

## Homework 4 — September 30

Lecturer: Hu Weiwu

Completed by: Zhang Jiawei

## 4.1

在 Linux/LoongArch64 ABI 的函数调用约定中, (char)0x61 以整型调用, 参数使用寄存器 \$a0-\$a7 传递; (short)0xffff 以整型调用, 参数使用寄存器 \$a0-\$a7 传递; 1 以整型调用, 参数使用寄存器 \$a0-\$a7 传递; 2 以整型调用, 参数使用寄存器 \$a0-\$a7 传递; 3.0 以浮点型调用, 参数使用寄存器 \$fa0-\$fa7 传递; 4.0 以浮点型调用, 参数使用寄存器 \$fa0-\$fa7 传递; sm 以整型调用, 参数使用寄存器 \$a0-\$a7 传递; bg 超过 2XLEN 位, 以整型调用, 参数使用引用传递; 9 以整型调用, 参数使用寄存器 \$a0-\$a7 传递。

## 4.2

(1) 举例如下:

```

1  # 线程1和线程2均运行以下代码
2  _start:
3      la    $t0, counter    # 加载 counter 的地址到 t0
4      lw    $t1, 0($t0)     # 从 counter 加载当前值到 t1
5      addi  $t1, $t1, 1     # t1 = t1 + 1
6      sw    $t1, 0($t0)     # 将 t1 的值存回 counter

```

显然, 两个线程可能同时读取 counter 的值, 然后同时对其进行加一操作, 导致最终的结果不正确。

(2) 使用 LL/SC 指令改写程序片段:

```

1  # 线程1和线程2均运行以下代码
2  _start:
3      la    $t0, counter    # 加载 counter 的地址到 t0
4  atomic_add:
5      ll.w  $t1, 0($t0)     # 加载链接: 从 counter 加载当前值到 t1
6      addi  $t2, $t1, 1     # t2 = t1 + 1
7      sc.w  $t2, $t2, 0($t0) # 存储条件: 尝试将 t2 存回 counter
8      beqz  $t2, atomic_add # 如果存储失败, 重试

```

使用 LL/SC 指令实现自旋锁, 可以保证对 counter 的操作是原子的, 从而避免了多线程并发问题。

## 4.3

安装 LoongArch 交叉编译器后, 先使用 loongarch64-unknown-linux-gnu-gcc 编译 C 程序, 再使用 loongarch64-unknown-linux-gnu-objdump -d 查看反汇编代码。我所写 C 程序如下:

```

1  #include <stdio.h>
2

```

```

3  int bubble_sort(int *arr, int n) {
4      int i, j, temp;
5      for (i = 0; i < n - 1; i++) {
6          for (j = 0; j < n - i - 1; j++) {
7              if (arr[j] > arr[j + 1]) {
8                  temp = arr[j];
9                  arr[j] = arr[j + 1];
10                 arr[j + 1] = temp;
11             }
12         }
13     }
14     return 0;
15 }

16
17 int main() {
18     int arr[] = {64, 34, 25, 12, 22, 11, 90};
19     int n = sizeof(arr) / sizeof(arr[0]);
20     bubble_sort(arr, n);
21     printf("Sorted array: \n");
22     for (int i = 0; i < n; i++) {
23         printf("%d ", arr[i]);
24     }
25     printf("\n");
26     return 0;
27 }

```

使用-O0 优化(不优化)反汇编代码中冒泡排序函数部分如下:

```

1  0000000000000083c <bubble_sort>:
2  83c: 02ff4063  addi.d $sp, $sp, -48
3  840: 29c0a076  st.d   $fp, $sp, 40
4  844: 02c0c076  addi.d $fp, $sp, 48
5  848: 29ff62c4  st.d   $a0, $fp, -40
6  84c: 001500ac  move   $t0, $a1
7  850: 29bf52cc  st.w   $t0, $fp, -44
8  854: 29bfb2c0  st.w   $zero, $fp, -20
9  858: 5000dc00  b      220 # 934 <bubble_sort+0xf8>
10 85c: 29bfa2c0  st.w   $zero, $fp, -24
11 860: 5000a400  b      164 # 904 <bubble_sort+0xc8>
12 864: 24ffeacc  ldptr.w $t0, $fp, -24
13 868: 0041098c  slli.d $t0, $t0, 0x2
14 86c: 28ff62cd  ld.d   $t1, $fp, -40
15 870: 0010b1ac  add.d  $t0, $t1, $t0
16 874: 2400018e  ldptr.w $t2, $t0, 0

