

簡介thread與CPU相關知識

中正大學，作業系統實驗室

羅習五 陽春副教授

shiwulo@gmail.com



課程單元

🍏 Multi-thread與計算機硬體

- ✿ volatile
- ✿ _Alignas (C11 & C++11)
- ✿ atomic operations (C11 & C++11)
- ✿ memory order (C11 and C++11)
- ✿ Intel VTune vs. Linux perf
- ✿ 實驗: memoryTest/... (pingpong、aligned、atomic、mutex、semaphore、spinlock)

🍏 基本lock機制

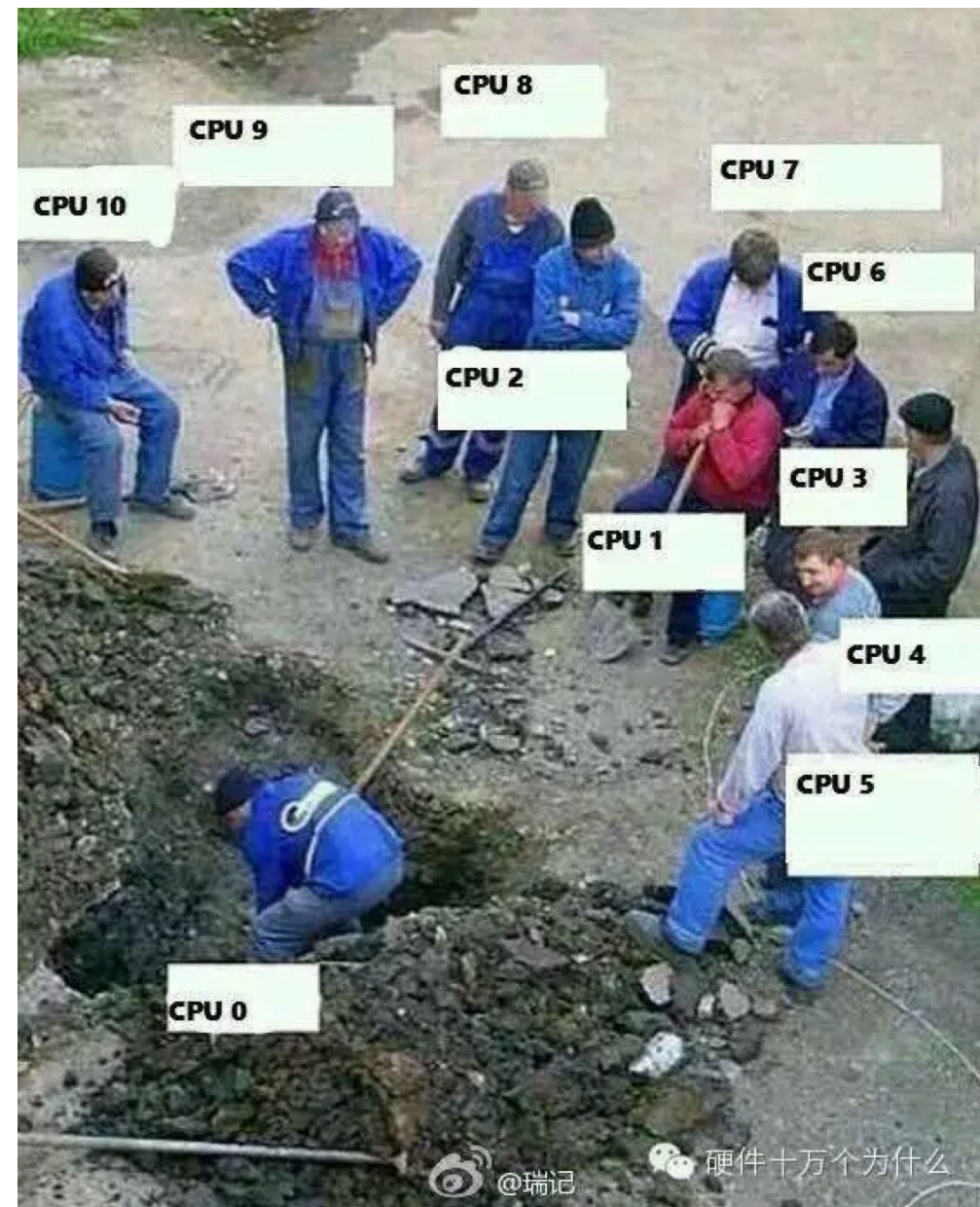
- ✿ semaphore
- ✿ mutex (可支援優先權控制、遞迴鎖、自適應鎖)
- ✿ spinlock
- ✿ pthread_rwlock
- ✿ 實驗: concurrentQ/

🍏 進階的lock機制

- ✿ lock-free queue
- ✿ sequential lock
- ✿ ticket lock
- ✿ r/w spinlock

🍏 Memory ordering

學習目的



安裝必要的軟體

必須安裝

♣ sudo apt install perf /*觀察CPU狀態的工具*/

♣ sudo apt-get install manpages-posix manpages-posix-dev /*pthread文件*/

建議安裝

♣ Intel C Compiler & tools

🍇 ftp://lonux.cs.ccu.edu.tw/parallel_studio_xe_2018_update3_cluster_edition.tgz

♣ KDE debugger

🍇 sudo apt install kdbg

♣ Microsoft VS code

🍇 ftp://lonux.cs.ccu.edu.tw/code_1.23.1-1525968403_amd64.deb

安裝 “perf ”, Ubuntu 18.04為例

```
$sudo apt install linux-tools-common
```

```
$sudo apt install linux-tools-4.15.0-22-generic linux-tools-generic
```

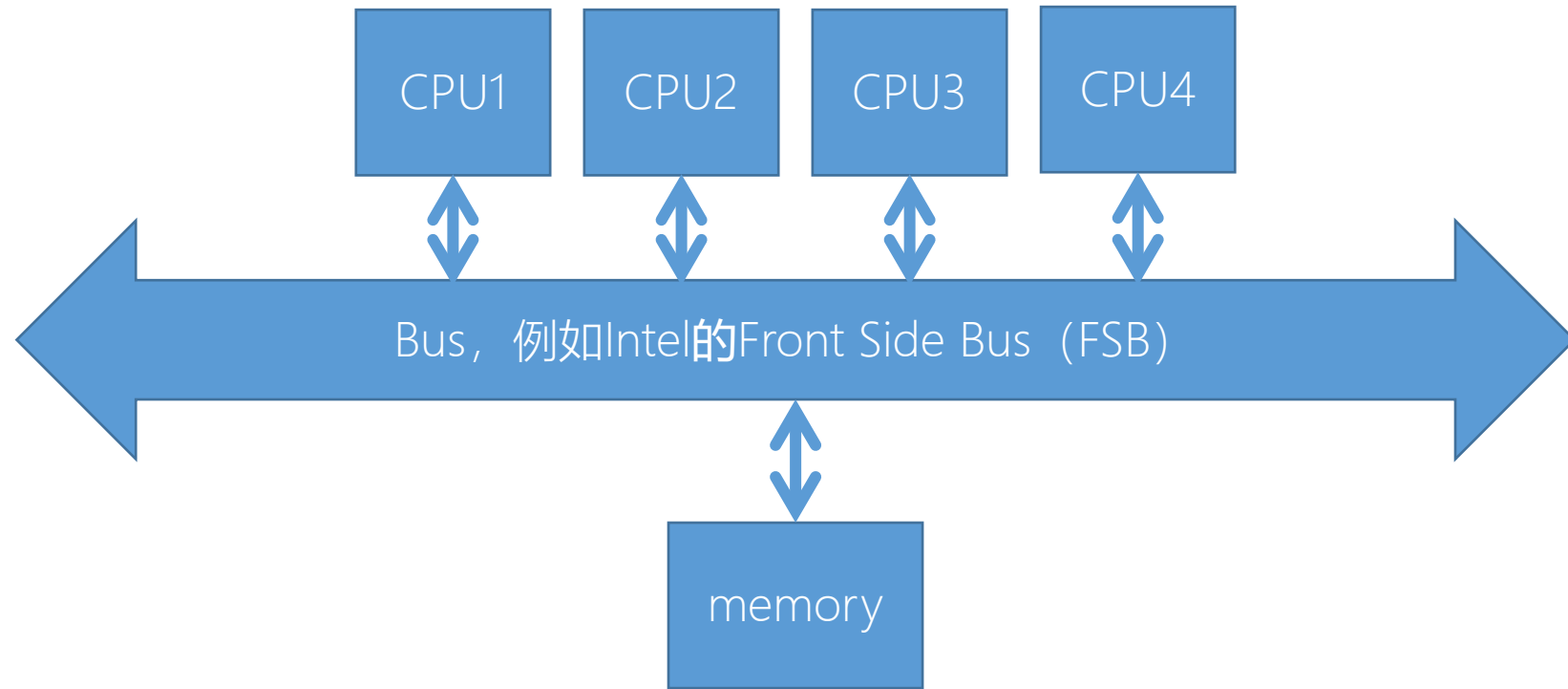
在撰寫multi-thread程式之前 選用「編譯器」、啟動最佳化

```
$gcc table.c                                /*table.c是一個簡單的表格加總*/  
user:      81.080010s                      /*gcc w/o optimization*/  
sys:       0.640063s  
$gcc -O3 table.c  
user:      4.794872s                        /*gcc with -O3*/  
sys:       0.579863s  
$icc -O3 table.c  
user:      0.358683s                        /*intel C compiler*/  
sys:       0.577878s
```



