本节重点：

> 1. 掌握Linux信号的基本概念
> 2. 掌握信号产生的一般方式
> 3. 理解信号递达和阻塞的概念，原理。
> 4. 掌握信号捕捉的一般方式。
> 5. 重新了解可重入函数的概念。
> 6. 了解竞态条件的情景和处理方式
> 7. 了解SIGCHLD信号， 重新编写信号处理函数的一般处理机制

信号入门

1. 生活角度的信号

> - 你在网上买了很多件商品，再等待不同商品快递的到来。但即便快递没有到来，你也知道快递来临时，你该怎么处理快递。也就是你能“识别快递”
> - 当快递员到了你楼下，你也收到快递到来的通知，但是你正在打游戏，需5min之后才能去取快递。那么在在这5min之内，你并没有下去去取快递，但是你是知道有快递到来了。也就是取快递的行为并不是一定要立即执行，可以理解成“在合适的时候去取”。
> - 在收到通知，再到你拿到快递期间，是有一个时间窗口的，在这段时间，你并没有拿到快递，但是你知道有一个快递已经来了。本质上是你“记住了有一个快递要去取”
> - 当你时间合适，顺利拿到快递之后，就要开始处理快递了。而处理快递一般方式有三种：1. 执行默认动作（幸福的打开快递，使用商品）2. 执行自定义动作（快递是零食，你要送给你你的女朋友）3. 忽略快递（快递拿上来之后，扔掉床头，继续开一把游戏）
> - 快递到来的整个过程，对你来讲是异步的，你不能准确断定快递员什么时候给你打电话

2. 技术应用角度的信号

> 1. 用户输入命令,在Shell下启动一个前台进程。
>   . 用户按`Ctrl-C`,这个键盘输入产生一个硬件中断，被OS获取，解释成信号，发送给目标前台进程
>   . 前台进程因为收到信号，进而引起进程退出

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>

int main()
{
    while(1){
        printf("I am a process, I am waiting signal!\n");
        sleep(1);
    }
}

[hb@localhost code_test]$ ./sig
I am a process, I am waiting signal!
I am a process, I am waiting signal!
I am a process, I am waiting signal!
^C
```

```
[hb@localhost code_test]$
```

- 请将生活例子和`Ctrl-C`信号处理过程相结合，解释一下信号处理过程
- 进程就是你，操作系统就是快递员，信号就是快递

3. 注意

1. `Ctrl-C`产生的信号只能发给前台进程。一个命令后面加个&可以放到后台运行,这样Shell不必等待进程结束就可以接受新的命令,启动新的进程。
2. Shell可以同时运行一个前台进程和任意多个后台进程,只有前台进程才能接到像`Ctrl-C`这种控制键产生的信号。
3. 前台进程在运行过程中用户随时可能按下`Ctrl-C`而产生一个信号,也就是说该进程的用户空间代码执行到任何地方都有可能收到`SIGINT`信号而终止,所以信号相对于进程的控制流程来说是异步(Asynchronous)的。

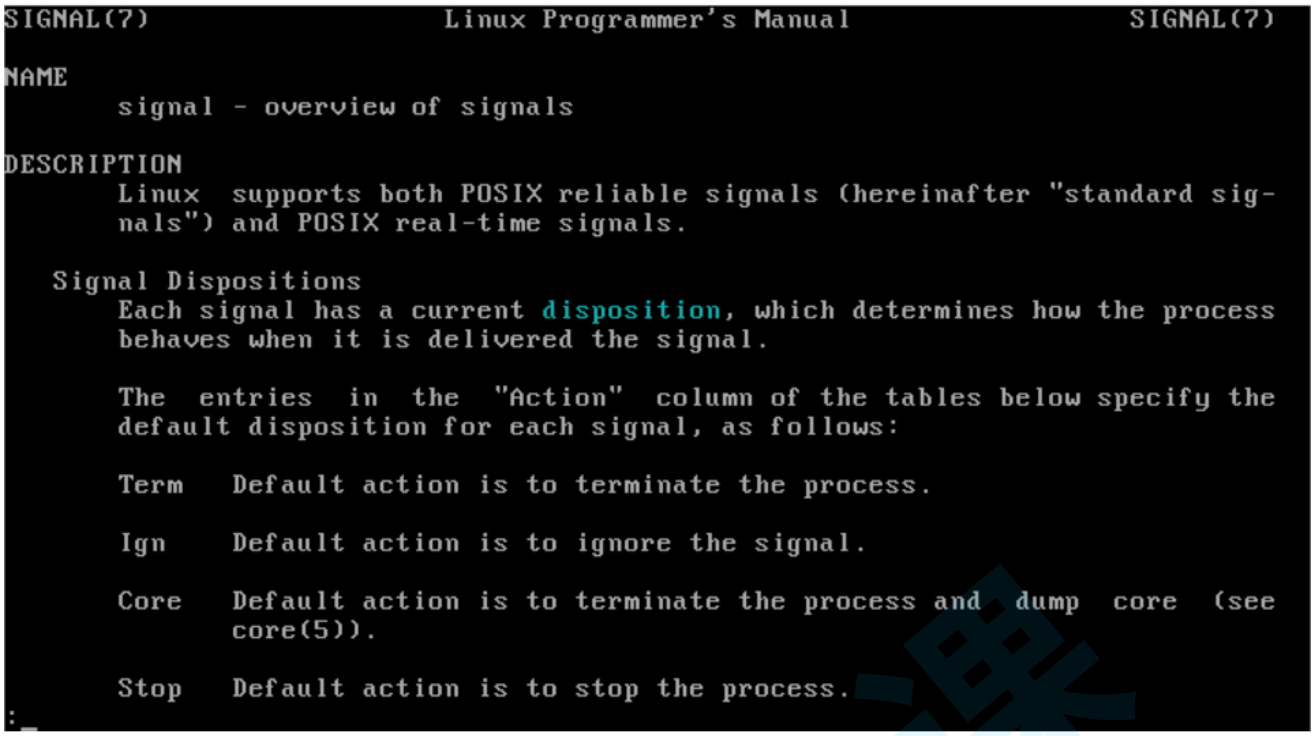
4. 信号概念

- 信号是进程之间事件异步通知的一种方式，属于软中断。

5. 用kill -l命令可以察看系统定义的信号列表

