第4季

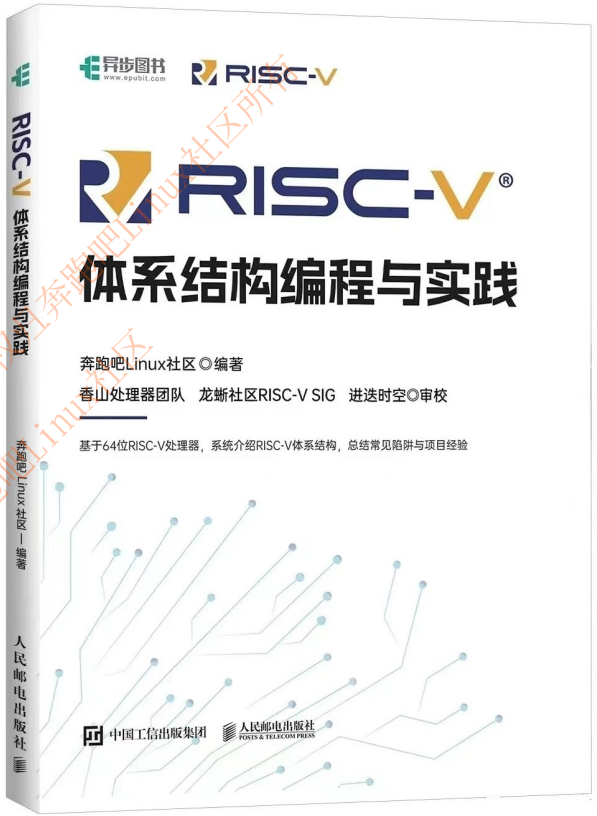
内嵌汇编

本节课主要内容

- 本章主要内容
  - GCC内嵌汇编代码基本用法
  - 案例分析
  - 内嵌汇编常见陷阱和总结
  - 实验

技术手册：

1. 《Using the GNU Compiler Collection，v9.3.0》第6.47章



本节课主要讲解书上第7章内容

扫码订阅RISC-V视频课程

GCC内嵌汇编

- 内嵌汇编（Inline Assembly Language）：在C语言中嵌入汇编代码
- 目的：
  - ✓ 优化：对于特定重要代码（time-sensitive）进行优化
  - ✓ C语言需要访问某些特殊指令来实现特殊功能比如 内存屏障指令
- 参考资料：《Using the GNU Compiler Collection，v9.3.0》第6.47章

6.47 How to Use Inline Assembly Language in C Code

The `asm` keyword allows you to embed assembler instructions within C code. GCC provides two forms of inline `asm` statements. A *basic* `asm` statement is one with no operands (see Section 6.47.1 [Basic Asm], page 568), while an *extended* `asm` statement (see Section 6.47.2 [Extended Asm], page 570) includes one or more operands. The extended form is preferred for mixing C and assembly language within a function, but to include assembly language at top level you must use basic `asm`.

You can also use the `asm` keyword to override the assembler name for a C symbol, or to place a C variable in a specific register.

内嵌汇编两种模式

- 基础内嵌汇编（Base Asm）：不带参数
- 扩展的内嵌汇编（Extended Asm）：C语言变量参数

基础内嵌汇编

```
asm （“汇编指令”）
```

asm *asm-qualifiers* (*AssemblerInstructions*)

- 格式：
  - ✓ asm关键字：表明这是一个GNU扩展
  - ✓ 修饰词（qualifiers）
  - ✓ volatile：在基础内嵌汇编中通常不需要这个修饰词
  - ✓ inline：内联，asm汇编的代码会尽可能小

- 汇编代码块
  - ✓ GCC编译器把内嵌汇编当成一个字符串
  - ✓ GCC编译器不会去解析和分析内嵌汇编。
  - ✓ 多条汇编指令，需要使用“\n\t”来换行

```
#define nop()            __asm__ __volatile__ ("nop")
```

arch/risc-v/include/asm/barrier.h

```
register struct task_struct *riscv_current_is_tp __asm__("tp");
```

arch/risc-v/include/asm/current.h

```
#define __io_pbr()      __asm__ __volatile__ ("fence io,i"  : : : "memory");
#define __io_par(v)     __asm__ __volatile__ ("fence i,ior" : : : "memory");
#define __io_pbw()      __asm__ __volatile__ ("fence iow,o" : : : "memory");
#define __io_paw()      __asm__ __volatile__ ("fence o,io"  : : : "memory");
```

