信号

相关概念

■ 信号是 Linux 进程间通信的最古老的方式之一，是事件发生时对进程的通知机制，有时也称之为软件中断，它是在软件层次上对中断机制的一种模拟，是一种异步通信的方式。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。
■ 发往进程的诸多信号，通常都是源于内核。引发内核为进程产生信号的各类事件如下：
1、对于前台进程，用户可以通过输入特殊的终端字符来给它发送信号。比如输入Ctrl+C 通常会给进程发送一个中断信号。
2、硬件发生异常，即硬件检测到一个错误条件并通知内核，随即再由内核发送相应信号给相关进程。比如执行一条异常的机器语言指令，诸如被 0 除，或者引用了无法访问的内存区域。
3、系统状态变化，比如 alarm 定时器到期将引起 SIGALRM 信号，进程执行的 CPU 时间超限，或者该进程的某个子进程退出。
4、运行 kill 命令或调用 kill 函数。
■ 使用信号的两个主要目的是：
1、让进程知道已经发生了一个特定的事情。
2、强迫进程执行它自己代码中的信号处理程序。
■ 信号的特点：
1、简单
2、不能携带大量信息
3、满足某个特定条件才发送
4、优先级比较高
■ 查看系统定义的信号列表：kill –l
■ 前 31 个信号为常规信号，其余为实时信号。

Linux信号一览

编号	信号名称	对应事件	默认动作
1	SIGHUP	用户退出shell时，由该shell启动的所有进程将收到这个信号	终止进程
2	**SIGINT**	当用户按下了<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号	终止进程
3	**SIGQUIT**	用户按下<Ctrl+\>组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出些信号	终止进程
4	SIGILL	CPU检测到某进程执行了非法指令	终止进程并产生core文件
5	SIGTRAP	该信号由断点指令或其他 trap指令产生	终止进程并产生core文件
6	SIGABRT	调用abort函数时产生该信号	终止进程并产生core文件
7	SIGBUS	非法访问内存地址，包括内存对齐出错	终止进程并产生core文件
8	SIGFPE	在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误	终止进程并产生core文件

编号	信号名称	对应事件	默认动作
9	**SIGKILL**	无条件终止进程。该信号不能被忽略，处理和阻塞	终止进程，可以杀死任何进程
10	SIGUSE1	用户定义的信号。即程序员可以在程序中定义并使用该信号	终止进程
11	**SIGSEGV**	指示进程进行了无效内存访问(段错误)	终止进程并产生core文件
12	SIGUSR2	另外一个用户自定义信号，程序员可以在程序中定义并使用该信号	终止进程
13	**SIGPIPE**	Broken pipe向一个没有读端的管道写数据	终止进程
14	SIGALRM	定时器超时，超时的时间 由系统调用alarm设置	终止进程
15	SIGTERM	程序结束信号，与SIGKILL不同的是，该信号可以被阻塞和终止。通常用来要示程序正常退出。执行shell命令Kill时，缺省产生这个信号	终止进程
16	SIGSTKFLT	Linux早期版本出现的信号，现仍保留向后兼容	终止进程

编号	信号名称	对应事件	默认动作
17	**SIGCHLD**	子进程结束时，父进程会收到这个信号	忽略这个信号
18	**SIGCONT**	如果进程已停止，则使其继续运行	继续/忽略
19	**SIGSTOP**	停止进程的执行。信号不能被忽略，处理和阻塞	为终止进程
20	SIGTSTP	停止终端交互进程的运行。按下<ctrl+z>组合键时发出这个信号	暂停进程
21	SIGTTIN	后台进程读终端控制台	暂停进程
22	SIGTTOU	该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生	暂停进程
23	SIGURG	套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达	忽略该信号
24	SIGXCPU	进程执行时间超过了分配给该进程的CPU时间，系统产生该信号并发送给该进程	终止进程

