## 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()

{

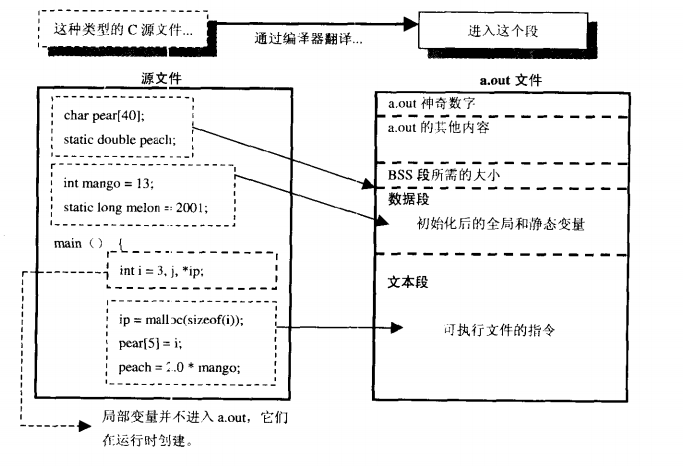
int i = 3, j, \*ip;

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：

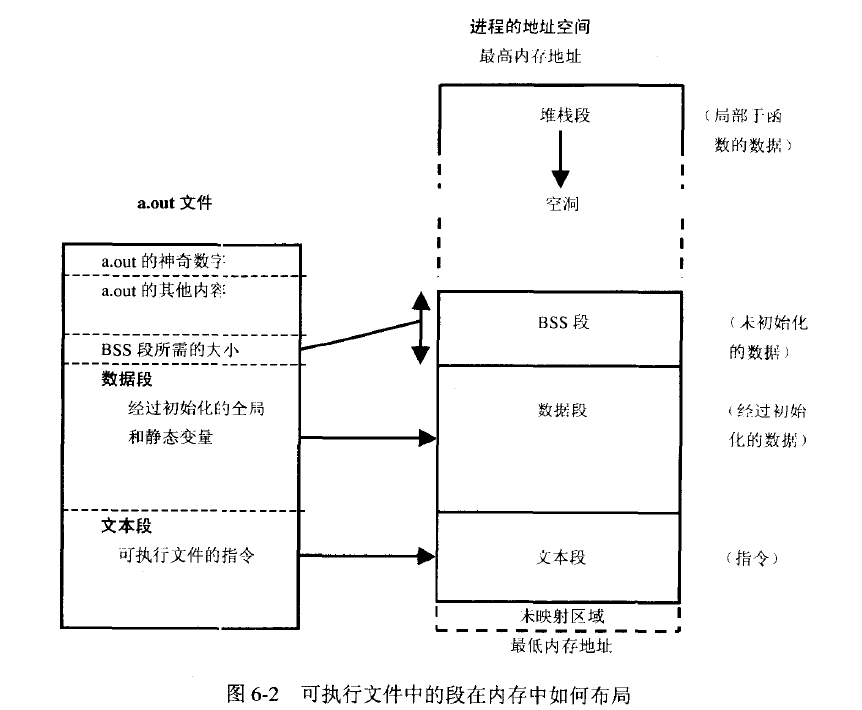
1. bss段只记录无初值全局变量，局部变量不影响bss，且bss段不占用a.out空间，只记录大小。
2. 数据段只记录有初值全局变量，局部变量不影响data段，data段占用a.out空间。
3. 局部变量的变化会影响text段，因为属于运行时的数据。
4. text段是最容易受优化措施影响的段。
5. a.out文件的大小受调试状态下编译的影响(gcc -g)，但是段不受影响。

## 运行时数据

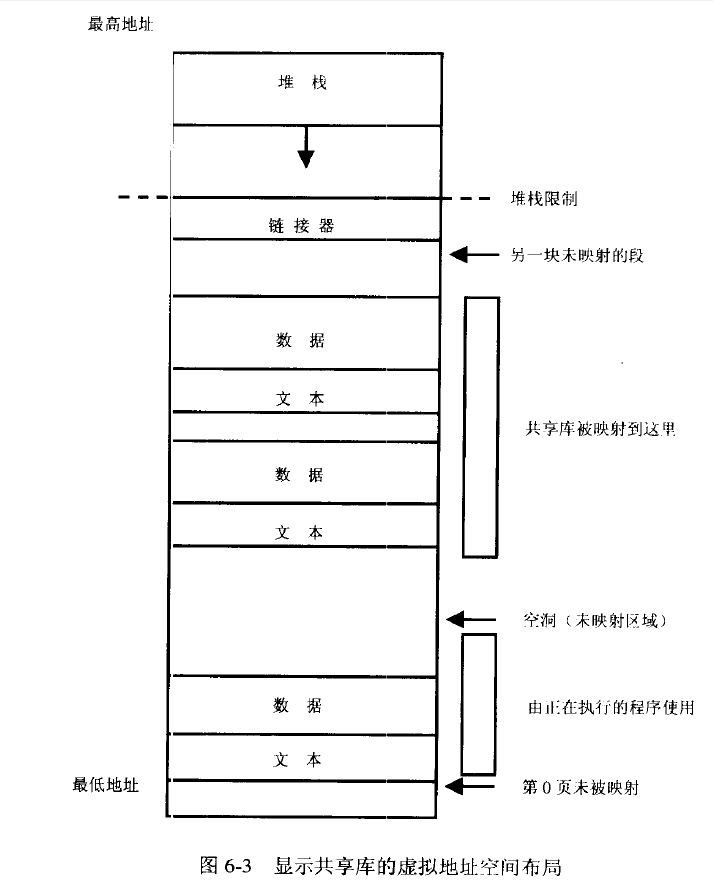
### 操作系统对a.out的操作

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.

