# Linux信号处理原理与实现机制

## 目录：

1. 信号的概述

1.1信号定义

1.2信号产生

1.3信号种类

1.4信号机制

1. 信号的发送

2.1终端发送

2.2内核发送

2.3进程发送

1. 信号的投递

3.1信号待决队列

3.2信号投递流程

1. 信号的处理

4.1 信号的阻塞

4.2 信号的忽略与捕获

4.3 异步信号安全

4.4 信号处理流程

## 信号的概述

**1.1信号定义**

在计算机科学中，信号是Unix、类Unix以及其他POSIX兼容的操作系统中进程间通讯的一种有限制的方式。它是一种异步的通知机制，用来提醒进程一个事件已经发生。当一个信号发送给一个进程，操作系统中断了进程正常的控制流程，此时，任何非原子操作都将被中断。如果进程定义了信号的处理函数，那么它将被执行，否则就执行默认的处理函数。其本质是在软件层次上对中断机制的一种模拟。

**1.2信号产生**

a)硬件产生：即通过终端按键产生的信号：

* ctrl + c：SIGINT(2)，发送给前台进程，& 进程放到后台运行，fg 把刚刚放到后台的进程，再放到前台来运行
* ctrl + z：SIGTSTP(20)，一般不用，除非有特定场景
* ctrl + | ：SIGQUIT(3)，产生 core dump 文件

产生 core dump 文件的条件：当前OS一定不要限制core dump文件的大小，ulimit -a。磁盘空间要足够

如何产生core dump：

* 1. 解引用空指针，收到11号信号，产生core dump文件
  2. 内存访问越界，程序一旦崩溃，就会收到11号信号，也就会产生core dump文件
  3. double free，收到6号信号，并产生core dump。
  4. free（NULL），不会崩溃

b)软件产生:即调用系统函数向进程发信号

* kill 函数
* abort：void abort(void);，收到 6 号信号，谁调用该函数，谁就收到信号
* alarm：unsigned int alarm(unsigned int seconds);，收到 14 号信号，告诉内核在 seconds 秒后给进程发送 SIGALRM 信号，该信号默认处理动作为终止当前进程。

**1.3信号种类**

1~31为标准（不可靠）信号，除了SIGUSR1和SIGUSR2之外，系统都赋予其特殊功能，如总线错误SIGBUS、非法内存访问SIGSEGV，常规信号不具有信号缓存特性，当一个信号在处理过程中，再来一个同样的信号，新来的信号会丢失，不会被缓存到队列。

32~64 为实时（可靠）信号,用户可以自定义功能，具有信号缓存特性。



我们先来解释一下信号0，其实0不算是一个信号，但是也可以算作是半个信号。因为发送信号0给一个进程或者线程，它会走发送检测过程，但是并不会真的投递给进程或者线程。检测流程会检测发送者是否有权限发送、进程是否存在，如果遇到问题就返回错误值。所以发送信号0可以用作检测进程是否存在的方法。

我们再来看一下实时信号，因为实时信号没有特定的含义，所以比较简单。实时信号的默认处理是终结进程，相关属性是可阻塞，可忽略，可捕获。它的一般使用方法都是进程发给其它进程或者线程来作为进程间通信的方法。其中32-33被glibc的pthread使用了。

标准信号一共有1-31共31个，我们按照它们的特点不同分类进行讲解：

首先说一下SIGKILL和一些暂停、继续相关的信号。其中SIGKILL和SIGSTOP是POSIX标准规定的不可阻塞、不可忽略、不可捕获的信号，它们的语义一定会得到执行。SIGCONT信号官方没有特别规定，它的实现上是不可阻塞、不可忽略的，虽然能捕获，但是相当于没捕获。因为捕获的意思是执行其信号处理函数就不再执行其默认处理了，但是SIGCONT的默认语义一定会得到执行。其它三个暂停信号SIGTSTP、SIGTTIN、SIGTTOU是不能阻塞的，但是可以忽略可以捕获，忽略或者捕获之后，它们的默认语义暂停程序就不会得到执行。SIGSTOP、SIGCONT，进程在想要暂停、恢复执行其它进程的时候可以发送这两个信号，内核里面再需要暂停、恢复执行进程的时候也会发送这两个信号。SIGTSTP是当在终端输入Ctrl+Z快捷键时，终端驱动会给当前进程发送这个信号。SIGTTIN是当后台进程读取终端的时候，终端会向进程发送的。SIGTTOU是在后台进程想要向终端输出的时候，终端会向进程发送的。这几个信号都是直接发送给进程的，因为它们的语义就是要操作整个进程。