```
[root@localhost ~]# 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
```

- 每个信号都有一个编号和一个宏定义名称,这些宏定义可以在signal.h中找到,例如其中有定 义`#define SIGINT 2`
- 编号34以上的是实时信号,本章只讨论编号34以下的信号,不讨论实时信号。这些信号各自在什么条件下产生,默认的处理动作是什么,在signal(7)中都有详细说明: `man 7 signal`



6. 信号处理常见方式概览

(sigaction函数稍后详细介绍),可选的处理动作有以下三种:

> 1. 忽略此信号。
> 2. 执行该信号的默认处理动作。
> 3. 提供一个信号处理函数,要求内核在处理该信号时切换到用户态执行这个处理函数,这种方式称为捕捉(Catch)一个信号。

产生信号

1. 通过终端按键产生信号

> SIGINT的默认处理动作是终止进程,SIGQUIT的默认处理动作是终止进程并且Core Dump,现在我们来验证一下。

Core Dump

> 首先解释什么是Core Dump。当一个进程要异常终止时,可以选择把进程的用户空间内存数据全部 保存到磁盘上,文件名通常是core,这叫做Core Dump。进程异常终止通常是因为有Bug,比如非法内存访问导致段错误,事后可以用调试器检查core文件以查清错误原因,这叫做Post-mortem Debug（事后调试）。一个进程允许产生多大的core文件取决于进程的Resource Limit(这个信息保存 在PCB中)。默认是不允许产生core文件的,因为core文件中可能包含用户密码等敏感信息,不安全。在开发调试阶段可以用ulimit命令改变这个限制,允许产生core文件。 首先用ulimit命令改变Shell进程的Resource Limit,允许core文件最大为1024K: `$ ulimit -c 1024`

```
[root@localhost test11]# ulimit -c 1024
[root@localhost test11]# ulimit -a
core file size          (blocks, -c) 1024
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7908
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 10240
cpu time               (seconds, -t) unlimited
```

> 然后写一个死循环程序:

```
[root@localhost test11]# cat test.c
#include <stdio.h>
int main()
{
        printf("pid is : %d\n",getpid());
        while(1);
        return 0;
}
```

> 前台运行这个程序,然后在终端键入Ctrl-C（貌似不行）或Ctrl-\（介个可以）：

```
[root@localhost test11]# ./test
pid is : 4506
^\Quit (core dumped)
[root@localhost test11]# ll
total 92
-rw-------. 1 root root 159744 Apr 21 18:04 core.4506
-rw-r--r--. 1 root root     61 Apr 21 18:00 Makefile
-rwxr-xr-x. 1 root root   4766 Apr 21 18:03 test
-rw-r--r--. 1 hb   hb       92 Apr 21 18:03 test.c
-rw-r--r--. 1 root root   627 Apr 21 17:36 test.cpp
[root@localhost test11]# _
```

> ulimit命令改变了Shell进程的Resource Limit,test进程的PCB由Shell进程复制而来,所以也具 有和Shell进程相同的Resource Limit值,这样就可以产生Core Dump了。 使用core文件：

