第一章 同步與互斥

1.1 內聯彙編

要深入理解 Linux 內核中的同步與互斥的實現，需要先瞭解一下內聯彙編：在 C 函數中使用彙編代碼。

現代編譯器已經足夠優秀，大部分的 C 代碼轉成匯編碼後，效率都很高。但是有些特殊的演算法需要我們手工優化，這時就需要手寫彙編代碼；或是有時需要調用特殊的彙編指令(比如使用 ldrex/strex 實現互斥訪問)，這都涉及內聯彙編。

實際上你完全可以不使用內聯彙編，單獨寫一個遵守 ATPCS 規則的彙編函數，讓 C 函數去調用它。但是在 C 函數中寫彙編代碼，可以不用另外新建一個彙編檔，比較方便。

內聯彙編的完整語法比較複雜，可以參考這 3 篇文章：

① GNU C 擴展彙編 https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

② ARM GCC 內嵌（inline）彙編手冊 http://blog.chinaunix.net/uid-20543672-id-3194385.html

③ C 內聯彙編 https://akaedu.github.io/book/ch19s05.html

這 3 章文章寫得細緻而深入，也有些難以理解。你跟著我們的視頻或文檔，就可以掌握到足夠的知識。

下面舉 3 個例子說明彙編函數、用 C 函數中使用內聯彙編的方法。

1.1.1 C 語言實現加法

使用 GIT 下載後，源碼在“07_驅動大全\source\01_inline_assembly\01_c_code\main.c”：

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int add(int a, int b)
05 {
06     return a+b;
07 }
08
09 int main(int argc, char **argv)
10 {
11     int a;
12     int b;
13
14     if (argc != 3)
15     {
16         printf("Usage: %s <val1> <val2>\n", argv[0]);
17         return -1;
18     }
19
20     a = (int)strtol(argv[1], NULL, 0);
21     b = (int)strtol(argv[2], NULL, 0);
22
23     printf("%d + %d = %d\n", a, b, add(a, b));
24     return 0;
```

```
25 }
26
```

上面的 add 函數代碼最簡單，但是對應的彙編也挺複雜：需要入棧、出棧等操作，效率不算高。看看 test.dis：

```
266 00010404 <add>:
267    10404:   b480            push    {r7}
268    10406:   b083            sub     sp, #12
269    10408:   af00            add     r7, sp, #0
270    1040a:   6078            str     r0, [r7, #4]
271    1040c:   6039            str     r1, [r7, #0]
272    1040e:   687a            ldr     r2, [r7, #4]
273    10410:   683b            ldr     r3, [r7, #0]
274    10412:   4413            add     r3, r2                  // 真正實現加法的只有這條指令
275    10414:   4618            mov     r0, r3
276    10416:   370c            adds    r7, #12
277    10418:   46bd            mov     sp, r7
278    1041a:   f85d 7b04       ldr.w   r7, [sp], #4
279    1041e:   4770            bx      lr
280
```

1.1.2 使用彙編函數實現加法

使用 GIT 下載後，源碼在“07_驅動大全\source\01_inline_assembly\02_assembly\add.S”：