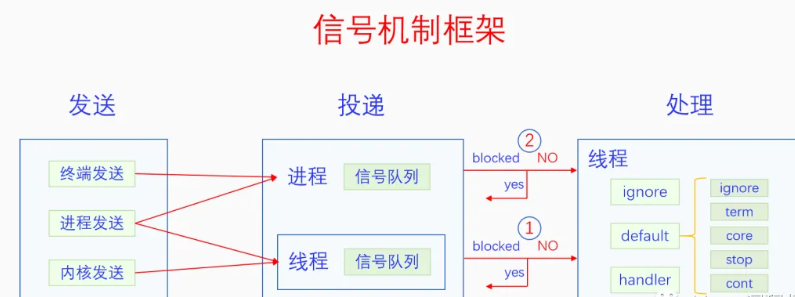
下面我们再来看6个标记紫色的信号，这几个信号都是和当前线程正在执行时发生异常有关。内核里单独把这6个信号放在一起成为同步信号。因为它们都是强制发送的，会忽略阻塞和忽略设置，所以图中把它们都看做是不可忽略不可阻塞的。但是它们是可以捕获的，让它们可以捕获的原因是因为这样可以让进程知道自己出错的原因，让进程可以在临死之前可以做一些记录工作，为程序员解BUG多提供一些信息。捕获了之后，原先默认的语义就不会执行，所以信号函数执行完之后它们还会继续执行。但是一般情况下这么做是没有意义的，所以一般都会在信号函数里退出进程。SIGSEGV的可捕获前面加了个[不]，代表的是不能二次捕获，也就是说如果在信号处理函数里面又发生了SIGSEGV，则这个SIGSEGV就不可捕获了，会走默认语义发生coredump并杀死进程。这些信号的发送方都是内核里异常处理相关的代码，信号都会发送给线程，因为是这些线程引起的这些问题，放到原线程里去处理比较好。

我们再接着看SIGABRT信号，这个信号比较特殊。它的目的是给库程序来用的。当库程序发现程序出现了不可挽回的错误，就会调用函数abort，这个函数会给当前线程发送信号SIGABRT。SIGABRT信号本身没什么特殊的，但是abort函数比较特殊。POSIX规范要求abort函数执行完成之后，进程一定要被杀死。于是abort函数的实现就是这样的，先取消阻塞SIGABRT信号，然后给当前线程发信号SIGABRT。无论SIGABRT信号是被忽略还是被捕获了，最后还是要返回到abort函数里面，然后abort函数就把SIGABRT信号的处理方式设置为默认，然后再发一个SIGABRT，这下进程就一定会死了。也就是说你可以捕获SIGABRT信号，但是进程最后还是一定会死。所以上图里说SIGABRT是不可阻塞、不可忽略、不可二次捕获的([不]可捕获代表的是不可二次捕获)。SIGABRT的不可二次捕获和SIGSEGV的不可二次捕获情形不太一样。如果是手工发送的SIGABRT信号，它就是一个普通的信号，没有前面说的逻辑。不过手工发送SIGABRT信号没有意义，一般都是使用abort函数来发送。其实遇到abort函数的SIGABRT信号也不是必死，有一种不规范的做法可以避免一死，那就是在信号处理函数中使用longjmp。但是这种做法没有意义，因为程序现在已经处于不一致状态了，coredump之后结束进程，然后好好地解bug才是最好的选择。

