

# 浙江大学

## 本科实验报告

课程名称:	汇编与接口
姓 名:	秦嘉俊
学 院:	计算机科学与技术学院
专 业:	计算机科学与技术
指导教师:	蔡铭

2023 年 11 月 5 日

# 浙江大学实验报告

课程名称: 汇编与接口 实验类型: 综合

实验项目名称: 步长机制探究实验

学生姓名: 秦嘉俊 学号: 3210106182 同组学生姓名: 秦嘉俊

实验地点: 玉泉 32 舍 411 实验日期: 2023 年 10 月 20 日

## 摘要

步长 (stride) 是 C 语言中的一个重要概念, 它与内存管理和指针操作密切相关。本次探究实验源于笔者在操作系统 lab2 实验中遇到的问题。笔者基于相关课程所学, 对步长机制进行了分析, 并通过实验验证了自己的结论。最后, 笔者还对编译器是如何处理步长进行了初步的探究。此外, 在探究实验中笔者还遇到了其他的问题, 如结构体对齐、PC 相对寻址等问题, 也一并记录在此。

## 一、背景说明

### 1.1 问题描述

本学期在完成操作系统 lab2 实验时, 我遇到了下面的问题:

这里我们在初始化线程的状态, 要把线程对应的信息 (如栈顶指针) 存放在线程对应的结构体中 (即 `task[i]` 指向的位置)。而 `task[i]` 指向的是 `kalloc()` 分配的一块内存的首地址, 这块内存用来存放线程的信息和栈空间。而栈一般是从高地地址长到低地址, 因此我选择将栈顶指针 `sp` 初始化为这一块内存的末尾地址 (这里内存的大小为 `PGSIZE` 字节), 于是便有了下面的代码:

```
// 这里的 task[i] 存放的是一个 struct *, 即一个结构体指针
// PGSIZE 是定义的一个宏, 表示一页的大小
for (int i = 1; i < NR_TASKS; i++) {
    task[i] = (struct task_struct*)kalloc();
    ...
}
```

```
task[i]->thread.sp = task[i] + PGSIZE; // kalloc 返回
    的是一页的首地址，即低地址
}
```

上面的代码看上去完成了我预期的任务，实际上却出现了问题。程序中我们的代码会调用 `sret` 语句，而执行完 `sret` 语句后程序并没有返回到我们预期的地址。而且在使用 GDB 调试时，我单步 `sret` 的下一条指令，程序就会卡住，无法再继续执行或者调试。

## 1.2 问题分析与解决

首先我查阅了 RISC-V 特权级 ISA 手册，了解 `sret` 语句背后到底发生了什么，看到下面的描述：

To support nested traps, each privilege mode  $x$  that can respond to interrupts has a two-level stack of interrupt-enable bits and privilege modes.  $xPIE$  holds the value of the interrupt-enable bit active prior to the trap, and  $xPP$  holds the previous privilege mode. The  $xPP$  fields can only hold privilege modes up to  $x$ , so MPP is two bits wide and SPP is one bit wide. When a trap is taken from privilege mode  $y$  into privilege mode  $x$ ,  $xPIE$  is set to the value of  $xIE$ ;  $xIE$  is set to 0; and  $xPP$  is set to  $y$ .

---

*For lower privilege modes, any trap (synchronous or asynchronous) is usually taken at a higher privilege mode with interrupts disabled upon entry. The higher-level trap handler will either service the trap and return using the stacked information, or, if not returning immediately to the interrupted context, will save the privilege stack before re-enabling interrupts, so only one entry per stack is required.*

An MRET or SRET instruction is used to return from a trap in M-mode or S-mode respectively. When executing an  $xRET$  instruction, supposing  $xPP$  holds the value  $y$ ,  $xIE$  is set to  $xPIE$ ; the privilege mode is changed to  $y$ ;  $xPIE$  is set to 1; and  $xPP$  is set to the least-privileged supported mode (U if U-mode is implemented, else M). If  $xPP \neq M$ ,  $xRET$  also sets MPRV=0.

---

*Setting  $xPP$  to the least-privileged supported mode on an  $xRET$  helps identify software bugs in the management of the two-level privilege-mode stack.*

图 1: xRET 机制

总结来说，在 RISC-V 中，我们可以使用 `sret` 语句从 S 模式返回，返回时会操作相对应的内核栈，具体栈操作在手册中有详细描述。这里我看到了 `sret` 会操作内核栈之后，我决定再次调试程序，查看内核栈的情况。

于是我单步调试到 `sret` 语句，然后查看当前的内核栈顶指针 `sp` 的值，并尝试查看内核栈的情况，发现此时 GDB 告诉我：Cannot access memory at address 0x88097000.

```

> 0x80200160 < dummy+12> sret
0x80200164 < switch_to> sd ra,48(a0)
0x80200168 < switch_to+4> sd sp,56(a0)
0x8020016c < switch_to+8> sd s0,64(a0)
0x80200170 < switch_to+12> sd s1,72(a0)
0x80200174 < switch_to+16> sd s2,80(a0)
0x80200178 < switch_to+20> sd s3,88(a0)
0x8020017c < switch_to+24> sd s4,96(a0)
0x80200180 < switch_to+28> sd s5,104(a0)
0x80200184 < switch_to+32> sd s6,112(a0)
0x80200188 < switch_to+36> sd s7,120(a0)
0x8020018c < switch_to+40> sd s8,128(a0)
0x80200190 < switch_to+44> sd s9,136(a0)
0x80200194 < switch_to+48> sd s10,144(a0)

remote Thread 1.1 In: dummy
0x88097000: Cannot access memory at address 0x88097000
(gdb) p/x $pc
$4 = 0x80200160
(gdb) p/x $sstatus
$5 = 0x80000000000006120
(gdb) p/x $sepc
$6 = 0x80200610
(gdb) x/5i $sepc
0x80200610 <dummy>: addi sp,sp,-48
0x80200614 <dummy+4>: sd ra,40(sp)
0x80200618 <dummy+8>: sd s0,32(sp)
0x8020061c <dummy+12>: addi s0,sp,48
0x80200620 <dummy+16>: jal ra,0x802016a8 <schedule_test>
(gdb) p/x $sstatus
$7 = 0x80000000000006120
(gdb) p/x $sp
$8 = 0x88097000
(gdb) x/5i $sp
0x88097000: Cannot access memory at address 0x88097000
(gdb)