```

```
17 878: 24ffeacc ldptr.w $t0, $fp, -24
18 87c: 02c0058c addi.d $t0, $t0, 1
19 880: 0041098c slli.d $t0, $t0, 0x2
20 884: 28ff62cd ld.d $t1, $fp, -40
21 888: 0010b1ac add.d $t0, $t1, $t0
22 88c: 2400018c ldptr.w $t0, $t0, 0
23 890: 001501cd move $t1, $t2
24 894: 6400658d bge $t0, $t1, 100 # 8f8 <bubble_sort+0xbc>
25 898: 24ffeacc ldptr.w $t0, $fp, -24
26 89c: 0041098c slli.d $t0, $t0, 0x2
27 8a0: 28ff62cd ld.d $t1, $fp, -40
28 8a4: 0010b1ac add.d $t0, $t1, $t0
29 8a8: 2400018c ldptr.w $t0, $t0, 0
30 8ac: 29bf92cc st.w $t0, $fp, -28
31 8b0: 24ffeacc ldptr.w $t0, $fp, -24
32 8b4: 02c0058c addi.d $t0, $t0, 1
33 8b8: 0041098c slli.d $t0, $t0, 0x2
34 8bc: 28ff62cd ld.d $t1, $fp, -40
35 8c0: 0010b1ad add.d $t1, $t1, $t0
36 8c4: 24ffeacc ldptr.w $t0, $fp, -24
37 8c8: 0041098c slli.d $t0, $t0, 0x2
38 8cc: 28ff62ce ld.d $t2, $fp, -40
39 8d0: 0010b1cc add.d $t0, $t2, $t0
40 8d4: 240001ad ldptr.w $t1, $t1, 0
41 8d8: 2500018d stptr.w $t1, $t0, 0
42 8dc: 24ffeacc ldptr.w $t0, $fp, -24
43 8e0: 02c0058c addi.d $t0, $t0, 1
44 8e4: 0041098c slli.d $t0, $t0, 0x2
45 8e8: 28ff62cd ld.d $t1, $fp, -40
46 8ec: 0010b1ac add.d $t0, $t1, $t0
47 8f0: 28bf92cd ld.w $t1, $fp, -28
48 8f4: 2500018d stptr.w $t1, $t0, 0
49 8f8: 28bfa2cc ld.w $t0, $fp, -24
50 8fc: 0280058c addi.w $t0, $t0, 1
51 900: 29bfa2cc st.w $t0, $fp, -24
52 904: 28bf52cd ld.w $t1, $fp, -44
53 908: 28bfb2cc ld.w $t0, $fp, -20
54 90c: 001131ac sub.w $t0, $t1, $t0
55 910: 0040818c slli.w $t0, $t0, 0x0
56 914: 02bff8d8c addi.w $t0, $t0, -1
57 918: 0040818d slli.w $t1, $t0, 0x0
58 91c: 28bfa2cc ld.w $t0, $fp, -24
59 920: 0040818c slli.w $t0, $t0, 0x0
```

```

60 924: 63ff418d blt      $t0, $t1, -192 # 864 <bubble_sort+0x28>
61 928: 28bfb2cc ld.w    $t0, $fp, -20
62 92c: 0280058c addi.w $t0, $t0, 1
63 930: 29bfb2cc st.w    $t0, $fp, -20
64 934: 28bf52cc ld.w    $t0, $fp, -44
65 938: 02bffd8c addi.w $t0, $t0, -1
66 93c: 0040818d slli.w $t1, $t0, 0x0
67 940: 28bfb2cc ld.w    $t0, $fp, -20
68 944: 0040818c slli.w $t0, $t0, 0x0
69 948: 63ff158d blt      $t0, $t1, -236 # 85c <bubble_sort+0x20>
70 94c: 0015000c move     $t0, $zero
71 950: 00150184 move     $a0, $t0
72 954: 28c0a076 ld.d    $fp, $sp, 40
73 958: 02c0c063 addi.d $sp, $sp, 48
74 95c: 4c000020 ret

