

Lab2: x86-64 内联汇编

小实验说明

小实验 (Labs) 是 ICS 这门课程里的一些综合编程题，旨在结合课堂知识解决一些实际中的问题。因为问题来自实际，所以有时候未必能立即在课本上找到相关知识的答案，而是需要“活学活用”。因此，大家需要利用互联网上的知识解决这些问题，但**不要**试图直接搜索这些问题的答案，即便有也不要点进去 (也请自觉不要公开发布答案)。

Deadline: 2024 年 11 月 17 日 23:59:59。

1. 背景

C 语言作为一种“高级的低级语言”，其中一个很大的特性就是能无缝地和汇编语言交互，即在程序中嵌入 (inline) 汇编。嵌入的汇编代码甚至可以参与到编译器优化中 (嵌入的指令不能被改变，但嵌入的汇编可能被移动、删除等)。在这个实验中，我们体验如何直接在 C 语言里合理地与编译器交互，直接操纵机器。

2. 实验要求

2.1 实验内容

在本实验中，所有函数的所有部分都必须使用 inline assembly 实现。除定义临时变量 (可以赋常量初值) 和 return 返回临时变量/参数外，不得使用任何表达式/条件/循环等 C 语言成分。

注意 Online Judge 只检查实现的正确性。我们会人工检查实验代码，确保是使用内联汇编实现的。

本实验借助 x86-64 内联汇编实现以下函数 (框架已在 asm-impl.c) 中：

```
// 返回有符号 64 位整数 a + b 的数值
int64_t asm_add(int64_t a, int64_t b);

// 返回无符号 64 位整数 x 二进制表示中 1 的数量
int asm_popcnt(uint64_t x);

// C 标准库中的 memcpy, 用于复制两段不相交的内存
void *asm_memcpy(void *dest, const void *src, size_t n);

// C 标准库中的 setjmp/longjmp, 用于控制流长跳转
int asm_setjmp(asm_jump_buf env);
void asm_longjmp(asm_jump_buf env, int val);
```

前面三个函数的行为非常明确，你只需用汇编实现以下函数的行为即可，通过阅读例子和文档，熟悉内联汇编的写法：

```
int64_t asm_add(int64_t a, int64_t b) {
    return a + b;
}
int asm_popcnt(uint64_t x) {
    int s = 0;
    for (int i = 0; i < 64; i++) {
        if ((x >> i) & 1) s++;
    }
    return s;
}
void *asm_memcpy(void *dest, const void *src, size_t n) {
    return memcpy(dest, src, n);
}
```

大家可能对 setjmp/longjmp 比较陌生，属于不太常用的 C 标准库函数。和一对函数用于实现控制流的长跳转。它们的声明包含在 setjmp.h，手册 (man setjmp) 的内容与我们需要实现的 asm_setjmp/asm_longjmp 行为一致：

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

以下摘自手册 setjmp (3):

setjmp/longjmp functions are used for performing "nonlocal gotos": transferring execution from one function to a predetermined location in another function. The setjmp() function dynamically establishes the target to which control will later be transferred, and longjmp() performs the transfer of execution.

The setjmp() function saves various information about the calling environment (typically, the stack pointer, the instruction pointer, possibly the values of other registers and the signal mask) in the buffer env for later use by longjmp(). In this case, setjmp() returns 0.

The `longjmp()` function uses the information saved in `env` to transfer control back to the point where `setjmp()` was called and to restore ("rewind") the stack to its state at the time of the `setjmp()` call. In addition, and depending on the implementation (see NOTES), the values of some other registers and the process signal mask may be restored to their state at the time of the `setjmp()` call.

Following a successful `longjmp()`, execution continues as if `setjmp()` had returned for a second time. This "fake" return can be distinguished from a true `setjmp()` call because the "fake" return returns the value provided in `val`. If the programmer mistakenly passes the value `0` in `val`, the "fake" return will instead return `1`.

简单来说，`setjmp` 会在调用时对当前程序的运行状态做一个轻量级快照 (保存在 `env` 参数中)，并返回 `0`。只要 `setjmp` 时调用的函数不返回，程序在运行过程中可以随时调用 `longjmp` 跳转到 `setjmp` 快照时的程序状态，无论中间间隔了多少函数调用，并且给 `setjmp` 一个特定的返回值。我们可以通过下面的小例子理解 `setjmp/longjmp` 的行为：

```
#include <setjmp.h>
#include <stdio.h>

jmp_buf env;

int f(int n) {
    if (n >= 8) longjmp(env, n); // 某个条件达成时，恢复快照
    printf("Call f(%d)\n", n);
    f(n + 1);
}

int main() {
    int r = setjmp(env); // 快照
    if (r == 0) {
        f(1);
    } else { // longjmp goes here
        printf("Recursion reaches %d\n", r);
    }
}
```

