## 关键字const

修饰变量时，表示该变量在赋初值之后不可修改；

如：const int limit = 10;

上面的代码等同于 int const limit = 10;

const 修饰它左侧的类型，左侧如果不存在就是右侧的类型。

如下面的两个定义是相同的，都表示指向只读整数类型的只读指针：

int const \* const p;

const int \* const p;

## 指针相容

指针相容，指的是指针A的类型和指针B的类型是相容的，那么可以A = B，A的类型和B是相同的，且A的类型的限定符包含B的类型的限定符；

如下面是相容的（compatible）：

char \*cp;

const char \*ccp;

ccp = cp;

cp和ccp都是相同类型的指针，指向char，ccp包含了const限定符和cp的限定符（无限定符）；

但下面的不相容（incompatible）：

char \*\*cpp;

const char \*\* ccpp;

ccpp = cpp;

cpp和ccpp都不含有限定符，const修饰的是char，两者是不同的类型，不相容。

## 类型转换

自动类型转换：当执行算数运算时，操作数的类型如果不同，就会发生转换，数据类型一般朝着浮点精度更高，长度更长的方向转换，整形数如果转换为signed，不会丢失信息，就转换为signed，否则转换为unsigned。

ANSI标准采用值保留原则（value preserving），K&R采用的是无符号保留原则（unsigned preserving），所以下面代码-1分别被解释为负数和正数。

如下面代码：

#include <stdio.h>

void main()

{

if( -1 < (unsigned char)1)

{

printf("-1 is less than (unsigned char)1: ANSI semantics.\n");

}

else

{

printf("-1 is NOT less than (unsigned char)1: K&R semantics.\n");

}

}

在使用位域或者掩码时才使用无符号数，一般情况下使用有符号数，或者强制转换所有变量为相同的类型，避免将负数解释为超大的数。

## NUL和NULL

NULL是一个宏，它在几个标准头文件中定义，0是一个整型常量，'\0'是一个字符常量，而NUL是一个字符常量的名字。这几个术语都不可互换。

1、NULL用于表示什么也不指向，也就是空指针((void \*)0)

2、0可以被用于任何地方，它是表示各种类型零值的符号并且编译器会挑出它

3、'\0'应该只被用于结束字符串

4、NUL没有被定义于C和C++，它不应该被使用除非你自己定义它，像：#define nul '\0

NULL是在<stddef.h>头文件中专门为空指针定义的一个宏。NUL是ASCII字符集中第一个字符的名称，它对应于一个零值。C语言中没有NUL这样的预定义宏。注意：在ASCII字符集中，数字0对应于十进制值48，不要把数字0和'\0'(NUL)的值混同起来。

NULL可以被定义为(void \*)0，而NUL可以被定义为'\0'。NULL和NUL都可以被简单地定义为0，这时它们是等价的，可以互换使用，但这是一种不可取的方式。为了使程序读起来更清晰，维护起来更容易，你在程序中应该明确地将NULL定义为指针类型，而将NUL定义为字符类型。

对指针进行解引用操作可以获得它的值。从定义来看，NULL指针并未指向任何东西。因此，对一个NULL指针进行解引用操作是非法的。在对指针进行解引用操作之前，必须确保它并非NULL指针。

例如：

#include <stdio.h>

#include <stdlib.h>  
    #define NUL '\0'

int \*ptr = NULL;  
    char ch = NUL;  
\r和\n都是一个字符，分别表示回车和换行，在ASC码里值分别为0x0D、0x0A，windows中文本一行的结尾都是有两个字符\r\n，但是在unix、linux里面只有一个字符\n，Mac里面则是只有字符\r，但是你是用printf("\r");或printf("\n");大部分程序（某些控制台程序除外）都是没有太大区别的。  
至于字符为什么写成\加上0、r和n等，是因为在C语言中表示字符或字符串时遇到\（转义字符）则不会把下一个字符识别为普通字符，比如说'\n'，表示ASC码0x0A，而不是值为0x6E的字母n。

## 字符串连接

相邻的字符串字面值会自动连接为一个字面值，有两种方式，如下：

printf("\

A favorite children's book,\

is 'muffy Gets It: the hilarious tale of a cat,\

a boy, and his machine gun'");

printf("A second favorite children's book is "

"'Thoms the tank engine and Naughty Enginedriver who"

" tied down Thoms's boiler safety value'");

## 运算顺序

.的优先级高于\*；\*p.f表示\*(p.f)，可以用p->f来表达(\*p).f

[]的优先级高于\*；int\*ap[]表示int\*(a[])，即数组元素是int指针。

函数()的优先级高于\*；int\*fp()表示int\*(fp())，表示函数返回值是int指针。

==和！=高于位操作和赋值；(val&mask!=0)表示val&(mask!=0)。c=getchar()!=EOF表示c=(getchar()!=EOF)

算数运算高于位移；msb<<4+lsb表示msb<<(4+lsb))

逗号运算符优先级最低；i=1,2表示(i=1),2，i的值是1，2被丢弃；

x = f() + g() \* h(); f,g,h三个函数的调用顺序并无确定顺序，确定的只有\*优先于+。函数参数调用中的计算顺序是不确定的。

所有的赋值符具有右结合性。位操作符具有左结合性。a = b = c，表示a = (b = c)。逻辑运算符是右结合性，以便更快得出结果，省去后续的计算。

谨记：算数运算符加入()来明确表明优先级，在运算表达式中不依赖于函数调用顺序；

## 关于结构

1、变量的定义和声明应该分开：

2、应该始终在结构定义中加入标签，在定义变量时减少重复编码。

//类型定义和变量声明分开

struct veg

{

int weight, price\_per\_1b;

};

struct veg onion, radish, turnip;

3、结构体在作为函数参数时是拷贝传值。

4、在结构体中定义指向自身的指针来实现链表、树等动态数据结构。

## 关于联合

1. 定义一组不可能同时存在的数据类型的组合；
2. 将同一数据解释成不同形式；

## 关于枚举

1. 缺省值从0开始，逐个增加1；
2. 枚举值类型是整型，且只能作为右值；
3. 枚举类型的变量可以作为左值，且可以被赋值为枚举值之外的值；

## 声明规则

1. 从它的最左边标识符开始读取，然后按照优先级顺序依次读取；
2. 优先级由高到低依次是：
   1. 声明中被括号括起来的那部分
   2. 后缀操作符：()表示这是一个函数，[]表示这是一个数组
   3. 前缀操作符：\*表示指向…的指针，函数的返回值类型，数组的元素类型；
3. 如果const、volatile关键字后面紧跟着类型说明符（如int，long等），那么它作用于类型说明符。在其他情况下，作用于它左边的紧邻的指针\*；

如 char\*const\*(\*next)();表示next是一个指针，指向一个函数，函数的返回值是一个指针，指针的类型是指向char的const指针。

再如char\*(\*c[10])(int \*\*p);表示c是一个数组，数组的类型是指针，指向一个函数，函数的参数是int的指针的指针，返回值是指向char的指针。

## 关于typedef

表示为原来的类型声明一个新的名字，并没有产生一个新的类型。

void ( \* signal **( int sig, void ( \* func ) ( int ) )** ) ( int ).

signal是一个函数，参数之一是一个整数，之二是一个函数指针，参数是证明，无返回值。Signal的返回值是一个指针，指向一个函数，函数的参数是int，无返回值。

typedef void(\*ptr\_to\_func)(int);表示ptr\_to\_func是一个函数指针，函数的参数是int，无返回值。

ptr\_to\_func signal(int sig, ptr\_to\_func);

1. 不要在一个typedef 中放入几个声明器：typedef int \*ptr, (\*fun)(), arr[5];
2. 不要把typedef嵌到声明中的中间部分。unsigned const long typedef int volatile \*kumquat;
3. typedef使用在数组，结构，指针及函数等组合类型、可移植类型；
4. typedef可以用于类型转换

## 关于可变参数宏

C99中规定宏可以像函数一样带有可变参数，比如

#define LOG(format, ...) fprintf(stdout, format, \_\_VA\_ARGS\_\_)

其中，...表示参数可变，\_\_VA\_ARGS\_\_在预处理中为实际的参数集所替换

GCC中同时支持如下的形式

#define LOG(format, args...) fprintf(stdout, format, args)

其用法和上面的基本一致，只是参数符号有变化

有一点需要注意，上述的宏定义不能省略可变参数，尽管你可以传递一个空参数，这里有必要提到"##"连接符号的用法。

"##"的作用是对token进行连接，在上例中，format、\_\_VA\_ARGS\_\_、args即是token，

如果token为空，那么不进行连接，所以允许省略可变参数(\_\_VA\_ARGS\_\_和args)，对上述变参宏做如下修改

#define LOG(format, ...) fprintf(stdout, format, ##\_\_VA\_ARGS\_\_)

#define LOG(format, args...) fprintf(stdout, format, ##args)

gcc用##解决该问题： 如果可变参数列表为空，就会将紧挨着")"的“,”去掉。

## 关于宏

当然宏定义非常重要的，它可以帮助我们防止出错，提高代码的可移植性和可读性等。

下面列举一些成熟软件中常用得宏定义

1，防止一个头文件被重复包含  
#ifndef COMDEF\_H  
#define COMDEF\_H

//头文件内容 …  
#endif

2，重新定义一些类型，防止由于各种平台和编译器的不同，而产生的类型字节数差异，方便移植。  
typedef  unsigned long int  uint32;      /\* Unsigned 32 bit value \*/

3，得到指定地址上的一个字节或字  
#define  MEM\_B( x )  ( \*( (byte \*) (x) ) )  
#define  MEM\_W( x )  ( \*( (word \*) (x) ) )

4，求最大值和最小值  
#define  MAX( x, y )  ( ((x) > (y)) ? (x) : (y) )  
#define  MIN( x, y )  ( ((x) < (y)) ? (x) : (y) )

5，得到一个field在结构体(struct)中的偏移量  
#define FPOS( type, field )   ( (dword) &(( type \*) 0)-> field )

6,得到一个结构体中field所占用的字节数  
#define FSIZ( type, field ) sizeof( ((type \*) 0)->field )

7，按照LSB格式把两个字节转化为一个word  
#define  FLIPW( ray ) ( (((word) (ray)[0]) \* 256) + (ray)[1] )

8，按照LSB格式把一个word转化为两个字节  
#define  FLOPW( ray, val )   
(ray)[0] = ((val) / 256);   
(ray)[1] = ((val) & 0xFF)

9，得到一个变量的地址（word宽度）  
#define  B\_PTR( var )  ( (byte \*) (void \*) &(var) )  
#define  W\_PTR( var )  ( (word \*) (void \*) &(var) )

10，得到一个字的高位和低位字节  
#define  WORD\_LO(xxx)  ((byte) ((word)(var) & 255))  
#define  WORD\_HI(xxx)  ((byte) ((word)(var) >> 8))

11，返回一个比X大的最接近的8的倍数  
#define RND8( x )       ((((x) + 7) / 8 ) \* 8 )

12，将一个字母转换为大写  
#define  UPCASE( c ) ( ((c) >= ’a' && (c) <= ’z') ? ((c) - 0×20) : (c) )

13，判断字符是不是10进值的数字  
#define  DECCHK( c ) ((c) >= ’0′ && (c) <= ’9′)

14，判断字符是不是16进值的数字  
#define  HEXCHK( c ) ( ((c) >= ’0′ && (c) <= ’9′) ||  
((c) >= ’A' && (c) <= ’F') ||  
((c) >= ’a' && (c) <= ’f') )

15，防止溢出的一个方法  
#define  INC\_SAT( val )  (val = ((val)+1 > (val)) ? (val)+1 : (val))

16，返回数组元素的个数  
#define  ARR\_SIZE( a )  ( sizeof( (a) ) / sizeof( (a[0]) ) )

17，对于IO空间映射在存储空间的结构，输入输出处理  
#define inp(port)         (\*((volatile byte \*) (port)))  
#define inpw(port)        (\*((volatile word \*) (port)))  
#define inpdw(port)       (\*((volatile dword \*)(port)))

#define outp(port, val)   (\*((volatile byte \*) (port)) = ((byte) (val)))  
#define outpw(port, val)  (\*((volatile word \*) (port)) = ((word) (val)))  
#define outpdw(port, val) (\*((volatile dword \*) (port)) = ((dword) (val)))

18，使用一些宏跟踪调试  
ANSI标准说明了五个预定义的宏名。它们是：  
\_\_LINE\_\_  
\_\_FILE\_\_  
\_\_DATE\_\_  
\_\_TIME\_\_  
\_\_STDC\_\_

如果编译不是标准的，则可能仅支持以上宏名中的几个，或根本不支持。记住编译程序 也许还提供其它预定义的宏名。  
是行连接符，会将下一行和前一行连接成为一行，即将物理上的两行连接成逻辑上的一行  
\_\_FILE\_\_ 是内置宏 代表源文件的文件名  
\_\_LINE\_\_ 是内置宏，代表该行代码的所在行号  
\_\_DATE\_\_宏指令含有形式为月/日/年的串，表示源文件被翻译到代码时的日期。  
源代码翻译到目标代码的时间作为串包含在\_\_TIME\_\_ 中。串形式为时：分：秒。  
如果实现是标准的，则宏\_\_STDC\_\_含有十进制常量1。如果它含有任何其它数，则实现是非标准的。

可以定义宏，例如:  
当定义了\_DEBUG，输出数据信息和所在文件所在行

#ifdef \_DEBUG  
#define DEBUGMSG(msg,date) printf(msg);printf(“%d%d%d”,date,\_LINE\_,\_FILE\_)  
#else  
#define DEBUGMSG(msg,date)  
#endif

19，宏定义防止使用是错误  
用小括号包含。  
例如：#define ADD(a,b) （a+b）

用do{}while(0)语句包含多语句防止错误

例如：#difne DO(a,b) a+b;  
a++;  
应用时：if(….)  
DO(a,b); //产生错误  
else  
解决方法: #difne DO(a,b) do{a+b;  
a++;}while(0)

为什么需要do{…}while(0)形式?

总结了以下几个原因：

1)，空的宏定义避免warning:

#define foo() do{}while(0)

2)，存在一个独立的block，可以用来进行变量定义，进行比较复杂的实现。

3)，如果出现在判断语句过后的宏，这样可以保证作为一个整体来是实现：

#define foo(x)

action1();

action2();

在以下情况下：

if(NULL == pPointer)

    foo();

就会出现action2必然被执行的情况，而这显然不是程序设计的目的。

4)，以上的第3种情况用单独的{}也可以实现，但是为什么一定要一个do{}while(0)呢，看以下代码：

#define switch(x,y) {int tmp; tmp=x;x=y;y=tmp;}

if(x>y)

   switch(x,y);

else        //error, parse error before else

   otheraction();

在把宏引入代码中，会多出一个分号，从而会报错。

使用do{….}while(0) 把它包裹起来，成为一个独立的语法单元，从而不会与上下文发生混淆。同时因为绝大多数的编译器都能够识别do{…}while(0)这种无用的循环并进行优化，所以使用这种方法也不会导致程序的性能降低。

为什么很多linux内核中宏#defines用do { … } while(0)？

有很多原因：

（Dave Miller的说法）：

编译器对于空语句会给出告警，这是为什么#define FOO do{ }while(0);

给定一个基本块（局部可视域），定义很多局部变量；

（Ben Collins的说法）：

在条件代码中，允许定义复杂的宏。可以想像有很多行宏，如下代码

#define FOO(x)   
printf(“arg is %sn”, x);   
do\_something\_useful(x);  
现在，想像下面的应用:  
if (blah == 2)  
FOO(blah);  
展开后代码为:  
if (blah == 2)  
printf(“arg is %sn”, blah);  
do\_something\_useful(blah);;  
就像你看到的，if仅仅包含了printf（），而do\_something\_useful()调用是无条件调用。因此，如果用do { … } while(0)，结果是：  
if (blah == 2)  
do {  
printf(“arg is %sn”, blah);  
do\_something\_useful(blah);  
} while (0);  
这才是所期望的结果。  
（Per Persson的说法）：  
像 Miller and Collins指出的那样，需要一个块语句包含多个代码行和声明局部变量。但是，本质如下面例子代码：  
#define exch(x,y) { int tmp; tmp=x; x=y; y=tmp; }  
上面代码在有些时候却不能有效工作，下面代码是一个有两个分支的if语句：  
if (x > y)  
exch(x,y);               // Branch 1  
else  
do\_something();      // Branch 2  
展开后代码如下：  
if (x > y)

{                // Single-branch if-statement!!!  
int tmp;            // The one and only branch consists  
tmp = x;           // of the block.  
x = y;  
y = tmp;  
}  
;                // empty statement  
else                             // ERROR!!! “parse error before else”  
do\_something();  
问题是分号（；）出现在块后面。解决这个问题可以用do{}while（0）：  
if (x > y)  
do {  
int tmp;  
tmp = x;  
x = y;  
y = tmp;  
} while(0);  
else  
do\_something();  
（ Bart Trojanowski的说法）：  
Gcc加入了语句解释，它提供了一个替代do-while-0块的方法。对于上面的解决方法如下，并且更加符合常理  
#define FOO(arg) ({   
typeof(arg) lcl;    
lcl = bar(arg);     
lcl;                    
})  
这是一个奇怪的循环，它根本就只会运行一次，为什么不去掉外面的do{..}while结构呢？我曾一度在心里把它叫做“怪圈”。原来这也是非常巧妙的技巧。在工程中可能经常会引起麻烦，而上面的定义能够保证这些麻烦不会出现。下面是解释：  
假设有这样一个宏定义  
#define macro(condition)   
if(condition) dosomething()  
现在在程序中这样使用这个宏：  
if(temp)  
macro(i);  
else  
doanotherthing();  
一切看起来很正常，但是仔细想想。这个宏会展开成：  
if(temp)  
if(condition) dosomething();  
else  
doanotherthing();  
这时的else不是与第一个if语句匹配，而是错误的与第二个if语句进行了匹配，编译通过了，但是运行的结果一定是错误的。  
为了避免这个错误，我们使用do{….}while(0) 把它包裹起来，成为一个独立的语法单元，从而不会与上下文发生混淆。同时因为绝大多数的编译器都能够识别do{…}while(0)这种无用的循环并进行优化，所以使用这种方法也不会导致程序的性能降低。

另一个讲解  
这是为了含多条语句的宏的通用性  
因为默认规则是宏定义最后是不能加分号的，分号是在引用的时候加上的  
比如定义了一个宏fw(a,b)，那么在c文件里一定是这样引用  
fw(a,b);  
如果不用do…while，那么fw就得定义成:  
#define fw(a,b) {read((a));write((b));}  
那这样fw(a,b);展开后就成了:  
{read(a);write(b);};  
最后就多了个分号，这是语法错误  
而定义成do…while的话，展开后就是:  
do{read(a);write(b);}while(0);    完全正确  
所以要写一个包含多条语句的宏的话，不用do…while是不可能的

宏中#和##的用法

一、一般用法  
我们使用#把宏参数变为一个字符串,用##把两个宏参数贴合在一起.  
用法:  
＃include<cstdio>  
＃include<climits>  
using namespace std;

#define STR(s)     #s  
#define CONS(a,b)  int(a##e##b)

int main()

{

printf(STR(vck));               // 输出字符串vck  
printf(%dn, CONS(2,3));  // 2e3 输出:2000  
return 0;  
}

二、当宏参数是另一个宏的时候  
需要注意的是凡宏定义里有用’#'或’##’的地方宏参数是不会再展开.

1, 非’#'和’##’的情况  
#define TOW      (2)  
#define MUL(a,b) (a\*b)

printf(%d\*%d=%dn, TOW, TOW, MUL(TOW,TOW));  
这行的宏会被展开为：  
printf(%d\*%d=%dn, (2), (2), ((2)\*(2)));  
MUL里的参数TOW会被展开为(2).

2, 当有’#'或’##’的时候  
#define A          (2)  
#define STR(s)     #s  
#define CONS(a,b)  int(a##e##b)

printf(“int max: %sn”,  STR(INT\_MAX));    // INT\_MAX ＃include<climits>  
这行会被展开为：  
printf(“int max: %sn”, #INT\_MAX);

printf(%sn, CONS(A, A));               // compile error  
这一行则是：  
printf(%sn, int(AeA));

INT\_MAX和A都不会再被展开, 然而解决这个问题的方法很简单. 加多一层中间转换宏.  
加这层宏的用意是把所有宏的参数在这层里全部展开, 那么在转换宏里的那一个宏(\_STR)就能得到正确的宏参数.

#define A           (2)  
#define \_STR(s)     #s  
#define STR(s)      \_STR(s)                 // 转换宏  
#define \_CONS(a,b)  int(a##e##b)  
#define CONS(a,b)   \_CONS(a,b)       // 转换宏

printf(int max: %sn, STR(INT\_MAX));          // INT\_MAX，int型的最大值，为一个变量 ＃include<climits>  
输出为: int max: 0x7fffffff  
STR(INT\_MAX) –>  \_STR(0x7fffffff) 然后再转换成字符串；

printf(%dn, CONS(A, A));  
输出为：200  
CONS(A, A)  –>  \_CONS((2), (2))  –> int((2)e(2))

三、’#'和’##’的一些应用特例  
1、合并匿名变量名  
#define  \_\_ANONYMOUS1(type, var, line)  type  var##line  
#define  \_ANONYMOUS0(type, line)  \_\_ANONYMOUS1(type, \_anonymous, line)  
#define  ANONYMOUS(type)  \_ANONYMOUS0(type, \_\_LINE\_\_)  
例：ANONYMOUS(static int);  即: static int \_anonymous70;  70表示该行行号；  
第一层：ANONYMOUS(static int);  –>  \_\_ANONYMOUS0(static int, \_\_LINE\_\_);  
第二层：–>  \_\_\_ANONYMOUS1(static int, \_anonymous, 70);  
第三层：–>  static int  \_anonymous70;  
即每次只能解开当前层的宏，所以\_\_LINE\_\_在第二层才能被解开；

2、填充结构  
#define  FILL(a)   {a, #a}

enum IDD{OPEN, CLOSE};  
typedef struct MSG{  
IDD id;  
const char \* msg;  
}MSG;

MSG \_msg[] = {FILL(OPEN), FILL(CLOSE)};  
相当于：  
MSG \_msg[] = {{OPEN, “OPEN”},  
{CLOSE, ”CLOSE“}};

3、记录文件名  
#define  \_GET\_FILE\_NAME(f)   #f  
#define  GET\_FILE\_NAME(f)    \_GET\_FILE\_NAME(f)  
static char  FILE\_NAME[] = GET\_FILE\_NAME(\_\_FILE\_\_);

4、得到一个数值类型所对应的字符串缓冲大小  
#define  \_TYPE\_BUF\_SIZE(type)  sizeof #type  
#define  TYPE\_BUF\_SIZE(type)   \_TYPE\_BUF\_SIZE(type)  
char  buf[TYPE\_BUF\_SIZE(INT\_MAX)];  
–>  char  buf[\_TYPE\_BUF\_SIZE(0x7fffffff)];  
–>  char  buf[sizeof 0x7fffffff];  
这里相当于：  
char  buf[11];

## 关于定义和声明

无论定义是什么样的，程序总是按照声明的形式进行解释执行。比如：

A文件中，定义了数组int a[4];

B文件中声明为extern int \*a; 在B文件中访问a时，那么无论是\*a的方式，还是a[0]的方式，都是会把实际a的值(a存储的值，以&a[0]为地址的四字节长度的地址值)作为地址，再把地址中的值取出来访问。

所以，实际编写代码时，必须保持变量的定义和声明是一致的，实际上，如果将数据封装在文件内部，不对外暴露内部的数据细节，就不会有这样的问题了。

## 链接

### 动态链接

目的是ABI(Application Binary Interface)，介于应用程序和函数库二进制可执行文件所提供的服务之间的接口，该接口保持稳定。

**例子：**

有如下文件，lib目录下是准备好的动态库的源文件，根目录下是使用动态库的源文件：

.

