hw05.syswrite

1. 結果(用證明的方式證明是對的)

   (1) store : memory_order_release、load : memory_order_acquire 結果正確

```
void p1(void) {
    printf("start p1\n");
    while (1) {
        atomic_store_explicit(&flag[1], 1,memory_order_release);
        atomic_thread_fence(memory_order_seq_cst);
        atomic_store_explicit(&turn, 0,memory_order_release);
        while (atomic_load_explicit(&flag[0],memory_order_acquire) && atomic_load_explicit(&turn,
memory_order_acquire)==0)
            ;   //waiting
        //底下程式碼用於模擬在critical section
        cpu_p1 = sched_getcpu();
        in_cs++;
        //nanosleep(&ts, NULL);
        if (in_cs == 2) fprintf(stderr, "p0及p1都在critical section\n");
        p1_in_cs++;
        //nanosleep(&ts, NULL);
        in_cs--;
        //離開critical section
        atomic_store_explicit(&flag[1], 0,memory_order_release);
    }
}
```

```
~/OS_HW/HW0C ./peterson_correct_acquire_release-g
start p0
start p1
進入次數（每秒） p0: 4397629, p1: 4398030, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4316651, p1: 4316770, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4744910, p1: 4746298, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4807747, p1: 4806967, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4810274, p1: 4812343, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4782031, p1: 4780404, 分別執行於 core#1 及 core#0
```

2. 相關說明與證明

```
void p0(void) {
    while (1) {
        atomic_store_explicit(&flag[0], 1,memory_order_release);
        atomic_thread_fence(memory_order_seq_cst);
        atomic_store_explicit(&turn, 1,memory_order_release);
        while (atomic_load_explicit(&flag[1],memory_order_acquire) && atomic_load_explicit(&turn,memory_order_acquire)==1)
                ;   //waiting
        //critical section
        cpu_p0 = sched_getcpu();
        in_cs++;    /
        if (in_cs == 2) fprintf(stderr, "p0及p1都在critical section\n");
        p0_in_cs++;
        in_cs--;
        //Peteron's solution的離開部分的程式碼
        atomic_store_explicit(&flag[0], 0,memory_order_release);
    }
}

void p1(void) {
    while (1) {
        atomic_store_explicit(&flag[1], 1,memory_order_release);
        atomic_thread_fence(memory_order_seq_cst);
        atomic_store_explicit(&turn, 0,memory_order_release);
        while (atomic_load_explicit(&flag[0],memory_order_acquire) && atomic_load_explicit(&turn,memory_order_acquire)==0)
                ;   //waiting
        //critical section
        cpu_p1 = sched_getcpu();
        in_cs++;
        if (in_cs == 2) fprintf(stderr, "p0及p1都在critical section\n");
        p1_in_cs++;
        in_cs--;
        //離開critical section
        atomic_store_explicit(&flag[1], 0,memory_order_release);
    }
}
```

使用 store : memory_order_release、load : memory_order_acquire 會達到以下的效果

(1)P0 load-acquire 之後可以看見 P1 store-release 之前寫入的資料；
(2)P1 load-acquire 之後可以看見 P0 store-release 之前寫入的資料。
(3)load-acquire 之後的 read/write 不能搬到 load-acquire 之前。
(4)store-release 之前的 read/write 不能搬到 store-release 之後。

證明如下：

(i) mutual exclusion:

- 執行順序存取的保證性

由於(4) store-release 之前的 read/write 不能搬到 store-release 之後
保證會先設定 flag 才會設定 turn，並且進入 critical section 後才會修改 flag 設定

- P0 和 P1 執行的互斥性

如果 P0 在 CS 內，那麼 flag[1]為 false(意味著 P1 已經離開 CS)，或者 turn 為 0(意味著 P1 只能在 while 等待，不能進入 CS)
並且保證會先設定 turn 再設定 flag 不會造成此現象

```
Process 1: turn = 0;
Process 0: turn = 1;
Process 0: flag[0] = true;
Process 0: while(flag[1] && turn==1) // terminates, since flag[1]==false
Process 0: enter critical section 0
Process 1: flag[1] = true;
Process 1: while(flag[0] && turn==0) // terminates, since turn==1
Process 1: enter critical section 1
Process 0: exit critical section 0
Process 1: exit critical section 1
```

(ii) progress:

- CS 為空，想進去的 thread 可以進去性

假設 P0 拿到 CS 做完後，CS 為空會用 store-release 修改 flag，
若 P1 在 while 等待會用 load-acquire 看到後，P1 就可以進入 CS；
否則 P1 還沒開始等，若 P1 先設定讓先，P0 後設定讓先，則 P1 進入 CS
否則 P1 進入等待，P0 一做完 P1 就可以進去 CS，證明了 P0 一離開 CS，CS 為空 P1 就有機會進去；P0 的證明同理，故達到 progress

(iii) bound waiting

- 等待有限次性

假設 P0 先拿到 CS，P0 做完後會改成設定讓 P1 先進去，在等待 P1 就可以等 1 次 P0 就進入 CS，P1 也同理，故達到 bound waiting