下面我们再看一下与终端相关的4个信号，SIGINT、SIGHUP、SIGQUIT、SIGTERM。你在终端上输入Ctrl+C，终端驱动就会给当前进程发送SIGINT，默认处理是杀死进程。你用kill命令给一个进程发信号，默认发的就是SIGTERM信号，默认处理也是杀死进程。当终端脱离进程的时候会给进程发SIGHUP，默认处理也是杀死进程。脱离终端有三种情况：一是物理终端与大型机断开了连接，现在已经没有物理终端了，所以这种情况不会有了；二是终端模拟器(也就是命令行窗口)被关闭了；三是我们通过ssh等工具连接到了网络终端，如果此时网络断了或者客户端程序死了。这三种情况终端驱动都会给关联的进程发送SIGHUP信号。最后一个信号是SIGTERM，当你在终端输入Ctrl+\的时候，终端驱动就会给当前进程发送SIGTERM信号，默认处理是coredump并杀死进程。

**1.4信号机制**

信号机制是UNIX系统最古老的机制之一，它不仅是内核处理程序在运行时发生错误的方式，还是终端管理进程的方式，并且还是一种进程间通信机制。信号机制由三部分构成，首先是信号是怎么产生的，或者说是谁发送的，然后是信号是怎么投递到进程或者线程的，最后是信号是怎么处理的。下面我们先看一张图，本文也将以这张图为基础进行展开：



## 信号的发送

我们来看一下信号发送的发送场景。具体的发送过程在下面信号的投递里面讲解。信号发送场景比较典型的有三种，一是终端发送，也就是我们在命令行运行程序时会遇到的情况；二是内核发送，内核也很庞大，里面的情况也很多，我们这里主要讲的是异常处理发送信号；三是进程发送，就是一个进程给另一个进程发。

**2.1终端发送**

下面代码展示了一个终端发送的场景：

/\* Send a signal to the slave \*/

static int pty\_signal(struct tty\_struct \*tty, int sig)

{

  struct pid \*pgrp;

  if (sig != SIGINT && sig != SIGQUIT && sig != SIGTSTP)

    return -EINVAL;

  if (tty->link) {

    pgrp = tty\_get\_pgrp(tty->link);

    if (pgrp)

      kill\_pgrp(pgrp, sig, 1);

    put\_pid(pgrp);

  }

  return 0;

}

**2.2内核发送**

我们最常遇到的信号SIGSEGV，一般都是在缺页异常里，如果我们访问的虚拟内存是未分配的虚拟内存，则会发生SIGSEGV。处理用户空间缺页异常的函数是do\_user\_addr\_fault，在这个函数里面会检测各种错误情况并最终调用函数\_\_bad\_area\_nosemaphore给当前线程发送信号SIGSEGV。

**2.3进程发送**

我们最常用的接口函数就是kill，它有两个参数，一个是进程标识符pid，一个是信号的值sig，就是把信号sig发给进程pid。raise函数给自己也就是当前线程发信号，它只有一个参数sig。killpg是给整个进程组发信号，在实现上是给进程组的每个进程都发信号。pthread\_kill是给同一个进程中的某个线程发信号。tgkill可以给其它进程中的某个线程发信号。sigqueue是用来发实时信号的，实时信号可以多带一个附加数据，当然可以用来发普通信号，但是这样附加数据就会被忽略。

|  |  |
| --- | --- |
| 函数 | 描述 |
| int kill(pid\_t pid, int sig); | 向指定的进程、指定进程组的所有成员或系统中的所有进程发送一个信号 |
| int raise(int sig); | 向调用该函数的线程发送一个信号 |
| int killpg(int pgrp, int sig); | 向指定进程组的所有成员发送一个信号 |
| int pthread\_kill(pthread\_t thread, int sig); | 向与调用者在同一进程中的指定POSIX线程发送一个信号 |
| int tgkill(pid\_t tgid, pid\_t tid, int sig); | 向指定进程内的指定线程发送一个信号（这是用于实现pthread\_kill(3)的系统调用） |
| int sigqueue(pid\_t pid, int sig, const union sigval value); | 向指定的进程发送一个带有附加数据的实时信号 |