├── lib

│   ├── libfruit.h

│   └── tomato.c

└── test.c

编译过程：

1、进入lib目录下，使用如下命令进行编译动态库：gcc tomato.c -fPIC -shared -o libfruit.so

-shared该选项指定生成动态连接库（让连接器生成T类型的导出符号表，有时候也生成弱连接W类型的导出符号），不用该标志外部程序无法连接。相当于一个可执行文件

-fPIC：表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要，而不能达到真正代码段共享的目的。

2、进入根目录下，使用如下命令进行编译：gcc test.c -L./lib -Wl,-rpath ./lib -lfruit -I./lib -o test

-L.：表示要连接的库在当前目录中

-lfruit：编译器查找动态连接库时有隐含的命名规则，即在给出的名字前面加上lib，后面加上.so来确定库的名称

-I：表示包含头文件的路径

-Wl,-rpath，指定编译好的程序在运行时动态库的目录。这种方法会将动态库路径写入到elf文件中去。-Wl, 表示后面的参数将传给 link 程序 ld （因为 gcc 可能会自动调用 ld ）。这里通过 gcc 的参数 "-Wl,-rpath" 指定。当指定多个动态库搜索路径时，路径之间用冒号 " ： " 分隔。注：也可以通过export LD\_LIBRARY\_PATH，但是会影响到其他程序的运行。当然如果有root权限的话，也可以修改/etc/ld.so.conf文件，将要添加的动态库搜索路径写入该文件中，然后调用/sbin/ldconfig来达到同样的目的。

3、通过命令ldd test来查看是否可以列出动态链接的库，如果可以看到libfruit.so则说明编译没有问题。

### 静态链接

**例子：**

有如下文件，lib目录下是准备好的动态库的源文件，根目录下是使用动态库的源文件：

.

├── lib

│   ├── libfruit.h

│   └── tomato.c

└── test.c

编译过程：

1. 进入lib目录下，使用如下命令进行编译目标文件：gcc -c tomato.c -o tomato.o
2. 继续使用如下命令编译生成静态库：ar rc libfruit.a tomato.o

该命令将hello.o添加到静态库文件libhello.a，ar命令就是用来创建、修改库的，也可以从库中提出单个模块，参数r表示在库中插入或者替换模块，c表示创建一个库，关于ar命令的详细使用规则可以参考文章http://blog.csdn.net/xuhongning/article/details/6365200。这一步将在libtest/lib目录中生成一个libhello.a文件。

3、进入根目录下，使用如下命令进行编译：gcc test.c -I./lib -L./lib -lfruit -static -o test1

-L.：表示要连接的库在当前目录中

-lfruit：编译器查找静态连接库时有隐含的命名规则，即在给出的名字前面加上lib，后面加上.a来确定库的名称

-I：表示包含头文件的路径

注：如果在-L的目录下有同名的动态库，那么会优先进行动态链接。此时，执行程序会找不到库，因为没有指定运行目录。

### 链接顺序

1. 在编译命令中，各个静态库的顺序很重要，所以建议把所有编译的源文件放在前面，而-l命令放在最后，例子：

gcc -lm main.c

/tmp/ccVKYhEq.o: In function `main':

main.c:(.text+0x31): undefined reference to `sin'

collect2: error: ld returned 1 exit status

1. 链接程序在链接时一般是优先链接动态库的，即在库的目录下有两个同名的库文件libxxx.a和libxxx.so，肯定链接的是libxxx.so，除非用-static参数指定链接静态库。
2. 静态库链接时搜索路径顺序：
   1. ld会去找GCC命令中的参数-L
   2. 再找gcc的环境变量LIBRARY\_PATH
   3. 再找内定目录 /lib /usr/lib /usr/local/lib 这是当初compile gcc时写在程序内的
3. 动态链接时、执行时搜索路径顺序:
   1. 编译目标代码时指定的动态库搜索路径；
   2. 环境变量LD\_LIBRARY\_PATH指定的动态库搜索路径；
   3. 配置文件/etc/ld.so.conf中指定的动态库搜索路径；
   4. 默认的动态库搜索路径/lib；
   5. 默认的动态库搜索路径/usr/lib。
4. 当对动态库与静态库混合连接的时候，使用-static会导致所有的库都使用静态连接的方式。这时需要作用-Wl的方式：

gcc test.c -L./lib -I./lib -Wl,-Bstatic -lfruit **-Wl,-Bdynamic -Wl,-rpath ./lib** –lmeat，粗体部分可以写成**-Wl,-Bdynamic,-rpath ./lib，具体原因见下面的参数说明**

另外还要注意系统的运行库使用动态连接的方式，所以当动态库在静态库前面连接时，必须在命令行最后使用动态连接的命令才能正常连接，如：

gcc test.c -L./lib -I./lib -Wl,-Bdynamic,-rpath ./lib -lmeat -Wl,-Bstatic -lfruit -Wl,-Bdynamic

最后的-Wl,-Bdynamic表示将缺省库链接模式恢复成动态链接。否则就会找不到库：

/usr/bin/ld: cannot find -lgcc\_s

/usr/bin/ld: cannot find -lgcc\_s

collect2: error: ld returned 1 exit status

**参数说明：**

其中用到的两个选项：-Wl,-Bstatic和-Wl,-Bdynamic。这两个选项是gcc的特殊选项，它会将选项的参数传递给链接器，作为链接器的选项。比如-Wl,-Bstatic告诉链接器使用-Bstatic选项，该选项是告诉链接器，对接下来的-l选项使用静态链接；-Wl,-Bdynamic就是告诉链接器对接下来的-l选项使用动态链接。下面是man gcc对-Wl,option的描述，

-Wl,option Pass option as an option to the linker. If option contains commas, it is split into multiple options at the commas. You can use this syntax to pass an argument to the option. For example, -Wl,-Map,output.map passes -Map output.map to the linker. When using the GNU linker, you can also get the same effect with -Wl,-Map=output.map.

下面是man ld分别对-Bstatic和-Bdynamic的描述，

-Bdynamic -dy -call\_shared Link against dynamic libraries. You may use this option multiple times on the command line: it affects library searching for -l options which follow it. -Bstatic -dn -non\_shared -static Do not link against shared libraries. You may use this option multiple times on the command line: it affects library searching for -l options which follow it. This option also implies --unresolved-symbols=report-all. This option can be used with -shared. Doing so means that a shared library is being created but that all of the library's external references must be resolved by pulling in entries from static libraries

### 关于interpositioning

也称为interposing，即通过编写与库函数同名的函数来取代库函数的行为。所以，不要让程序中的任何符号成为全局的，除非有意的把它们作为程序的接口之一。

## 关于a.out

a.out是assembler output（汇编程序输出）的缩写。文件以7F开头，紧跟在后面的第二个到第四个字节为“ELF”。ELF原意为“Extensible Linker Format”，可扩展连接格式。现在代表“Excutable and Linking Format”，可执行文件和连接格式。

Segment，段表示一个二进制文件相关的内容块，里面保存了和某种特定类型（如符号表条目）相关的所有信息。

Section是ELF文件中的最小组织单位，一个段一般包含几个section。

例子程序：

#include <stdlib.h>

char pear[40];

static double peach;

int mango = 13;

static long melon = 2001;

void main()

{

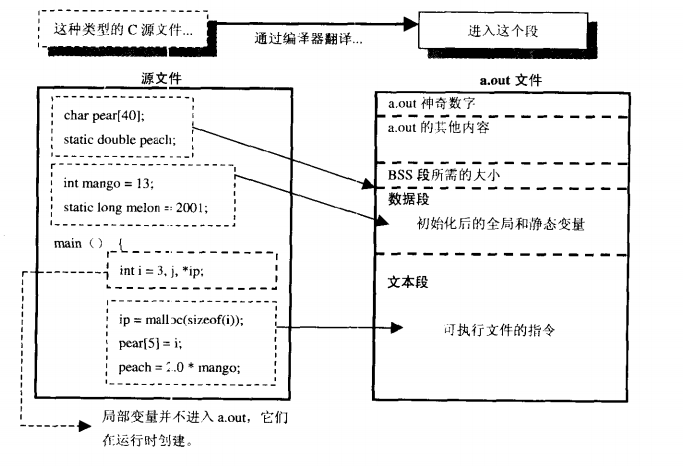
int i = 3, j, \*ip;

ip = malloc(sizeof(i));

pear[5] = i;

peach = 2.0 \* mango;

}



Bss段：“Block Stated by Symbol”由符号开始的块，它是旧式IBM704汇编程序的一个伪指令，Unix就用这个名字，沿用至今。BSS段只保存没有值的全局变量，所以它并不需要保存这些变量的映像。运行时所需要的BSS段的大小记录在目标文件中，但BSS段（不像其他段）并不占据目标文件的任何空间。

### 查看a.out文件的命令：

#### size

使用size命令可以查看可执行文件中的三个段的大小：

**$ size a.out**

text data bss dec hex filename

1473 312 72 1857 741 a.out

#### nm

Nm命令中，b表示私有（static），B表示全局的。

**$ nm -S a.out**

00002010 B \_\_bss\_start

00002020 00000001 b completed.7217

w \_\_cxa\_finalize@@GLIBC\_2.1.3

00002000 D \_\_data\_start

00002000 W data\_start

00000430 t deregister\_tm\_clones

000004c0 t \_\_do\_global\_dtors\_aux

00001edc t \_\_do\_global\_dtors\_aux\_fini\_array\_entry

00002004 D \_\_dso\_handle

00001ee0 d \_DYNAMIC

00002010 D \_edata

00002068 B \_end

000005e4 T \_fini

000005f8 00000004 R \_fp\_hw

00000510 t frame\_dummy

00001ed8 t \_\_frame\_dummy\_init\_array\_entry

00000718 r \_\_FRAME\_END\_\_

00001fd8 d \_GLOBAL\_OFFSET\_TABLE\_

w \_\_gmon\_start\_\_

00000600 r \_\_GNU\_EH\_FRAME\_HDR

0000037c T \_init

00001edc t \_\_init\_array\_end

00001ed8 t \_\_init\_array\_start

000005fc 00000004 R \_IO\_stdin\_used

w \_ITM\_deregisterTMCloneTable

w \_ITM\_registerTMCloneTable

000005e0 00000002 T \_\_libc\_csu\_fini

00000580 0000005d T \_\_libc\_csu\_init

U \_\_libc\_start\_main@@GLIBC\_2.0

0000051d 00000061 T main

U malloc@@GLIBC\_2.0

00002008 00000004 D mango

0000200c 00000004 d melon

00002028 00000008 b peach

00002040 00000028 B pear

00000470 t register\_tm\_clones

000003e0 T \_start

00002010 D \_\_TMC\_END\_\_

00000420 00000004 T \_\_x86.get\_pc\_thunk.bx

00000519 T \_\_x86.get\_pc\_thunk.dx

#### readelf

用readelf -s 或 objdump -t 查看符号表  
用readelf -S 或 objdump -h 查看段表

**$ readelf -s a.out**

Symbol table '.dynsym' contains 8 entries:

Num: Value Size Type Bind Vis Ndx Name

0: 00000000 0 NOTYPE LOCAL DEFAULT UND

1: 00000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_deregisterTMCloneTab

2: 00000000 0 FUNC WEAK DEFAULT UND \_\_cxa\_finalize@GLIBC\_2.1.3 (2)

3: 00000000 0 FUNC GLOBAL DEFAULT UND malloc@GLIBC\_2.0 (3)

4: 00000000 0 NOTYPE WEAK DEFAULT UND \_\_gmon\_start\_\_

5: 00000000 0 FUNC GLOBAL DEFAULT UND \_\_libc\_start\_main@GLIBC\_2.0 (3)

6: 00000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_registerTMCloneTable

7: 000005fc 4 OBJECT GLOBAL DEFAULT 16 \_IO\_stdin\_used

Symbol table '.symtab' contains 70 entries:

Num: Value Size Type Bind Vis Ndx Name

0: 00000000 0 NOTYPE LOCAL DEFAULT UND

1: 00000154 0 SECTION LOCAL DEFAULT 1

2: 00000168 0 SECTION LOCAL DEFAULT 2

3: 00000188 0 SECTION LOCAL DEFAULT 3

4: 000001ac 0 SECTION LOCAL DEFAULT 4

5: 000001cc 0 SECTION LOCAL DEFAULT 5

6: 0000024c 0 SECTION LOCAL DEFAULT 6

7: 000002ea 0 SECTION LOCAL DEFAULT 7

8: 000002fc 0 SECTION LOCAL DEFAULT 8

9: 0000032c 0 SECTION LOCAL DEFAULT 9

10: 0000036c 0 SECTION LOCAL DEFAULT 10

11: 0000037c 0 SECTION LOCAL DEFAULT 11

12: 000003a0 0 SECTION LOCAL DEFAULT 12

13: 000003d0 0 SECTION LOCAL DEFAULT 13

14: 000003e0 0 SECTION LOCAL DEFAULT 14

15: 000005e4 0 SECTION LOCAL DEFAULT 15

16: 000005f8 0 SECTION LOCAL DEFAULT 16

17: 00000600 0 SECTION LOCAL DEFAULT 17

18: 00000634 0 SECTION LOCAL DEFAULT 18

19: 00001ed8 0 SECTION LOCAL DEFAULT 19

20: 00001edc 0 SECTION LOCAL DEFAULT 20

21: 00001ee0 0 SECTION LOCAL DEFAULT 21

22: 00001fd8 0 SECTION LOCAL DEFAULT 22

23: 00002000 0 SECTION LOCAL DEFAULT 23

24: 00002020 0 SECTION LOCAL DEFAULT 24

25: 00000000 0 SECTION LOCAL DEFAULT 25

26: 00000000 0 FILE LOCAL DEFAULT ABS crtstuff.c

27: 00000430 0 FUNC LOCAL DEFAULT 14 deregister\_tm\_clones

28: 00000470 0 FUNC LOCAL DEFAULT 14 register\_tm\_clones

29: 000004c0 0 FUNC LOCAL DEFAULT 14 \_\_do\_global\_dtors\_aux

30: 00002020 1 OBJECT LOCAL DEFAULT 24 completed.7217

31: 00001edc 0 OBJECT LOCAL DEFAULT 20 \_\_do\_global\_dtors\_aux\_fin

32: 00000510 0 FUNC LOCAL DEFAULT 14 frame\_dummy

33: 00001ed8 0 OBJECT LOCAL DEFAULT 19 \_\_frame\_dummy\_init\_array\_

34: 00000000 0 FILE LOCAL DEFAULT ABS test.c

35: 00002028 8 OBJECT LOCAL DEFAULT 24 peach

36: 0000200c 4 OBJECT LOCAL DEFAULT 23 melon

37: 00000000 0 FILE LOCAL DEFAULT ABS crtstuff.c

38: 00000718 0 OBJECT LOCAL DEFAULT 18 \_\_FRAME\_END\_\_

39: 00000000 0 FILE LOCAL DEFAULT ABS

40: 00001edc 0 NOTYPE LOCAL DEFAULT 19 \_\_init\_array\_end

41: 00001ee0 0 OBJECT LOCAL DEFAULT 21 \_DYNAMIC

42: 00001ed8 0 NOTYPE LOCAL DEFAULT 19 \_\_init\_array\_start

43: 00000600 0 NOTYPE LOCAL DEFAULT 17 \_\_GNU\_EH\_FRAME\_HDR

44: 00001fd8 0 OBJECT LOCAL DEFAULT 22 \_GLOBAL\_OFFSET\_TABLE\_

45: 000005e0 2 FUNC GLOBAL DEFAULT 14 \_\_libc\_csu\_fini

46: 00002040 40 OBJECT GLOBAL DEFAULT 24 pear

47: 00000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_deregisterTMCloneTab

48: 00000420 4 FUNC GLOBAL HIDDEN 14 \_\_x86.get\_pc\_thunk.bx

49: 00002000 0 NOTYPE WEAK DEFAULT 23 data\_start

50: 00002010 0 NOTYPE GLOBAL DEFAULT 23 \_edata

51: 000005e4 0 FUNC GLOBAL DEFAULT 15 \_fini

52: 00000519 0 FUNC GLOBAL HIDDEN 14 \_\_x86.get\_pc\_thunk.dx

53: 00000000 0 FUNC WEAK DEFAULT UND \_\_cxa\_finalize@@GLIBC\_2.1

54: 00000000 0 FUNC GLOBAL DEFAULT UND malloc@@GLIBC\_2.0

55: 00002000 0 NOTYPE GLOBAL DEFAULT 23 \_\_data\_start

56: 00000000 0 NOTYPE WEAK DEFAULT UND \_\_gmon\_start\_\_

57: 00002004 0 OBJECT GLOBAL HIDDEN 23 \_\_dso\_handle

58: 000005fc 4 OBJECT GLOBAL DEFAULT 16 \_IO\_stdin\_used

59: 00002008 4 OBJECT GLOBAL DEFAULT 23 mango

60: 00000000 0 FUNC GLOBAL DEFAULT UND \_\_libc\_start\_main@@GLIBC\_

61: 00000580 93 FUNC GLOBAL DEFAULT 14 \_\_libc\_csu\_init

62: 00002068 0 NOTYPE GLOBAL DEFAULT 24 \_end

63: 000003e0 0 FUNC GLOBAL DEFAULT 14 \_start

64: 000005f8 4 OBJECT GLOBAL DEFAULT 16 \_fp\_hw

65: 00002010 0 NOTYPE GLOBAL DEFAULT 24 \_\_bss\_start

66: 0000051d 97 FUNC GLOBAL DEFAULT 14 main

67: 00002010 0 OBJECT GLOBAL HIDDEN 23 \_\_TMC\_END\_\_

68: 00000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_registerTMCloneTable

69: 0000037c 0 FUNC GLOBAL DEFAULT 11 \_init

**$ readelf -S a.out**

There are 29 section headers, starting at offset 0x17dc:

Section Headers:

[Nr] Name Type Addr Off Size ES Flg Lk Inf Al

[ 0] NULL 00000000 000000 000000 00 0 0 0

[ 1] .interp PROGBITS 00000154 000154 000013 00 A 0 0 1

[ 2] .note.ABI-tag NOTE 00000168 000168 000020 00 A 0 0 4

[ 3] .note.gnu.build-i NOTE 00000188 000188 000024 00 A 0 0 4

[ 4] .gnu.hash GNU\_HASH 000001ac 0001ac 000020 04 A 5 0 4

[ 5] .dynsym DYNSYM 000001cc 0001cc 000080 10 A 6 1 4

[ 6] .dynstr STRTAB 0000024c 00024c 00009d 00 A 0 0 1

[ 7] .gnu.version VERSYM 000002ea 0002ea 000010 02 A 5 0 2

[ 8] .gnu.version\_r VERNEED 000002fc 0002fc 000030 00 A 6 1 4

[ 9] .rel.dyn REL 0000032c 00032c 000040 08 A 5 0 4

[10] .rel.plt REL 0000036c 00036c 000010 08 AI 5 22 4

[11] .init PROGBITS 0000037c 00037c 000023 00 AX 0 0 4

[12] .plt PROGBITS 000003a0 0003a0 000030 04 AX 0 0 16

[13] .plt.got PROGBITS 000003d0 0003d0 000010 08 AX 0 0 8

[14] .text PROGBITS 000003e0 0003e0 000202 00 AX 0 0 16

[15] .fini PROGBITS 000005e4 0005e4 000014 00 AX 0 0 4

[16] .rodata PROGBITS 000005f8 0005f8 000008 00 A 0 0 4

[17] .eh\_frame\_hdr PROGBITS 00000600 000600 000034 00 A 0 0 4

[18] .eh\_frame PROGBITS 00000634 000634 0000e8 00 A 0 0 4

[19] .init\_array INIT\_ARRAY 00001ed8 000ed8 000004 04 WA 0 0 4

[20] .fini\_array FINI\_ARRAY 00001edc 000edc 000004 04 WA 0 0 4

[21] .dynamic DYNAMIC 00001ee0 000ee0 0000f8 08 WA 6 0 4

[22] .got PROGBITS 00001fd8 000fd8 000028 04 WA 0 0 4

[23] .data PROGBITS 00002000 001000 000010 00 WA 0 0 4

[24] .bss NOBITS 00002020 001010 000048 00 WA 0 0 32

[25] .comment PROGBITS 00000000 001010 000023 01 MS 0 0 1

[26] .symtab SYMTAB 00000000 001034 000460 10 27 45 4

[27] .strtab STRTAB 00000000 001494 00024a 00 0 0 1

[28] .shstrtab STRTAB 00000000 0016de 0000fc 00 0 0 1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),

L (link order), O (extra OS processing required), G (group), T (TLS),

C (compressed), x (unknown), o (OS specific), E (exclude),

p (processor specific)

### 局部变量对各段的影响

先把前面例子中的a.out文件信息和size信息打印出来：

-rwxrwxr-x 1 jason jason 7268 Nov 3 14:04 a.out\*

**$ size a.out**

text data bss dec hex filename

1473 312 72 1857 741 a.out

按照下面表格对代码进行修改，并观察：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 修改 | a.out大小 | 变化 | bss | 变化 | data | 变化 | text | 变化 | sum | 变化 |
| 无 | 7268 |  | 72 |  | 312 |  | 1473 |  | 1857 |  |
| int a[1000000]; | 7352 | 84 | 72 | 0 | 316 | 4 | 1676 | 203 | 2064 | 207 |
| int a[1000000] = {0}; | 7384 | 32 | 72 | 0 | 320 | 4 | 1757 | 81 | 2149 | 85 |
| int a[1000000] = {0}; int b[1000000]; | 7384 | 0 | 72 | 0 | 320 | 0 | 1757 | 0 | 2149 | 0 |
| int a[1000000] = {0}; int b[1000000] = {0}; | 7384 | 0 | 72 | 0 | 320 | 0 | 1773 | 16 | 2165 | 16 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000000]; | 7384 | 0 | 72 | 0 | 320 | 0 | 1773 | 0 | 2165 | 0 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000001] = {0}; | 7384 | 0 | 72 | 0 | 320 | 0 | 1805 | 32 | 2197 | 32 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000001] = {0}; static int x; | 7408 | 24 | 72 | 0 | 320 | 0 | 1805 | 0 | 2197 | 0 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000001] = {0}; static int x,y; | 7432 | 24 | 72 | 0 | 320 | 0 | 1805 | 0 | 2197 | 0 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000001] = {0}; static int x,y,z=0; | 7452 | 20 | 72 | 0 | 320 | 0 | 1805 | 0 | 2197 | 0 |
| int a[1000000] = {0}; int b[1000000] = {0}; int c[1000001] = {0}; static int x,y,z=1; | 7456 | 4 | 72 | 0 | 324 | 4 | 1805 | 0 | 2201 | 4 |

结论：

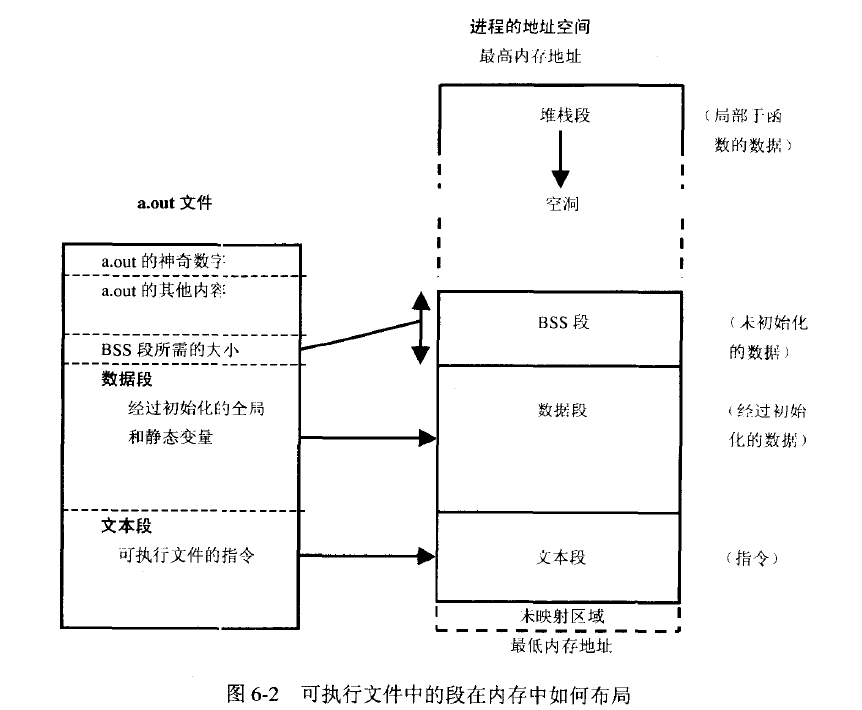
1. bss段只记录无初值全局变量，局部变量不影响bss，且bss段不占用a.out空间，只记录大小。
2. 数据段只记录有初值全局变量，局部变量不影响data段，data段占用a.out空间。
3. 局部变量的变化会影响text段，因为属于运行时的数据。
4. text段是最容易受优化措施影响的段。
5. a.out文件的大小受调试状态下编译的影响(gcc -g)，但是段不受影响。

