Computer Architecture: Hardware-Software Codesign **Fall 2024**

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)0xfff` 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：

```
# 线程1和线程2均运行以下代码
_start:
  la    $t0, counter  # 加载 counter 的地址到 t0
  lw    $t1, 0($t0)    # 从 counter 加载当前值到 t1
  addi  $t1, $t1, 1    # t1 = t1 + 1
  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：

```
# 线程1和线程2均运行以下代码
_start:
  la    $t0, counter  # 加载 counter 的地址到 t0
atomic_add:
  ll.w$t1, 0($t0)    # 加载链接：从 counter 加载当前值到 t1
  addi  $t2, $t1, 1    # t2 = t1 + 1
  sc.w$t2, $t2, 0(t0) # 存储条件：尝试将 t2 存回 counter
  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：

```
#include <stdio.h>

```

```c
int bubble_sort(int *arr, int n) {
   int i, j, temp;
   for (i = 0; i < n - 1; i++) {
      for (j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
   return 0;
}

int main() {
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);
   bubble_sort(arr, n);
   printf("Sorted array: \n");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
   return 0;
}
```

使用-O0 优化(不优化)反汇编代码中冒泡排序函数部分如下:

```
000000000000083c <bubble_sort>:
83c: 02ff4063  addi.d $sp, $sp, -48
840: 29c0a076  st.d   $fp, $sp, 40
844: 02c0c076  addi.d $fp, $sp, 48
848: 29ff62c4  st.d   $a0, $fp, -40
84c: 001500ac  move     $t0, $a1
850: 29bf52cc  st.w   $t0, $fp, -44
854: 29bfb2c0  st.w   $zero, $fp, -20
858: 5000dc00  b        220  # 934 <bubble_sort+0xf8>
85c: 29bfa2c0  st.w   $zero, $fp, -24
860: 5000a400  b        164  # 904 <bubble_sort+0xc8>
864: 24ffeacc  ldptr.w $t0, $fp, -24
868: 0041098c  slli.d $t0, $t0, 0x2
86c: 28ff62cd  ld.d   $t1, $fp, -40
870: 0010b1ac  add.d $t0, $t1, $t0
874: 2400018e  ldptr.w $t2, $t0, 0
```

```
  878: 24ffeacc  ldptr.w $t0, $fp, -24
  87c: 02c0058c  addi.d $t0, $t0, 1
  880: 0041098c  slli.d $t0, $t0, 0x2
  884: 28ff62cd  ld.d   $t1, $fp, -40
  888: 0010b1ac  add.d $t0, $t1, $t0
  88c: 2400018c  ldptr.w $t0, $t0, 0
  890: 001501cd  move     $t1, $t2
  894: 6400658d  bge      $t0, $t1, 100 # 8f8 <bubble_sort+0xbc>
  898: 24ffeacc  ldptr.w $t0, $fp, -24
  89c: 0041098c  slli.d $t0, $t0, 0x2
  8a0: 28ff62cd  ld.d   $t1, $fp, -40
  8a4: 0010b1ac  add.d $t0, $t1, $t0
  8a8: 2400018c  ldptr.w $t0, $t0, 0
  8ac: 29bf92cc  st.w   $t0, $fp, -28
  8b0: 24ffeacc  ldptr.w $t0, $fp, -24
  8b4: 02c0058c  addi.d $t0, $t0, 1
  8b8: 0041098c  slli.d $t0, $t0, 0x2
  8bc: 28ff62cd  ld.d   $t1, $fp, -40
  8c0: 0010b1ad  add.d $t1, $t1, $t0
  8c4: 24ffeacc  ldptr.w $t0, $fp, -24
  8c8: 0041098c  slli.d $t0, $t0, 0x2
  8cc: 28ff62ce  ld.d   $t2, $fp, -40
  8d0: 0010b1cc  add.d $t0, $t2, $t0
  8d4: 240001ad  ldptr.w $t1, $t1, 0
  8d8: 2500018d  stptr.w $t1, $t0, 0
  8dc: 24ffeacc  ldptr.w $t0, $fp, -24
  8e0: 02c0058c  addi.d $t0, $t0, 1
  8e4: 0041098c  slli.d $t0, $t0, 0x2
  8e8: 28ff62cd  ld.d   $t1, $fp, -40
  8ec: 0010b1ac  add.d $t0, $t1, $t0
  8f0: 28bf92cd  ld.w   $t1, $fp, -28
  8f4: 2500018d  stptr.w $t1, $t0, 0
  8f8: 28bfa2cc  ld.w   $t0, $fp, -24
  8fc: 0280058c  addi.w $t0, $t0, 1
  900: 29bfa2cc  st.w   $t0, $fp, -24
  904: 28bf52cd  ld.w   $t1, $fp, -44
  908: 28bfb2cc  ld.w   $t0, $fp, -20
  90c: 001131ac  sub.w $t0, $t1, $t0
  910: 0040818c  slli.w $t0, $t0, 0x0
  914: 02bffd8c  addi.w $t0, $t0, -1
  918: 0040818d  slli.w $t1, $t0, 0x0
  91c: 28bfa2cc  ld.w   $t0, $fp, -24
  920: 0040818c  slli.w $t0, $t0, 0x0
```