```
total 16
-rw-r--r--. 1 root root   62 Feb 28 17:32 Makefile
-rwxr-xr-x. 1 root root 5870 Feb 28 18:03 test
-rw-r--r--. 1 root root  139 Feb 28 17:30 test.c
[root@bit_tech test78]# ./test
Segmentation fault (core dumped)
[root@bit_tech test78]# ls
core.2884  Makefile  test  test.c
[root@bit_tech test78]# gdb test
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-75.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /BIT/project/test78/test...done.
(gdb) core-file core.2884
[New Thread 2884]
Missing separate debuginfo for
Try: yum --enablerepo='*-debug*' install /usr/lib/debug/.build-id/d5/942f21cf3002919e55180a33793d3c25a060f3
Reading symbols from /lib/libc-2.12.so...Reading symbols from /usr/lib/debug/lib/libc-2.12.so.debug...done.
done.
Loaded symbols for /lib/libc-2.12.so
Reading symbols from /lib/ld-2.12.so...Reading symbols from /usr/lib/debug/lib/ld-2.12.so.debug...done.
done.
Loaded symbols for /lib/ld-2.12.so
Core was generated by `./test'.
Program terminated with signal 11, Segmentation fault.
#0  0x080483ad in fun () at test.c:8
8                       *p = 100;
(gdb)
```

2. 调用系统函数向进程发信号

> 首先在后台执行死循环程序,然后用kill命令给它发SIGSEGV信号。

```
[root@localhost test11]# ./test &
[3] 4568
[root@localhost test11]# kill -SIGSEGV 4568
[root@localhost test11]#
[3]   Segmentation fault      (core dumped) ./test
[root@localhost test11]# cat test.c
#include <stdio.h>
int main()
{
        while(1);
        return 0;
}
```

多按一次回车

- 4568是test进程的id。之所以要再次回车才显示`Segmentation fault`,是因为在4568进程终止掉 之前已经回到了Shell提示符等待用户输入下一条命令,Shell不希望Segmentation fault信息和用 户的输入交错在一起,所以等用户输入命令之后才显示。
- 指定发送某种信号的kill命令可以有多种写法,上面的命令还可以写成`kill -SIGSEGV 4568`或`kill -11 4568`, 11是信号SIGSEGV的编号。以往遇 到的段错误都是由非法内存访问产生的,而这个程序本身没错,给它发SIGSEGV也能产生段错误。

kill命令是调用kill函数实现的。kill函数可以给一个指定的进程发送指定的信号。raise函数可以给当前进程发送指定的信号(自己给自己发信号)。

```
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
这两个函数都是成功返回0,错误返回-1。
```

abort函数使当前进程接收到信号而异常终止。

```
#include <stdlib.h>
void abort(void);
就像exit函数一样,abort函数总是会成功的,所以没有返回值。
```

3. 由软件条件产生信号

SIGPIPE是一种由软件条件产生的信号,在“管道”中已经介绍过了。本节主要介绍alarm函数 和SIGALRM信号。

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
调用alarm函数可以设定一个闹钟,也就是告诉内核在seconds秒之后给当前进程发SIGALRM信号, 该信号的默认处理动作是终止当前进程。
```

这个函数的返回值是0或者是以前设定的闹钟时间还余下的秒数。打个比方,某人要小睡一觉,设定闹钟为30分钟之后响,20分钟后被人吵醒了,还想多睡一会儿,于是重新设定闹钟为15分钟之后响,“以前设定的闹钟时间还余下的时间”就是10分钟。如果seconds值为0,表示取消以前设定的闹钟,函数的返回值仍然是以前设定的闹钟时间还余下的秒数（自己验证一下？）

例 alarm

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

int main()
{
    int count = 14;
    alarm(1);
    for(;1;count++){
        printf("count = %d\n",count);
    }

    return 0;
}
```

这个程序的作用是1秒钟之内不停地数数,1秒钟到了就被SIGALRM信号终止。

4. 硬件异常产生信号

硬件异常被硬件以某种方式被硬件检测到并通知内核,然后内核向当前进程发送适当的信号。例如当前进程执行了除以0的指令,CPU的运算单元会产生异常,内核将这个异常解释 为SIGFPE信号发送给进程。再比如当前进程访问了非法内存地址,,MMU会产生异常,内核将这个异常解释为SIGSEGV信号发送给进程。

信号捕捉初识

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

void handler(int sig)
{
    printf("catch a sig : %d\n", sig);
}

int main()
{
    signal(2, handler); //前文提到过，信号是可以被自定义捕捉的，siganl函数就是来进行信号捕捉的，提前了解一下
    while(1);
    return 0;
}

[hb@localhost code_test]$ ./sig
^Ccatch a sig : 2
^Ccatch a sig : 2
^Ccatch a sig : 2
^Ccatch a sig : 2
^\Quit (core dumped)
[hb@localhost code_test]$
```

模拟一下野指针异常

```
//默认行为
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

void handler(int sig)
{
    printf("catch a sig : %d\n", sig);
}

int main()
{
    //signal(SIGSEGV, handler);
    sleep(1);
    int *p = NULL;
    *p = 100;

    while(1);
    return 0;
}
[hb@localhost code_test]$ ./sig
Segmentation fault (core dumped)
[hb@localhost code_test]$


//捕捉行为
```