## 运行时数据

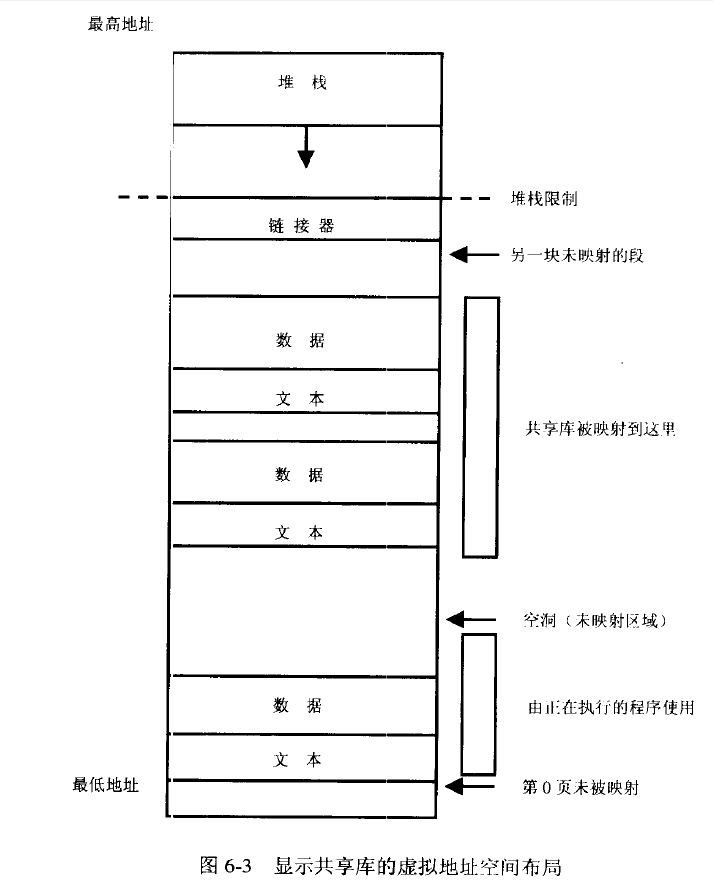
### 操作系统对a.out的操作

段在正在执行的程序中是一块内存区域，每个区域都有特定的目的。载入器只是取文件中每个段的映像，并直接将它们放入内存中。

文本段包含了程序的指令。数据段包含经过初始化的全局和静态变量以及它的值。BSS段的大小可以从文件中得到，链接器得到这个大小的内存块紧跟在数据段之后，该内存块的地址空间全部清零。数据段和BSS段统称为数据区。还需要堆栈段保存局部变量、临时变量、传递到函数的参数等。还需要堆空间用于动态申请的内存。虚拟地址空间的最低部分未被映射，任何对它的引用都是非法的。典型的，它是地址零开始的几K字节，它用于捕捉用于空指针和小整型值的指针引用内存的情况。



当考虑共享库的时，进程的地址空间如下：



### C语言运行时系统对a.out的操作

运行时数据结构有好几种：堆栈，活动记录（activation record），数据，堆等。

#### 堆栈段

堆栈段包含一种单一的数据结构——堆栈，它是一块动态内存区域，实现了一种后进先出的结构。函数可以通过参数或者全局指针访问它所调用的局部变量。运行时系统维护一个指针（常位于寄存器中），通常成为sp(stack pointer)，用于提示栈当前的顶部位置。

堆栈段有三个用途;

1. 堆栈为函数内部声明的局部变量提供存储空间，这些变量亦被称为自动变量。
2. 进行函数调用时，堆栈存储与此有关的一些维护信息。这些信息被称为堆栈结构（stack frame），或被称为过程活动记录（procedure activation record），它包括函数调用的地址（即所调用的函数结束后跳回的地方）、任何不适合装入寄存器的参数以及一些寄存器值得保存。
3. 堆栈也可被称为暂时存储区。有时需要一些临时存储区，比如计算一个很长的算术表达式时，可以把部分结果压入到栈中，当需要时再把它取出来。

在绝大部分的处理器中，堆栈是向下增长的，也即是朝着低地址的方向生长。如下面程序例子：

#include <stdio.h>

void f2();

int j;

int k = 10;

void f1()

{

int i[100000];

printf("The stack f1 is near %p.\n", &i);

f2();

}

void f2()

{

int i[100000];

printf("The stack f2 is near %p.\n", &i);

}

void main()

{

int i;

printf("The stack top is near %p.\n", &i);

f1();

printf("The bss addr is near %p.\n", &j);

printf("The data addr is near %p.\n", &k);

printf("The text addr is near %p.\n", &f1);

}

The stack top is near 0x7ffd90a6c9e4.

The stack f1 is near 0x7ffd90a0af40.

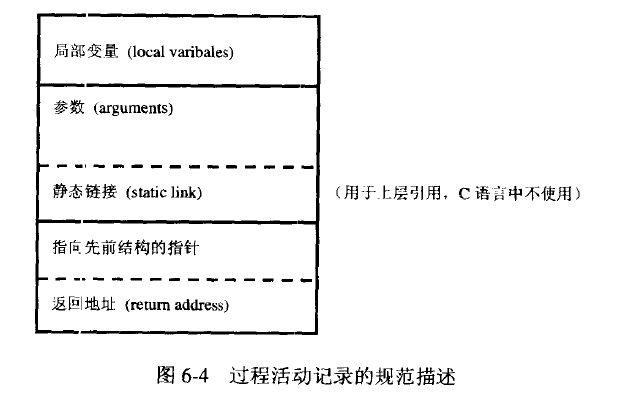
The stack f2 is near 0x7ffd909a94a0.

The bss addr is near 0x601048.

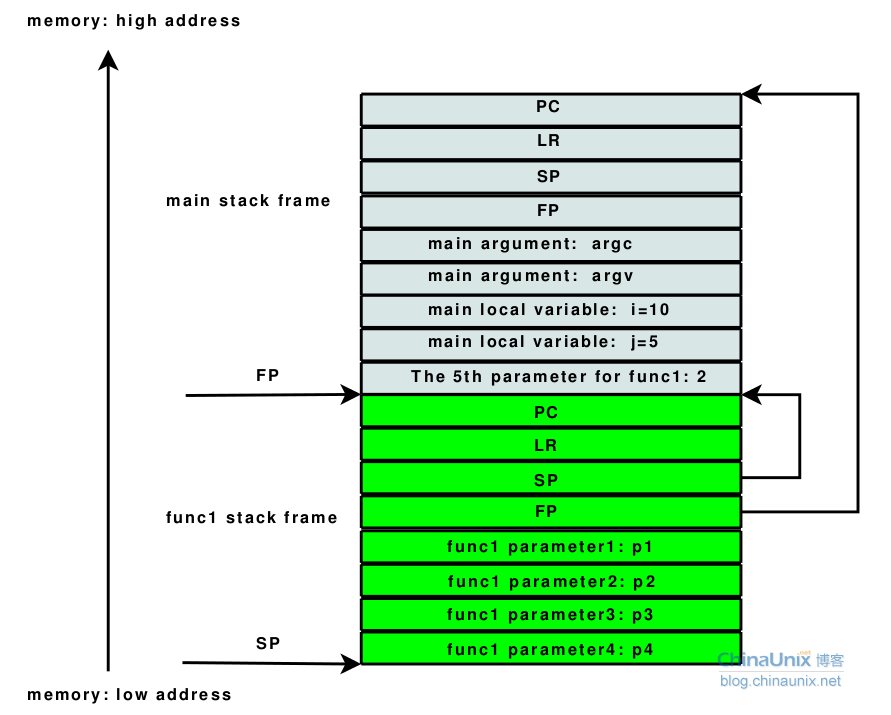
The data addr is near 0x601040.

The text addr is near 0x400596.

C语言提供了跟踪函数调用链，当下一个return语句执行时，控制将返回何处。当每个函数调用时，都会产生一个过程活动记录（或类似的结构），结构的具体细节在不同的编译器中各不相同，这些字段的次序很可能不相同，而且还可能存在一个调用函数前保存寄存器值得区域。运行时系统维护一个指针（常常位于寄存器中），通常称为fp(frame pointer)，用于提示活动堆栈结构。它的值是最靠近堆栈顶部的过程活动记录的地址。



每个函数所使用的栈空间是一个栈帧，所有的栈帧就组成了这个进程完整的栈。而fp就是栈基址寄存器，指向当前函数栈帧的栈底，sp则指向当前函数栈帧的栈顶。通过sp和fp所指出的栈帧可以恢复出母函数的栈帧，以此类推就可以backtrace出所有函数的调用顺序。



上图描述的是ARM的栈帧布局方式，main stack frame为调用函数的栈帧，func1 stack frame为当前函数(被调用者)的栈帧，栈底在高地址，栈向下增长。此图是网上的图，理论上应该是上图的格式，fp、sp、lr和pc这四个寄存器是非常特殊的寄存器，它们记录了当前正在运行的函数一些重要信息，在刚进入一个新的函数开始执行的时候，它们保存的是上个函数的信息，需要将它们入栈保存起来，这很重要！这些并没有定义在ATPCS（ARM System Developer's Guide中定义的函数调用的规则）中，ATPCS规定的是函数调用的时候参数如何传递，以及函数返回值的保存等。上面的这些个人觉得是一种默契，定义函数现场的保存及恢复，这些默契包括ATPCS都是人为的一种约束，目的是为了保证程序运行中不会出错，具体怎样实现应该是不同的编译器不尽相同。

例子：

#include <stdio.h>

void a(int i)

{

if(i > 0)

{

a(--i);

}

else

{

printf("i has reached zero!\n");

}

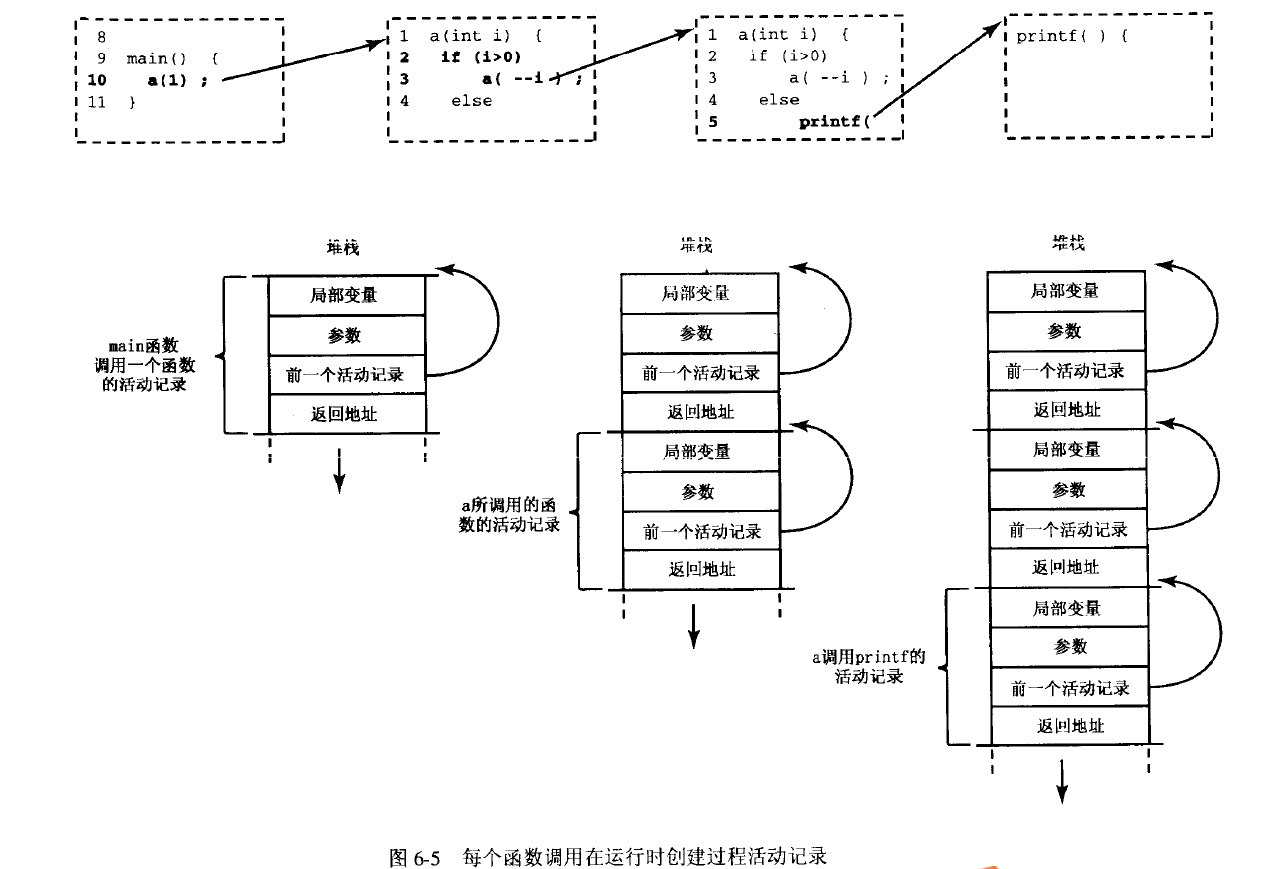
}

void main()

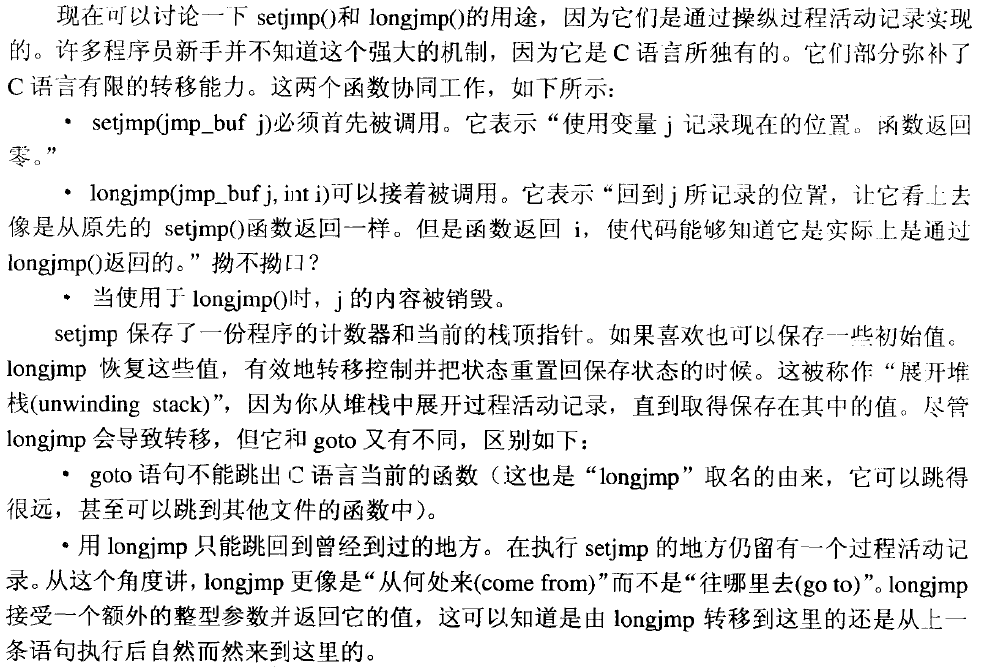
{

a(1);

}



### Setjmp和longjmp



Setjmp和longjmp最大的作用是用来恢复错误，只要还没有从函数中返回，一旦发现一个不可恢复的错误，可以把控制转移到主输入循环，并从那里重新开始。在C++中是catch和throw。

## 关于内存

常见错误：

1. 解除引用一个非法指针，如空指针，未赋初值的指针，或已经释放掉内存的指针。
2. 访问越界，如越过数组边界，或者在动态分配的内存两端之外写入数据。
3. 重复释放内存，释放未经过malloc的内存，释放空指针，释放正在使用的内存。

当内存出现非法使用时，系统会向当天进程发出sigsegv信号，可以自己实现信号的处理函数，并退出程序。

## 关于信号

信号被认为是一种软件中断（区别于硬件中断）。信号机制提供了一种在单进程/线程 下处理异步事件的方法。具体过程是当进程运行到某处，接受到一个信号，保留“现场”，响应信号（注意这里的响应是一种宏观意义上的响应，对信号的忽略（SIG\_IGN）也被以为是一种响应，后面会详细谈到信号响应的方式。），在返回到刚刚保存的地方继续运行。

由进程的某个操作产生的信号称为同步信号(synchronous signals),例如除0；由象用户击键这样的进程外部事件产生的信号叫做异步信号。(asynchronous signals)。

### 信号机制

信号处理可以分成两个阶段 : 信号产生并通知到接收方 (generation), 接收方进行处理 (deliver)

**内核如何向一个进程发送信号**:

内核给一个进程发送软中断信号的方法，是在进程所在的进程表项的信号域设置对应于该信号的位。由此可以看出，进程对不同的信号可以同时保留，但对于同一个信号，进程并不知道在处理之前来过多少个。

如果信号发送给一个正在睡眠的进程，那么要看该进程进入睡眠的优先级，如果进程睡眠在可被中断的优先级上，则唤醒进程；否则仅设置进程表中信号域相应的位，而不唤醒进程。

**内核在什么时机处理**:

进程检查是否收到信号的时机是：一个进程在即将从内核态返回到用户态时；或者，在一个进程要进入或离开一个适当的低调度优先级睡眠状态时。

内核处理一个进程收到的信号的时机是在一个进程从内核态返回用户态时,或者，在一个进程要进入或离开一个适当的低调度优先级睡眠状态时。

所以，当一个进程在内核态下运行时，软中断信号并不立即起作用，要等到将返回用户态时才处理。进程只有处理完信号才会返回用户态，进程在用户态下不会有未处理完的信号。

内核处理一个进程收到的软中断信号是在该进程的上下文中，因此，进程必须处于运行状态。

处理信号有三种类型：进程接收到信号后退出；进程忽略该信号；进程收到信号后执行用户设定用系统调用signal的函数。

当进程接收到一个它忽略的信号时，进程丢弃该信号，就象没有收到该信号似的继续运行。

如果进程收到一个要捕捉的信号，那么进程从内核态返回用户态时执行用户定义的函数。

执行用户定义的函数的方法很巧妙，内核是在用户栈上创建一个新的层，该层中将返回地址的值设置成用户定义的处理函数的地址，这样进程从内核返回弹出栈顶时就返回到用户定义的函数处，从函数返回再弹出栈顶时，才返回原先进入内核的地方。

这样做的原因是用户定义的处理函数不能且不允许在内核态下执行（如果用户定义的函数在内核态下运行的话，用户就可以获得任何权限）。

在信号的处理方法中有几点特别要引起注意:

第一，在一些系统中，当一个进程处理完中断信号返回用户态之前，内核清除用户区中设定的对该信号的处理例程的地址，即下一次进程对该信号的处理方法又改为默认值，除非在下一次信号到来之前再次使用signal系统调用。这可能会使得进程在调用signal之前又得到该信号而导致退出。在BSD中，内核不再清除该地址。但不清除该地址可能使得进程因为过多过快的得到某个信号而导致堆栈溢 出。为了避免出现上述情况。在BSD系统中，内核模拟了对硬件中断的处理方法，即在处理某个中断时，阻止接收新的该类中断。

第二，如果要捕捉的信号发生于进程正在一个系统调用中时，并且该进程睡眠在可中断的优先级上，这时该信号引起进程作一次longjmp，跳出睡眠 状态，返回用户态并执行信号处理例程。

当从信号处理例程返回时，进程就象从系统调用返回一样，但返回了一个错误代码，指出该次系统调用曾经被中断。这要注意的是，BSD系统中内核可以自动地重新开始系统调用。

第三，若进程睡眠在可中断的优先级上，则当它收到一个要忽略的信号时，该进程被唤醒，但不做longjmp，一般是继续睡眠。但用户感觉不到进程曾经被唤醒，而是象没有发生过该信号一样。

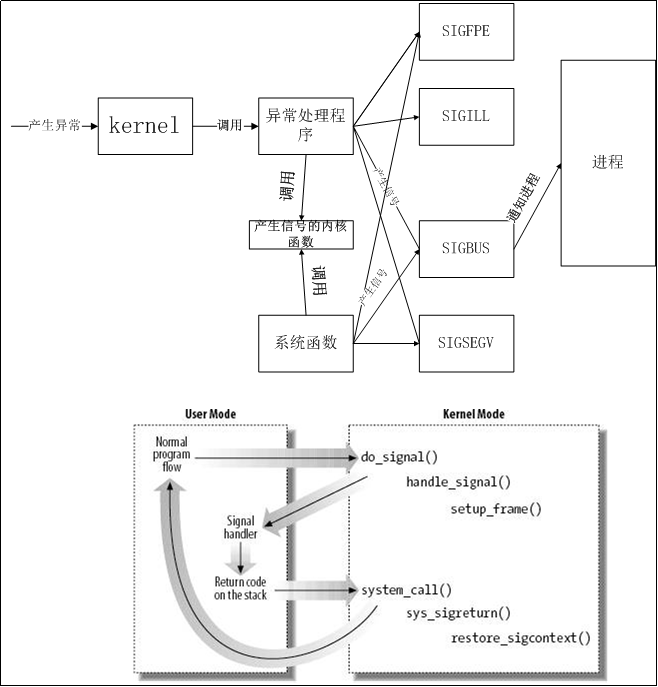
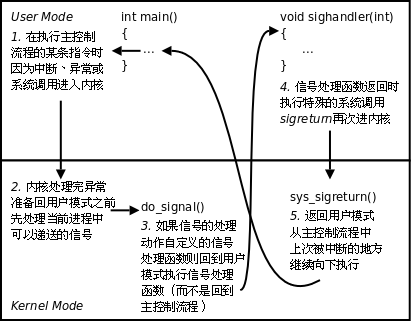
第四，内核对子进程终止（SIGCLD）信号的处理方法与其他信号有所区别。

当进程检查出收到了一个子进程终止的信号时，缺省情况下，该进程就象没有收到该信号似的，如果父进程执行了系统调用wait，进程将从系统调用wait中醒来并返回wait调用，执行一系列wait调用的后续操作（找出僵死的子进程，释放子进程的进程表项），然后从wait中返回。SIGCLD信号的作用是唤醒一个睡眠在可被中断优先级上的进程。如果该进程捕捉了这个信号，就象普通信号处理一样转到处理例程。如果进程忽略该信号，那么系统调用wait的动作就有所不同，因为SIGCLD的作用仅仅是唤醒一个睡眠在可被 中断优先级上的进程，那么执行wait调用的父进程被唤醒继续执行wait调用的后续操作，然后等待其他的子进程。

如果一个进程调用signal系统调用，并设置了SIGCLD的处理方法，并且该进程有子进程处于僵死状态，则内核将向该进程发一个SIGCLD信号。

setjmp和longjmp的作用

在介绍信号的时候，我们看到多个地方要求进程在检查收到信号后，从原来的系统调用中直接返回，而不是等到该调用完成。这种进程突然改变其上下文的情况，就是 使用setjmp和longjmp的结果。setjmp将保存的上下文存入用户区，并继续在旧的上下文中执行。这就是说，进程执行一个系统调用，当因为资 源或其他原因要去睡眠时，内核为进程作了一次setjmp，如果在睡眠中被信号唤醒，进程不能再进入睡眠时，内核为进程调用longjmp，该操作是内核 为进程将原先setjmp调用保存在进程用户区的上下文恢复成现在的上下文，这样就使得进程可以恢复等待资源前的状态，而且内核为setjmp返回1，使得进程知道该次系统调用失败。这就是它们的作用。



信号分成两种：   
regular signal( 非实时信号 ), 对应的编码值为 [1,31]  
real time signal 对应的编码值为 [32,64]

编码为 0 的信号 不是有效信号，只用于检查是当前进程否有发送信号的 权限 ，并不真正发送。

线程会有自己的悬挂信号队列 , 并且线程组也有一个信号悬挂队列 .

信号悬挂队列保存 task 实例接收到的信号 , 只有当该信号被处理后它才会从悬挂队列中卸下 .

信号悬挂队列还有一个对应的阻塞信号集合 , 当一个信号在阻塞信号集合中时 ,task 不会处理该被阻塞的信号 (但是该信号依旧在悬挂队列中 ). 当阻塞取消时 , 它会被处理 .

对一个信号 , 要三种处理方式 :

忽略该信号 ;

采用默认方式处理 ( 调用系统指定的信号处理函数 );

使用用户指定的方式处理 ( 调用用户指定的信号处理函数 ).

