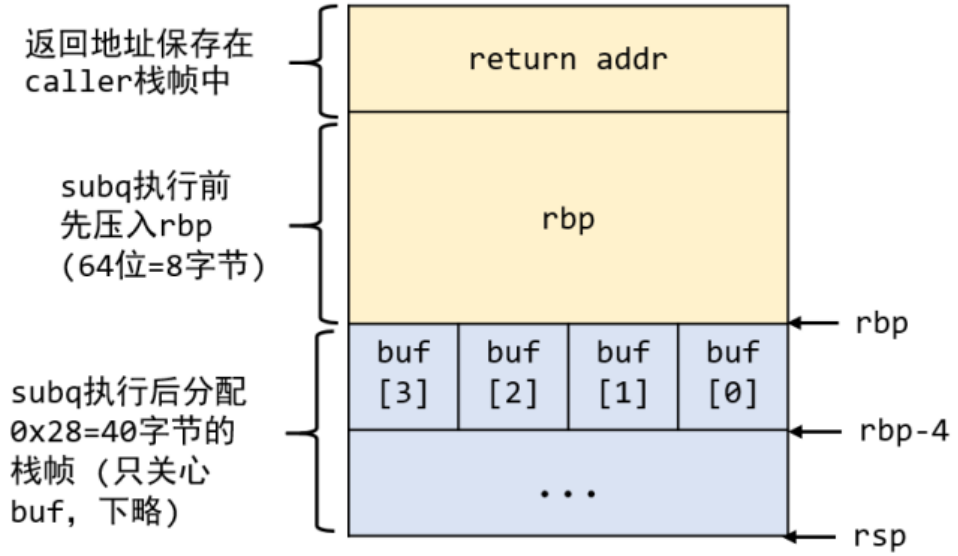
期末复习(2)

《计算机系统概论》习题课

张宇轩
yuxuanzh23@mails.tsinghua.edu.cn

作业四 · Q1 溢出

```
00000000004005d6 <overflow>:
 4005d6: 55                      push %rbp
 4005d7: 48 89 e5                mov %rsp, %rbp
 4005da: 48 83 ec 28             sub $0x28, %rsp
 4005db: 53                      push %rbx
 4005df: bb 00 00 00 00          mov $0x0, %ebx
 4005e4: eb 0d                   jmp 4005f3 <overflow+0x1d>
 4005e6: 48 63 c3                movslq %ebx, %rax
 4005e9: 8b 55 dc                mov -0x8(%rbp), %edx        # edx = val
 4005ec: 88 54 05 e0             mov %dl, -0x4(%rbp, %rax, 1) #buf
 4005f0: 8d 5b 01                lea 0x1(%rbx), %ebx         # i++
 4005f3: 48 8d 75 dc             lea -0x8(%rbp), %rsi
 4005f7: bf d4 06 40 00          mov $0x4006d4, %edi         # "%x"地址是 0x4006d4
 4005fc: b8 00 00 00 00          mov $0x0, %eax
 400601: e8 ba fe ff ff          callq 4004c0 <scanf>
 400606: 83 f8 ff                cmp $0xffffffff, %eax       # EOF 值为-1
 400609: 75 db                   jne 4005e6 <overflow+0x10>
 40060b: b8 6d 3b 00 00          mov $0x3b6d, %eax           # 0x3b6d=15213
 400610: 5b                      pop %rbx
 400614: 48 83 c4 28             add $0x28, %rsp
 400615: 5d                      pop %rbp
 400616: c3                      retq
0000000000400617 <main>:
 400617: 55                      push %rbp
 400618: 48 89 e5                mov %rsp, %rbp
 40061b: e8 b6 ff ff ff          callq 4005d6 <overflow>
 400620: 03 05 22 0a 20 00       add 0x200a22(%rip), %eax    # val += one
 400626: 3d 6d 3b 00 00          cmp $0x3b6d, %eax
 40062b: 74 0c                   je 400639 <main+0x22>
 40062d: bf d7 06 40 00          mov $0x4006d7, %edi
 400632: e8 69 fe ff ff          callq 4004a0 <puts>         # printf("Boom!\n")
 400637: eb 0a                   jmp 400643 <main+0x2c>
 400639: bf dd 06 40 00          mov $0x4006dd, %edi
 40063e: e8 5d fe ff ff          callq 4004a0 <puts>         # printf("????\n")
 400643: bf 00 00 00 00          mov $0x0, %edi
 400648: e8 43 fe ff ff          callq 400490 <exit>         # exit 不使用 rbp
```



