# 0108信号

**用户 -> 内核，为什么？**

**信号捕捉的流程**

不管进程愿不愿意

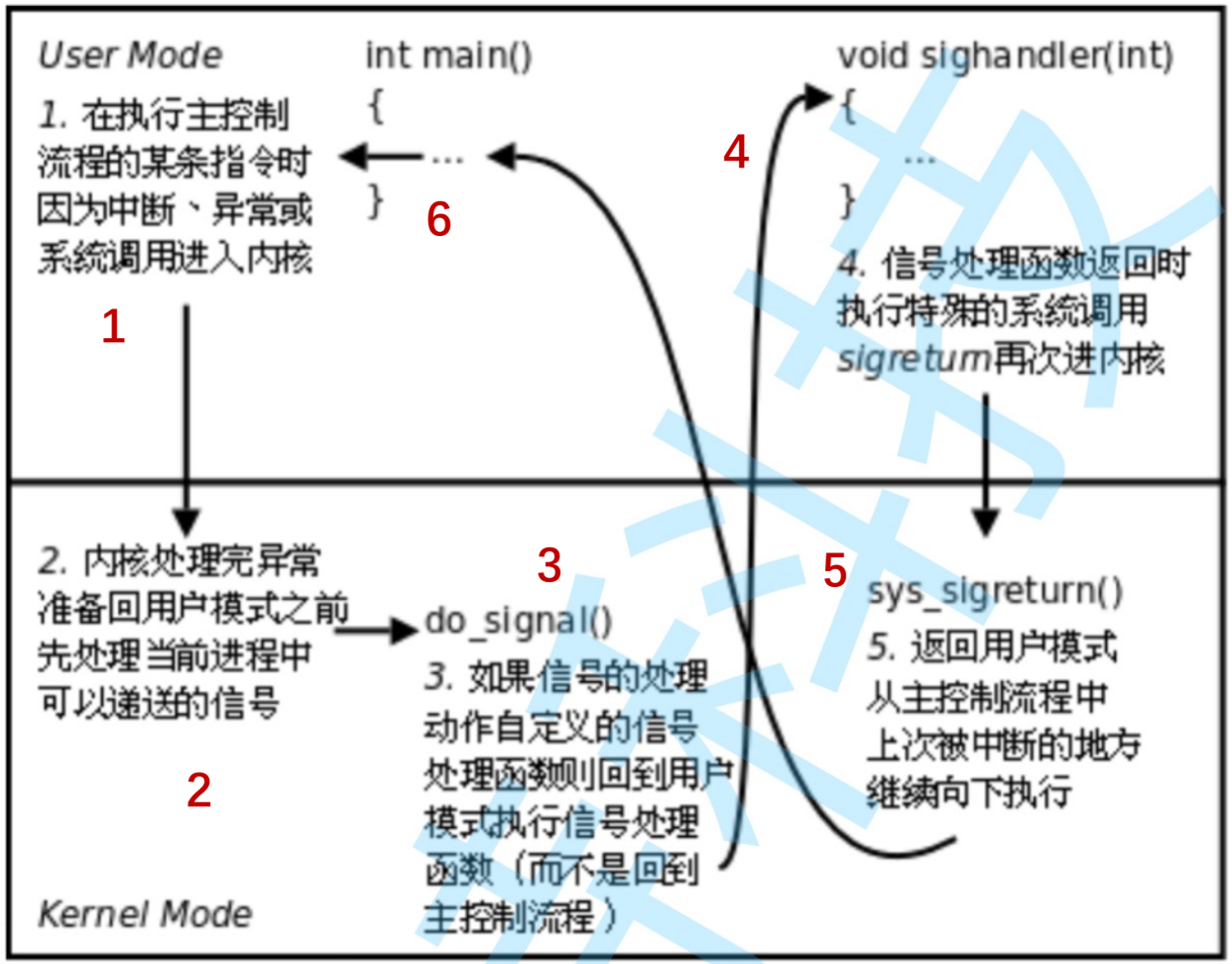
在进程处理信号的某个时间点是肯定要陷入内核的

从内核态返回用户态的时候，可以顺便处理一下信号

默认和忽略这两个动作是很好理解的
因为本来就是内核的动作，所以进程从内核态返回用户态的时候，直接做这个动作即可

最难理解的是信号捕捉的动作

图 **33.2.** 信号的捕捉




现在操作系统，发现我们的信号处理方法是自定义的。

现在有两个问题：

1. 我当前是什么状态？**内核态**

2. 我当前的状态，能不能执行user handler方法
   **能！但是没必要**

**OS能做到帮用户执行handler方法，但它不愿意**
**因为操作系统不相信任何人！**
**如果handler里面有一些非法操作呢？所以不能用内核身份去执行这些代码！**

# 信号的操作

1. signal
2. signaction

TERMINAL

```
SIGACTION(2)                          Linux Programmer's Manual

NAME
       sigaction - examine and change a signal action

SYNOPSIS
       #include <signal.h> 信号编号                结构体类型，输入型参数

       int sigaction(int signum, const struct sigaction *act,
                     struct sigaction *oldact);                    输出型参数

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       sigaction(): _POSIX_C_SOURCE >= 1 || _XOPEN_SOURCE || _POSIX_SOURCE

       siginfo_t: _POSIX_C_SOURCE >= 199309L

DESCRIPTION
```

我们可以预测到
sigaction这个结构体，里面肯定包含处理方法的回调

结构体里面展示的这五个字段
我们只考虑第一个和第三个

```
signum specifies the signal and can be any valid signal except SIGKILL and SIGSTOP.

If act is non-NULL, the new action for signal signum is installed from act.   If  o
NULL, the previous action is saved in oldact.

The sigaction structure is defined as something like:

    struct sigaction {
        void     (*sa_handler)(int);
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int        sa_flags;
        void     (*sa_restorer)(void);
    };

On some architectures a union is involved: do not assign to both sa_handler and sa_