编号	信号名称	对应事件	默认动作
25	SIGXFSZ	超过文件的最大长度设置	终止进程
26	SIGVTALRM	虚拟时钟超时时产生该信号。类似于SIGALRM，但是该信号只计算该进程占用CPU的使用时间	终止进程
27	SGIPROF	类似于SIGVTALRM，它不公包括该进程占用CPU时间还包括执行系统调用时间	终止进程
28	SIGWINCH	窗口变化大小时发出	忽略该信号
29	SIGIO	此信号向进程指示发出了一个异步IO事件	忽略该信号
30	SIGPWR	关机	终止进程
31	SIGSYS	无效的系统调用	终止进程并产生core文件
34 ~ 64	SIGRTMIN ~ SIGRTMAX	LINUX的实时信号，它们没有固定的含义（可以由用户自定义）	终止进程

信号的五种默认动作

■ 查看信号的详细信息：man 7 signal
■ 信号的 5 中默认处理动作
1、 Term 终止进程
2、 Ign 当前进程忽略掉这个信号
3、 Core 终止进程，并生成一个Core文件
4、 Stop 暂停当前进程
5、 Cont 继续执行当前被暂停的进程
■ 信号的几种状态：产生、未决、递达
■ SIGKILL 和 SIGSTOP 信号不能被捕捉、阻塞或者忽略，只能执行默认动作。

信号的相关函数

```
■ int kill(pid_t pid, int sig);
■ int raise(int sig);
■ void abort(void);
■ unsigned int alarm(unsigned int seconds);
■ int setitimer(int which, const struct itimerval *new_val, struct itimerval
*old_value);
■ sighandler_t signal(int signum, sighandler_t handler);
■ int sigaction(int signum, const struct sigaction *act, struct sigaction
*oldact);
```

kill、raise和abort函数

说明

这三个函数都是给进程发送信号使用的。执行时都有两个基本参数：进程号和信号宏或者是信号的编号。

```
/*
函数原型：
    #include <sys/types.h>
    #include <signal.h>

    int kill(pid_t pid, int sig);
        - 功能：给任何的进程或者进程组pid, 发送任何的信号 sig
        - 参数：
            - pid ：
                > 0 ： 将信号发送给指定的进程
                = 0 ： 将信号发送给当前的进程组
                = -1 ： 将信号发送给每一个有权限接收这个信号的进程
                < -1 ： 这个pid=某个进程组的ID取反 （-12345）
            - sig ： 需要发送的信号的编号或者是宏值，0表示不发送任何信号

        kill(getppid(), 9); //向父进程发送9号信号，杀死进程
        kill(getpid(), 9);  //向当前进程发送9号信号

    int raise(int sig);
        - 功能：给当前进程发送信号
        - 参数：
            - sig ： 要发送的信号
        - 返回值：
            - 成功 0
```

```
            - 失败 非0
        相当于 kill(getpid(), sig);

    void abort(void);
        - 功能： 发送SIGABRT信号给当前的进程，杀死当前进程
        相当于 kill(getpid(), SIGABRT);
*/
```

代码示例

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

int main(int argc, char **argv)
{
    pid_t pid = fork();

    if (pid > 0)
    {
        //父进程
        printf("parent process \n");
        sleep(2);
        printf("kill child process \n");
        // kill(pid, SIGINT);  //向指定的线程发送SIGINT信号
        // kill(0, SIGINT); //向当前线程所在组发送SIGINT信号，最后的输出不执行

        printf("child has been killed \n");
    }
    else if (pid == 0)
    {
        //子进程
        for (int i = 0; i < 5; i++)
        {
            printf("child process %d \n", i);
            sleep(1);
        }
    }

    return 0;
}
```

alarm定时器函数

说明

在这里插入代码片

```bash

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

        - 功能：设置定时器（闹钟）。函数调用，开始倒计时，当倒计时为0的时候，
                函数会给当前的进程发送一个信号：SIGALARM
```