对于某些信号只能采用默认的方式处理 (eg:SIGKILL,SIGSTOP).

信号处理可以分成两个阶段 : 信号产生并通知到接收方 (generation), 接收方进行处理 (deliver)

.........

**简介**

Unix 为了允许用户态进程之间的通信而引入signal. 此外, 内核使用signal 给进程通知系统事件.近30 年来, signal 只有很小的变化 .

以下我们先介绍linux kernel 如何处理signal, 然后讨论允许进程间 exchange 信号的系统调用.

**The Role of Signals**

signal 是一种可以发送给一个进程或一组进程的短消息( 或者说是信号 , 但是这么容易和信号量混淆). 这种消息通常只是一个整数 , 而不包含额外的参数 .

linux 提供了很多种signal, 这些signal 通过宏来标识( 这个宏作为这个信号的名字). 并且这些宏的名字的开头是SIG.eg: 宏SIGCHLD , 它对应的整数值为17, 用来表示子进程结束时给父进程发送的消息 ( 即当子进程结束时应该向父进程发送标识符为17 的signal/ 消息/ 信号) . 宏SIGSEGV, 它对应的整数值为11, 当进程引用一个无效的物理地址时( 内核) 会向进程发送标识符为11 的signal/ 消息/ 信号 ( 参考linux 内存管理的页错误异常处理程序, 以及linux 中断处理).

信号有两个目的:  
1. 使一个进程意识到一个特殊事件发生了( 不同的事件用不同的signal 标识)   
2. 并使目标进程进行相应处理(eg: 执行的信号处理函数 , signal handler). 相应的处理也可以是忽略它 .

当然 , 这两个目的不是互斥的 , 因为通常一个进程意识到一个事件发生后就会执行该事件相应的处理函数 .

下表是linux2.6 在80x86 上的前31 个signals 及其相关说明 . 这些信号中有些是体系结构相关的(eg:SIGCHLD,SIGSTOP), 有些则专门了某些体系结构才存在的(eg:SIGSTKFLT) ( 可以参考中断处理 , 里面也列出了一些异常对应的signal).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The first 31 signals in Linux/i386 | | | | |
| **#** | **Signal name** | **Default action** | **Comment** | **POSIX** |
| 1 | SIGHUP | Terminate | Hang up controlling terminal or process | Yes |
| 2 | SIGINT | Terminate | Interrupt from keyboard | Yes |
| 3 | SIGQUIT | Dump | Quit from keyboard | Yes |
| 4 | SIGILL | Dump | Illegal instruction | Yes |
| 5 | SIGTRAP | Dump | Breakpoint for debugging | No |
| 6 | SIGABRT | Dump | Abnormal termination | Yes |
| 6 | SIGIOT | Dump | Equivalent to SIGABRT | No |
| 7 | SIGBUS | Dump | Bus error | No |
| 8 | SIGFPE | Dump | Floating-point exception | Yes |
| 9 | SIGKILL | Terminate | Forced-process termination | Yes |
| 10 | SIGUSR1 | Terminate | Available to processes | Yes |
| 11 | SIGSEGV | Dump | Invalid memory reference | Yes |
| 12 | SIGUSR2 | Terminate | Available to processes | Yes |
| 13 | SIGPIPE | Terminate | Write to pipe with no readers | Yes |
| 14 | SIGALRM | Terminate | Real-timerclock | Yes |
| 15 | SIGTERM | Terminate | Process termination | Yes |
| 16 | SIGSTKFLT | Terminate | Coprocessor stack error | No |
| 17 | SIGCHLD | Ignore | Child process stopped or terminated, or got signal if traced | Yes |
| 18 | SIGCONT | Continue | Resume execution, if stopped | Yes |
| 19 | SIGSTOP | Stop | Stop process execution | Yes |
| 20 | SIGTSTP | Stop | Stop process issued from tty | Yes |
| 21 | SIGTTIN | Stop | Background process requires input | Yes |
| 22 | SIGTTOU | Stop | Background process requires output | Yes |
| 23 | SIGURG | Ignore | Urgent condition on socket | No |
| 24 | SIGXCPU | Dump | CPU time limit exceeded | No |
| 25 | SIGXFSZ | Dump | File size limit exceeded | No |
| 26 | SIGVTALRM | Terminate | Virtual timer clock | No |
| 27 | SIGPROF | Terminate | Profile timer clock | No |
| 28 | SIGWINCH | Ignore | Window resizing | No |
| 29 | SIGIO | Terminate | I/O now possible | No |
| 29 | SIGPOLL | Terminate | Equivalent to SIGIO | No |
| 30 | SIGPWR | Terminate | Power supply failure | No |
| 31 | SIGSYS | Dump | Bad system call | No |
| 31 | SIGUNUSED | Dump | Equivalent to SIGSYS | No |

上述signal 称为**regular signal**. 除此之外, POSIX 还引入了另外一类singal 即**real-time signal**. real time signal 的标识符的值从32 到64. 它们与reagular signal 的区别在于每一次发送的real time signal 都会被加入悬挂信号队列，所以多次发送的real time signal 会被缓存起来( 而不会导致后面的被忽略掉) . 而同一种( 即标识符一样) regular signal 不会被缓存, 即如果同一个signal 被发送多次 , 它们只有一个会被放入接受进程的悬挂队列 .

虽然linux kernel 并没有使用real time signal. 但是它也( 通过特殊的系统调用) 支持posix定义的real time signal.

有很多系统调用可以给进程发送singal, 也有很多系统调可以指定进程在接收某一个signal 时应该如何响应( 即实行哪一个函数). 下表给出了这类系统调用: ( 关于这些系统调用的更多信息参考下文)

|  |  |
| --- | --- |
| **System call** | **Description** |
| kill( ) | Send a signal to a thread group |
| tkill( ) | Send a signal to a process |
| tgkill( ) | Send a signal to a process in a specific thread group |
| sigaction( ) | Change the action associated with a signal |
| signal( ) | Similar to sigaction( ) |
| sigpending( ) | Check whether there are pending signals |
| sigprocmask( ) | Modify the set of blocked signals |
| sigsuspend( ) | Wait for a signal |
| rt\_sigaction( ) | Change the action associated with a real-time signal |
| rt\_sigpending( ) | Check whether there are pending real-time signals |
| rt\_sigprocmask( ) | Modify the set of blocked real-time signals |
| rt\_sigqueueinfo( ) | Send a real-time signal to a thread group |
| rt\_sigsuspend( ) | Wait for a real-time signal |
| rt\_sigtimedwait( ) | Similar to rt\_sigsuspend( ) |

signal 可能在任意时候被发送给一个状态未知的进程 . 当信号被发送给一个当前并不正在执行的进程时, 内核必须把先把该信号保存直到该进程恢复执行. (to do ???????)

被阻塞的信号尽管会被加入进程的悬挂信号队列 , 但是在其被解除阻塞之前不会被处理(deliver),Blocking a signal (described later) requires that delivery of the signal be held off until it is later unblocked, which acer s the problem of signals being raised before they can be delivered.

内核把信号传送分成两个阶段:   
signal generation: 内核更新信号的目的进程的相关数据结构 , 这样该进程就能知道它接收到了一个信号. 觉得称为收到信号阶段更恰当. 这个generation 翻译成目的进程接收也不错 .

signal delivery(): 内核强制目的进程处理接收到的信号，这主要是通过修改进程的执行状态或者在目的进程中执行信号处理函数来实现的 . 觉得称为处理收到的信号阶段更恰当 . diliver 这里翻译成处理更恰当 .

deliver 的翻译: 有很多个 , 估计翻译成in computing 比较合理

一个genearated signal 最多只能deliver 一次( 即一个信号最多只会被处理一次) . signal 是可消耗资源 , 一旦一个signal 被deliver, 那么所有进程对它的引用都会被取消 .

已经产生但是还未被处理(deliver) 的信号称为**pending signal**( 悬挂信号). 对于regular signal, 在某一个时刻 , 一种signal 在一个进程中只能有一个实例( 因为进程没有用队列缓存其收到的signal) . 因为有31 种regualar signal , 所以一个进程某一个时刻可以有31 个各类signal 的实例. 此外因为linux 进程对real time signal 采用不同的处理方式, 它会保存接收到的real time signal 的实例 , 所以可以同时有很多同种signal 的实例 .

问题: 不同种类的信号的优先级( 从值较小的开始处理) .

一般而言 , 一个信号可能会被悬挂很长是时间( 即一个进程收到一个信号后 , 该信号有可能在该进程里很久 , 因为进程没空来处理它), 主要有如下因素:

1. 信号通常被当前进程处理 . Signals are usually delivered only to the currently running process (that is, to the current process).

2. 某种类型的信号可能被本进程阻塞. 只有当其被取消阻塞好才会被处理 .

3. 当一个进程执行某一种信号的处理函数时 , 一般会自动阻塞这种信号 , 等处理完毕后才会取消阻塞 . 这意味着一个信号处理函数不会被同种信号阻塞 .

尽管信号在概念上很直观 , 但是内核的实现却相当复杂. 内核必须:

1. 记录一个进程阻塞了哪些信号

2. 当从核心态切换到用户态时 , 检查进程是否接受到了signal.( 几乎每一次时钟中断都要干这样的事 , 费时吗?).

3. 检查信号是否可以被忽略. 当如下条件均满足时则可被忽略:

   1). 目标进程未被其它进程traced( 即PT\_PTRACED==0). 但一个被traced 的进程收到一个信号时 , 内核停止目标线程 , 并且给tracing 进程发送信号SIGCHLD. tracing 进程可能会通过SIGCONT来恢复traced 进程的执行

   2). 目标进程未阻塞该信号 .

   3). 信号正被目标进程忽略( 或者由于忽略是显式指定的或者由于忽略是默认操作).

4. 处理信号 . 这可能需要切换到信号处理函数

此外, linux 还需要处理BSD, System V 中signal 语义的差异性 . 另外 , 还需要遵守POSIX 的定义 .

**处理信号的方式 (Actions Performed upon Delivering a Signal)**

一个进程可以采用三中方式来响应它接收到的信号:

1.(ignore) 显示忽略该信号

2.(default) 调用默认的函数来响应该信号( 这些默认的函数由内核定义) , 一般这些默认的函数都分成如下几种( 采用哪一种取决于信号的类型 , 参考前面的表格):   
Terminate: The process is terminated (killed)   
Dump: The process is terminated (killed) and a core file containing its execution context is created, if possible; this file may be used for debug purposes.   
Ignore:The signal is ignored.   
Stop:The process is stopped, i.e., put in the TASK\_STOPPED state.   
Continue:If the process was stopped (TASK\_STOPPED), it is put into the TASK\_RUNNING state.

3.(catch) 调用相应的信号处理函数 ( 这个信号处理函数通常是程序员在运行时指定的). 这意味着进程需要在执行时显式地指明它需要catch 哪一种信号. 并且指明其处理函数 . catch 是一种主动处理的措施 .

注意上述的三个处理方式被标识为:ignore, default, catch. 这三个处理方式以后会通过这三个标识符引用 .

注意阻塞一个信号和忽略一个信号是不同 , 一个信号被阻塞是就当前不会被处理 , 即一个信号只有在解除阻塞后才会被处理 . 忽略一个信号是指采用忽略的方式来处理该信号( 即对该信号的处理方式就是什么也不做) .

SIGKILL 和SIGSTOP 这两个信号不能忽略 , 不能阻塞 , 不能使用用户定义的函数(caught) . 所以总是执行它们的默认行为 . 所以 , 它们允许具有恰当特权级的用户杀死别的进程, 而不必在意被杀进程的防护措施 ( 这样就允许高特权级用户杀死低特权级的用户占用大量cpu 的时间) .

注: 有两个特殊情况. 第一 , 任意进程都不能给进程0( 即swapper 进程) 发信号 . 第二 , 发给进程1 的信号都会被丢弃(discarded), 除非它们被catch. 所以进程 0 不会死亡, 进程1 仅在int 程序结束时死亡 .

一个信号对一个进程而言是**致命的(fatal)**, 当前仅当该信号导致内核杀死该进程 . 所以,SIGKILL总是致命的. 此外 , 如果一个进程对一个信号的默认行为是terminate 并且该进程没有catch 该信号 , 那么该信号对这个进程而言也是致命的 . 注意 , 在catch 情况下 , 如果一个进程的信号处理函数自己杀死了该进程 , 那么该信号对这个进程而言不是致命的 , 因为不是内核杀死该进程而是进程的信号处理函数自己杀死了该进程.

**POSIX 信号以及多线程程序**

POSIX 1003.1 标准对多线程程序的信号处理有更加严格的要求:   
( 由于linux 采用轻量级进程来实现线程 , 所以对linux 的实现也会有影响)

1. 多线程程序的所有线程应该共享信号处理函数 , 但是每一个线程必须有自己的mask of pending and blocked signals

2. POSIX 接口kill( ), sigqueue( ) 必须把信号发给线程组 , 而不是指定线程. 另外内核产生的SIGCHLD, SIGINT, or SIGQUIT 也必须发给线程组 .

3. 线程组中只有有一个线程来处理(deliver) 的共享的信号就可以了 . 下问介绍如何选择这个线程 .

4. 如果线程组收到一个致命的信号 , 内核要杀死线程组的所有线程, 而不是仅仅处理该信号的线程 .

为了遵从POSIX 标准, linux2.6 使用轻量级进程实现线程组.

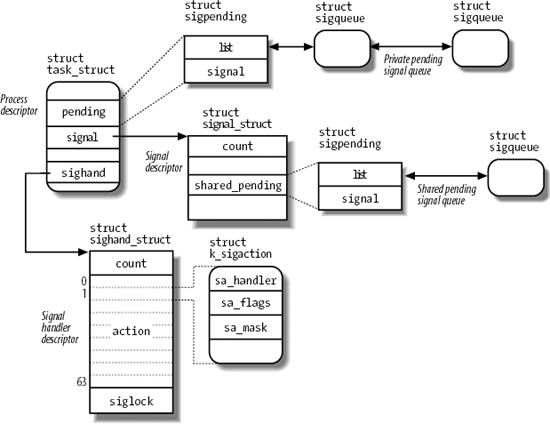
下文中 , 线程组表示OS 概念中的进程, 而线程表示linux 的轻量级进程. 进程也( 更多地时候)表示linux 的轻量级进程 . 另外每一个线程有一个私有的悬挂信号列表 , 线程组共享一个悬挂信号列表 .

**与信号有关的数据结构**

注:pending/ 悬挂信号, 表示进程收到信号 , 但是还没有来得及处理 , 或者正在处理但是还没有处理完成 .

对于每一个进程, 内核必须知道它当前悬挂(pending) 着哪些信号或者屏蔽(mask) 着哪些信号 .还要知道线程组如何处理信号. 为此内核使用了几个重要的数据结构( 它们可通过task 实例访问), 如下图:

**The most significant data structures related to signal handling**



( 注意task 中的一些关于signal 的成员在上图中没有表现出来)

task 中关于signal 的成员列在下表中:

|  |  |  |
| --- | --- | --- |
| **Process descriptor fields related to signal handling** | | |
| **Type** | **Name** | **Description** |
| struct signal\_struct \* | **signal** | Pointer to the process's signal descriptor( **线程组共用**的信号) |
| struct sighand\_struct \* | **sighand** | Pointer to the process's signal handler descriptor(**线程组共用**) |
| sigset\_t | blocked | Mask of blocked signals( 线程私有) |
| sigset\_t | real\_blocked | Temporary mask of blocked signals (used by thert\_sigtimedwait( ) system call) ( 线程私有) |
| struct**sigpending** | pending | Data structure storing the **private pending signal** s |
| unsigned long | sas\_ss\_sp | Address of alternative signal handler stack.( 可以不提供) |
| size\_t | sas\_ss\_size | Size of alternative signal handler stack( 可以不提供) |
| int (\*) (void \*) | Notifier | Pointer to a function used by a device driver to block some signals of the process |
| void \* | notifier\_data | Pointer to data that might be used by the notifier function (previous field of table) |
| sigset\_t \* | notifier\_mask | Bit mask of signals blocked by a device driver through a notifier function |

**blocked 成员**保存进程masked out 的signal . 其类型为**sigset\_t**, 定义如下:

    typedef struct {

        unsigned long sig[2];

    } sigset\_t;

sizeof(long)==32, sigset\_t 被当成了bit array 使用. 正如前文提到的,linux 有64 种信号 ,[1,31] 为regular signal, [32,64] 为real time signal. 每一种对应sigset\_t 中一个bit.

**信号描述符& 信号处理函数描述符**

task 的signal, sighand 成员分别是信号描述符与信号处理函数描述符 .

**signal 成员**是一个指针 , 它指向结构体signal\_struct 的实例 , 该实例保存了线程组悬挂着的信号 . 也就是说线程组中的所有进程( 这里称为task 更合理) 共用同一个signal\_struct 实例. signal\_struct 中的shared\_pending 成员保存了所有悬挂的信号( 以双向链表组织) . 此外signal\_struct 中还保存了许多其它的信息(eg: 进程资源限制信息, pgrp, session 信息) .

下表列出了signal\_struct 中与信号处理有关的成员:

|  |  |  |
| --- | --- | --- |
| **The fields of the signal descriptor related to signal handling** | | |
| **Type** | **Name** | **Description** |
| atomic\_t | count | Usage counter of the signal descriptor |
| atomic\_t | live | Number of live processes in the thread group |
| wait\_queue\_head\_t | wait\_chldexit | Wait queue for the processes sleeping in await4( ) system call |
| struct task\_struct \* | curr\_target | Descriptor of the last process in the thread group that received a signal |
| struct**sigpending** | **shared\_pending** | Data structure storing the shared pending signals |
| int | group\_exit\_code | Process termination code for the thread group |
| struct task\_struct \* | group\_exit\_task | Used when killing a whole thread group |
| int | notify\_count | Used when killing a whole thread group |
| int | group\_stop\_count | Used when stopping a whole thread group |
| unsigned int | flags | Flags used when delivering signals that modify the status of the process |

除了signal 成员外 , 还有一个**sighand 成员** 用来指明相应的信号处理函数.

sighand 成员是一个指针 , 指向一个sighand\_struct 变量 , 该变量为线程组共享 . 它描述了一个信号对应的信号处理函数.

sighand\_struct 成员如下:

|  |  |  |
| --- | --- | --- |
| **The fields of the signal handler descriptor** | | |
| **Type** | **Name** | **Description** |
| atomic\_t | count | Usage counter of the signal handler descriptor |
| struct k\_sigaction [64] | action | Array of structures specifying the actions to be performed upon delivering the signals |
| spinlock\_t | siglock | Spin lock protecting both the signal descriptor and the signal handler descriptor |

sighand\_struct 中的重要成员是action, 它是一个数组 , 描述了每一种信号对应的信号处理函数 .

**sigaction 数据结构**

某一些平台上, 会赋予一个signal 一些只能内核才可见的属性. 这些属性与sigaction( 它在用户态也可见) 构成了结构体k\_sigaction. 在x86 上,k\_sigaction 就是sigaction.

注: 用户使用的sigaction 和内核使用的sigaction 结构体有些不同但是 , 它们存储了相同的信息( 自己参考一下用户态使用的sigaction 结构体吧).

内核的sigaction 的结构体的成员如下:

1)sa\_handler: 类型为 void (\*)(int):

    这个字段指示如何处理信号 . 它可以是指向处理函数的指针 , 也可以是SIG\_DFL(==0) 表示使用默认的处理函数 , 还可以是SIG\_IGN(==1) 表示忽略该信号

2)sa\_flags: 类型为unsigned long:

   指定信号如何被处理的标志 , 参考下表 ( 指定信号如何处理的标志) .

3)sa\_mask: 类型为sigset\_t:

   指定当该信号处理函数执行时,sa\_mask 中指定的信号必须屏蔽 .

**指定信号如何处理的标志**

注: 由于历史的原因 , 这些标志的前缀为SA\_, 这和irqaction 的flag 类似 , 但其实它们没有关系.

|  |  |
| --- | --- |
| **Flags specifying how to handle a signal** | |
| **Flag Name** | **Description** |
| SA\_NOCLDSTOP | Applies only to SIGCHLD ; do not send SIGCHLD to the parent when the process is stopped |
| SA\_NOCLDWAIT | Applies only to SIGCHLD ; do not create a zombie when the process terminates |
| SA\_SIGINFO | Provide additional information to the signal handler |
| SA\_ONSTACK | Use an alternative stack for the signal handler |
| SA\_RESTART | Interrupted system calls are automatically restarted |
| SA\_NODEFER, SA\_NOMASK | Do not mask the signal while executing the signal handler |
| SA\_RESETHAND,  SA\_ONESHOT | Reset to default action after executing the signal handler |

**悬挂的信号队列 (sigpending)**

通过前文我们知道有些系统调用能够给线程组发信号(eg:kill, rt\_sigqueueinfo), 有些操作给指定的进程发信号(eg:tkill, tgkill) .

为了区分这两类, task 中其实有两种悬挂信号列表:   
1.task 的 pending 字段表示了本task 上私有的悬挂信号( 列表)   
2.task 的signal 字段中的shared\_pending 字段则保存了线程组共享的悬挂信号( 列表).

悬挂信号 列表用数据结构sigpending 表示 , 其定义如下:   
     struct **sigpending** {   
        struct list\_head list;   
        sigset\_t signal;   
    }

其signal 成员指明当前悬挂队列悬挂了哪些信号 .

其list 字段其实是一个双向链表的头 , 链表的元素的类型是sigqueue. sigqueue 的成员如下:

|  |  |  |
| --- | --- | --- |
| **The fields of the sigqueue data structure** | | |
| **Type** | **Name** | **Description** |
| struct list\_head | list | Links for the pending signal queue's list |
| spinlock\_t \* | lock | Pointer to the siglock field in the signal handler descriptor corresponding to the pending signal |
| Int | flags | Flags of the sigqueue data structure |
| siginfo\_t | info | Describes the event that raised the signal |
| struct  user\_struct \* | user | Pointer to the per-user data structure of the process's owner |

( 注:sigqueue 的名字有queue, 但它其实只是悬挂队列的一个元素 . 它会记录一个被悬挂的信号的信息)

**siginfo\_t**是一个包含128 byte 的数据结构 , 用来描述一个指定信号的发生，其成员如下:   
si\_signo: 信号ID

si\_errno: 导致这个信号被发出的错误码. 0 表示不是因为错误才发出信号的 .

si\_code: 标识谁发出了这个信号 . 参考下表 :

|  |  |
| --- | --- |
| **The most significant signal sender codes** | |
| **Code Name** | **Sender** |
| SI\_USER | kill( ) and raise( ) |
| SI\_KERNEL | Generic kernel function |
| SI\_QUEUE | sigqueue( ) |
| SI\_TIMER | Timer expiration |
| SI\_ASYNCIO | Asynchronous I/O completion |
| SI\_TKILL | tkill() and tgkill() |

\_sifields: 这个字段是一个union, 它有不少 成员 , 哪一个成员有效取决于信号 . 比如对于SIGKILL, 则它会记录信号发送者的PID,UID; 对于SIGSEGV, 它会存储导致访问出错的内存地址 .

**操作信号数据结构的函数**

一些宏和函数会使用信号数据结构 . 在下文的解说中, set 表示指向sigset\_t 变量的指针, nsig表示信号的标识符( 信号的整数值).mask 是一个unsign long bit mask.

**sigemptyset**(set) and **sigfillset** (set)

把set 所有bit 设置为 0 或者1 .

**sigaddset**(set,nsig) and **sigdelset** (set,nsig)

把set 中对应与nsig 的bit 设置为1 或者 0. In practice, sigaddset( ) reduces to:   
    set->sig[(nsig - 1) / 32] |= 1UL << ((nsig - 1) % 32);

and sigdelset( ) to:   
    set->sig[(nsig - 1) / 32] &= ~(1UL << ((nsig - 1) % 32));

**sigaddsetmask**(set,mask) and **sigdelsetmask** (set,mask)

根据mask 的值设置set. 仅能设置1-32 个signal. The corresponding functions reduce to:      
      set->sig[0] |= mask;

and to:   
set->sig[0] &= ~mask;

**sigismember**(set,nsig)

返回set 中对应nsig 的bit 的值. In practice, this function reduces to:

    return 1 & (set->sig[(nsig-1) / 32] >> ((nsig-1) % 32));

**sigmask**(nsig)

根据信号标志码nsig 等到它的在sigset\_t 中的bit 位的index.