利用 `setjmp/longjmp`，大家可以在编写递归搜索 (DFS) 时，在找到一个合法解以后立即退出递归。`setjmp/longjmp` 也是计算机系统研究中的一个有用的 hacking 技巧——它能以非常小的代价实现进程状态的快照。请大家 STFW 理解 `jmp_buf` 的定义和这两个函数的用法。在本实验中，用内联汇编实现 `asm_setjmp` 快照、`asm_longjmp` 长跳转，但无需考虑信号 (信号机制将在后续课程中介绍)。

常见的困惑：这玩意有啥用？

当大家 STFW 时，可能会发现理解 `setjmp/longjmp` 时遇到一定的困难，并且不知道我们到底为什么要用这样的作业折磨大家。此时一定是想放弃的——抱一下大腿、去网上找一个答案就完事了。事实上，这个实验帮助你在用户态理解“进程的状态”——在你完成实验以后，其实你其实也掌握了操作系统如何在一个 CPU 上模拟出多个程序同时运行的假象——完成这个实验会对程序的机器级表示有全新、深刻的理解。

2.2 代码获取与提交

Academic Integrity

从网上或别人那里复制几行代码非常简单——但你如果遵守 academic integrity，自己尝试完成，就会遇到巨大的困难 (尤其如果你没有试着用正确的工具、正确的方法诊断问题)。这是我们给你必要的训练。

请使用我们的 Makefile 编译 (在实验目录中执行 `make`)，确保 Git 记录完整。

在 Lab1 的 `ics-workbench` 中，在终端执行

`git pull origin lab2`

可以获得实验的框架代码。提交方法同 Lab1: 在实验的工作目录中执行 `make submit`。需要设置 `STUID` (学号) 和 `STUNAME` (中文姓名) 环境变量。

2.3 评分与正确性标准

在你提交的代码中，`asm-impl.c` 应该只包含代码的。`asm_jmp_buf` 请定义在 `asm.h` 中。其他代码 (如测试代码) 请存放在其他文件中，以免因包含禁止的操作被 Online Judge 拒绝。我们会将你的 `asm-impl.c` 和 `asm.h` 复制到指定位置。

Labs 完全客观评分；评分方法请阅读[实验须知](#)。我们的测试用例分为 Easy 和 Hard。Easy 用例不涉及 `setjmp/longjmp`——如果 `setjmp/longjmp` 对你来说实在太困难，不妨先试试其他的函数。

2.4 常见问题

(TBD)

3. 内联汇编 (Inline Assembly)

3.1 先看一个例子

在解释内联汇编的语法等之前，我们先看一个实现“加一”的例子：

```
int inc(int x) {
    asm ("incl %[t];"
        : [t] "+r"(x)
        );
    return x;
}
```

其中，内联汇编是一条 `incl` 指令，它用 “[t]” 指定了一个名为 “t” 的操作数 (operand)，它能够被分配到任意寄存器，并且同时作为输入和输出 (“+r”)，在内联汇编被调用前，编译器会确保 %[t] 寄存器的值是 x 的数值。它会被编译成如下汇编代码 (使用 -Os 优化大小)：

```
0000000000000000 <inc>:
0:  89 f8                mov     %edi,%eax
2:  ff c0                inc     %eax
4:  c3                   retq
```

在上述代码中，[t] 被分配为了 `eax` 寄存器。因此内联汇编提供了一种在 C 里打个汇编“补丁”，又可以和 C 语言世界里的表达式/变量交互的机制。

3.2 理解内联汇编

大家应该还记得，狭义的“编译”只完成 .c 到 .s 的转换。换句话说，狭义的编译器其实并不管 .s 到 .o 的生成——它完全不知道汇编代码的含义！因此，如果我们考虑一个没有编译优化，而是直接“直接翻译”C 语言到汇编的编译器，它看到这么一段代码：

```
void foo(int n) {
    for (int i = 0; i < n; i++) {
        asm ("incl %[t]" : [t] "+r"(i)); // 汇编实现的i++;
    }
}
```

不管三七二十一就先直接翻译出一个循环 (假设把循环变量放在 `eax` 寄存器)：

```
mov  $0x0, %eax
.loop:
cmp  %edi, %eax
jge  .ret
// 循环体
add  $0x1, %eax
jmp  .loop
.ret:
retq
```

然后下一步是把内联汇编“粘贴”到循环体里，并且满足内联汇编指定的条件 (在这里，我们希望变量 `i` 在某个寄存器中，刚好这已经被满足了，`i` 在 `eax` 寄存器，于是循环体里只需要填入一条 `incl` 指令即可)：

```
mov  $0x0, %eax
.loop:
cmp  %edi, %eax
jge  .ret
incl %eax
add  $0x1, %eax
jmp  .loop
.ret:
retq
```