```
        - 参数：
            seconds：倒计时的时长，单位：秒。如果参数为0，定时器无效（不进行倒计时，不发信号）。
                        取消一个定时器，通过alarm(0)。
        - 返回值：
            - 之前没有定时器，返回0
            - 之前有定时器，返回之前的定时器剩余的时间

    - SIGALARM ：默认终止当前的进程，每一个进程都有且只有唯一的一个定时器。
        alarm(10);   -> 返回0
        过了1秒
        alarm(5);    -> 返回9

    alarm(100) -> 该函数是不阻塞的

    实际的时间 = 内核时间 + 用户时间 + 消耗的时间
    进行文件IO操作的时候比较浪费时间

    定时器，与进程的状态无关（自然定时法）。无论进程处于什么状态，alarm都会计时。
```

代码示例

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

int main(int argc, char **argv)
{
    //利用定时器判断1秒中能在控制台输出多少个数字
    alarm(1); //定时一秒钟

    int i = 1;
    while (1)
    {
        printf("%d\n", i++);
    }

    return 0;
}
```

setitimer周期定时器函数

说明

```c
/*
    函数原型
    #include <sys/time.h>
    int setitimer(int which, const struct itimerval *new_value, struct itimerval *old_value);
```

```
    - 功能：设置周期定时器（周期闹钟）。可以替代alarm函数。精度微妙us，可以实现周期性定时
        - 参数：
            - which : 定时器以什么时间计时
              ITIMER_REAL: 真实时间，时间到达，发送 SIGALRM    常用
              ITIMER_VIRTUAL: 用户时间，时间到达，发送 SIGVTALRM
              ITIMER_PROF: 以该进程在用户态和内核态下所消耗的时间来计算，时间到达，发送
SIGPROF

            - new_value: 设置定时器的属性

                struct itimerval {      // 定时器的结构体
                struct timeval it_interval;  // 每个阶段的时间，间隔时间
                struct timeval it_value;     // 延迟多长时间执行定时器
                };
                struct timeval {        // 时间的结构体
                    time_t      tv_sec;     //  秒数
                    suseconds_t tv_usec;    //  微秒
                };

            - old_value : 记录上一次的定时的时间参数，一般不使用，指定NULL

            - 函数执行机制：
                先对it_value倒计时，当it_value为零时触发信号，然后重置为it_interval，继
续对it_value倒计时，一直这样循环下去。
                也就是程序一开始是按照it_value进行一段延迟，到了时间之后开始周期周期定
时器，这个周期定时器的时间则是按照it_interval
                注意：再it_value到期之后其实就已经会执行第一次定时器了，而不是还要等上
it_interval的时间才执行第一次定时器
        - 返回值：
            成功 0
            失败 -1 并设置错误号
*/
```

代码示例

```c
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>

// 过3秒以后，每隔2秒钟定时一次
int main()
{

    struct itimerval new_value;

    // 设置间隔的时间
    new_value.it_interval.tv_sec = 2;
    new_value.it_interval.tv_usec = 0;

    // 设置延迟的时间,3秒之后开始第一次定时。之后每个2秒进行周期性的定时器操作。该代码中，到了3
秒执行第一次定时器会终止程序
    new_value.it_value.tv_sec = 3;
    new_value.it_value.tv_usec = 0;

    int ret = setitimer(ITIMER_VIRTUAL, &new_value, NULL); // 非阻塞的
    printf("定时器开始了...\n");
```

```
    if (ret == -1)
    {
        perror("setitimer");
        exit(0);
    }

    getchar();

    return 0;
}
```

