2 示例程序

2.1 C 与汇编的混合编程实验原理

C 语言调用汇编有两个关键点——调用与传参。对于调用，我们需要在汇编程序中通过.global 定义一个全局函数，然后该函数就可以在 C 代码中通过 extend 关键字加以声明，使其能够在 C 代码中直接调用。

关于 C 与汇编的混合编程的参数传递，ARM64 提供了 31 个 64 位通用寄存器（X0-X30），对应低 32 位为 W0-W30，而 X31 为堆栈指针寄存器。各自的用途详见表 2-1。参数传递用到的是 x0~x7 这 8 个寄存器，若参数个数大于 8 个则需要使用堆栈来传递参数。

表2-1　ARM64 常用寄存器用途

寄存器	用途
x0~x7	传递参数和返回值，多余的参数用堆栈传递，64位的返回结果保存在x0中。
X8	用于保存子程序的返回地址。
x9~x15	临时寄存器，也叫可变寄存器，无需保存。
x16~x17	子程序内部调用寄存器，使用时不需要保存，尽量不要使用。
x18	平台寄存器，它的使用与平台相关，尽量不要使用。
x19~x28	临时寄存器，子程序使用时必须保存。
x29	帧指针寄存器（FP），用于连接栈帧，使用时必须保存。
x30	链接寄存器（LR），用于保存子程序的返回地址。
x31	堆栈指针寄存器（SP），用于指向每个函数的栈顶。

2.1.1 C 语言调用汇编实现累加和求值

本示例实现的功能是：输入一个正整数，输出从 0 到该正整数的所有正整数的累加和，输入输出功能在 C 代码中实现，计算功能通过调用汇编函数实现。需要传入的参数是输入的正整数，汇编传出的参数为累加和，因此只用到一个 x0 寄存器即可实现参数传递功能。

步骤 1 创建 sum.c 文件

执行命令 vi sum.c 编写 C 程序，按“A”进入编辑模式后输入代码。

```
vim sum.c
```

编写内容如下：

```
#include <stdio.h>
extern int add(int num);    //声明外部调用，函数名为 add。
int main()
{
        int i,sum;
        scanf("%d",&i);  //输入初始正整数。
        sum=add(i);  //调用汇编函数 add，返回值赋值给 sum。
        printf("sum=%d\n",sum);  //将累加和输出。
        return 0;
}
```

编写完成后按“ESC”键进入命令行模式，输入“:wq”后回车保存并退出编辑。

步骤 2 创建 add.s 文件

执行 vi add.s 编写所调用的汇编代码，内容如下：

```
.global add   //定义全局函数，函数名为 add。
add:                    //lable:add
        ADD x1,x1,x0   //将 x0+x1 的值存入 x1 寄存器。
        SUB x0,x0,#1   //将 x0-1 的值存入 x0 寄存器。
        CMP x0,#0      //比较 x0 和 0 的大小。
        BNE add        //若 x0 与 0 不相等，跳转到 add；x0=0 则继续执行。
        MOV x0,x1      //将 x1 的值存入 x0（需要 x0 来返回值）。
    RET
```

步骤 3 使用 gcc 编译生成可执行文件

GCC 是由 GNU 开发的编程语言译器。

GCC 最基本的语法是：gcc [filenames] [options]

其中[options]就是编译器所需要的参数，[filenames]给出相关的文件名称。

-c：只编译，不链接成为可执行文件，编译器只是由输入的.c 等源代码文件生成.o 为后缀的目标文件，通常用于编译不包含主程序的子程序文件。

-o output_filename：确定输出文件的名称为 output_filename，同时这个名称不能和源文件同名。如果不给出这个选项，gcc 就给出预设的可执行文件 a.out。