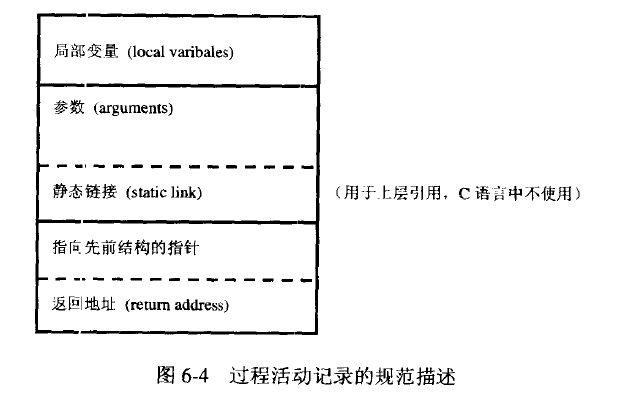
The stack f2 is near 0x7ffd909a94a0.

The bss addr is near 0x601048.

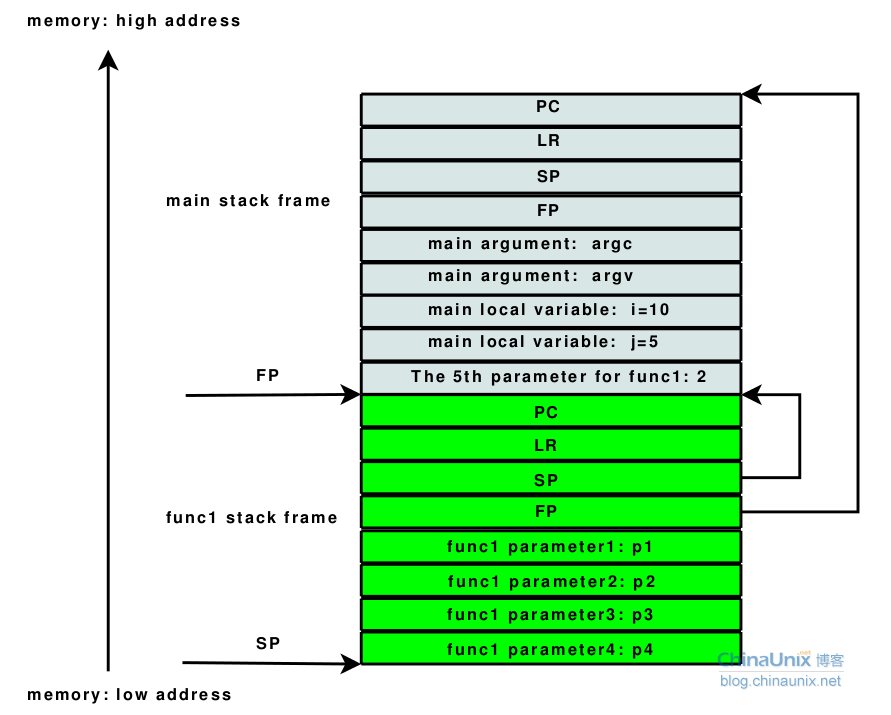
The data addr is near 0x601040.

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");