```

使用-O1 优化反汇编代码中冒泡排序函数部分如下:

```

1 000000000000083c <bubble_sort>:
2 83c: 0280040c li.w    $t0, 1
3 840: 64006585 bge     $t0, $a1, 100 # 8a4 <bubble_sort+0x68>
4 844: 004080b0 slli.w $t4, $a1, 0x0
5 848: 02800411 li.w    $t5, 1
6 84c: 02c01092 addi.d $t6, $a0, 4
7 850: 50002c00 b       44 # 87c <bubble_sort+0x40>
8 854: 02c0118c addi.d $t0, $t0, 4
9 858: 58001d8f beq     $t0, $t3, 28 # 874 <bubble_sort+0x38>
10 85c: 2400018d ldptr.w $t1, $t0, 0
11 860: 2400058e ldptr.w $t2, $t0, 4
12 864: 67fff1cd bge     $t2, $t1, -16 # 854 <bubble_sort+0x18>
13 868: 2500018e stptr.w $t2, $t0, 0
14 86c: 2980118d st.w    $t1, $t0, 4
15 870: 53ffe7ff b       -28 # 854 <bubble_sort+0x18>
16 874: 02bffe10 addi.w $t4, $t4, -1
17 878: 58002e11 beq     $t4, $t5, 44 # 8a4 <bubble_sort+0x68>
18 87c: 0040820c slli.w $t0, $t4, 0x0
19 880: 64001e2c bge     $t5, $t0, 28 # 89c <bubble_sort+0x60>
20 884: 0015008c move     $t0, $a0
21 888: 02bffa0f addi.w $t3, $t4, -2
22 88c: 00df01ef bstrpick.d $t3, $t3, 0x1f, 0x0
23 890: 002c81ef alsld $t3, $t3, $zero, 0x2
24 894: 0010c9ef add.d   $t3, $t3, $t6
25 898: 53ffc7ff b       -60 # 85c <bubble_sort+0x20>
26 89c: 02bffe10 addi.w $t4, $t4, -1

```

```

27 8a0: 53ffdfbf b      -36 # 87c <bubble_sort+0x40>
28 8a4: 00150004 move    $a0, $zero
29 8a8: 4c000020 ret

```

使用-O2 优化反汇编代码中冒泡排序函数部分如下:

```

1 00000000000008dc <bubble_sort>:
2 8dc: 0280040c li.w    $t0, 1
3 8e0: 64004d85 bge     $t0, $a1, 76 # 92c <bubble_sort+0x50>
4 8e4: 004080b0 slli.w  $t4, $a1, 0x0
5 8e8: 02c01092 addi.d  $t6, $a0, 4
6 8ec: 02800411 li.w    $t5, 1
7 8f0: 64004630 bge     $t5, $t4, 68 # 934 <bubble_sort+0x58>
8 8f4: 02bffa0f addi.w  $t3, $t4, -2
9 8f8: 00df01ef bstrpick.d $t3, $t3, 0x1f, 0x0
10 8fc: 002c81ef alsld  $t3, $t3, $zero, 0x2
11 900: 0015008c move    $t0, $a0
12 904: 0010c9ef add.d   $t3, $t3, $t6
13 908: 2400018d ldptr.w $t1, $t0, 0
14 90c: 2400058e ldptr.w $t2, $t0, 4
15 910: 64000dcd bge     $t2, $t1, 12 # 91c <bubble_sort+0x40>
16 914: 2500018e stptr.w $t2, $t0, 0
17 918: 2980118d st.w    $t1, $t0, 4
18 91c: 02c0118c addi.d  $t0, $t0, 4
19 920: 5fffe98f bne     $t0, $t3, -24 # 908 <bubble_sort+0x2c>
20 924: 02bffe10 addi.w  $t4, $t4, -1
21 928: 5fffc111 bne     $t4, $t5, -56 # 8f0 <bubble_sort+0x14>
22 92c: 00150004 move    $a0, $zero
23 930: 4c000020 ret
24 934: 02bffe10 addi.w  $t4, $t4, -1
25 938: 53ffbbff b      -72 # 8f0 <bubble_sort+0x14>