```

图 2: 单步调试

我很惊讶，于是我重新回到了最开始初始化 `sp` 的代码部分（即问题提出部分的代码）并进行重新调试，中间我输出了 `task[i]` 的值，发现了问题

```

> 56 task[i]->thread.sp = task[i] + PGSIZE; // kalloc 返回的是一组物理地址，应该使用
57 }
58
59 printk("...proc_init done!\n");
60 }
61
62 // arch/riscv/kernel/proc.c
63 void dummy() {
64     schedule_test();
65     uint64 MIO = 1000000007;

0x80200570 <task_init+372> auipc a4,0x5
0x80200574 <task_init+376> addi a4,a4,-1368
0x80200578 <task_init+380> lv a5,-20(s0)
0x8020057c <task_init+384> slli a5,a5,0x3
0x80200580 <task_init+388> add a5,a4,a5
0x80200584 <task_init+392> ld a5,0(a5)
0x80200588 <task_init+396> auipc a4,0x3
0x8020058c <task_init+400> ld a4,-1392(a4)
0x80200590 <task_init+404> sd a4,48(a5)
> 0x80200594 <task_init+408> auipc a4,0x5
0x80200598 <task_init+412> addi a4,a4,-1404
0x8020059c <task_init+416> lv a5,-20(s0)
0x802005a0 <task_init+420> slli a5,a5,0x3
0x802005a4 <task_init+424> add a5,a4,a5
0x802005a8 <task_init+428> ld a4,0(a5)
0x802005ac <task_init+432> lui a5,0xa0
0x802005b0 <task_init+436> add a3,a4,a5
0x802005b4 <task_init+440> auipc a4,0x5

remote Thread 1.1 In: task_init
(gdb) n
(gdb) p/x task[i]
$1 = 0x87ffe000
(gdb) p/x task[i]+0x1000
$2 = 0x8809e000
(gdb)

```

图 3: 单步调试

可以看到这里（`PGSIZE=0x1000`），我们发现 `task[i]+PGSIZE` 应该等于 `0x87fff000`，而不是这里计算出来的 `0x8809e000`！

而且我找到了 `task[i]->thread.sp=task[i]+PGSIZE;` 所对应的汇编代码如下：

```

0x80200594 <task_init+408> auipc a4,0x5
0x80200598 <task_init+412> addi a4,a4,-1404
0x8020059c <task_init+416> lw a5,-20(s0)
0x802005a0 <task_init+420> slli a5,a5,0x3
0x802005a4 <task_init+424> add a5,a4,a5
0x802005a8 <task_init+428> ld a4,0(a5)
0x802005ac <task_init+432> lui a5,0xa0
0x802005b0 <task_init+436> add a3,a4,a5
0x802005b4 <task_init+440> auipc a4,0x5
0x802005b8 <task_init+444> addi a4,a4,-1436
0x802005bc <task_init+448> lw a5,-20(s0)
0x802005c0 <task_init+452> slli a5,a5,0x3
0x802005c4 <task_init+456> add a5,a4,a5
0x802005c8 <task_init+460> ld a5,0(a5)
0x802005cc <task_init+464> mv a4,a3
0x802005d0 <task_init+468> sd a4,56(a5)

```

图 4: RISC-V 汇编代码

这里结合其他部分的汇编代码可以知道，0x80200594~0x802005a8 是在取出 `task[i]` 的值到 `a4`。而 0x802005ac~0x802005b0 是在计算 `task[i]+PGSIZE` 并将结果存在 `a3` 中。随后 0x802005b4~0x802005c8 取出 `task[i]` 地址并放到 `a5` 中，这部分过程与之前相同。最后将计算出来的值存入 `task[i]->thread.sp` 中去。

这里可以看到，我们在计算 `+PGSIZE` 时，将 `a5` 设置为 0xA0000，而不是我们预期的 `PGSIZE` 的值 0x1000，这足足差了 160 倍！这时我已经想到了原因：C 语言在访问数组里元素时，`a[i]` 即表示第 `i` 个元素，但实际上它对应的地址是 `a+i*sizeof(a)`！也就是说，因为这里的 `task[i]` 是一个 `struct *`，编译器也认为这里是在访问数组元素，因此为我们填充了步长，因此最后翻译出来的汇编代码里的偏移量已经是 `PGSIZE` 的 160 倍，而这 160 恰好是这个结构体的大小。

