作業 0A：實現 Peterson’s solution

資工三 406410114 郭晏誠

1. 直接打 make 會得到四個執行檔案

(1)請將這四個執行檔案都執行過一次，請問你的執行節果為何？請附上畫面截圖（問題一）

(i) ./peterson_trival-03

結果錯誤 只有進去第一次

```
~/OS_HW/HW0A ./peterson_trival-O3
p0: start
p1: start
進入次數（每秒） p0:  1986, p1:     0, 分別執行於 core#0 及 core#0
進入次數（每秒） p0:     0, p1:     0, 分別執行於 core#0 及 core#0
進入次數（每秒） p0:     0, p1:     0, 分別執行於 core#0 及 core#0
進入次數（每秒） p0:     0, p1:     0, 分別執行於 core#0 及 core#0
進入次數（每秒） p0:     0, p1:     0, 分別執行於 core#0 及 core#0
進入次數（每秒） p0:     0, p1:     0, 分別執行於 core#0 及 core#0
進入次數（每秒） p0:     0, p1:     0, 分別執行於 core#0 及 core#0
進入次數（每秒） p0:     0, p1:     0, 分別執行於 core#0 及 core#0
```

(ii) ./peterson_trival-g

結果有時正確有時錯誤

正確如下：(可能 CPU 運算有時候快有時慢)

```
~/OS_HW/HW0A ./peterson_trival-g
p1: start
p0: start
進入次數（每秒） p0: 7434836, p1: 7439298, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7233260, p1: 7233261, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7754947, p1: 7755417, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7603041, p1: 7603037, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7552664, p1: 7552661, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7559066, p1: 7559057, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7576736, p1: 7577429, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7576975, p1: 7576957, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7565939, p1: 7565956, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7565092, p1: 7564661, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7506074, p1: 7505997, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7505593, p1: 7505210, 分別執行於 core#0 及 core#1
```

錯誤如下：

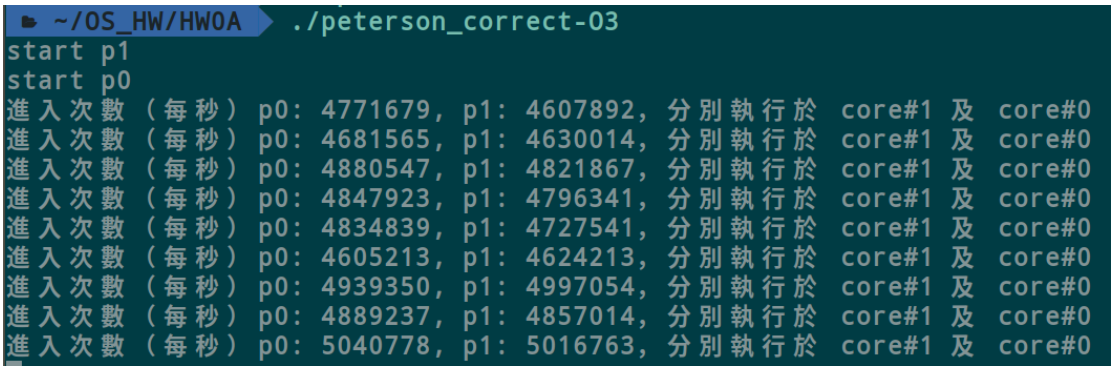
```
~/OS_HW/HW0A ./peterson_trival-g
p1: start
p0: start
進入次數（每秒） p0: 7555592, p1: 7557852, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7538058, p1: 7538129, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7621202, p1: 7621179, 分別執行於 core#0 及 core#1
p0及p1都在critical section
進入次數（每秒） p0: 7583760, p1: 7583774, 分別執行於 core#0 及 core#1
```