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

void handler(int sig)
{
    printf("catch a sig : %d\n", sig);
}

int main()
{
    //signal(SIGSEGV, handler);
    sleep(1);
    int *p = NULL;
    *p = 100;

    while(1);
    return 0;
}
[hb@localhost code_test]$ ./sig
[hb@localhost code_test]$ ./sig
catch a sig : 11
catch a sig : 11
catch a sig : 11
```

由此可以确认，我们在C/C++当中除零，内存越界等异常，在系统层面上，是被当成信号处理的。

总结思考一下

> - 上面所说的所有信号产生，最终都要有OS来进行执行，为什么？OS是进程的管理者
> - 信号的处理是否是立即处理的？在合适的时候
> - 信号如果不是被立即处理，那么信号是否需要暂时被进程记录下来？记录在哪里最合适呢？
> - 一个进程在没有收到信号的时候，能否能知道，自己应该对合法信号作何处理呢？
> - 如何理解OS向进程发送信号？能否描述一下完整的发送处理过程？

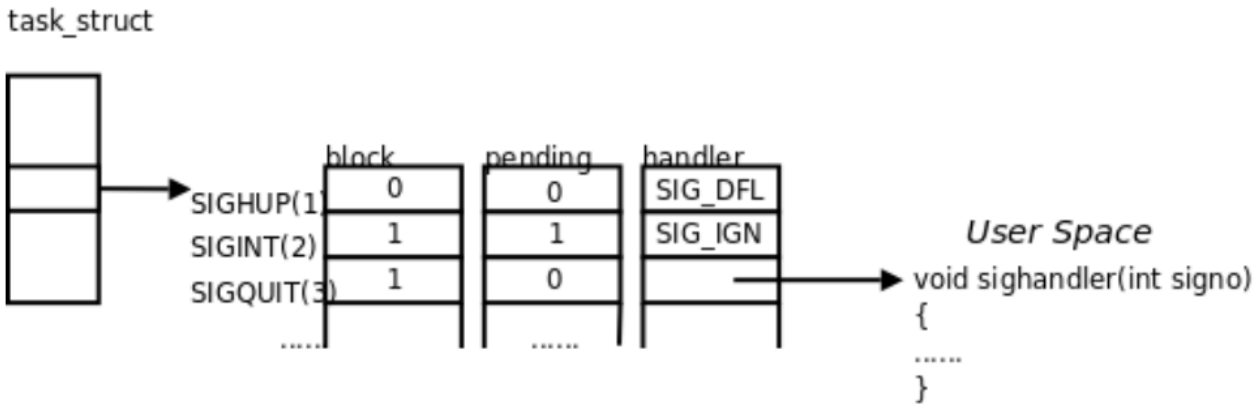
阻塞信号

1. 信号其他相关常见概念

> - 实际执行信号的处理动作称为信号递达(Delivery)
> - 信号从产生到递达之间的状态,称为信号未决(Pending)。
> - 进程可以选择阻塞 (Block)某个信号。
> - 被阻塞的信号产生时将保持在未决状态,直到进程解除对此信号的阻塞,才执行递达的动作.
> - 注意,阻塞和忽略是不同的,只要信号被阻塞就不会递达,而忽略是在递达之后可选的一种处理动作。

2. 在内核中的表示

信号在内核中的表示示意图



- 每个信号都有两个标志位分别表示阻塞(block)和未决(pending),还有一个函数指针表示处理动作。信号产生时,内核在进程控制块中设置该信号的未决标志,直到信号递达才清除该标志。在上图的例子中,SIGHUP信号未阻塞也未产生过,当它递达时执行默认处理动作。
- SIGINT信号产生过,但正在被阻塞,所以暂时不能递达。虽然它的处理动作是忽略,但在没有解除阻塞之前不能忽略这个信号,因为进程仍有机会改变处理动作之后再解除阻塞。
- SIGQUIT信号未产生过,一旦产生SIGQUIT信号将被阻塞,它的处理动作是用户自定义函数sighandler。如果在进程解除对某信号的阻塞之前这种信号产生过多次,将如何处理?POSIX.1允许系统递送该信号一次或多次。Linux是这样实现的:常规信号在递达之前产生多次只计一次,而实时信号在递达之前产生多次可以依次放在一个队列里。本章不讨论实时信号。

3. sigset_t

从上图来看,每个信号只有一个bit的未决标志,非0即1,不记录该信号产生了多少次,阻塞标志也是这样表示的。因此,未决和阻塞标志可以用相同的数据类型sigset_t来存储,sigset_t称为信号集,这个类型可以表示每个信号的“有效”或“无效”状态,在阻塞信号集中“有效”和“无效”的含义是该信号是否被阻塞,而在未决信号集中“有效”和“无效”的含义是该信号是否处于未决状态。下一节将详细介绍信号集的各种操作。 阻塞信号集也叫做当前进程的信号屏蔽字(Signal Mask),这里的“屏蔽”应该理解为阻塞而不是忽略。

4. 信号集操作函数

sigset_t类型对于每种信号用一个bit表示“有效”或“无效”状态,至于这个类型内部如何存储这些bit则依赖于系统实现,从使用者的角度是不必关心的,使用者只能调用以下函数来操作sigset_ t变量,而不应该对它的内部数据做任何解释,比如用printf直接打印sigset_t变量是没有意义的

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember (const sigset_t *set, int signo);
```