同学们不妨可以在循环体里增加其他代码 (例如 `printf("i = %d\n", i);`)，观察汇编代码发生的变化——内联汇编的主要功能是在 C 里“嵌入”一段可以和 C 交互的代码。

3.3 内联汇编

准确地说，编译器会把内联的代码成一个黑盒子 (你甚至可以使用 `.byte` 直接用二进制方式书写指令序列)，然后编译器会生成满足内联汇编要求的代码。在本实验中，我们只需要用到如下的内联汇编语法 (内联汇编可以跳转到程序的其他部分，不过我们暂时用不到)

```
asm ( // 1. 汇编代码 (字符串)
    : // 2. 汇编代码的输出操作数
    : // 3. 汇编代码的输入操作数
    : // 4. 汇编代码可能改写的寄存器 (clobber)
    );
```

其中：

1. 汇编代码就是一个普通的 C 字符串。汇编代码会原样传递给汇编器 (assembler)。汇编代码用换行或 “;” 隔开，因此如果要书写多行的汇编，可以借助 C 语言预编译字符串拼接：

```
asm ("movl $1, %eax\n"
    "movl $1, %ebx;"
    "movl $1, %ecx;"
    ... );
```
2. 输出操作数，相当于“告诉”编译器内联汇编的结果放在了哪里，并且希望把它们复制给 C 语言世界里的什么变量。之前的 "+r"(x) 的含义就是告诉 gcc 可以任意选择寄存器，但输出给 x 变量；
3. 输入操作数，用于把 C 语言世界中的变量值传递进内联汇编，例如 "a"(x) 表示把 x 的值复制给 rax 寄存器。
4. 汇编代码中可能修改的寄存器。编译器在将 C 代码编译成汇编代码时，会为局部变量和临时结果分配寄存器或堆栈上的内存。因此，内联汇编必须不遗漏地声明它们可能会修改的寄存器，否则可能会产生毁灭性的后果。

因此，你必须非常谨慎地“告诉”编译器汇编代码的行为——黑盒子的输入、输出是什么，以及黑盒子可能影响的寄存器 (clobber)。带着这个基本概念，阅读文档时应该不再感到困难了。

由于内联汇编是一个行为明确的“黑盒子”，因此编译器能对内联汇编进行相当程度的优化，例如如果我们将刚才的代码稍作修改：

```
int inc(int y, int x) {
    for (int i = 0; i < y; i++) {
        asm ("incl %[x];"
            : [x] "+r"(x)
            );
    }
    return x;
}
```

编译后会得到如下结果：

```
0000000000000000 <inc>:
0:  89 f0                mov     %esi,%eax
2:  31 d2                xor     %edx,%edx
4:  39 fa                cmp     %edi,%edx
6:  7d 06                jge     e <inc+0xe>
8:  ff c0                inc     %eax
a:  ff c2                inc     %edx
c:  eb f6                jmp     4 <inc+0x4>
e:  c3                  retq
```