arch/risc-v/include/asm/io.h

```
static inline void wait_for_interrupt(void)
{
        __asm__ __volatile__ ("wfi");
}
```

arch/risc-v/include/asm/processor.h

扩展内嵌汇编

- 格式：
  - ✓ asm关键字：表明这是一个GNU扩展
  - ✓ 修饰词（qualifiers）
    - volatile：用来关闭GCC优化
    - inline：内联，asm汇编的代码会尽可能小
    - goto：在内嵌汇编里会跳转到C语言的标签里

```
asm asm-qualifiers ( AssemblerTemplate
                   : OutputOperands
                   [ : InputOperands
                   [ : Clobbers ] ])
```

```
asm 修饰词 (
            指令部
              : 输出部
              : 输入部
              : 损坏部
        );
```

扩展内嵌汇编 – 输出部

- 输出部：用于描述在指令部中**可以被修改的C语言变量**以及约束条件
  - ✓ 每个输出约束（constraint）通常以“=”号开头，接着的字母表示对操作数类型的说明，然后是关于变量结合的约束。

    “=/+” + 约束修饰符 + 变量

  - ✓ 输出部通常使用“=”或者“+”作为输出约束，其中“=”表示被修饰的操作数只具有可写属性，“+”表示被修饰的操作数只具有可读可写属性。
  - ✓ 输出部可以是空的

扩展内嵌汇编 – 输入部与损坏部

- 输入部：用来描述在指令部**只能被读取访问的C语言变量**以及约束条件
  - ✓ 输入部描述的参数是只有只读属性，不要试图去修改输入部的参数的内容，因为GCC编译器假定，输入部的参数的内容在内嵌汇编之前和之后都是一致的
  - ✓ 在输入部中不能使用“=”或者“+”约束条件，否则编译器会报错。
  - ✓ 输入部可以是空的。

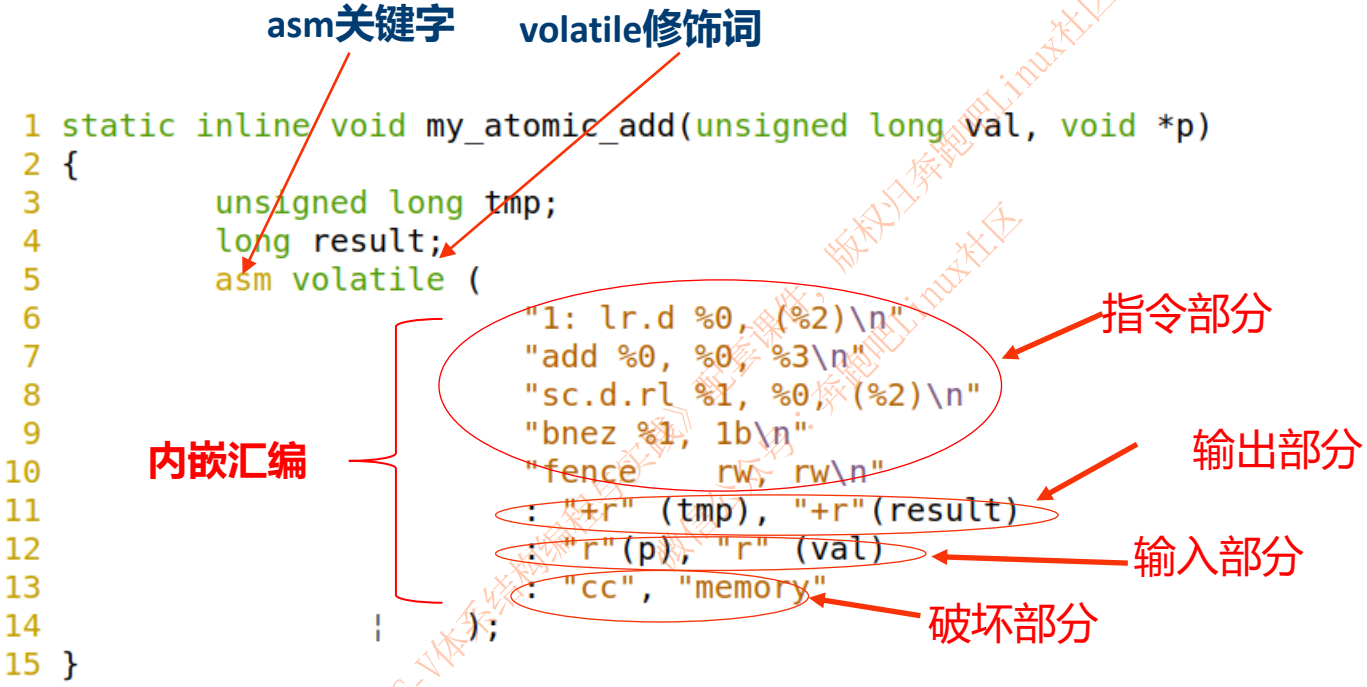
```
atomic_add.c: In function 'my_atomic_add':
atomic_add.c:16:8: error: input operand constraint contains '+'
   16 |       );
      |       ^
```