```
01 .text            // 放在程式碼片段
02 .global  add     // 實現全域函數 add
03 .thumb           // 使用 thumb 指令, main.c 預設使用 thumb 指令, 所以這裡也使用 thumb 指令
04
05 add:
06      add r0, r0, r1
07      bx lr
08
```

根據 ATPCS 規則，main 函式呼叫 add(a, b)時，會把第一個參數存入 r0 寄存器，把第二個參數存入 r1 寄存器。

在上面第 06 行裡，把 r0、r1 累加後，結果存入 r0：根據 ATPCS 規則，r0 用來保存返回值。

可以看到，這個 add 函數連棧都沒有使用，非常高效。

這只是一個很簡單的例子，我們工作中並不使用彙編來進行“加法優化”，在計算量非常大的地方可以考慮單獨編寫彙編函數實現優化。

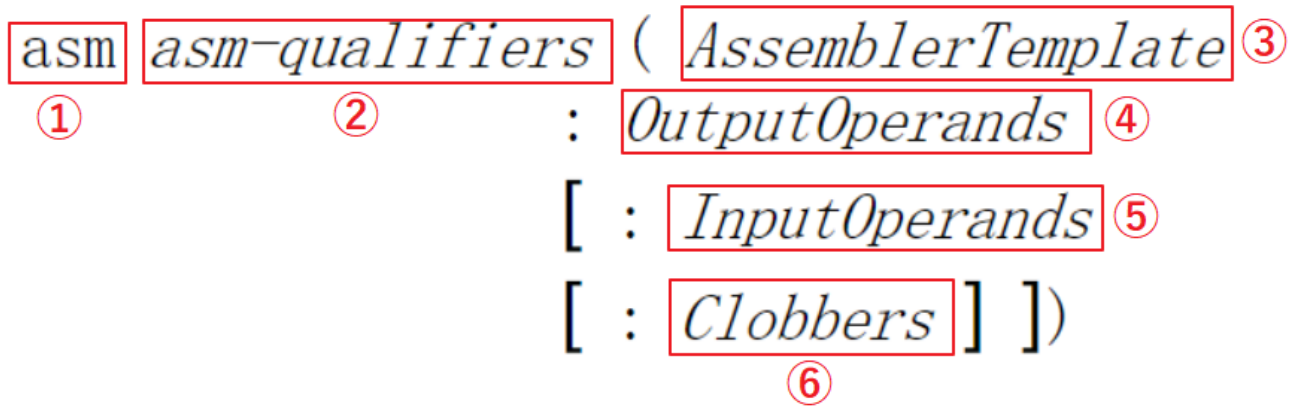
1.1.3 內聯彙編語法

從上面例子可以看到，我們完全可以新建一個彙編檔，在 ATPCS 規則之下編寫代碼，這樣 C 函數就可以直接調用彙編函數。

但是，需要新建彙編檔，有點麻煩。

使用內聯彙編，可以在 C 代碼中內嵌彙編代碼。

先看看內聯彙編的語法。




```
int add(int a, int b)
{
    int sum;
    __asm__ volatile (
        "add %0, %1, %2"
        :"=r"(sum)
        :"r"(a), "r"(b)
        :"cc"
    );
    return sum;
}
```

內聯彙編語法：

① asm

也可以寫作“__asm__”，表示這是一段內聯彙編。

② asm-qualifiers

有 3 個取值：volatile、inline、goto。

volatile 的意思是易變的、不穩定的，用來告訴編譯器不要隨便優化這段代碼，否則可能出問題。比如彙編指令“mov r0, r0”，它把 r0 的值複製到 r0，並沒有實際做什麼事情，你的本意可能是用這條指令來延時。編譯器看到這指令後，可能就把它去掉了。加上 volatile 的話，編譯器就不會擅自優化。

其他 2 個取值我們不關心，也比較難以理解，不講。

③ AssemblerTemplate

彙編指令，用雙引號包含起來，每條指令用“\n”分開，比如：

```
“mov  %0, %1\n”
“add  %0, %1, %2\n”
```

④ OutputOperands

輸出操作數，內聯彙編執行時，輸出的結果保存在哪裡。

格式如下，當有多個變數時，用逗號隔開：

```
[ [asmSymbolicName] ] constraint (cvariablename)
```

asmSymbolicName 是符號名，隨便取，也可以不寫。

constraint 表示約束，有如下常用取值：

constraint	描述
m	memory operand，表示要傳入有效的位址，只要 CPU 能支援該位址，就可以傳入
r	register operand，寄存器運算元，使用寄存器來保存這些運算元
i	immediate integer operand，表示可以傳入一個立即數

constraint 前還可以加上一些修飾字元，比如“=r”、“+r”、“=&r”，含義如下：

constraint Modifier Characters	描述
=	表示內聯彙編會修改這個運算元，即：寫
+	這個運算元即被讀，也被寫
&	它是一個 earlyclobber 運算元

cvariablename：C 語言的變數名。

示例 1 如下：