- 绕开 `val += one;`，直接跳转到 `400626` 处开始执行：

  任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 26

- 直接跳转到 `400639` 处执行 `else` 分支（**必须先载入 `"????\n"` 才能 `printf()`**）：

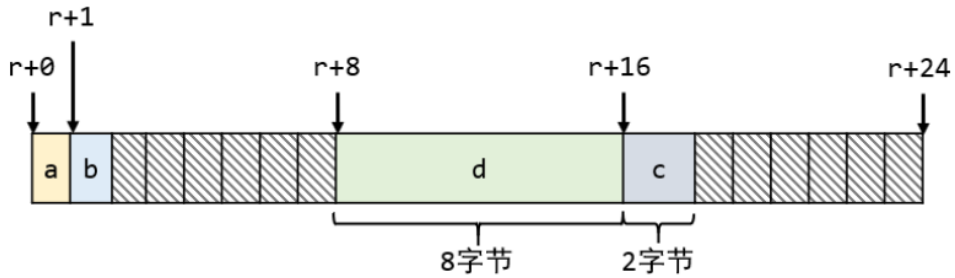
  任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 任填 39

作业四 · Q2 结构体

```
struct matrix_entry {
    char a;
    char b;
    double d;
    short c;
};
struct matrix_entry matrix[5][__空格(1)__];

int return_entry(int i, int j) { return matrix[i][j].c; }
```

```
return_entry:
    movslq      %esi, %rsi
    movslq      %edi, %rdi
    leaq        (%rsi,%rsi,2), %rax
    leaq        0(,%rax,8), %rdx
    leaq        (%rdi,%rdi,4), %rax
    leaq        (%rdi,%rax,4), %rcx
    __空格(2)__  0(,%rcx,8), %rax
    movswl      matrix+ __空格(3)__(%rdx,%rax), %eax
    ret

    .comm       matrix, __空格(4)__

# .comm后的第二个参数表示matrix占据空间的大小，以字节为单位
```

注意到是在X86-64上（即机器字长为64位），因此内存按 64/8 = 8字节 对齐，此时 `struct matrix_entry` 的内存布局如下（每一小格为1个字节，`r` 代表 `matrix_entry` 的首地址位置）：



`matrix_entry` 的总大小为 `24` 字节

- 参数 `j`：
  i. `%esi` 扩展至 `%rsi`；
  ii. `%rax = %rsi + 2 * %rsi = 3 * %rsi`；
  iii. `%rdx = 8 * %rax = 8 * 3 * %rsi = 24 * rsi`，而 `24` 即 `matrix_entry` 的总大小。
- 参数 `i`：
  i. `%edi` 扩展至 `%rdi`；
  ii. `%rax = %rdi + 4 * %rdi = 5 * %rdi`；
  iii. `%rcx = %rdi + 4 * %rax = 21 * %rdi`；
  iv. `%rax = 8 * %rax = 168 * %rdi = (7 * 24) * %rdi`，由上知一行共 `7` 个元素，因此 `空格(1)` 的值为 `7`；
  因为是计算，所以 `空格(2)` 显然为 `leaq`。

作业四 · Q3 跳转表 (PIC)

```
switch_eg:
        cmpq            $6, %rdi
        movq            %rdx, %rcx
        ja              .L8                             # __指令1__
        leaq            __空格1__(%rip), __空格2__
        movslq          (__空格3__, __空格4__,  4), %rdi
        __空格5__       %r8, %rdi
        jmp             *%rdi                           # 以%rdi为目标地址直接跳过去

        .section        .rodata                         # 只读数据段
.L4:
        .long           .L8-.L4
        .long           .L3-.L4
        .long           .L5-.L4
        .long           .L9-.L4
        .long           .L8-.L4
        .long           .L7-.L4
        .long           .L7-.L4
```