- 损坏部（Clobbers）
  - ✓ “memory”告诉GCC编译器内联汇编指令改变了内存中的值，强迫编译器在执行该汇编代码前存储所有缓存的值，在执行完汇编代码之后重新加载该值，目的是防止编译乱序。
  - ✓ “cc”表示内嵌汇编代码修改了状态寄存器相关的标志位。
  - ✓ 当输入部分和输出部分显式地使用了通用寄存器时，应该在损坏明确告诉编译器

扩展内嵌汇编 – 参数表示

- 指令部中的 参数表示：
  在内嵌汇编代码中，使用%0 对应输出部和输入部的第一个参数，使用%1表示第二个参数。

例子：扩展内嵌汇编



书上例7-2

输出部和输入部的约束修饰符

表 1.13　GCC 内联操作符和修饰符

操作符/修饰符	说明
=	被修饰的操作数只写
+	被修饰的操作数具有可读可写属性
&	被修饰的操作数只能作为输出

参见《Using the GNU Compiler Collection，v9.3.0》第6.47.3.3章

输出部和输入部的约束修饰符 – 通用

表 7.2　　内嵌汇编常见操作数约束符

操作符/修饰符	说　明
p	内存地址
m	内存变量
r	通用寄存器
o	内存地址，基地址寻址
i	立即数
V	内存变量，不允许偏移的内存操作数
n	立即数

参见《Using the GNU Compiler Collection，v9.3.0》第6.47.3.1章

输出部和输入部的约束修饰符 – RISC-V

表 7.3　　RISC-V 架构中特有的操作数约束符 ①

约束符	说　明
f	表示浮点数寄存器
I	表示 12 位有符号的立即数
J	表示值为 0 的整数
A	表示存储到通用寄存器中的一个地址
K	表示 5 位无符号的立即数，用于 CSR 访问指令

参见《Using the GNU Compiler Collection，v9.3.0》第6.47.3章

“A”约束符

例子：my_atomic_add()函数是把val的值原子地加到指针变量p指向的内存中

```
1   static inline void my_atomic_add(unsigned long val, void *p)
2   {
3       unsigned long tmp;
4       long result;
5       asm volatile (
6               "1: lr.d %0, (%2)\n"
7               "add %0, %0, %3\n"
8               "sc.d.rl %1, %0, (%2)\n"
9               "bnez %1, 1b\n"
10              "fence    rw, rw\n"
11              : "+r" (tmp), "+r"(result)
12              : "r"(p), "r" (val)
13              : "memory"
14               );
15  }
```

“lr.d %0, (%2)”指令用来加载指针变量p的值到tmp变量中，这里使用“（）”来表示访问内存地址的内容。

```
1   static inline void my_atomic_add(unsigned long val, void *p)
2   {
3       unsigned long tmp;
4       int result;
5       asm volatile (
6               "1: lr.d %0, %2\n"
7               "add %0, %0, %3\n"
8               "sc.d.rl %1, %0, %2\n"
9               "bnez %1, 1b\n"
10              : "+r" (tmp), "+r"(result), "+A"(*(unsigned long *)p)
11              : "r" (val)
12              : "memory"
13               );
14  }
```

在第10行中，输出部分的第3个参数使用了约束符“A”，并且参数变成了“*(unsigned long *)p”，第6行指令也变成了“lr.d %0, %2”。

汇编符号名字来替代以前缀%

```
int add(int i, int j)
{
        int res = 0;

        asm volatile (
                "add %0, %1, %2"
                : "=r" (res)
                : "r" (i), "r" (j)
        );

        return res;
}
```

```
1   int add(int i, int j)
2   {
3       int res = 0;
4
5       asm volatile (
6       "add %[result], %[input_i], %[input_j]"
7       : [result] "=r" (res)
8       : [input_i] "r" (i), [input_j] "r" (j)
9       );
10
11      return res;
12  }
```