最后，我将 C 语言代码改为：

```

// 这里的 task[i] 存放的是一个 struct *, 即一个结构体指针
// PGSIZE 是定义的一个宏，表示一页的大小
for (int i = 1; i < NR_TASKS; i++) {
    task[i] = (struct task_struct*)kalloc();

    ...
    task[i]->thread.sp = (int64)task[i] + PGSIZE;    //
    kalloc 返回的是一页的首地址，即低地址
}

```

这里我通过强制类型转换，将 `struct *` 当作无符号 64 位整数。可以看到这时的汇编代码里，我们会把 `a5` 设置为 0x1000，没有为我们自动填充步长。后面 `sret` 也能返回到预期地址。

```
> 55 task[i] -> thread.sp = (uint64)task[i] + PGSIZE;
56 // task[i] -> thread.sp = task[i] + PGSIZE; // kalloc 返回的是一页的首地址，即低地址
57 }

0x80200590 <task_init+404> sd a4,48(a5)
0x80200594 <task_init+408> auipc a4,0x5
0x80200598 <task_init+412> addi a4,a4,-1404
0x8020059c <task_init+416> lw a5,-20(s0)
0x802005a0 <task_init+420> slli a5,a5,0x3
0x802005a4 <task_init+424> add a5,a4,a5
0x802005a8 <task_init+428> ld a5,0(a5)
0x802005ac <task_init+432> mv a3,a5
0x802005b0 <task_init+436> auipc a4,0x5
0x802005b4 <task_init+440> addi a4,a4,-1432
0x802005b8 <task_init+444> lw a5,-20(s0)
0x802005bc <task_init+448> slli a5,a5,0x3
0x802005c0 <task_init+452> add a5,a4,a5
0x802005c4 <task_init+456> ld a5,0(a5)
> 0x802005c8 <task_init+460> lui a4,0x1
0x802005cc <task_init+464> add a4,a3,a4
0x802005d0 <task_init+468> sd a4,56(a5)
0x802005d4 <task_init+472> lw a5,-20(s0)
```

图 5: 修改后

## 二、探索过程

### 2.1 基本原理

在我们最开始学 C 语言的时候，我们知道可以通过 `int a[n];` 定义一个长度为 `n` 的 `int` 类型的数组，即数组共有 `n` 个元素，每个元素的类型为 `int`。如果要访问数组中的元素，我们可以通过 `a[i]` 来访问第 `i` 个元素（从 0 开始计数）。

后来学汇编时，数组的定义和使用与 C 语言等更高抽象层次的语言有所不同。因为汇编更接近底层，我们需要了解数据类型的大小，例如在现在的 64 位电脑上 `int` 类型的数据占 4 个字节，`long` 类型的数据占 8 个字节。而在使用汇编语言访问数组中的元素时，我们需要知道数组的首地址，然后通过偏移量来访问数组中的元素。这里的偏移量与 C 语言有所不同，`a[i]` 在汇编中对应的偏移量为 `i*sizeof(a)`，而不是 `i`。这是因为数组里的每个元素都会占 `sizeof(a)` 个字节，如 `int a[10];` 里有 10 个元素，每个元素占 8 字节，总共是 80 字节。而一般计算机的最小寻址单元都是字节，因此汇编里的偏移量就是字节偏移量，而不是元素偏移量。

而在抽象级别高于汇编语言的编程语言中，为了让用户更好地书写代码，不必过分在乎底层的细节（如元素的字节偏移量），编译器会为我们自动步长，这样我们访问数组里的元素就只需要使用下标即可。

### 2.2 步长机制的相关例子

#### 2.2.1 实验环境

Windows 10 64 位，Windows Subsystem for Linux (WSL) Ubuntu 20.04.5  
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1 20.04.2)

## 2.2.2 实例分析

本部分中使用的测试代码均已放在 `code/` 文件夹下。

- 首先我们分析最基本的情形：一维数组

C 代码如下：(code/1d.c)

```
#include <stdio.h>
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int main()
{
    for (int i = 0; i < 10; i++) {
        printf("%d\n", a[i]);
    }
}
```

随后我们使用 `gcc -c -g 1.c` 指令，即可将 C 源代码编译为 `1.o` 可重定位目标文件，随后 `gcc 1.o` 生成可执行文件。最后我们通过 `gdb a.out` 运行并调试程序。

```
B+> 0x55555555149 <main>          endbr64
0x5555555514d <main+4>          push    %rbp
0x5555555514e <main+5>          mov     %rsp,%rbp
0x55555555151 <main+8>          sub     $0x10,%rsp
0x55555555155 <main+12>         lea     0x2ec4(%rip),%rax      # 0x555555558020 <a>
0x5555555515c <main+19>         mov     %rax,-0x8(%rbp)
0x55555555160 <main+23>         movl    $0x0,-0xc(%rbp)
0x55555555167 <main+30>         jmp     0x55555555197 <main+78>
0x55555555169 <main+32>         mov     -0xc(%rbp),%eax
0x5555555516c <main+35>         cltq
0x5555555516e <main+37>         lea     0x0(,%rax,4),%rdx
0x55555555176 <main+45>         lea     0x2ea3(%rip),%rax      # 0x555555558020 <a>
0x5555555517d <main+52>         mov     (%rdx,%rax,1),%eax
0x55555555180 <main+55>         mov     %eax,%esi
0x55555555182 <main+57>         lea     0xe7b(%rip),%rdi      # 0x555555556004
0x55555555189 <main+64>         mov     $0x0,%eax
0x5555555518e <main+69>         callq   0x55555555050 <printf@plt>
0x55555555193 <main+74>         addl    $0x1,-0xc(%rbp)
0x55555555197 <main+78>         cmpl    $0x9,-0xc(%rbp)
0x5555555519b <main+82>         jle     0x55555555169 <main+32>
0x5555555519d <main+84>         mov     $0x0,%eax
0x555555551a2 <main+89>         leaveq
0x555555551a3 <main+90>         retq
```