-g：产生符号调试工具（GNU 的 gdb）所必要的符号资讯，要想对源代码进行调试，我们就必须加入这个选项。

执行 gcc sum.c add.s -o sum 进行编译，然后执行./sum 运行，输入 100，返回累加和 5050，如图 2-3 所示：

```
gcc sum.c add.s -o sum
./sum
```

```
[root@ecs-01 sum]# gcc sum.c add.s -o sum
[root@ecs-01 sum]# ./sum
100
sum=5050
[root@ecs-01 sum]#
```

图2-1 编译运行

编译成功，程序执行结果正确。

2.1.2 C 语言内嵌汇编

C 语言是无法完全代替汇编语言的，一方面是其效率比 C 要高，另一方面是某些特殊的指令在 C 语法中是没有等价的语法的。例如：操作某些特殊的 CPU 寄存器如状态寄存器、操作主板上的某些 I/O 端口或者对性能要求极其苛刻的场景等，我们都可以通过在 C 语言中内嵌汇编代码来满足要求。

在 C 语言代码中内嵌汇编语句的基本格式为：

```
__asm__ __volatile__ (“asm code”
: 输出操作数列表
: 输入操作数列表
: clobber 列表
);
```

说明：

1. __asm__前后各两个下划线，并且两个下划线之间没有空格，用于声明这行代码是一个内嵌汇编表达式，是内嵌汇编代码时必不可少的关键字。
2. 关键字 volatile 前后各两个下划线，并且两个下划线之间没有空格。该关键字告诉编译器不要优化内嵌的汇编语句，如果想优化可以不加 volatile； 在很多时候，如果不使用该关键字的话，汇编语句有可能被编译器修改而无法达到预期的执行效果。
3. 括号里面包含四个部分：汇编代码（asm code）、输出操作数列表（output）、输入操作数列表（input）和 clobber 列表（破坏描述符）。这四个部分之间用“:”隔开。其中，输入操作数列表部分和 clobber 列表部分是可选的，如果不使用 clobber 列表部分，则格式可以简化为：

```
__asm__ __volatile__ (“asm code”:output：input);
```

如果不使用输入部分，则格式可以简化为：

```
__asm__ __volatile__ (“asm code”:output::changed);
```

此时，即使输入部分为空，输出部分之后的“：”也是不能省略的。另外，输入部分和 clobber 列表部分是可选的，如果都为空，则格式可以简化为：

__asm__ __volatile__ (“asm code”:output);

4. 括号之后要以“;”结尾。

以下示例程序实现的是计算累加和功能，与 2.4.1 中示例程序的功能相同，用到了 C 语言内嵌汇编的方法，汇编指令部分与 2.4.1 相同。

步骤 1 创建 builtin.c 文件

执行命令 vi builtin.c 编写 c 程序。

编写内容如下：

```
#include <stdio.h>
int main()
{
int val;
scanf("%d",&val);
__asm__ __volatile__(
                                "add:\n"
                                "ADD x1,x1,x0\n"
                                "SUB x0,x0,#1\n"
                                "CMP x0,#0\n"
                                "BNE add\n"
                                "MOV x0,x1\n"
                                :"=r"(val)
//r 代表存放在某个通用寄存器中，即在汇编代码里用一个寄存器代替()部分
//中定义的 c 变量即 val；=代表只写，即在汇编代码里只能改变 C 变量的
//值，而不能取它的值。
                                :"0"(val)
        //0 代表与第一个输出参数共用同一个寄存器。
                                :
                            );
printf("sum is %d \n",val);
return 0;
    }
```

输入完成后保存并退出。

步骤 2 编译并运行可执行文件

输入命令 gcc builtin.c -o builtin 进行编译，编译成功后输入./builtin 执行程序，输入 100，返回累加和 5050。

```
gcc builtin.c -o builtin
./builtin
```

如图 2-8 所示：

```
[root@ecs-01 builtin]# gcc builtin.c -o builtin
[root@ecs-01 builtin]# ./builtin
100
sum is 5050
[root@ecs-01 builtin]#
```