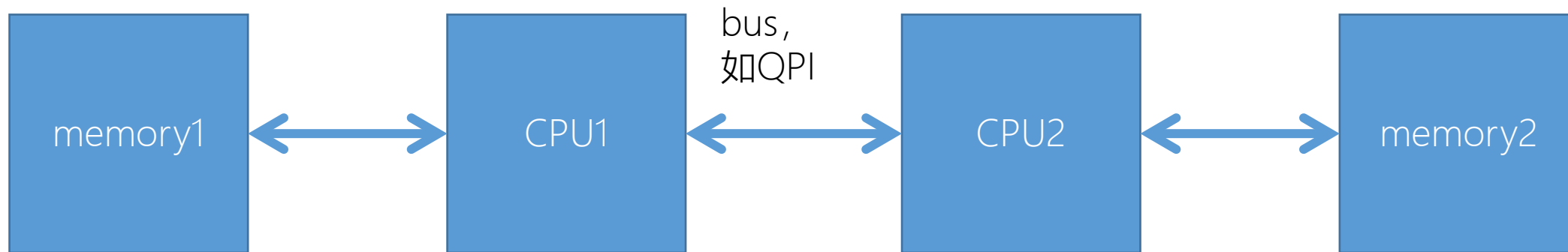

Multi-thread與計算機硬體

NUMA與UMA架構

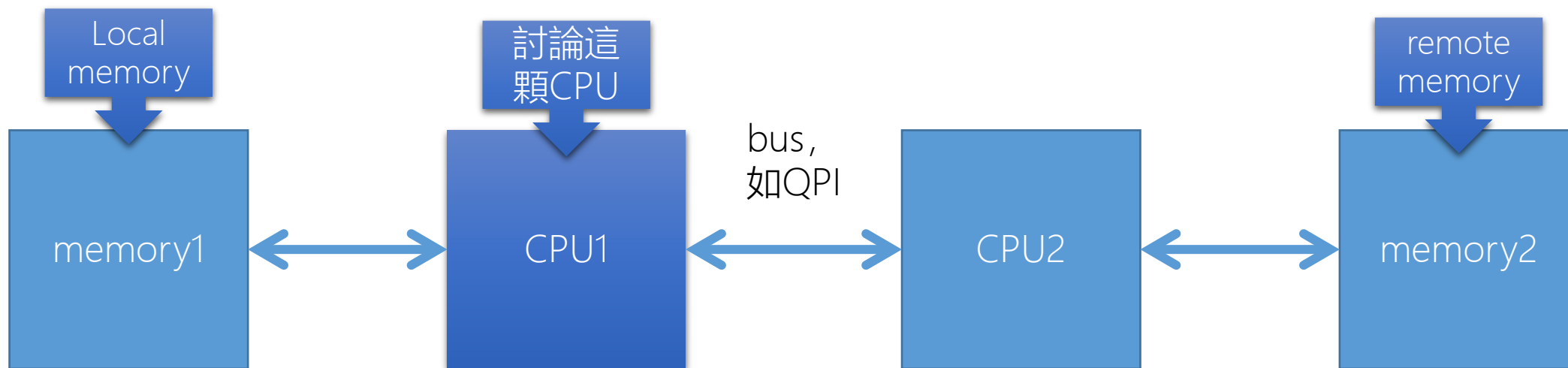


所有處理器都使用同一個記憶體，對程式設計師而言，記憶體的架構非常容易了解，但缺點是記憶體頻寬有限，不適用於「非常多」處理器

NUMA與UMA架構



NUMA與UMA架構



NUMA與UMA架構

latency

```
Measuring idle latencies (in ns)...
```

Socket	Memory node 0	Memory node 1
0	67.5	125.2
1	126.5	68.5

bandwidth

```
Measuring Memory Bandwidths between nodes within system  
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)  
Using Read-only traffic type
```

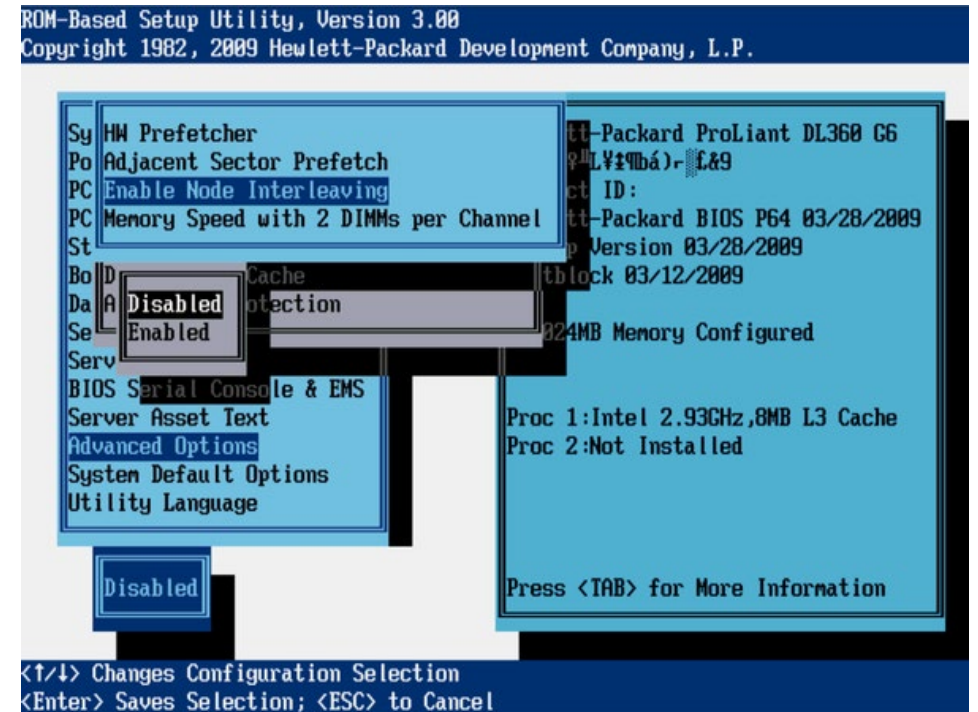
Socket	Memory node 0	Memory node 1
0	38477.8	19373.1
1	19510.3	38610.4

以Intel為例，存取local memory的latency為remote memory的1/2，bandwidth則為remote的2倍。程式設計師必須妥善的規劃記憶體存取方式。因為remote和local memory的頻寬、延遲不一樣。

<https://software.intel.com/en-us/articles/intelr-memory-latency-checker>

NUMA的進階考量

- 🍏 通常NUMA的機器在BIOS的地方可以設定是否啟動interleaving
- 🍏 啟動的好處：
 - 🍀 使用起來就跟UMA機器的感覺一模一樣
 - 🍀 任意程式就算沒有對NUMA進行優化，也可以存取全部的頻寬
- 🍏 不啟動的好處
 - 🍀 對NUMA優化過的程式可以盡量的將記憶體存取集中在local memory
 - 🍀 作業系統、應用程式都可以針對NUMA進行優化 (Linux支援NUMA)

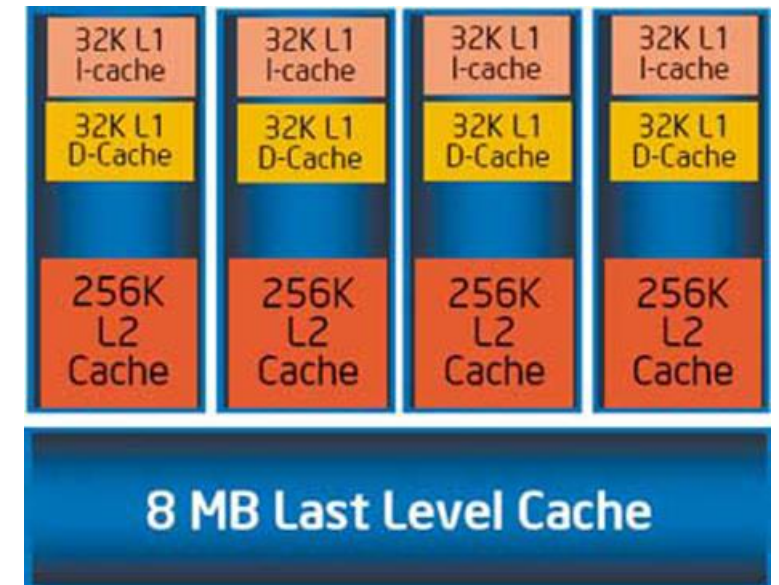


SMP與UMA

- 🍏 SMP是Symmetric multiprocessing的縮寫，指的是每一個處理器的功能都是一模一樣
- 🍏 UMA指的是記憶體架構，每顆處理器存取任意記憶體，其頻寬、延遲都是一樣的
- 🍏 現在大部分的小型電腦（例如：PC、部分server）是SMP、UMA架構
- 🍏 CPU大部分是SMP架構。server或workstation的記憶體架構可能是UMA或NUMA

SMP與UMA與multi-core

- 🍏 對程式設計師而言，普通的multi-core架構相當於是SMP+UMA架構。
- 🍏 目前Intel、AMD、ARM等公司，幾乎都讓同一個封裝上的多核心處理器共用LLC（last level cache，例如L3 cache）
- 🍏 這讓multi-core在資料傳遞上變得比傳統的SMP要來得快速（因為可以透過讀寫LLC傳遞資料）





CPU設計與multithread的影響

pipeline & superscalar

- 右手邊的圖是Intel的CPU方塊圖，對程式設計師而言要注意的是「Out-of-Order execution」
- 因為out-of-order execution因此下列程式碼對CPU而言都是正確的執行方式

範例一：

A=1;

B=2;

範例二：

B=2;

A=1;

- 如果程式碼有相依性，那麼CPU會保證執行的順序。例如：

範例一：

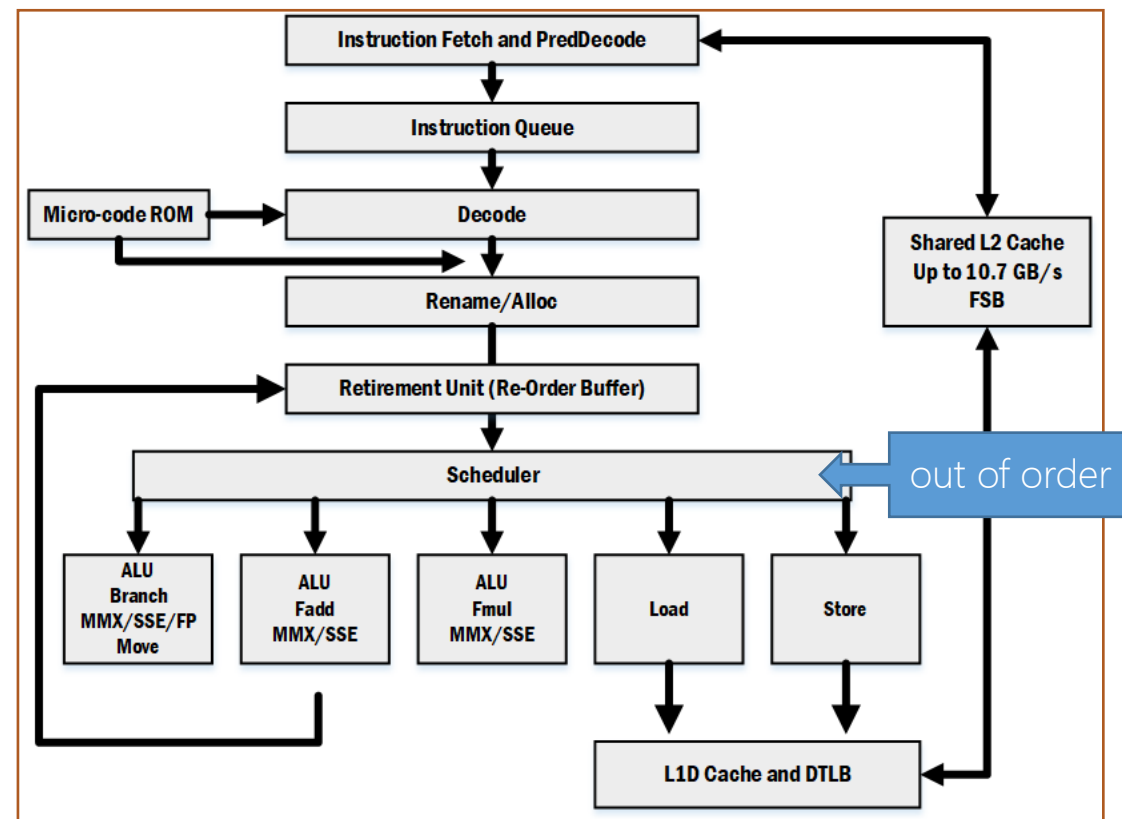
A=*alpha;