The  sa_restorer element is obsolete and should not be used.  POSIX does not specif
element.
```

```cpp
void handler(int signum)
{
    cout << "获取了一个信号: " << signum << endl;
}
int main()
{
    // 内核数据类型，但是我们下面这些定义是在user栈上定义的
    struct sigaction act, oact;
    act.sa_flags = 0;
    sigemptyset(&act.sa_mask);
    act.sa_handler = handler;
    // 把上面定义的东西设置到进程的pcb当中去
    sigaction(2, &act, &oact);
    cout << "default action: " << (int)oact.sa_handler << endl;
    while (true)
        sleep(1);
    return 0;
}
```

那么左边这些结果是很简单的

结果也是意料之中

**现在提出一个问题：**
**如果处理信号的时候，执行自定义动作，如果处理信号期间，又来了同样的信号，OS如何处理？**

```
17      sigemptyset(&act.sa_mask);
18      act.sa_handler = handler;
19      // 把上面定义的东西设置到进程的pcb当中去
20      sigaction(2, &act, &oact);
21      cout << "default action: " << (int)oact.sa_handler << endl;
```

DEBUG CONSOLE TERMINAL PORTS

TERMINAL ./mysigna

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0108]$ ./mysignal
default action: 0
^C获取了一个信号: 2
^C获取了一个信号: 2
^C获取了一个信号: 2
^C获取了一个信号: 2
```

如果我现在正在自定义处理一个信号，
还没处理完

又有一个信号来了，OS怎么办

**本质：为什么要有block**

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。

通过上面的文字，现在我们来证明以下两个问题：

1. 如果刚才那个代码，我在处理信号的时候发送2号信号，OS是不会处理的
2. pass

```
void handler(int signum)
{
    cout << "获取了一个信号: " << signum << endl;
    sleep(10);//这里sleep 10s 如果在此期间发送2号信号，是没用反应的，因为被block了
}
int main()
{
    // 内核数据类型，但是我们下面这些定义是在user栈上定义的
    struct sigaction act, oact;
    act.sa_flags = 0;
    sigemptyset(&act.sa_mask);
    act.sa_handler = handler;
    // 把上面定义的东西设置到进程的pcb当中去
    sigaction(2, &act, &oact);
    cout << "default action: " << (int)oact.sa_handler << endl;
    while (true)
        sleep(1);
    return 0;
}
```