- 函数sigemptyset初始化set所指向的信号集,使其中所有信号的对应bit清零,表示该信号集不包含 任何有效信号。
- 函数sigfillset初始化set所指向的信号集,使其中所有信号的对应bit置位,表示 该信号集的有效信号包括系统支持的所有信号。
- 注意,在使用sigset_ t类型的变量之前,一定要调 用sigemptyset或sigfillset做初始化,使信号集处于确定的状态。初始化sigset_t变量之后就可以在调用sigaddset和sigdelset在该信号集中添加或删除某种有效信

> 号。

这四个函数都是成功返回0,出错返回-1。sigismember是一个布尔函数,用于判断一个信号集的有效信号中是否包含某种 信号,若包含则返回1,不包含则返回0,出错返回-1。

sigprocmask

调用函数sigprocmask可以读取或更改进程的信号屏蔽字(阻塞信号集)。

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
返回值:若成功则为0,若出错则为-1
```

> 如果oset是非空指针,则读取进程的当前信号屏蔽字通过oset参数传出。如果set是非空指针,则 更改进程的信号屏蔽字,参数how指示如何更改。如果oset和set都是非空指针,则先将原来的信号 屏蔽字备份到oset里,然后根据set和how参数更改信号屏蔽字。假设当前的信号屏蔽字为mask,下表说明了how参数的可选值。

SIG_BLOCK	set包含了我们希望添加到当前信号屏蔽字的信号，相当于mask=mask|set
SIG_UNBLOCK	set包含了我们希望从当前信号屏蔽字中解除阻塞的信号，相当于mask=mask&~set
SIG_SETMASK	设置当前信号屏蔽字为set所指向的值，相当于mask=set

> 如果调用sigprocmask解除了对当前若干个未决信号的阻塞,则在sigprocmask返回前,至少将其中一个信号递达。

sigpending

```
#include <signal.h>
sigpending
读取当前进程的未决信号集,通过set参数传出。调用成功则返回0,出错则返回-1。 下面用刚学的几个函数做个实验。程序如下:
```

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

void printsigset(sigset_t *set)
{
    int i=0;
    for(;i<32;i++){
        if ( sigismember(set, i) ){
            putchar('1');
        }else{
            putchar('0');
        }
    }
    puts("");
}
int main()
{
    sigset_t s, p;
    sigemptyset(&s);
    sigaddset(&s, SIGINT);
    sigprocmask(SIG_BLOCK, &s, NULL);
    while(1){
        sigpending(&p);
        printsigset(&p);
        sleep(1);
    }
    return 0;
}
```