B=A+1;

範例二：

A=*alpha;

```
if (A==1) {  
    /*會在A==1之後*/  
}
```



pipeline & superscalar

如果compiler打開優化（例如：O3），那麼compiler可能也會將指令重排，特別是沒有前後相關的指令，下列二個範例對編譯器而言都是一樣的

範例一： 範例二：

A=1; B=2;

B=2; A=1;

🍏 在multi-threading的情況下，如果一定要保證順序，可能需要使用memory barrier（也稱之為fence後面會介紹）

補充：x86的組語中的memory barrier

- 🍏 `lfence (asm), void _mm_lfence (void)` ; 讀不能越過lfence
- 🍏 `sfence (asm), void _mm_sfence (void)` ; 寫不能越過sfence
- 🍏 `mfence (asm), void _mm_mfence (void)` ; 讀寫都不能越過mfence

驗證：memoryModel.c

```
1.  #include <stdio.h>
2.  int main(int argc, char** argv) {
3.      volatile int a;
4.      int b;
5.      /*反組譯以後會發現b和a的設定順序顛倒*/
6.      /*編譯指令icc -O3 memoryModel.c, O3叫編譯器進行程式碼優化*/
7.      /*使用icc或gcc都是同樣的結果, 請看下一頁分析*/
8.      b = 0xdead;
9.      a = 0xc0fe;
10.     printf("a = %x, b = %x\n", a, b);
11. }
```

驗證

memoryModel_reorder.c

```
$ gcc -O3 ? ? ? .c
/*使用gcc或icc*/
$ icc -O3 ? ? ? .c
$ gdb ./a.out
```

```
3.      volatile int a;
4.      int b;
8.      b = 0xdead;
9.      a = 0xc0fe;
```

(gdb) disassemble main

Dump of assembler code for function main:

```
0x0000000000400b10 <+0>:      push    %rbp
0x0000000000400b11 <+1>:      mov     %rsp,%rbp
0x0000000000400b14 <+4>:      and     $0xffffffffffffff80,%rsp
0x0000000000400b18 <+8>:      sub     $0x80,%rsp
0x0000000000400b1f <+15>:     xor     %esi,%esi
0x0000000000400b21 <+17>:     mov     $0x3,%edi
0x0000000000400b26 <+22>:     callq   0x400b60 <__intel_new_feature_proc_init>
0x0000000000400b2b <+27>:     stmxcsr (%rsp)
0x0000000000400b2f <+31>:     movl    $0xc0fe,0x4(%rsp)
0x0000000000400b37 <+39>:     mov     $0x401b44,%edi
0x0000000000400b3c <+44>:     orl     $0x8040,(%rsp)
0x0000000000400b43 <+51>:     mov     $0xdead,%edx
0x0000000000400b48 <+56>:     xor     %eax,%eax
0x0000000000400b4a <+58>:     mov     0x4(%rsp),%esi
0x0000000000400b4e <+62>:     ldmxcsr (%rsp)
0x0000000000400b52 <+66>:     callq   0x400970 <printf@plt>
0x0000000000400b57 <+71>:     xor     %eax,%eax
0x0000000000400b59 <+73>:     mov     %rbp,%rsp
0x0000000000400b5c <+76>:     pop     %rbp
0x0000000000400b5d <+77>:     retq
0x0000000000400b5e <+78>:     xchg    %ax,%ax
```

創作共用-姓名 標示-非商業性-相同方式分享
End of assembler dump.

指令順序對multi-thread的重要性

- 🍏 在許多multi-thread的程式中，都是透過設定變數，控制不同thread之間的交互關係
 - ♣️ 執行的前後關係
 - ♣️ 存取全域變數或全域資料結構的正確性
- 🍏 在相關的程式碼必須確認執行的順序的正確性（請參閱後面的memory ordering）
- 🍏 除了compiler會調動程式碼以外，CPU也會動態調動程式碼（OoO），因此必須使用編譯器、函數庫所提供的memory barrier確保關鍵程式碼的執行順序

pipeline & superscalar & cache

- 🍏 一道指令 (instruction) 可能會分成多個步驟進行，因此可能會有「半成品」出現
- 🍏 例如：x86允許從任意位置開始，讀取一個word，這個word可能跨過一個cache line，因此可能需要多個machine cycle才能完成
- 🍏 「0xc0fe」跨過二個cache line，因此CPU會

1. 先讀??0xc0
 2. 在讀0xfe??
 3. 將??0xc0與0xfe??組合成0xc0fe
- 在multithreading的情況下，如果要保證load、store是atomic operation，需要宣告為atomic_t

cache

word1	word2	word3	word4
			?? 0xc0
0xfe ??			

如何讓x86及x64的load & store 變成atomic operation

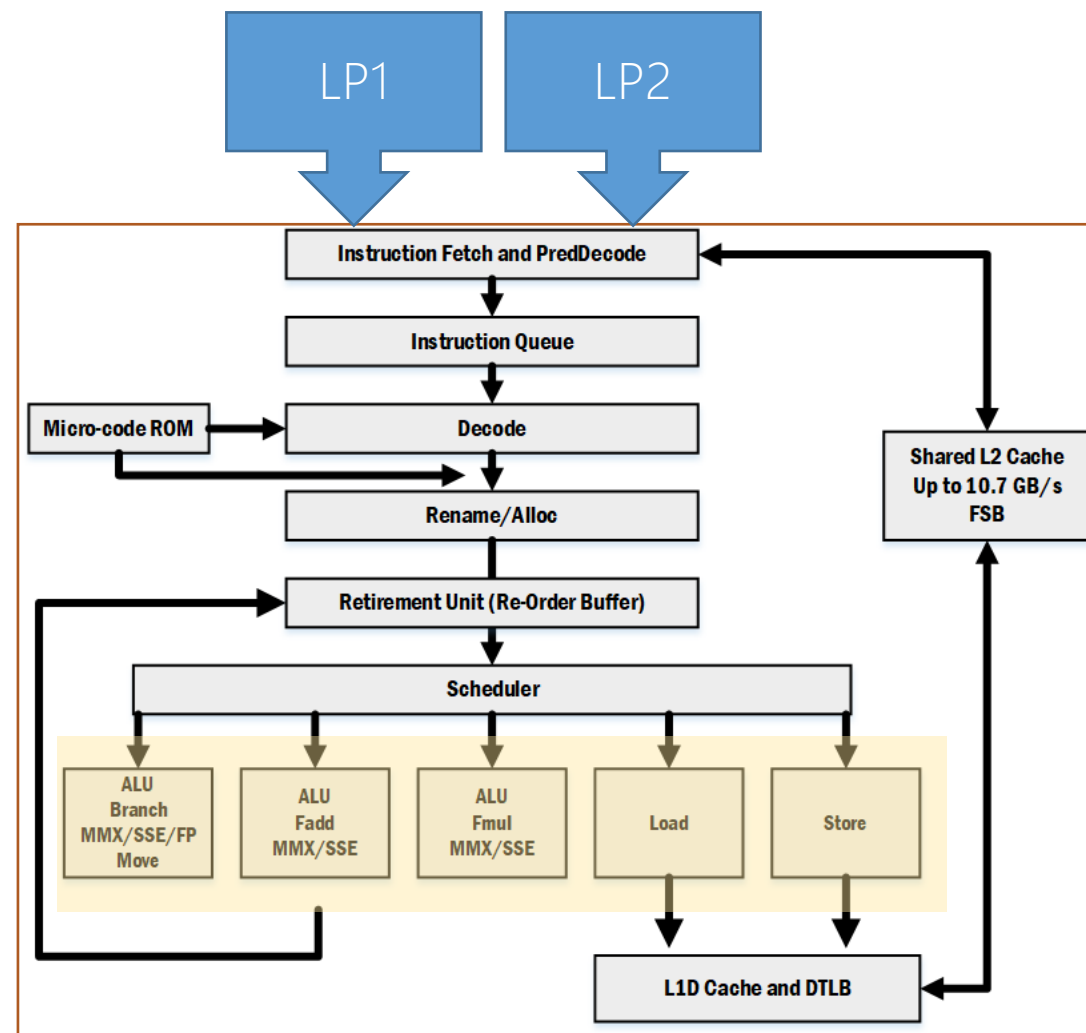
- 🍏 在Intel的文件上寫明，如果資料型別，對齊該型別的長度，那麼這個對該型別的變數的load和store是atomic operation
 - ♣ 例如：int是4 bytes，開始位置就必須是0, 4, 8, ...
 - ♣ 例如：long是8bytes，開始位置就必須是0, 8, 16, ...
 - ♣ 大部分bit的型別，不是atomic operation
- 🍏 C11和C++11可以使用**alignas**設定對齊位置

alignas & aligned_alloc

- 🍏 //僅適用於x86, 因為假設cache line為64B, int為4B
- 🍏 struct alignas(64) DS { //對cache line對齊
- 🍏 //讓a,b,c,d的load、store變成atomic operation
- 🍏 int alignas(int) a, b, c, d;
- 🍏 atomic_int spinlock;
- 🍏 };
- 🍏 //動態分配對齊記憶體 mallooc
- 🍏 void *aligned_alloc(size_t alignment, size_t size);

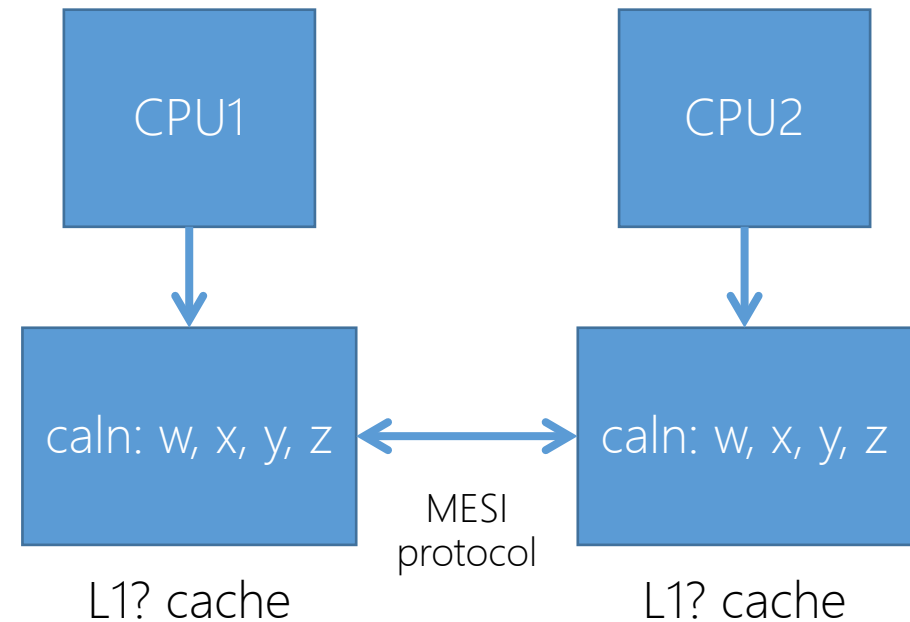
再論硬體架構

- 如右圖所示，以Intel core i7為例，一個core上可以執行2個logical processor (LP)
- 每一個LP可以執行獨立的程式（相同程式也可）
- 透過同時執行多個程式，可以讓CPU的執行單元（execution unit）的使用率提高（右圖黃色的部分），進而增加整體速度
- 如果LP1與LP2會互相**惡性**競爭資源，可能會讓速度變得更慢
 - 例如：LP1等LP2執行完成，但LP1不斷的偵測LP2是否完成工作，造成LP2一直被干擾
 - 解決方法：在spinlock中加入“**pause**”指令