**sigandsets**(d,s1,s2), **sigorsets** (d,s1,s2), and **signandsets** (d,s1,s2)

      伪代码如下:d=s1 & s2; d=s1|s2, d=s1 & (~s2)

**sigtestsetmask**(set,mask)

如果mask 中的为1 的位在set 中的相应位也为1, 那么返回1. 否则返回0. 只适用于1-32个信号.

**siginitset**(set,mask)

用mask 设置set 的1-32 个信号, 并把set 的33-63 个信号清空.

**siginitsetinv**(set,mask)

用(!mask) 设置set 的1-32 个信号, 并把set 的33-63 个信号设置为1.

**signal\_pending**(p)

检查p 的 t->thread\_info->flags 是否为 TIF\_SIGPENDING. 即检查p 是否有 悬挂的非阻塞信号.

**recalc\_sigpending\_tsk**(t) and **recalc\_sigpending** ( )

第一个函数检查 t->pending->signal 或者 t->signal->shared\_pending->signal 上是否有悬挂的非阻塞信号. 若有设置 t->thread\_info->flags 为 TIF\_SIGPENDING.

recalc\_sigpending( ) 等价于 recalc\_sigpending\_tsk(current) .

**rm\_from\_queue**(mask,q)

清掉悬挂信号队列q 中的由mask 指定的信号.

**flush\_sigqueue**(q)

清掉悬挂信号队列q 中的信号.

**flush\_signals**(t)

删除t 收到的所有信号. 它会清掉 t->thread\_info->flags 中的TIF\_SIGPENDING 标志, 并且调用flush\_sigqueue 把t->pending 和 t->signal->shared\_pending 清掉 .

**Generating a Signal**

很多内核函数会产生signal, 它完成处理处理的第一个阶段(generate a signal) , 即更新信号的目标进程的相应字段 . 但是它们并不直接完成信号处理的第二阶段(deliver the signal), 但是它们会根据目标进程的状态或者唤醒目标进程或者强制目标进程receive the signal .

注:generating a signal 这个阶段是从源进程发起一个信号 , 然后源进程在内核态下修改目标进程的相应状态, 然后可能源进程还会唤醒目的进程 .

无论一个信号从内核还是从另外一个进程被发送给另一个线程( 目标进程) , 内核都会执行如下的函数之一来发送信号:

|  |  |
| --- | --- |
| **Kernel functions that generate a signal for a process** | |
| **Name** | **Description** |
| send\_sig( ) | Sends a signal to a single process |
| send\_sig\_info( ) | Like send\_sig( ) , with extended information in a siginfo\_tstructure |
| force\_sig( ) | Sends a signal that cannot be explicitly ignored or blocked by the process |
| force\_sig\_info( ) | Like force\_sig( ) , with extended information in a siginfo\_tstructure |
| force\_sig\_specific( ) | Like force\_sig( ) , but optimized for SIGSTOP and SIGKILL signals |
| sys\_tkill( ) | System call handler of tkill( ) |
| sys\_tgkill( ) | System call handler of tgkill( ) |

所有这些函数最终都会调用 **specific\_send\_sig\_info**( ) .

无论一个信号从内核还是从另外一个进程被发送给另一个线程组( 目标进程), 内核都会执行如下的函数之一来发送信号:

|  |  |
| --- | --- |
| **Kernel functions that generate a signal for a thread group** | |
| **Name** | **Description** |
| send\_group\_sig\_info( ) | Sends a signal to a single thread group identified by the process descriptor of one of its members |
| kill\_pg( ) | Sends a signal to all thread groups in a process group |
| kill\_pg\_info( ) | Like kill\_pg( ) , with extended information in a siginfo\_tstructure |
| kill\_proc( ) | Sends a signal to a single thread group identified by the PID of one of its members |
| kill\_proc\_info( ) | Like kill\_proc( ) , with extended information in a siginfo\_tstructure |
| sys\_kill( ) | System call handler of kill( ) |
| sys\_rt\_sigqueueinfo( ) | System call handler of rt\_sigqueueinfo( ) |

这些函数最终都调用 **group\_send\_sig\_info**( ) .

**specific\_send\_sig\_info 函数说明**

这个函数给指定的目标线程( 目标进程) 发送一个信号 . 它有三个参数:

参数sig: 信号( 即某一个信号) .

参数info: 或者是 siginfo\_t 变量地址或者如下三个特殊值:   
0 : 表示信号由用户态进程发送;   
1 : 表示信号由核心态( 进程) 发送;  
2 : 表示信号由核心态( 进程) 发送, 并且信号是SIGKILL 或者SIGSTOP.

参数t: 目标进程的task 实例指针

specific\_send\_sig\_info 调用时必须禁止本cpu 的中断 , 并且获得t->sighand->siglock spin lock. 它会执行如下操作:

1. 检查目标线程是否忽略该信号, 若是返回0. 当如下三个条件均满足时则可认为忽略该信号:  
   1). 目标线程未被traced( 即t->ptrace 不含PT\_PTRACED 标志).  
   2). 该信号未被目标线程阻塞( 即sigismember(&t->blocked, sig) == 0).  
   3). 该信号被目标线程显式地忽略( 即t->sighand->action[sig-1].sa\_handler == SIG\_IGN)或者隐式忽略( 即handler==SIG\_DFT 并且信号为SIGCONT, SIGCHLD, SIGWINCH, or SIGURG.).

2. 检查信号是否是非实时信号(sig<32) 并且同样的信号是否已经在线程的私有悬挂信号队列中了, 若是则返回0.

3. 调用send\_signal(sig, info, t, &t->pending) 把信号加入目标线程的私有悬挂信号队列中.下文会详述.

4. 如果send\_signal 成功并且信号未被目标线程阻塞, 则调用**signal\_wake\_up**( ) 来通知目标进程有新的信号达到. 这个函数执行如下步骤:  
   1). 把标志TIF\_SIGPENDING 加到t->tHRead\_info->flags 中  
   2). 调用try\_to\_wake\_up(). 如果目标线程处于TASK\_INTERRUPTIBLE 或者TASK\_STOPPED 并且信号是SIGKILL 则唤醒目标线程.  
   3). 如果try\_to\_wake\_up 返回0, 则目标线程处于runnable 状态, 之后检查目标线程是否在别的CPU 上执行, 如果是则向该CPU 发送处理器中断以强制该cpu 重调度目标线程( 注: 目前我们并未考虑多处理器的情况). 因为每一个线程在从schedule() 返回时都会检查是否存在悬挂的信号,所以这个处理器中断将会使目标线程很快就看到这个新的悬挂信号.

5. 返回1( 表示信号已经成功generated.)

**send\_signal 函数**

这个函数接受四个参数:sig, info, t, signals. 其中sig, info,t 在**specific\_send\_sig\_info**中已经介绍过了. signals 则是t 的pending queue 的首地址 . 它的执行流程如:

1. 若info==2, 那么这个信号是SIGKILL 或是SIGSTOP, 并且由kernel 通过force\_sig\_specific产生. 此时直接跳到9. 因为这种情况下, 内核会立即执行信号处理, 所以不用把该信号加入信号悬挂队列中.

2. 如果目标进程的用户当前的悬挂信号数目(t->user->sigpending) 小于目标进程的最大悬挂信号数目(t->signal->rlim[RLIMIT\_SIGPENDING].rlim\_cur), 则为当前信号分配一个sigqueue 变量,标识为q

3. 如果目标进程的用户当前的悬挂信号数目太大, 或者上一步中分配sigqueue 变量失败, 则跳到9.

4. 增加目标进程的用户当前的悬挂信号数目(t->user->sigpending) 以及t-user 的引用数.

5. 把信号q 加入目标线程的悬挂队列:  
    list\_add\_tail(&q->list, &signals->list);

6. 填充q, 如下

    if ((unsigned long)info == 0) {  
        q->info.si\_signo = sig;  
        q->info.si\_errno = 0;  
        q->info.si\_code = SI\_USER;  
        q->info.\_sifields.\_kill.\_pid = current->pid;  
        q->info.\_sifields.\_kill.\_uid = current->uid;  
    } else if ((unsigned long)info == 1) {  
        q->info.si\_signo = sig;  
        q->info.si\_errno = 0;  
        q->info.si\_code = SI\_KERNEL;  
        q->info.\_sifields.\_kill.\_pid = 0;  
        q->info.\_sifields.\_kill.\_uid = 0;  
    } else  
        copy\_siginfo(&q->info, info);

函数copy\_siginfo 用caller 传进来的info 填充q->info

7. 设置悬挂信号队列中的mask 成员的与sig 相应的位( 以表示该信号在悬挂信号队列中)  
    sigaddset(&signals->signal, sig);

7. **返回0**以表示信号被成功加入悬挂信号队列.

9. 如果执行这一步, 则该信号不会被加入信号悬挂队列, 原因有如下三个:1) 有太多的悬挂信号了, 或者2) 没有空闲的空间来分配sigqueue 变量了, 或者3) 该信号的处理由内核立即执行. 如果信号是实时信号并且通过内核函数发送并且显式要求加入队列, 那么返回错误代码-EAGAIN( 代码类似如下):  
    if (sig>=32 && info && (unsigned long) info != 1 &&  
                   info->si\_code != SI\_USER)  
        return -EAGAIN;

10. 设置悬挂信号队列中的mask 成员的与sig 相应的位( 以表示该信号在悬挂信号队列中)  
    sigaddset(&signals->signal, sig);

11. 返回0. 尽管该信号没有放到悬挂信号队列中, 但是相应的signals->signal 中已经设置了

即使没有空间为信号分配sigqueue 变量，也应该让目标信号知道相应的信号已经发生, 这一点很重要. 考虑如下情形: 目标进程使用了很多内存以致于无法再分配sigqueue 变量了, 但是内核必须保证对目标进程依的kill 依然能够成功, 否则管理员就没有机会杀死目标进程了.

**group\_send\_sig\_info 函数**

函数 group\_send\_sig\_info 把一个信号发给一个线程组 . 这个函数有三个参数:sig, info, p . (和specific\_send\_sig\_info 类似).

这个函数的执行流程如下 :

1. 检查参数sig 的正确性:

  if (sig < 0 || sig > 64)

     return -EINVAL;

2. 如果信号的发送进程处于用户态, 则检查这个发送操作是否允许. 仅当满足如下条件之一( 才视为允许):

  1). 发送者进程有恰当的权限( 通常发送者进程应该是system administrator).

  2). 信号为SIGCONT, 并且目标进程和发送者进程在同一个login session.

  3). 目标进程和发送者进程属于同一个用户

3. 如果用户态的进程不能发送此信号, 则返回-EPERM. 如果sig==0, 则立即返回.( 因为0 是无效的信号). 如果sighand==0, 也立即返回, 因为此时目标进程正在被杀死, 从而sighand 被释放.

    if (!sig || !p->sighand)

        return 0;

4. 获得锁 p->sighand->siglock, 并且关闭本cpu 中断.

5. 调用handle\_stop\_signal 函数, 这个函数检查sig 是否会和现有的悬挂的信号冲突, 会的话解决冲突. 这个函数的步骤如下:

  1). 如果线程组正在被杀死(SIGNAL\_GROUP\_EXIT) ，则返回.

  2). 如果sig 是IGSTOP, SIGTSTP, SIGTTIN, SIGTTOU 中的一种, 则调用rm\_from\_queue, 把线程组中所有悬挂的SIGCONT 删除. 注意: 包含线程组共享的悬挂信号队列中的(p->signal->shared\_pending) 以及每一个线程私有悬挂队列中的.

  3). 如果sig 是SIGCONT, 则调用rm\_from\_queue, 把线程组中所有悬挂的SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU 删除. 注意: 包含线程组共享的悬挂信号队列中的(p->signal->shared\_pending) 以及每一个线程私有悬挂队列中的. 之后为每一个线程调用try\_to\_wake\_up.

6. 检查线程组是否忽略该信号, 如果忽略返回0.

7. 如果是非实时信号, 并且该线程组已经有这种悬挂的信号了, 那么返回0:

    if (sig<32 && sigismember(&p->signal->shared\_pending.signal,sig))

        return 0;

8. 调用send\_signal( ) 把信号加到线程组的共享悬挂信号队列中, 如果send\_signal 返回非0 值,则group\_send\_sig\_info 退出并把该非零值返回.

9. 调用\_ \_group\_complete\_signal( ) 来唤醒线程组中的一个轻量级进程. 参考下文.

10. 释放p->sighand->siglock 并且打开本地中断.

11. 返回 0 (success).

函数 **\_ \_group\_complete\_signal**( ) 扫描目标线程组 , 并且返回一个能够处理 (receive) 该新信号的进程 . 这样的进程必须同时具备如下的条件 :

1) 该进程不阻塞新信号.

2) 进程的状态不是EXIT\_ZOMBIE, EXIT\_DEAD, TASK\_TRACED, or TASK\_STOPPED. 但是当信号是SIGKILL 是, 进程的状态允许是TASK\_TRACED or TASK\_STOPPED.

3) 进程不处于正在被杀死的状态, 即状态不是PF\_EXITING.

4) 或者进程正在某一个cpu 上执行, 或者进程的TIF\_SIGPENDING 的标志未被设置.

一个线程组中满足上诉条件的线程( 进程) 可能很多, 根据如下原则选择一个:

1) 如果group\_send\_sig\_info 中的参数p 指定的进程满足上述条件, 则选择p.

2) 否则从最后一个接收线程组信号的线程(p->signal->curr\_target) 开始查找满足上述条件的线程, 找到为止.

( 如果线程组中没有一个线程满足上述条件怎么办?)

如\_ \_group\_complete\_signal( ) 成功找到一个进程( 表示为selected\_p), 那么:

1. 检查该信号是否是致命的, 若是, 通过给线程组中的每一个线程发送SIGKILL 来杀死线程组

2. 若不是, 调用signal\_wake\_up 来唤醒selected\_p 并告知它有新的悬挂信号,

**Delivering a Signal**

通过上面的介绍, 内核通过修改目标进程的状态, 告知目标进程有新的信号到达. 但是目标进程对到达的新信号的处理(deliver signal) 我们还没有介绍. 下面介绍目标进程如何在内核的帮助下处理达到的新信号.

注意当内核( 代码) 要把进程从核心态恢复成用户态时( 当进程从异常/ 中断处理返回时), 内核会检查该进程的 TIF\_SIGPENDING 标识 , 如果存在悬挂的信号 , 那么将先处理该信号 .

这里需要介绍一下背景: 当进程在用户态( 用U1 表示) 下由于中断/ 异常而进入核心态, 那么需要把U1 的上下文记录到该进程的内核堆栈中.

为了处理非阻塞的信号 , 内核调用**do\_signal**函数 . 这个函数接受两个参数:

regs: 指向U1 上下文在内核堆栈的首地址 ( 参考进程管理).

oldest:  保存了一个变量的地址, 该变量保存了被阻塞的信号的信息( 集合). 如果该参数为NULL, 那么这个地址就是&current->blocked ( 如下文). 注意当自定义信号处理函数结束后,会把oldest 设置为当前task 的阻塞信号集合.( 参考源代码, 以及rt\_frame 函数).

我们这里描述的do\_signal 流程将会关注信号delivery( 处理), 而忽略很多细节, eg: 竞争条件 ,产生core dump, 停止和杀死线程组等等 .

一般,do\_signal 一般仅在进程即将返回用户态时执行 . 因此 , 如果一个中断处理函数调用do\_signal, 那么do\_signal 只要按如下方式放回:  
    if ((regs->xcs & 3) != 3)  
        return 1;

如果oldest 为NULL, 那么 do\_signal 会把它设置为当前进程阻塞的信号:

    if (!oldset)  
        oldset = &current->blocked;

do\_signal 的核心是一个循环 , 该循环调用dequeue\_signal 从进程的私有悬挂信号队列和共享悬挂队列获取未被阻塞的信号. 如果成功获得这样的信号, 则通过handle\_signal 调用相应的信号处理函数, 否则退出do\_signal .

( 这个循环不是用C 的循环语句来实现, 而是通过修改核心栈的regs 来实现. 大概的流程可以认为如下: 当由核心态时切换向用户态时, 检查是否有非阻塞的悬挂信号, 有则处理( 包含: 准备信号处理函数的帧, 切换到用户态以执行信号处理函数, 信号处理函数返回又进入核心态), 无则返回原始的用户态上下文)

dequeue\_signal 先从私有悬挂信号列表中按照信号值从小到大取信号，取完后再从共享悬挂信号列表中取 . ( 注意取后要更新相应的信息)

接着我们考虑, do\_signal 如何处理获得的信号( 假设用signr 表示) .

首先 , 它会检查是否有别的进程在监控(monitoring) 本进程 , 如果有 , 调用do\_notify\_parent\_cldstop 和schedule 来让监控进程意识到本进程开始信号处理了.

接着,do\_signal 获得相应的信号处理描述符( 通过current->sig->action[signr-1]) , 从而获得信号处理方式的信息 . 总共有**三种处理方式: 忽略 , 默认处理 , 使用用户定义的处理函数**.

如果是**忽略 , 那么什么也不做**:

if (ka->sa.sa\_handler == SIG\_IGN)

        continue;

**执行默认的信号处理函数**

如果指定的是默认的处理方式. 那么do\_signal 使用默认的处理方式来处理信号 . ( 进程 0 不会涉及 , 参考前文)

对于init 进程除外 , 则它要丢弃信号:  
    if (current->pid == 1)  
        continue;

对于其它进程, 默认的处理方式取决于信号 .

第一类: 这类信号的默认处理方式就是不处理

    if (signr==SIGCONT || signr==SIGCHLD ||

            signr==SIGWINCH || signr==SIGURG)

        continue;//

第二类: 这类信号的默认处理方式如下:

    if (signr==SIGSTOP || signr==SIGTSTP ||

            signr==SIGTTIN || signr==SIGTTOU) {

        if (signr != SIGSTOP &&

               is\_orphaned\_pgrp(current->signal->pgrp))

            continue;

        do\_signal\_stop(signr);

    }

这里, SIGSTOP 与其他的信号有些微的区别.

SIGSTOP 停止整个线程组. 而其它信号只会停止不在孤儿进程组中的进程( 线程组).

孤儿进程组(orphand process group).

**非孤儿进程组**指如果进程组A 中有一个进程有父亲, 并且该父进程在另外一个进程组B 中, 并且这两个进程组A,B 都在用一个会话(session) 中, 那么进程组A 就是非孤儿进程组. 因此如果父进程死了, 但是启动在进程的session 依旧在, 那么进程组A 都不是孤儿.

注: 这两个概念让我迷糊.

**do\_signal\_stop**检查当前进程是否是线程组中的第一个正在被停止的进程, 如果是, 它就激活一个组停(group stop) 。本质上, 它会把信号描述符的 group\_stop\_count 字段设置为正值, 并且唤醒线程组中的每一个进程。每一个进程都会查看这个字段从而认识到正在停止整个线程组, 并把自己的状态改为 TASK\_STOPPED, 然后调用schedule. do\_signal\_stop 也会给线程组的父进程发送SIGCHLD, 除非父进程已经被设置为SA\_NOCLDSTOP flag of SIGCHLD.

默认行为是dump 的信号处理可能会进程工作目录下创建一个core 文件. 这个文件列出了进程的地址空间和cpu 寄存器的值. do\_signal 创建这个文件后, 就会杀死整个线程组. 剩下18 个信号的默认处理是terminate, 这仅仅是简单地杀死整个线程组. 为此,do\_signal 调用了do\_group\_exit。

**使用指定的函数来处理信号(catching the signal)**

如果程序为信号设置了处理函数 , 那么do\_signal 将会通过调用**handle\_signal**来强制该信号函数被执行:

    handle\_signal(signr, &info, &ka, oldset, regs);

    if (ka->sa.sa\_flags & SA\_ONESHOT)

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

return 1;

如果用户在为信号设置信号处理函数时指定了 SA\_ONESHOT , 那么当该信号处理函数第一次执行后 ,其将会被reset. 即以后来的这样的信号将会使用默认的处理函数 .

Notice how do\_signal( ) returns after having handled a single signal. Other pending signals won't be considered until the next invocation of do\_signal( ) . This approach ensures that real-time signals will be dealt with in the proper order.

执行一个信号处理函数相当复杂 , 因为需要内核小心处理用户信号处理函数的调用栈, 然后把控制权交给用户处理函数( 注意这里涉及内核态到用户态的转换) .

用户的信号处理函数定义在用户态中并且包含在用户代码段中，它需要在用户态(U2) 下执行. hande\_signal 函数在核心态下执行. 此外, 由于当前的核心态是在前一个用户态(U1) 转过来, 这意味着当信号处理函数(U2) 结束, 回到内核态, 然后内核态还需要回到U1, 而当从U2 进入核心态后, 内核栈存放的已经不再是U1 的上下文了( 而是U2), 此外一般信号处理函数中还会发生系统调用( 用户态到核心态的转换), 而系统调用结束后要回到信号处理函数. 

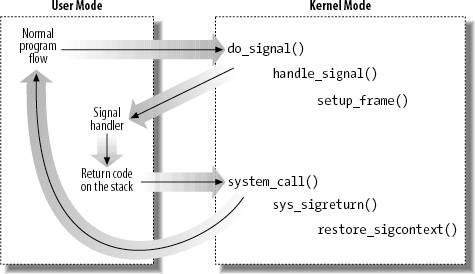
注意: 每一个内核态切换到用户态, 进程的内核堆栈都会被清空.

那么handle\_signal 如何调用信号处理函数呢??

Linux 采用的方法如下: 每次调用信号处理函数之前, 把U1 的上下文拷贝到信号处理函数的栈中(一般信号处理函数的栈也是当前进程的用户态的栈, 但是程序员也可以在设置信号处理函数时指定一个自己定义的栈, 但是这里不影响这个方法, 所以我们只描述信号处理函数使用进程用户态的栈的情况). 然后再执行信号处理函数. 而当信号处理函数结束之后, 会调用sigreturn() 从U2 的栈中把U1 的上下文拷贝到内核栈中.

下图描述了信号处理函数的执行流程. 一个非阻塞的信号发给目标进程. 当一个中断或异常发生后, 目标进程从用户态(U1) 进入核心态. 在它切换回用户态(U1) 之前, 内核调用do\_signal. 这个函数逐一处理悬挂的非阻塞信号. 而如果目标进程设置了对信号的处理函数, 那么它会调用handle\_signal 来调用自定义的信号处理函数( 这期间需要使用 setup\_frame 或setup\_rt\_frame来为信号处理函数设置栈 ), 此时当切换到用户态时, 目标进程执行的是信号处理函数而不是U1.当信号处理函数结束后, 位于 setup\_frame 或setup\_rt\_frame 栈之上的**返回代码** ( return code)被执行, 这返回代码会执行sigreturn 或者rt\_sigreturn 从而把U1 的上下文从setup\_frame或setup\_rt\_frame 栈中拷贝到核心栈. 而这结束后, 内核可以切换回U1.

注意: 信号有三种处理方式, 只有使用自定义处理函数才需要这样麻烦啊.



接下来我们需要仔细瞧瞧这一切怎么发生的.

Setting up the frame

为了能恰当地为信号处理函数设置栈,handle\_signal 调用setup\_frame( 当信号没有相应的siginfo\_t 时) 或者setup\_rt\_frame( 当信号有相应的siginfo\_t 时). 为了判断采用哪一种, 需要参考 sigaction 中的sa\_flag 是否包含SA\_SIGINO.

**setup\_frame**接受四个参数, 如下:

sig: 信号标识

ka: 与信号相关的 k\_sigaction 实例

oldest: 进程阻塞的信号

regs: U1 上下为在核心栈的地址.

setup\_frame 函数会在用户栈中分配一个sigframe 变量, 该变量包含了能够正确调用信号处理函数的信息( 这些信息会被 sys\_sigreturn 使用 ). sigframe 的成员如下( 其示意图如下):

pretcode : 信号处理函数的返回地址. 其指向标记为 kernel\_sigreturn 的代码

sig : 信号标识.

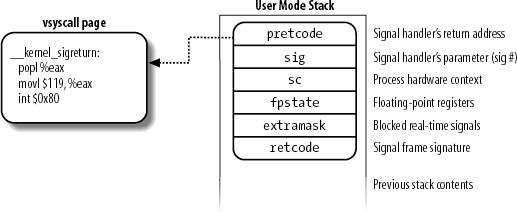
sc : sigcontext 变量. 它包含了U1 的上下文信息, 以及被进程阻塞的非实时信号的信息.

fpstate : \_fpstate 实例, 用来存放U1 的浮点运算有关的寄存器.

extramask : 被进程阻塞的实时信号的信息 .

retcode :8 字节的返回代码, 用于发射 sigreturn 系统调用. 早期版本的linux 用于信号处理函数返回后的善后处理.linux2.6 则用于特征标志, 所以调试器能够知道这是一个信号处理函数的栈.

