**OShw2 報告**

學號：1083339　姓名：楊侑承

**什麼是 CS？**

Critical section，指的是 process 中存取與其他 process 共享 resource 的程式片段。

**什麼是 CS 問題？**

CS 問題，是指有些共享 resource 有無法同時被多個 process 存取的特性，為了解決這些共享 resource 的同步問題，因此對 process 的 CS 設計一個對存取共享 resource 的權限。

```
1  //Ex code
2  //c是P0,P1的共享resource
3  //c=0
4  //執行c=c+1 再執行c=c-1 最後c應該等於0
5
6  P0 // c=c+1
7  {
8      load r5 , c;
9      addi r5, r5, 1;
10     store r5, c;
11 }
12 P1 // c=c-1
13 {
14     load r6 , c;
15     addi r6, r6, -1;
16     store r6, c;
17 }
```

```
18 //instruction 可能的堆疊
19 load r5, c; // r5=0
20 load r6, c; // r6=0
21 addi r5, r5, 1; // r5=1
22 addi r6, r6, -1; // r6=-1
23 store r5, c; // c=1
24 store r6, c; // c=-1
25
```

**CS 問題解法的必要條件**

1. 排他性
2. 只有在 CS 中的 process 能阻止其他 process 進入
3. 有限的等待

**Peterson's Solution**

它提供了一個好的方式去描述 CS 問題應該如何解決的邏輯，並說明處理 CS 問題時的三個必要條件該如何處理。這解法僅限於 2 個 process 的 CS 問題處理。

```
43  //Ex code
44  bool flag[2] = { false };
45  int turn;
46
47  while(1) // process 0 (i = 0, j = 1)
48  {
49      flag[i] = true;
50      turn = j;
51      while (flag[j] && turn == j);
52      /*cs*/
53      flag[i] = false;
54      /*rs*/
55  }
56
57  while(1) // process 1 (i = 1, j = 0)
58  {
59      turn = j;
60      flag[i] = true;
61      while (flag[j] && turn == j);
62      /*cs*/
63      flag[i] = false;
64      /*rs*/
65  }
66
```

**Peterson's Solution 的問題**

如果兩個 process 的 turn 跟 flag 順序寫不一樣時 會卡死

```
26
27  //Ex code
28  //Pi是當前的process Pj是同時執行的另外一個process
29   bool flag[2] = { false };
30   int turn;
31
32  while (1)
33  {
34      flag[i] = true;
35      turn = j;
36      while (flag[j] && turn == j);
37      /*cs*/
38      flag[i] = false;
39      /*rs*/
40  }
41
```

```
67  //instruction 可能的堆疊
68  flag[0] = true; // P0
69  turn = 0; // P1
70  turn = 1; // P0
71  flag[1] = true; // P1
72  while (flag[1] && turn == 1); // P0 無限loop
73  while (flag[0] && turn == 0);
74      //下略
```

而在現代計算機組織中，硬體設計所依據的框架是 Tomasulo's algorithm。它在硬體時會優先處理已經 ready 的 instruction，因此它的運算順序是 out of order，所以會導致 Peterson's Solution 的執行順序並非原本在理論上的順序。

## 解決方案

軟體的方案因為硬體設計而無法被正確執行，硬體的方案則是過於複雜而且高階語言難以利用。然而 OS 工程師可以使用軟硬體結合來簡化硬體方案複雜的關鍵方案，例如 Mutex Locks 與 Semaphores。但這兩套方案在不當的混合使用以處裡 CS 問題時，容易發生許多錯誤。因此，現行的 OS 偏向使用一個高階的同步結構，monitor。

Monitor 由三個部分構成，共享 resource 宣告區、operations 區、初始化區。並且在 monitor 中，定義了一個新的型態，通常叫 condition。這個型態中定義對共享 resource 執行存取時一個可靠的 CS 問題處理方案。此外 monitor 還保障在 monitor 中的共享 resource 與 operations 的互斥性質，並且 monitor 在同一時間內只為一個 process 提供服務。

```
75
76  class Monitor
77  {
78  public:
79      class condition
80      {
81      public:
82          condition() { S = 1; };
83          condition(int n) { S = n; };
84          ~condition() {};
85          wait() {
86              S--;
87              if (S < 0)
88              {
89                  Queue.push(this process);
90                  sleep();
91              }
92          };
93          signal() {
94              S++;
95              if (S->value <= 0) {
96                  wakeup(Queue.pop());
97              }
98          };
99      private:
100         int S;
101         Queue<process> list;
102     };
103     Monitor() {};
104     ~Monitor() {};
105     virtual init();
106 private:
107
108 };
109
```

```
110 Monitor monitor_name
111 {
112     // 共享resource宣告
113
114     // operations
115     pro1() { ... }
116     pro2() { ... }
117
118     // 初始化
119     init() { ... }
120 };
121
```