## 信号的投递

上面我们讲了信号的发送场景，这一部分我们讲一讲信号是怎么发送到进程的。我们先讲一下进程的接收装置和信号来了放哪里、怎么放，然后再具体讲投递流程。

**3.1信号待决队列**

每个进程都有一个信号队列，每个线程也有一个信号队列。信号队列的数据结构如下所示：

linux-src/include/linux/signal\_types.h

struct sigpending {

struct list\_head list;

sigset\_t signal;

};

可以看到信号队列非常简单，sigset是个bit flag，代表当前队列里有哪些信号，list是信号列表的头指针。下面我们来看一下信号队列里的条目。

struct sigqueue {

struct list\_head list;

int flags;

kernel\_siginfo\_t info;

struct ucounts \*ucounts;

};

每发送一次信号都会生成一个sigqueue，sigqueue里面包含了很多和信号相关的信息。

在Linux里面，每个task\_struct都代表一个线程，里面包含了一个sigpending 。Linux里面没有直接代表进程的结构体，但是一个进程的所有线程都共享同一个signal\_struct。signal\_struct里面也包含了一个sigpending，这个sigpending代表进程的信号队列。

**3.2信号投递流程**

我们前面说了很多发送信号的方法，总体上可以分为两类，普通发送和强制发送。异常处理发送信号都是用的强制发送，其它的基本上都是用的普通发送，但也有一些其它情况用的是强制发送。这两类方法方法最终都会调用同一个函数来发送信号，即send\_signal，我们来看一下：

static int send\_signal(int sig, struct kernel\_siginfo \*info, struct task\_struct \*t, enum pid\_type type)

send\_signal做了一些简单的处理，然后直接调用\_\_send\_signal，进行实际的信号发送操作。

\_\_send\_signal先调用prepare\_signal：prepare\_signal对暂停恢复类的信号先做了一下预处理，然后查看信号是否被忽略。然后\_\_send\_signal根据PID类型决定是把信号放到进程队列里还是线程队列里。接着会判断信号是不是不可靠信号（标准信号）,对于标准信号，如果信号队列里已经有一个了，就不再接收了，这么做是为了兼容过去。然后调用\_\_sigqueue\_alloc分配一个信号条目sigqueue，分配好之后填充各种数据，然后把它加入到队列中去。最后调用complete\_signal，此函数会选择一个合适的线程来唤醒，一般会唤醒当前线程。唤醒的线程很可能醒来就去进行信号处理。

接着我们来看强制发送和普通发送：

1. **强制发送：**

* 入口函数是force\_sig\_info\_to\_task，它会先把信号的阻塞和忽略取消掉，然后再调用函数send\_signal进行发送。
* 内核还封装了几个函数来辅助强制发送，分别是force\_sig\_info、force\_sig、force\_fatal\_sig、force\_exit\_sig、force\_sigsegv、force\_sig\_fault\_to\_task、force\_sig\_fault。

入口函数代码如下：

static int

force\_sig\_info\_to\_task(struct kernel\_siginfo \*info, struct task\_struct \*t, enum sig\_handler handler)

{

  unsigned long int flags;

  int ret, blocked, ignored;

  struct k\_sigaction \*action;

  int sig = info->si\_signo;

  spin\_lock\_irqsave(&t->sighand->siglock, flags);

  action = &t->sighand->action[sig-1];

  ignored = action->sa.sa\_handler == SIG\_IGN;

  blocked = sigismember(&t->blocked, sig);

  if (blocked || ignored || (handler != HANDLER\_CURRENT)) {

    action->sa.sa\_handler = SIG\_DFL;

    if (handler == HANDLER\_EXIT)

      action->sa.sa\_flags |= SA\_IMMUTABLE;

    if (blocked) {

      sigdelset(&t->blocked, sig);

      recalc\_sigpending\_and\_wake(t);

    }

  }

  /\*

   \* Don't clear SIGNAL\_UNKILLABLE for traced tasks, users won't expect

   \* debugging to leave init killable. But HANDLER\_EXIT is always fatal.