可以看到，gcc 依然正确地把 x 分配给了 eax 寄存器，并将循环变量分配给了 edx 寄存器。

3.4 文档与福利

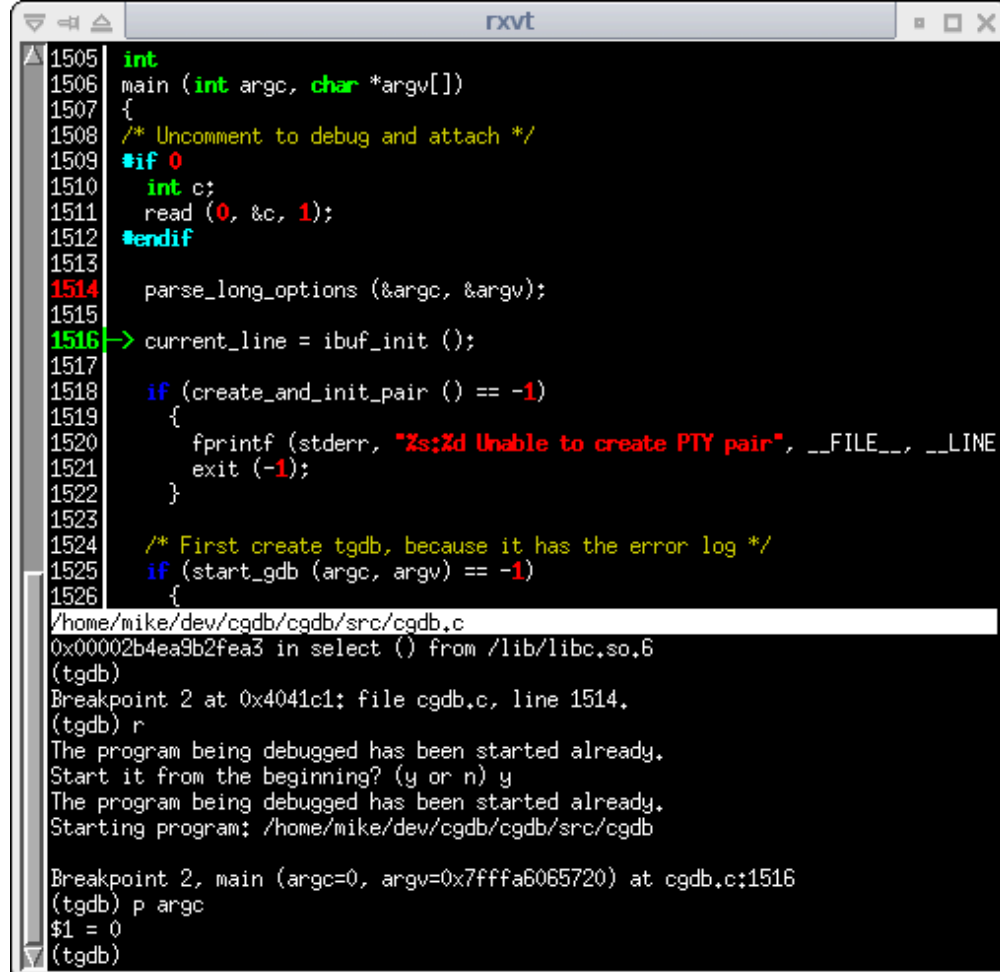
在这个实验中，你需要 RTFM。我们提供了一些编写内联汇编时常用的文档。在做这个实验的过程中，你会发现官方文档还是最好用的：

- GCC 官方文档：[How to Use Inline Assembly Language in C Code](#)，权威、全面。
- [GCC-Inline-Assembly-HOWTO](#)，更容易上手，有相当多的例子供大家学习和参考，但部分内容过时 (少量代码无法在当前 gcc 编译通过)。

以及，无论如何，你们都会在实现 setjmp/longjmp (以及其他函数) 时遇到巨大的困难，出现错误的计算、segmentation fault 等.....怎么办？盯着汇编代码死看？

不要忘记：在你感到不爽的时候，一定有工具可以帮你。如果想排查自己的汇编哪里错了，最好的办法就是跑一跑，调试它——你的汇编不会很长，如果能动态在运行时检查程序的执行流和寄存器的值，那真是再好不过了。好消息是 gdb 就能帮我们搞定！

当然，你也可以用其他第三方的工具，例如 cgdb:



```
1505 int
1506 main (int argc, char *argv[])
1507 {
1508     /* Uncomment to debug and attach */
1509     #if 0
1510     int c;
1511     read (0, &c, 1);
1512     #endif
1513
1514     parse_long_options (&argc, &argv);
1515
1516     current_line = ibuf_init ();
1517
1518     if (create_and_init_pair () == -1)
1519     {
1520         fprintf (stderr, "%s:%d Unable to create PTY pair", __FILE__, __LINE);
1521         exit (-1);
1522     }
1523
1524     /* First create tgdb, because it has the error log */
1525     if (start_gdb (argc, argv) == -1)
1526     {
1527         /home/mike/dev/cgdb/cgdb/src/cgdb.c
1528         0x00002b4ea9b2fea3 in select () from /lib/libc.so.6
1529         (tgdb)
1530         Breakpoint 2 at 0x4041c1: file cgdb.c, line 1514.
1531         (tgdb) r
1532         The program being debugged has been started already.
1533         Start it from the beginning? (y or n) y
1534         The program being debugged has been started already.
1535         Starting program: /home/mike/dev/cgdb/cgdb/src/cgdb
1536
1537         Breakpoint 2, main (argc=0, argv=0x7ffffa6065720) at cgdb.c:1516
1538         (tgdb) p argc
1539         $1 = 0
1540         (tgdb)
```

怎么样，非常方便吧？给大家两个有用的窍门：

1. 使用 gdb 脚本 (可以直接用 `-ex` 传递给 gdb)，从而不必每次都重复键入大量的命令 (file asm-64, b asm_setjmp 等等)；
2. 使用 asm layout (想同时观看寄存器的数值可以使用 regs layout)，配合一个足够大的终端窗口，能够更方便地调试汇编代码、观察每条汇编指令执行后寄存器现场的变化。这对于你理解你写的内联汇编是否符合预期；
3. 在调试的时候尽可能使用简单的测试用例、精简你的代码，减少其他代码 (例如 printf) 对你的干扰。你可以使用宏来控制这些代码的预编译。

通过这个例子，你理解到 STFW 是非常重要的——好的工具能极大提高你的效率，即便有经验的程序员，使用了合适的工具也能有数倍的效率提升。