(iii) ./peterson_correct-g

結果正確（若多核心有可能換核心執行因為 CPU 比較燙）

(iv) ./peterson_correct-03

結果正確



2. 請試著解釋「為什麼」 peterson_trival-03 的執行結果是錯的

- 請反組譯「 peterson_trival-03 」中的 P0 函數
- 你可以修改「peterson_trival.c」盡量讓程式碼精簡

(1)gdb peterson_trival-03 和 gdb peterson_correct-03 可發現 correct 多了 mfence

▼gdb peterson_trival-03

```
~/OS_HW/HW0A  gdb ./peterson_trival-03
>>> disassemble p0
Dump of assembler code for function p0:
   0x00000000004008d0 <+0>:     sub    $0x8,%rsp
   0x00000000004008d4 <+4>:     mov    $0x400a64,%edi
   0x00000000004008d9 <+9>:     callq  0x4006b0 <puts@plt>
   0x00000000004008de <+14>:    mov    0x2007b8(%rip),%eax
   0x00000000004008e4 <+20>:    nopl   0x0(%rax)
   0x00000000004008e8 <+24>:    cmp    $0x1,%eax
   0x00000000004008eb <+27>:    jne    0x4008e8 <p0+24>
   0x00000000004008ed <+29>:    jmp    0x4008ed <p0+29>
End of assembler dump.
```

▼gdb peterson_correct-03

```
~/OS_HW/HW0A  gdb ./peterson_correct-03
Dump of assembler code for function p0:
   0x0000000000400940 <+0>:     sub    $0x8,%rsp
   0x0000000000400944 <+4>:     mov    $0x400b50,%edi
   0x0000000000400949 <+9>:     callq  0x4006b0 <puts@plt>
   0x000000000040094e <+14>:    xchg   %ax,%ax
   0x0000000000400950 <+16>:    movl   $0x1,0x20073e(%rip)
   0x000000000040095a <+26>:    mfence
   0x000000000040095d <+29>:    mfence
   0x0000000000400960 <+32>:    movl   $0x1,0x200736(%rip)
   0x000000000040096a <+42>:    mfence
   0x000000000040096d <+45>:    jmp    0x40097b <p0+59>
   0x000000000040096f <+47>:    nop
```

(2)peterson_trival-03 的執行結果是錯是因為沒有 mfence 讓讀寫串行化，導致 p0 跟 p1 可能通時進入 critical section 進行修改全域資源，可能造成錯誤。

(3)mfence 補充

(i)定義：保證 Memory access 的讀寫都串行化，其實是在一系列 Memory access 中添加若干延遲(屏障)，可用來保證此指令的 Memory access 完成後，後面的指令才可 Memory access，相關指令還有還有 lfence(讀串行化)、sfence(寫串行化)

(ii) x86 mfence 實作概念

(a) sfence/mfenceg 會將 store buffer 中緩存的修改刷入 L1 cache 中，使其他 cpu 核可以觀察到這些修改

(b) sfence/mfence 之後的寫操作，不會被調度到 sfence/mfence 之前。

(iii)效能優化

(a) 新的 Intel 處理器為了提升效能，新增了 memory order

(b) 允許預讀取操作先前存到 load buffer 中。

(c) 允許 store buffer 的緩存，讓先寫再讀操作變成先讀再寫。

(d) string 操作指令和繞過 cache 的 write 指令（MOVNTI, MOVNTDQ 等）有 priority。

3. 請說明在你的電腦上，上述二個程式的正確與否，並說明速度的快慢

- 「peterson_trival-g」的速度比「 peterson_correct-03」還要來得快(excel 畫成表)
- 需附上 cpu 型號與 gcc 版本

(1) peterson_trival-g 是錯的，peterson_correct-03 是正確的

peterson_trival-g 錯誤如下：

```
~/OS_HW/HW0A ./peterson_trival-g
p1: start
p0: start
進入次數（每秒） p0: 7555592, p1: 7557852, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7538058, p1: 7538129, 分別執行於 core#0 及 core#1
進入次數（每秒） p0: 7621202, p1: 7621179, 分別執行於 core#0 及 core#1
p0及p1都在critical section
進入次數（每秒） p0: 7583760, p1: 7583774, 分別執行於 core#0 及 core#1
```

