Linux征途

vivieu.blog.chinaunix.net

业精于勤，荒于嬉；行成于思，毁于随。

vivieu

博客访问： 409975
博文数量： 238
博客积分： 3997
博客等级： 少校
技术积分： 2136
用 户 组： 普通用户
注册时间： 2010-06-19 17:26

加关注 短消息
论坛 加好友

个人简介

Never save something for a special occasion. Every day in your life is a special occasion.

gcc基础

2011-07-08 13:42:11

分类：

原文地址：gcc基础 作者：chengyake

1.本文不是教程，只是描述c语言（gcc环境），编译器，连接器，加载器，at&t汇编，ia32一些相关知识和笔记，很多需要深入的地方需要大家寻找相关的资料学习。如果发现错误，请留言或通知我jinglexy at yahoo dot com dot cn，这个是我的msn。打字不易，请转载时保留作者：http://www.cppblog.com/jinglexy

2.gcc安装的各个部分：

	g++	c++编译器，链接时使用c++库
	gcc	c编译器，链接时使用c库
	cc1	实际的c编译器
	cc1plus	实际的c++编译器
	collect2	使用collect2产生特定的全局初始化代码，后台处理是传递参数给ld完成实际的链接工作。
	crt0.o	初始化和结束代码
	libgcc	平台相关的库

gcc安装需要的文件：

gcc-core-3.4.6.tar.gz2 gcc核心编译器，默认只包含c编译器

gcc-g++-3.4.6.tar.bz2 g＋＋编译器

gcc-testsuite-3.4.6.tar.bz2 测试套件

./configure && make && make install

3.binutils安装的各个部分

	as	gnu汇编工具
	gprof	性能分析工具
	ld	gnu链接器
	make	
	objcopy	目标文件从二进制格式翻译或复制到另一种
	objdump	显示目标文件的各种信息
	strings	显示文件的字符串
	strip	去除符合表
	readelf	分析elf并显示信息

链接器可以读写各种目标文件中的信息，通过BFD（binary file descriptor）提供的工具实现，BFD定义了类似a.out, elf, coff等目标文件的格式。

4.gcc预处理程序

1)define指令

#可将传递的宏字符串化

##将两个名字连接成一个（注意不是连接成字符串）

例：#define TEST(ARGTERM) \

printf(“the term “ #ARGTERM “is a string\n”)

使用__VA_ARGS__定义可变参数宏

例：#define err(...)　　fprintf(stderr, __VA_ARGS)

　　err (“%s %d\n”, “error code is”, 48);

为了消除无参数时的逗号，可以用下面方法定义：

　　# define err(...)　　fprintf(stderr, ##__VA_ARGS)

　　一种等同的方法是：

　　#define dprintf(fmt, arg...)　　printf(fmt, ##arg)

其他例：#define　PASTE(a, b)　　a##b

2)error 和 warning指令

#error “y here? bad boy!”

3)if, elif, else, endif指令

　　支持的运算符：加减乘除，位移，&&，||，!等

　　示例：#if defined (CONFIG_A) || defined (CONFIG_B)

　　.....

　　#endif

4)gcc预定义宏

	__BASE_FILE__	完整的源文件名路径
	__cplusplus	测试c++程序
	__DATE__	
	__FILE__	源文件名
	__func__	替代__FUNCTION__，__FUNCTION__以被GNU不推荐使用
	__TIME__	
	__LINE__	
	__VERSION__	gcc版本

5)几个简单例子：

例1：

```
#define    min(X,    Y)    \
        (__extension__ ({typeof (X) __x = (X), __y = (Y);    \
        (__x < __y) ? __x : __y; }))
#define    max(X,    Y)    \
       (__extension__ ({typeof (X) __x = (X), __y = (Y);    \
       (__x > __y) ? __x : __y; }))
```

这样做的目的是消除宏对X，Y的改变的影响，例如：result = min(x++, --y); printf(x, y);

补充：圆括号定义的符合语句可以生成返回值，例：

```
result = ({ int a = 5;
            int b;
            b = a + 3;
            });                    将返回8
```

例2：

```
#define dprintfbin(buf, size)     do{   int i;                    \
                printf("%s(%d)@",                                 \
                        __FUNCTION__, __LINE__);                  \
                for(i = 0; i < size - 1; i++){                    \
                        if(0 == i % 16)                           \
                                printf("\n");                     \
                        printf("0x%02x ", ((char*)buf)[i]);   \
                }                                                 \
                printf("0x%02x\n", ((char*)buf)[i]);              \
}while(0)
```