图 6: 汇编代码

其中，我们关心的步长部分在 `main+37~main+57` 的地方。这里首先我们的循环遍历 `i` 存在寄存器 `%rax` 中，于是通过 `lea 0x0(,%rax,4), %rdx` 取出下标对应的偏移量。可以看到这里的 `4` 就是 `sizeof(a)`，即步长。随后的一

条 `lea` 指令取出数组 `a` 的首地址。（但是这里很奇怪的是出现了 `%rip` 寄存器，这样的寻址方式之前没有学过，具体原因可见 3.1 部分）

随后 `mov (%rdx, %rax, 1), %eax` 相当于从内存中取 `a+sizeof(a)*i` 地址的值，即 `a[i]` 并存到 `%eax` 中。至此我们取出了数组中的元素，可以看到步长主要是由 `lea` 实现的。

我之前自己看过 *Computer Systems: A Programmer's Perspective* 这本书，里面有介绍过 x86-64 汇编在进行数组访问的一个特点：x86-64 的内存引用指令可以通过形如 `movl (%rdx, %rcx, 4), %eax` 的方式来计算内存地址（这里 `$rdx` 存放数组首地址，`%rcx` 存放下标 `i`，4 表示步长），这样可以简化数组访问。我想这样的设计，可能一方面就是为了让数组的步长在汇编代码中得到更好的体现吧。

- 接着我们分析二维数组的情形：

C 代码如下：(code/2d.c)

```
#include <stdio.h>
long long a[5][5] = {
    0, 1, 2, 3, 4,
    5, 6, 7, 8, 9,
    10, 11, 12, 13, 14,
    15, 16, 17, 18, 19,
    20, 21, 22, 23, 24
};
int main()
{
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++) {
            printf("%lld\n", a[i][j]);
        }
}
```

我们用和刚刚一样的办法得到汇编代码如下：



```

3+> 0x55555555149 <main> endbr64
0x5555555514d <main+4> push %rbp
0x5555555514e <main+5> mov %rsp,%rbp
0x55555555151 <main+8> sub $0x10,%rsp
0x55555555155 <main+12> movl $0x0,-0x8(%rbp)
0x5555555515c <main+19> jmp 0x555555551b5 <main+108>
0x5555555515e <main+21> movl $0x0,-0x4(%rbp)
0x55555555165 <main+28> jmp 0x555555551ab <main+98>
0x55555555167 <main+30> mov -0x4(%rbp),%eax
0x5555555516a <main+33> movslq %eax,%rcx
0x5555555516d <main+36> mov -0x8(%rbp),%eax
0x55555555170 <main+39> movslq %eax,%rdx
0x55555555173 <main+42> mov %rdx,%rax
0x55555555176 <main+45> shl $0x2,%rax
0x5555555517a <main+49> add %rdx,%rax
0x5555555517d <main+52> add %rcx,%rax
0x55555555180 <main+55> lea 0x0(,%rax,8),%rdx
0x55555555188 <main+63> lea 0x2e91(%rip),%rax # 0x555555558020 <a>
0x5555555518f <main+70> mov (%rdx,%rax,1),%rax
0x55555555193 <main+74> mov %rax,%rsi
0x55555555196 <main+77> lea 0xe67(%rip),%rdi # 0x555555556004
0x5555555519d <main+84> mov $0x0,%eax
0x555555551a2 <main+89> callq 0x55555555050 <printf@plt>
0x555555551a7 <main+94> addl $0x1,-0x4(%rbp)
0x555555551ab <main+98> cmpl $0x4,-0x4(%rbp)
0x555555551af <main+102> jle 0x55555555167 <main+30>
0x555555551b1 <main+104> addl $0x1,-0x8(%rbp)
0x555555551b5 <main+108> cmpl $0x4,-0x8(%rbp)
0x555555551b9 <main+112> jle 0x5555555515e <main+21>
0x555555551bb <main+114> mov $0x0,%eax
0x555555551c0 <main+119> leaveq
0x555555551c1 <main+120> retq

```

图 7: 汇编代码

其中，我们关心的步长部分在 `main+30~main+74` 的地方。首先这里是两层循环，我们的 `i` 和 `j` 分别存在栈上 `-0x8(%rbp)` 和 `-0x4(%rbp)` 的位置。我们把循环变量 `i` 取出来放到 `%rdx`, `%rax` 中，通过 `%rax<<2+%rdx` 得到 `5*i`，随后再把循环遍历变量 `j` 与 `%rax` 相加，得到 `5*i+j`，即 `a[i][j]` 展开为一维数组的下标 `a[5*i+j]`。接着我们把下标转换为字节偏移量，方法和上面相同：因为这里是 `long long` 类型，

可以看到 `main+55` 的 `lea 0x0(,%rax,8), %rdx` 的元组第三个元素是 8，即步长为 8。

随后 `mov (%rdx, %rax, 1), %rax` 相当于从内存中取 `a+8*(5*i+j)` 地址的值，即 `a[i][j]` 并存到 `%rax` 中。至此我们取出了二维数组中的元素。

- 然后我们分析结构体数组的情形：(code/struct.c)

