作業系統概論HW2

1081546 萬彥君

Critical Section problem(簡稱CS)

滿足CS問題需要以下三個條件

1. mutual exclusion:如果一個process在其CS內執行，那麼其他程序都不能在其CS內執行

2. progress:如果沒有process在其CS內執行且有程序需進入CS ，那麼只有不在Remainder section(RS)內執行的process可參加選擇，以確定誰能下一個進入CS ，且這種選擇不能無限推遲

3. bounded waiting:從一個程序做出進入CS的請求，直到該請求允許為止，其他process允許進入其CS內的次數有上限

Peterson's Solution

Peterson演算法是一種經典的基於軟體的CS問題演算法，可能現代計算機體系架構基本機器語言有些不同，不能確保正確執行。

Peterson演算法適用於兩個Process在CS與RS間交替執行,為了方便，當使用P1時，P0來標示另一個Process。Peterson演算法需要在兩個程序之間共享兩個資料項：

分別為 **int** turn 跟**bool** flag[2]

變數turn表示一個Process可以進入其CS ，即如果turn == 1，則允許Process在其CS執行。 flag表示該Process是否有意願想要進入CS ，如果flag [1]為true，即表示P1想進入其CS 。

以下為P0的程式部分

```
19      //p0
20      void procedure0()
21      {
22          do{
23              flag[0] = true;
24              turn = 1;
25              while (flag[1] && turn == 1){}/*若flag[1]為false，P0就進入cs；若flag[1]為turePO循
26                                              環等待，只要P1退出cs，P0即可進入*/
27              visit();/*訪問cs*/
28              flag[0] = false;/*訪問cs完成，procedure0釋放出cs*/
29
30              /*remainder section*/
31          } while (true);
32      }
```

以下為P1的程式部分

```
33      //p1
34      void procedure1()
35      {
36          do{
37              flag[1] = true;
38              turn = 0;
39              while (flag[0] && turn == 0){}/*若flag[1]為false，P1就進入cs；若flag[0]為turePO循
40                                            環等待，只要P0退出cs，P1即可進入*/
41              visit();/*訪問cs*/
42              flag[1] = false;/*訪問cs完成，procedure1釋放出cs*/
43
44              /*remainder section*/
45          } while (true);
46      }
```

證明Peterson‘s solution可以滿足mutual exclusion 及 progress和bounded waiting:

證明mutual exclusion: P0和P1不可能同時進入他們的CS。

證明 progress: 在Peterson's solution中改良為flag[i]，turn兩個變數去控制，如果P0想執行但P1不想執行，(flag[0]=true但flag[1]=false的情況)，即便是輪到P1執行(turn=1)，也能夠因為P1不想執行而使P0順利進入它的CS。因為擋住P0進入它的CS的條件是while(flag[1] && turn==1);，因此只要flag[1]=false，P0就能進入它的CS。

證明 bounded waiting:因為P0進CS前做了flag[0]=TRUE;，turn = 1，做的是「我想執行了，但先讓給對方執行」，因此對兩支process來說，都不會無限插隊。

然而，Peterson這個解法在現代硬體在優化執行流程 時，可能會出問題。
原因在於現代的編譯器和多核CPU因為優化程式的原因，最擅長的事情就是指令亂序執行。Compiler做的是靜態亂序優化，CPU做的是動態亂序優化。簡單來說，就是指令最終在CPU的執行順序和我們在程序中寫的順序可能是大相徑庭的。當然這種亂序執行是要在保證最終執行結果正確的前提下的，大多數情況下都不會引起問題，我們對指令的亂序執行也毫無感知。但是在一些特殊的情況下，比如Peterson演算法裡，亂序優化可能會引起問題。通常情況下，亂序優化都可以把對不同地址的load操作提到store之前去，我想這是因為load操作如果cache命中的話，要比store快很多。以Process0為例，看這3行。

```
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1){}
```

前兩行是store，第三行是load。但是對同一變量turn的store再load，亂序優化是不可能對他們交換順序的。但是flag[0]和flag[1]是不同的變量，先store後load就可能被亂序優化成先load flag[1]，再store flag[0]。假設兩個線程都已退出CS，準備再次進入，此時flag[0]和flag[1]都是false。按亂序執行先load，兩個Process都會有while條件為假，則同時都可以進入了CS ， mutual exclusion失效！這就是在有些情況下要保持程式的順序一致性的重要。

這個問題怎麼解決呢？就是使用Memory Barrier。顧名思義，它就像個屏障一樣擺在兩段code之間，阻止Compiler或者CPU在這兩段code之間進行亂序優化。Memory Barrier 有兩種類型，Compiler Barrier 以及 CPU Barrier。當程式編譯時，Compiler會對生成的可執行程式碼做一定最佳化，造成亂序執行或者甚至不執行。

以下為memory barrier 的Compiler barrier

1.gcc Compiler在遇到內嵌組譯語句：asm volatile("" ::: "memory");

也可以用 gcc 內建的函式__sync_synchronize();來呼叫

2.Microsoft Visual C++：_ReadWriteBarrier() MemoryBarrier()

3.Intel C++：__memory_barrier()

以下為memory barrier 的CPU barrier

1.x86 CPU Barrier

lfence (asm), void _mm_lfence(void): load barrier(可讀)

sfence (asm), void _mm_sfence(void):store barrier(可寫)

mfence (asm), void _mm_mfence(void):load and store barrier(可讀可寫)

2.ARMv7 CPU Barrier

dmb (asm) :dmb(data memroy barrier) 保證在dmb之前的memory訪問指令在它之後的memory訪問指令之前完成，也就是阻止了亂序。

dsb (asm) :dsb(date synchronization barrier) 更嚴格一些，保證在dsb完成之前，所有它之前的指令都執行完成。

isb (asm) :isb(instruction synchronization barrier) 最嚴格，它會清空Processor的流水線，當然就能保證之前的所有指令執行完，它之後的指令必須從cache或memory獲取。