这个比较简单，不用解释了

例3：

```
#ifdef __cplusplus
extern "C"{
#endif
int foo1(void);
int foo2(void);
#ifdef __cplusplus
}
#endif
```

作用：在c＋＋程序中使用c函数及库，c＋＋编译程序时将函数名粉碎成自己的方式，在没有extern的情况下可能是_Z3_foo1，_Z3_foo2将导致连接错误，这里的extern表示在连接库时，使用foo1，foo2函数名。

5.gcc编译的一些知识

```
gcc   -E   hello.c   -o   hello.i                    只预处理
gcc   -S   hello.c   -o   hello.s                    只编译
gcc   -c   -fpic   first.c   second.c
```

编译成共享库：-fpic选项告诉连接器使用got表定位跳转指令，使加载器可以加载该动态库到任何地址（具体过程可在本文后面找到）

6.gcc对c语言的扩展

```
void fetal_error()   __attribute__(noreturn);  声明函数：无返回值
__attribute__((noinline)) int foo1(){……}定义函数：不扩展为内联函数
int getlim()   __attribute__((pure, noinline));声明函数：不内联，不修改全局变量
void mspec(void)   __attribute__((section(“specials”)));声明函数：连接到特定节中
```

补充：除非使用-O优化级别，否则函数不会真正的内联。

其他属性：

函数	always_inline	
函数	const	同pure
函数	constructor	加入到crt0调用的初始化函数表
函数	deprecated	无论何时调用函数，总是让编译器警告
函数	destructor	
函数	section	放到命名的section中，而不是默认的.text
变量	aligned	分配该变量内存地址时对齐属性，例： int value __attribute__((aligned(32)));
变量	deprecated	无论何时引用变量，总是让编译器警告

变量	packed	使数据结构使用最小的空间，例如：
		typedef struct zrecord{
		char a;
		int b
		__attribute((packed));
		}zrecord_t;
		变量b在内存中和a没有空隙

变量	section	同上，例： int trigger __attribute__((section(“domx”))) = 0;
类型	aligned	同上，例： struc blockm{ char j[3];

		}__attribute__((aligned(32)));
类型	deprecated	同上
类型	packed	同上

gcc内嵌函数：

```
void *__builtin_return_address(unsigned int level);
void *__builtin_frame_address(unsigned int leve);
```

以上两个函数可以用于回溯函数栈，如果编译器优化成noframe呢，谁愿意验证一下？

gcc使用__asm__, __typeof__, __inline__替代asm, typeof, inline。-std和-ansi会使后者失去功能。

标识符局部化，使用__label__标签：

```
int main(……){
            {
                                    __label__ jmp1;
                                    goto jmp1;
            }
            goto jmp1;                              /* 错误：jmp1未定义 */
}
```

typeof的一些技巧：

	char *chptr	a char point
	typeof (*chptr) ch;	a char
	typeof (ch) *chptr2;	a char point
	typeof(chptr) chparray[10];	ten char pointers
	typeof(*chptr) charray[10];	ten char
	typeof (ch) charray2[10];	ten chars

7.objdump程序

	-a		文档头文件信息
	-d		可执行代码的反汇编
	-D		反汇编可执行代码及数据
	-f		完整文件头的内容
	-h		section表
	-p		目标格式的文件头内容