C 代码如下：

```

#include <stdio.h>
struct a {
    char a1;
    int a2;
    long long a3;
}tmp[5];

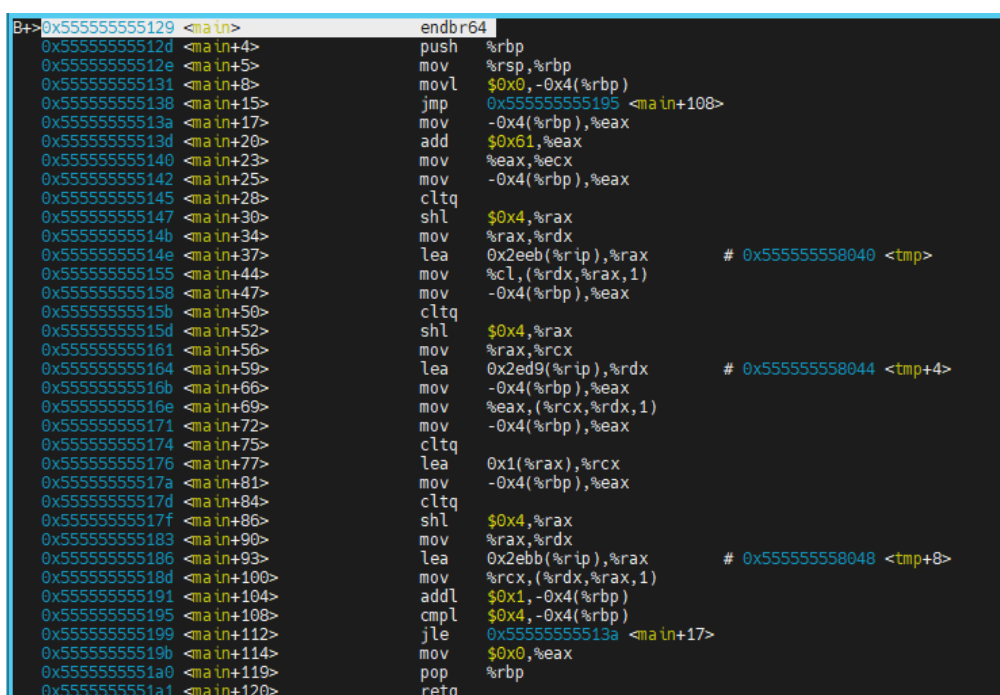
```

```

int main()
{
    for (int i = 0; i < 5; i++) {
        tmp[i].a1 = 'a' + i;
        tmp[i].a2 = i;
        tmp[i].a3 = i + 1;
    }
}

```

我们用和刚刚一样的办法得到汇编代码如下：



```

0x55555555129 <main>          endbr64
0x5555555512d <main+4>        push    %rbp
0x5555555512e <main+5>        mov     %rsp,%rbp
0x55555555131 <main+8>        movl    $0x0,-0x4(%rbp)
0x55555555138 <main+15>       jmp     0x55555555195 <main+108>
0x5555555513a <main+17>       mov     -0x4(%rbp),%eax
0x5555555513d <main+20>       add     $0x61,%eax
0x55555555140 <main+23>       mov     %eax,%ecx
0x55555555142 <main+25>       mov     -0x4(%rbp),%eax
0x55555555145 <main+28>       cltq
0x55555555147 <main+30>       shl     $0x4,%rax
0x5555555514b <main+34>       mov     %rax,%rdx
0x5555555514e <main+37>       lea     0x2eeb(%rip),%rax      # 0x555555558040 <tmp>
0x55555555155 <main+44>       mov     %cl,(%rdx,%rax,1)
0x55555555158 <main+47>       mov     -0x4(%rbp),%eax
0x5555555515b <main+50>       cltq
0x5555555515d <main+52>       shl     $0x4,%rax
0x55555555161 <main+56>       mov     %rax,%rcx
0x55555555164 <main+59>       lea     0x2ed9(%rip),%rdx      # 0x555555558044 <tmp+4>
0x55555555166 <main+66>       mov     -0x4(%rbp),%eax
0x5555555516e <main+69>       mov     %eax,(%rcx,%rdx,1)
0x55555555171 <main+72>       mov     -0x4(%rbp),%eax
0x55555555174 <main+75>       cltq
0x55555555176 <main+77>       lea     0x1(%rax),%rcx
0x5555555517a <main+81>       mov     -0x4(%rbp),%eax
0x5555555517d <main+84>       cltq
0x5555555517f <main+86>       shl     $0x4,%rax
0x55555555183 <main+90>       mov     %rax,%rdx
0x55555555186 <main+93>       lea     0x2ebb(%rip),%rax      # 0x555555558048 <tmp+8>
0x5555555518d <main+100>      mov     %rcx,(%rdx,%rax,1)
0x55555555191 <main+104>      addl    $0x1,-0x4(%rbp)
0x55555555195 <main+108>      cmpl    $0x4,-0x4(%rbp)
0x55555555199 <main+112>      jle     0x5555555513a <main+17>
0x5555555519b <main+114>      mov     $0x0,%eax
0x555555551a0 <main+119>      pop     %rbp
0x555555551a1 <main+120>      retq

```

图 8: 汇编代码

其中，我们关心的步长部分在 `main+25~main+44` 的地方。首先这里是一层循环，我们的 `i` 存在栈上 `-0x4(%rbp)` 的位置。我们把循环变量 `i` 取出来放到 `%eax` 中，并符号拓展，通过 `%rax<<2` 得到 `16*i`，随后再把 `tmp` 的首地址与 `%rax` 相加，得到 `tmp+16*i`，即 `tmp[i]`，最后根据结构体变量的位置，将要修改的值存入对应的偏移地址即可（这里是由 `lea offset(%rip), %rax` 这条指令来实现的）