```
DEBUG CONSOLE   TERMINAL   PORTS
TERMINAL
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0108]$ ./mysignal
default action: 0
^C获取了一个信号：2
^C^C^C^C^C^C^C^C^C^C^C^C^C^C^C^C获取了一个信号：2
```

此时是没反应的

```cpp
void showPending(sigset_t *pending)
{
    for (int sig = 1; sig <= 31; sig++)
    {
        if (sigismember(pending, sig))
            cout << "1";
        else
            cout << "0";
    }
    cout << '\n';
}
void handler(int signum)
{
    cout << "获取了一个信号: " << signum << endl;
    int cnt = 10;
    sigset_t pending;
    while (cnt--)
    {
        sigpending(&pending);
        showPending(&pending);
    }
}
```

我们也可以换一种方式表示

看看位图里面是不是真的如我们所说那样

```
(base) [yufc@VM-12-12-centos:~/Files/BitC
default action: 0
^C获取了一个信号: 2
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
^C0100000000000000000000000000000
01000000000000000000000000000000
01000000000000000000000000000000
^C0100000000000000000000000000000
01000000000000000000000000000000
^C0100000000000000000000000000000
获取了一个信号: 2
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
^C0100000000000000000000000000000
01000000000000000000000000000000
01000000000000000000000000000000
01000000000000000000000000000000
01000000000000000000000000000000
获取了一个信号: 2
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

十秒过后
又可以处理2号信号了

**现在我想除了2号信号**

**3号4号这些都屏蔽，都不要被递达了，就要用到sa_mask了**

```c
int main()
{
    // 内核数据类型，但是我们下面这些定义是在user栈上定义的
    struct sigaction act, oact;
    act.sa_flags = 0;
    sigemptyset(&act.sa_mask);
    act.sa_handler = handler;

    //
    sigaddset(&act.sa_mask, 3);
    sigaddset(&act.sa_mask, 4);
    sigaddset(&act.sa_mask, 5);
    sigaddset(&act.sa_mask, 6);
    sigaddset(&act.sa_mask, 7);
```

mysignal.cc | sendSig.sh | killTheProc.sh

Files ▸ BitCodeField ▸ 0108 ▸ killTheProc.sh

```
1   kill -9 $(pidof mysignal)
```

Files ▸ BitCodeField ▸ 0108 ▸ sendSig.sh

```
1   kill -3 $(pidof mysignal)
2   sleep 1
3   kill -4 $(pidof mysignal)
4   sleep 1
5   kill -5 $(pidof mysignal)
6   sleep 1
7   kill -6 $(pidof mysignal)
8   sleep 1
9   kill -7 $(pidof mysignal)
10  sleep 1
```

```
default action: 0
^C获取了一个信号：2
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
^C01000000000000000000000000000000
01000000000000000000000000000000
01000000000000000000000000000000
01000000000000000000000000000000
01000000000000000000000000000000
01000000000000000000000000000000
01000000000000000000000000000000
获取了一个信号：2
01000000000000000000000000000000
01100000000000000000000000000000
01110000000000000000000000000000
01111000000000000000000000000000
01111100000000000000000000000000
01111110000000000000000000000000
01111111000000000000000000000000
01111111100000000000000000000000
01111111110000000000000000000000
01111111111000000000000000000000
Illegal instruction
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField
0108]$
```

```
(base) [yufc@VM-12-12-centos:~/Files]$ cd ~/File
/BitCodeField/0108
(base) [yufc@VM-12-12-centos:~/Files/BitCodeFiel
/0108]$ bash sendSig.sh
kill: usage: kill [-s sigspec | -n signum | -sig
spec] pid | jobspec ... or kill -l [sigspec]
kill: usage: kill [-s sigspec | -n signum | -sig
spec] pid | jobspec ... or kill -l [sigspec]
kill: usage: kill [-s sigspec | -n signum | -sig
spec] pid | jobspec ... or kill -l [sigspec]
kill: usage: kill [-s sigspec | -n signum | -sig
spec] pid | jobspec ... or kill -l [sigspec]
(base) [yufc@VM-12-12-centos:~/Files/BitCodeFiel
/0108]$ bash sendSig.sh
kill: usage: kill [-s sigspec | -n signum | -sig
spec] pid | jobspec ... or kill -l [sigspec]
kill: usage: kill [-s sigspec | -n signum | -sig
spec] pid | jobspec ... or kill -l [sigspec]
kill: usage: kill [-s sigspec | -n signum | -sig
spec] pid | jobspec ... or kill -l [sigspec]
kill: usage: kill [-s sigspec | -n signum | -sig
spec] pid | jobspec ... or kill -l [sigspec]
(base) [yufc@VM-12-12-centos:~/Files/BitCodeFiel
/0108]$ bash sendSig.sh
(base) [yufc@VM-12-12-centos:~/Files/BitCodeFiel
/0108]$
```