記憶體的一致性 (Cache Coherence)

- 右手邊的圖假設CPU1與CPU2同時擁有存放相同變數的cache line, 稱之為caln, caln中存放w, x, y, z四個變數
- 當CPU1修改“w”, CPU會透過MESI通訊機制通知CPU2的“cache”
 - 通常CPU2的cache會將invalidate caln
 - 當CPU2要再讀取caln時, CPU2會向CPU1索取caln
 - 藉由上述方法保證資料的一致性
- **注意:**
 - 如果w已經被讀到CPU2的暫存器, 那麼MESI不會更新CPU2的暫存器
 - 因此寫程式時必須用**volatile**關鍵字, 強制CPU每次都從cache(/RAM)讀取資料



驗證

volatile.c

```
1. int main(int arg, char** argv) {  
2.     volatile int vol=0;  
3.     while(1) {  
4.         vol=vol+1;  
5.     }  
6. }
```

notvolatile.c

```
1. int main(int arg, char** argv) {  
2.     int vol=0;  
3.     while(1) {  
4.         vol=vol+1;  
5.     }  
6. }
```

驗證

```
shiwulo@NUC :~/sp/ch12$ icc -O3 volatile.c -o volatile
shiwulo@NUC :~/sp/ch12$ icc -O3 nonvolatile.c -o notvolatile
shiwulo@NUC:~/sp/ch12$ ./notvolatile &
[1] 20285
shiwulo@NUC:~/sp/ch12$ ./volatile &
[2] 20287
shiwulo@NUC:~/sp/ch12$ sudo perf top -e mem-stores
```

驗證

Samples: 51K of event '**mem-stores**', Event count (approx.): 5563314379

Overhead	Shared Object	Symbol
98.79%	volatile	[.] main
0.10%	[kernel]	[k] format_decode
0.07%	[kernel]	[k] vsnprintf

/*上面例子是perf top的執行結果，可以看到只有volatile這支程式寫入記憶體*/
/*notvolatile並沒有將資料寫入記憶體，因此在perf top中看不到notvolatile*/

```
shiwulo@NUC:~/sp/ch12$ ps -a | grep volatile
```

```
20285 pts/0    00:04:37 notvolatile
```

```
20287 pts/0    00:04:27 volatile
```

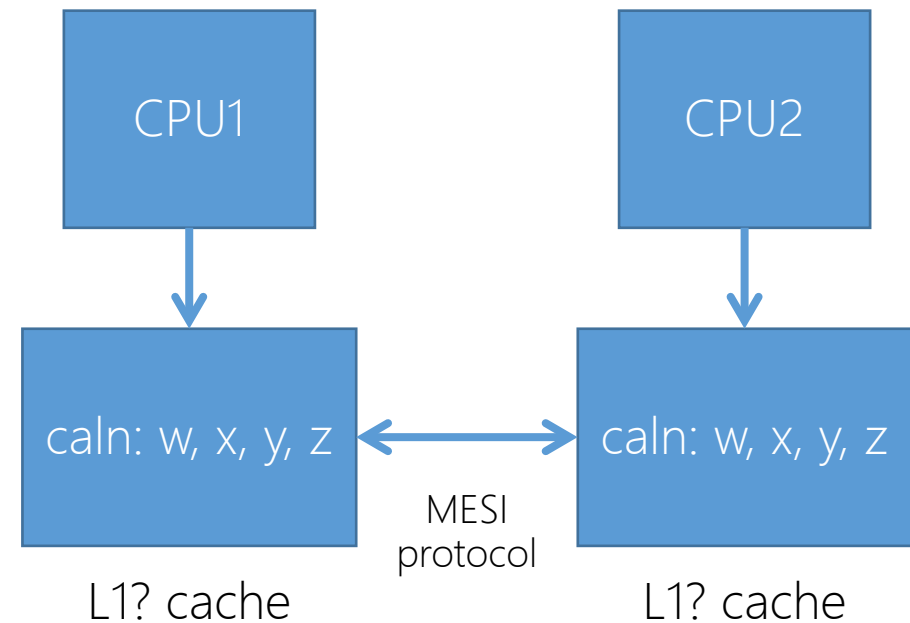
/*系統裡面nonvolatile、volatile都在執行*/

指向volatile的指標 – volatilePtr.c

```
1.  volatile int global;
2.  int main(int argv, char** argc) {
3.      volatile int* volPtr;
4.      int* ptr;
5.      volPtr = &global;
6.      ptr = &global;
7.      *volPtr = 0xc0fe;
8.      //底下這一行在某些處理器、compiler
9.      //可能不會立即更新記憶體
10.     *ptr = 0xdead;
11.     return 0;
12. }
```

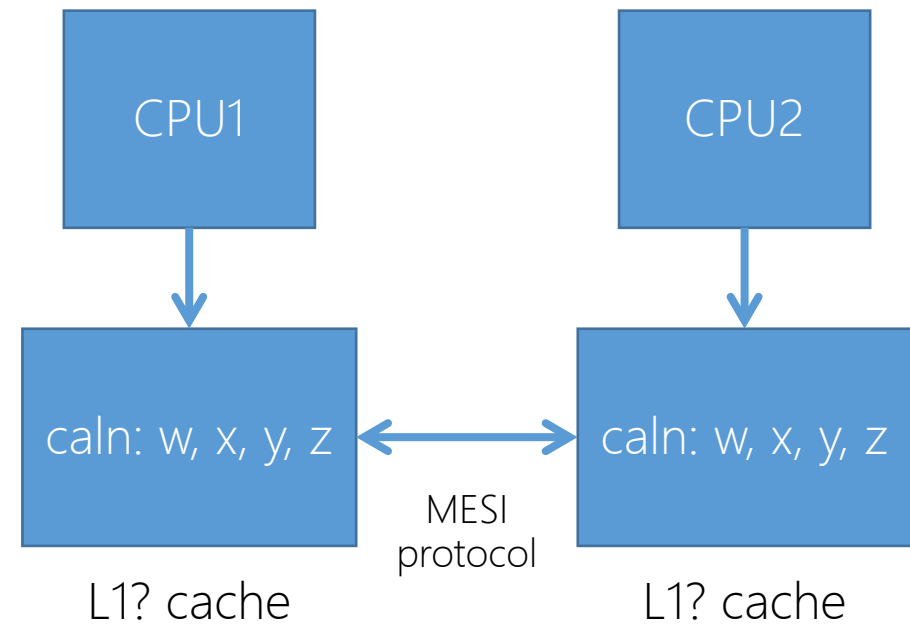

記憶體的一致性 (Cache Coherence)

- 右手邊的圖假設CPU1與CPU2同時擁有存放相同變數的cache line, 稱之為caln, caln中存放w, x, y, z四個變數
- 當CPU1修改“w”, CPU會透過MESI通訊機制通知CPU2的“cache”
 - 通常CPU2的cache會將invalidate caln (包含: w, x, y, z)
 - 當CPU2要再讀取caln時, CPU2會向CPU1索取caln
 - 藉由上述方法保證資料的一致性
- **注意:**
 - 在學理上, 如果CPU2在等CPU1修改“w”, 因此CPU2不斷的讀“w”。在MESI中並不會產生額外的traffic, 當CPU1更新“w”時, 這時才會產生一個MESI traffic



記憶體的一致性 (Cache Coherence)

- MESI (Modify, Exclusive, Shared, Invalid) 是耗時的
- 避免不必要的MESI
 - 假如CPU1需要w, CPU2需要y, 但這二個CPU並沒有用w和y做任何同步、交換資訊 (synchronization)
 - 硬體不了解軟體是否使用這些變數進行同步, 因此硬體會執行MESI以保證 cache coherence
 - 這個現象就是 **false sharing**, 會造成大量的MESI的資料交換, 拖慢CPU速度
- 解決之道: 避免將不相關的變數放在一起



使用struct代替array

- 🍏 如果只為了一個變數，就去避免false sharing會耗費掉很多的cache memory
 - 🍀 例如：int是4 bytes，如果對齊cache line，在x64上必須對齊64 byte
- 🍏 解決方法，因為將相關的變數宣告為struct，該struct再對cache line做對齊
- 🍏 例如：
 1. `struct alignas(16) DS {`
 2. `int a, b, c, d;`
 3. `atomic_int spinlock;`
 4. `};`

Intel使用的記憶體同步方法

- 🍏 Intel使用snooping記憶體同步（memory consistency）模型
- 🍏 Snooping有多種實現的方法，Intel採用以write-invalidation為基礎的MESI
- 🍏 在Intel的PMU（performance monitor unit，perf及VTune的硬體基礎）文件上，使用xsnp代表snooping
- 🍏 將MESI分成M、E、S三個事件
- 🍏 因此在Intel的處理器上可以觀察Snooping和MESI相關事件，判斷資料同步事件發生的頻率

spinlock對速度的影響

- 🍏 問題：如果A thread不斷的測試（即：讀取）B thread的某個變數，但B thread的主要執行迴圈與該變數無關，那麼A不斷的讀取是否會影響B的執行速度
- 🍏 答：根據MESI的原理，A不會影響B。實際執行也是如此。

驗證

```
1.  volatile atomic_int complete=0; //complete是二個thread共用的變數
2.  void workingThread(void) {
3.      volatile double result;
4.      struct timespec start, end;
5.      clock_gettime(CLOCK_MONOTONIC, &start);
6.      for (volatile long long j=0; j<100000; j++)
7.          for (volatile long long i=0; i<100000; i++)
8.              result += (double)i*j;
9.      clock_gettime(CLOCK_MONOTONIC, &end);
10.     long long delta = timespec2ns(end) - timespec2ns(start);
11.     atomic_store_explicit(&complete, 1, memory_order_relaxed);
12.     printf("res=%f, time = %lld.%lld\n", result, delta/1000000, delta%1000000);
13. }
14. volatile int doSleep=0;
15. void waitingThread(void) {
16.     int local=0;
17.     while (local != 1) {
18.         local=atomic_load_explicit(&complete, memory_order_seq_cst); //密集迴圈不斷的讀取complete
19.         if (doSleep==1) sleep(1); //如果doSleep設定為1, 則不會執行密集的讀取, 每秒讀取一次
20.     }
21. }
```