为了提高代码可读性，可以使用汇编符号名字来替代以前缀%来表示的操作数

实验1：实现简单的memcpy函数

1. 实验目的

熟悉内嵌汇编的使用。

2. 实验内容

使用内嵌汇编的方式来实现简单的memcpy()函数：从0x8020000地址拷贝32字节到0x8021000地址处，并使用GDB来验证数据是否拷贝正确。

陷阱与坑

- GDB不能单步内嵌汇编。
- **笨叔建议：**使用纯汇编的方式验证过之后，再移植到内嵌汇编中。
- 认真仔细检查，内嵌汇编的参数，很容易搞错。
- 输出部和输入部的 修饰符 不能用错，否则 程序 会跑错

```
static void my_memcpy_asm_test(unsigned long src, unsigned long dst,
                unsigned long size)
{
        unsigned long tmp = 0;
        unsigned long end = src + size;

        asm volatile (
                        "1: ld %1, (%2)\n"
                        "sd %1, (%0)\n"
                        "addi %0, %0, 8\n"
                        "addi %2, %2, 8\n"
                        "blt %2, %3, 1b"
                        :"+r" (dst), "+r" (tmp), "+r" (src)
                        : "r" (end)
                        : "memory");
}
```

“+r” (dst)改成“=r” (dst)会导致程序崩溃

在输出部的dst参数，从“+r”（dst）改成“=r”（dst）后，结果导致程序崩溃。

原因是参数dst在sd/addi指令中，它既要读取，又要写入。

实验2：使用汇编符号名的方式来写内嵌汇编

1. 实验目的

熟悉内嵌汇编的使用。

2. 实验内容

在实验1的基础上尝试使用汇编符号名的方式来编写内嵌汇编。

实验3：使用内嵌汇编来完善 memset函数

本次实验要求大家使用 GCC内嵌汇编来完成__memset_16bytes汇编函数。

内嵌汇编的高级玩法： 和宏结合

- 在带参数的宏中，“#”运算符作为一个预处理运算符，可以把记号转换成字符串
- “##”：用于连接参数和另一个标识符，形成新的标识符。

```
60 #define ATOMIC_OP(op, asm_op, I, asm_type, c_type, prefix)              \
61 static __always_inline                                                  \
62 void atomic##prefix##_##op(c_type i, atomic##prefix##_t *v)             \
63 {                                                                       \
64         __asm__ __volatile__ (                                         \
65                 "       amo" #asm_op "." #asm_type " zero, %1, %0"     \
66                 : "+A" (v->counter)                                    \
67                 : "r" (I)                                              \
68                 : "memory");                                           \
69 }                                                                       \
70
71 #define ATOMIC_OPS(op, asm_op, I)                                       \
72         ATOMIC_OP (op, asm_op, I, w,  int,   )
73
74 ATOMIC_OPS(add, add,  i)
75 ATOMIC_OPS(sub, add, -i)
76 ATOMIC_OPS(and, and,  i)
77 ATOMIC_OPS( or,  or,  i)
78 ATOMIC_OPS(xor, xor,  i)
```

“##”号，它把“atomic_”与宏的参数op拼接在一起构成函数名

这里”#”在C语言的宏中是一个预处理预算符。
“amoadd.w zero, %1, %0\n\n”

ATOMIC_OP是一个宏，这里定义好几个宏，例如or，xor，and等操作，第二个参数是 汇编指令，例如add指令

arch/risc-v/include/asm/atomic.h

ATOMIC_OP(add, add，i) 宏展开之后：

```
static __always_inline
void atomic_add(int i, atomic_t *v)
{
        __asm__ __volatile__ (
                "amoadd.w zero, %1, %0"
                : "+A" (v->counter)
                : "r" (i)
                : "memory");
}
```

内嵌汇编实验4：使用内嵌汇编与宏的结合

实验要求：

1. 实现一个宏MY_OPS(ops， instruction), 可以实现对某个内存地址实现or， xor， and， andnot等ops操作。

实现这个一个MY_OPS的宏，
#define MY_OPS(ops, asm_ops)

展开之后变成这样的一个函数：
static inline my_asm_##ops(unsigned long mask, void *p)