- 仔细阅读题目：**PIC代码采用"相对定位"**，即跳转表中的每一项不再是对应分支的实际地址，而是相对于跳转表基地址的偏移（参考课件《6.1 汇编与C语言-5》P33）
  访问对应的分支的C语言伪代码如下：

```
r8 = jump_table_addr;
jump_table_offset = r8[x]; // x是索引，实际地址偏移还需要乘以 sizeof(long) = 4
target_branch_addr = jump_table_addr + jump_table_offset;
rdi = target_branch_addr;
```

求出 `jump_table_offset` 对应于 `movslq` 操作，即 `rdi = r8[x] = *(r8 + x * 4)`，

见课件《6.1 汇编与C语言-5》P.33

事实

- （共享库）代码段中的任意指令与数据段中的任意变量之间的 ” 距离”在运行时是一个常量，与代码和数据加载的**绝对内存位置无关**

方法（编译器）

- 为了利用这一特点，编译器在数据段的开头创建了一个全局偏移表（GOT），每个全局数据对象(全局变量)都对应一个偏移表项
- 编译器同时为GOT中的每个表项生成了一个重定位记录
- 每个包含全局数据引用的目标模块都有其自己的GOT

方法（链接器）

- 动态链接器**重定位GOT中的每个表项，使其包含正确的绝对地址**

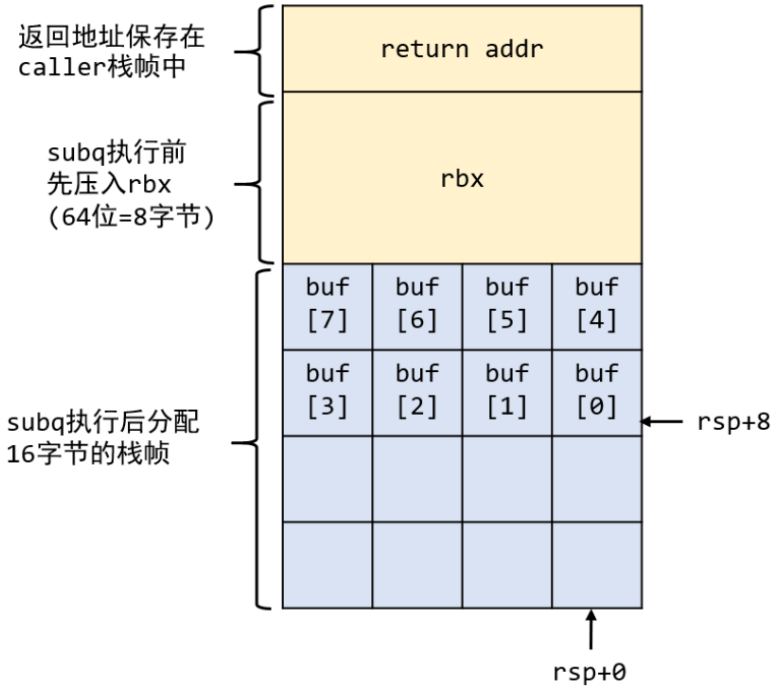
在运行时，全局变量通过GOT被间接引用

作业四 · Q4 echo

```c
void echo() {
    char buf[8];
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushq   %rbx
    xorl    %eax, %eax
    subq    $16, %rsp
    leaq    8(%rsp), %rbx
    movq    %rbx, %rdi
    call    gets
    movq    %rbx, %rdi
    call    puts
    addq    $16, %rsp
    popq    %rbx
    ret
```