結果

```
$/timedetail ./readWriteShareVar
```

```
time = 26.790812018
```

經過時間:

CPU花在執行程式的時間:

CPU於usr mode執行此程式所花的時間 :

```
$/timedetail ./readWriteShareVar sleep
```

```
time = 26.123233926
```

經過時間:

CPU花在執行程式的時間:

CPU於usr mode執行此程式所花的時間 :

CPU於krl mode執行此程式所花的時間 :

26.792373282s

53.582502s

53.582502s

28.5050704s

26.124335s

26.112339s

0.011996s

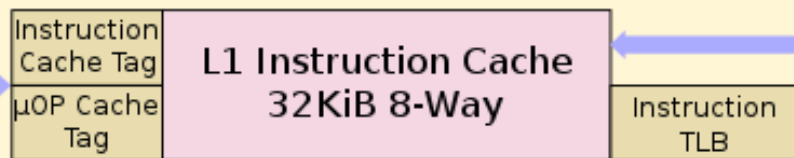
經過的時間
是一樣的

經過的時間
是一樣的

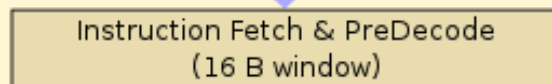
特別的同步指令 - read-modify-write

- 🍏 撰寫multithread程式時，常常需要進行同步化
 - ♣️ 例如：二個程式同時存取一個資料結構、共用一個queue
 - ♣️ 換句話說，必須避免二個程式同時修改資料結構，使得這個資料結構變成不一致的狀況
- 🍏 如果是簡單型別，可以使用`atomic_xxx`（如：`atomic_int`）
- 🍏 如果要鎖住資料結構，通常需要
 - ♣️ 檢查目前是否有人正在修改（read）
 - ♣️ 如果沒人在修改，就先「鎖上」（modify-write），然後進去修改
 - ♣️ 因此硬體必須提供特別的指令，可以同時1. 檢查 2. 檢查通過立刻上鎖
 - ♣️ POSIX提供高階的鎖定機制，如：semaphore、mutex、spinlock、rwlock
 - ♣️ C++11、C11將低階的硬體指令變成標準函數，如`atomic_fetch_add`

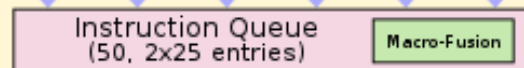
Front End



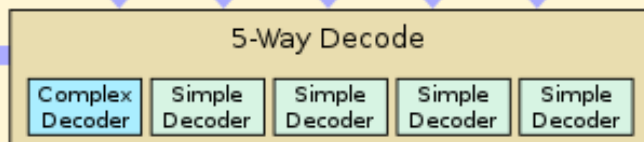
16 Bytes/cycle



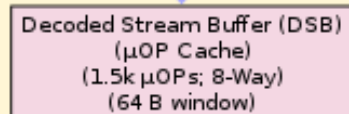
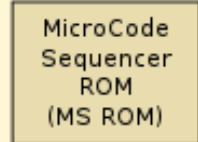
MOP MOP MOP MOP MOP MOP



MOP MOP MOP MOP MOP



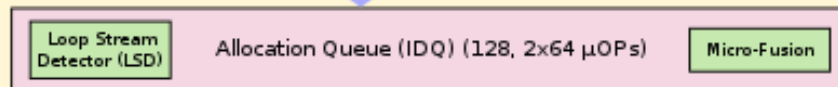
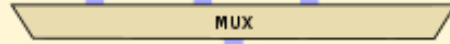
1-4 μOPs μOP μOP μOP μOP



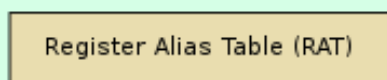
4 μOPs

6 μOPs

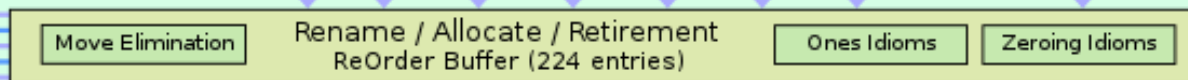
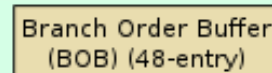
5 μOPs



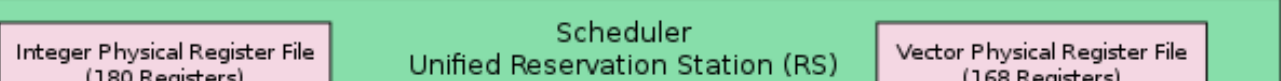
μOP μOP μOP μOP μOP μOP



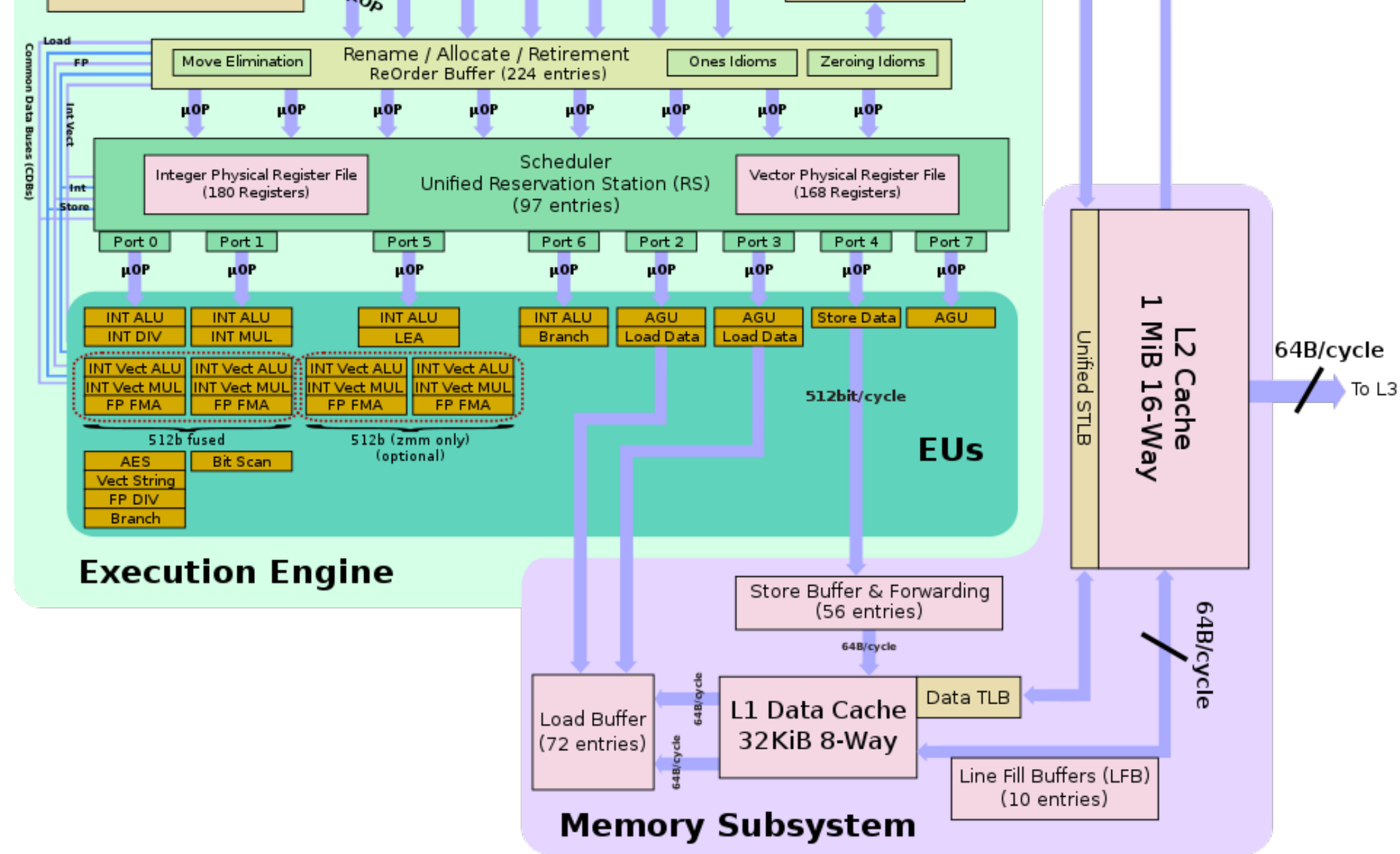
2x4 μOP



μOP μOP μOP μOP μOP μOP μOP μOP



64B/cycle



程式設計師需要特別了解的

- 🍏 cache miss (尤其是load miss) 、 (major) page fault、context switch、TLB miss、system call的數量、instruction per cycle
- 🍏 是否可以使用SIMD指令 (例如: BCD轉int)
- 🍏 是否可以減少判斷 (例如: 改成用hash) , 避免判斷句
- 🍏 如果是稀疏的table是否有其它種表示方法, 避免cache miss
- 🍏 PMU中有很多事件似乎是供給compiler工程師分析程式碼的品質, 一般工程師不需要特別注意

Memory Ordering Machine Clears

- 🍏 Intel的PMU能支援Memory Ordering Machine Clears, 事件名稱為 “`machine_clears.memory_ordering`”
- 🍏 Memory Ordering Machine Clears與false sharing有強烈的關係, 可以將它視為發生false sharing的指標

觀察異常的同步現象

- 一個寫得很好的平行化程式，每個處理器應該可以獨立做運算，只有在必要的時候，才進行資料交互換
- 因此可以觀察資料交換的頻率，以確認程式只有在必要的時候進行同步、資料交換。並且確保沒有發生 false-sharing
- 可以觀察 “`mem_load_l3_hit_retired.xsnp_hitm`” 確認資料的交換頻率，這一個事件的意義是：所有在 Level 3 cache 中完成（即：retired）的資料抓抓取的指令碼中，用 snooping（即：xsnp）抓到資料，而且這個資料是修改過的（即：hitm）
- 觀察 “`machine_clears.memory_ordering`” 確保沒有發生 false-sharing
- VTune 不支援上述二個效能指標，因此必須使用 `perf` 進行量測
- 發生上述二個情況時的間接影響是：CPI（clock per instruction）變差

使用小工具

//使用方法：`sudo perfstat` 執行檔 參數1 參數2

//如果要產生制式化的顯示，執行 `touch ~/.perflog`

//如果要產生可讀的的顯示，執行 `rm ~/.perflog`

`$sudo ./perfstat ./memoryTest/mem_pingpong`

//使用方法 `sudo perfrecord` 執行檔 參數1 參數2 ..