# 补充

**信号捕捉，并没有创建新的进程或线程**

该函数重入了：
一个函数在同一时间被多个执行流同时进入

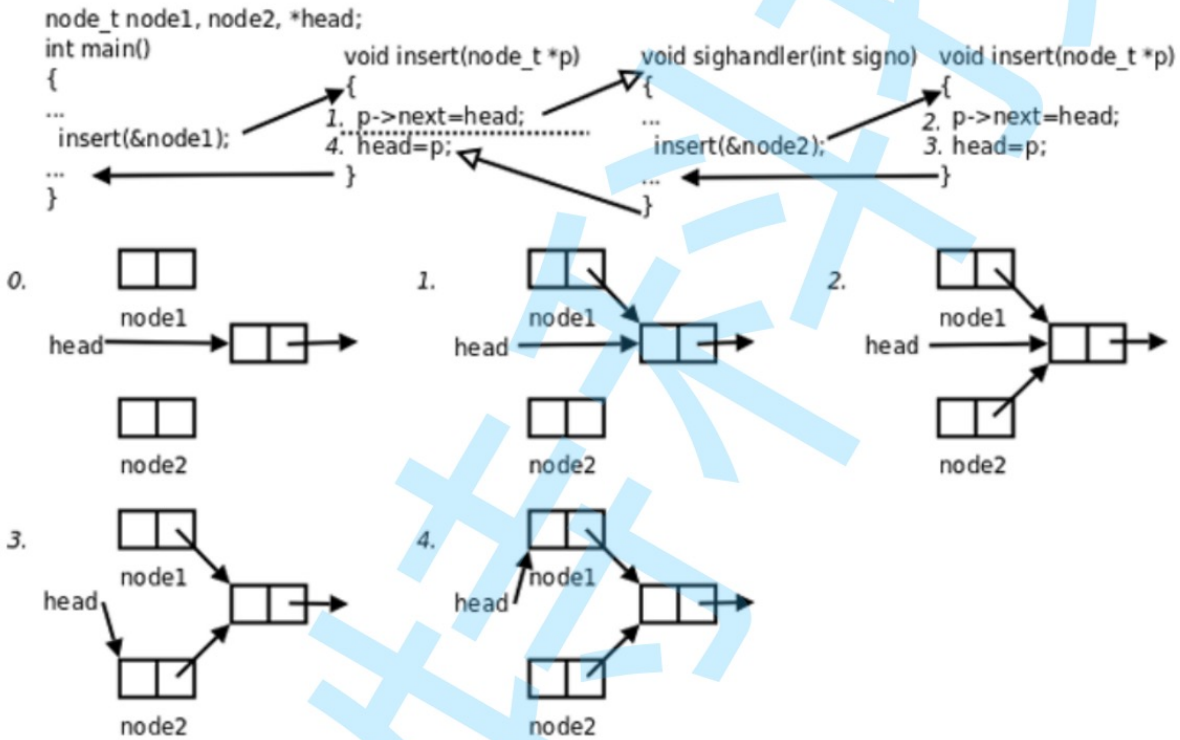
假设，比如我在调用链表插入这个insert函数，调用到一半，进程被陷入内核，然后回来的时候，需要处理2号信号，刚好2号信号，就是调用这个insert函数，
此时insert函数被重入！