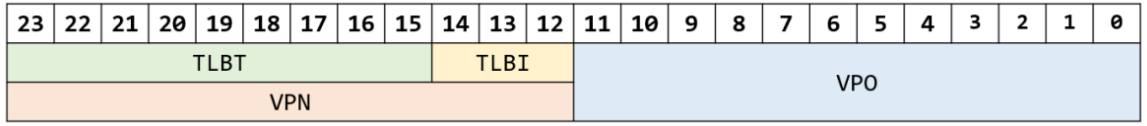
栈上内容如下所示（每一个方格代表一个字节）：

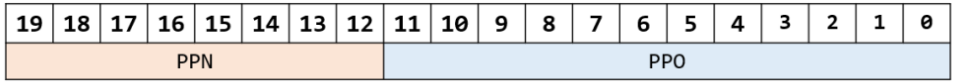


以下的长度是指 `strlen()` 结果，即输入的字符串长度（不计入 `NULL`）：

- 写入 `buf` 但不溢出：长度介于 `[0, 7]`，此时 `buf[7]` 恰好为 `NULL`，不会溢出；
- 写入 `buf` 且更新栈上 `%rbx` 但不更新返回地址：即要求 `NULL` 覆盖栈上 `%rbx` 中的任意一个字节，且不会覆盖返回地址，那么长度介于 `[8, 15]`：
  - `8` 的时候 `buf[7]` 为最后一个输入字符，`NULL` 恰好覆盖栈上 `%rbx` 的最低字节；
  - `15` 的时候 `NULL` 恰好覆盖栈上 `%rbx` 的最高字节，并不影响返回地址。
- 写入 `buf` 且更新栈上 `%rbx` 和返回地址：由上一小题可知长度至少为 `15 + 1 = 16`，此时 `NULL` 恰好覆盖返回地址的最低字节；由于我们只考虑攻击是否成功，因此输入的字符串长度上限没有限制，即长度为 `[16, +inf)`。

作业五 · Q1 页表





假定所有的数据保存在内存中，对照上面的地址格式，写出虚拟地址对应的域后依次查TLB和页表，并记录从获取有效物理地址到**取出实际数据**的实际用时：

- `0x01DBE3`：
  i. `VPO = 0xBE3`，`VPN = 0x1D`，`TLBI = 0x5`，`TLBT = 0x003`
  ii. 查询 `index = 5` 的 TLB set，不包含 TLB tag `0x003`，触发 TLB Miss：用时 10 ns
  iii. 按 `VPN = 0x1D` 查询页表，`Valid` 位为 `0`，触发 Page Fault（见课件《7-虚存》P.23）：用时 100 ns
  iv. 处理 Page Fault，更新页表与TLB：用时 108 ns **（见下面勘误）**
  v. 再次访问TLB，TLB Hit：用时 10 ns
  vi. 获得物理地址，访问内存取出数据：用时 100 ns
  总共用时：10 + 100 + 108 + 10 + 100 = 328 ns
- `0x9E6CF2`：
  i. `VPO = 0xCF2`，`VPN = 0x9E6`，`TLBI = 0x6`，`TLBT = 0x13C`
  ii. 查询 `index = 6` 的 TLB set，包含 TLB tag `0x13C`，TLB Hit：`PPN = 0x7F`，用时 10 ns
  iii. 获得物理地址，访问内存取出数据：用时 100 ns
  总共用时：10 + 100 = 110 ns

> ⚠ **勘误：实际缺页处理用时**
>
> 缺页处理需要向外存发出请求，取出新的页表加载到内存中，显然 外存访存速度 << 内存访存速度（回顾课件《3.1-C语言与汇编》P.7）。
> 本次作业出题时出现了笔误：处理缺页用时实际为 10^8 ns，尽管不影响作答，严谨起见在此特别纠正。

作业五 · Q2 重定位

在链接阶段需要全局重定位的：**被引用的** 全局变量、全局函数入口地址（见课件《6.1 汇编与C语言-5》P.25小结）。

> ✓ 作业解答节选
>
> 节选 计28/经22-左晨阳同学、计27-郭宇翼同学。

```c
// foo.c
extern void (*test1)();

static void test2() {}

void test3() {}

extern void test4();

void test_call(void (*test5)()) {
    test1();
    test2();
    test3();
    test4();
    test5();
}
```

