1、nested函数共有9个参数，其中存在两个聚合体，struct small和struct big，根据LoongArch64的函数调用约定，即LP64的API调用规范，

* 参数a, b, c, d均使用寄存器来传递，分别对应a0, a1, a2，a3寄存器，但是因为a, b, c小于64位，所以会先符号扩展到64位再进行传递。
* 浮点数e为float型，跟聚合体参数传递方式一致，使用一个寄存器来进行传递，对应a4寄存器。
* 浮点数f为double型，根据调用约定，其大于XLEN但少于2\*XLEN位，所以可以使用一对寄存器来传递，对应a5,a6。
* 结构体g为struct small，其大小小于XLEN,所以可以使用寄存器来进行传递，对应a7，其中布局和small结构体内的布局一致。
* 结构体h的参数传递由于已经没有可用的寄存器，所以在栈上进行传递，会填充一些未定义的位。
* 整型i是long型，已无可用的寄存器，所以也通过栈进行传递。
* 返回值则使用a0寄存器进行传递。

2、

（1）、

示例程序：

//共享变量

.global

shared\_var: .word 0

flag: .char 0

//线程函数

ld.w a0, share\_var

addi a0, a0, 1

st.w a0, share\_var

st.b a1, flag

对于该线程函数，如果两个线程几乎同时运行这段代码，它们可能会读取share\_var的相同值，然后各自增加1并存储回去。这将导致share\_var只增加1，而不是预期的2，因为一个线程的更新被另一个线程覆盖了。

(2)、

利用

ll.w a0, share\_var

addi a0, a0, 1

sc.w a1, a0, share\_var

beqz a1, 1b

st.b a1, flag

这个版本中ll.w指令读取share\_var的值，并将其链接到后续的sc.w指令。如果share\_var在ll.w和sc.w之间没有被其他线程修改，sc.w就会将新值存储到share\_var，并将a1设置为0。否则，sc.w不会存储新值，并将a1设置为非零值。beqz指令检查a1是否为零，如果不是（表示存储失败），它就会跳转回标签1，重新尝试整个操作。这样，我们就可以确保share\_var的更新是原子的，不会被其他线程的操作干扰。同样，对flag的更新也应该在确保data已经成功更新之后进行。

3、

C语言源代码如下：

#include <stdio.h>

void bubbleSort(int arr[], int n) {

    for (int i = 0; i < n-1; i++) {

        for (int j = 0; j < n-i-1; j++) {

            if (arr[j] > arr[j+1]) {

                int temp = arr[j];

                arr[j] = arr[j+1];

                arr[j+1] = temp;

            }

        }

    }

}

int main() {

    int arr[] = {64, 34, 25, 12, 22, 11, 90};

    int n = 7

    bubbleSort(arr, n);

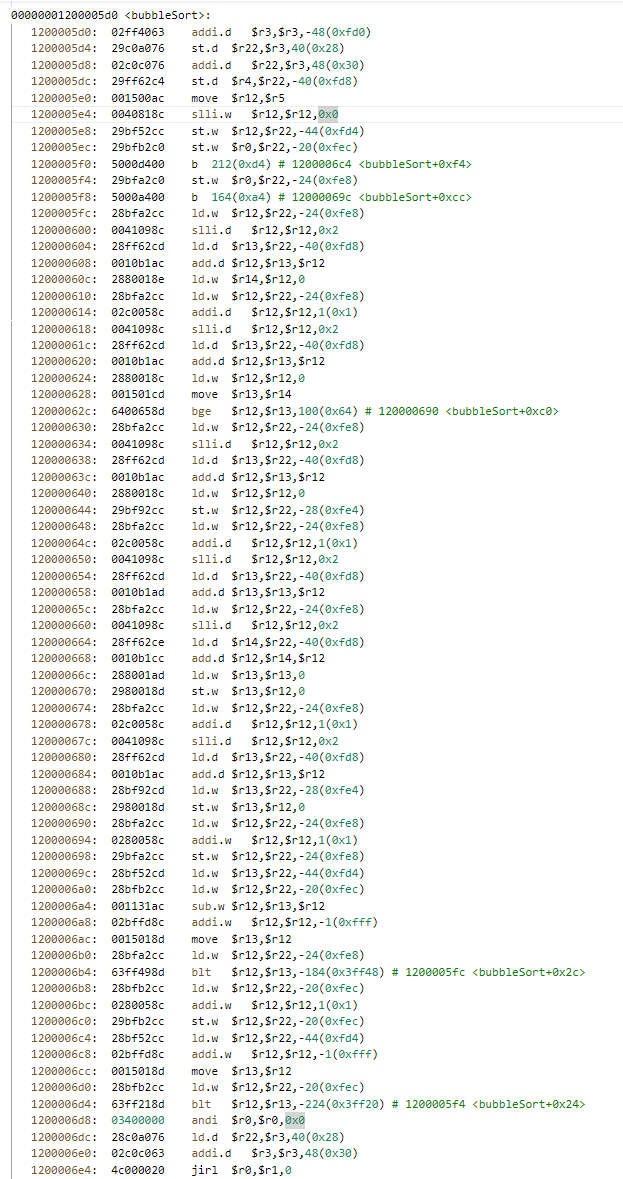
    return 0;