```
[result] "=r" (sum)
```

它的意思是彙編代碼中會通過某個寄存器把結果寫入 sum 變數。在彙編代碼中可以使用“%[result]”來引用它。

示例 2 如下：

```
"=r" (sum)
```

在彙編代碼中可以使用“%0”、“%1”等來引用它，這些數值怎麼確定後面再說。

⑤ InputOperands

輸入運算元，內聯彙編執行前，輸入的資料保存在哪裡。

格式如下，當有多個變數時，用逗號隔開：

```
[ [asmSymbolicName] ] constraint (cexpression)
```

asmSymbolicName 是符號名，隨便取，也可以不寫。

constraint 表示約束，參考上一小節，跟 OutputOperands 類似。

cexpression：C 語言的運算式。

示例 1 如下：

```
[a_val]"r"(a), [b_val]"r"(b)
```

它的意思變數 a、b 的值會放入某些寄存器。在彙編代碼中可以使用%[a_val]、%[b_val]使用它們。

示例 2 如下：

```
"r"(a), "r"(b)
```

它的意思變數 a、b 的值會放入某些寄存器。在彙編代碼中可以使用%0、%1 等使用它們，這些數值後面再說。

⑥ Clobbers

在彙編代碼中，對於“OutputOperands”所涉及的寄存器、記憶體，肯定是做了修改。但是彙編代碼中，也許要修改的寄存器、記憶體會更多。比如在計算過程中可能要用到 r3 保存臨時結果，我們必須在“Clobbers”中聲明 r3 會被修改。

下面是一個例子：

```
: "r0", "r1", "r2", "r3", "r4", "r5", "memory"
```

我們常用的是有“cc”、“memory”，意義如下：

Clobbers	描述
cc	表示彙編代碼會修改“flags register”
memory	表示彙編代碼中，除了“InputOperands”和“OutputOperands”中指定的之外，還會會讀、寫更多的記憶體

1.1.4 編寫內聯彙編實現加法

使用 GIT 下載後，源碼在“07_驅動大全\source\01_inline_assembly\03_inline_assembly\main.c”：

```
04 int add(int a, int b)
05 {
06     int sum;
07     __asm__ volatile (
08             "add %0, %1, %2"
09             :"=r"(sum)
10             :"r"(a), "r"(b)
11             :"cc"
12     );
13     return sum;
```

```
int add(int a, int b)
{
    int sum;
    __asm__ volatile (
        "add %0, %1, %2"
        :"=r"(sum)
        :"r"(a), "r"(b)
        :"cc"
    );
    return sum;
}
```

第0个操作数
第1个
第2个

所以第 08 行代碼就是：把第 1、2 個運算元相加，存入第 0 個運算元。也就是把 a、b 相加，存入 sum。

還可以使用另一種寫法，在 Linux 內核中這種用法比較少見。
使用 GIT 下載後，源碼在“07_驅動大全\source\01_inline_assembly\03_inline_assembly\main2.c”：

```
int add(int a, int b)
{
    int sum;
    __asm__ volatile (
        "add %[result], %[val1], %[val2]"
        :[result]"=r"(sum)
        :[val1]"r"(a), [val2]"r"(b)
        :"cc"
    );
    return sum;
}
```

1.1.5 earlyclobber 的例子

OutputOperands 的約束中經常可以看到“=&r”，其中的“&”表示 earlyclobber，它是最難理解的。有一些輸出操作數在彙編代碼中早早就被寫入了新值 A，在這之後，彙編代碼才去讀取某個輸入運算元，這個輸出操作數就被稱為 earlyclobber(早早就被改了)。

這可能會有問題：假設早早寫入的新值 A，寫到了 r0 寄存器；後面讀輸入運算元時得到數值 B，也可能寫入 r0 寄存器，這新值 A 就被破壞了。

核心原因就在於輸出操作數、輸入運算元都用了同一個 r0 寄存器。為什麼要用同一個？因為編譯器不知道你是 earlyclobber 的，它以為是先讀入了所有輸入運算元，都處理完了，才去寫輸出操作數的。按這流程，沒人來覆蓋新值 A。

所以，如果彙編代碼中某個輸出操作數是 earlyclobber 的，它的 constraint 就要加上“&”，這就是告訴編譯器：給我分配一個單獨的寄存器，別為了省事跟輸入運算元用同一個寄存器。

使用 GIT 下載後，源碼在“07_驅動大全\source\01_inline_assembly\04_earlyclobber\main.c”：