调试器呢？网上的gdb教程已足够的多，不再画蛇添足了。

8.平台IA32的一些知识

指令码格式：

指令前缀（0～4字节）	操作码（1～3字节）	可选修饰符（0～4字节）	可选数据元素（0～4字

指令前缀：较重要的有内存锁定前缀（smp系统中使用）

操作码：ia32唯一必须的部分

修饰符：使用哪些寄存器，寻址方式，SIB字节

数据元素：静态数值或内存位置

ia32比较重要的技术：指令预取，解码管线，分支预测，乱序执行引擎

（网络上可以找到很多相关的文章）

通用寄存器（8个32位）：eax, ebx, ecx, edx, esi, edi, esp, ebp

端寄存器（6个16位）：cs, ds, ss, es, fs, gs

指令指针（1个32位）：eip

浮点寄存器（8个80位）：形成一个fpu堆栈

控制寄存器（5个32位）：cr0，cr1，cr2，cr3，cr4

较重要的是cr0：控制操作模式和处理器状态

cr3：内存分页表描述寄存器

调试寄存器（8个32位）：

标识寄存器（1个32位）：状态，控制，系统（共使用17位）：陷阱，中断，进位，溢出等

说明：mmx使用fpu堆栈作为寄存器，sse，sse2，sse3没有寄存器，只提供相关的指令功能。

9.gas汇编工具：as（at&t风格）语法说明

使用$标识立即数
再寄存器前面加上%
源操作数在前，目标操作数在后
使用$获取变量地址
长跳转使用：ljmp $section, $offset

一个简单的汇编语言程序框架：

```
.section .data
                ……
.section .bss
                ……
.section .text
.globl _start
_start:
        ……
```

范例：

```
#cpuid2.s View the CPUID Vendor ID string using C library calls
.section .datatext
output:
       .asciz "The processor Vendor ID is '%s'\n"
.section .bss
       .lcomm buffer, 12
.section .text
.globl _start
_start:
       movl $0, %eax
       cpuid
       movl $buffer, %edi
       movl %ebx, (%edi)
       movl %edx, 4(%edi)
       movl %ecx, 8(%edi)
       pushl $buffer
       pushl $output
       call printf
       addl $8, %esp
       pushl $0
   call exit
```

伪指令说明：

data	.ascii	定义字符串，没有\0结束标记
data	.asciz	有\0结束标记
data	.byte	字节
data	.int	32位

data	.long	32位
data	.shot	16位
bss	.lcomm	对于上面的例子是声明12字节的缓冲区，l标识local，仅当前汇编程序可用
bss	.comm	通用内存区域
data/text	.equ	.equ LINUX_SYS_CALL, 0x80 movl $ LINUX_SYS_CALL, %eax 说明：equ不是宏而是常量，会占据数据/代码段空间

指令集说明：

<table>
<tr><td></td><td>movb/movw/movl</td><td colspan="3"></td></tr>
<tr><td></td><td>cmov</td><td colspan="3">根据cf，of，pf，zf等标识位判断并mov</td></tr>
<tr><td></td><td>xchg</td><td colspan="3">操作时会lock内存，非常耗费cpu时间</td></tr>
<tr><td></td><td>bswap</td><td colspan="3">翻转寄存器中字节序</td></tr>
<tr><td></td><td>xadd</td><td colspan="3"></td></tr>
<tr><td></td><td>pushx，popx</td><td colspan="3"></td></tr>
<tr><td></td><td>pushad，popad</td><td colspan="3"></td></tr>
<tr><td></td><td>jmp</td><td colspan="3"></td></tr>
<tr><td></td><td>call</td><td colspan="3"></td></tr>
<tr><td></td><td>cmp</td><td colspan="3"></td></tr>
<tr><td></td><td>jz/jb/jne/jge</td><td colspan="3"></td></tr>
<tr><td></td><td>loop</td><td colspan="3"></td></tr>
<tr><td></td><td>addb/addw/addl</td><td colspan="3"></td></tr>
<tr><td></td><td>subb/subw/subl</td><td colspan="3"></td></tr>
<tr><td></td><td>dec/inc</td><td colspan="3"></td></tr>
<tr><td rowspan="4"></td><td rowspan="4">mulb/muw/mull
无符号乘法</td><td>源操作数长度</td><td>目标操作数</td><td>目标位置</td></tr>
<tr><td>8位</td><td>al</td><td>ax</td></tr>
<tr><td>16位</td><td>ax</td><td>dx:ax</td></tr>
<tr><td>32位</td><td>eax</td><td>edx:eax</td></tr>
<tr><td></td><td>imul有符合乘法</td><td colspan="3"></td></tr>
<tr><td rowspan="4"></td><td rowspan="4">divb/divw/divl
无符合除法
（被除数在eax中，除数在指令中给出）</td><td>被除数</td><td>被除数长</td><td>商</td><td>余数</td></tr>
<tr><td>ax</td><td>16位</td><td>al</td><td>ah</td></tr>
<tr><td>dx:ax</td><td>32位</td><td>ax</td><td>dx</td></tr>
<tr><td>edx:eax</td><td>64位</td><td>eax</td><td>edx</td></tr>
<tr><td></td><td>idiv有符合除法</td><td colspan="3"></td></tr>
<tr><td></td><td>sal/shl/sar/shr</td><td colspan="3">移位</td></tr>
<tr><td></td><td>rol/ror/rcl/rcr</td><td colspan="3">循环移位</td></tr>
<tr><td></td><td>leal</td><td colspan="3">取地址：leal output, %eax
等同于：movl $output, %eax</td></tr>
<tr><td></td><td>rep</td><td colspan="3">rep movsb 执行ecx次</td></tr>
<tr><td></td><td>lodsb/lodsw/lodsl
stosb/stosw/stosl</td><td colspan="3">取存内存中的数据</td></tr>
</table>

