0104信号

产生信号的方式：
1. 键盘

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

NAME
       signal - ANSI C signal handling

SYNOPSIS
       #include <signal.h>

       typedef void (*sighandler_t)(int);

       sighandler_t signal(int signum, sighandler_t handler);
```

这个函数的作用就是捕捉信号

回调函数，通过回调的方式，修改对应的信号捕捉方法

信号处理的方式
1. 默认
2. 忽略
3. **自定义捕捉**

sighandler_t就是一个函数指针
那么收到信号去干嘛呢？
就是作sighandler_t的事情

```cpp
#include <iostream>
#include <signal.h>
#include <unistd.h>

void catchSig(int sig)
{
    // 此时的signum就是捕捉到信号的编号
    std::cout << "进程捕捉到了一个信号,正在处理中: " << sig << " pid: " << getpid() << std::endl;
}
int main()
{
    signal(SIGINT, catchSig);
    while (true)
    {
        std::cout << "我是一个进程，我正在运行..., pid: " << getpid() << std::endl;
    }
    return 0;
}
```

捕捉到了，就过来做这件事

捕捉SIGINT这个信号！

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ make
make: `mysignal' is up to date.
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./mysignal
我是一个进程，我正在运行..., pid: 686
我是一个进程，我正在运行..., pid: 686
我是一个进程，我正在运行..., pid: 686
^C进程捕捉到了一个信号,正在处理中: 2 pid: 686
我是一个进程，我正在运行..., pid: 686
我是一个进程，我正在运行..., pid: 686
^C进程捕捉到了一个信号,正在处理中: 2 pid: 686
我是一个进程，我正在运行..., pid: 686
我是一个进程，我正在运行..., pid: 686
我是一个进程，我正在运行..., pid: 686
^C进程捕捉到了一个信号,正在处理中: 2 pid: 686
我是一个进程，我正在运行..., pid: 686
```

此时ctrl+c
进程不会退出了！
因为信号被捕捉了！

signal函数
仅仅只是修改进程对特定信号的后续处理动作，不是直接调用对应的处理动作

意思就是，调用完signal
catchSig并没有被调用
只有等到信号来了，才会被调用

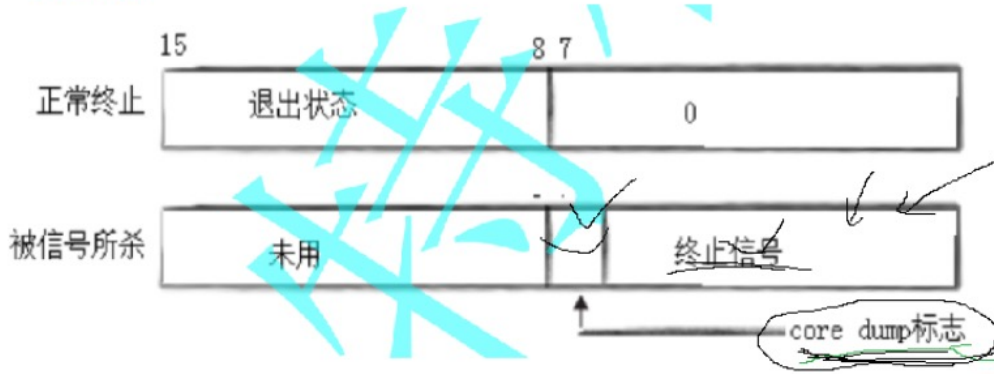
Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from **abort**(3)
SIGFPE	8	Core	Floating point exception
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

Term
和
Core是什么意思

重要概念：
核心转储

一般而言，云服务器（生产环境）的核心转储功能是被关闭的！

2. 核心转储



是否发生了核心转储

当进程出现某种异常的时候，是否由OS将当前进程在内存中的相关核心数据，转存到磁盘中！

主要是为了调试

core 应用场景

一般而言，云服务器(生产环境)的核心转储功能是被关闭的！

3. 验证进程等待中的core dump标记位

4. 为什么生产环境一般都是要关闭core dump?