信号捕捉函数之一signal

说明

每个信号触发之后都会有默认的动作，为了能够自己处理信号的动作，需要使用信号捕捉函数。

signal函数

```
/*
    #include <signal.h>
    typedef void (*sighandler_t)(int);
    sighandler_t signal(int signum, sighandler_t handler);
        - 功能：设置某个信号的捕捉行为
        - 参数：
            - signum: 要捕捉的信号
            - handler: 捕捉到信号要如何处理
                - SIG_IGN ：  忽略信号
                - SIG_DFL ：  使用信号默认的行为
                - 回调函数 ：  这个函数是内核调用，程序员只负责写，捕捉到信号后如何去处理信
号。
                回调函数：
                    - 需要程序员实现，提前准备好的，函数的类型根据实际需求，看函数指针的定义
                    - 不是程序员调用，而是当信号产生，由内核调用
                    - 函数指针是实现回调的手段，函数实现之后，将函数名放到函数指针的位置就可
以了。

        - 返回值：
            成功，返回上一次注册的信号处理函数的地址。第一次调用返回NULL
            失败，返回SIG_ERR，设置错误号

    SIGKILL SIGSTOP不能被捕捉，不能被忽略。
*/
```

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void myalarm(int num) {
    printf("捕捉到了信号的编号是：%d\n", num);
    printf("xxxxxxx\n");
}

// 过3秒以后，每隔2秒钟定时一次
int main() {
```

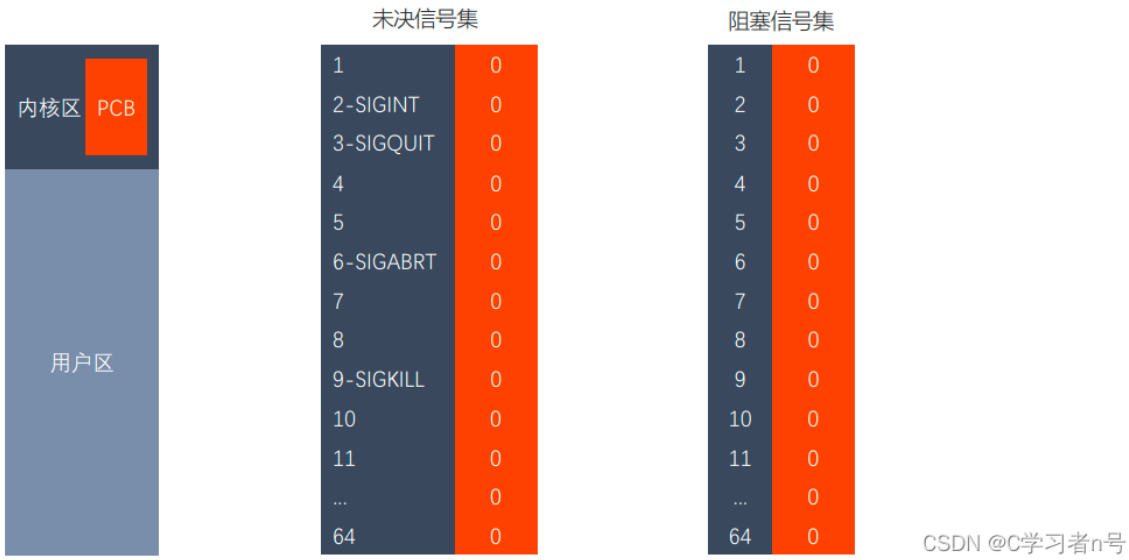
```
    // 注册信号捕捉
    // signal(SIGALRM, SIG_IGN);
    // signal(SIGALRM, SIG_DFL);
    // void (*sighandler_t)(int); 函数指针，int类型的参数表示捕捉到的信号的值。
    signal(SIGALRM, myalarm);

    struct itimerval new_value;

    // 设置间隔的时间
    new_value.it_interval.tv_sec = 2;
    new_value.it_interval.tv_usec = 0;