}

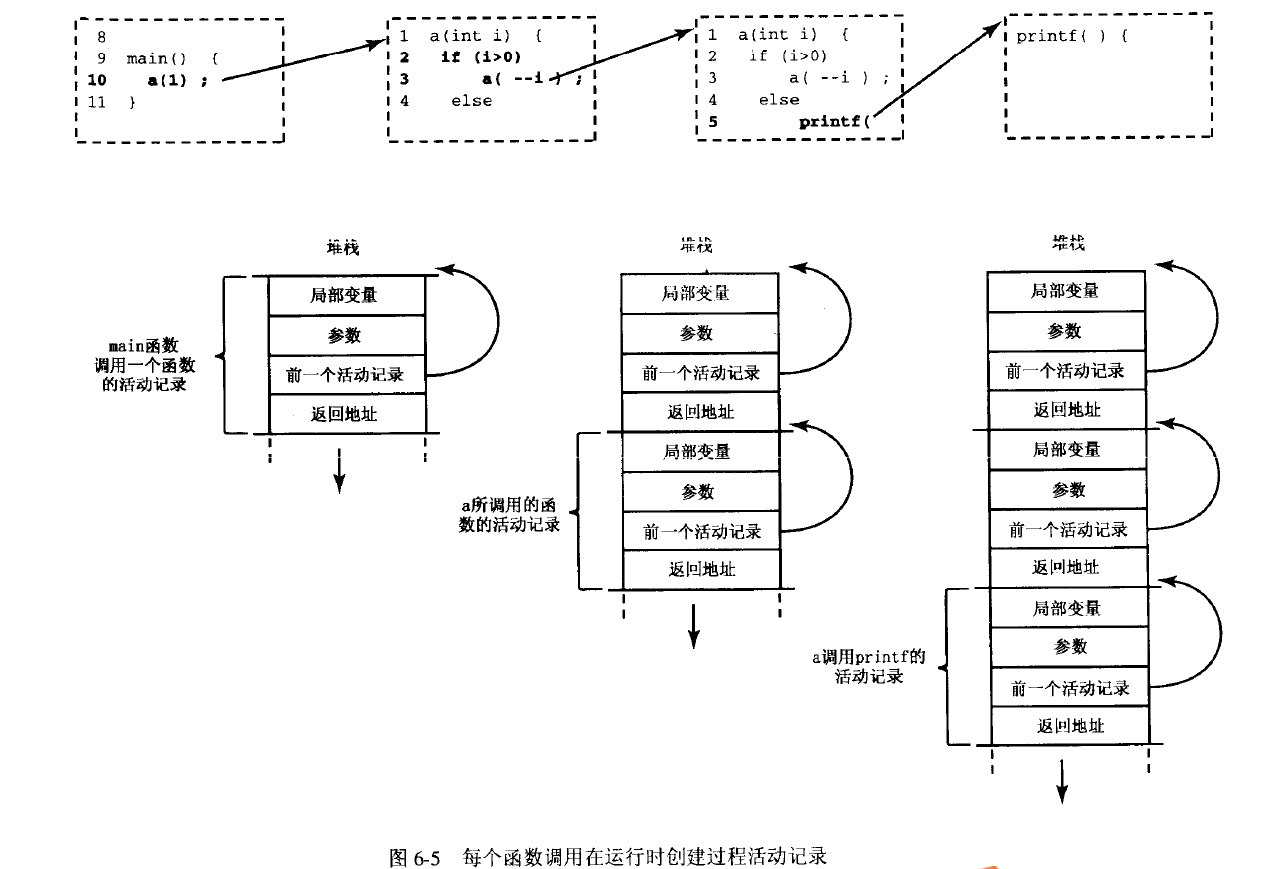
}

void main()

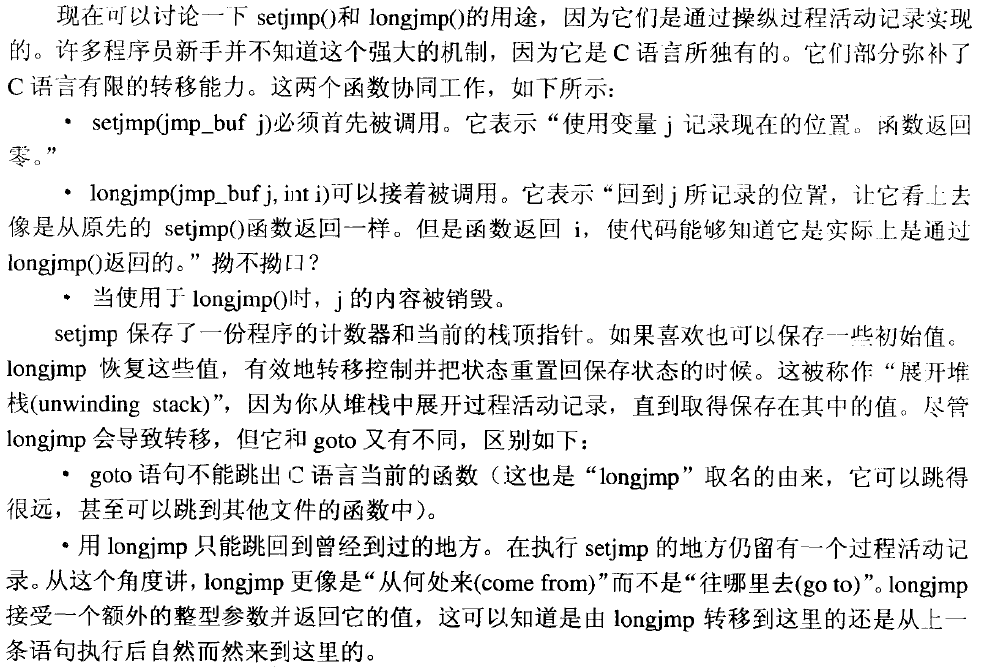
{

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。

### 信号机制

在这一节中，我们将介绍内核如何实现信号机制。

即内核如何向一个进程发送信号、进程如何接收一个信号、进程怎样控制自己对信号的反应、内核在什么时机处理和怎样处理进程收到的信号。

**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。

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)。

### 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。

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。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。

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()函数

最后一个发送信号的函数是raise(),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。

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。

### 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。

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)

### 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.*

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  
};

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