**Frame on the User Mode stack**



setup\_frame 函数首先获得sigframe 变量的地址, 如下:

frame =(regs->esp - sizeof(struct sigframe)) & 0xfffffff8

注意: 默认地信号处理函数使用得到栈是进程在用户态下的栈, 但是用户在设置信号处理函数时可以指定. 这里只讨论默认情况. 对于用户指定其实也一样.

另外由于栈从大地址到小地址增长, 所以上面的代码要看明白了. 此外还需要8 字节对齐.

之后使用 access\_ok 来验证 frame 是否可用, 之后用\_\_put\_user 来填充frame 各个成员. 填充好之后, 需要修改核心栈, 这样从核心态切换到用户态时就能执行信号处理函数了, 如下:

    regs->esp = (unsigned long) frame;

**regs->eip** = (unsigned long) ka->sa.sa\_handler;

    regs->eax = (unsigned long) sig;

    regs->edx = regs->ecx = 0;

    regs->xds = regs->xes = regs->xss = \_ \_USER\_DS;

    regs->xcs = \_ \_USER\_CS;

setup\_rt\_frame 和setup\_frame 类似, 但是它在用户栈房的是一个rt\_sigframe 的实例, rt\_sigframe 除了sigframe 外还包含了siginfo\_t( 它描述了信号的信息). 另外它使用 \_ \_kernel\_rt\_sigreturn.

**Evaluating the signal flags**

设置好栈后,handle\_signal 检查和信号有关的flags. 如果没有设置 SA\_NODEFER , 那么在执行信号处理函数时, 就要阻塞sigaction.sa\_mask 中指定的所有信号以及sig 本身. 如下:

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

        spin\_lock\_irq(&current->sighand->siglock);

       sigorsets(&current->blocked, &current->blocked, &ka->sa.sa\_mask);

        sigaddset(&current->blocked, sig);

        recalc\_sigpending(current);

        spin\_unlock\_irq(&current->sighand->siglock);

}

如前文所述,recalc\_sigpending 会重新检查进程是否还有未被阻塞的悬挂信号, 并依此设置进程的 TIF\_SIGPENDING 标志.

注意: sigorsets(&current->blocked, &current->blocked, &ka->sa.sa\_mask) 等价于current->blocked |= ka->sa.sa\_mask. 而current->blocked 原来的值已经存放在frame 中了.

handle\_signal 返回到do\_signal 后,do\_signal 也立即返回.

**Starting the signal handler**

do\_signal 返回后, 进程由核心态切换到用户态, 于是执行了信号处理函数.

**Terminating the signal handler**

信号处理函数结束后, 因为其返回值的地址( pretcode 指定的 ) 是\_ \_kernel\_sigreturn 指向的代码段, 所以就会执行\_ \_kernel\_sigreturn 指向的代码. 如下:

    \_ \_kernel\_sigreturn:

      popl %eax

      movl $\_ \_NR\_sigreturn, %eax

      int $0x80

这会导致 sigreturn 被执行 ( 会导致从用户态切换到核心态).

sys\_sigreturn 函数可以计算得到sigframe 的地址. 如下:

    frame = (struct sigframe \*)(regs.esp - 8);

    if (verify\_area(VERIFY\_READ, frame, sizeof(\*frame)) {

        force\_sig(SIGSEGV, current);

         return 0;

    }

接着, 它要从frame 中把进程真正阻塞的信号信息拷贝到current->blocked 中. 结果那些在sigaction 中悬挂的信号解除了阻塞. 之后调用 recalc\_sigpending.

接着 sys\_sigreturn 需要调用restore\_sigcontext 把frame 的sc( 即U1 的上下文) 拷贝到内核栈中并把frame 从用户栈中删除.

\_ \_kernel\_sigreturn 的处理与这类似.

**重新执行系统调用(**被信号处理掐断的系统调用 **)**

注: 当用核心态转向用户态时, 该核心态可能是系统调用的核心态.

小小总结 : 当内核使用用户指定的处理方式时 , 因为是从用户态转向内核态再转向用户态 , 所以其处理比较复杂 . 如下描述 : 当从用户态 (U1) 转入内核态后 , 在内核态试图回到 U1 时 , 会先判断是否有非阻塞的悬挂信号, 如果有就会先调用用户的处理函数 ( 即进入用户态 , 这里是用户态 2), 处理完后 , 再回到内核态 , 然后再回到U1.  注意在 U2 中也有可能发生系统调用从而再次进入内核态 . ( 注意在 U2 过程中 , 系统处于关中断状态 , 所以信号处理应该尽可能地快 ), 我们知道当用户态进入核心态时会把用户态的信息保存在核心态的栈中 ,  为了避免在从 U2 因系统调用再进入核心态是破坏 U1 在核心态中的信息 , 在进入 U2 之前 , 要不 U1 在核心栈中的信息拷贝到 U1 的栈中 , 并在 U2 返回后 , 再把 U2 栈中保存 U1 的信息拷贝会核心栈 .

注 :U2 使用的栈可以和 U1 是同一个栈 , 也可以是用户在设置信号处理函数时指定的一段内存 .

当一个进程调用某些并不能马上满足的系统调用(eg: 写文件) 时, 内核会把该进程的状态设置为TASK\_INTERRUPTIBLE 或者TASK\_UNINTERRUPTIBLE.

当一个进程( 表示为wp) 处于TASK\_INTERRUPTIBLE 状态, 而另外一个进程又给它发信号, 那么内核会把wp 的状态的进程设置为TASK\_RUNNING( 但是此时wp 的系统调用仍未完成). 而当wp 切换会用户态时, 这个信号会被deliver. 如果这种情况真的发生了, 则系统调用服务例程并没有成功完成任务, 但是会返回错误码EINTR , ERESTARTNOHAND , ERESTART\_RESTARTBLOCK , ERESTARTSYS ,或 ERESTARTNOINTR. ( 参考中断处理的从中断返回部分).

从实践上看, 用户获得的错误代码是是EINTR, 这意味着系统调用没有成功完成. 程序员可以决定是否再次发起该系统调用. 其余的错误代码由内核使用来判断是否在信号处理之后自动重新执行该系统调用.

下表列出了这些错误代码在每一种可能的中断行为下对未完成系统调用的影响. 表中用的词定义如下:

Terminate: 该系统调用不会被内核自动重新执行. 而用户得到的该系统调用的返回值是-EINTER.对程序员而言该系统调用失败.

Reexecute: 内核会强制进程在用户态下自动重新执行该系统调用( 通过把中断号放到eax, 执行int 0x80 或者sysenter 指令). 但是这对程序员透明.

Depends: 如果当被deliver 的信号设置了 SA\_RESTART 标志, 那么自动重新执行该系统调用. 否则中止系统调用并返回-EINTER.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Reexecution of system calls** | | | | |
| **Error codes and their impact on system call execution** | | | | |
| **Signal**  **Action** | **EINTR** | **ERESTARTSYS** | **ERESTARTNOHAND**  **ERESTART\_RESTARTBLOCK** | **ERESTARTNOINTR** |
| Default | Terminate | Reexecute | Reexecute | Reexecute |
| Ignore | Terminate | Reexecute | Reexecute | Reexecute |
| Catch | Terminate | Depends | Terminate | Reexecute |

注: ERESTARTNOHAND , ERESTART\_RESTARTBLOCK 使用不同的机制来重新自动执行系统调用( 参下文).

当 delivering 一个信号时, 内核必须确信进程正在执行系统调用中，这样它才能reexecute 该系统调用, 而 regs 中的成员orig\_eax 就是干这个事情的. 回想一下这个成员在中断/ 异常时如何被初始化的:

Interrupt: 它等于 IRQ 数值 - 256.

0x80 exception ( 或者 sysenter): 它等于系统调用的编号.

Other exceptions: 它等于-1.

所以如果该值>=0, 那么可确定进程是在处于系统调用中被信号处理唤醒的( 即信号处理唤醒一个等待系统调用完成( 状态为 TASK\_INTERRUPTIBLE ) 的进程). 所以内核在delivering 信号时, 能够返回上述的错误代码, 并作出恰当的挽救.

**重启被非自定义信号处理函数中断的系统调用**

注：上面语句的中断不是OS 中的中断, 而是日常生活中的中断的含义.

如果系统调用因为信号的默认处理函数或者信号的忽略处理而中断( 即由系统调用把task 的状态改为可中断状态, 但是却被信号的默认处理函数或者忽略信号操作把该task 的状态改为running,如前文所述), 那么do\_signal 函数需要分析系统调用的错误码来决定是否自动重新执行被停止的系统调用. 如果需要重启该系统调用, 那么必须修改regs 中的内容, 从而在切换到用户态后, 在用户态下再次执行该系统调用( 即再次在用户态下让eax 存放系统调用的编号, 然后执行int 0x80或者sysenter). 如下代码:

    if (regs->orig\_eax >= 0) {

        if (regs->eax == -ERESTARTNOHAND || regs->eax == -ERESTARTSYS ||

              regs->eax == -ERESTARTNOINTR) {

            regs->eax = regs->orig\_eax;

            regs->eip -= 2;

        }

        if (regs->eax == -ERESTART\_RESTARTBLOCK) {

            regs->eax = \_\_NR\_restart\_syscall;

            regs->eip -= 2;

        }

    }

regs->eax 存放系统调用的编号 . 此外,int 0x80 或者sysreturn 均为2 字节. 所以regs->eip -=2 等价于切换到用户态后重新执行int 0x80 或者sysretrun 指令.

对于错误码 ERESTART\_RESTARTBLOCK, 它需要使用restart\_syscall 系统调用, 而不是使用原来的系统调用. 这个错误码只用在与时间有关的系统调用. 一个典型的例子是 nanosleep( ) : 想象一下, 一个进程调用这个函数来暂停20ms, 10ms 后由于一个信号处理发生( 从而激活这个进程), 如果这信号处理后重新启动这个系统调用, 那么它在重启的时候不能直接再次调用nanosleep, 否则将会导致该进程睡觉30ms. 事实上, nanosleep 会在当前进程的thread\_info 的restart\_block 中填写下如果需要重启nanosleep, 那么需要调用哪一个函数, 而如果其被信号处理中断, 那么它会返回-ERESTART\_RESTARTBLOCK, 而在重启该系统调用时,sys\_restart\_syscall 会根据restart\_block 中的信息调用相应的函数. 通常这个函数会计算出首次调用与再次调用的时间间距, 然后再次暂停剩余的时间段.

**重启由自定义信号处理函数中断的系统调用**

在这种情况下,handle\_signal 会分析错误码以及 sigaction 中的标志是否包含了SA\_RESTART, 从而决定是否重启未完成的系统调用. 代码如下:

    if (regs->orig\_eax >= 0) {

        switch (regs->eax) {

            case -ERESTART\_RESTARTBLOCK:

             case -ERESTARTNOHAND:

                regs->eax = -EINTR;

                break;

            case -ERESTARTSYS:

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

                    regs->eax = -EINTR;

                    break;

                 }

            /\* fallthrough \*/

            case -ERESTARTNOINTR:

                regs->eax = regs->orig\_eax;

                regs->eip -= 2;

        }

}

如果需要重启系统调用, 其处理与do\_signal 类似. 否则向用户态返回 -EINTR.

问题 :

在信号处理函数中可以发生中断吗 , 可以再 发出系统调用吗，可以发出异常吗 ?

如果不行 会有什么影响 ??

**与信号处理相关的系统调用**

因为当进程在用户态时, 允许发送和接受信号. 这意味着必须定义一些系统调用来允许这类操作.不幸的是, 由于历史的原因这些操作的语义有可能会重合, 也意味着某些系统调用可能很少被用到. 比如,sys\_sigaction, sys\_rt\_sigaction 几乎相同, 所以C 的接口sigaction 只调用了sys\_rt\_siaction. 我们将会描述一些重要的系统调用.

**进程组**: Shell 上的一条命令行形成一个进程组 . 注意一条命令其实可以启动多个程序 . 进程组的 ID 为其领头进程的 ID.

**kill( ) 系统调用**

原型为: int kill(pid\_t pid, int sig)

其用来给一个线程组( 传统意义上的进程) 发信息. 其对应的系统服务例程(service routine)是sys\_kill. sig 参数表示待发送的信号,pid 根据其值有不同的含义, 如下:

pid > 0: 表示信号sig 发送到由pid 标识的线程组( 即线程组的PID==pid).

pid = 0: 表示信号sig 发送到发送进程所在的进程组中的所有线程组.

pid = -1: 表示信号sig 发送到除进程0, 进程1, 当前进程外的所有进程

pid < -1: 表示信号sig 发送到进程组-pid 中的所有线程组.

服务例程sys\_kill 会初始化一个siginfo\_t 变量, 然后调用kill\_something\_info. 如下:

    info.si\_signo = sig;

    info.si\_errno = 0;

    info.si\_code = SI\_USER;

    info.\_sifields.\_kill.\_pid = current->tgid;

    info.\_sifields.\_kill.\_uid = current->uid;

    return kill\_something\_info(sig, &info, pid);

kill\_something\_info 会调用kill\_proc\_info( 这个函数调用 group\_send\_sig\_info 把信号发给线程组 ) 或者 kill\_pg\_info( 这个会扫描目标进程组然后逐一调用send\_sig\_info ) 或者为系统中的每一个进程调用group\_send\_sig\_info( 当pid=-1 时).

系统调用kill 可以发送任意信号, 然而它不保证该信号被加到目标进程的悬挂信号队列中. ( **这个是指对于非实时信号 它也有可能会丢弃该信号吗????**) 对于实时信号, 可以使用rt\_sigqueueinfo.

System V and BSD Unix 还有killpg 系统调用, 它可以给一组进程发信号. 在linux 中, 它通过kill 来实现. 另外还有一个raise 系统调用, 它可以给当前进程发信号. 在linux 中,killpg, raise 均以库函数提供.

**tkill( ) & tgkill( ) 系统调用**

这两个函数给指定线程发信号. pthread\_kill 使用它们之一来实现. 函数原型为:

int tkill(int tid, int sig);

long sys\_tgkill (int tgid, int pid, int sig);

tkill 对应的服务例程是sys\_tkill, 它也会填充一个siginfo\_t 变量, 进程权限检查, 然后掉用specific\_send\_sig\_info.

tgkill 与tkill 的差别在于它多了一个tgid 的参数, 它要求pid 必须是tgid 中的线程. 其对应的服务例程是sys\_tgkill, 它做的事情和sys\_tkill 类似, 但它还检查了pid 是否在tgid 中. 这种检查在某些情况下可以避免 race condition. 比如: 一个信号被发给了线程组A 中的一个正在被杀死的线程(killing\_id), 如果另外一个线程组B 很快地创建一个新的线程并且其PID= killing\_id,那么信号有可能会发送到线程组B 中的新建的线程. tgkill 可以避免这种情况, 因为线程组A,B的ID 不一样.

**设置信号处理函数**

程序员可以通过系统调用sigaction (sig,act,oact) 来为信号sig 设置用户自己的信号处理函数act. 当然如果用户没有设置, 那么系统会使用默认的信号处理函数. 其函数原型为:

int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact);

oldact 用来保存信号signum 的旧的信号处理函数( 因为signum 的新的信号处理函数是act, 保存旧的是希望能够恢复使用旧的信号处理函数).

其对应的服务例程是sys\_sigaction, 它首先检查act 地址的有效性, 然后act 的内容拷贝到一个类型为 k\_sigaction 的 本地变量new\_ka ，如下:

    \_ \_get\_user(new\_ka.sa.sa\_handler, &act->sa\_handler);

    \_ \_get\_user(new\_ka.sa.sa\_flags, &act->sa\_flags);

    \_ \_get\_user(mask, &act->sa\_mask);

    siginitset(&new\_ka.sa.sa\_mask, mask);

接着调用 do\_sigaction 把new\_ka 拷贝到current->sig->action[sig-1] 中的. 类似如下:

    k = &current->sig->action[sig-1];

    if (act) {

        \*k = \*act;

        sigdelsetmask(&k->sa.sa\_mask, sigmask(SIGKILL) | sigmask(SIGSTOP));

        if (k->sa.sa\_handler == SIG\_IGN || (k->sa.sa\_handler == SIG\_DFL &&

         (sig==SIGCONT || sig==SIGCHLD || sig==SIGWINCH || sig==SIGURG))) {

        rm\_from\_queue(sigmask(sig), &current->signal->shared\_pending);

            t = current;

            do {

                rm\_from\_queue(sigmask(sig), &current->pending);

                recalc\_sigpending\_tsk(t);

                t = next\_thread(t);

            } while (t != current);

        }

    }

POSIX 规定当默认行为是忽略时, 把信号处理函数设置为SIG\_IGN 或者SIG\_DFT 会导致悬挂的信号被丢弃. 此外, SIKKILL 和SIGSTOP 永远不会被屏蔽 ( 参考上述代码).

此外, sigaction 系统调用还允许程序员初始化sigaction 中的sa\_flags.

System V 也提供signal 系统调用. C 库的signal 使用rt\_sigaction 来实现. 但是linux 仍然有相应的服务例程sys\_signal. 如下:

    new\_sa.sa.sa\_handler = handler;

    new\_sa.sa.sa\_flags = SA\_ONESHOT | SA\_NOMASK;

    ret = do\_sigaction(sig, &new\_sa, &old\_sa);

    return ret ? ret : (unsigned long)old\_sa.sa.sa\_handler;

**获得被阻塞的悬挂信号**

系统调用 sigpending () 允许 用户获得当前线程被阻塞的悬挂信号. 函数原型为:

int sigpending(sigset\_t \*set);

set 用来接收被阻塞的悬挂信号的信息.

其对应的服务例程是sys\_sigpending, 其实现代码如下:

    sigorsets(&pending, &current->pending.signal,

                        &current->signal->shared\_pending.signal);

    sigandsets(&pending, &current->blocked, &pending);

    copy\_to\_user(set, &pending, 4);

**修改被阻塞的信号的集合**

系统函数sigprocmask 可以用来修改当前线程的阻塞信号集合. 但是它仅适用于非实时信号. 函数原型为:

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

假设在执行这个函数之前线程的阻塞信号的集合为bs. 执行这个函数之后线程的阻塞信号的集合为nbs.

oldsett: 用于返回( 返回) 线程当前阻塞的信号的集合(\*oldest=bs)

set: 用于存储信号集合. 怎么用它还取决于how 参数.

how: 执行线程的新的阻塞信号集合如果通过set 参数获得. 其可能的值及其含义如下:

SIG\_BLOCK: nbs=bs|set

SIG\_UNBLOCK:nbs=bs-set

SIG\_SETMASK:nbs=set

其对应的服务例程是 sys\_sigprocmask( ) . 它调用copy\_from\_user 把set 值拷贝到本地变量new\_set, 并把bs 拷贝到oldset 中. 其执行的代码类似如下:

    if (copy\_from\_user(&new\_set, set, sizeof(\*set)))

        return -EFAULT;

    new\_set &= ~(sigmask(SIGKILL)|sigmask(SIGSTOP));

    old\_set = current->blocked.sig[0];

    if (how == SIG\_BLOCK)

        sigaddsetmask(&current->blocked, new\_set);

    else if (how == SIG\_UNBLOCK)

        sigdelsetmask(&current->blocked, new\_set);

    else if (how == SIG\_SETMASK)

        current->blocked.sig[0] = new\_set;

    else

        return -EINVAL;

    recalc\_sigpending(current);

    if (oset && copy\_to\_user(oset, &old\_set, sizeof(\*oset)))

            return -EFAULT;

    return 0;

**悬挂( 暂停) 进程**

系统调用 sigsuspend 的原型如下:

int sigsuspend(const sigset\_t \*mask);

其含义是: 把本线程的阻塞信号设置为mask 并把线程状态设置为 TASK\_INTERRUPTIBLE. 并且只有当一个 nonignored, nonblocked 的信号发到本线程后才会把本线程唤醒(deliver 该信号, 系统调用返回).

其相应的服务例程为sys\_sigsuspend, 执行的代码为:

    mask &= ~(sigmask(SIGKILL) | sigmask( SIGSTOP ));

    saveset = current->blocked;// saveset 本地局部变量

    siginitset(&current->blocked, mask);

    recalc\_sigpending(current);

    regs->eax = -EINTR;

    while (1) {

        current->state = TASK\_INTERRUPTIBLE;

        schedule( );

        if (do\_signal(regs, &saveset))// 把阻塞信号集合恢复为saveset

            return -EINTR;

}

( 注意, 本系统调用本身期望它被信号处理函数中断.)

函数schedule 会导致执行别的进程( 线程), 而当本进程再次执行时( 即上面的schedule 返回了), 它会调用do\_signal 来处理其未被阻塞的悬挂的信号, 然后恢复线程的阻塞信号集合(saveset). 如果do\_signal 返回非0(do\_signal 中调用用户自定义信号处理函数或者杀死本线程时返回非0), 那么该系统调用返回.

即只有当本线程处理完不被阻塞的信号( ==(!mask)| SIGKILL| SIGSTOP) 后, 它才会返回.

**实时信号的系统调用**

前面所述的系统调用仅适用于非实时信号,linux 还引入了支持实时信号的系统调用.

一些实时系统调用( 如: rt\_sigaction, rt\_sigpending, rt\_sigprocmask, rt\_sigsuspend) 与它们的非实时的版本类似( 只是在名字加了rt\_). 下面仅简单描述两个实时信号的系统调用.

rt\_sigqueueinfo( ): 把一个实时信号发给线程组( 放到线程组的共享悬挂信号列表中). 库函数sigqueue 利用这个系统调用来实现.

rt\_sigtimedwait( ): 把阻塞的悬挂信号从悬挂信号队列中删除, 如果在调用这个系统调用时还没有相应的阻塞悬挂信号, 那么它会把本进程(task) 阻塞一段时间. 库函数sigwaitinfo,sigtimedwait 通过这个系统调用实现.