图2-2 执行 builtin

编译成功，程序执行结果正确。

2.2 内存拷贝及优化实验原理

优化效果通过计算对应代码段的执行时间来判断。具体方案是通过 C 语言调用汇编，在 C 代码中计算时间，在汇编代码中设计不同的方案，对比每种方案的执行时间，判断优化效果。本示例的优化针对内存读写，示例程序功能是内存拷贝，拷贝功能在汇编函数中实现。

说明：在使用 ldrb/ldp 和 str/stp 等访存指令时，要注意区分这三种形式：

1. 前索引方式，形如：ldrb w2,[X1,#1] //将 x1+1 指向的地址处的一个字节放入 w2 中，x1 寄存器的值保持不变。
2. 自动索引方式，形如：ldrb w2,[X1,#1]! //将 x1+1 指向的地址处的一个字节放入 w2 中，然后 x1+1→x1。
3. 后索引方式，形如 ldrb w2,[X1],#1 //将 x1 指向的地址处的一个字节放入 w2 中，然后 x1+1→x1。

2.2.1 基础代码

本程序由两部分组成：

第一部分是主函数，采用 Linux C 语言编码，用来测试内存拷贝函数的执行时间；

第二部分是内存拷贝函数，采用 GNU ARM64 汇编语言编码。为了较为准确的测量内存拷贝函数 memorycopy()的执行时间，调用了 clock_gettime()来分别记录 memorycopy()执行前和执行后的系统时间，以纳秒为计时单位。

步骤 1 创建 time.c 文件

执行 vi time.c 编写 c 语言计时程序。

通过 clock_gettime 函数来计算时间差，从而得出所求代码的执行时间，代码内容如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define len 60000000   //内存拷贝长度为 60000000
char src[len],dst[len];   //源地址与目的地址
long int len1=len;
```

```
extern void memorycopy(char *dst,char *src,long int len1);  //声明外部函数
int main()
{
        struct timespec t1,t2;  //定义初始与结束时间
        int i,j;
            //为初始地址段赋值，以便后续从该地址段读取数据拷贝
         for(i=0;i<len-1;i++)
        {
            src[i]='a';
        }
        src[i]=0;
        clock_gettime(CLOCK_MONOTONIC,&t1);    //计算开始时间。
        memorycopy(dst,src,len1);  //汇编调用，执行相应代码段。
        clock_gettime(CLOCK_MONOTONIC,&t2);  //计算结束时间。
        //得出目标代码段的执行时间。
        printf("memorycopy time is %11u ns\n",t2.tv_nsec-t1.tv_nsec);
        return 0;
   }
```

内存拷贝函数 memorycopy()的功能是实现将尺寸为 len（这里设置为 60000000）的 src 字符数组的内容拷贝到同样尺寸的 dst 字符数组中。memorycopy()函数用 AArch64 汇编代码实现。

在本例中，需要传递的参数有三个：

第一个参数是目标字符串的首地址，用寄存器 x0 来传递；

第二个参数是源字符串的首地址，用寄存器 x1 来传递；

第三个参数是传输的字节数目，用寄存器 x2 来传递。

步骤 2 创建 copy.s 文件

执行 vi copy.s 编写优化前的原始汇编代码。

代码如下：

```
.global memorycopy   //声明全局函数
memorycopy:
            ldrb w3,[x1],#1   //从源字符串地址中读取
            str w3,[x0],#1   //向目的字符串地址中写
            sub x2,x2,#1
            cmp x2,#0   //判断是否结束
bne memorycopy
ret
```

步骤 3 编译并运行可执行文件

执行 gcc time.c copy.s -o m1 完成编译，并执行。

```
gcc time.c copy.s -o m1
./m1
```

结果如下，可以看到 memorycopy 函数具体的执行时间为 47514738ns，如图 3-3 所示：