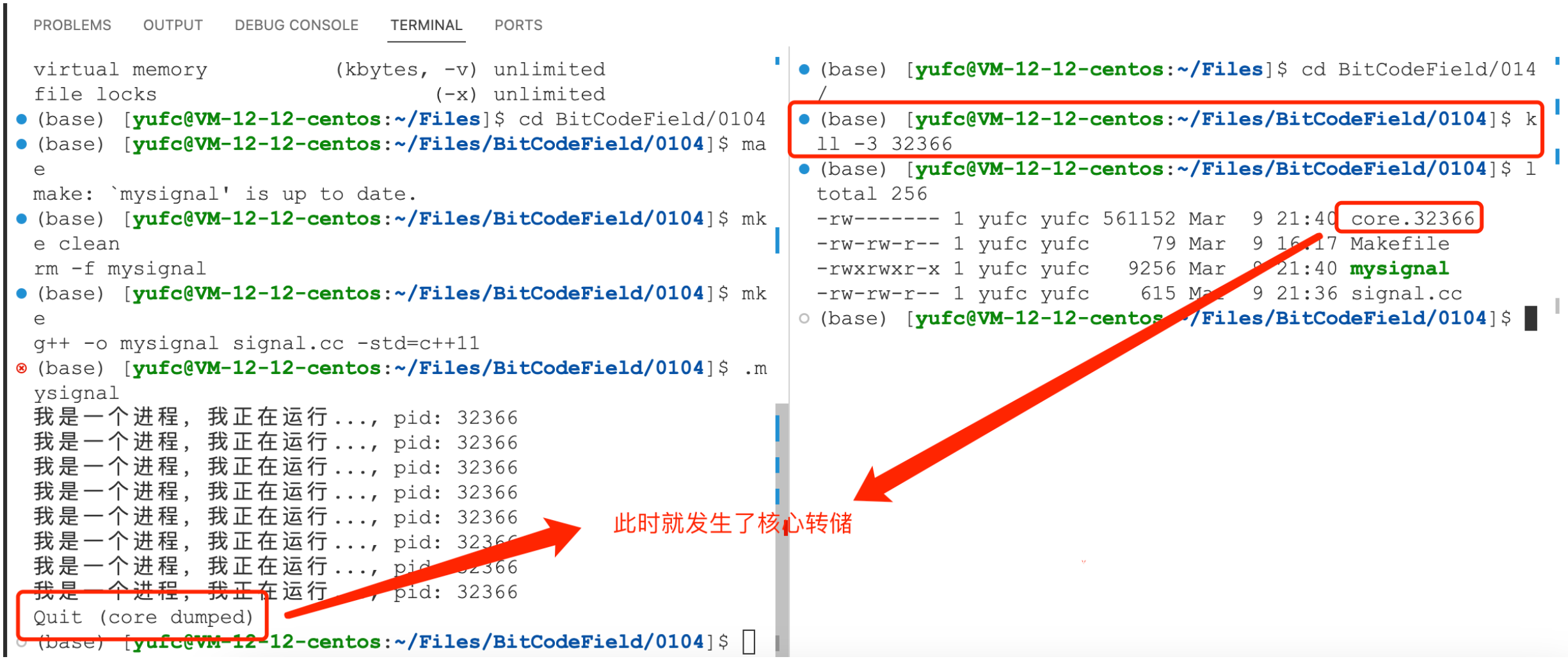
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(base) [yufc@VM-12-12-centos:~/Files]$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7260
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 100002
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 7260
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
(base) [yufc@VM-12-12-centos:~/Files]$
```

我们可以看到，我们服务器的核心转储是被禁用的了
怎么打开呢？

```
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
(base) [yufc@VM-12-12-centos:~/Files]$ ulimit -c 10240
(base) [yufc@VM-12-12-centos:~/Files]$ ulimit -a
core file size          (blocks, -c) 10240
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7260
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 100002
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 7260
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
(base) [yufc@VM-12-12-centos:~/Files]$
```

这个东西只是在当前会话下打开的
终端删了就会恢复的，所以随便搞


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
virtual memory (kbytes, -v) unlimited
file locks (-x) unlimited
(base) [yufc@VM-12-12-centos:~/Files]$ cd BitCodeField/0104
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ma
e
make: `mysignal' is up to date.
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ mk
e clean
rm -f mysignal
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ mk
e
g++ -o mysignal signal.cc -std=c++11
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ .m
ysignal
我是一个进程，我正在运行..., pid: 32366
我是一个进程，我正在运行..., pid: 32366
我是一个进程，我正在运行..., pid: 32366
我是一个进程，我正在运行..., pid: 32366
我是一个进程，我正在运行..., pid: 32366
我是一个进程，我正在运行..., pid: 32366
我是一个进程，我正在运行..., pid: 32366
我是一个进程，我正在运行..., pid: 32366
Quit (core dumped)
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$
此时就发生了核心转储
(base) [yufc@VM-12-12-centos:~/Files]$ cd BitCodeField/014
/
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ k
ll -3 32366
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ l
total 256
-rw------- 1 yufc yufc 561152 Mar 9 21:40 core.32366
-rw-rw-r-- 1 yufc yufc 79 Mar 9 16:17 Makefile
-rwxrwxr-x 1 yufc yufc 9256 Mar 9 21:40 mysignal
-rw-rw-r-- 1 yufc yufc 615 Mar 9 21:36 signal.cc
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$