**如果一个函数，被重入不会出现问题，这种函数叫做可重入函数**

**如果像上面这个insert函数，被重入会出现问题，这样的函数叫做不可重入函数**

可重入函数 vs 不可重入函数
：是函数的一种特征，目前我们用的90%函数，都是不可重入的！

## 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?

# 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

```
void changeFlag(int signum)
{
    (void)signum;
    cout <<"change flag: " << flag;
    flag = 1;
    cout << "->" << flag << endl;
}
int main()
{
    signal(2, changeFlag);
    while (!flag)
        cout << "进程正常退出后: " << flag << endl;
    return 0;
}
```

期望在信号的处理动作里面

把flag改成1

然后让进程正常退出

```
TERMINAL
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0108]$ make
g++ -o mysignal mysignal.cc -std=c++11 -fpermissive
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0108]$ ./mysignal
^Cchange flag: 0->1
进程正常退出后: 1
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0108]$
```

这个也是我们的期望之中，但是我们编译环境会有不同

编译器有时候会自动的给我们进行代码优化

gcc、g++

mysignal1.cc | mysignal.cc | Makefile

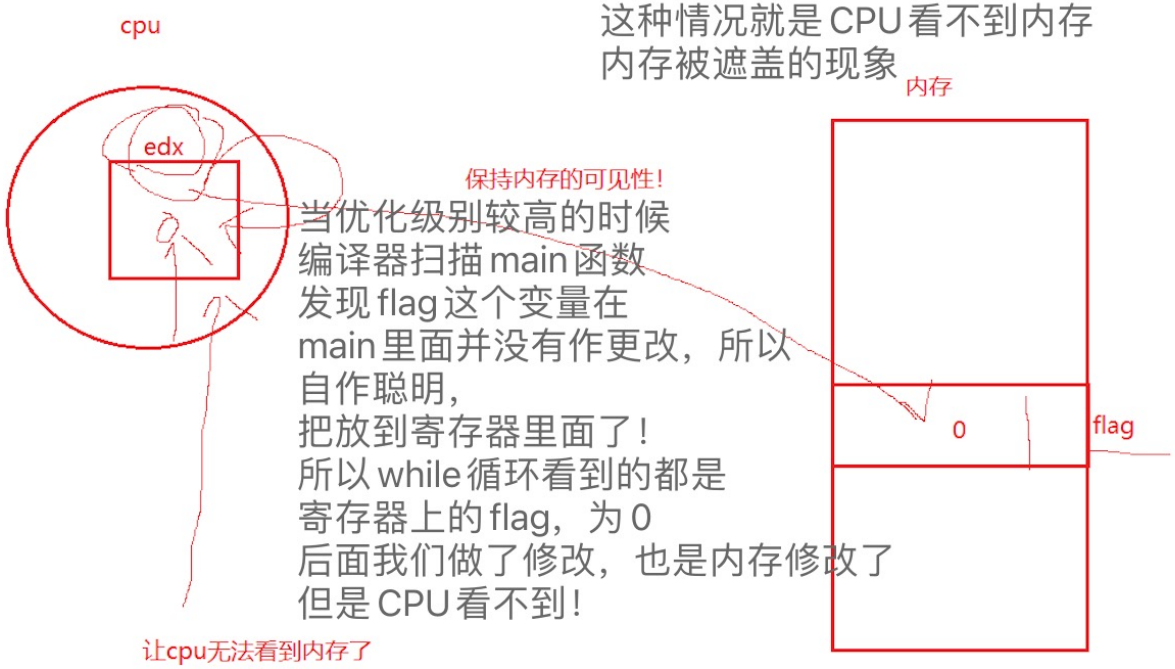
Files ▸ BitCodeField ▸ 0108 ▸ Makefile

```
1  mysignal:mysignal.cc
2      g++ -o $@ $^ -std=c++11 -O3
3  .PHONY:clean
4  clean:
5      rm -f mysignal
```