值得注意的是，理论上这里我们的结构体定义里应该只有 `1(char)+4(int)+8(long long)=13` 个字节，但是在汇编代码中，我们的步长是 16（左移 4 位，相当于乘 16）。具体原因可见 3.2 部分。

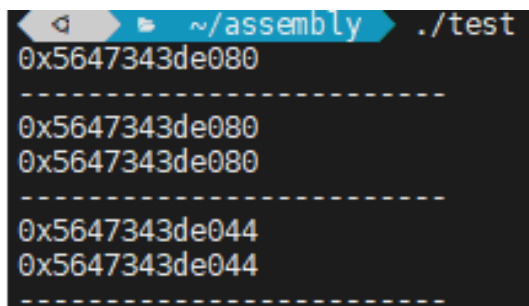
- 最后我们分析指针的情形，这里与我们最开始发现问题的代码类似，因此这里不再做详细的分析，而是观察编译器什么时候会为我们填充步长。

C 代码如下: (code/pointer.c)

```
#include <stdio.h>
struct a {
    char a1;
    int a2;
    long long a3;
}tmp[5];

int main()
{
    printf("%p\n", &tmp[4]);
    struct a* point_to_tmp = &tmp[0];
    printf("-----\n");
    long long b;
    b = (long long)(point_to_tmp + 4);
    printf("0x%llx\n", b);
    struct a* bb;
    bb = point_to_tmp + 4;
    printf("0x%llx\n", (long long)bb);
    printf("-----\n");
    b = (long long)point_to_tmp + 4;
    printf("0x%llx\n", b);
    bb = (struct a*)((long long)point_to_tmp + 4);
    printf("0x%llx\n", (long long)bb);
    printf("-----\n");
}
```

这次我们直接在终端里输入 `gcc 2.c -o test` 随后直接运行 `./test` 即可，看到输出如下：



```

~/.assembly ./test
0x5647343de080
-----
0x5647343de080
0x5647343de080
-----
0x5647343de044
0x5647343de044
-----

```

图 9: 输出

这里我们先输出 `tmp[4]` 的地址，随后通过尝试用整型变量 `b` 和结构体指针 `bb` 来获取地址。第一部分的值是我们直接将指针用于计算，第二部分的值是我们先将指针强制类型转换为 64 位整数再进行计算。

可以看到，编译器是否会填充步长，取决于在运算过程中是否有数组/指针的类型出现。

如 `b = (long long)(point_to_tmp + 4); bb = point_to_tmp + 4;` 均会填充步长，即使 `b` 并不是指针类型而只是一个整型常量。反过来看，即使 `bb` 是一个指针类型，但是因为赋值运算符的右边并没有出现数组/指针的类型，编译器也不会为我们填充步长，只是把 `point_to_tmp` 当作一个简单的整型常量来处理。

这一点，我在上网搜搜相关资料结论时发现了不同的说法，有人认为：指针的步长在定义指针的时候就已经确定了。但是在刚刚的示例中，我们可以看到即使是指针，有的时候也不会被填充步长，这让我对他的说法产生了怀疑。于是我又设计了下面的例子来检验这一点：(`code/stride.c`)

```

#include <stdio.h>
struct a {
    char a1;
    int a2;
    long long a3;
}tmp[5];

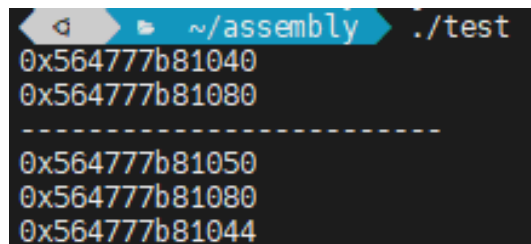
int main()
{
    printf("%p\n", &tmp[0]);
    printf("%p\n", &tmp[4]);
    struct a* point_to_tmp = &tmp[0];

```

```

printf("-----\n");
int *b;
b = (int *)point_to_tmp + 4;
printf("%p\n", b);
b = (int *)(point_to_tmp + 4);
printf("%p\n", b);
b = (int *)((long long)point_to_tmp + 4);
printf("0x%llx\n", (long long)b);
}

```



```

~/assembly ./test
0x564777b81040
0x564777b81080
-----
0x564777b81050
0x564777b81080
0x564777b81044

```

图 10: 输出

可以看到，这里我们把一个结构体指针经过运算后的值赋值给一个 `int` 指针。而只有在我们将 `struct *` 强制类型转换为 `int *` 时步长才是 4 个字节，其他时候并不如此。这说明了指针的步长并不是在定义指针的时候就已经确定了，而是在运算过程中根据运算符右边的类型来确定的。因此我认为我的结论是正确的，而不是上面那个人的说法。

### 2.2.3 编译器如何处理步长

无论是网上的资料，还是课内所学都停在了上一步，即老师告诉了我们编译器会为我们做填充步长的工作，但并没有告诉我们编译器是如何做，什么时候做的。本着求知探索的精神，我决定继续往下看，编译器是如何处理步长的。

但是因为本身我们还没有学过《编译原理》这门课，我对编译器的了解不多，因此过程中肯定会遇到看不懂的地方，我的想法是目标导向，即我只要知道编译器是如何处理步长的，而不必深究编译器的实现细节。