### 信号种类

信号的种类非常多，每个信号用一个整型常量宏表示，以SIG开头，比如SIGCHLD、SIGINT、SIGIO等，它们在系统头文件中定义,也可以通过在shell下键入kill –l查看信号列表，或者键入man 7 signal查看更详细的说明。要将信号的值，转换为其意义string.h中提供了一个函数char\* strsignal(int sig)。

在Shell下输入kill –l 可显示Linux 系统支持的全部依赖：

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

信号的值定义在signal.h中，在Linux中没有16和32这两个信号。上面信号的含义如下：

(1) SIGHUP：当用户退出Shell时，由该Shell启的发所有进程都退接收到这个信号，默认动作为终止进程。

(2) SIGINT：用户按下组合键ctrl+C时，用户端时向正在运行中的由该终端启动的程序发出此信号。默认动作为终止进程。

(3) SIGQUIT：当用户按下组合键ctrl+\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此信号。默认动作为终止进程并产生core文件。

(4) SIGILL ：CPU检测到某进程执行了非法指令。默认动作为终止进程并产生core文件。

(5) SIGTRAP：该信号由断点指令或其他trap指令产生。默认动作为终止进程并产生core文件。

(6) SIGABRT：调用abort函数时产生该信号。默认动作为终止进程并产生core文件。

(7) SIGBUS：非法访问内存地址，包括内存地址对齐（alignment）出错，默认动作为终止进程并产生core文件。

(8) SIGFPE：在发生致命的算术错误时产生。不仅包括浮点运行错误，还包括溢出及除数为0等所有的算术错误。默认动作为终止进程并产生core文件。

(9) SIGKILL：无条件终止进程。本信号不能被忽略、处理和阻塞。默认动作为终止进程。它向系统管理员提供了一种可以杀死任何进程的方法。

(10) SIGUSR1：用户定义的信号，即程序可以在程序中定义并使用该信号。默认动作为终止进程。

(11) SIGSEGV：指示进程进行了无效的内存访问。默认动作为终止进程并使用该信号。默认动作为终止进程。

(12) SIGUSR2：这是另外一个用户定义信号，程序员可以在程序中定义并使用该信号。默认动作为终止进程。

(13) SIGPIPE：Broken pipe：向一个没有读端的管道写数据。默认动作为终止进程。

(14) SIGALRM：定时器超时，超时的时间由系统调用alarm设置。默认动作为终止进程。

(15) SIGTERM：程序结束(terminate)信号，与SIGKILL不同的是，该信号可以被阻塞和处理。通常用来要求程序正常退出。执行Shell命令kill时，缺少产生这个信号。默认动作为终止进程。

(16) SIGCHLD：子程序结束时，父进程会收到这个信号。默认动作为忽略该信号。

(17) SIGCONT：让一个暂停的进程继续执行。

(18) SIGSTOP：停止(stopped)进程的执行。注意它和SIGTERM以及SIGINT的区别：该进程还未结束，只是暂停执行。本信号不能被忽略、处理和阻塞。默认作为暂停进程。

(19) SIGTSTP：停止进程的动作，但该信号可以被处理和忽略。按下组合键ctrl+z时发出该信号。默认动作为暂停进程。

(20) SIGTTIN：当后台进程要从用户终端读数据时，该终端中的所有进程会收到SIGTTIN信号。默认动作为暂停进程。

(21) SIGTTOU：该信号类似于SIGTIN，在后台进程要向终端输出数据时产生。默认动作为暂停进程。

(22) SIGURG：套接字（socket）上有紧急数据时，向当前正在运行的进程发出此信号，报告有紧急数据到达。默认动作为忽略该信号。

(23) 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。默认动作为终止进程。

(24) SIGXFSZ：超过文件最大长度的限制。默认动作为yl终止进程并产生core文件。

(25) SIGVTALRM：虚拟时钟超时时产生该信号。类似于SIGALRM，但是它只计算该进程占有用的CPU时间。默认动作为终止进程。

(26) SIGPROF：类似于SIGVTALRM，它不仅包括该进程占用的CPU时间还抱括执行系统调用的时间。默认动作为终止进程。

(27) SIGWINCH：窗口大小改变时发出。默认动作为忽略该信号。

(28) SIGIO：此信号向进程指示发出一个异步IO事件。默认动作为忽略。

(29) SIGPWR：关机。默认动作为终止进程。

(30) SIGRTMIN~SIGRTMAX：Linux的实时信号，它没有固定的含义(或者说可以由用户自由使用)。注意，Linux线程机制使用了前3个实时信号。所有的实时信号的默认动作都是终止进程。

发出信号的原因很多，这里按发出信号的原因简单分类，以了解各种信号：

（1） 与进程终止相关的信号。当进程退出，或者子进程终止时，发出这类信号。

（2） 与进程例外事件相关的信号。如进程越界，或企图写一个只读的内存区域（如程序正文区），或执行一个特权指令及其他各种硬件错误。

（3） 与在系统调用期间遇到不可恢复条件相关的信号。如执行系统调用exec时，原有资源已经释放，而目前系统资源又已经耗尽。

（4） 与执行系统调用时遇到非预测错误条件相关的信号。如执行一个并不存在的系统调用。

（5） 在用户态下的进程发出的信号。如进程调用系统调用kill向其他进程发送信号。

（6） 与终端交互相关的信号。如用户关闭一个终端，或按下break键等情况。

（7） 跟踪进程执行的信号。

Linux支持的信号列表如下。很多信号是与机器的体系结构相关的，首先列出的是POSIX.1中列出的信号：

信号 值 处理动作 发出信号的原因

----------------------------------------------------------------------

SIGHUP 1 A 终端挂起或者控制进程终止

SIGINT 2 A 键盘中断（如break键被按下）

SIGQUIT 3 C 键盘的退出键被按下

SIGILL 4 C 非法指令

SIGABRT 6 C 由abort(3)发出的退出指令 Process aborted

SIGBUS 7Bus error: "access to undefined portion

of memory object"

SIGFPE 8 C 浮点异常

SIGKILL 9 AEF Kill信号 Kill (terminate immediately)

SIGSEGV 11 C 无效的内存引用

SIGPIPE 13 A 管道破裂: 写一个没有读端口的管道

SIGALRM 14 A 由alarm(2)发出的信号 Signal

raised by alarm

SIGTERM 15 A 终止信号

SIGUSR1 30,10,16 A 用户自定义信号1

SIGUSR2 31,12,17 A 用户自定义信号2

SIGCHLD 20,17,18 B 子进程结束信号

SIGCONT 19,18,25 进程继续（曾被停止的进程）

SIGSTOP 17,19,23 DEF 终止进程

SIGTSTP 18,20,24 D 控制终端（tty）上按下停止键 ，停止进程的运行, 但该信号可以被处理和忽略. 用户键入SUSP字符时(通常是Ctrl-Z)发出这个信号

SIGTTIN 21,21,26 D 后台进程企图从控制终端读

SIGTTOU 22,22,27 D 后台进程企图从控制终端写

下面的信号没在POSIX.1中列出，而在SUSv2列出

信号 值 处理动作 发出信号的原因

--------------------------------------------------------------------

SIGBUS 10,7,10 C 总线错误(错误的内存访问)

SIGPOLL A Sys V定义的Pollable事件，与SIGIO同义

SIGPROF 27,27,29 A Profiling定时器到

SIGSYS 12,-,12 C 无效的系统调用 (SVID)

SIGTRAP 5 C 跟踪/断点捕获

SIGURG 16,23,21 B Socket出现紧急条件(4.2 BSD)

SIGVTALRM 26,26,28 A 实际时间报警时钟信号(4.2 BSD)

SIGXCPU 24,24,30 C 超出设定的CPU时间限制(4.2 BSD)

SIGXFSZ 25,25,31 C 超出设定的文件大小限制(4.2 BSD)

（对于SIGSYS，SIGXCPU，SIGXFSZ，以及某些机器体系结构下的SIGBUS，Linux缺省的动作是A (terminate)，SUSv2 是C (terminate and dump core)）。

下面是其它的一些信号

信号 值 处理动作 发出信号的原因

----------------------------------------------------------------------

SIGIOT 6 C IO捕获指令，与SIGABRT同义

SIGEMT 7,-,7

SIGSTKFLT -,16,- A 协处理器堆栈错误

SIGIO 23,29,22 A 某I/O操作现在可以进行了(4.2 BSD)

SIGCLD -,-,18 A 与SIGCHLD同义

SIGPWR 29,30,19 A 电源故障(System V)

SIGINFO 29,-,- A 与SIGPWR同义

SIGLOST -,-,- A 文件锁丢失

SIGWINCH 28,28,20 B 窗口大小改变(4.3 BSD, Sun)

SIGUNUSED -,31,- A 未使用的信号(will be SIGSYS)

（在这里，- 表示信号没有实现；有三个值给出的含义为，第一个值通常在Alpha和Sparc上有效，中间的值对应i386和ppc以及sh，最后一个值对应mips。信号29在Alpha上为SIGINFO / SIGPWR ，在Sparc上为SIGLOST。）

处理动作一项中的字母含义如下

A 缺省的动作是终止进程

B 缺省的动作是忽略此信号

C 缺省的动作是终止进程并进行内核映像转储（dump core）

D 缺省的动作是停止进程

E 信号不能被捕获

F 信号不能被忽略

上 面介绍的信号是常见系统所支持的。以表格的形式介绍了各种信号的名称、作用及其在默认情况下的处理动作。各种默认处理动作的含义是：终止程序是指进程退 出；忽略该信号是将该信号丢弃，不做处理；停止程序是指程序挂起，进入停止状况以后还能重新进行下去，一般是在调试的过程中（例如ptrace系统调 用）；内核映像转储是指将进程数据在内存的映像和进程在内核结构中存储的部分内容以一定格式转储到文件系统，并且进程退出执行，这样做的好处是为程序员提 供了方便，使得他们可以得到进程当时执行时的数据值，允许他们确定转储的原因，并且可以调试他们的程序。

注意 信号SIGKILL和SIGSTOP既不能被捕捉，也不能被忽略。信号SIGIOT与SIGABRT是一个信号。可以看出，同一个信号在不同的系统中值可能不一样，所以建议最好使用为信号定义的名字，而不要直接使用信号的值。

### 产生信号

信号的生成来自内核，让内核生成信号的请求有很多方式：

1. 某用户：用户能够通过输入组合键（CTRL+C、CTRL+\，CTRL+Z等），或者是终端驱动程序分配给信号控制字符的其他任何键来请求内核产生信号，kill命令；
2. 内核：当进程执行出错时，内核会给进程发送一个信号，例如非法段存取(内存访问违规)、浮点数溢出等；
3. 进程：一个进程可以通过系统调用kill或者sigqueue给另一个进程发送信号，一个进程可以通过信号和另外一个进程进行通信。当检测到某种软件条件已经具备时发出信号，如由alarm或settimer设置的定时器超时时将生成SIGALRM信号。

值得注意的是当我们发送信号时受到权限的限制，发送一个信号到另一个没有权限的进程是不合法的。我们通过使用CTRL+C组合键发送信号SIGINT给当前的进程。但是这种方法只能发送少部分信号且并不适用所有的进程比如后台进程和守护进程。守护进程不必说，连终端都没有。交互shell （interactive shell）在启动一个后台进程的时候，会自动把中断和退出信号设置为忽略。

#### kill命令

首先是shell命令kill其用法如下：

kill [-s signal|-p] [-q sigval] [-a] [--] pid...

-s signal signal可以是诸如SIGINT，SIGQUIT之类的宏，亦可以是1,2,3...这样的值，可以随意使用，你开心就好。

-q queue sigval是值，可以伴随信号传递，但是这里只可以是一个integer，在进程中可以使用sigaction()接收到这个值，与之对应的是另一个函数sigqueue()。

pid就是目标进程的进程id，可以是一个或者多个。但是发送信号时，要确保你所使用的用户是具有发送信号到目标进程的权限的。

kill的选项远不止这些，但是通常这些已经够用了。如有兴趣请自行 “man 1 kill”查看。

shell 命令行下是交互式模式 (interactive),

运行是脚本时是非交互式模式 (non-interactive).

非交互式 shell 默认禁用了 job control,

这时启动后台进程时 shell 会设置后台进程忽略 SIGINT 等信号.

因此默认情况下 shell 脚本启动的后台进程会忽略 SIGINT 等信号.

可以在 shell 脚本中设置 set -m 打开 job control, 避免这个问题.

#### kill()系统调用

和shell命令kill有一个同名的系统调用kill(),其原型是这样的：

int kill(pid\_t pid, int sig);

相信看了上边的shell指令，这个函数的用法就一目了然了吧。pid是目标进程的pid，sig是要发送的信号。如果参数sig为0，将不发送信号。和其他函数一样它也是成功返回0，失败-1。然而真的这么简单吗？事实上不是。pid这个参数在这里大有学问。它的取值不仅仅可以是进程id，它甚至可以是负的。如果你对linux下编程熟悉的话，这样的用法肯定接触过，获取消息队列时使用的msgrcv()函数，其中的msgtype参数也具有类似的用法。当然扯远了。

pid>0 此时正式最普通的一种情况，pid是要目标进程的pid。

pid=0 那么kill()会将信号发送给调用进程同组的所有进程，也包括他自己。

pid=-1 那么信号将被发送至所有它具有权限发送信号的每一个进程（init进程和调用进程除外）。

pid<-1 信号会发送sig信号到组id等于该pid绝对值的进程组中的每一个进程。

如果pid在以上四种情况之外，无法匹配到目标进程，那么就会返回-1，errno被设置为ESRCH。当没有权限发送时kill()也将失败返回-1，errno会被设置为EPERM。

下面是一些可能返回的错误代码：

EINVAL：指定的信号sig无效。

ESRCH：参数pid指定的进程或进程组不存在。注意，在进程表项中存在的进程，可能是一个还没有被wait收回，但已经终止执行的僵死进程。

EPERM： 进程没有权力将这个信号发送到指定接收信号的进程。因为，一个进程被允许将信号发送到进程pid时，必须拥有root权力，或者是发出调用的进程的UID 或EUID与指定接收的进程的UID或保存用户ID（savedset-user-ID）相同。如果参数pid小于-1，即该信号发送给一个组，则该错误 表示组中有成员进程不能接收该信号。

与kill()类似的还有一个函数killpg()，用法简单多了，也不浪费篇幅了，查看manual就能搞定。

#### raise()系统调用

它只接受一个参数signal，然后把该信号传递给调用进程：int raise(int sig);//成功返回0，失败返回-1

由于这个函数不需要引用进程ID，它是被纳入C99标准的函数。

#### pause()系统调用

系统调用pause的作用是等待一个信号。该调用的声明格式如下：

int pause(void);

在使用该调用的进程中加入以下头文件：

#include <unistd.h>

该调用使得发出调用的进程进入睡眠，直到接收到一个信号为止。该调用总是返回-1，并设置错误代码为EINTR（接收到一个信号）。

#### alarm()和 setitimer()系统调用

系统调用alarm的功能是设置一个定时器，当定时器计时到达时，将发出一个信号给进程。该调用的声明格式如下：

unsigned int alarm(unsigned int seconds);

在使用该调用的进程中加入以下头文件：

#include <unistd.h>

系 统调用alarm安排内核为调用进程在指定的seconds秒后发出一个SIGALRM的信号。如果指定的参数seconds为0，则不再发送 SIGALRM信号。后一次设定将取消前一次的设定。该调用返回值为上次定时调用到发送之间剩余的时间，或者因为没有前一次定时调用而返回0。

注意，在使用时，alarm只设定为发送一次信号，如果要多次发送，就要多次使用alarm调用。

对于alarm，这里不再举例。现在的系统中很多程序不再使用alarm调用，而是使用setitimer调用来设置定时器，用getitimer来得到定时器的状态，这两个调用的声明格式如下：

int getitimer(int which, struct itimerval \*value);

int setitimer(int which, const struct itimerval \*new\_value, struct itimerval \*old\_value);

在使用这两个调用的进程中加入以下头文件：

#include <sys/time.h>

该系统调用给进程提供了三个定时器，它们各自有其独有的计时域，当其中任何一个到达，就发送一个相应的信号给进程，并使得计时器重新开始。三个计时器由参数which指定，如下所示：

TIMER\_REAL：按实际时间计时，计时到达将给进程发送SIGALRM信号。

ITIMER\_VIRTUAL：仅当进程执行时才进行计时。计时到达将发送SIGVTALRM信号给进程。

ITIMER\_PROF：当进程执行时和系统为该进程执行动作时都计时。与ITIMER\_VIR-TUAL是一对，该定时器经常用来统计进程在用户态和内核态花费的时间。计时到达将发送SIGPROF信号给进程。

定时器中的参数value用来指明定时器的时间，其结构如下：

struct itimerval {

struct timeval it\_interval; /\* 下一次的取值 \*/

struct timeval it\_value; /\* 本次的设定值 \*/

};

该结构中timeval结构定义如下：

struct timeval {

long tv\_sec; /\* 秒 \*/

long tv\_usec; /\* 微秒，1秒 = 1000000 微秒\*/

};

在setitimer 调用中，参数ovalue如果不为空，则其中保留的是上次调用设定的值，通常用不上，设置为NULL，它是用来存储上一次setitimer调用时设置的new\_value值。

定时器将it\_value递减到0时，产生一个信号，并将it\_value的值设 定为it\_interval的值，然后重新开始计时，如此往复。

基于此机制，setitimer既可以用来延时执行，也可定时执行。

假如it\_value为0是不会触发信号的，所以要能触发信号，it\_value得大于0；如果it\_interval为零，只会延时，不会定时（也就是说只会触发一次信号)。

调用成功时，返回0；错误时，返回-1，并设置相应的错误代码errno：

EFAULT：参数value或ovalue是无效的指针。

EINVAL：参数which不是ITIMER\_REAL、ITIMER\_VIRT或ITIMER\_PROF中的一个。

下面是关于setitimer调用的一个简单示范，在该例子中，每隔一秒发出一个SIGALRM，每隔0.5秒发出一个SIGVTALRM信号：

#include <sys/time.h>

#include <signal.h>

#include <unistd.h>

#include <stdio.h>

int sec;

void sigroutine(int signo)

{

switch(signo)

{

case SIGALRM:

{

printf("Catch a signal -- SIGALRM\n");

break;

}

case SIGVTALRM:

{

printf("Catch a signal -- SIGVTALRM\n");

break;

}

}

}

void main()

{

struct itimerval value, ovalue, value2;

sec = 5;

printf("Process id = %d.\n", getpid());

signal(SIGALRM, sigroutine);

signal(SIGVTALRM, sigroutine);

value.it\_value.tv\_sec = 1;

value.it\_value.tv\_usec = 0;

value.it\_interval.tv\_sec = 1;

value.it\_interval.tv\_usec = 0;

setitimer(ITIMER\_REAL, &value, &ovalue);

value2.it\_value.tv\_sec = 0;

value2.it\_value.tv\_usec = 500000;

value2.it\_interval.tv\_sec = 0;

value2.it\_interval.tv\_usec = 500000;

setitimer(ITIMER\_VIRTUAL, &value2, &ovalue);

for(;;);

}

该例子的屏幕拷贝如下：

localhost:~$ ./timer\_test

process id is 579

Catch a signal – SIGVTALRM

Catch a signal – SIGALRM

Catch a signal – SIGVTALRM

Catch a signal – SIGVTALRM

Catch a signal – SIGALRM

Catch a signal –GVTALRM

除了这几种产生信号的shell命令和函数之外还有一些情况下可以产生信号，比如alarm(),settimer()之类的一些与时间相关的函数，以及一些常见的软硬件错误都会产生信号。

调用pause和alarm是通过信号实现的进程暂停和定时器，调用alarm是通过信号通知进程定时器到时。

（1） int raise(int sig); 对当前进程发送指定信号

（2） int pause(void); 将进程挂起等待信号

（3） int kill(pid\_t pid,int sig); 通过进程编号发送信号

（4） unsigned int alarm(unsigned int seconds); 指定时间（秒）发送SIGALRM信号。 seconds 为0 时取消所有已设置的alarm请求；

（5）int sigqueue(pid\_t pid,int sig,const union sigval val);类似于kill函数，多了附带共用体 union sigval形数，将共用体中的成员 int sival\_int 或 void \*sival\_ptr 的值传递给 信号处理函数中的定义类型 siginfo\_t 中的 int si\_int 或 void \*si\_ptr；

（6）int setitimer(int which,const struct itimerval \*value,struct itimerval \*oldvalue); 可定时发送信号，根据which可指定三种信号类型：SIGALRM、SIGVTALRM 和 SIGPROF；作用时间也因which值不同而不同;struct itimerval 的成员 it\_interval定义间隔时间，it\_value 为0时，使计时器失效；

（7） void abort(void) 将造成进程终止；除非捕获SIGABORT信号;

### 信号处理

进程接收到信号以后，可以有如下3种选择进行处理：

1. 接收默认处理：接收默认处理的进程通常会导致进程本身消亡。例如连接到终端的进程，用户按下CTRL+c，将导致内核向进程发送一个SIGINT的信号，进程如果不对该信号做特殊的处理，系统将采用默认的方式处理该信号，即终止进程的执行；对于大部分的信号，Linux系统都有默认的处理方式。而大部分默认的处理方式是终止程序并转储core文件。
2. 忽略信号：进程可以通过代码，显示地忽略某个信号的处理，例如：signal(SIGINT,SIGDEF)；但是某些信号是不能被忽略的，
3. 捕捉信号并处理：进程可以事先注册信号处理函数，当接收到信号时，由信号处理函数自动捕捉并且处理信号。

有两个信号既不能被忽略也不能被捕捉，它们是SIGKILL和SIGSTOP。即进程接收到这两个信号后，只能接受系统的默认处理，即终止线程。

程序不可捕获、阻塞或忽略的信号有：SIGKILL,SIGSTOP

不能恢复至默认动作的信号有：SIGILL,SIGTRAP

默认会导致进程流产的信号有：SIGABRT、SIGBUS、SIGFPE、SIGILL、SIGIOT、SIGQUIT、SIGSEGV、SIGTRAP、SIGXCPU、SIGXFSZ

默认会导致进程退出的信号有：SIGALRM、SIGHUP、SIGINT、SIGKILL、SIGPIPE、SIGPOLL、SIGPROF、SIGSYS、SIGTERM、SIGUSR1、SIGUSR2、SIGVTALRM

默认会导致进程停止的信号有：SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU

默认进程忽略的信号有：SIGCHLD、SIGPWR、SIGURG、SIGWINCH

#### signal

要处理信号，Linux系统处理信号的接口有两个sigaction(),signal(),较简单的是signal()函数，其形式如下：

typedef void (\*sighandler\_t)(int);

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

siganl()函数有两个参数其中有一个int的参数便是要处理的信号，诸如SIGINT的宏。另一个参数类型为sighandler\_t的函数指针，handler指针对应的函数我们称之为：信号处理函数（signal-handler function）。可见signal()的第二个参数是一个信号处理函数，返回值也是一个信号处理函数,失败返回宏SIG\_ERR（SIGKILL和SIGSTOP的默认行为分别是杀死和停止一个进程，任何试图改变这两个信号的处理方式的行为都将返回错误）。这样经典形式的函数在Linux上我们经常会经常碰到。signal()函数的作用就是建立一个signum信号的处理函数(establish a signal handler function)。通俗一点来说就是当signum信号到来时，进程会保存当前的进程栈，转去执行siganl()中指定的handler函数。之前提到过，信号的响应方式有多种，因此handler不仅可以是一个函数指针也可以是ISO C为我们定义的宏：SIG\_IGN,SIG\_DFL,和他们的名字一样SIG\_IGN是忽略这个信号，SIG\_DFL是保持这个信号的默认处理方式（默认处理方式也可以可以是SIG\_IGN ，比较绕，但是合理）。前文提到的三个宏定义分别如下(/usr/include/bits/signum.h)：

#define SIG\_ERR ((\_\_sighandler\_t) -1)

#define SIG\_DFL ((\_\_sighandler\_t) 0)

#define SIG\_IGN ((\_\_sighandler\_t) 1)

#include <signal.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>

int g\_Seq = 0;

void signalHandler(int iSigNo)

{

int seq = g\_Seq++;

int ret = 0;

//输出不加上回车，会放在缓冲区中，直到下次触发清缓冲区

printf("%d Enter SignHandler,signo:%d.\n", seq, iSigNo);

ret = sleep(3);

printf("%d Leave SignHandler,signo:%d. sleepret: %d\n", seq, iSigNo, ret);

}

int main(void)

{

char buf[8];

int ret;

int compareRet = 0;

//推测: 在按下对应的ctrl+C 或者ctrl+\时，信号就会产生

signal(SIGINT, signalHandler);

signal(SIGQUIT, signalHandler);

do

{

//若输入字符后不输入回车，输入ctrl+C会产生信号，处理完信号后会清空缓冲区

ret = read(STDIN\_FILENO, buf, sizeof(buf) - 1);

if(ret<0)

{

printf("read fail.\n");

break;

}

else

{

printf("ret: %d.\n", ret);

}

buf[ret == 0 ? 0 : ret - 1] = '\0';

printf("Get: %s, %d.\n", buf, strlen(buf));

compareRet = strcmp(buf, "quit");

printf("Compare: %d.\n", compareRet);

}

while(compareRet != 0);

return 0;

}

程序运行时，针对于如下几种输入情况(要输入得快)，看输出结果：

1、 CTRL+c] [CTRL+c] [CTRL+c]

2、 [CTRL+c] [CTRL+\]

3、 hello [CTRL+\] [Enter]

4、 [CTRL+\] hello [Enter]

5、 hel [CTRL+\] lo[Enter]

1. 只能看到两次信号处理，第三次的信号处理被丢掉了，由此得出相同类型的信号是串行处理的，且信号可能会丢失。在按下第一次时，信号即被处理了，这是在按第二次时是可以被添加到pending队列的，但是只能被添加一次，所以第三次按键就被忽略了。

^C0 Enter SignHandler,signo:2.

^C^C0 Leave SignHandler,signo:2. sleepRet: 0.

1 Enter SignHandler,signo:2.

1 Leave SignHandler,signo:2. sleepRet: 0.

1. 看到两个类型的信号都被处理了，但SIGINT的信号处理函数被SIGQUIT中断了。**不同类型的信号可以被中断？**

^C2 Enter SignHandler,signo:2.

^\3 Enter SignHandler,signo:3.

3 Leave SignHandler,signo:3. sleepRet: 0.

2 Leave SignHandler,signo:2. sleepRet: 2.

1. SIGQUIT信号中断了输入过程，当信号处理返回后，回车，只读入了回车符，之前的字符被清除掉了

hello^\4 Enter SignHandler,signo:3.

4 Leave SignHandler,signo:3. sleepRet: 0.

ret: 1.

Get: , 0.

Compare: -1.