```
4: int add(int a, int b)
5: {
6:     int sum;
7:     __asm__ volatile (
8:         "add %0, %1, %2\n"
9:         "add %1, #1\n"
10:        "add %2, #1\n"
11:        :"=r"(sum)
12:        :"r"(a), "r"(b)
13:        :"cc"
14:    );
15:    return sum;
16: }
```

```
00010404 <add>:
   10404:       b480            push    {r7}
   10406:       b085            sub     sp, #20
   10408:       af00            add     r7, sp, #0
   1040a:       6078            str     r0, [r7, #4]
   1040c:       6039            str     r1, [r7, #0]
   1040e:       687b            ldr     r3, [r7, #4]
   10410:       683a            ldr     r2, [r7, #0]
   10412:       4413            add     r3, r2
   10414:       f103 0301       add.w   r3, r3, #1
   10418:       f102 0201       add.w   r2, r2, #1
   1041c:       60fb            str     r3, [r7, #12]
   1041e:       68fb            ldr     r3, [r7, #12]
   10420:       4618            mov     r0, r3
   10422:       3714            adds    r7, #20
   10424:       46bd            mov     sp, r7
   10426:       f85d 7b04       ldr.w   r7, [sp], #4
   1042a:       4770            bx      lr
```

上面的代碼中，輸出操作數%0 對應的寄存器是 r3，輸入運算元%1 對應的寄存器也是 r3。
第 8 行更新了%0 的值後，第 9 行修改%1 的值，由於%0、%1 是同一個寄存器，所以%0 的值也被修改了。
最終返回的累加值是錯的，增加了 1，如下圖所示：

```
[root@board:~]# /mnt/test 1 2
1 + 2 = 4
```

怎麼修改？在第 11 行加“&”就可以了，這是告訴編譯器，對於%0 運算元它是 earlyclobber 的，不能跟其他運算元共用寄存器，如下：

```
4:  int add(int a, int b)
5:  {
6:      int sum;
7:      __asm__ volatile (
8:          "add %0, %1, %2\n"
9:          "add %1, #1\n"
10:         "add %2, #1\n"
11:         :"=&r"(sum)
12:         :"r"(a), "r"(b)
13:         :"cc"
14:     );
15:     return sum;
16: }
```

```
00010404 <add>:
   10404:       b480            push    {r7}
   10406:       b085            sub     sp, #20
   10408:       af00            add     r7, sp, #0
   1040a:       6078            str     r0, [r7, #4]
   1040c:       6039            str     r1, [r7, #0]
   1040e:       687a            ldr     r2, [r7, #4]
   10410:       6839            ldr     r1, [r7, #0]
   10412:       eb02 0301       add.w   r3, r2, r1
   10416:       f102 0201       add.w   r2, r2, #1
   1041a:       f101 0101       add.w   r1, r1, #1
   1041e:       60fb            str     r3, [r7, #12]
   10420:       68fb            ldr     r3, [r7, #12]
   10422:       4618            mov     r0, r3
   10424:       3714            adds    r7, #20
   10426:       46bd            mov     sp, r7
   10428:       f85d 7b04       ldr.w   r7, [sp], #4
   1042c:       4770            bx      lr
```

從右邊的反匯編碼可以知道，%0 跟%1、%2 使用不一樣的寄存器，所以後面第 9、10 行無法影響到%0 的值。

程式運行結果如下圖所示：

```
[root@board:~]# /mnt/test 1 2
1 + 2 = 3
```