    // 设置延迟的时间,3秒之后开始第一次定时
    new_value.it_value.tv_sec = 3;
    new_value.it_value.tv_usec = 0;

    int ret = setitimer(ITIMER_REAL, &new_value, NULL); // 非阻塞的
    printf("定时器开始了...\n");

    if(ret == -1) {
        perror("setitimer");
        exit(0);
    }

    getchar();

    return 0;
}
```

信号集及相关函数

说明

■ 许多信号相关的系统调用都需要能表示一组不同的信号，多个信号可使用一个称之为信号集的数据结构来表示，其系统数据类型为 sigset_t。
■ 在 PCB 中有两个非常重要的信号集。一个称之为 “阻塞信号集” ，另一个称之为“未决信号集” 。这两个信号集都是内核使用位图机制来实现的。但操作系统不允许我们直接对这两个信号集进行位操作。而需自定义另外一个集合，借助信号集操作函数来对 PCB 中的这两个信号集进行修改。
■ 信号的 “未决” 是一种状态，指的是从信号的产生到信号被处理前的这一段时间。
■ 信号的 “阻塞” 是一个开关动作，指的是阻止信号被处理，但不是阻止信号产生。
■ 信号的阻塞就是让系统暂时保留信号留待以后发送。由于另外有办法让系统忽略信号，所以一般情况

下信号的阻塞只是暂时的，只是为了防止信号打断敏感的操作。



机制理解

1.用户通过键盘 Ctrl + C, 产生2号信号SIGINT (信号被创建)

2.信号产生但是没有被处理 （未决）
- 在内核中将所有的没有被处理的信号存储在一个集合中 （未决信号集）
- SIGINT信号状态被存储在第二个标志位上
  - 这个标志位的值为0， 说明信号不是未决状态
  - 这个标志位的值为1， 说明信号处于未决状态

3.这个未决状态的信号，需要被处理，处理之前需要和另一个信号集（阻塞信号集），进行比较
- 阻塞信号集默认不阻塞任何的信号
- 如果想要阻塞某些信号需要用户调用系统的API

4.在处理的时候和阻塞信号集中的标志位进行查询，看是不是对该信号设置阻塞了
- 如果没有阻塞，这个信号就被处理
- 如果阻塞了，这个信号就继续处于未决状态，直到阻塞解除，这个信号就被处理

综上： 信号产生 --> 信号未决 ---> 阻塞判断 ---> 不阻塞，直接处理；阻塞，等待中，仍处于未决状态

函数集合一

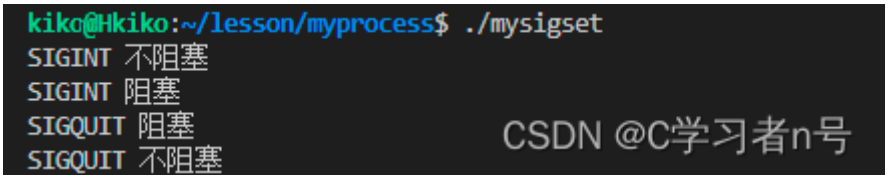
```
/*
    以下信号集相关的函数都是对自定义的信号集进行操作。以下函数对于修改和添加操作都是对自定义的阻塞信号集进行操作。未决信号是要程序触发才能实现的。

    int sigemptyset(sigset_t *set);
        - 功能：清空信号集中的数据,将信号集中的所有的标志位置为0
        - 参数：set,传出参数，需要操作的信号集
        - 返回值：成功返回0， 失败返回-1

    int sigfillset(sigset_t *set);
        - 功能：将信号集中的所有的标志位置为1
        - 参数：set,传出参数，需要操作的信号集
        - 返回值：成功返回0， 失败返回-1

    int sigaddset(sigset_t *set, int signum);
        - 功能：设置信号集中的某一个信号对应的标志位为1，表示阻塞这个信号
        - 参数：
```

```
            - set：传出参数，需要操作的信号集
            - signum：需要设置阻塞的那个信号
        - 返回值：成功返回0，  失败返回-1

    int sigdelset(sigset_t *set, int signum);
        - 功能：设置信号集中的某一个信号对应的标志位为0，表示不阻塞这个信号
        - 参数：
            - set：传出参数，需要操作的信号集
            - signum：需要设置不阻塞的那个信号
        - 返回值：成功返回0，  失败返回-1

    int sigismember(const sigset_t *set, int signum);
        - 功能：判断某个信号是否阻塞
        - 参数：
            - set：需要操作的信号集
            - signum：需要判断的那个信号
        - 返回值：
            1 ：  signum被阻塞
            0 ：  signum不阻塞
            -1 ：  失败