//輸出結果：產生檔案，檔名為 執行檔.perf.data

//觀看執行結果，執行 `sudo perf report` 執行檔.perf.data

`$ sudo ./perfrecord ./memoryTest/mem_pingpong`

`$ sudo perf report -i ./memoryTest/mem_pingpong.perf.data`

	pingpong, 二個排在一起的volatile int, 造成false sharing	將volatile變數配置到不同 L1d cache line	二個thread隨意的存取宣告為volatile的變數	存取全域變數, 使用mutex保護	存取全域變數, 使用semaphore保護	存取全域變數使用spinlock保護	二個thread, 使用atomic operation對全域變數進行存取	pxz (平行化版的XZ), CPU使用率800%	tarxz (呼叫了tar及xz二支程式), CPU使用率200%	ls	tar
cpu-cycles:u	150,798,170,668	110,244,097,468	155,093,349,925	30,714,134,115	76,596,778,660	31,519,252,934	29,899,643,321	2,218,372,719	730,791,840	49,198,594	376,873
instructions:u	120,345,103,964	120,940,803,624	80,336,910,505	12,701,091,039	15,225,342,389	4,634,718,837	2,511,251,165	1,710,737,832	1,303,051,095	89,581,443	658,724
cache-references:u	4,708,903,084	73,305,091	5,882,332,352	1,138,499,414	3,002,816,846	1,019,898,761	1,292,933,171	45,700,197	17,677,505	792,728	16,224,042
cache-misses:u	19,997,599	25,136,630	8,343,099	6,286,716	23,123,945	7,659,135	11,108,749	24,564,655	6,605,107	185,921	2,611,093
branch-instructions:u	40,003,326,481	40,156,494,045	20,063,969,025	3,313,308,433	4,709,543,228	1,812,626,890	365,426,582	228,058,971	94,784,226	22,544,429	38,196,400
branch-misses:u	4,035,751	3,503,351	2,215,276	35,870,709	90,256,805	53,672,945	2,859,445	12,465,909	3,642,617	346,031	1,512,294
bus-cycles:u	1,066,280,162	780,906,617	1,099,114,326	214,687,146	534,055,353	230,248,022	210,311,015	20,651,201	5,906,988	982,226	2,726,880
cpu-clock:u	181,534	131,653	188,407	103,846	226,666	37,665	34,834	1,000	1,001	1,000	1,000
task-clock:u	181,534	131,653	188,407	103,846	226,666	37,665	34,834	1,000	1,001	1,000	1,000
執行時間 (使用wait4測量)	24.520	16.839	23.846	15.376	32.510	4.847	2.885				
context-switches (使用wait4測量)	2	3	3	1,215	73,873	3	3				
L1-dcache-loads	20,080,762,609	20,293,573,013	20,061,862,820	3,305,884,074	4,095,703,737	1,874,044,747	357,904,831	519,992,693	319,506,413	30,417,608	166,743,987
L1-dcache-load-misses	1,040,835,772	14,993,061	1,089,323,038	293,057,596	564,336,076	255,117,385	143,752,693	24,089,386	7,789,069	504,203	5,080,518
L1-dcache-stores	20,008,290,221	20,117,477,698	20,023,044,923	2,053,751,951	2,726,905,417	1,150,434,651	238,849,777	242,302,658	104,017,068	13,357,491	49,741,268
L1-icache-load-misses	25,251,344	12,433,470	17,743,787	8,882,572	34,256,707	4,986,900	2,021,601	2,852,175	4,312,216	1,224,435	8,313,351
l2_rqsts.demand_data_rd_hit:u	2,164,676	4,201,227	1,845,406	29,948,570	5,222,095	5,658,732	160,977	4,893,813	1,678,074	112,331	1,327,178
l2_rqsts.demand_data_rd_miss	195,588,025	10,262,512	280,804,231	169,195,367	332,469,920	116,001,408	1,374,263	13,240,102	3,640,483	530,190	2,983,783
mem_inst_retired.lock_loads:u	874,049	796,934	999,528	426,514,153	642,848,645	218,787,152	194,610,271	49,308	87,404	12,410	120,741
mem_inst_retired.split_loads:u	19,736	69,777	107,332	22,317	39,562	18,134	9,252	1,756,674	1,808,718	25,984	280,355
mem_inst_retired.split_stores:u	35,396	42,140	222,116	17,346	56,430	4,001	6,416	5,717	32,852	61,994	43,224
mem_load_l3_hit_retired.xsnp_hit:u	24,567	15,296	31,234	74,517,828	64,196,125	36,184,632	4,640	1,462	6,691	159	8,386
mem_load_l3_hit_retired.xsnp_hitm:u	89,303	17,735	75,011	42,746,571	230,451,349	73,700,570	193,514,277	5,736	14,475	79	18,490
mem_load_l3_hit_retired.xsnp_miss:u	33,860	17,264	16,868	5,272,555	32,311,502	12,030,370	1,862	2,696	20,210	97	24,004
mem_load_l3_hit_retired.xsnp_none:u	1,418,726	969,671	1,787,940	542,191	1,813,056	147,303	164,123	2,891,896	1,219,762	43,982	1,105,569
machine_clears.memory_ordering:u	857,422,654	5,643	813,540,719	15,570,872	47,409,380	4,985,424	1,873	145,558	55,255	1,739	32,093
dTLB-loads	20,077,200,396	19,936,586,090	20,041,014,490	3,180,902,122	4,031,547,457	1,796,606,615	206,491,046	530,956,923	344,903,112	29,675,241	188,051,111
dTLB-load-misses	589,416	476,528	597,851	392,934	895,378	165,721	106,513	4,729,600	1,054,650	6,398	182,379
dTLB-stores	20,068,401,460	19,891,165,623	20,037,561,929	2,016,013,457	2,698,948,574	1,341,934,985	204,402,386	245,040,835	106,640,305	15,387,458	49,913,106
dTLB-store-misses	33,164	19,483	55,533	24,793	63,850	11,752	7,704	353,479	88,338	910	26,080
iTLB-loads	1,088,048	721,980	1,305,550	2,396,169	2,395,895	718,450	110,583	267,944	161,346	45,834	321,320
iTLB-load-misses	599,136	427,992	867,566	4,594,344	9,870,999	187,230	84,981	25,766	39,210	6,931	51,532

	pingpong, 二個排在一起的volatile int, 造成false sharing	將volatile變數配置到不同 L1d cache line	二個thread隨意的存取宣告為volatile的變數	存取全域變數, 使用mutex保護	存取全域變數, 使用semaphore保護	存取全域變數使用spinlock保護	二個thread, 使用atomic operation對全域變數進行存取	pxz (平行化版的xz), CPU使用率800%	tarxz (呼叫了tar及xz二支程式), CPU使用率200%	ls	tar
cpu-cycles:u	150,798,170,668	110,244,097,468	155,093,349,925	30,114,134,115	76,596,778,660	31,519,252,934	29,899,643,321	2,218,372,719	730,791,84	49,198,594	376,873
instructions:u	120,345,103,964	120,940,803,624	80,336,910,505	1,039	15,225,342,389	4,634,718,837	2,511,251,165	1,710,737,832	1,303,05	89,581,443	658,724
cache-references:u	4,708,903,084	73,305,091	5,882,333,353	188,414	2,882,816,846	1,019,898,761	1,292,933,171	45,788,187	17,677	782,738	16,224,042
cache-misses:u	19,997,599	25,136,630			945	7,659,135	11,108,749	24,5			611,093
branch-instructions:u	40,003,326,481	40,156,494,045			228	1,812,626,890	365,426,582	228,			196,400
branch-misses:u	4,035,751	3,503,351			805	53,672,945	2,859,445	12,4			512,294
bus-cycles:u	1,066,280,162	780,906,617	1,055,114,520	214,887,140	554,855,353	230,248,022	210,311,015	20,854,201	5,566,566	582,220	2,726,880
cpu-clock:u	181,534	131,653	188,407	103,846	226,666	37,665	34,834	1,000	1,001	1,000	1,000
task-clock:u	181,534	131,653	188,407	103,846	226,666	37,665	34,834	1,000	1,001	1,000	1,000
執行時間 (使用wait4測量)	24.520	16.839	23.846	15.376	32.510	4.847	2.885				
context-switches (使用wait4測量)	2	3	3	1,215	73,873	3	3				
L1-dcache-loads	20,080,762,609	20,293,573,013	20,061,862,820	3,305,884,074	4,095,703,737	1,874,044,747	357,904,831	519,992,693	319,506,413	30,417,608	166,743,987
L1-dcache-load-misses	1,040,835,772	14,993,061	1,089,323,038	293,057,596	564,336,076	255,117,385	143,752,693	24,089,386	7,789,069	504,203	5,080,518
L1-dcache-stores	20,008,290,221	20,117,477,698	20,023,044,923	2,053,751,951	2,726,905,417	1,150,434,651	238,849,777	242,302,658	104,017,068	13,357,491	49,741,268
L1-icache-load-misses	25,251		17,743,787	8,882,572	34,256,707	4,986,900	2,021,601	2,852,175	4,312,216	1,224,435	8,313,351
i2_rqsts.demand_data_rd_hit:u	2,164		1,845,406	29,948,570	5,222,095	5,658,732	160,977	4,893,813	1,678,074	112,331	1,327,178
i2_rqsts.demand_data_rd_miss	95,588		280,804,231	169,195,367	332,469,920	116,001,408	1,374,263	13,240,102	3,640,483	530,190	2,983,783
mem_inst_retired.lock_loads:u			999,528	426,514,153	642,848,645	218,787,152	194,610,271	49,308	87,404	12,410	120,741
mem_inst_retired.split_loads:u	19,		107,332	22,317	39,562	18,134	9,252	1,756,674	1,808,718	25,984	280,355
mem_inst_retired.split_stores:u	35,		222,116	17,346	56,430	4,001	6,416	5,717	32,852	61,994	43,224
mem_load_l3_hit_retired.xsnp_hit:u	24,		31,234	74,517,828	64,196,125	36,184,632	4,640	1,462	6,691	159	8,386
mem_load_l3_hit_retired.xsnp_hitm:u	89,303	17,735	75,011	42,746,571	230,451,349	73,700,570	193,514,277	5,736	14,475	79	18,490
mem_load_l3_hit_retired.xsnp_miss:u	33,860	17,264	16,868	5,272,555	32,311,502	12,030,370		1,862	2,696	20,210	97
mem_load_l3_hit_retired.xsnp_none:u	1,418,726	969,671	1,787,940	542,191	1,813,056	147,303	164,123	2,891,896	1,219,762	43,982	1,105,569
machine_clears.memory_ordering:u	857,422,654	5,643	813,540,719	15,570,872	47,409,380	4,985,424	1,873	145,558	55,255	1,739	32,093
dTLB-loads	20,077,200,396	19,936,586,090	20,041,014,490	3,180,902,122	4,031,547,457	1,796,606,615	206,491,046	530,956,923	344,903,112	29,675,241	188,051,111
dTLB-load-misses	589,416	476,528	597,851	392,934	895,378	165,721	106,513	4,729,600	1,054,650	6,398	182,379
dTLB-stores	20,068,401,460	19,891,165,623	20,037,561,929	2,016,013,457	2,698,948,574	1,341,934,985	204,402,386	245,040,835	106,640,305	15,387,458	49,913,106
dTLB-store-misses	33,164	19,483	55,533	24,793	63,850	11,752	7,704	353,479	88,338	910	26,080
iTLB-loads	1,088,048	721,980	1,305,550	2,396,169	2,395,895	718,450	110,583	267,944	161,346	46,834	321,320
iTLB-load-misses	599,136	427,992	867,566	4,594,344	9,870,999	187,230	84,981	25,766	39,210	6,931	51,532

