1.3 原子操作的實現原理與使用

在上面的第 2 個失敗例子裡，問題在於對 valid 變數的修改被打斷了。如果對 valid 變數的操作不能被打斷，就解決這個問題了。

這可以使用原子操作，所謂“原子操作”就是這個操作不會被打斷。Linux 有 2 種原子操作：原子變數、原子位元。

1.3.1 原子變數的內核操作函數

原子變數的操作函數在 Linux 內核檔 arch\arm\include\asm\atomic.h 中。

原子變數類型如下，實際上就是一個結構體(內核檔 include/linux/types.h)：

```
typedef struct {
    int counter;
} atomic_t;
```

特殊的地方在於它的操作函數，如下(下表中 v 都是 atomic_t 指標)：

函數名	作用
atomic_read(v)	讀出原子變數的值，即 v->counter
atomic_set(v,i)	設置原子變數的值，即 v->counter = i
atomic_inc(v)	v->counter++
atomic_dec(v)	v->counter--
atomic_add(i,v)	v->counter += i
atomic_sub(i,v)	v->counter -= i
atomic_inc_and_test(v)	先加 1，再判斷新值是否等於 0；等於 0 的話，返回值為 1
atomic_dec_and_test(v)	先減 1，再判斷新值是否等於 0；等於 0 的話，返回值為 1

1.3.2 原子變數的內核實現

注意：SMP 就是 Symmetric Multi-Processors，對稱多處理器；UP 即 Uni-Processor，系統只有一個單核 CPU。

這些函數都是在 Linux 內核檔 arch\arm\include\asm\atomic.h 中。

atomic_read，atomic_set 這些操作都只需要一條彙編指令，所以它們本身就是不可打斷的。

問題在於 atomic_inc 這類操作，要讀出、修改、寫回。

以 atomic_inc 為例，在 atomic.h 文件中，如下定義：

```
#define atomic_inc(v)        atomic_add(1, v)
```

atomic_add 又是怎樣實現的呢？用下面這個宏：

```
ATOMIC_OPS(add, +=, add)
```

把這個宏展開：

```
#define ATOMIC_OPS(op, c_op, asm_op)                    \
    ATOMIC_OP(op, c_op, asm_op)                 \
    ATOMIC_OP_RETURN(op, c_op, asm_op)              \
    ATOMIC_FETCH_OP(op, c_op, asm_op)
```

從上面的宏可以知道，一個 ATOMIC_OPS 定義了 3 個函數。比如“ATOMIC_OPS(add, +=, add)”就定義了這 3 個函數：

```
atomic_add
atomic_add_return
atomic_atomic_fetch_add 或 atomic_fetch_add_relaxed
```

我們以 ATOMIC_OP(add, +=, add)為例，看它是如何實現 atomic_add 函數的，對於 UP 系統、SMP 系統，分別有不同的實現方法。

1.3.2.1 ATOMIC_OP 在 UP 系統中的實現

對於 ARMv6 以下的 CPU 系統，不支援 SMP。原子變數的操作簡單粗暴：關中斷，中斷都關了，誰能來打斷我？代碼如下(arch\arm\include\asm\atomic.h)：

```
#define ATOMIC_OP(op, c_op, asm_op)                     \
static inline void atomic_##op(int i, atomic_t *v)          \
{                                                       \
    unsigned long flags;                                \
                                                        \
    raw_local_irq_save(flags);      先关中断            \
    v->counter c_op i;                                  \
    raw_local_irq_restore(flags);   再恢复原来的中断    \
}                                                       \
```

1.3.2.2 ATOMIC_OP 在 SMP 系統中的實現

對於 ARMv6 及以上的 CPU，有一些特殊的彙編指令來實現原子操作，不再需要關中斷，代碼如下(arch\arm\include\asm\atomic.h)：

```
#define ATOMIC_OP(op, c_op, asm_op)                     \
static inline void atomic_##op(int i, atomic_t *v)          \
{                                                       \
    unsigned long tmp;                                  \
    int result;                                         \
                                                        \
    prefetchw(&v->counter);                             \
    __asm__ __volatile__("@ atomic_" #op "\n"           \
"1: ldrex   %0, [%3]\n"         1.读出                  \
"   " #asm_op " %0, %0, %4\n"   2.修改                  \
"   strex   %1, %0, [%3]\n"     3.写入                  \
"   teq %1, #0\n"   4.中途被别人先修改了？              \
"   bne 1b"           5.那就重做一次                    \
    : "=&r" (result), "=&r" (tmp), "+Qo" (v->counter)   \
    : "r" (&v->counter), "Ir" (i)                       \
    : "cc");                                            \
}                                                       \
```

在 ARMv6 及以上的架構中，有 ldrex、strex 指令，ex 表示 exclude，意為獨佔地。這 2 條指令要配合使用，舉例如下：

① 讀出：ldrex　r0, [r1]

讀取 r1 所指記憶體的資料，存入 r0；並且標記 r1 所指記憶體為“獨佔訪問”。

如果有其他程式再次執行“ldrex　r0, [r1]”，一樣會成功，一樣會標記 r1 所指記憶體為“獨佔訪問”。

② 修改 r0 的值

③ 寫入：strex　r2, r0, [r1]：

如果 r1 的“獨佔訪問”標記還存在，則把 r0 的新值寫入 r1 所指記憶體，並且清除“獨佔訪問”的標記，把 r2 設為 0 表示成功。

如果 r1 的“獨佔訪問”標記不存在了，就不會更新記憶體，並且把 r2 設為 1 表示失敗。

假設這樣的搶佔場景：

① 程式 A 在讀出、修改某個變數時，被程式 B 搶佔了；

② 程式 B 先完成了操作，程式 B 的 strex 操作會清除“獨佔訪問”的標記；

③ 輪到程式 A 執行剩下的寫入操作時，它發現獨佔訪問”標記不存在了，於是取消寫入操作。

這就避免了這樣的事情發生：程式 A、B 同時修改這個變數，並且都自認為成功了。

舉報個例子，比如 atomic_dec，假設一開始變數值為 1，程式 A 本想把值從 1 變為 0；但是中途被程式 B 先把值從 1 變成 0 了；但是沒關係，程式 A 裡會再次讀出新值、修改、寫入，最終這個值被程式 A 從 0 改為-1。