```cpp
int main()
{
    // signal(SIGINT, catchSig);//特定信号的处理动作，一般只有一个，所有捕捉到了，进程不退出了
    // signal(SIGQUIT, catchSig);//3号信号是有coredump的
    while (true)
    {
        std::cout << "我是一个进程，我正在运行..., pid: " << getpid() << std::endl;
        int a = 0;
        int b = 4/a; //除0错误
        // sleep(1);
        std::cout<<"run here ... "<<std::endl;
    }
    return 0;
}
```

此时就会有一个除0错误
然后被终止

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ll
total 20
-rw-rw-r-- 1 yufc yufc    82 Mar  9 21:48 Makefile
-rwxrwxr-x 1 yufc yufc 9256 Mar  9 21:40 mysignal
-rw-rw-r-- 1 yufc yufc   718 Mar  9 21:48 signal.cc
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ make clean
rm -f mysignal
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ make
g++ -o mysignal signal.cc -std=c++11 -g
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./mysignal
我是一个进程，我正在运行..., pid: 1599
Floating point exception (core dumped)
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$
```

```
(base) [yufc@VM-12-12-centos:~/Files]$ cd BitCodeField/0104/
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ kill -3 32366
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ l
total 256
-rw------- 1 yufc yufc 561152 Mar  9 21:40 core.32366
-rw-rw-r-- 1 yufc yufc     79 Mar  9 16:17 Makefile
-rwxrwxr-x 1 yufc yufc   9256 Mar  9 21:40 mysignal
-rw-rw-r-- 1 yufc yufc    615 Mar  9 21:36 signal.cc
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ l
total 272
-rw------- 1 yufc yufc 561152 Mar  9 21:48 core.1599
-rw-rw-r-- 1 yufc yufc     82 Mar  9 21:48 Makefile
-rwxrwxr-x 1 yufc yufc  26344 Mar  9 21:48 mysignal
-rw-rw-r-- 1 yufc yufc    718 Mar  9 21:48 signal.cc
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$
```

我们说这个core文件，主要是为了调试
那具体怎么用呢？

```
-rw-rw-r-- 1 yufc yufc    718 Mar  9 21:48 signal.cc
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ gb mysignal
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
Copyright (C) 2013 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 "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/yufc/Files/BitCodeField/0104/mysignal...done.
(gdb) l
9           // 此时的signum就是捕捉到信号的编号
10          std::cout << "进程捕捉到了一个信号,正在处理中: " << sig << " pid: " <
< getpid() << std::endl;
11      }
12      int main()
13      {
14          // signal(SIGINT, catchSig);//特定信号的处理动作，一般只有一个，所有
捕捉到了，进程不退出了
15          // signal(SIGQUIT, catchSig);/
16          while (true)
17          {
18              std::cout << "我是一个进程
std::endl;
(gdb)
```