通过一个宏来实现多个类似的函数，这是Linux内核常用的技巧

实验5：实现读和写系统寄存器的宏

实验要求：

1. 实现一个read_csr(csr)以及write_csr(val, csr)的宏，可以读取RSIC-V中的系统寄存器。

内嵌汇编：goto

- 内嵌汇编的goto模板，可以跳转到C语言的label标签里。

```
asm goto (
        AssemblerTemplate
        : /*输出部为空*/
        : 输入部
        : 破坏部
        : GotoLabels)
```

- Goto模板的输出部必须为空。
- 新增一个gotolabels的部，里面列出了C语言的label，是允许跳转的label

Goto的栗子

```
1    static int test_asm_goto(int a)
2    {
3        asm goto (
4                "addi %0, %0, -1\n"
5                "beqz %0, %l[label]\n"
6                :
7                : "r" (a)
8                : "memory"
9                : label);
10
11       return 0;
12
13   label:
14       printf("%s: a = %d\n", __func__, a);
15       return 1;
16   }
```

判断参数a是否为1。如果为1的话，跳转到label标签处，否则就直接返回0

实验6：goto模板的内嵌汇编

实验要求：

1. 使用goto模板来实现一个内嵌汇编函数，判断函数参数是否为1，为1的话，跳转到label中，并且打印参数的值。

```
static int test_asm_goto(int a)
```

案例1：使用了错误的约束修饰符

```
 5 static void my_memcpy_asm_test(unsigned long src, unsigned long dst,
 6                 unsigned long size)
 7 {
 8         unsigned long tmp = 0;
 9         unsigned long end = src + size;
10
11         asm volatile (
12                         "1: ld %1, (%2)\n"
13                         "sd %1, (%0)\n"
14                         "addi %0, %0, 8\n"
15                         "addi %2, %2, 8\n"
16                         "blt %2, %3, 1b"
17                         : "=r" (dst), "+r" (tmp), "+r" (src)
18                         : "r" (end)
19                         : "memory");
20 }
```

```
benshushu:inline_asm# ./example7-8-memcpy_test
0x2ae08c52a0 0x2ae08c55d0
[  204.842403] example7-8-memc[260]: unhandled signal 11 code 0x1 at 0x0000002ae08e6000 in
[  204.843862] CPU: 1 PID: 260 Comm: example7-8-memc Not tainted 5.15.0+ #64
[  204.845599] Hardware name: riscv-virtio,qemu (DT)
[  204.846225] epc : 0000002ae08c2736 ra : 0000002ae08c27e4 sp : 0000003ffdeec9e0
[  204.846881]  gp : 0000002ae08c4800 tp : 0000003f9ebdc310 t0 : 0000003ffdeec4e8
[  204.847289]  t1 : 0000000000000001 t2 : 0000000000000010 s0 : 0000003ffdeeca20
[  204.847681]  s1 : 0000000000000000 a0 : 0000002ae08c52a0 a1 : 0000002ae08c55d0
[  204.848055]  a2 : 0000000000000320 a3 : 0000002ae08e6000 a4 : 0000000000000055
[  204.849215]  a5 : 0000002ae08e5ce0 a6 : fffffffffffffff6 a7 : 0000000000000040
[  204.850196]  s2 : 0000002ab550d180 s3 : 0000000000000000 s4 : 0000002ab550ced0
[  204.851028]  s5 : 0000003f99a242c8 s6 : 0000002ab54e23d0 s7 : 0000002ab550d090
[  204.851852]  s8 : 0000002ab550d150 s9 : 0000002ab54bd850 s10: 0000000000000000
[  204.852819]  s11: 0000002ab54bd7c0 t3 : 0000000000000000 t4 : 0000002ae08c55d0
[  204.854394]  t5 : 0000003ffdeec510 t6 : 000000000000002a
[  204.855087] status: 8000000200006020 badaddr: 0000002ae08e6000 cause: 000000000000000f
Segmentation fault
```

案例2：

实现两字节交换的功能，其中有两处明显的错误，请大家认真阅读并找出来。

```
 6 static void swap_data(unsigned char *src, unsigned char *dst,unsigned int size)
 7 {
 8         unsigned int len = 0;
 9         unsigned int tmp;
10
11         asm volatile (
12                 "1: lhu a5, (%[src])\n"
13                 "sll a6, a5, 8\n"
14                 "srl a7, a5, 8\n"
15                 "or %[tmp], a6, a7\n"
16                 "sh %[tmp], (%[dst])\n"
17                 "addi %[src], %[src], 2\n"
18                 "addi %[dst], %[dst], 2\n"
19                 "addi %[len], %[len], 2\n"
20                 "bltu %[len], %[size], 1b\n"
21                 : [dst] "+r" (dst), [len] "+r"(len), [tmp] "+r" (tmp)
22                 : [src] "r" (src), [size] "r" (size)
23                 : "memory"
24         );
25 }
```