gas程序范例（函数调用）：

文件1：area.s定义函数area

```
# area.s - The areacircumference function
.section .text
.type area, @function
.globl area
area:
     pushl %ebp
     movl %esp, %ebp
     subl $4, %esp
     fldpi
     filds 8(%ebp)
     fmul %st(0), %st(0)
     fmulp %st(0), %st(1)
     fstps -4(%ebp)
     movl -4(%ebp), %eax
     movl %ebp, %esp
     popl %ebp
     ret
```

文件2：functest4.s调用者

```
# functest4.s - An example of using external functions
.section .data
precision:
     .byte 0x7f, 0x00
.section .bss
     .lcomm result, 4
.section .text
.globl _start
_start:
     nop
     finit
     fldcw precision

     pushl $10
     call area
     addl $4, %esp
     movl %eax, result

     pushl $2
     call area
     addl $4, %esp
     movl %eax, result

     pushl $120
     call area
     addl $4, %esp
     movl %eax, result

     movl $1, %eax
     movl $0, %ebx
     int $0x80
```

10.读连接器和加载器的一些笔记，感谢原作者colyli at gmail dot com，看了他翻译的lnl及写的一个os，受益匪浅。如果不是很深入的研究连接器和加载器的话，了解一些原理就足够了。举个例子说明吧：

```
 1 #include <unistd.h>
 2 #include <stdlib.h>
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6 int a = 1;
 7 int main()
 8 {
 9                 printf("value: %d\n", a);
10
11                 return 0;
12 }
```

编译指令：gcc -c hello.c -o hello.o　　汇编

gcc -o hello hello.o　　编译

objdump -d hello.o　　反汇编目标文件

objdump -d hello　　反汇编可执行文件

比较两端结果：

objdump -d hello.o	objdump -d hello
00000000 <main>: 0: 55 push %ebp 1: 89 e5 mov %esp,%ebp 3: 83 ec 08 sub $0x8,%esp 6: 83 e4 f0 and $0xfffffff0,%esp 9: b8 00 00 00 00 mov $0x0,%eax e: 83 c0 0f add $0xf,%eax 11: 83 c0 0f add $0xf,%eax 14: c1 e8 04 shr $0x4,%eax 17: c1 e0 04 shl $0x4,%eax 1a: 29 c4 sub %eax,%esp 1c: 83 ec 08 sub $0x8,%esp 1f: ff 35 00 00 00 00 pushl 0x0 25: 68 00 00 00 00 push $0x0 2a: e8 fc ff ff ff call 2b <main+0x2b> 2f: 83 c4 10 add $0x10,%esp 32: b8 00 00 00 00 mov $0x0,%eax 37: c9 leave 38: c3 ret	08048368 <main>: 8048368: 55 push %ebp 8048369: 89 e5 mov %esp,%ebp 804836b: 83 ec 08 sub $0x8,%esp 804836e: 83 e4 f0 and $0xfffffff0,%esp 8048371: b8 00 00 00 00 mov $0x0,%eax 8048376: 83 c0 0f add $0xf,%eax 8048379: 83 c0 0f add $0xf,%eax 804837c: c1 e8 04 shr $0x4,%eax 804837f: c1 e0 04 shl $0x4,%eax 8048382: 29 c4 sub %eax,%esp 8048384: 83 ec 08 sub $0x8,%esp 8048387: ff 35 94 95 04 08 pushl 0x8049594 804838d: 68 84 84 04 08 push $0x8048484 8048392: e8 19 ff ff ff call 80482b0 <printf@plt> 8048397: 83 c4 10 add $0x10,%esp 804839a: b8 00 00 00 00 mov $0x0,%eax 804839f: c9

```
        leave
  80483a0: c3
ret
  80483a1: 90
    nop
  80483a2: 90
           nop
  80483a3: 90
    nop
```