*/
```

示例1

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

int main()
{

    // 创建一个信号集
    sigset_t set;

    // 清空信号集的内容
    sigemptyset(&set);

    // 判断 SIGINT 是否在信号集 set 里
    int ret = sigismember(&set, SIGINT);
    if (ret == 0)
    {
        printf("SIGINT 不阻塞\n");
    }
    else if (ret == 1)
    {
        printf("SIGINT 阻塞\n");
    }

    // 添加几个信号到信号集中
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGQUIT);

    // 判断SIGINT是否在信号集中
    ret = sigismember(&set, SIGINT);
    if (ret == 0)
    {
        printf("SIGINT 不阻塞\n");
```

```
    }
    else if (ret == 1)
    {
        printf("SIGINT 阻塞\n");
    }

    // 判断SIGQUIT是否在信号集中
    ret = sigismember(&set, SIGQUIT);
    if (ret == 0)
    {
        printf("SIGQUIT 不阻塞\n");
    }
    else if (ret == 1)
    {
        printf("SIGQUIT 阻塞\n");
    }

    // 从信号集中删除一个信号
    sigdelset(&set, SIGQUIT);

    // 判断SIGQUIT是否在信号集中
    ret = sigismember(&set, SIGQUIT);
    if (ret == 0)
    {
        printf("SIGQUIT 不阻塞\n");
    }
    else if (ret == 1)
    {
        printf("SIGQUIT 阻塞\n");
    }

    return 0;
}
```



函数集合二

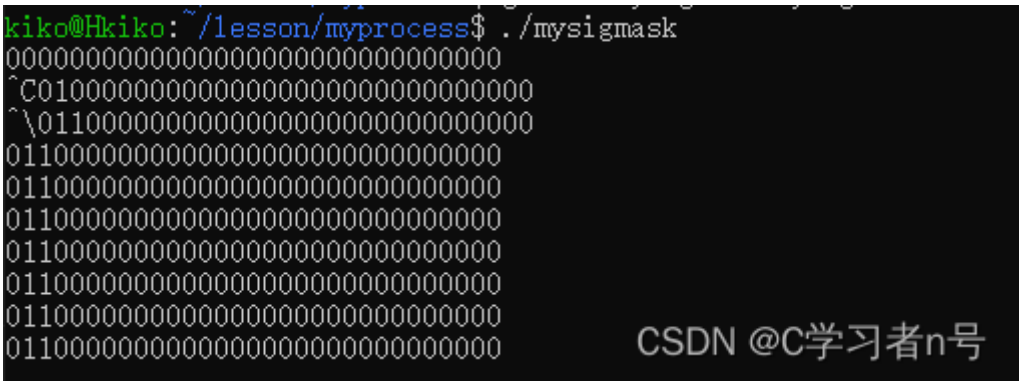
```
/*
    int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
        - 功能：将自定义信号集中的数据设置到内核中（设置阻塞，解除阻塞，替换）
        - 参数：
            - how : 如何对内核阻塞信号集进行处理
                SIG_BLOCK: 将用户设置的阻塞信号集添加到内核中，内核中原来的数据不变
                    假设内核中默认的阻塞信号集是mask， mask | set
                SIG_UNBLOCK: 根据用户设置的数据，对内核中的数据进行解除阻塞
                    mask &= ~set
                SIG_SETMASK:覆盖内核中原来的值

            - set : 已经初始化好的用户自定义的信号集
            - oldset : 保存设置之前的内核中的阻塞信号集的状态，可以是 NULL
        - 返回值：
            成功：0
```

```
            失败：-1，并且设置错误号：EFAULT、EINVAL

    int sigpending(sigset_t *set);
        - 功能：获取内核中的未决信号集
        - 参数：set,传出参数，保存的是内核中的未决信号集中的信息。
*/
```

