杂项记录

一些常识

屈庆磊

quqinglei@icloud.com

2013 年 *5* 月

目录

1 pkg-config

```
gcc -o test test.c `pkg-config --libs --cflags libpng`
```

2 线程和栈

每个线程都有独立的栈空间，这是必然的要不就乱套了，这个问题想想也会明白，因为每个线程其实都是一个单独的调用单位，在 Linux 下他们都属于较为轻量级的进程，每个线程里的函数都是顺序执行的，顺序执行就需要栈的支持，如果线程之间共用同一个栈那就乱套了，这个问题可以想想汇编语言和 c 语言的关系，特别是调用函数时的压栈，以及在函数执行时的出栈过程，你就想到了。

3 main 函数和 int

在新的规范中，main 函数必须是 int 类型的，这是规定，从下面的几个例子中我们能看出个所以然来:

```
int main()
{
        return 0;
        /* if return -1 you will get 255 after run ./a.out, use
        echo $? to get the return value */
}

/* gcc a.c
$ ./a.out
$ echo $?
0
*/
```

4 函数和汇编

```
CODE
void funtction1() {
        int A = 10;
        A += 66;
}

compiles to...
funtction1:
1        pushl %ebp #
2        movl %esp, %ebp #,
3        subl $4, %esp #,
4        movl $10, -4(%ebp) #, A
5        leal -4(%ebp), %eax #,
```

```
6         addl $66, (%eax) #, A
7         leave
8         ret
Explanation:
1. push ebp
2. copy stack pointer to ebp
3. make space on stack for local data
4. put value 10 in A (this would be the address A has now)
5. load address of A into EAX (similar to a pointer)
6. add 66 to A
... don't think you need to know the rest
```

(1) 把 ebp 的值压入到栈中

(2) 把栈指针复制到 ebp

(3) [tbd]

5 机器相关

5.1 关于代码段、数据段、堆、栈段、BSS 段

代码段 通常用来存放程序执行代码，大小在编译后就已经确定，此内存区域通常属于只读，某些架构也允许代码段可写，也就是说允许修改程序。在代码段中也有可能包含一些只读的常量变量，例如字符串常量。

数据段 通常用来存放程序中已经初始化的全局变量的一块内存区域，数据段属于静态内存分配。

堆 堆是用来存放进程运行中被动态分配的内存段，它的大小不确定，可以使用 **malloc()/calloc()** 函数去向操作系统索取，可以使用 **free()** 函数去释放。

栈 是用户用来存放临时创建的局部变量，也就是说我们函数括号内定义的变量 [1]除此之外，在函数被调用时其参数也会被压入发起调用的进程栈中，并且待到调用结束后函数的返回值也会被存放回栈中。由于栈的先进先出的特点，所以它特别适合用来保存或者恢复调用现场。

BSS 通常是指用来存放程序中未初始化的全局变量的一块内存区域，属于静态内存分配。

5.2 一个栈最大可以为多少，为什么，栈的大小可以设置么

决定栈大小的根本是 **SP** 的大小，也就是堆栈寄存器的大小，在 **8086** 中 **SP** 的变化范围是 **0-FFFFH**[2]栈的大小在一些系统上是可以设置的，在 Linux 上可以通过 **ulimit -a** 查看栈的默认大小，也可以通过如 **ulimit -s 8192** 的方法去设置栈的大小，当参数大于机器本身的栈指针寄存器的大小时，系统会提示设置为无限的。当然栈的大小不可能是无限的它最终还是受 **SP** 的影响。在 32 位机器上，它的范围为 **0x00 - 0xFFFFFFFF** 在 64 位机器上范围为 **0x00 - 0xFFFFFFFFFFFFFFFF**

[1]**static** 声明的变量除外，它意味着在数据段中存放变量

[2]因为 8086 机器的栈顶指针寄存器长度为 16 位

```
# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7899
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 7899
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited

# ulimit -s 32767
```

5.3 关于栈的定义摘自维基百科

http://en.wikipedia.org/wiki/Data_segment The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. A "stack pointer" register tracks the top of the stack; it is adjusted each time a value is "pushed" onto the stack. The set of values pushed for one function call is termed a "stack frame". A stack frame consists at minimum of a return address. Automatic variables are also allocated on the stack. The stack area traditionally adjoined the heap area and they grew towards each other; when the stack pointer met the heap pointer, free memory was exhausted. With large address spaces and virtual memory techniques they tend to be placed more freely, but they still typically grow in opposite directions. On the standard PC x86 architecture the stack grows toward address zero, meaning that more recent items, deeper in the call chain, are at numerically lower addresses and closer to the heap. On some other architectures it grows the opposite direction.

下面我们在 **Linux C** 代码里尝试使用栈大小限制，看代码

5.4 段错误，栈溢出

```c
/* stackOverFlow.c */
#include <stdio.h>

double fact(double n)
{
        if (n <= 0) return 1;
        else return n * fact(n - 1);
}

int main()
{
        printf("fact(5) = %g\n", fact(5));
        printf("fact(10) = %g\n", fact(10));
        printf("fact(100) = %g\n", fact(100));
        printf("fact(1000) = %g\n", fact(1000));
        printf("fact(1000000) = %g\n", fact(1000000));
}
```

下面是在 Linux 32bit 的测试结果：

```
# gcc stackOverFlow.c -o stackOverFlow
fact(5) = 120
fact(10) = 3.6288e+06
fact(100) = 9.33262e+157
fact(1000) = inf
Segmentation fault (core dumped)

/* stackOverFlowSimple.c */
int main()
{
        main();
}
```