首先我从网上补充了一些编译器的相关知识：

- 对于输入进来的程序，编译器将其视作一个很长的字符串，首先进行预处理和词法分析；随后是语法和语义分析。这两部分是编译器的前端。一个编译器有不同的前端，用来对应不同的语言。比如 GCC 有 C++ 前端、C 前端、Fortran 前端等。

- 随后编译器会生成中间表示并进行优化，然后送到不同架构的后端生成对应的汇编代码，如 GCC 有 x86-64 的后端，arm 的后端等等。
- 我对编译器的几个阶段进行了了解，推测步长功能应该是在前端的语法分析或者语义阶段完成的。

为了更好的分析，我下载了 GCC 的源码：

```
$ wget http://ftp.gnu.org/gnu/gcc/gcc-9.4.0/gcc-9.4.0.tar.gz
$ tar -zxvf gcc-9.4.0.tar.gz
$ cd gcc-9.4.0
$ code .
```

在 `gcc-9.4.0/gcc` 文件夹下，可以看到有各种语言的前端代码，这里我主要在 `c` 和 `c-family` 中通过关键词寻找。经过大半天的搜索，最后我在 `gcc/c/c-fold.c` 中找到了我想要的答案，`c_fold_array_ref` 函数。

```
/* Try to fold ARRAY_REF ary[index] if possible and not handled by
normal fold, return NULL_TREE otherwise. */

static tree
c_fold_array_ref (tree type, tree ary, tree index)
{
    ...
    tree elem_type = TREE_TYPE (TREE_TYPE (ary));
    unsigned elem_nchars = (TYPE_PRECISION (elem_type)
                           / TYPE_PRECISION (char_type_node));
    unsigned len = (unsigned) TREE_STRING_LENGTH (ary) /
        elem_nchars;
    ...
    const unsigned char *ptr
    = ((const unsigned char *)TREE_STRING_POINTER (ary) + i *
       elem_nchars);
    return native_interpret_expr (type, ptr, elem_nchars);
}
```

可以看到，在语法分析阶段，编译器会遍历 AST 树，如果遇到了数组访问，会通过 `elem_nchars` 来确定步长，即数组每个元素的大小，相当于 `sizeof()`。随后会通过 `native_interpret_expr` 产生一个引用，而这里的指针 `ptr`，就已经是进行步长伸缩之后的地址了（`ary + i * elem_nchars`）。

也就是说，在语法分析阶段，编译器就已经为我们填充了步长，这样在编译器后端转化为汇编代码时，就已经是字节偏移量了。

目前为止，我们基本上了解了步长机制的完整过程，从编译器里的处理，到 C 语言里的不同写法，都有了一个基本的认识。

## 三、效果与结论

本次探究步长机制的实验的结论如下：

- C 语言中，数组的步长机制是由编译器在语法分析阶段完成的，编译器会为我们填充步长，这样在编译器后端转化为汇编代码时可以直接转换为字节偏移量。
- 编译器是否会填充步长，取决于在运算过程中是否有数组/指针的类型出现。如果表达式中没有数组/指针类型，即使左值是指针类型，也不会填充步长；如果表达式中有数组/指针类型，即使左值不是指针类型，也会填充相对应的步长。

## 四、实验中遇到的其他问题及探索

实际上这部分内容与步长机制无关，但是是在我们实验过程中（如调试汇编的时候）遇到的问题，实验更重要的是学到东西，增长见识，因此我将与主题无关的问题写在这里。

### 4.1 重定位 PC 相对引用

在 2.2.2 中，我们遇到了汇编代码中使用 `%rip` 来寻址的方式，这种方式之前并没有学过，我翻阅了 *Computer Systems: A Programmer's Perspective* 里链接这一章，结合网上的资料，我了解到这是一种重定位 PC 相对引用的方式。

以 2.2.2 中第一个 C 语言程序为例，我们生成 `test.o` 和 `test.out`，并分别通过 `objdump -d test.o` 和 `objdump -d test.out` 反汇编出原来的汇编代码。

```

0000000000000000 <main>:
0: f3 0f 1e fa          endbr64
4: 55                   push    %rbp
5: 48 89 e5             mov     %rsp,%rbp
8: 48 83 ec 10          sub     $0x10,%rsp
c: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
13: e0 2e               jmp     43 <main+0x43>
15: 8b 45 fc             mov     -0x4(%rbp),%eax
18: 48 98               cltq
1a: 48 8d 14 85 00 00 00 lea     0x0(,%rax,4),%rdx
21: 00
22: 48 8d 05 00 00 00 00 lea     0x0(%rip),%rax    # 29 <main+0x29>
29: 8b 04 02             mov     (%rdx,%rax,1),%eax
2c: 89 c6               mov     %eax,%esi
2e: 48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi    # 35 <main+0x35>
35: b8 00 00 00 00      mov     $0x0,%eax
3a: e8 00 00 00 00      callq   3f <main+0x3f>
3f: 83 45 fc 01         addl    $0x1,-0x4(%rbp)
43: 83 7d fc 09         cmpl    $0x9,-0x4(%rbp)
47: 7e cc               jle     15 <main+0x15>
49: b8 00 00 00 00      mov     $0x0,%eax
4e: c9                 leaveq   %eax
4f: c3                 retq

0000000000000119 <main>:
119: f3 0f 1e fa          endbr64
11d: 55                   push    %rbp
11e: 48 89 e5             mov     %rsp,%rbp
11f: 48 83 ec 10          sub     $0x10,%rsp
120: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
123: e0 2e               jmp     118c <main+0x43>
125: 8b 45 fc             mov     -0x4(%rbp),%eax
128: 48 98               cltq
12a: 48 8d 14 85 00 00 00 lea     0x0(,%rax,4),%rdx
131: 00
132: 48 8d 05 ae 2e 00 00 lea     0x2eae(%rip),%rax    # 4020 <a>
139: 8b 04 02             mov     (%rdx,%rax,1),%eax
13c: 89 c6               mov     %eax,%esi
13e: 48 8d 3d 86 0e 00 00 lea     0xe86(%rip),%rdi    # 2004 <_IO_stdin_used+0x4>
145: b8 00 00 00 00      mov     $0x0,%eax
148: e8 c8 fe ff ff      callq   1050 <printf@plt>
14b: 83 45 fc 01         addl    $0x1,-0x4(%rbp)
14e: 83 7d fc 09         cmpl    $0x9,-0x4(%rbp)
151: 7e cc               jle     115e <main+0x15>
153: b8 00 00 00 00      mov     $0x0,%eax
158: c9                 leaveq   %eax
159: c3                 retq
15a: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)