Q2 test1是外部声明的函数指针，需要链接期重定位

test2是静态全局函数，不能被外部调用，不需要重定位

test3是非静态的全局函数，需要重定位

test4是外部定义的全局函数，需要链接期重定位

test5是传入的函数指针参数，不是全局变量，不需要重定位

```c
// foo.c
extern void (*test1)();   // 表明 test1 是一个函数指针，指向的函数无参数，无返回值，且是外部文件中定义的
static void test2() {}    // 静态全局，不重定位
void test3() {}           // 全局函数，重定位

extern void test4();

void test_call(void (*test5)()) {   // ① 传入一个函数指针，当局部变量
    test1();
    test2();
    test3();
    test4();
    test5();
}
// ② 注意：当其它代码中运行 test_call，且实参是一个外部文件的函数时 test5 也可能会全局重定位
```

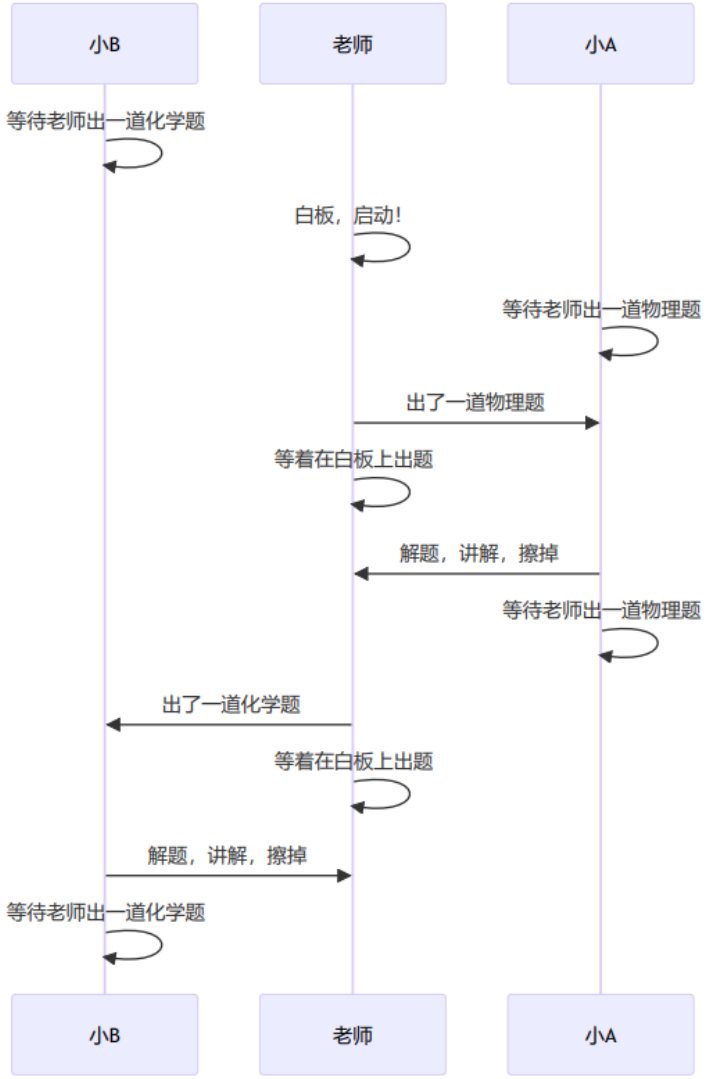
补：回答

作业六 · Q2 信号量

```
sem_t board;      // 白板是否为空
sem_t physics;    // 白板上是否为物理题
sem_t chemistry;  // 白板上是否为化学题

void init() {
    Sem_init(&board, 0, ___(A) ___);
    Sem_init(&physics, 0, ___(B) ___);
    Sem_init(&chemistry, 0, ___(C) ___);
}
void teacher() {
    while (1) {
        Course c = (rand() & 1) ? PHYSICS : CHEMISTRY;
        ____(D) ____;
        在白板上写题目;
        if (c == PHYSICS) {
            ____(E) ____;
        } else {  // c == CHEMISTRY
            ____(F) ____;
        }
    }
}
void studentA() {
    while (1) {
        P(___(G) ___);
        解答物理题，将其擦掉;
        V(___(H) ___);
    }
}
void studentB() {
    while (1) {
        P(___(I) ___);
        解答化学题，将其擦掉;
        V(___(J) ___);
    }
}
```