記憶體的
特殊存取方式

常用的Linux指令

紅色部分與
記憶體存取
異常相關

確認問題後， 該如何找出造成該問題的程式

```
$ sudo perf report -i mem_pingpong.perf.data
```

```
/*針對要進一步了解的事件「選進去」，再選擇「Annotate thread」*/
```

```
| incq 0x20387d(%rip)    # 604518 <ds+0x8>  
48.64 | cmp %rdx,%rax  
| ↑ jg c  
| mov $0x401cd4,%edi  
| xor %eax,%eax  
| mov ds+0x8,%rsi  
| ↑ jmpq 400a40 <printf@plt>  
| 33: inc %rdx  
| incq 0x203853(%rip)    # 604510 <ds>  
51.36 | cmp %rdx,%rax  
| ↑ jg c
```

如果編譯時使用了-g

有時候bug只有在開啟-O3發生

```

|                                     ds.a++;
|
|     mov     ds,%rax
40.82 |     mov     %rax,-0x30(%rbp)
|     mov     $0x1,%eax
|     add     -0x30(%rbp),%rax
|     mov     %rax,-0x28(%rbp)
|     mov     -0x28(%rbp),%rax
|     mov     %rax,ds
|     ↑ jmp    2f
|
|                                     else
|
|                                     ds.b++;
|6b:   mov     ds+0x8,%rax
59.18 |     mov     %rax,-0x20(%rbp)
```


小結

- 🍏 了解支援多執行緒的硬體
- 🍏 了解pipeline及super-scaler及compiler都可能將「表面上」看起來不相關的程式碼對調
- 🍏 了解一道組合語言可能分成多個步驟執行
- 🍏 了解在x86於cache上保證了記憶體的一致性
- 🍏 了解有些指令雖然是複雜動作（例如：read-modify-write）但是CPU的設計者，將這些特別指令設計成atomic operation



thread的programming
language/model

C11 thread

```
1.  #include <threads.h>
2.  int run(void* par) {
3.      for (int i=0; i< 100000; i++)
4.          printf("%5d ", i);
5.  }
6.  int main(int argc, char** argv) {
7.      thrd_t thr1, thr2, thr3, thr4;
8.      thrd_create(&thr1, run, NULL);
9.      thrd_create(&thr2, run, NULL);
10.     thrd_create(&thr3, run, NULL);
11.     thrd_create(&thr4, run, NULL);
12. }
```

結果

```
$ gcc c11thread.c
c11thread.c:1:10: fatal error: threads.h: No such
file or directory
  #include <threads.h>
           ^~~~~~
compilation terminated
/*在ubuntu 18.04上libc並未支援c11 <threads.h>*/
```

C++ thread

```
1.  #include <thread>
2.  int run(int par) {
3.      for (int i=0; i< 100000; i++)
4.          switch (par) {
5.              case 1: printf(RED"%5d ", i); break;
6.              case 2: printf(GREEN"%5d ", i); break;
7.              case 3: printf(CYAN"%5d ", i); break;
8.              case 4: printf(YELLOW"%5d ", i); break;
9.          }
10. }
11. int main(int argc, char** argv) {
12.     std::thread thrd1 (run,0); std::thread thrd2 (run,1);
13.     std::thread thrd3 (run,2); std::thread thrd4 (run,3);
14.     thrd1.join();thrd2.join();
15.     thrd3.join();thrd4.join();
16. }
```

結果

```
$ g++ -pthread cppThread.cpp
```

```
$ ./timedetail ./a.out
```

```
0 1 2 3 4 5 6 7 8 9 10 11
12 0 1 2 3 4 5 6 7 8 9 10 11
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0 26 11 27 12 28 13 29 14 30 31 17
31 18 19 32
```

CPU花在執行程式的時間: 0.083779s

Kernel協助處理的時間: 0.156387s

Page fault, 但沒有造成I/O: 138

Page fault, 但觸發I/O: 0

自願性的context switch: 19395

非自願性的context switch: 4

結果

- 🍏 C++的thread使用了glibc的pthread實現
- 🍏 pthread使用Linux kernel的clone() system call實現
- 🍏 對Linux而言thread和process於kernel內部是一樣的。差異只是：Thread之間共享記憶體，process之間並未共享記憶體。
 - 🍀 Linux內部都是使用task_struct控制thread和process
 - 🍀 Linux的這三個system call: fork、vfork、clone在核心內部都呼叫do_fork()
- 🍏 除非有特別需求，否則直接呼叫clone並不會帶來太大的效能改善
 - 🍀 特別需求如：希望每一個thread有獨立的open-file table
 - 🍀 使用clone的時候要特別注意stack的傳遞方式 &newStack[size-1]

我們使用的函數庫

- 🍏 選擇使用<pthread.h>函數庫
- 🍏 使用<stdatomic.h>，功能幾乎和<atomic.h>一樣
- 🍏 <atomic.h>及<stdatomic.h>主要提供下列功能
 - 🍀 定義各種基本型別的atomic_t
 - 🍀 定義各種atomic operation
 - 🍀 定義了memory barrier
- 🍏 C11的<atomic.h>的功能，大致相等於C++11的<atomic>

編譯方式

```
gcc xxx.c -g -pthread -o xxx  
gcc xxx.c -O3 -pthread -o xxx  
/*gcc可以替代成icc或clang*/
```

base.c (比較基準)

```
1. #include <stdio.h>
2. int global=0;
3. int main(int argc, char **argv)
4. {
5.     int i;
6.     for (i=0; i<2000000000; i++) {
7.         global+=1;
8.     }
9.     printf("1000000000+1000000000 = %d\n", global);
10.    return 0;
11. }
```

執行結果

```
$ ./time_detail ./base  
1000000000+1000000000 = 2000000000
```

經過時間:	4.709708408s
CPU花在執行程式的時間:	4.709391s
CPU於usr mode執行此程式所花的時間:	4.709391s
CPU於krl mode執行此程式所花的時間:	0.000000s
Page fault, 但沒有造成I/O:	63
Page fault, 並且觸發I/O:	0
自願性的context switch:	1
非自願性的context switch:	33

pthread_create()

```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void  
*(*function)(void *), void *argument)
```

- 🍏 tid, thread id
- 🍏 attr, 新建立的thread的屬性, 使用pthread_attr_init及相關函數初始化, 一般填入NULL
- 🍏 function, 該thread所要執行的函數的名稱, 該函數的回傳值和參數都是void*
- 🍏 argument傳遞給上述函數 (即: thread) 的參數
- 🍏 正確回傳0, 錯誤則看errno

建立一個pthread (nosync.c)

```
1.  int global=0;
2.  void thread(void) {
3.      int i;
4.      for (i=0; i<1000000000; i++)
5.          global+=1;    //大亂鬥，二個執行緒同時修改global變數
6.  }
7.  int main(void) {
8.      pthread_t id1, id2;
9.      pthread_create(&id1, NULL, (void *) thread, NULL);
10.     pthread_create(&id2, NULL, (void *) thread, NULL);
11.     pthread_join(id1, NULL);
12.     pthread_join(id2, NULL);
13.     printf("1000000+1000000 = %d\n", global);
14. }
```

執行結果

```
$ ./time_detail ./nosync  
1000000+1000000 = 1014803209
```

經過時間：	6.911069487s
CPU花在執行程式的時間：	13.808802s
CPU於usr mode執行此程式所花的時間：	13.808802s
CPU於kr1 mode執行此程式所花的時間：	0.000000s
Page fault, 但沒有造成I/O:	76
Page fault, 並且觸發I/O:	0
自願性的context switch:	3
非自願性的context switch:	55

結果討論

- 🍏 平行度幾乎是最大化，經過時間6.9秒，CPU時間為13.8。這個程式只有二個執行緒，達到的平行度為2（理論最高值）
- 🍏 但6.9秒還是比完全不用執行緒的base要來得慢（4.7秒），表示overhead很大
- 🍏 因為二個執行緒共用一個變數，而且沒有任何同步機制，這會造成二個執行緒彼此覆寫全域變數

A collage of various Linux mascot penguins in different costumes, including a firefighter, a doctor, a chef, and a pirate. The word "semaphore" is written in yellow text across the center.

semaphore

```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

sem: 要初始化的semaphore的物件指標

pshared: 0該semaphore給執行緒使用, 1給行程使用

value: 要將semaphore初始化成value

sem_post() and sem_wait()

1. `#include <semaphore.h>`

2. `int sem_post(sem_t *sem);`

3. `int sem_wait(sem_t *sem);`

🍏 `sem_post`離開全域變數存取區間。在意義上可視為unlock。

🍏 `sem_wait`準備進入全域變數存取區間。在意義上可視為lock。

🍏 存取區間即為critical section，在critical section裡面的程式碼存取「同樣的資料」，為了避免資料被隨意的修改，因此可使用semaphore之類的技術，保證一次只能有一個人修改。

使用semaphore

```
1.  int global=0; sem_t semaphores;
2.  void thread(void) {
3.      int i;
4.      for (i=0; i<1000000000; i++) {
5.          sem_wait(&semaphores); //向OS要求進入critical section修改global
6.          global+=1;
7.          sem_post(&semaphores); //告訴OS修改完成，離開critical section
8.      } }
9.  int main(void) {
10.     pthread_t id1, id2;
11.     sem_init(&semaphores, 0, 1); /*0:thread使用, 1:semaphore只允許一個人修改global*/
12.     pthread_create(&id1, NULL, (void *) thread, NULL);
13.     pthread_create(&id2, NULL, (void *) thread, NULL);
14.     pthread_join(id1, NULL); pthread_join(id2, NULL);
15.     printf("1000000000+1000000000 = %d\n", global);
16. }
```

執行結果