5.5 ♠ 为什么软中断使用寄存器状态

个人认为，软中断模仿硬件中断，而检查寄存器状态查找中断向量表的速度应该最为快速，关于到底时为什么我至今没有太为理解，QUESTION 存在疑问。下面是一些参数的存放位置：

- 0st 参数 EAX，一般用来存放系统调用号码
- 1st EBX，第一个参数的存放位置
- 2nd ECX，第二个参数的存放位置
- 3rd EDX，第三个参数的存放位置
- 4th ESI，第四个参数的存放位置
- 5th EDI，第五个参数的存放位置

如果想查看具体内容，请参考 *Linux System Calls for HLA Programmers*

5.6 大端与小端测试

现在的因特尔系列的计算机基本都是小端，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节。下面的代码可以测试机器是大端还是小端:

```
1  #include <stdio.h>
2
3  int bigendian()
4  {
5          int num = 0x12345678;
6          char *low = (char*) &num;
7          return (*low == 0x78) ? 0 : 1;
8  }
9
10 int main()
11 {
12         if (bigendian())
13                 printf("Big endian\n");
14         else
15                 printf("Small endian\n");
16
17         return 0;
18 }
```

0x12345678 转换为二进制数据则为 29 位的 0b10010001101000101011001111000，占 8 个字节。

第五行代码 取 i 的地址

第六行代码 查看第一个字节存放的是否为低位数据，如果为 0x78 则验证了系统为小端编码，反之则验证为大端编码。

下面有一个例子，可以看得比较明白，如果是低端编码，我们可以看到在低位内存地址存放的是低位数据，而高位地址则存放的为高位数据。

```
1  #include <stdio.h>
2
3  int main()
4  {
5          int i = 0;
6          int num = 0x12345678;
7          char *low = (char*) &num;
8          for (i = 0; i < 4; i++) {
9                  printf("address %p, num 0x%x\n", low + i,  *(low + i));
10         }
11
12         return 0;
13 }
```

下面是程序在因特尔平台的输出样本

```
address 0xbfc30644, num 0x78
address 0xbfc30645, num 0x56
address 0xbfc30646, num 0x34
address 0xbfc30647, num 0x12
```

6 安装相关

6.1 在 GNU/Linux 下安装 dos 模拟器，以及汇编工具

下面是安装步骤:

(1) 安装 dosbox，通过执行命令 *apt-get install dosbox*，这个软件在 windows 上也有相应的版本。地址是：www.dosbox.com，可以直接从官网下载。

(2) 下载 masm 汇编工具，我把它备份到了我的 dropbox 上，帐号为：quqinglei@icloud.com，密码是那个我记忆中最熟悉的那个，最简单的那个，哈哈，不告诉你。在网上能搜到，如果搜不到可以发给我邮件。

(3) 安装完 dosbox 后，可以通过其内置的 mount 命令，把本地的文件夹 mount 到模拟的 dos 机器上。方法如下:

(a) 创建一个文件夹，如/home/lei/8086/

(b) 在 dosbox 的文本界面里输入 *mount c /home/lei/8086*

(c) 切换到 *C:* 然后用 *dir* 查看是否已经挂载成功

7 调试示例

7.1 断点与寄存器调试

7.1.1 代码

```
;FileName: hello.asm
;Compile: nasm -f elf hello.asm -g -F stabs
;        gcc -o hello hello.o -g

section .text
global main
main:
        mov eax, 4 ;4号调用
        mov ebx, 1 ;ebx送1表示stdout
        mov ecx, msg ;字符串的首地址送如ecx
        mov edx, 14 ;字符串的长度送入edx
        int 80h ;系统调用
        mov eax, 1 ;1号调用，表示退出
        int 80h ;系统调用
msg:
        db "Hello World!", 0ah, 0dh
```

```
# nasm -f elf hello.asm -g -F stabs
# gcc -o hello hello.o -g
```

7.1.2 调试工具的使用介绍

gdb list 列出代码和行号

```
# gdb hello
(gdb) list
1        ;FileName: hello.asm
2        ;Compile: nasm -f elf hello.asm -g -F stabs
3        ;         gcc -o hello hello.o -g
4
5        section .text
6        global main
7        main:
8                mov eax, 4 ;4 号调用
9                mov ebx, 1 ;ebx 送 1 表示 stdout
10                mov ecx, msg ; 字符串的首地址送如 ecx
(gdb)
```

gdb break 可以根据行号、函数、条件生成断点

```
(gdb) break 8
Breakpoint 1 at 0x80483e0: file hello.asm, line 8.
(gdb) breka 9
Undefined command: "breka".  Try "help".
(gdb) break 10
Breakpoint 2 at 0x80483ea: file hello.asm, line 10.
(gdb) break 11
Breakpoint 3 at 0x80483ef: file hello.asm, line 11.
(gdb) run
Starting program: /home/lei/asm/hello
Breakpoint 1, 0x080483e0 in main ()
(gdb) p $eax
$1 = 1
(gdb) n
Single stepping until exit from function main,
which has no line number information.

Breakpoint 2, 0x080483ea in main ()
(gdb) p $eax
$2 = 4
(gdb)
```

info registers 用来查看所有寄存器里存储的数据

```
(gdb) info registers
eax            0x4        4
```

```
ecx            0xbffff344        -1073745084
edx            0xbffff2d4        -1073745196
ebx            0x1        1
esp            0xbffff2ac        0xbffff2ac
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x80483ea         0x80483ea <main+10>
eflags         0x246        [ PF ZF IF ]
cs             0x73        115
ss             0x7b        123
ds             0x7b        123
es             0x7b        123
fs             0x0        0
gs             0x33        51
(gdb)
```