   \*/

  if (action->sa.sa\_handler == SIG\_DFL &&

      (!t->ptrace || (handler == HANDLER\_EXIT)))

    t->signal->flags &= ~SIGNAL\_UNKILLABLE;

  ret = send\_signal(sig, info, t, PIDTYPE\_PID);

  spin\_unlock\_irqrestore(&t->sighand->siglock, flags);

  return ret;

}

1. **普通发送：**其主要发送函数有kill和tgkill。kill是发送信号给进程，只能向整个进程发送信号。而tgkill是发送信号给线程，允许向指定线程组中的特定线程发送信号。给进程发信号的接口函数最终都是调用的group\_send\_sig\_info。给线程发信号的接口函数最终都是调用的do\_send\_specific。因而，kill最后调用group\_send\_sig\_info；tgkill最后调用do\_send\_specific。

下面是Kill函数根据pid的发送对象：

* pid＞0：发送信号给进程ID等于pid的进程。
* pid＝0：发送信号给调用进程所在的同一个进程组的每一个进程。
* pid＝-1：有权限向调用进程发送信号的所有进程发出信号，init进程和进程自身除外。
* pid＜-1：向进程组-pid发送信号。

## 信号的处理

信号发送可以发送给进程或者线程，但是信号的处理是在线程中进行的，因为线程是代码执行的单元。线程首先处理自己队列里的信号，自己的处理完了再去处理进程队列里的信号。处理的时候要考虑信号掩码(mask)，被掩码阻塞的信号暂时不处理，还放回原队列中去。

信号处理方式有三种：

1. 如果程序什么也没设置的话，走默认处理(default)方式。默认处理有五种情况，不同的信号，其默认处理方式不同。这五种情况分别是ignore(忽略)、term(终结进程也就是杀死进程)、core(coredump内存转储并杀死进程)、stop(暂停进程)、cont(continue恢复执行进程)。
2. 还有一种方式是进程提前通过接口函数signal或者sigaction设置了处理方式，设置IGN来忽略信号
3. 最后一种是设置一个信号处理函数handler来处理信号。

这里需要注意，默认处理中的忽略和进程主动设置的忽略，两者的逻辑是不同的，一个是默认处理是忽略，一个是进程主动要求要忽略。你想要忽略一个默认处理不是忽略的信号，就必须要主动设置忽略。

下面我们来看一下信号的处理。在处理之前我们要先看看信号的阻塞，阻塞的信号是暂时不被处理的。然后看看信号的忽略与捕获，这是设置信号如何处理。最后再看信号的处理流程。

**4.1 信号的阻塞**

当一个线程暂时不想接收信号的时候，比如说正在处理一个关键的任务，就可以暂时屏蔽信号，等任务完成之后就可以解决屏蔽了。信号屏蔽是线程局部的，每个线程都可以有自己私有信号屏蔽设置。屏蔽信号的接口函数有两个，sigprocmask和pthread\_sigmask。前者是单线程时代的接口函数，后者是多线程时代POSIX规定的接口函数。两者实际上没有区别，下面我们以pthread\_sigmask为例来讲解一下。

int pthread\_sigmask(int how, const sigset\_t \*set, sigset\_t \*oldset);

sigset\_t是一个bit flag，参数set可以用来指定想屏蔽的信号，oldset用来返回线程之前的数据。how用来指定如何设置屏蔽参数，有三个值：SIG\_BLOCK，在线程原来blocked的基础上再增加blocked信号；SIG\_UNBLOCK，从线程原来的blocked信号中移除一些blocked信号；SIG\_SETMASK，直接把blocked信号设置为set，不考虑原有的设置。

