 10000; i++) {
        printf("%d\n", i);
        sleep(3);
    }
}
```

- 当对SIGSTOP改变信号处理函数时，会返回错误
- 该段代码一个最有意思的地方在于：当每次键入Ctrl + C时，都将使得sleep(3)提前结束
- 此外，尽量不要在信号处理函数中使用非重入的函数调用，如printf标准I/O库函数通常会带有缓冲，其输出结果可能包含原有缓冲区中的一些信息

## 发送信号: kill()

原型: int kill(pid_t pid, int sig)：返回0表示成功，返回-1表示失败

参数pid可有4种情况

- pid > 0：表示唯一的特定进程，信号将会发送给PID为pid的进程
- pid == 0：表示当前调用的同一进程组，包括调用进程自身
- pid == -1：如果特权级进程发起这一调用，那么会发送信号给系统中的所有进程
- pid < -1：表示该pid的绝对值所代表的进程组

如果将sig参数指定为0，相当于发送空信号，可以用于检测指定的pid进程是否存在

- kill调用失败，且errno为ESRCH
- kill调用失败也可能是权限不够的问题，例如普通用户向init进程发送信号，别人当然不会鸟你

## 信号集

在很多时候，我们需要将一个一个的信号(整型)打包成一个集合，进行传递或者是进行系统调用，这种打包的结构称为信号集，类型为sigset_t：Linux中通过位掩码来实现sigset_t，当然也可以使用整型数组实现

两个初始化方法

- *int sigemptyset(sigset_t *set);*：初始化一个未包含任何成员信号的信号集，类似于对指针变量的bzero或者是memset
- *int sigfillset(sigset_t *set);*：初始化一个包含所有成员信号的信号集，包括实时信号

*因为不同的内核对于sigset_t类型的实现不同，所以必须使用这两个方法对信号集进行初始化*

向信号集中添加/删除信号

- *int sigaddset(sigset_t *set, int sig);*：向某一信号集中添加信号
- *int sigdelset(sigset_t *set, int sig);*：向某一信号集中移除信号

判断信号集中是否存在某信号：*int sigismember(const sigset_t *sigset, int sig);*

完全可以将信号集当做是一个容器来对待，这和Java中的List以及Python中的list没什么区别

## 信号掩码

内核会为每个进程维护一个信号掩码，即一组信号，并将阻塞其针对该进程的传递：例如某一进程不想收到SIGINT信号，则可将SIGINT信号加入至该进程的信号掩码中，当内核向进程发送该信号时，将会被阻塞。换而言之，进程将不会收到该信号的传递

向信号掩码中添加信号：*int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);*

- 该调用算是比较典型的策略模式应用，策略由how参数指定
- how参数的策略
  - SIG_BLOCK：将set所指向的信号集内信号添加至进程信号掩码中：添加
  - SIG_UNBLOCK：将set所指向信号集中的信号从进程信号掩码中移除：移除
  - SIG_SETMASK：将set所指向信号集赋给进程信号掩码：覆盖更新
- 可以使用该调用来获取当前的进程信号掩码：set参数指定为NULL，oldset参数不为空

```c
sigset_t blockSet;
sigfillset(&blockSet);
if (sigprocmask(SIG_BLOCK, &blockSet, NULL) == -1) {
    perror("sigprocmask");
}
```

阻塞除SIGKILL和SIGSTOP以外的所有信号

标准信号不会存在排队处理：假设进程一开始阻塞了SIGINT信号的传递，期间产生了多个SIGINT信号，那么后续进程从信号掩码中移除SIGINT信号时，将会只收到一次该信号的传递：实时信号则可以有有限的队列

**且就算没有阻塞信号，其所收到的信号也可能要比发送给它的要少的多。所以，这也是为什么信号不适合作为IPC工具的原因**