在 ARMv6 及以上的架構中，原子操作不再需要關閉中斷，關中斷的花銷太大了。並且關中斷並不適合 SMP 多 CPU 系統，你關了 CPU0 的中斷，CPU1 也可能會來執行些操作啊。

在 ARMv6 及以上的架構中，原子操作的執行過程是可以被打斷的，但是它的效果符合“原子”的定義：一個完整的“讀、修改、寫入”原子的，不會被別的程式打斷。它的思路很簡單：如果被別的程式打斷了，那就重來，最後總會成功的。

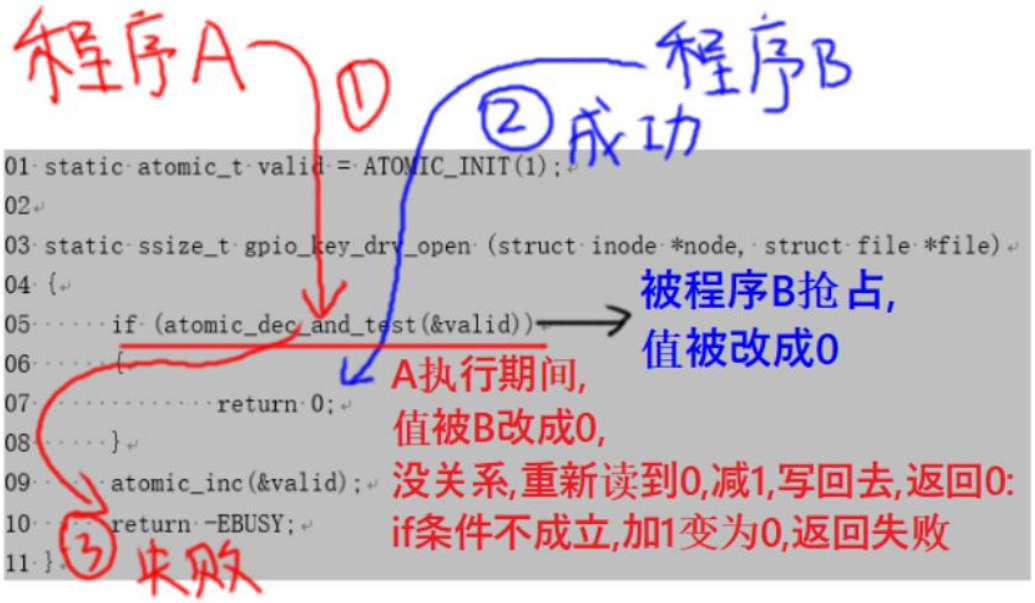
1.3.3 原子變數使用案例

現在可以使用原子變數實現：只能有一個 APP 訪問驅動程式。代碼如下：

```
01 static atomic_t valid = ATOMIC_INIT(1);
02
03 static ssize_t gpio_key_drv_open (struct inode *node, struct file *file)
04 {
05     if (atomic_dec_and_test(&valid))
06     {
07         return 0;
08     }
09     atomic_inc(&valid);
10     return -EBUSY;
11 }
12
13 static int gpio_key_drv_close (struct inode *node, struct file *file)
14 {
15     atomic_inc(&valid);
16     return 0;
17 }
18
```

第 5 行的 atomic_dec_and_test，這是一個原子操作，在 ARMv6 以下的 CPU 架構中，這個函數是在關中斷的情況下執行的，它確實是“原子的”，執行過程不被打斷。

但是在 ARMv6 及以上的 CPU 架構中，這個函數其實是可以被打斷的，但是它實現了原子操作的效果，如下圖所示：



1.3.4 原子位介紹

1.3.4.1 原子位元的內核操作函數

能操作原子變數，再去操作其中的某一位，不是挺簡單的嘛？不過不需要我們自己去實現，內核做好了。

原子位元的操作函數在 Linux 內核檔 arch\arm\include\asm\bitops.h 中，下表中 p 是一個 unsigned long 指標。

函數名	作用
set_bit(nr,p)	設置(*p)的 bit nr 為 1
clear_bit(nr,p)	清除(*p)的 bit nr 為 0
change_bit(nr,p)	改變(*p)的 bit nr，從 1 變為 0，或是從 0 變為 1
test_and_set_bit(nr,p)	設置(*p)的 bit nr 為 1，返回該位的老值
test_and_clear_bit(nr,p)	清除(*p)的 bit nr 為 0，返回該位的老值
test_and_change_bit(nr,p)	改變(*p)的 bit nr，從 1 變為 0，或是從 0 變為 1；返回該位的老值

1.3.4.2 原子位的內核實現

在 ARMv6 以下的架構裡，不支援 SMP 系統，原子位元的操作函數也是簡單粗暴：關中斷。以 set_bit 函數為例，代碼在內核檔 arch\arm\include\asm\bitops.h 中，如下

在 ARMv6 及以上的架構中，不需要關中斷，有 ldrex、strex 等指令，這些指令的作用在前面介紹過。還是以 set_bit 函數為例，代碼如下：

我不再使用原子位操作來寫代碼，留給你們練習吧。