}

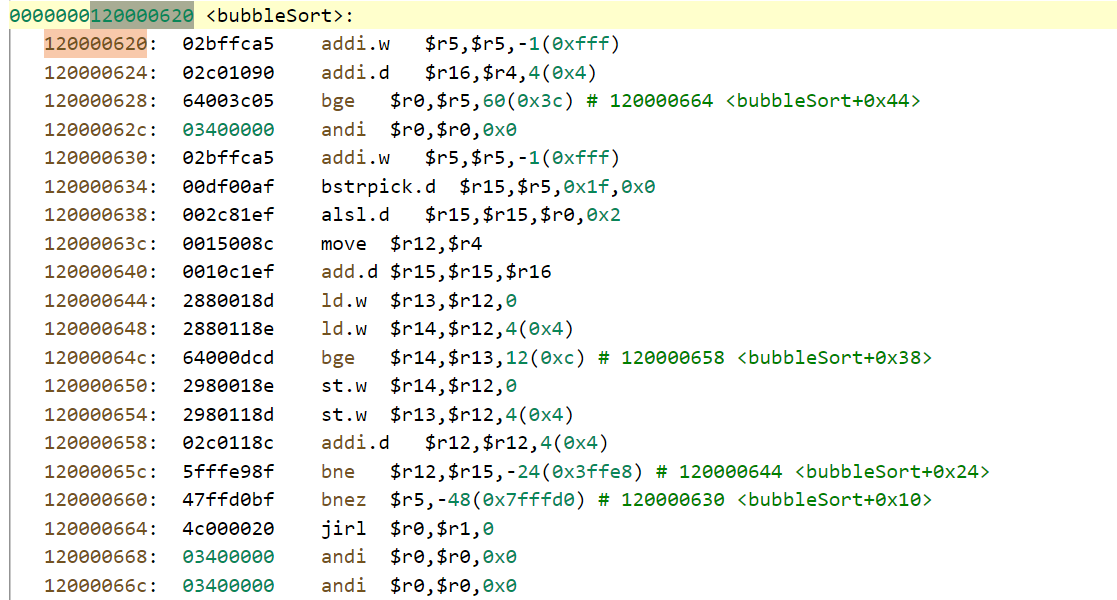
执行：

stu@stu:~/CAL/mdf$ loongarch64-linux-gnu-gcc test.c -o test  
stu@stu:~/CAL/mdf$ loongarch64-linux-gnu-gcc test.c -o test.o  
stu@stu:~/CAL/mdf$ loongarch64-linux-gnu-objdump -d -S test.o > test.S  
stu@stu:~/CAL/mdf$ loongarch64-linux-gnu-gcc test.c -o test.o -O2  
stu@stu:~/CAL/mdf$ loongarch64-linux-gnu-objdump -d -S test.o > test2.S

查看汇编文件，主要是bubble\_sort部分，如下



利用-O2进行编译优化，其编译出的汇编文件如下：



对比发现，-O2编译优化下其函数部分精简了不少，优化后的冒泡排序算法采用了更少的跳转指令，循环内部占有更大部分，而未优化之前在循环逻辑判断部分花费了更大的开销，造成损失。另外-O2会进行循环展开，循环不变量提取，充分利用流水线的特点，访存指令也会在代码中的比例会更少，以减少访存带来的损失。

显然，-O2编译后的冒牌排序算法，性能会更好。

4、

C语言程序：

#include <stdio.h>

struct MixedData {

    char c;

    short s;

    int i;

    long l;

    float f;

    double d;

    long double ld;

};

int main() {

    struct MixedData data;

    printf("Size of structure: %lu bytes\n", sizeof(data));

    return 0;