简单说明：由于程序运行时访问内存，执行跳转都需要确切的地址。所以汇编处理的目标文件里面没有包含，而是把这个工作放到连接器中：即定位地址。

当程序需要动态链接到某个库上时，使用该库的got表动态定位跳转即可。

具体可以看colyli大侠的《链接器和加载器Beta 2》，及《从程序员角度看ELF》

11.连接器脚本ld—script（相关内容来自《GLD中文手册》）

ld --verbose查看默认链接脚本

ld把一定量的目标文件跟档案文件连接起来,并重定位它们的数据,连接符号引用.一般在编译一个程序时,最后一步就是运行ld。

实例1：

```
SECTIONS
{
        . = 0x10000;
        .text : { *(.text) }
        . = 0x8000000;
        .data : { *(.data) }
        .bss : { *(.bss) }
}
```

注释：“.”是定位计数器，设置当前节的地址。

实例2：

```
floating_point = 0;
     SECTIONS
     {
. = ALIGN(4);
         .text :
             {
                 *(.text)
                   _etext = .;
    PROVIDE(etext = .);
         }

      . = ALIGN(4);
            _bdata = (. + 3) & ~ 3;
            .data : { *(.data) }
        }
```

注释：定义一个符合_etext，地址为.text结束的地方，注意源程序中不能在此定义该符合，否则链接器会提示重定义，而是应该象下面这样使用：

```
extern char _etext;
```

但是可以在源程序中使用etext符合，连接器不导出它到目标文件。

实例3：

```
            SECTIONS {
                  outputa 0x10000 :
                      {
                      all.o
                      foo.o (.input1)
                      }
                  outputb :
                      {
```

```
                foo.o (.input2)
                foo1.o (.input1)
                }
            outputc :
                {
                *(.input1)
                *(.input2)
                }
        }
```

这个例子是一个完整的连接脚本。它告诉连接器去读取文件all.o中的所有节，并把它们放到输出节outputa的开始位置处，该输出节是从位置0x10000处开始的。从文件foo.o中来的所有节.input1在同一个输出节中紧密排列。 从文件foo.o中来的所有节.input2全部放入到输出节outputb中，后面跟上从foo1.o中来的节.input1。来自所有文件的所有余下的.input1和.input2节被写入到输出节outputc中。

示例4：连接器填充法则：

```
SECTIONS { .text : { *(.text) } LONG(1) .data : { *(.data) } }
```

错误

```
SECTIONS { .text : { *(.text) ; LONG(1) } .data : { *(.data) } }
```

正确

示例5：VMA和LMA不同的情况

```
SECTIONS
    {
    .text 0x1000 : { *(.text) _etext = . ; }
    .mdata 0x2000 :
        AT ( ADDR (.text) + SIZEOF (.text) )
        { _data = . ; *(.data); _edata = . ;  }
    .bss 0x3000 :
        { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

程序：

```
extern char _etext, _data, _edata, _bstart, _bend;
char *src = &_etext;
char *dst = &_data;

/* ROM has data at end of text; copy it. */
while (dst &lt; &_edata) {
    *dst++ = *src++;
}

/* Zero bss */
for (dst = &_bstart; dst&lt; &_bend; dst++)
    *dst = 0;
```

示例6：linux-2.6.14/arch/i386/kernel $ vi vmlinux.lds.S

linux内核的链接脚本，自行分析吧，有点复杂哦。

上一篇：gcc编译环境
下一篇：date常用用法

0

相关热门文章

Java的String和StringBuffer和...
大数据处理工具之Hive安装配置...
signal函数、sigaction函数及...
Ubuntu系统编译安装Linux新内...
实战Django：Rango Part6
A sample .exrc file for vi e...
IBM System p5 服务器 HACMP ...
busybox的httpd使用CGI脚本(Bu...
Solaris PowerTOP 1.0 发布
For STKMonitor
求教这个命令什么意思，我是新...
sed -e "/grep/d" 是什么意思...
谁能够帮我解决LINUX 2.6 10...
现在的博客积分不会更新了吗？...
shell怎么读取网页内容...

给主人留下些什么吧！~~

评论热议

请登录后评论。

登录 注册