优化级别最高的！

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0108]$ make
g++ -o mysignal mysignal.cc -std=c++11 -fpermissive
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0108]$ ./mysignal
^Cchange flag: 0->1
进程正常退出后：1
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0108]$ make clean;make
rm -f mysignal
g++ -o mysignal mysignal.cc -std=c++11 -O3
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0108]$ ./mysignal
^Cchange flag: 0->1
^Cchange flag: 1->1
^Cchange flag: 1->1
^Cchange flag: 1->1
^Cchange flag: 1->1
^Cchange flag: 1->1
^Cchange flag: 1->1
^Cchange flag: 1->1
^Cchange flag: 1->1
```

为什么我们的进程没有退出呢？




这种情况就是CPU看不到内存
内存被遮盖的现象

当优化级别较高的时候
编译器扫描main函数
发现flag这个变量在
main里面并没有作更改，所以
自作聪明，
把放到寄存器里面了！
所以while循环看到的都是
寄存器上的flag，为0
后面我们做了修改，也是内存修改了
但是CPU看不到！

```
int flag = 0;
void changeFlag(int signum)
{
    (void)signum;
    cout <<"change flag: "<< flag;
    flag = 1;
    cout << "->" << flag << endl;
}

int main()
{
    signal(2, changeFlag);
    while(!flag);
    cout << "进程正常退出后: " << flag << endl;
}
```

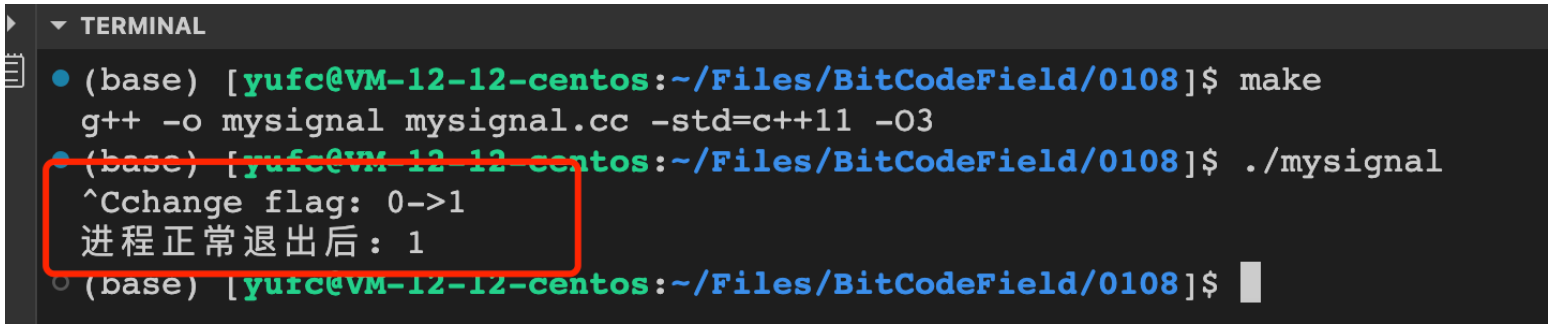
cout访问的是内存，所以打印的是1
但是我们的while循环访问的一直都是edx

**这种优化，是编译的时候就优化好了！**

```cpp
volatile int flag = 0;