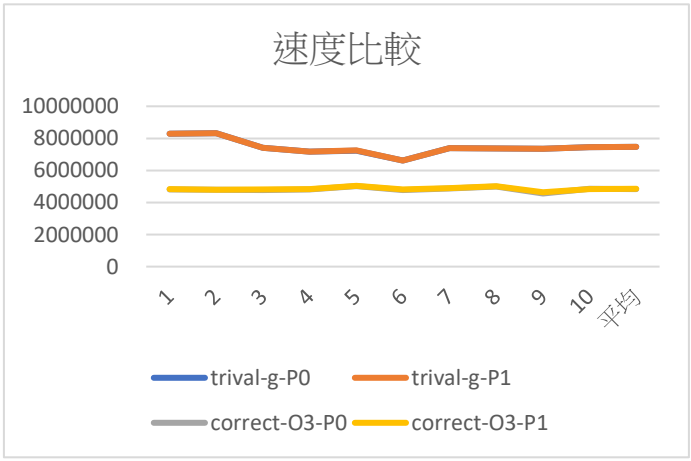
peterson_correct-03 運行正確：

```
~/OS_HW/HW0A ./peterson_correct-03
start p1
start p0
進入次數（每秒） p0: 4771679, p1: 4607892, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4681565, p1: 4630014, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4880547, p1: 4821867, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4847923, p1: 4796341, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4834839, p1: 4727541, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4605213, p1: 4624213, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4939350, p1: 4997054, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 4889237, p1: 4857014, 分別執行於 core#1 及 core#0
進入次數（每秒） p0: 5040778, p1: 5016763, 分別執行於 core#1 及 core#0
```

(2) 速度比較

(i)明顯 peterson_trival-g 比 peterson_correct-03 快 1.53 倍

	peterson_trival-g		peterson_correct-03	
實驗次數	P0	P1	P0	P1
1	8297021	8287943	4815676	4839527
2	8320601	8321127	4797272	4800218
3	7410145	7410987	4785693	4808459
4	7180751	7181048	4819306	4836898
5	7239093	7245478	5021556	5040582
6	6617149	6617073	4782907	4808973
7	7385823	7385823	4874496	4900021
8	7379195	7379111	5000973	5018152
9	7345502	7347220	4561879	4639658
10	7450935	7458881	4853873	4844018
平均	7462622	7463469	4831363	4853651



(ii)gcc version： 4.9.3

(iii)CPU 型號

Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.20 GHz

4. 盡可能的解釋即可，最起碼附上每一行組語的意義

(1) gdb ./peterson_correct-03 因為 mfence 變慢了

```
#每一行意思
#使 03 變慢的原因
>>> disassemble p0
Dump of assembler code for function p0:
   0x0000000000400940 <+0>: sub    $0x8,%rsp          # stack frame
   0x0000000000400944 <+4>: mov    $0x400b50,%edi     #用 edi 保存地址 0x400b50
    #printf("start p0\n");
   0x0000000000400949 <+9>: callq  0x4006b0 <puts@plt> #找到 puts 函數地址 0x4006b0
   0x000000000040094e <+14>:    xchg   %ax,%ax        # NOP
    #atomic_store(&flag[0], 1);
   0x0000000000400950 <+16>:    movl   $0x1,0x20073e(%rip) #設定 flag rip 相對定址
                                                          # 0x601098 <flag>
   0x000000000040095a <+26>:    mfence             #內存屏障 atomic_store
    # atomic_thread_fence(memory_order_seq_cst);
   0x000000000040095d <+29>:    mfence             #內存屏障 atomic_thread_fence
    #atomic_store(&turn, 1);
   0x0000000000400960 <+32>:    movl   $0x1,0x200736(%rip) #設定 turn rip 相對定址
                                                           # 0x6010a0 <turn>
   0x000000000040096a <+42>:    mfence             #內存屏障 atomic_store
   0x000000000040096d <+45>:    jmp    0x40097b <p0+59> #無條件跳轉
   0x000000000040096f <+47>:    nop    #不改變任何程式可存取的暫存器

    #while (atomic_load(&flag[1]) && atomic_load(&turn)==1)
   0x0000000000400970 <+48>:    mov    0x20072a(%rip),%eax  #把 turn 讀出來
                                                           #0x6010a0 <turn>
   0x0000000000400976 <+54>:    cmp    $0x1,%eax          #提早做 compare 因為是效能瓶頸
   0x0000000000400979 <+57>:    jne    0x400985 <p0+69>   #符合條件時跳轉
   0x000000000040097b <+59>:    mov    0x20071b(%rip),%eax # flag[1]# 0x60109c <flag+4>
   0x0000000000400981 <+65>:    test   %eax,%eax            #&&運算
   0x0000000000400983 <+67>:    jne    0x400970 <p0+48>     #符合條件時跳轉
   0x0000000000400985 <+69>:    movl   $0x0,0x200709(%rip)#(&flag[0],0)
                                                          #0x601098 <flag>
   0x000000000040098f <+79>:    mfence                  #內存屏障 atomic_store
   0x0000000000400992 <+82>:    jmp    0x400950 <p0+16>  #無條件跳轉回到 while 上面
End of assembler dump.
```