```

使用-O3 优化反汇编代码中冒泡排序函数部分与-O2 优化反汇编代码相同,故不再重复给出。

不同的优化级别对编译生成的汇编代码有显著影响:

-O0:无优化,代码较长,执行效率低,适用于调试。

-O1:基本优化,代码长度减少,执行效率提升,适用于需要一定优化但仍需调试的场景。

-O2:较高优化,代码显著减少,执行效率显著提升,适用于大多数应用程序。

-O3: 最高级别优化,进一步优化代码长度和执行效率,适用于对性能要求极高的应用程序。(在本题中看不出来效果)

#### 4.4

所写 C 程序如下:

```

1 #include <stdio.h>

```

```
2
3 struct exp
4 {
5     char a;
6     short b;
7     int c;
8     long d;
9     float e;
10    double f;
11    long double g;
12 };
13
14 int main()
15 {
16     struct exp s;
17     printf("Size of struct: %lu\n", sizeof(s));
18     printf("Size of char: %lu\n", sizeof(s.a));
19     printf("Size of short: %lu\n", sizeof(s.b));
20     printf("Size of int: %lu\n", sizeof(s.c));
21     printf("Size of long: %lu\n", sizeof(s.d));
22     printf("Size of float: %lu\n", sizeof(s.e));
23     printf("Size of double: %lu\n", sizeof(s.f));
24     printf("Size of long double: %lu\n", sizeof(s.g));
25     return 0;
26 }
```

输出结果如下：

```
1 Size of struct: 48
2 Size of char: 1
3 Size of short: 2
4 Size of int: 4
5 Size of long: 8
6 Size of float: 4
7 Size of double: 8
8 Size of long double: 16
```

显然有  $48 > 1 + 2 + 4 + 8 + 4 + 8 + 16$ ，这是由于结构体内部的成员在内存中是按照对齐方式排列的，char 类型的成员 a 占用 1 字节，short 类型的成员 b 占用 2 字节，从第 2 字节开始，int 类型的成员 c 占用 4 字节，从第 4 字节开始，long 类型的成员 d 占用 8 字节，从第 8 字节开始，float 类型的成员 e 占用 4 字节，从第 12 字节开始，double 类型的成员 f 占用 8 字节，从第 16 字节开始，long double 类型的成员 g 占用 16 字节，从第 32 字节开始，所以结构体的大小为 48 字节。

更改结构体中的成员顺序：

```
1 struct exp
2 {
3     char a;
4     long double g;
5     short b;
6     int c;
7     long d;
8     float e;
9     double f;
10 };
```

输出结果如下：

```
1 Size of struct: 64
2 Size of char: 1
3 Size of short: 2
4 Size of int: 4
5 Size of long: 8
6 Size of float: 4
7 Size of double: 8
8 Size of long double: 16
```

显然有  $64 > 1 + 2 + 4 + 8 + 4 + 8 + 16$ ，这也是由于结构体内部的成员在内存中是按照对齐方式排列的，char 类型的成员 a 占用 1 字节，long double 类型的成员 g 占用 16 字节，从第 16 字节开始，short 类型的成员 b 占用 2 字节，从第 32 字节开始，int 类型的成员 c 占用 4 字节，从第 36 字节开始，long 类型的成员 d 占用 8 字节，从第 40 字节开始，float 类型的成员 e 占用 4 字节，从第 48 字节开始，double 类型的成员 f 占用 8 字节，从第 56 字节开始，所以结构体的大小为 64 字节。