获得这样的序列图后，结合 `P`，`V` 原语可知：

- "自环"表示当前线程正在监听某个信号量直到其大于 `0`，对应于 `P()`；
- "单向传递"表示线程设置某个信号量以通知其他线程工作，对应于 `V()`。



作业六 · Q3 信号处理

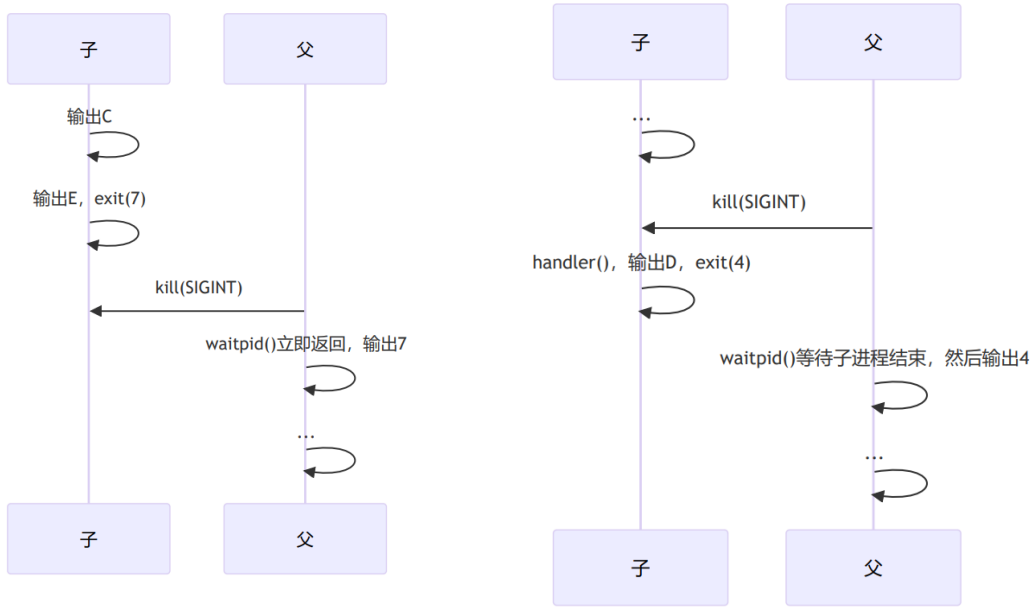
```c
void handler (int sig) {
    printf("D");
    exit(4);
}
int main() {
    int pid, status;
    signal(SIGINT, handler);
    printf("A");
// ----------------
    pid = fork();
    printf("B");
// ----------------
    if (pid == 0) {
        printf("C");
    } else {
        kill(pid, SIGINT);
        waitpid(pid, &status, 0);
        printf("%d", WEXITSTATUS(status));
    }
    printf("E");
    exit(7);
}
```

以下哪些是可能的输出结果（多选，根据标答完全匹配给分）：_____

A. `ABCBE7E` B. `ABD7E` C. `ABBCE4E` D. `ABCDB4E` E. `ABBD4E`

在 `fork()` 之前：

1. 通过 `signal()` 修改接收到信号 `SIGINT` 后，触发 `handler`
2. 在 `fork()` 之前只有一个进程，因此首先必定会输出一个 `A`：选项A,B,C,D,E符合



由上述分析可知，输出结果中 `4` 和 `D` 要么同时存在要么同时不存在，不可能只出现其中之一，选项C是不合理的；父进程与子进程都会输出一个 `B`，除非子进程被提前kill了（同上，`4` 和 `D` 同时存在），而选项B的输出只有一个 `B`，有 `D` 却没有 `4`，不合理。