如果你要阻塞SIGKILL和SIGSTOP，此函数实现并不会返回错误值，而是会默默地忽略，返回设置成功。

**4.2 信号的忽略与捕获**

信号的忽略与捕获的设置方法是一样，其实忽略可以看成是一种特殊的捕获，相当于是信号处理函数是空函数。

设置信号处理方式的接口函数有两个，signal和sigaction。signal是早期的设置函数，适用于标准信号，比较简单。sigaction是后来新增的接口函数，功能比较强大，适用于实时信号，当然也可以用于标准信号。

我们先来看signal函数接口：signal有两个参数，第一个是信号数值，第二个是信号处理函数。

sighandler\_t signal(int signum, sighandler\_t handler)；

信号处理函数的接口如下所示：第二个参数可以传递特殊值SIG\_IGN，代表忽略这个信号，还可以传递特殊值SIG\_DFL，代表恢复信号的默认处理方式。

typedef void (\*sighandler\_t)(int);

下面再来看sigaction函数的接口：有三个参数，第一个参数是信号数值，第二个参数是要设置的情况，第三个参数会返回旧的设置情况，可以为NULL。int sigaction(int signum, const struct sigaction \*restrict act, struct sigaction \*restrict oldact);

下面我们看一下struct sigaction结构体的定义：

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*)

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

};

**4.3 异步信号安全**

我们可以通过设置信号处理函数来捕获信号，那信号处理函数能像普通函数一样什么接口函数都能调用吗？不能，我们只能调用异步信号安全的函数。很多常用的函数都不是信号安全函数，不能在信号处理函数里面调用，比如printf。那要是想在信号处理函数里面输出数据该咋办呢？可以使用write接口函数，这个函数是异步信号安全的。具体都有哪些函数是异步信号安全的呢，可以看 https://man7.org/linux/man-pages/man7/signal-safety.7.html。

**4.4 信号处理流程**

信号处理是在线程从内核空间返回用户空间的时候处理的。而从内核空间返回用户空间是和架构相关的，所以这一部分的代码是在架构代码里面的。下面我们以x86为例讲解一下(代码进行了删减)。

tatic unsigned long exit\_to\_user\_mode\_loop(struct pt\_regs \*regs, unsigned long ti\_work)

{

  while (ti\_work & EXIT\_TO\_USER\_MODE\_WORK) {

    if (ti\_work & (\_TIF\_SIGPENDING | \_TIF\_NOTIFY\_SIGNAL))

      handle\_signal\_work(regs, ti\_work);

  }

  return ti\_work;

}  
static void handle\_signal\_work(struct pt\_regs \*regs, unsigned long ti\_work)

{

  if (ti\_work & \_TIF\_NOTIFY\_SIGNAL)

    tracehook\_notify\_signal();

  arch\_do\_signal\_or\_restart(regs, ti\_work & \_TIF\_SIGPENDING);

}

void arch\_do\_signal\_or\_restart(struct pt\_regs \*regs, bool has\_signal)

{

  struct ksignal ksig;

  if (has\_signal && get\_signal(&ksig)) {

    handle\_signal(&ksig, regs);

    return;

  }

  restore\_saved\_sigmask();

}

可以看出线程在返回到用户空间之前不断地检查有没有信号要处理。如果有的话就使用函数get\_signal取出一个信号，然后在函数handle\_signal里面去执行。

**我们先来看get\_signal函数的具体流程：**

1. 先看有没有STOP相关的信号，如果有的话执行处理。
2. 然后去取一个信号出来，先取同步信号，同步信号只从当前线程的信号队列里去取。

* 如果没有同步信号的话就去取其它信号，其它信号先从线程的信号队列里面去取，
* 如果没有的话就再去进程的信号里面去取。

c) 处理获取的信号：

* 如果取到的信号的处理设置是忽略，或者是默认处理但默认处理方式也是忽略，则继续取下一个信号。
* 如果取到的信号没有设置信号处理函数，则在这里执行其默认处理，终结进程或者coredump之后再终结进程。

d) 返回值

* 如果没有取到信号则get\_signal返回值为0
* 如果取到了信号，且信号设置了信号处理函数则返回值为1，且输出参数ksig会包含相应信号的相关的信息。然后把ksig传递给函数handle\_signal来处理。