书上例7-9

```
benshushu:example7-9# ./in_test
0 1 2 3 4 5 6 7 8 9
[ 1170.694573] in_test[339]: unhandled signal 11 code 0x1 at 0x0000000000000100 in in_test[2ac9e84000+1000]
[ 1170.696940] CPU: 1 PID: 339 Comm: in_test Not tainted 5.15.0+ #64
[ 1170.697534] Hardware name: riscv-virtio,qemu (DT)
[ 1170.697962] epc : 0000002ac9e846a4 ra : 0000002ac9e8468c sp : 0000003ff22e3a30
[ 1170.698449]  gp : 0000002ac9e86800 tp : 0000003fa156b310 t0 : 0000003fa145ce78
[ 1170.698863]  t1 : 0000003fa14afa0a t2 : 0000002ac9e865d0 s0 : 000000000000000a
[ 1170.699253]  s1 : 0000002ac9e872a0 a0 : 000000000000000a a1 : 0000002ac9e872e0
[ 1170.700324]  a2 : 000000000000000a a3 : 0000000000010001 a4 : 0000000000000000
[ 1170.700956]  a5 : 0000000000000100 a6 : 0000000000010000 a7 : 0000000000000001
[ 1170.701594]  s2 : 000000000000000a s3 : 0000002ac9e84828 s4 : 0000002ac9e872c0
[ 1170.702259]  s5 : 0000003f99a242c8 s6 : 0000002ab54e23d0 s7 : 0000002ab550d9c0
[ 1170.703613]  s8 : 0000002ab550d980 s9 : 0000002ab54bd850 s10: 0000000000000000
[ 1170.704445]  s11: 0000002ab54bd7c0 t3 : 0000000000056a0a t4 : 0000000000000009
[ 1170.705069]  t5 : 0000000000000003 t6 : 000000000000002a
[ 1170.705492] status: 8000000200006020 badaddr: 0000000000000100 cause: 000000000000000f
Segmentation fault
benshushu:example7-9#
```

运行出现段错误

第一个错误

在第12~15行中显式地使用了a5、a6以及a7三个通用寄存器，那么需要在破坏部里声明这些寄存器已经被内嵌汇编使用了

修改方法：

```
 6 static void swap_data(unsigned char *src, unsigned char *dst,unsigned int size)
 7 {
 8         unsigned int len = 0;
 9         unsigned int tmp;
10
11         asm volatile (
12                 "1: lhu a5, (%[src])\n"
13                 "sll a6, a5, 8\n"
14                 "srl a7, a5, 8\n"
15                 "or %[tmp], a6, a7\n"
16                 "sh %[tmp], (%[dst])\n"
17                 "addi %[src], %[src], 2\n"
18                 "addi %[dst], %[dst], 2\n"
19                 "addi %[len], %[len], 2\n"
20                 "bltu %[len], %[size], 1b\n"
21                 : [dst] "+r" (dst), [len] "+r"(len), [tmp] "+r" (tmp)
22                 : [src] "r" (src), [size] "r" (size)
23                 : "memory", "a5", "a6", "a7"
24         );
25 }
```

在破坏部里声明

第二个错误

```
benshushu:example7-9# ./in_test
0 1 2 3 4 5 6 7 8 9
0x2ae28ed2a0
0x2ae28ed2aa
1 0 3 2 5 4 7 6 9 8
munmap_chunk(): invalid pointer
Aborted
benshushu:example7-9#
```

swap_data()函数前后打印的buf地址发生了变化

```
 6 static void swap_data(unsigned char *src, unsigned char *dst,unsigned int size)
 7 {
 8         unsigned int len = 0;
 9         unsigned int tmp;
10
11         asm volatile (
12                 "1: lhu a5, (%[src])\n"
13                 "sll a6, a5, 8\n"
14                 "srl a7, a5, 8\n"
15                 "or %[tmp], a6, a7\n"
16                 "sh %[tmp], (%[dst])\n"
17                 "addi %[src], %[src], 2\n"
18                 "addi %[dst], %[dst], 2\n"
19                 "addi %[len], %[len], 2\n"
20                 "bltu %[len], %[size], 1b\n"
21                 : [dst] "+r" (dst), [len] "+r"(len), [tmp] "+r" (tmp),
22                 [src] "+r" (src)
23                 : [size] "r" (size)
24                 : "memory", "a5", "a6", "a7"
25         );
26 }
```