(2) gdb ./peterson_trival-g

#每一行意思

#使 peterson_trival-g 錯的原因

```
>>> disassemble p0
Dump of assembler code for function p0:
   0x000000000040087b <+0>: push   %rbp                #stack frame
   0x000000000040087c <+1>: mov    %rsp,%rbp           #rsp =rbp
   0x000000000040087f <+4>: mov    $0x400af0,%edi      #用 edi 保存地址 0x400b50
   0x0000000000400884 <+9>: callq  0x400690 <puts@plt> #printf("start p0\n");
   0x0000000000400889 <+14>:    movl   $0x1,0x200801(%rip)#設定 flag0# 0x601094 <flag0>
   0x0000000000400893 <+24>:    movl   $0x1,0x2007ef(%rip) #設定 turn# 0x60108c <turn>
   0x000000000040089d <+34>:    nop     #不改變任何程式可存取的暫存器
##while (flag1==1 && turn==1) 不見了
   0x000000000040089e <+35>:    mov    0x2007ec(%rip),%eax#拿 flag1 #0x601090 <flag1>
   0x00000000004008a4 <+41>:    cmp    $0x1,%eax     #比對
   0x00000000004008a7 <+44>:    jne    0x4008b4 <p0+57>  #符合條件時跳轉
   0x00000000004008a9 <+46>:    mov    0x2007dd(%rip),%eax #turn = 1  # 0x60108c <turn>
   0x00000000004008af <+52>:    cmp    $0x1,%eax     #比對
   0x00000000004008b2 <+55>:    je     0x40089e <p0+35>  #符合條件時跳轉
   0x00000000004008b4 <+57>:    movl   $0x0,0x2007d6(%rip) #flag0=0 # 0x601094 <flag0>
   0x00000000004008be <+67>:    jmp    0x400889 <p0+14>    #無條件跳轉回到 while 上面
End of assembler dump.
```

// youtube 筆記
正確的 P0

```
1. atomic_int turn=0;   //要用 atomic_int 才可傳入 atomic_store
2. atomic_int flag[2] = {0, 0};
3. void p0(void) {
4.     printf("start p0\n");
5.     while (1) {
6.         atomic_store(&flag[0], 1); //address type: atomic_int
7.         atomic_thread_fence(memory_order_seq_cst);//避免第 6 和
                                                    // 第 8 對調
8.         atomic_store(&turn, 1);
9.         while (atomic_load(&flag[1]) && atomic_load(&turn)==1)
10.            ;   //waiting
11.        //底下程式碼用於模擬在 critical section
12.        in_cs++; //進入 critical section++
13.        nanosleep(&ts, NULL);
14.        if (in_cs == 2) fprintf(stderr, "p0 及 p1 都在 critical section\n");
15.        p0_in_cs++;
16.        nanosleep(&ts, NULL);
17.        in_cs--;//出去 critical section++
18.        //離開 critical section
19.        atomic_store(&flag[0], 0); //把 0 存到 flag
20.    }
21.}
```

```
gcc -pthread -O3 peterson_correct.c -o peterson_correct-O3 ##-O3 優化
gcc -pthread -g  peterson_correct.c -o peterson_correct-g ##-g =-O0 不優化
gcc -pthread -O3 peterson_trival.c -o peterson_trival-O3
gcc -pthread -g  peterson_trival.c -o peterson_trival-g
```