3.59题
3.61题
3.63题
3.65题
3.67题
3.69题
3.71题

```
#include <stdio.h>
#define BUFSIZE 64
```

3.59题

计算两个 64 位有符号值 x 和 y 的 128 位乘积

当 x 和 y 都是128位时：

$$x = 2^{64} \times x_k + x_i，其中x_i表示低64位，x_k表示高64位$$

$$y = 2^{64} \times y_k + y_i，其中y_i表示低64位，y_k表示高64位$$

$$\begin{aligned} x \times y &= (2^{64} \times x_k + x_i) \times (2^{64} \times y_k + y_i) \\ &= x_k \times y_k \times 2^{128} + (x_k \times y_i + x_i \times y_k) \times 2^{64} + x_i \times y_i \\ &= (x_k \times y_i + x_i \times y_k) \times 2^{64} + x_i \times y_i，超出128位的部分舍去 \\ &= (x_k \times y_i + x_i \times y_k + z_k) \times 2^{64} + z_i \end{aligned}$$

$$其中zi是没有超出64位的部分，z_k是超出的部分$$

```
y in %rdx, x in %rsi
 store_prod:
  movq  %rdx, %rax      # %rax = %rdx (%rax = yi)
  cqto                  # %rdx 设置为 %rax 的符号扩展位，即 %rdx = yk
  movq  %rsi, %rcx      # %rcx = %rsi (%rcx = xi)
  sarq  $63, %rcx       # %rcx 右移63位，为 y 的符号扩展位，即 %rcx = xk
  imulq  %rax, %rcx     # %rcx = %rax * %rcx = yi * xk 有符号乘法，截取到64位
  imulq  %rsi, %rdx     # %rdx = %rdx * %rsi = yk * xi 有符号乘法，截取到64位
  addq  %rdx, %rcx      # %rcx = xk * yi + xi * yk
  mulq  %rsi            # R[%rdx]:R[%rax] <- %rsi*R[%rax]
                        # 无符号全乘法，高位 %rdx = zk，低位 %rax = zi
  addq  %rcx, %rdx      # %rdx = %rdx + %rcx = xk * yi + xi * yk + zk
  movq  %rax, (%rdi)    # 低64位 %rax 放在低地址
  movq  %rdx, 8(%rdi)   # 高64位 %rdx 放在高地址
  ret
```

3.61题

假如`xp`为空指针，对空指针`NULL`读数据的操作错误。

```
long cread_alt(long *xp) {
  long temp = 0;
  long *p = xp ? xp : &temp;
  return *p;
}
```

3.63题

```
long switch_prob (long x, long n) {
    long result = x;
    switch (n) {
        case 60:
        case 62:
            result = 8 * x;
            break;
        case 63:
            result = x >> 3;
            break;
        case 64:
            result = (result << 4) - x;
            x = result;
        case 65:
            x = x * x;
        default:
            result = x + 0x4b;
    }
    return result;
}
```

3.65题

- **A.** 从2至5行可以看出，`%rdx`和`%rax`所保存的指针所指向的元素进行了交换，而在内循环中`A[i][j]`的指针每次只移动一个单位，而寄存器`%rdx`偏移量为8，所以寄存器`%rdx`保存着指向数组元素`A[i][j]`的指针
- **B.** 另一个寄存器`%rax`保存着指向数组元素`A[j][i]`的指针
- **C.** 已知寄存器`%rax`每次移动一行的距离，所以有

$$M = 120/8 = 15$$

3.67题

- **A.** 如下图

相对 `%rsp` 的偏移地址	储存值
`%rsp + 104`	
……	……
`%rsp + 64`、`%rdi`	
……	……
`%rsp + 24`	z
`%rsp + 16`	&z
`%rsp + 8`	y
`%rsp`	x

- **B.** 传递的是 `%rdi`，即 `%rsp + 64` 表示的栈地址，可以认为这是为结构体 r 分配空间的一部分
- **C.** 栈指针 `%rsp` + 偏移量来访问结构体参数 s 的元素
- **D.** 通过C中描述的传递的栈地址`%rdi`，即`%rsp + 64`加偏移量，间接储存在栈上
- **E.** 调用函数 process 寄存器 `%rsp` 会减去8来存一个返回地址。从 process 返回后 eval 通过访问栈指针 `%rsp` + 偏移量来访问结构体 r 的元素。

相对 `%rsp` 的偏移地址	储存值
`%rsp + 104`	
……	……
`%rsp + 80`	z
`%rsp + 72`	x
`%rsp + 64`、`%rdi`	y
……	……
`%rsp + 24`	z
`%rsp + 16`	&z
`%rsp + 8`	y
`%rsp`	x

- **F.** 当结构体用一个寄存器储存不下时，传递函数参数和作为返回值，都是通过栈来进行储存和访问

3.69题

```
   void test(long i, b_struct *bp)
   i in %rdi, bp in %rsi
1  <test>:
2  mov  0x120(%rsi), %ecx
   # %ecx = %rsi + 0x120 (288)，访问 bp 的 first，%ecx 储存 n
```

```
3  add  (%rsi), %rcx
   # %rcx = %rcx + %rsi，访问 bp 的 last
   # sizeof(int) + CNT * sizeof(a_struct) = 288
4  lea  (%rdi, %rdi, 4), %rax
5  lea  (%rsi, %rax, 8), %rax
   # %rax = bp + 40 * i, bp 是结构体的指针，指向 first
   # sizeof(a_struct) = 40
6  mov  0x8(%rax), %rdx
   # %rdx = %rax + 0x8 = bp + 40 * i + 8，指向结构体或者说 idx
   # first 占 8 个字节
7  movslq %ecx, %rcx
   # mov 符号扩展，n 从双字扩展到四字，x 元素类型为 long
8  mov  %rcx, 0x10(%rax, %rdx, 8)
   # (%rax + 8 * %rdx + 16) = %rcx, long 8字节, 16 = 8 first 长 + 8 idx 长
9  retq
```

- **A.**

$$CNT = (288 - 8)/40 = 7$$

- **B.**

```
typedef struct {
    long idx;
    long x[4];
}
```

3.71题

```
void good_echo() {
    char buf[BUFSIZE];
    int i;
    while (1) {
        if (!fgets(buf, BUFSIZE, stdin))
            return;
        for (i = 0; buf[i] && buf[i] != '\n'; i++)
            if( putchar(buf[i]) == EOF)
                return;
        if (buf[i] == '\n') {
            putchar('\n');
            return;
        }
    }
}
```