示例2

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

int main()
{

    // 设置2、3号信号阻塞。二号信号通过ctrl + c触发，三号信号通过ctrl + \ 触发
    sigset_t set;
    sigemptyset(&set);  //清空操作，全部置为0

    // 将2号和3号信号添加到信号集中
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGQUIT);

    // 修改内核中的阻塞信号集
    sigprocmask(SIG_BLOCK, &set, NULL);

    int num = 0;

    while (1)
    {
        num++;
        // 获取当前的未决信号集的数据，32个信号依次全部获取，for循环中打印出每一位信号的状态
        sigset_t pendingset;
        sigemptyset(&pendingset);
        sigpending(&pendingset);

        // 遍历前32位
        for (int i = 1; i <= 31; i++)
        {
            if (sigismember(&pendingset, i) == 1)
            {
                printf("1");
            }
            else if (sigismember(&pendingset, i) == 0)
            {
                printf("0");
            }
            else
            {
                perror("sigismember");
                exit(0);
            }
        }
```

```
        printf("\n"); //下面代码注释掉的话程序一直会输出未决信号，因为2个信号都被阻塞了。必须解除程序才可以正常运行结束。
        sleep(1);
        if (num == 10)
        {
            // 解除阻塞
            sigprocmask(SIG_UNBLOCK, &set, NULL);
        }
    }

    return 0;
}
```



信号捕捉函数之二sigaction

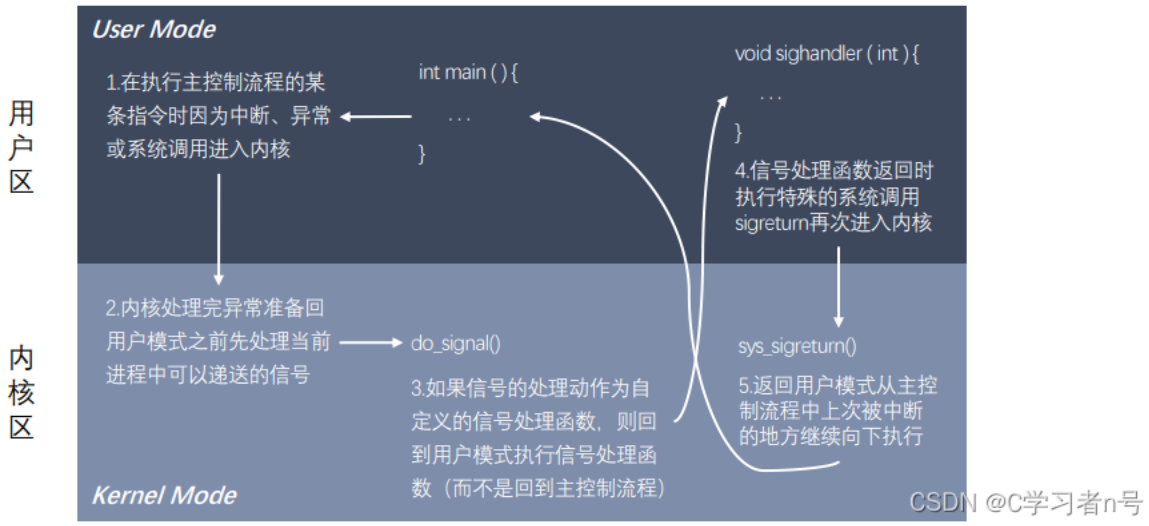
函数说明

```
#include <signal.h>
    int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