```
[root@ecs-01 memory]# gcc time.c copy.s -o m1
[root@ecs-01 memory]# ./m1
memorycopy time is    47514738 ns
```

图2-3 原始程序执行时间

接下来我们基于原始代码进行修改，观察他们的执行时间进行对比。

2.2.2 循环展开优化

循环展开是最常见的代码优化思路，通过减少指令总数来实现代码优化。

步骤 1 创建 2 倍展开优化 copy121.s 文件

先将 copy.s 展开两倍,命名为 copy121.s，代码如下：

```
.global memorycopy
memorycopy:
sub x1,x1,#1   //传进去的地址首地址为 0，减 1 是移动到 0 前面的地址-1
sub x0,x0,#1
lp:
ldrb w3,[x1,#1]!   //将地址+1 后就移动到了首地址 0
ldrb w4,[x1,#1]!   //一次循环读两个字节
str w3,[x0,#1]!
str w4,[x0,#1]!   //一次循环写两个字节
sub x2,x2,#2
cmp x2,#0
bne lp
ret
```

步骤 2 编译并运行执行文件

执行命令 gcc time.c copy121.s -o m121 编译并运行程序。

```
gcc time.c copy121.s -o m121
./m121
```

在进行了循环展开后，这次 memorycopy 函数的执行时间变为了 36851856 ns，如图 3-5 所示：

```
[root@Malluma memory]# gcc time.c copy121.s -o m121
[root@Malluma memory]# ./m121
memorycopy time is    36851856 ns
```

图2-4 编译执行程序 m121

步骤 3 创建 4 倍展开优化 copy122.s 文件

然后输入以下代码，并将其编译，得到 m122，具体代码如下：

```
.global memorycopy
memorycopy:
        sub x1,x1,#1
```

```
        sub x0,x0,#1
lp:
        ldrb w3,[x1,#1]!
        ldrb w4,[x1,#1]!
        ldrb w5,[x1,#1]!
        ldrb w6,[x1,#1]!
        str w3,[x0,#1]!
        str w4,[x0,#1]!
        str w5,[x0,#1]!
        str w6,[x0,#1]!
        sub x2,x2,#4
        cmp x2,#0
        bne lp
ret
```

步骤 4 编译并运行可执行文件

输入命令 gcc time.c copy122.s -o m122 执行程序。

```
gcc time.c copy122.s -o m122
./m122
```

结果为 33292783ns，如图 3-6 所示：

```
[root@Malluma memory]# gcc time.c copy122.s -o m122
[root@Malluma memory]# ./m122
memorycopy time is    33292783 ns
```

图2-5 编译执行程序 m122

函数执行时间基于二倍的基础上也得到了优化，那么目前根据实验现象可以看出，更多次数的循环展开可能会让程序的执行时间得到更好的优化。

2.2.3 内存突发传输方式优化

之前两次优化每次内存读写都是以一个字节为单位进行的，这样效率很低。由于内存在连续读/写多个数据时，其性能要优于非连续读/写数据的方式，此次优化思路是一次对多个字节进行读写。这就用到了 ldp 指令和 stp 指令，这两条指令可以一次访问 16 个字节的内存数据，大大提高了内存读写效率。

步骤 1 创建内存突发传输优化 copy21.s 文件

```
.global memorycopy
memorycopy:
ldp x3,x4,[x1],#16   //ldp 指令将 x1 向上加 16 个字节后存放到 x3 和 x4 中
stp x3,x4,[x0],#16
sub x2,x2,#16
cmp x2,#0
bne memorycopy
ret
```

步骤 2 编译并运行可执行文件

编译执行 gcc time.c copy21.s -o m21 和./m21 命令。

```
gcc time.c copy21.s -o m21
./m21
```

如图 3-16 所示：

```
[root@ecs-01 memory]# gcc time.c copy21.s -o m21
[root@ecs-01 memory]# ./m21
memorycopy time is    12785612 ns
```

图2-6 执行 m21 程序

一次对 16 字节读写程序执行效率明显优于单字节读写。