}

输出为：Size of structure: 48 bytes

重新修改Struct结构体内的成员布局为：

struct MixedData {

char c；

long double ld;

    double d;

    long l;

    float f;

    int i;

    short s;

}

输出变为：Size of structure: 64 bytes

结构体的每个元素都会对齐到其自身大小的倍数的地址上。例如，int 类型通常会对齐到4字节的地址上，double 类型会对齐到8字节的地址上。这也就是为什么当ld在c后面时，会增加额外的内存空间，因为其要对齐到一个较大的内存地址上。

5、

#include <unistd.h>

int main() {

    char c;

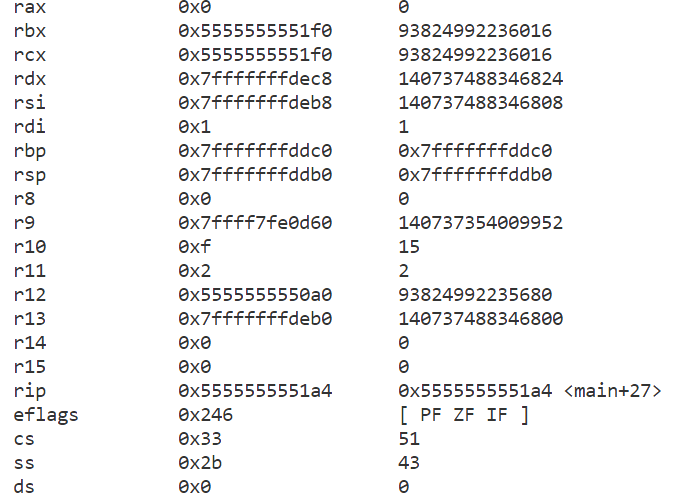
    read(STDIN\_FILENO, &c, 1);

    write(STDOUT\_FILENO, &c, 1);

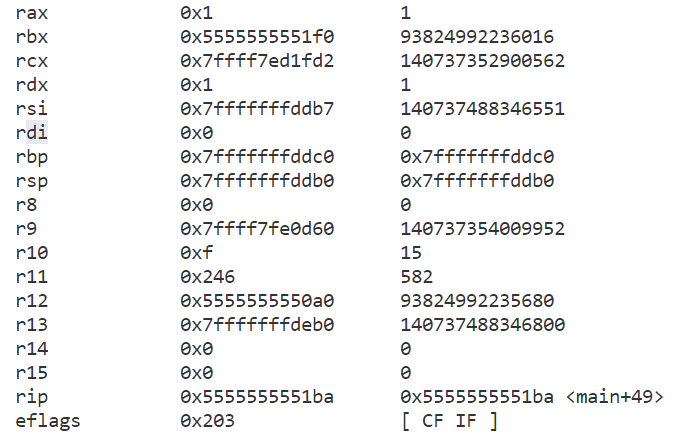
    return 0;

}

系统调用前，寄存器信息如下，注意已经执行完系统调用传参的部分，，可以看到其中的系统调用寄存器内值已被修改。



执行完系统调用的syscall后，同样其中rax, rdi, rsi, rdx寄存器的值均被修改，rip寄存器中的地址也被更新。



在x86\_64（也就是64位）系统上，系统调用利用四个寄存器来进行传参，其中系统调用号放在rax寄存器中。系统调用的参数依次放在rdi、rsi、rdx、r10、r8和r9寄存器中。执行syscall指令进行系统调用。系统调用的返回值在rax寄存器中。

上述两句代码等效于：

asm volatile (

        "mov $0, %%rdi\n"        // 文件描述符0代表标准输入

        "mov %0, %%rsi\n"        // rsi寄存器保存字符的地址

        "mov $1, %%rdx\n"        // 读取1个字符

        "mov $0, %%rax\n"        // 系统调用号0代表read

        "syscall\n"              // 执行系统调用

        :                        // 输出部分为空

        : "r" (&c)               // 输入部分，将字符地址传给rsi

        : "rax", "rdi", "rsi", "rdx" // 被更改的寄存器

    );

    asm volatile (

        "mov $1, %%rdi\n"        // 文件描述符1代表标准输出

        "mov %0, %%rsi\n"        // rsi寄存器保存字符的地址

        "mov $1, %%rdx\n"        // 写入1个字符

        "mov $1, %%rax\n"        // 系统调用号1代表write

        "syscall\n"              // 执行系统调用

        :                        // 输出部分为空

        : "r" (&c)               // 输入部分，将字符地址传给rsi

        : "rax", "rdi", "rsi", "rdx" // 被更改的寄存器

    );