        - 功能：检查或者改变信号的处理。信号捕捉
        - 参数：
            - signum : 需要捕捉的信号的编号或者宏值（信号的名称）
            - act : 捕捉到信号之后的处理动作
            - oldact : 上一次对信号捕捉相关的设置，一般不使用，传递NULL
        - 返回值：
            成功 0
            失败 -1

     struct sigaction {
        // 函数指针，指向的函数就是信号捕捉到之后的处理函数
        void     (*sa_handler)(int);
        // 不常用
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        // 临时阻塞信号集，在信号捕捉函数执行过程中，临时阻塞某些信号。
        sigset_t   sa_mask;
        // 使用哪一个信号处理对捕捉到的信号进行处理
        // 这个值可以是0，表示使用sa_handler,也可以是SA_SIGINFO表示使用sa_sigaction
        int        sa_flags;
        // 被废弃掉了
        void     (*sa_restorer)(void);
    };
```

示例

```c
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void myalarm(int num) {
    printf("捕捉到了信号的编号是：%d\n", num);
    printf("xxxxxxx\n");
}

// 过3秒以后，每隔2秒钟定时一次
int main() {

    struct sigaction act;
    act.sa_flags = 0;
    act.sa_handler = myalarm;
    sigemptyset(&act.sa_mask);  // 清空临时阻塞信号集

    // 注册信号捕捉
    sigaction(SIGALRM, &act, NULL);

    struct itimerval new_value;

    // 设置间隔的时间
    new_value.it_interval.tv_sec = 2;
    new_value.it_interval.tv_usec = 0;

    // 设置延迟的时间,3秒之后开始第一次定时
    new_value.it_value.tv_sec = 3;
    new_value.it_value.tv_usec = 0;

    int ret = setitimer(ITIMER_REAL, &new_value, NULL); // 非阻塞的
    printf("定时器开始了...\n");

    if(ret == -1) {
        perror("setitimer");
        exit(0);
    }

    // getchar();
    while(1);

    return 0;
}
```

内核实现信号捕捉的过程



SIGCHLD信号

说明

■ SIGCHLD信号产生的条件（3个）
□ 子进程终止时
□ 子进程接收到 SIGSTOP 信号停止时
□ 子进程处在停止态，接受到SIGCONT后唤醒时
■ 以上三种条件都会给父进程发送 SIGCHLD 信号，父进程默认会忽略该信号
这里，涉及到了之前的僵尸进程。使用SIGCHLD信号解决僵尸进程的问题。

代码详解

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

void myFun(int num) {
    printf("捕捉到的信号 : %d\n", num);
    // 回收子进程PCB的资源。
    // while(1) {
    //     wait(NULL);
    // }
    // 为了能够多次回收结束的子进程，需要使用waitpid函数并且设置为不阻塞。避免一直在循环等待接收，阻止程序的正常运行。
    // 而且，如果有多个子进程发出了SIGCHLD信号，它的速度是很快的，但是未决信号集只会记录一个，其余的会舍弃，所以需要循环回收。
    while(1) {
       int ret = waitpid(-1, NULL, WNOHANG);
       if(ret > 0) {
           printf("child die , pid = %d\n", ret);
       } else if(ret == 0) {
           // 说明还有子进程或者
           break;
       } else if(ret == -1) {
           // 没有子进程
           break;
```

```
        }
    }
}

int main() {

    // 提前设置好阻塞信号集，阻塞SIGCHLD，因为有可能子进程很快结束，父进程还没有注册完信号捕捉
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGCHLD);
    sigprocmask(SIG_BLOCK, &set, NULL);

    // 创建一些子进程
    pid_t pid;
    for(int i = 0; i < 20; i++) {
        pid = fork();
        if(pid == 0) {
            break;
        }
    }

    if(pid > 0) {
        // 父进程

        // 捕捉子进程死亡时发送的SIGCHLD信号
        struct sigaction act;
        act.sa_flags = 0;
        act.sa_handler = myFun;
        sigemptyset(&act.sa_mask);
        sigaction(SIGCHLD, &act, NULL);

        // 注册完信号捕捉以后，解除阻塞
        sigprocmask(SIG_UNBLOCK, &set, NULL);

        while(1) {
            printf("parent process pid : %d\n", getpid());
            sleep(2);
        }
    } else if( pid == 0) {
        // 子进程
        printf("child process pid : %d\n", getpid());
    }

    return 0;
}
```