把gdb调起来

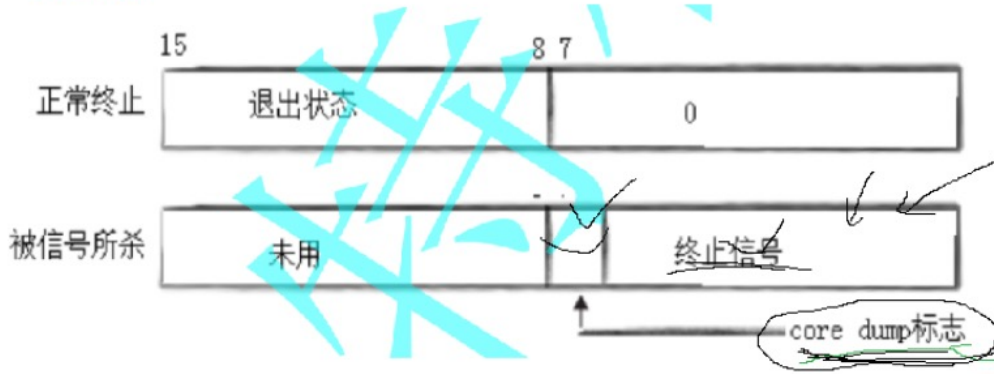
原来我们是要逐行去debug找错误的地方的
现在有了core文件
我们可以直接定位错误的地方

```
std::endl;
(gdb) core-file core.1599
[New LWP 1599]
Core was generated by `./mysignal'.
Program terminated with signal 8, Arithmetic exception.
#0  0x000000000040090a in main () at signal.cc:20
20              int b = 4/a; //除0错误
Missing separate debuginfos, use: debuginfo-install glibc-2.17-317.el7.x86_64 lib
gcc-4.8.5-44.el7.x86_64
(gdb)
```

直接找到这一行了

Ln 17, Col 6 Spaces: 4

2. 核心转储



是否发生了核心转储

当进程出现某种异常的时候，是否由OS将当前进程在内存中的相关核心数据，转存到磁盘中！

主要是为了调试

core 应用场景

一般而言，云服务器(生产环境)的核心转储功能是被关闭的！

3. 验证进程等待中的core dump标记位

4. 为什么生产环境一般都是要关闭core dump?

因此，此时这里的core dump就是标记是否发生了核心转储！

下面，我们就来验证一下

```cpp
using namespace std;
void catchSig(int sig)
{
    // 此时的signum就是捕捉到信号的编号
    std::cout << "进程捕捉到了一个信号,正在处理中: " << sig << " pid: " << getpid() << std::endl;
}
int main()
{
    pid_t id = fork();
    if (id == 0)
    {
        // 子进程
        sleep(1);
        int a = 100;
        a /= 0;
        exit(0);
    }

    int status = 0;
    waitpid(id, &status, 0);
    cout << "父进程: " << getpid() << "子进程: " << id
         << " exit sig: " << (status & 0x7f) << " is core: " << ((status >> 7) & 1) << endl;
    return 0;
}
```

```
(base) [yufc@VM-12-12-centos:~/Files]$ cd /home/yufc/Files/BitCodeField/0104/
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ make
g++ -o mysignal signal.cc -std=c++11 -g
signal.cc: In function 'int main()':
signal.cc:20:11: warning: division by zero [-Wdiv-by-zero]
          a /= 0;
            ^
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./mysignal
父进程: 2460子进程: 2461 exit sig: 8 is core: 0
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ulimit -c 10240
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./mysignal
父进程: 2619子进程: 2620 exit sig: 8 is core: 1
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$
```

发生了核心转储!

使用系统调用向进程发送信号

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

NAME
       kill - send signal to a process

SYNOPSIS
       #include <sys/types.h>
       #include <signal.h>

       int kill(pid_t pid, int sig);

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

       kill(): _POSIX_C_SOURCE >= 1 || _XOPEN_SOURCE || _POSIX_SOURCE
```

```cpp
//怎么调用 ./mykill 2 pid
static void Usage(string proc)
{
    cout << "Usage:\r\n\t" << proc << "signumber processid" << endl;
}
int main(int argc, char* argv[])
{
    if(argc != 3)
    {
        Usage(argv[0]);
        exit(1);
    }

    int signumber = atoi(argv[1]);
    int procid = atoi(argv[2]);
    kill(procid,signumber);
    return 0;
}
```

模拟实现的一个kill

OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./a.ot
7164
7164
7164
7164
7164
7164
7164
Killed
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$
```

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ mke
g++ -o mykill mykill.cc -std=c++11 -g
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ .mykill
Usage:
        ./mykillsignumber processid
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./mkill -2 6807
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./mkill -9 6807
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./mkill 9 7164
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$
```

同样，也可以写一个raise命令

```
RAISE(3)                    Linux Programmer's Manual                    RAISE(3)

NAME
       raise - send a signal to the caller

SYNOPSIS
       #include <signal.h>

       int raise(int sig);

