

1. critical section

中文譯作為臨界區段，指設計一個能夠存取共用資源的協議，有以下的特性：

每個 process 有一段程式稱為 critical section，而它們執行時必須是 mutually exclusive (互斥獨立)，也就是當一個 process 在執行 critical section 裡的程式內容時，其他 process 只能等待。

它的流程是：

```
flag[i] = TRUE;  
turn = j;  
while (flag[j] && turn == j);  
  
critical section  
  
flag[i] = FALSE;  
  
remainder section
```

1. Permission request: 將進入 critical section 的通道 block 起來，以免其他 process 進入
2. Critical section: 執行 critical section 的程式內容
3. Exit notification: 解除 critical section 通道的 block
4. Remainder section: 執行 remainder section 的程式內容

2. critical section requirement

要解決 critical section 必須要滿足以下三個條件

1. mutual exclusion
2. process: 程式要持續有進展，process 在非 remainder section 時可以決定誰進入 critical section，一個 process 不能阻擋其他 process 進入 critical section
3. bounded waiting: 若一個 process 要執行 critical section，其他等待的 processes 數量必須是有限的，這會使 starvation 不會發生

3. Peterson's Solution

peterson's solution 是一個利用軟體解決 critical section problem 的方法

以下程式碼擷取自維基百科
(<https://zh.wikipedia.org/wiki/Peterson%E7%AE%97%E6%B3%95>)

```
//flag[] is boolean array; and turn is an integer
flag[0] = false;
flag[1] = false;
int turn;

P0: flag[0] = true;
turn = 1;
while (flag[1] == true && turn == 1)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[0] = false;

P1: flag[1] = true;
turn = 0;
while (flag[0] == true && turn == 0)
{
    // busy wait
}
// critical section
...
// end of critical section
flag[1] = false;
```

利用一個布林陣列 flag 紀錄哪一個 process 有意願進入 critical section
利用一個整數變數 turn 紀錄哪一個 process 能夠進入 critical section, turn = i , j 這個指令會有先後順序, 所以一定能決定一個 process 先執行

此方法符合上面提過需要滿足的三個條件

1. mutual exclusion: 同一時間, turn 只會有一個值, 也就是只有一個 process 能進入 critical section
2. process: 若 process i 不想執行, 那其 flag[i] = false, 不會影響到 process j 進入 critical section
3. 當 process i 執行完 critical section 後, 回到開始會將 turn = j, 使得 process j 能夠進入 critical section (如果 process j 有意願), 如此就不會發生一個process 一直執行而使其他 process 發生 starvation 的狀況

4. Peterson's Solution 的問題

按照以下三行程式當作例子說明：

```
flag[ i ] = true;
turn = j;
while ( flag[ j ] && turn == j );
```

將他轉為 machine language 會變成：

```
store true, i ( flag )
store turn, j
load value 1, j ( flag )
load value 2, j
```

若是按照 in order 的順序執行自然沒有問題，但是 CPU 將之做優化後會變成以下，無 dependency 的 instructions 可以往前移

```
load value 1, j ( flag )
store true, i ( flag )
store turn, j
load value 2, j
```

這會造成程式不是按照 in order 的順序執行，也就是 $\text{flag}[j]$ 先讀入，這是錯誤的，正確來講，我們必須確保 $\text{flag}[i] = \text{true}$ 與 $\text{turn} = j$ 這兩行先做完，才能做 $\text{while}(\text{flag}[j] \&& \text{turn} == j)$

將以上做統整，就是現代硬體會將 instructions 做最佳化，也就是會移動 no dependency 的 instructions 的位置，這會導致原本必須按照順序順序執行的 Peterson's Solution 變成錯誤的執行順序，進而導致錯誤的結果。

5. Peterson's Solution 問題的解決方法

首先先解釋 Memory Barriers，Memory Barriers 會使 instructions 在執行時，按照特定（此指令所在的位置）的順序執行，例如：

instruction 1 instruction 2 instruction 3 ----- Memory Barriers ----- instruction 4 instruction 5 instruction 6	Memory Barriers 之後的三道指令一定要在前三道指令執行完才能執行，最佳化時也不會將 Memory Barriers 前後的指令做跨越移動
---	---

以 C++ 的指令來說，在 while 之前加上 `pthread_barrier_wait(& barrier_start);`，並在執行完 critical section 後加上 `pthread_barrier_wait (& barrier_end);`，這樣就可以使裡面的內容全部按照正確的順序執行

```
flag[ i ] = true;
turn = j;
pthread_barrier_wait (& barrier_start );
while ( flag[ j ] && turn == j ) ;
    critical section();
pthread_barrier_wait (& barrier_end );
    remainder section();
```