C语言程序gdb调试

示例程序：1020.c

int main()

{

int a=1;

if(a++>0 || a++<2)

printf("true\n");

else

printf("false\n");

printf("a=%d\n",a);

return 0;

}

在命令行下输入**gcc -g 1020.c -o 1020**编译此程序，其中的-g表示在编译时添加调试信息。

在命令行下输入gdb -q 1020 进入调试界面，其中-q表示减少gdb的一些提示信息

gdb) b main //设置断点

gdb) r //运行程序

gdb) l 或list //查看程序源代码

gdb)disassem //查看程序的汇编源代码

gdb)p $eip //查看当前指令的地址

显示eip寄存器的值如 0x804841d <main+9>，从上面disassem的显示结果可以看到下一条要执行指令为 movl $0x1,0x1c(%esp)，即表示赋值语句a=1，a在内存中的地址是%esp+0x1c

gdb) ni 或stepi //执行一条汇编指令gdb)n 或 step //执行c程序的一行；

gdb)p $eip

显示eip寄存器的值如 0x804841d <main+17>，表示已经执行完movl $0x1,0x1c(%esp)，下一条要执行指令为cmpl $0x0,0x1c(%esp) ，即c语言代码a++>0中的先将a与0比较(即计算a-0)，根据结果设置%eflag的中各个标志位

gdb)ni

执行cmpl $0x0, 0x1c(%esp)，下一条语句为setg %al

gdb)ni

执行setg %al,即根据刚才a-0的结果来设置%al的值，如果结果为正，则%al中的值为1，否则为0

gdb)ni

执行addl $0x1,0x1c(%esp)，即执行a++>0中的a++运算，从这几行汇编代码可以很清晰的看出a++中的++是用后加一

gdb)ni

执行test %al,%al 将%al中的值相与，并根据结果来设置标志位，即当%al中的值为0时，则结果为0，于是零标志位ZF=1，如果%al中的值不为0，则结果不为0，零标志位ZF=0

gdb)ni

执行jne 0x8048447 <main+51>，如果不等于0，也就是a++>0为true，则跳转到<main+51>的指令处，这也是本例中的执行过程，而如果等于0，则说明a++>0为false，那么就不跳转，继续判断a++<2，对应的代码为

<main+34> cmpl $0x1,0x1c(%esp) //在编译时，自动将a<2替换成a<=1

<main+39> setle %al //小于等于时，设置%al为1

<main+42> addl $0x1,0x1c(%esp) //完成a++<2的a++运算（用后加一）

<main+49>je 0x8048455 <main+65> //如果刚才的 a++<2不成立，那么说明整个表达工的值为false，那就跳转到<main+65>处，去打印false；否则a++<2成立，那就不跳转，下一条执行<main+51>处的指令，去打印true

gdb)ni

执行<main+51> movl $0x8048540 ,(%esp)此时可以用x/16bx 0x08048540来查看此处内存的值，可以看到0x08048540开始的5个字节值为0x74 0x 72 0x75 0x65 0x00，即为字符串“true”——最后一个字节0x00为结束符，同时也可以再往后看到6个字节为0x66 0x61 0x

6c 0x 73 0x65 0x00，即为字符串false，这行代码的意思是将true这个字符串的起始地址值放至(%esp)中

gdb)ni

执行<main+58> call 0x8048050语句，即调用这个地址处的函数，来打印字符串，刚才放置在(%esp)处的地址值为此函数的参数

gdb)ni

执行<main+63> jmp 0x8048461<main+77>此时打印完了true，就准备去执行语句printf，而后面的<main+65>与<main+72>就是打印false，

gdb)ni

执行mov $0x804854b,%eax;此时可以用x /16bx 0x0804854b来查看内存的值，为0x61 0x3d0x25 0x64 0x0a即字符串”a=%d\n”

mov 0x1c(%esp),%edx //将a的值放入到%edx中

mov %edx,0x4(%esp) //将a的值放入到%esp+4处

mov %eax,(%esp) //将字符串”a=%d\n”放到%esp处

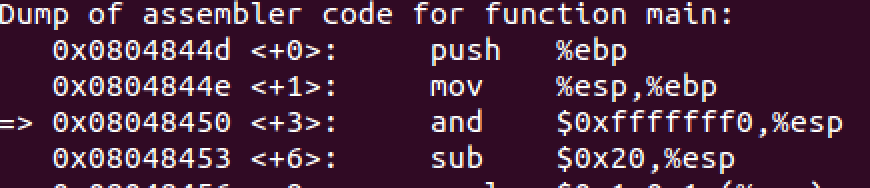
call 0x8048340 //调用函数printf，此函数会自动处理参数，将第二个参数（即a）的值替换第一个字符串中的%d，

mov $0x0,%eax //即return 0中的0

leave

ret //对应return

另：在ubuntu 14.0等其他版本中，其自带的gcc为4.8.4版本，将1020.c编译时会产生不同的汇编代码，其调试过程与上述基本相同，在gdb调试时会显示箭头指示当前执行的汇编指令行：



汇编代码解析如下：

0x0804844d <+0>: push %ebp //将帧指针压栈

0x0804844d <+1>: mov %esp,%ebp //准备创建新的栈帧

0x0804844d <+3>: and $0xffffff0, %esp //栈顶指针对齐

0x0804844d <+6>: sub $0x20,%esp //给栈分配32个字节，此时栈底指针为%ebp，栈顶指针为%esp

0x0804844d <+9>:movl $0x1,0x1c(%esp) //在%esp+0x1c处，也就是%ebp-4处，是放的局部变量a，此时执行的是语句a=1

0x0804844d <+17> :mov 0x1c(%esp),%eax //将a的值放于寄存器中

0x0804844d <+21>:lea 0x1(%eax),%edx //计算a+1，即a++中的加1操作，并将结果放于%edx中

0x0804844d <+24>:mov %edx,0x1c(%esp) //a=a+1，此时完成了a++操作

0x0804844d <+28>:test %eax,%eax //测试%eax，其结果就是%eax的值，相当于将%eax减去0，即做运算a-0（注意此时%eax中保存的值仍然为未加1前的a，+1后的值已经保存到内存中去了），并根据结果设置标志位

0x0804844d <+30>:jg 0x804847d 如果a>0，则跳转去执行print(“true”)，后面的代码与上述的代码基本相同。