1. 先处理信号，处理完信号之后，读入输入的字符。

^\6 Enter SignHandler,signo:3.

hello

6 Leave SignHandler,signo:3. sleepRet: 0.

ret: 6.

Get: hello, 5.

Compare: -1.

1. 同上面的3和4，只能处理信号之后的字符

hel^\7 Enter SignHandler,signo:3.

lo

7 Leave SignHandler,signo:3. sleepRet: 0.

ret: 3.

Get: lo, 2.

Compare: -1.

### 不可靠信号与可靠信号的语义

信号的可靠与不可靠主要体现在两个方面：

* 对于不可靠信号，进程每次处理信号后，都会将信号的处理方式设置为默认动作。而对于可靠信号，它的处理函数执行以后，对该信号的处理方式不会发生变化。
* 信号可能会丢失。

由于Linux信号机制基本上从早期的UNIX系统上的信号机制移植过来的，所以Linux仍旧支持这些早期的不可靠信号。但是Linux也对不可靠信号做了（上面两点区别的第一小点）改进，即不可靠信号处理方式，不会在处理函数执行后变成默认方式。所以，在Linux上对于不可靠信号与可靠信号的区别就在于是否支持排队。

### 信号集

在处理信号相关的函数时，我们时常需要一种的特殊的数据结构来表示一组信号的集合，这样的集合我们称之为信号集，其数据类型表示为sigset\_t，通常是用位掩码的形式来实现的。

在进程表的表项中有一个软中断信号域，该域中每一位对应一个信号，当有信号发送给进程时，对应位置位。由此可以看出，进程对不同的信号可以同时保留，但对于同一个信号，进程并不知道在处理之前来过多少个。

### 信号屏蔽

信号屏蔽字它定义了要阻塞递送到当前进程的信号集，每一个进程都有一个信号屏蔽字（signal mask）。sigprocmask()函数可以检测和更改当前进程的信号屏蔽字。其原型：

int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);当oldset是一个非空指针的话，调用sigprocmask之后，oldset便返回了之前的信号屏蔽字。set参数会结合how参数对当前的信号屏蔽字做出修改。（和之前一节提到过的一样有两个特殊的信号，你不可以屏蔽它们是：SIGKILL和SIGSTOP）。

### 标准信号和实时信号

Linux上signal()注册的信号处理函数在执行时，会自动的将当前的信号添加到进程的信号屏蔽字当中。当信号处理函数返回时，会恢复之前的信号屏蔽字。这意味着，当信号处理函数执行时，它不会递归的中断自身。如果某个或多个信号在进程屏蔽了该信号的期间来到过一次或者多次，我们称这样的信号叫做未决的(pending)信号。那么在调用sigprocmask()解除这个信号屏蔽之后，该信号会在sigprocmask ()返回之前，递送给（SUSv3 规定至少传递一个信号）当前进程。

进程维护了一个数据结构来保存未决的信号，我们可以通过sigpending()来获取哪些信号是未决的：

int sigpending(sigset\_t \*set);//return 0 on success，or -1 on error。set参数返回的便是未决的信号集。之后便可以通过使用sigismember()来判断，set中包含哪些信号。

对于未决的信号，标准信号和实时信号的处理方式是不一样的：

* Realtime signals provide an increased range of signals that can be used for application-defined purposes. Only two standard signals are freely available for application-defined purposes: SIGUSR1 and SIGUSR2.
* Realtime signals are queued. If multiple instances of a realtime signal are sent to a process, then the signal is delivered multiple times. By contrast, if we send further instances of a standard signal that is already pending for a process, that *signal is delivered only once.*
* *When sending a realtime signal, it is possible to specify data (an integer or pointer value) that accompanies the signal. The signal handler in the receiving process can retrieve this data.*
* *The order of delivery of different realtime signals is guaranteed. If multiple different realtime signals are pending, then the lowest-numbered signal is delivered first. In other words, signals are prioritized, with lower-numbered signals having higher priority. When multiple signals of the same type are queued, they are delivere—along with their accompanying data—in the order in which they were sent.*

在Linux系统中，信号的可靠性是指信号是否会丢失，或者说该信号是否支持排除。SIGHUP( 1 ) ~ SIGSYS( 31 )之间的信号都是继承自UNIX系统是不可靠信号。Linux系统根据POSIX标准定义了SIGRTMIN(33) ~ SIGRTMAX(64)之间的信号，它们都是可靠信号，也称为实时信号。

编码为 0 的信号 不是有效信号，只用于检查是当前进程否有发送信号的 权限 ，并不真正发送。

当导致产生信号的事件发生时，内核就产生一个信号。信号产生后，内核通常会在进程表中设置某种形的标志。当内核设置了这个标志，我们就说内核向一个进程递送了一个信号。信号产生(generate)和递送(delivery)之间的时间间隔，称主信号未决(pending)。

进程可以调用sigpending将信号设为阻塞，如果为进程产生一个阻塞信号，而对信号的动作是捕捉该信号(即不忽略信号)，则内核将为该进程的此信号保持为未决状态，直到该进程对此信号解除阻塞或者对此信号的响应更改为忽略。如果在进程解除对某个信号的阻塞之前，这种信号发生了多次，那么如果信号被递送多次（即信号在未决信号队列里面排队），则称之为可靠信号；只被递送一次的信号称为不可靠信号。

信号实质上是软中断，中断有优先级，信号也有优先级。如果一个进程有多个未决信号，则对于同一个未决的实时信号，内核将按照发送的顺序来递送信号。如果存在多个未决信号，则值（或者说编号）越小的越先被递送。如果即存在不可靠信号，又存在可靠信号（实时信号），虽然POSIX对这一情况没有明确规定，但Linux系统和大多数遵循POSIX标准的操作系统一样，将优先递送不可靠信号。

实时信号的宏是SIGRTMIN，可以在此基础上递增自定义新的宏。

### 实时信号传递数据

#### sigaction()

      之前我们已经解释了signal()函数，sigaction()是另外一种选择，它功能更加强大，兼容性更好，任何时候我们都应优先考虑使用sigaction(),即使signal()更加简单灵活。其函数原型：

int sigaction(int signum, const struct sigaction \*act,struct sigaction \*oldact);//Return 0 on success，or -1 on error

与sigprocmask类似地，oldact返回之前的信号设置，act用来设置新的信号处理。signum自然不用解释，这是要处理的信号。这个函数的关键之处就是struct sigaction这个和函数同名的结构体。当然要使用sigaction()还是得从struct sigaction入手，它的定义：

struct sigaction {

    union {  
        void (\*sa\_handler)(int);                                   
        void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

}\_\_sigaction\_handler;                                  //Address of handler  
    sigset\_t sa\_mask;                                        //Signals blocked during the handler invocation  
    int sa\_flags;                                                //Flags controlling handler invocation  
    void (\*sa\_restorer)(void);                             //Restore,not use  
};

该结构体的各字段含义及使用方式：

1、字段sa\_handler是一个函数指针，用于指向原型为void handler(int)的信号处理函数地址， 即老类型 的信号处理函数；

2、字段sa\_sigaction也是一个函数指针，用于指向原型为：

void handler(int iSignNum,siginfo\_t \*pSignInfo,void \*pReserved)；

的信号处理函数，即新类型的信号处理函数。

该函数的三个参数含义为：

iSignNum ：传入的信号

pSignInfo ：与该信号相关的一些信息，它是个结构体

pReserved ：保留，现没用

3、字段sa\_handler和sa\_sigaction只应该有一个生效，如果想采用老的信号处理机制，就应该让sa\_handler指向正确的信号处理函数；否则应该让sa\_sigaction指向正确的信号处理函数，并且让字段 sa\_flags包含SA\_SIGINFO选项。

4、字段sa\_mask是一个包含信号集合的结构体，该结构体内的信号表示在进行信号处理时，将要被阻塞的信号。针对sigset\_t结构体，有一组专门的函数对它进行处理，它们是：

#include

int sigemptyset(sigset\_t \*set); // 清空信号集合set

int sigfillset(sigset\_t \*set); // 将所有信号填充进set中

int sigaddset(sigset\_t \*set, int signum); // 往set中添加信号signum

int sigdelset(sigset\_t \*set, int signum); // 从set中移除信号signum

int sigismember(const sigset\_t \*set, int signum); // 判断signnum是不是包含在set中

例如，如果打算在处理信号SIGINT时，只阻塞对SIGQUIT信号的处理，可以用如下种方法：

struct sigaction act;

sigemptyset(&act.sa\_mask);

sigaddset(&act\_sa\_mask,SIGQUIT);

sigaction(SIGINT,&act,NULL);

5、 字段sa\_flags是一组掩码的合成值，指示信号处理时所应该采取的一些行为，各掩码的含义为：

掩码 描述

SA\_RESETHAND 处理完毕要捕捉的信号后，将自动撤消信号处理函数的注册，即必须再重新注册信号处理函数，才能继续处理接下来产生的信号。该选项不符合一般的信号处理流程，现已经被废弃。

SA\_NODEFER 在处理信号时，如果又发生了其它的信号，则立即进入其它信号的处理，等其它信号处理完毕后，再继续处理当前的信号，即递规地处理。如果sa\_flags包含了该掩码，则结构体sigaction的sa\_mask将无效！

SA\_RESTART 如果在发生信号时，程序正阻塞在某个系统调用，例如调用read()函数，则在处理完毕信号后，接着从阻塞的系统返回。该掩码符合普通的程序处理流程，所以一般来说，应该设置该掩码，否则信号处理完后，阻塞的系统调用将会返回失败！

SA\_SIGINFO 指示结构体的信号处理函数指针是哪个有效，如果sa\_flags包含该掩码，则sa\_sigactiion指针有效，否则是sa\_handler指针有效。

sa\_mask是一组信号集，**当调用信号处理函数之前会将这组信号集添加到进程的信号屏蔽字中，直到信号处理函数返回。利用sa\_mask参数，我们可以指定一组信号，让我们的信号处理函数不被这些信号打断。与前面的signal()一样，默认还是会把引发信号处理函数的信号，自动的添加到进程的信号屏蔽字中的。**sa\_flags参数，如果有经验的话，我们不难猜到这肯定是一组选项，毕竟身经百战了嘛。那我们就来看看这组选项是什么意思：

|  |  |
| --- | --- |
| sa\_flags | 说明 |
| SA\_INTERRUPT | 由此信号中断的系统调用不会自动重启。 |
| SA\_NOCLDSTOP | 当signum为SIGCHLD时，当因接受一信号的子进程停止或者恢复时，将不会产生此信号（有点绕）.但是子进程终止时，仍会产生此信号。  *(If sig is SIGCHLD, don’t generate this signal when a child process is stopped or resumed as a consequence of receiving a signal.)* |
| SA\_NOCLDWAIT | 当signum为SIGCHLD时，子进程终止时不会转化为僵尸进程。此时调用wait()，则阻塞到所有子进程都终止，才返回-1，errno被视之为ECHILD。 |
| SA\_NODEFER | 捕获该信号的时候，不会在执行信号处理函数之前将该信号自动添加到进程的信号屏蔽字中。 |
| SA\_ONSTACK | 调用信号处理函数时，使用sigaltstack()安装的备用栈。 |
| SA\_RESETHAND | 当捕获该信号时，会在调用信号处理函数之前将信号处理函数设置为默认值SIG\_DFL,并清除SA\_SIGINFO标志。 |
| SA\_RESTART | 被此信号中断的系统调用，会自动重启。 |
| SA\_SIGINFO | 调用信号处理函数时附带了额外的数据要处理，具体见下文。 |

sa\_restorer和名字一样为保留参数，不需要使用。最后我们要看的是\_\_sigaction\_handler，这是一个联合体（当然啦，这是废话）。sa\_handler和sa\_sigaction都是信号处理函数的指针，所以一次只能选择两者中的一个。如果sa\_mask中设置了SA\_SIGINFO位那么就按照void (\*sa\_sigaction)(int, siginfo\_t \*, void \*)的形式的函数调用信号处理函数，否则使用 void (\*sa\_handler)(int)这样的函数。下面我们再来看一看sa\_sigaction这个函数：

void sa\_sigaction(int signum, siginfo\_t\* info, void\* context);

siginfo\_t是一个结构体，其结构和实现相关，我的CentOS7系统上是这样的：

siginfo\_t {  
    int si\_signo; /\* Signal number \*/  
    int si\_errno; /\* An errno value \*/  
    int si\_code; /\* Signal code \*/  
    int si\_trapno; /\* Trap number that caused hardware-generated signal (unused on most architectures) \*/  
    pid\_t si\_pid; /\* Sending process ID \*/  
    uid\_t si\_uid; /\* Real user ID of sending process \*/  
    int si\_status; /\* Exit value or signal \*/  
    clock\_t si\_utime; /\* User time consumed \*/  
    clock\_t si\_stime; /\* System time consumed \*/  
    sigval\_t si\_value; /\* Signal value \*/  
    int si\_int; /\* POSIX.1b signal \*/  
    void \*si\_ptr; /\* POSIX.1b signal \*/  
    int si\_overrun; /\* Timer overrun count; POSIX.1b timers \*/  
    int si\_timerid; /\* Timer ID; POSIX.1b timers \*/  
    void \*si\_addr; /\* Memory location which caused fault \*/  
    long si\_band; /\* Band event (was int in glibc 2.3.2 and earlier) \*/  
    int si\_fd; /\* File descriptor \*/  
    short si\_addr\_lsb; /\* Least significant bit of address (since Linux 2.6.32) \*/  
}

每个字段的含义后边都加了清晰的注释，但是还有一个参数使我们需要特别注意的，其中si\_value字段用来接收伴随着信号发送过来的数据，其类型是一个sigval\_t的联合体，其定义（我的系统是在路径/usr/include/bits/siginfo.h 上）：

# define \_\_have\_sigval\_t 1

/\* Type for data associated with a signal. \*/  
typedef union sigval  
{  
    int     sival\_int;  
    void\* sival\_ptr;  
} sigval\_t;  
#endif

在实际编程中，到底选择sival\_int还是sival\_ptr字段，还是取决于你的应用程序。但是由于指针的作用范围只能在进程的内部，如果发送一个指针到另一个进程一般没有什么实际的意义。

#### sigqueue()

      之前我们提到了发送实时信号时可以附带数据，kill(),raise()等函数的参数注定他们无法附带更多的数据，这里我们要认识一个新的函数sigqueue()专门用于在发送信号的时候，附加传递额外的数据。

int sigqueue(pid\_t pid, int sig, const union sigval value);//Return 0 on success ,or -1 on error

前两个参数和kill()一致，但是不同于kill(),这里不能将pid只能是单个进程，而不像kill()那样丰富的用法。value的类型便是在上边提及的sigval\_t，于是就清晰了：发送进程在这里发送的value在接受进程中通过信号处理函数sa\_sigaction中的siginfo\_t info参数就可以拿到了。

### strace命令

### 信号处理函数中的安全函数

A signal handler function must be very careful, since processing elsewhere may

be interrupted at some arbitrary point in the execution of the program. POSIX

has the concept of "safe function". If a signal interrupts the execution of

an unsafe function, and handler calls an unsafe function, then the behavior of

the program is undefined.

POSIX.1-2004 (also known as POSIX.1-2001 Technical Corrigendum 2) requires an

implementation to guarantee that the following functions can be safely called

inside a signal handler:

\_Exit()

\_exit()

abort()

accept()

access()

aio\_error()

aio\_return()

aio\_suspend()

alarm()

bind()

cfgetispeed()

cfgetospeed()

cfsetispeed()

cfsetospeed()

chdir()

chmod()

chown()

clock\_gettime()

close()

connect()

creat()

dup()

dup2()

execle()

execve()

fchmod()

fchown()

fcntl()

fdatasync()

fork()

fpathconf()

fstat()

fsync()

ftruncate()

getegid()

geteuid()

getgid()

getgroups()

getpeername()

getpgrp()

getpid()

getppid()

getsockname()

getsockopt()

getuid()

kill()

link()

listen()

lseek()

lstat()

mkdir()

mkfifo()

open()

pathconf()

pause()

pipe()

poll()

posix\_trace\_event()

pselect()

raise()

read()

readlink()

recv()

recvfrom()

recvmsg()

rename()

rmdir()

select()

sem\_post()

send()

sendmsg()

sendto()

setgid()

setpgid()

setsid()

setsockopt()

setuid()

shutdown()

sigaction()

sigaddset()

sigdelset()

sigemptyset()

sigfillset()

sigismember()

signal()

sigpause()

sigpending()

sigprocmask()

sigqueue()

sigset()

sigsuspend()

sleep()

sockatmark()

socket()

socketpair()

stat()

symlink()

sysconf()

tcdrain()

tcflow()

tcflush()

tcgetattr()

tcgetpgrp()

tcsendbreak()

tcsetattr()

tcsetpgrp()

time()

timer\_getoverrun()

timer\_gettime()

timer\_settime()

times()

umask()

uname()

unlink()

utime()

wait()

waitpid()

write()

## STDIN\_FILENO的作用及与stdin 的区别

1.STDIN\_FILENO的作用

STDIN\_FILENO属于系统API接口库，其声明为 int 型，是一个打开文件句柄，对应的函数主要包括 open/read/write/close 等系统级调用。

操作系统一级提供的文件API都是以文件描述符来表示文件。STDIN\_FILENO就是标准输入设备（一般是键盘）的文件描述符。

2.区别

1）数据类型不一致：

stdin类型为 FILE\*

STDIN\_FILENO类型为 int

使用stdin的函数主要有：fread、fwrite、fclose等，基本上都以f开头

使用STDIN\_FILENO的函数有：read、write、close等

2）stdin等是FILE \*类型，属于标准I/O，高级的输入输出函数。在<stdio.h>。

STDIN\_FILENO等是文件描述符，是非负整数，一般定义为0, 1, 2，属于没有buffer的I/O，直接调用系统调用，在<unistd.h>。

3）STDIN\_FILENO 是标准输入的文件描述符

详见/usr/include/unistd.h

代码:

/\* Standard file descriptors. \*/

#define STDIN\_FILENO 0 /\* Standard input. \*/

#define STDOUT\_FILENO 1 /\* Standard output. \*/

#define STDERR\_FILENO 2 /\* Standard error output. \*/

4）层次不一样。STDIN 属于标准库处理的输入流，其声明为 FILE 型的，对应的函数前面都有f开头，如fopen/fread/fwrite/fclose 标准库调用等

STDIN\_FILENO属于系统API接口库，其声明为 int 型，是一个打开文件句柄，对应的函数主要包括 open/read/write/close 等系统级调用。

标准库内封装了系统 API 调用，如 fread 内部实现调用 read。

5）操作系统一级提供的文件API都是以文件描述符来表示文件。STDIN\_FILENO就是标准输入设备（一般是键盘）的文件描述符。

6）曾经很纳闷，为什么一个整形fd（STDIN\_FILENO=0）就可以代表一个打开的文件呢？

后来明白了，和我们的手机号是类似的，手机号只是一个9位数的整形，但是在移动通话网络里，它就可以用来区分不同的手机终端。

标准C++一级提供的文件操作函数库都是用FILE\*来表示文件，stdin就是指向标准输入设备文件的FILE\*。

## read函数

使用下面的函数，read读出的字符串包含了在shell输入的回车字符，即当不输入任何字符，直接输入回车时，ret的值为1。如果被信号中断了，当处理完信号之后，之前输入的字符被清空了。

ret = read(STDIN\_FILENO, buf, sizeof(buf) - 1);

if(ret<0)

{

perror("\nread fail.");

break;

}

else

{

printf("\nret: %d.", ret);

}

read函数从打开的设备或文件中读取数据。

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

返回值：成功返回读取的字节数，出错返回-1并设置errno，如果在调read之前已到达文件末尾，则这次read返回0

参数

count

是请求读取的字节数，读上来的数据保存在缓冲区buf中，同时文件的当前读写位置向后移。注意这个读写位置和使用C标准I/O库时的读写位置有可能不同，这个读写位置是记在内核中的，而使用C标准I/O库时的读写位置是用户空间I/O缓冲区中的位置。比如用fgetc读一个字节，fgetc有可能从内核中预读1024个字节到I/O缓冲区中，再返回第一个字节，这时该文件在内核中记录的读写位置是1024，而在FILE结构体中记录的读写位置是1。注意返回值类型是ssize\_t，表示有符号的size\_t，这样既可以返回正的字节数、0（表示到达文件末尾）也可以返回负值-1（表示出错）。

read函数返回时，返回值说明了buf中前多少个字节是刚读上来的。有些情况下，实际读到的字节数（返回值）会小于请求读的字节数count，例如：读常规文件时，在读到count个字节之前已到达文件末尾。例如，距文件末尾还有30个字节而请求读100个字节，则read返回30，下次read将返回0。

从终端设备读，通常以行为单位，读到换行符就返回了。

从网络读，根据不同的传输层协议和内核缓存机制，返回值可能小于请求的字节数，后面socket编程部分会详细讲解。

## sleep函数

sleep函数会被信号中断，那信号处理函数中的sleep会被信号中断吗？会

sleep() makes the calling process sleep until seconds seconds have elapsed or a signal arrives which is not ignored.

头文件：#include <unistd.h>

定义函数：unsigned int sleep(unsigned int seconds);

函数说明：sleep()会令目前的进程暂停, 直到达到参数seconds 所指定的时间, 或是被信号所中断.

返回值：若进程/线程挂起到参数所指定的时间则返回0，若有信号中断则返回剩余秒数。

linux下Sleep()函数：单位为毫秒

如果需要更精确可以用usleep单位为微秒

线程休眠函数:void usleep(unsigned long usec);

linux下delay()函数：

原型为extern void delay(unsigned int msec);

它可以延时msec\*4毫秒，也就是如果想延时一秒钟的话，可以这么用 delay(250);

## ctrl + 按键

ctrl+c和ctrl+z都是中断命令,但是他们的作用却不一样.

ctrl+c是强制中断程序的执行,,进程已经终止。会产生**SIGINT**信号

ctrl+z的是将任务中止（暂停的意思）,但是此任务并没有结束,他仍然在进程中他只是维持挂起的状态,会产生**SIGTSTP**信号。

用户可以使用fg/bg操作继续前台或后台的任务,fg命令重新启动前台被中断的任务,bg命令把被中断的任务放在后台执行.

例如:当你vi一个文件是,如果需要用shell执行别的操作,但是你又不打算关闭vi,因为你得存盘推出,你可以简单的按下ctrl+z,shell会将vi进程挂起~,当你结束了那个shell操作之后,你可以用fg命令继续vi你的文件。

总结:

　　(1) CTRL+Z挂起进程并放入后台

　　(2) jobs 显示当前暂停的进程

　　(3) bg %N 使第N个任务在后台运行(%前有空格)

　　(4) fg %N 使第N个任务在前台运行

　　默认bg,fg不带%N时表示对最后一个进程操作!

ctrl-d 不是发送信号，而是表示一个特殊的二进制值，表示 EOF。

注：在shell中，ctrl-d表示推出当前shell.会产生**SIGHUP**信号。

## time命令

time命令获取命令执行时间，其中包括命令的实际运行时间（real time），以及运行在用户态的时间（user time）和内核态的时间（sys time）。 它的使用方法是在待执行的命令前加上time即可。

$ time foo

real 0m0.003s

user 0m0.000s

sys 0m0.004s

$

Real指的是实际经过的时间，User和Sys指的是该进程使用的CPU时间。

1. Real是墙上时间(wall clock time)，也就是进程从开始到结束所用的实际时间。这个时间包括其他进程使用的时间片和进程阻塞的时间（比如等待I/O完成）。

2. User指进程执行用户态代码（核心之外）所使用的时间。这是执行此进程所消耗的实际CPU时间，其他进程和此进程阻塞的时间并不包括在内。

3. Sys指进程在内核态消耗的CPU时间，即在内核执行系统调用所使用的CPU时间。

那么，为什么进程开始到结束所经过的时间会比进程所消耗的用户时间和系统时间(user time + sys time)小呢？

User+Sys为进程所使用的实际CPU时间。注意，如果有多个线程，User+Sys的时间有可能大于Real时间。同时，User和Sys时间包括子进程所使用的时间。

time命令的输出数据是由几个不同的系统调用得来的。User time和Sys time从wait(2)或times(2)系统调用（依赖不同的系统）得来。Real time是由gettimeofday(2)中结束时间和起始时间相减得到。不同的操作系统还可能有其他的信息，比如time可以记录上下文切换的次数。

在多处理器的系统上，一个进程如果有多个线程或者有多个子进程可能导致Real time比CPU time（User + Sys time）要小，这是因为不同的线程或进程可以并行执行。