判断指定信号是否在目标集合中

定义信号集对象，并清空初始化

ctrl + c

设置阻塞信号集，阻塞SIGINT信号

获取未决信号集

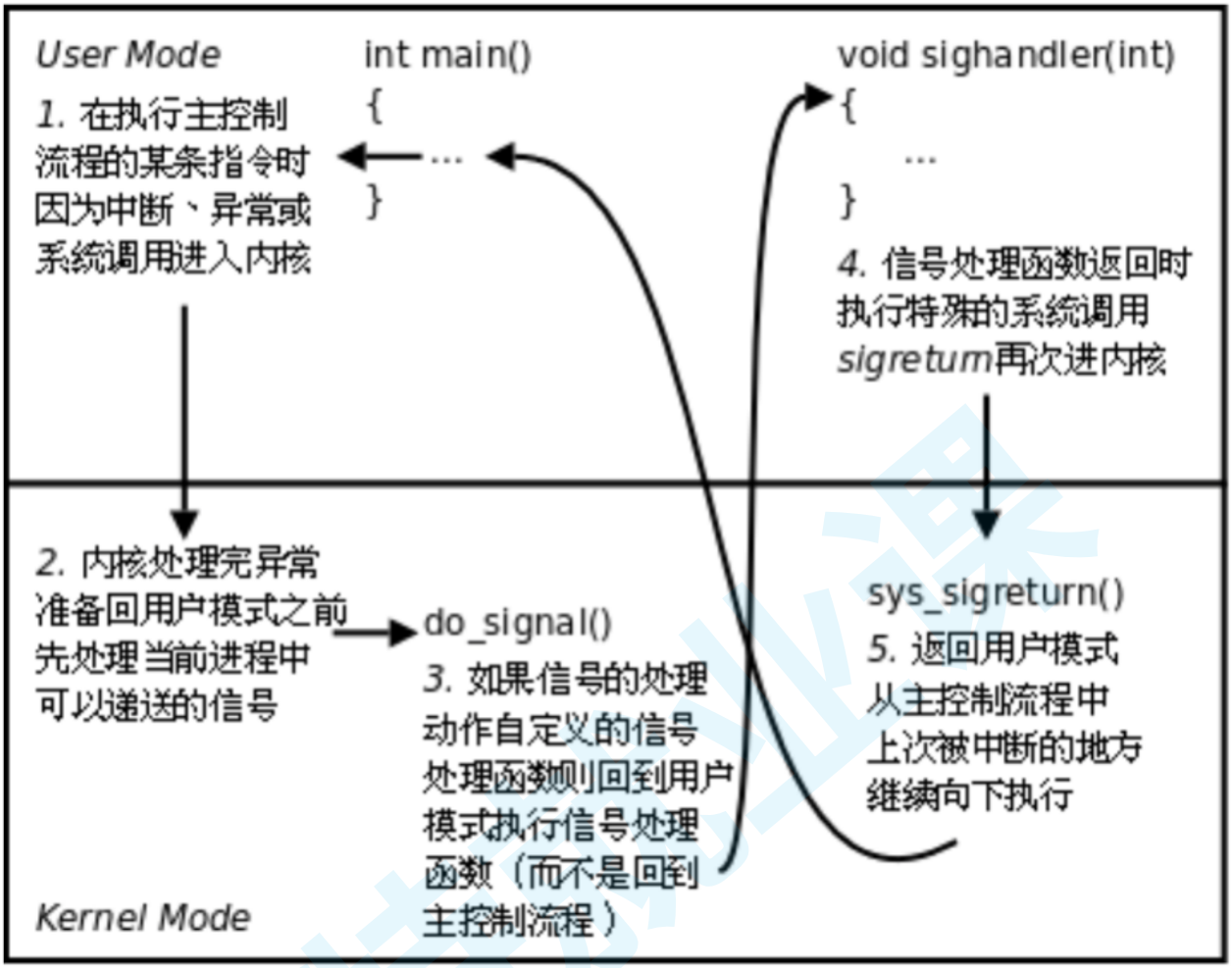
```
[root@localhost sig]# ./test
10000000000000000000000000000000
10000000000000000000000000000000
10000000000000000000000000000000
10000000000000000000000000000000
^C10100000000000000000000000000000
10100000000000000000000000000000
10100000000000000000000000000000
10100000000000000000000000000000
^\Quit (core dumped)
```

键入ctrl + c（SIGINT），
该信号被test阻塞，所以一直
处于未决状态，不被处理

程序运行时,每秒钟把各信号的未决状态打印一遍,由于我们阻塞了SIGINT信号,按Ctrl-C将会 使SIGINT信号处于未决状态,按Ctrl-\仍然可以终止程序,因为SIGQUIT信号没有阻塞。

捕捉信号

图 **33.2.** 信号的捕捉



1. 内核如何实现信号的捕捉

如果信号的处理动作是用户自定义函数,在信号递达时就调用这个函数,这称为捕捉信号。由于信号处理函数的代码是在用户空间的,处理过程比较复杂,举例如下: 用户程序注册了SIGQUIT信号的处理函数sighandler。 当前正在执行main函数,这时发生中断或异常切换到内核态。 在中断处理完毕后要返回用户态的main函数之前检查到有信号SIGQUIT递达。 内核决定返回用户态后不是恢复main函数的上下文继续执行,而是执行sighandler函 数,sighandler和main函数使用不同的堆栈空间,它们之间不存在调用和被调用的关系,是 两个独立的控制流程。 sighandler函数返回后自动执行特殊的系统调用sigreturn再次进入内核态。 如果没有新的信号要递达,这次再返回用户态就是恢复main函数的上下文继续执行了。