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

使用-O1 优化反汇编代码中冒泡排序函数部分如下：

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

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

使用-O2 优化反汇编代码中冒泡排序函数部分如下：

```
00000000000008dc <bubble_sort>:
  8dc: 0280040c  li.w   $t0, 1
  8e0: 64004d85  bge      $t0, $a1, 76 # 92c <bubble_sort+0x50>
  8e4: 004080b0  slli.w $t4, $a1, 0x0
  8e8: 02c01092  addi.d $t6, $a0, 4
  8ec: 02800411  li.w   $t5, 1
  8f0: 64004630  bge      $t5, $t4, 68 # 934 <bubble_sort+0x58>
  8f4: 02bffa0f  addi.w $t3, $t4, -2
  8f8: 00df01ef  bstrpick.d $t3, $t3, 0x1f, 0x0
  8fc: 002c81ef  alsl.d $t3, $t3, $zero, 0x2
  900: 0015008c  move     $t0, $a0
  904: 0010c9ef  add.d $t3, $t3, $t6
  908: 2400018d  ldptr.w $t1, $t0, 0
  90c: 2400058e  ldptr.w $t2, $t0, 4
  910: 64000dcd  bge      $t2, $t1, 12 # 91c <bubble_sort+0x40>
  914: 2500018e  stptr.w $t2, $t0, 0
  918: 2980118d  st.w   $t1, $t0, 4
  91c: 02c0118c  addi.d $t0, $t0, 4
  920: 5fffe98f  bne      $t0, $t3, -24 # 908 <bubble_sort+0x2c>
  924: 02bffe10  addi.w $t4, $t4, -1
  928: 5fffca11  bne      $t4, $t5, -56 # 8f0 <bubble_sort+0x14>
  92c: 00150004  move     $a0, $zero
  930: 4c000020  ret
  934: 02bffe10  addi.w $t4, $t4, -1
  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。(本题无效果)

4.4

所写 C 程序如下：

```
#include <stdio.h>
struct exp{
```

```
    char a;
    short b;
    int c;
    long d;
    float e;
    double f;
    long double g;
};

int main(){
    struct exp s;
    printf("Size of struct: %lu\n", sizeof(s));
    printf("Size of char: %lu\n", sizeof(s.a));
    printf("Size of short: %lu\n", sizeof(s.b));
    printf("Size of int: %lu\n", sizeof(s.c));
    printf("Size of long: %lu\n", sizeof(s.d));
    printf("Size of float: %lu\n", sizeof(s.e));
    printf("Size of double: %lu\n", sizeof(s.f));
    printf("Size of long double: %lu\n", sizeof(s.g));
    return 0;
}
```

输出结果如下：

```
Size of struct: 48
Size of char: 1
Size of short: 2
Size of int: 4
Size of long: 8
Size of float: 4
Size of double: 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：

```
struct exp
{
    char a;
```

```
    long double g;
    short b;
    int c;
    long d;
    float e;
    double f;
};
```

输出结果如下：

```
Size of struct: 64
Size of char: 1
Size of short: 2
Size of int: 4
Size of long: 8
Size of float: 4
Size of double: 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。

4.5

我使用 x86 汇编，所写程序如下：

```
section .bss
buffer resb 1 ; 用于存储输入的字符

section .text
global _start

_start:
    ; 系统调用号 (sys_read)
    mov rax, 0      ; 系统调用号 (sys_read)
    mov rdi, 0      ; 文件描述符 (stdin)
    mov rsi, buffer ; 缓冲区地址
    mov rdx, 1      ; 读取的字节数
    syscall         ; 调用内核

    ; 系统调用号 (sys_write)
    mov rax, 1      ; 系统调用号 (sys_write)
```

```
17      mov rdi, 1      ; 文件描述符 (stdout)
18      mov rsi, buffer ; 缓冲区地址
19      mov rdx, 1      ; 写入的字节数
20      syscall         ; 调用内核
21
22      ; 系统调用号 (sys_exit)
23      mov rax, 60     ; 系统调用号 (sys_exit)
24      xor rdi, rdi    ; 退出状态码
25      syscall         ; 调用内核
```

通过 gdb 调试，观察 `sys_read` 系统调用前后的寄存器状态：

```
(gdb) stepi
13          syscall             ; 调用内核
(gdb) info registers
rax            0x0                 0
rbx            0x0                 0
rcx            0x0                 0
rdx            0x1                 1
rsi            0x402000            4202496
rdi            0x0                 0
rbp            0x0                 0x0
rsp            0x7fffffffde30      0x7fffffffde30
```

图 **4.1.** sys_read 系统调用前寄存器状态

```
(gdb) info register
rax            0x1                 1
rbx            0x0                 0
rcx            0x40101b            4198427
rdx            0x1                 1
rsi            0x402000            4202496
rdi            0x0                 0
rbp            0x0                 0x0
rsp            0x7fffffffde30      0x7fffffffde30
```

图 **4.2.** sys_read 系统调用后寄存器状态

对照 x86 的 ABI，系统调用号存储在 `rax` 寄存器中，文件描述符存储在 `rdi` 寄存器中，缓冲区地址存储在 `rsi` 寄存器中，读取的字节数存储在 `rdx` 寄存器中，且使用 syscall 指令来触发系统调用。观察 gdb 调试结果，发现 `rcx` 寄存器的值从 0 变为 0x40101b，这是因为 `rcx` 寄存器在 `sys_read` 系统调用之后被修改为当前指令的 pc 值，以回到用户态继续执行。