修复：把src放到输出部，属性为可读可写

Root cause：

参数src指定的属性不正确。参数src应该具有可读可写属性，因为第17行修改src指针的指向。

```
1    call    malloc@plt
2    beq a0,zero,.L2
3    mv  s1,a0
4
5        #APP
6    # 11 "in_test.c" 1
7    1: lhu a5, (s1)
8    sll a6, a5, 8
9    srl a7, a5, 8
10   or a2, a6, a7
11   sh a2, (a4)
12   addi s1, s1, 2
13   addi a4, a4, 2
14   addi a3, a3, 2
15   bltu a3, a1, 1b
16
17   # 0 "" 2
18   #NO_APP
19
20   mv  a0,s1
21   call    free@plt
```

修改前的反汇编

```
1    call    malloc@plt
2    beq a0,zero,.L2
3    mv  s1,a0
4    mv  a1,s1
5
6    #APP
7    # 11 "in_test_fix.c" 1
8        1: lhu a5, (a1)
9    sll a6, a5, 8
10   srl a7, a5, 8
11   or a2, a6, a7
12   sh a2, (a4)
13   addi a1, a1, 2
14   addi a4, a4, 2
15   addi a3, a3, 2
16   bltu a3, a0, 1b
17
18   # 0 "" 2
19   #NO_APP
20
21
22   mv  a0,s1
23   call    free@plt
```

修改后的反汇编

本节课 小结

- 目标：看懂GCC内嵌汇编 以及 学会使用GCC内嵌汇编

- 最后总结一下使用内嵌汇编常见的陷阱。
  - ✓ 需要明确每个C语言参数的约束条件，例如这个参数是应该在输出部还是输入部。
  - ✓ 正确使用每个C语言参数的约束符，使用错误的读写属性会导致程序出错。
  - ✓ 当输入部和输出部显式地使用了通用寄存器时应该在损坏部明确告诉编译器。
  - ✓ 如果内嵌汇编代码修改了内存地址的值，则需要在破坏部使用“memory”参数。
  - ✓ 如果内嵌汇编代码隐含了内存屏障语义，例如获取/释放屏障（acquire/release），则需要在破坏部使用“memory”参数。
  - ✓ 如果内嵌汇编使用lr.d以及sc.d.rl等原子操作指令，建议使用“A”约束符来实现地址寻址。

在广袤的宇宙与有限的时空中，
能通过文字和视频与你共同学习RISC-V，
是我们无比的荣幸！

笨叔

文字不如声音，声音不如视频

扫描订阅RISC-V视频课程

第4季 奔跑吧Linux社区 视频课程

RISC-V体系结构编程与实践

主讲：笨叔

课程名称	进度	时长（分钟）
第1课：课程介绍（免费）	完成	20
第2课：RISC-V体系结构介绍（免费）	完成	47
第3课：RISC-V处理器微架构（免费）	完成	48
第4课：搭建RISC-V开发环境（免费）	完成	30
第5课：RISC-V指令集（免费）	完成	128
第6课：RISC-V函数调用规范	完成	40
第7课：RISC-V GNU AS汇编器	完成	42
第8课：RISC-V GNU 链接脚本	完成	90
第9课：RISC-V GNU 内嵌汇编	完成	52
第10课：RISC-V异常处理	完成	80
第11课：RISC-V中断处理	完成	52
第12课：RISC-V内存管理	完成	116
第13课：内存管理实验讲解	完成	36
第14课：cache基础知识	完成	78
第15课：缓存一致性	完成	96
第16课：RISC-V TLB管理	完成	54
第17课：RISC-V原子操作	未录制	
第18课：RISC-V内存屏障	未录制	
第19课：BenOS操作系统相关知识	未录制	
第20课：RVV可伸缩矢量计算	未录制	
第21课：RISC-V压缩指令	未录制	
第22课：RISC-V虚拟化	未录制	
		总计17小时

更多精彩内容马上献上....
微信公众号：奔跑吧Linux社区

视频课程持续更新中...