2. sigaction

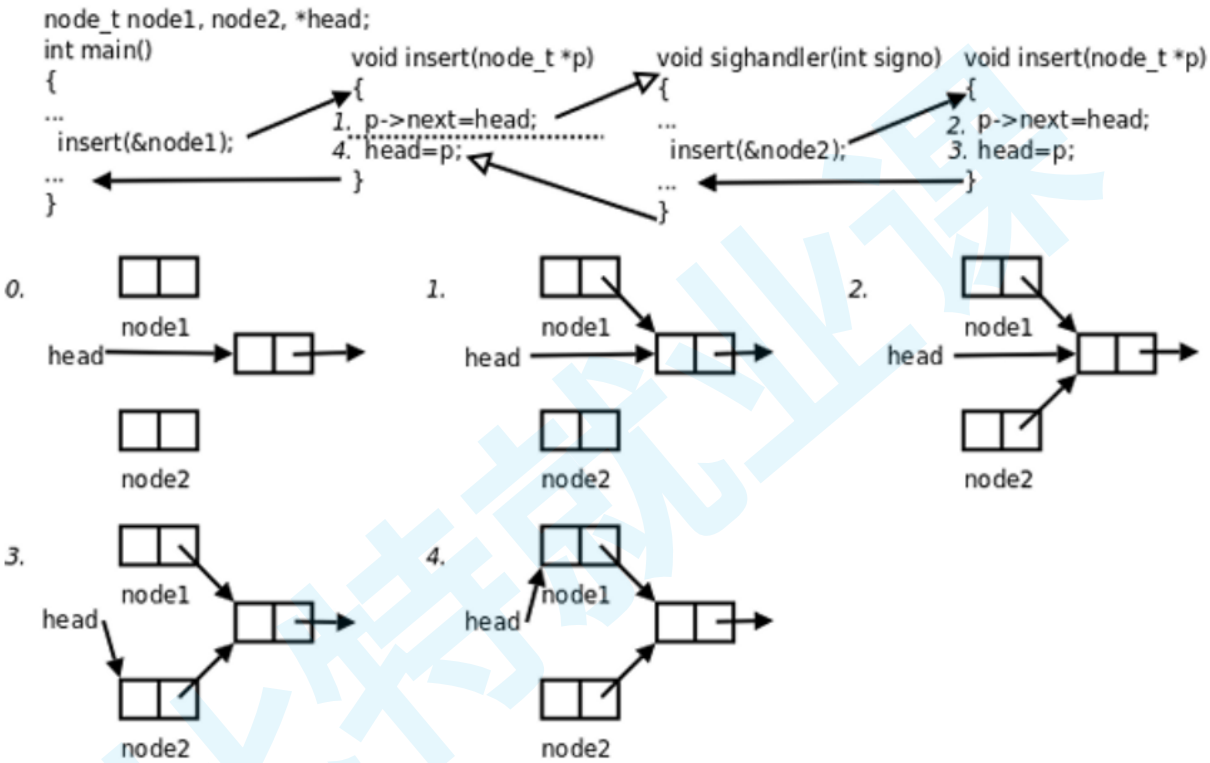
```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
```

- sigaction函数可以读取和修改与指定信号相关联的处理动作。调用成功则返回0,出错则返回- 1。signo是指定信号的编号。若act指针非空,则根据act修改该信号的处理动作。若oact指针非 空,则通过oact传出该信号原来的处理动作。act和oact指向sigaction结构体:

- 将sa_handler赋值为常数SIG_IGN传给sigaction表示忽略信号,赋值为常数SIG_DFL表示执行系统默认动作,赋值为一个函数指针表示用自定义函数捕捉信号,或者说向内核注册了一个信号处理函 数,该函数返回值为void,可以带一个int参数,通过参数可以得知当前信号的编号,这样就可以用同一个函数处理多种信号。显然,这也是一个回调函数,不是被main函数调用,而是被系统所调用。

当某个信号的处理函数被调用时,内核自动将当前信号加入进程的信号屏蔽字,当信号处理函数返回时自动恢复原来的信号屏蔽字,这样就保证了在处理某个信号时,如果这种信号再次产生,那么 它会被阻塞到当前处理结束为止。 如果在调用信号处理函数时,除了当前信号被自动屏蔽之外,还希望自动屏蔽另外一些信号,则用sa_mask字段说明这些需要额外屏蔽的信号,当信号处理函数返回时自动恢复原来的信号屏蔽字。 sa_flags字段包含一些选项,本章的代码都把sa_flags设为0,sa_sigaction是实时信号的处理函数,本章不详细解释这两个字段,有兴趣的同学可以在了解一下。

可重入函数



- main函数调用insert函数向一个链表head中插入节点node1,插入操作分为两步,刚做完第一步的 时候,因为硬件中断使进程切换到内核,再次回用户态之前检查到有信号待处理,于是切换 到sighandler函数,sighandler也调用insert函数向同一个链表head中插入节点node2,插入操作的 两步都做完之后从sighandler返回内核态,再次回到用户态就从main函数调用的insert函数中继续 往下执行,先前做第一步之后被打断,现在继续做完第二步。结果是,main函数和sighandler先后 向链表中插入两个节点,而最后只有一个节点真正插入链表中了。

- 像上例这样,insert函数被不同的控制流程调用,有可能在第一次调用还没返回时就再次进入该函数,这称为重入,insert函数访问一个全局链表,有可能因为重入而造成错乱,像这样的函数称为 不可重入函数,反之,如果一个函数只访问自己的局部变量或参数,则称为可重入(Reentrant) 函数。想一下,为什么两个不同的控制流程调用同一个函数,访问它的同一个局部变量或参数就不会造成错乱?