DESCRIPTION
       The  raise()  function  sends  a signal to the calling process or thread.  In a single-threaded program it is
       equivalent to

           kill(getpid(), sig);
```

和它等价

```
static void Usage(string proc)
{
    cout << "Usage:\r\n\t" << proc << "signumber processid" << endl;
}
int main(int argc, char *argv[])
{
    cout << "我开始运行了" << endl;
    sleep(1);
    raise(8);
    return 0;
}
```

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ touch myraise.cc
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ make
g++ -o myraise myraise.cc -std=c++11 -g
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./myraise
我开始运行了
Floating point exception
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$
```

abort调用

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

```
ABORT(3)                                                    Linux Programm

NAME
       abort - cause abnormal process termination

SYNOPSIS
       #include <stdlib.h>

       void abort(void);
```

abort() 通常用来终止进程

```cpp
#include <iostream>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    std::cout << "我正在运行" << std::endl;
    sleep(1);
    abort();
    return 0;
}
```

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ make
g++ -o mysignal signal.cc -std=c++11 -g
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./mysignal
我正在运行
Aborted
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$
```

b. 系统调用接口

如何理解？用户调用系统接口 -> 执行OS对应的系统调用代码-> OS提取参数，或者设置特定的数值-> OS向目标进程写信号-> 修改对应进程的信号标记位->进程后续会处理信号-> 执行对应的处理动作！

c. 由软件条件产生信号

管道，读端不光不读，而且还关闭了，写端一直在写，会发生什么问题？写没有意义！OS会自动终止对应的写端进程，通过发送信号的方式，SIGPIPE！ 13) SIGPIPE

1. 创建匿名管道 2. 让父进程进行读取，子进程进行写入(why?) 3. 父子可以通信一段时间 4. 让父进程 关闭读端 && waitpid()，子进程只要一直写入就行 5. 子进程退出，父进程waitpid拿到子进程的退出status 6. 提取退出信号！

闹钟问题，可以写一下我刚刚写的两份代码　　1. IO的效率其实非常低 尤其是带上网络 2. 理解定时

如何理解软件条件给进程发送信号：a. OS先识别到某种软件条件触发或者不满足 b. OS 构建信号，发送给指定的进程

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？）

```cpp
int main()
{
    // 验证1s之内，我们一共会进行多少次count++
    // 1. 为什么我们只计算到1w+左右呢？主要是因为用了cout + 网络发送 = IO

    alarm(1);
    int count = 0;
    while (true)
    {
        std::cout << "count: " << count++ << std::endl;
    }
    return 0;
}
```

信号产生的第四种方式：硬件异常

```cpp
void handler(int signum)
{
    sleep(1);
    cout << "获得了一个信号: " << signum << endl;
}
int main()
{
    signal(SIGFPE, handler);
    int a = 100;
    a /= 0;
    while (true)
        sleep(1);
    return 0;
}
```

这个代码很简单
除0错误
会得到8号信号
但是我们看现象...

```
.(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ ./mysignal
获得了一个信号: 8
获得了一个信号: 8
```

为什么开始死循环了？

d. 硬件异常产生信号

如何理解除0呢?

1. 进行计算的是CPU，这个硬件
2. CPU内部是有寄存器的，状态寄存器（位图），有对应的状态标记位，溢出标记位，OS会自动进行计算完毕之后的检测！如果溢出标记位是1，OS里面识别到有溢出问题，立即只要找到当前谁在运行提取PID，OS完成信号发送的过程，进程会在合适的时候，进行处理
3. 一旦出现硬件异常，进程一定会退出吗？不一定！一般默认是退出，但是我们即便不退出，我们也做不了什么
4. 为什么会死循环？寄存器中的异常一直没有被解决！

如何理解野指针或者越界问题?

1. 都必须通过地址，找到目标位置
2. 我们语言上面的地址，全部都是虚拟地址
3. 将虚拟地址转成物理地址
4. 页表 + MMU(Memory Manager Unit, 硬件！！)

5. 野指针，越界-》非法地址-》MMU转化的时候，一定会报错!

所有的信号，有他的来源，但最终全部都是被OS识别，解释，并发送的!

```
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$ 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
(base) [yufc@VM-12-12-centos:~/Files/BitCodeField/0104]$
```

段错误