void changeFlag(int signum)
{
    (void)signum;
    cout << "change flag: " << flag;
    flag = 1;
    cout << "->" << flag << endl;
}
int main()
{
    signal(2, changeFlag);
    while (!flag)
    {
        ;
    }
    cout << "进程正常退出后: " << flag << endl;
    return 0;
}
```

所以为了避免这种情况
需要程序员显性告诉编译器，这个东西不能优化！



## 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。

| Signal | Value | Action | Comment |
|---|---|---|---|
| **SIGHUP** | 1 | Term | Hangup detected on controlling terminal or death of controlling process |
| **SIGINT** | 2 | Term | Interrupt from keyboard |
| **SIGQUIT** | 3 | Core | Quit from keyboard |
| **SIGILL** | 4 | Core | Illegal Instruction |
| **SIGABRT** | 6 | Core | Abort signal from **abort**(3) |
| **SIGFPE** | 8 | Core | Floating point exception |
| **SIGKILL** | 9 | Term | Kill signal |
| **SIGSEGV** | 11 | Core | Invalid memory reference |
| **SIGPIPE** | 13 | Term | Broken pipe: write to pipe with no readers |
| **SIGALRM** | 14 | Term | Timer signal from **alarm**(2) |
| **SIGTERM** | 15 | Term | Termination signal |
| **SIGUSR1** | 30,10,16 | Term | User-defined signal 1 |
| **SIGUSR2** | 31,12,17 | Term | User-defined signal 2 |
| **SIGCHLD** | 20,17,18 | Ign | Child stopped or terminated |
| **SIGCONT** | 19,18,25 | Cont | Continue if stopped |
| **SIGSTOP** | 17,19,23 | Stop | Stop process |
| **SIGTSTP** | 18,20,24 | Stop | Stop typed at terminal |
| **SIGTTIN** | 21,21,26 | Stop | Terminal input for background process |
| **SIGTTOU** | 22,22,27 | Stop | Terminal output for background process |

子进程退出的时候是安安静静地退出吗？

我们学进程等待的时候，我们知道如果不选择阻塞等待，我们的父进程就要轮询检测子进程状态。

那么，子进程退出的时候，为啥不告诉父进程一声？而是等着父进程来检测？

**其实不是，子进程退出的时候，会通过OS向父进程发送17号信号（SIGCHLD），只不过是，进程对于17号信号的默认处理方式是：忽略！而已。**

其实SIGCHLD这个机制，其实是Linux自己才有的，别的OS没有

通过SIGCHLD那是不是不用父进程去轮训检测了是不是可以写一个通过信号回收子进程呢？

```cpp
// SIGCHLD
void handler(int signum)
{
    cout << "子进程退出: " << signum << endl;
}
int main()
{
    signal(SIGCHLD, handler);
    if (fork() == 0)
    {
        sleep(1);
        exit(0);
    }
    while (true)
        sleep(1);
    return 0;
}
```

DEBUG CONSOLE TERMINAL PORTS

TERMINAL

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0108]$ ./mysignal
子进程退出: 17
```

**证明子进程退出确实会向父进程发送17号信号**

那么现在又个问题
如果10个子进程都有退出
那么这个很简单
while遍历所有子进程就行

但是如果10个里面只有5个要退出呢？
比如现在5个进程都发了2号信号，
但是实际上！我们只能看到1个，因为pending位图里面只有一个位置，那怎么办？
如果同时发，就会出问题。（实时信号是会保存的，我们先不考虑）

所以while循环也要检测第6次，第7次...，如果第六个进程不退出，就一直阻塞在那里了。

所以我们可以用vector这些存好要退出的子进程的pid等非阻塞等待的方法

如果我们不想等待子进程
而且想让他自动释放
怎么办？

```
(base) [yufc@VM-12-12-centos:~/Files]$ ps axj | grep mysignal
 8128 19526 19526  8128 pts/74   19526 S+    1001   0:00 ./mysignal
19526 19527 19526  8128 pts/74   19526 Z+    1001   0:00 [mysignal] <defunct>
19570 19663 19662 19570 pts/75   19662 S+    1001   0:00 grep --color=auto mysignal
(base) [yufc@VM-12-12-centos:~/Files]$
```

如果按照以前的方式

肯定是会僵尸的，这个不用讲

```
int main()
{
    signal(SIGCHLD, SIG_IGN);//手动设置子进程忽略
    if (fork() == 0)
    {
        cout << "child: " << getpid() << endl;
        sleep(5);
        exit(0);
    }

    while (true)
```

我们手动设置子进程忽略

```
19570 19983 19982 19570 pts/75   19982 S+    1001   0:00 grep --color=auto mysignal
(base) [yufc@VM-12-12-centos:~/Files]$ ps axj | grep mysignal
 8128 20080 20080  8128 pts/74   20080 S+    1001   0:00 ./mysignal
19570 20315 20314 19570 pts/75   20314 S+    1001   0:00 grep --color=auto mysignal
(base) [yufc@VM-12-12-centos:~/Files]$
```

此时，就没有僵尸了！

现在的问题是：OS不是默认是忽略吗？我们手动设置干啥？
事实上，我们实验发现，我们显式在用户层面设置了，现象完全不一样。