如果一个函数符合以下条件之一则是不可重入的:

- 调用了malloc或free,因为malloc也是用全局链表来管理堆的。
- 调用了标准I/O库函数。标准I/O库的很多实现都以不可重入的方式使用全局数据结构。

volatile

> - 该关键字在C当中我们已经有所涉猎，今天我们站在信号的角度重新理解一下

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

int flag = 0;

void handler(int sig)
{
    printf("chage flag 0 to 1\n");
    flag = 1;
}

int main()
{
    signal(2, handler);
    while(!flag);
    printf("process quit normal\n");
    return 0;
}

[hb@localhost code_test]$ cat Makefile
sig:sig.c
    gcc -o sig sig.c #-O2
.PHONY:clean
clean:
    rm -f sig

[hb@localhost code_test]$ ./sig
^Cchage flag 0 to 1
process quit normal
```

> 标准情况下，键入`CTRL-C`,2号信号被捕捉，执行自定义动作，修改`flag=1`，`while`条件不满足,退出循环，进程退出

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

int flag = 0;

void handler(int sig)
{
    printf("chage flag 0 to 1\n");
    flag = 1;
}


int main()
```

```
{
    signal(2, handler);
    while(!flag);
    printf("process quit normal\n");
    return 0;
}

[hb@localhost code_test]$ cat Makefile
sig:sig.c
    gcc -o sig sig.c -O2
.PHONY:clean
clean:
    rm -f sig

[hb@localhost code_test]$ ./sig
^Cchage flag 0 to 1
^Cchage flag 0 to 1
^Cchage flag 0 to 1
```

> 优化情况下，键入 `CTRL-C` ,2号信号被捕捉，执行自定义动作，修改 `flag=1` ，但是 `while` 条件依旧满足,进程继续运行！但是很明显flag肯定已经被修改了，但是为何循环依旧执行？很明显， `while` 循环检查的flag，并不是内存中最新的flag，这就存在了数据二异性的问题。 `while` 检测的flag其实已经因为优化，被放在了CPU寄存器当中。如何解决呢？很明显需要 `volatile`

```
[hb@localhost code_test]$ cat sig.c
#include <stdio.h>
#include <signal.h>

volatile int flag = 0;

void handler(int sig)
{
    printf("chage flag 0 to 1\n");
    flag = 1;
}

int main()
{
    signal(2, handler);
    while(!flag);
    printf("process quit normal\n");
    return 0;
}

[hb@localhost code_test]$ cat Makefile
sig:sig.c
    gcc -o sig sig.c -O2
.PHONY:clean
clean:
    rm -f sig


[hb@localhost code_test]$ ./sig
```

```
^Cchage flag 0 to 1
process quit normal
```

> - `volatile`作用：保持内存的可见性，告知编译器，被该关键字修饰的变量，不允许被优化，对该变量的任何操作，都必须在真实的内存中进行操作

SIGCHLD信号 - 选学了解

进程一章讲过用wait和waitpid函数清理僵尸进程,父进程可以阻塞等待子进程结束,也可以非阻 塞地查询是否有子进程结束等待清理(也就是轮询的方式)。采用第一种方式,父进程阻塞了就不 能处理自己的工作了;采用第二种方式,父进程在处理自己的工作的同时还要记得时不时地轮询一 下,程序实现复杂。

其实,子进程在终止时会给父进程发SIGCHLD信号,该信号的默认处理动作是忽略,父进程可以自 定义SIGCHLD信号的处理函数,这样父进程只需专心处理自己的工作,不必关心子进程了,子进程 终止时会通知父进程,父进程在信号处理函数中调用wait清理子进程即可。

请编写一个程序完成以下功能:父进程fork出子进程,子进程调用exit(2)终止,父进程自定 义SIGCHLD信号的处理函数,在其中调用wait获得子进程的退出状态并打印。

事实上,由于UNIX 的历史原因,要想不产生僵尸进程还有另外一种办法:父进程调 用sigaction将SIGCHLD的处理动作置为SIG_IGN,这样fork出来的子进程在终止时会自动清理掉,不 会产生僵尸进程,也不会通知父进程。系统默认的忽略动作和用户用sigaction函数自定义的忽略 通常是没有区别的,但这是一个特例。此方法对于Linux可用,但不保证在其它UNIX系统上都可 用。请编写程序验证这样做不会产生僵尸进程。

测试代码

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

void handler(int sig)
{
    pid_t id;
    while( (id = waitpid(-1, NULL, WNOHANG)) > 0){
        printf("wait child success: %d\n", id);
    }
    printf("child is quit! %d\n", getpid());
}

int main()
{
    signal(SIGCHLD, handler);
    pid_t cid;
    if((cid = fork()) == 0){//child
        printf("child : %d\n", getpid());
        sleep(3);
        exit(1);
    }

    while(1){
        printf("father proc is doing some thing!\n");

        sleep(1);
```

```
    }
    return 0;
}
```