**接着再来看handle\_signal函数：**

static void

handle\_signal(struct ksignal \*ksig, struct pt\_regs \*regs)

{

  bool stepping, failed;

  struct fpu \*fpu = &current->thread.fpu;

  if (v8086\_mode(regs))

    save\_v86\_state((struct kernel\_vm86\_regs \*) regs, VM86\_SIGNAL);

  /\* Are we from a system call? \*/

  if (syscall\_get\_nr(current, regs) != -1) {

    /\* If so, check system call restarting.. \*/

    switch (syscall\_get\_error(current, regs)) {

    case -ERESTART\_RESTARTBLOCK:

    case -ERESTARTNOHAND:

      regs->ax = -EINTR;

      break;

    case -ERESTARTSYS:

      if (!(ksig->ka.sa.sa\_flags & SA\_RESTART)) {

        regs->ax = -EINTR;

        break;

      }

      fallthrough;

    case -ERESTARTNOINTR:

      regs->ax = regs->orig\_ax;

      regs->ip -= 2;

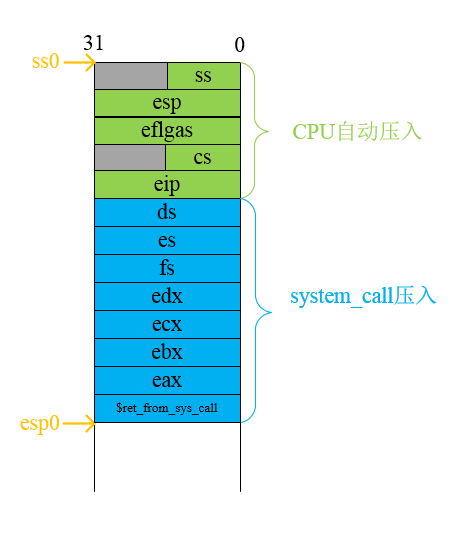
      break;

    }

  }

信号处理程序的代码是在用户态的，而从系统调用返回到用户态前还是属于内核态，CPU是禁止内核态执行用户态代码的，那么怎么办？答案是先返回到用户态执行信号处理程序;然后再返回到内核态，在内核态完成收尾工作。这里有两个问题需要解决：

**1.怎么从内核态返回到用户态**：CPU要从内核栈中找到返回到用户态的地址，我们把这个返回地址修改为信号处理程序的入口(即修改内核态eip)，这样当从系统调用返回到用户态时，就可以执行信号处理程序了。这个过程是通过调用setup\_frame函数来实现的。我们来看看内核栈的内存布局图：



图中的eip就是内核态返回到用户态后开始执行的第一条指令地址

**2.怎么返回到内核态并恢复原来内核栈的内容**：在用户态栈空间构建一个 Frame（帧）；然后setup\_frame()函数把原来内核栈的内容保存到帧中。修改用户栈eip的值，修改后跳转到用户栈的某一处，在这里进行sigreturn() 系统调用：sigreturn再调用 restore\_sigcontext()，这个函数从帧中读取原来内核栈的数据，然后恢复。

**参考文献：**  
《Linux Kernel Development》  
《Understanding the Linux Kernel》  
《Professional Linux Kernel Architecture》  
《The Linux Programming Interface》

https://man7.org/linux/man-pages/man7/signal.7.html  
https://man7.org/linux/man-pages/man7/signal-safety.7.html  
https://man7.org/linux/man-pages/man7/sigevent.7.html

http://akaedu.github.io/book/ch33.html  
http://gityuan.com/2015/12/20/signal/  
https://zhuanlan.zhihu.com/p/77598393  
https://zhuanlan.zhihu.com/p/79062142  
https://zhuanlan.zhihu.com/p/77627175  
https://zhuanlan.zhihu.com/p/78653866