```

(a) test.o

(b) test.out

可以看到，在 **test.o** 中使用 **%rip** 寻址的地方的偏移量都是 0，而在 **test.out** 中，这些偏移量都被填充了。

这是因为在我们的代码中，array **a** 是一个全局变量，在编译时编译器并不能确定它的地址，因此在 **test.o** 中，编译器只能将这些偏移量填充为 0；在加载时，加载器会把重定位后计算出来的偏移量直接复制到内存（即机器码中 00 字节的部分），这样我们就不需要修改指令，就可以正确执行这些指令。

## 4.2 结构体对齐问题

在 2.2.2 中，我们的结构体同时包含了 **char**、**int**、**long long** 类型的变量，理论上这个结构体应该只有 1+4+8=13 个字节，但实际上它所占用的大小却是 16 字节。这让我想起了《计算机组成与设计》课上老师所讲的对齐问题：部分硬件设计要求我们访问的地址必须是某个值 **K** 的倍数（一般为 2、4、8）。

对于 x86-64 而言，我了解到对于大多数 x86-64 指令保持数据对齐可以提高效率，但并不会影响程序的正确性。在我们之前的结构体的例子中，我们在汇编代码可以看到 **.align 8**，即要求我们的结构体 8 字节对齐，因此实际的排布应该是

char	int	long long
0	4~7	8~15

## 四、实验体会与经验教训

在做探究实验中，我感到了学习计算机科学的挑战和乐趣，同时也为我提供了宝贵的经验和教训。

首先，这次实验让我重新认识了步长（Stride）这一重要概念。虽然这个机制并没有什么复杂的地方，在《程序设计与算法基础》里就讲过。但当时我只是简单地记住了这个用法，没有深入思考。在这个实验中，我探索了 C 语言中步长机制的实



现，以及编译器是如何处理步长的。我还深入研究了底层的汇编语言，以理解步长在计算机系统在实际运作方式。通过这个过程，我对计算机内部的工作原理有了更清晰的认识。

在探索的过程中，我也遇到了很多问题，但是与以前不同的地方在于：探索中的问题并不是教科书上规范的问题，有标准的答案，网上也有现成的参考；相反我需要利用已经学到的知识，结合能搜到的相关内容，动手实践，最后自己思考分析问题原因以及原理。

比如探索中的 PC 相对引用部分，虽然我在操作系统课程中听到老师提到这个概念，也在书上看到相关内容，但和我实际遇到的例子是相差甚远的。而我通过对比.o和.out 的汇编代码，结合理论知识，才能够对现象进行解释。这也是一个从理论到实践的过程。

此外，这次实验还让我感受到了之前所学的知识如何串联在一起。这次实验里我用到了《程序设计与算法基础》、《计算机组成与设计》、《操作系统原理与实践》、《汇编语言》的相关知识，以前学习的过程中他们相对独立，可能感受不到他们之间的联系。但是在我掌握的好的基础上，用这些知识来解决实际问题，就会有融会贯通之感，这或许也是计算机系统的魅力所在吧。

我还有一个教训是，**尽信书不如无书**。如关于指针的步长机制，我如果没有亲自实验，而是直接照搬网上搜到的结论，那就犯了大错。实际上写博客的也是人，他是在自己的环境上跑的实验，也可能受到特定环境的影响，而且结论并没有经过权威验证。我应该时刻保持怀疑的态度，**大胆猜想，谨慎求证，独立思考**，才能够真正掌握知识。

总的来说，这次《汇编与接口》课程的探究实验让我受益匪浅。我不仅加深了对计算机系统内部原理的理解，还培养了自主学习和解决问题的能力。我相信这些经验和教训将在我的学术和职业生涯中发挥重要作用，帮助我不断进步。通过将理论知识转化为实践经验，我将更自信地面对未来的计算机科学挑战。这个实验不仅仅是课程要求，更是我个人成长和职业发展的一部分。

综上所述，如有错误还请老师批评指正。

## 五、参考资料

- Computer Systems: A Programmer's Perspective, 3rd Edition Chapter 3: Machine-Level Representation of Programs and Chapter 7: Linking
- 知乎文章 [GCC 的整体架构](https://zhuanlan.zhihu.com/p/37252649)<https://zhuanlan.zhihu.com/p/37252649>
- 《计算机组成与设计》的课程 PPT