```
$ ./time_detail ./semaphore
1000000000+1000000000 = 2000000000
```

經過時間:	276.5058334s
CPU花在執行程式的時間:	547.802264s
CPU於usr mode執行此程式所花的時間:	324.851308s
CPU於krl mode執行此程式所花的時間:	222.950956s
Page fault, 但沒有造成I/O:	76
Page fault, 並且觸發I/O:	0
自願性的context switch:	276217
非自願性的context switch:	313z

nosync

5.76788879s
9.187857s
9.187857s
0.000000s
75
0
3
11

結果討論

- 🍏 semaphore是功能強大的同步函數，他的初始值可以是任何整數，通常semaphore的初始值N代表最多可以有N個執行緒修改全域變數或進入critical section
- 🍏 使用sem_wait()時，如果沒有lock成功，會造成context switch
- 🍏 spinlock（後面會介紹），如果lock不成功會在user space一直等待下去，不會造成context switch
- 🍏 小結：如果critical section較短，通常使用spinlock。critical section較長，通常使用semaphore之類的。如果處理器的數量夠多，又想最佳化latency，可以仔細的考慮是否使用spinlock。



mutex

mutex

1. `#include <pthread.h>`
2. `int pthread_mutex_init(pthread_mutex_t * mutex, pthread_mutexattr_t * attr)`
3. `int pthread_mutex_lock(pthread_mutex_t *mutex);`
4. `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

🍏 attr: 設定這個mutex的屬性，使用預設屬性則傳入NULL

使用mutex

```
1.  int global=0;
2.  pthread_mutex_t mutex;
3.  void thread(void) {
4.      for (int i=0; i<1000000000; i++) {
5.          pthread_mutex_lock(&mutex);
6.          global+=1;
7.          pthread_mutex_unlock(&mutex);
8.      }
9.  }
10. int main(void) {
11.     pthread_t id1, id2;
12.     pthread_mutex_init(&mutex, NULL);          //mutex預設是unlock
13.     pthread_create(&id1,NULL,(void *) thread,NULL); pthread_create(&id2,NULL,(void *) thread,NULL);
14.     pthread_join(id1,NULL); pthread_join(id2,NULL);
15.     printf("1000000000+1000000000 = %d\n", global);
16. }
```

執行結果

```
$ ./time_detail ./mutex  
1000000000+1000000000 = 2000000000
```

經過時間:	123.151282951s
CPU花在執行程式的時間:	244.681460s
CPU於usr mode執行此程式所花的時間:	150.435999s
CPU於kr1 mode執行此程式所花的時間:	94.245461s
Page fault, 但沒有造成I/O:	76
Page fault, 並且觸發I/O:	0
自願性的context switch:	10402
非自願性的context switch:	125

semaphore

276.5058334s
547.802264s
324.851308s
222.950956s
76
0
276217
313

結果討論

- 🍏 mutex的值只能是1, 0, 這意味著最多只有一個thread能進入critical section
- 🍏 大部分的情況, 我們只允許一次一個人進入critical section
- 🍏 mutex的系統負擔比semaphore小很多



semaphore與mutex的 效能比較

2job.c (比較基準)

```
1.  int global=0;
2.  pthread_mutex_t mutex;
3.  void thread(void) {
4.      int local=0; int i;
5.      for (i=0; i<1000000000; i++)
6.          local+=1;
7.      pthread_mutex_lock(&mutex);
8.      global+=local;
9.      pthread_mutex_unlock(&mutex);
10. }
11. int main(void) {
12.     pthread_t id1, id2;
13.     pthread_mutex_init(&mutex, NULL);
14.     pthread_create(&id1, NULL, (void *) thread, NULL);
15.     pthread_create(&id2, NULL, (void *) thread, NULL);
16.     pthread_join(id1, NULL); pthread_join(id2, NULL);
17.     printf("1000000000+1000000000 = %d\n", global);
18. }
```

執行結果

```
$ ./time_detail ./2job  
1000000000+1000000000 = 2000000000
```

經過時間:	2.245269591s
CPU花在執行程式的時間:	4.458316s
CPU於usr mode執行此程式所花的時間:	4.458316s
CPU於krl mode執行此程式所花的時間:	0.000000s
Page fault, 但沒有造成I/O:	76
Page fault, 並且觸發I/O:	0
自願性的context switch:	3
非自願性的context switch:	273

執行結果分析

- 🍏 job2高度的平行化，執行時間為2.245秒，user為4.458秒。換句話說幾乎所有執行時間都是平行運算。
- 🍏 更重要的是在整合結果時使用mutex，因此overhead相當低。kernel時間為0，context switch只發生3次。
- 🍏 job2告訴我們，可以的話，將工作完全切開，使用區域變數平行計算，最後再做結果的合併（這時候只要用簡單的同步即可）。

結果比較 (回合數: 1,000,000,000)

	base	nosync (錯誤)	semaphore	mutex	2job (理論上應該是最快)
real	4.244s	5.371s	289.102s	217.471s	2.131s
user	4.240s	10.520s	333.272s	225.196s	4.248s
sys	0.000s	0.000s	231.756s	88.820s	0.000s

結果比較 (回合數: 1,000,000,000)

	base	nosync (錯誤)	semaphore	mutex	2job (理論上應該是最快)
real	4.248s				2.131s
user	4.248s				4.248s
sys	0.000s	0.000s	231.756s	88.820s	0.000s

比較，處理器變為二倍，執行時間所短為1/2，極限！

user的值幾乎一樣，代表所做的工作是一樣多

小結

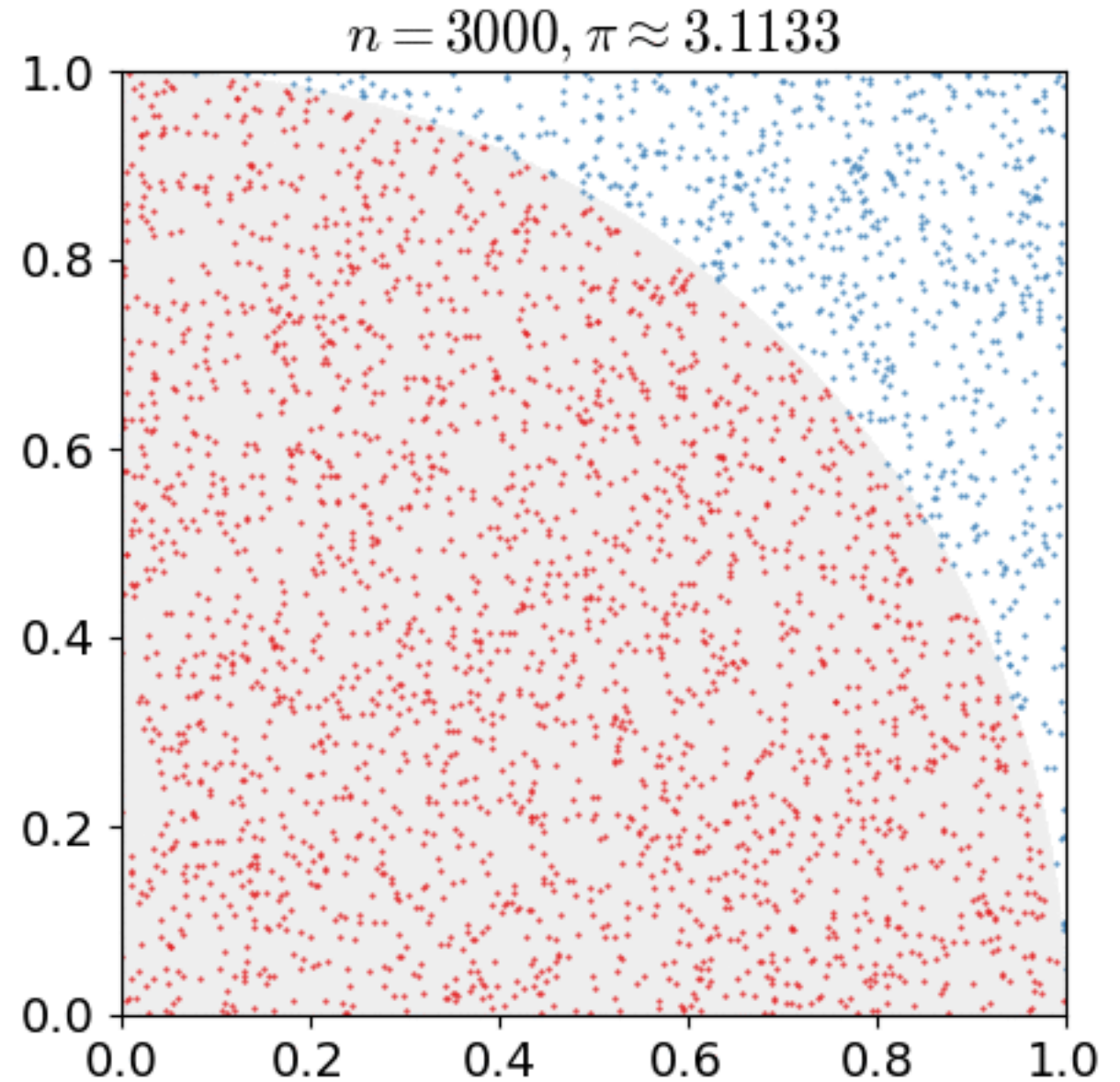
- 🍏 瞭解pthread_create和pthread_join
- 🍏 瞭解mutex、semaphore、lock-free三種鎖定方法
- 🍏 比較各種同步方法
- 🍏 最好的方法就是「盡量不同步，並且結果正確」



使用pthread計算pi

蒙特卡羅方法

- 蒙特卡羅法：隨機打出一些點，看這些點是否落在函數與X, Y軸所構成的面積內，計算落入，和沒落入的點數，即可以知道該函數的積分值
- 於「計算pi」上的應用
 - 如右圖，隨機產生介於(0,1)的(x,y)點
 - 落在圓內的標示為紅色，圓外標示為藍色
 - 即算紅點占所有點（紅點+藍點）的比例就可以算出圓面積
 - 半徑為1，圓面積為A，即可算出pi



pi_drand48_r.c

```
1. void thread(void* rand_buffer)
2. {
3.     for(i=0;i<local_loopcount;i++) {
4.         /*使用drand48_r由於產生random number相關的資料結構都
5.         保存在rand_buffer, 因此不需要lock-unlock機制*/
6.         drand48_r(rand_buffer, &rand_d);point_x = rand_d;
7.         drand48_r(rand_buffer, &rand_d);point_y = rand_d;
8.         if( (point_x*point_x + point_y*point_y) < 1.0)
9.             hit+=1;
10.    }
11.    pthread_mutex_lock(&mutex);
12.    global_hit += hit;
13.    pthread_mutex_unlock(&mutex);
14. }
```

```
1. void main(int argc,char*argv[]) {
2.     pthread_mutex_init(&mutex,NULL);
3.     for(i=0;i<num_thread;i++) {
4.         /*給drand48_r相對應的random seed*/
5.         /*這裡假設rand和drand48_r用不同的隨機產生法*/
6.         srand48_r(rand(), &rand_buffer[i]);
7.         /*將random buffer透過參數傳遞給thread*/
8.         pthread_create(&id[i],NULL,(void *)thread,
9.             &rand_buffer[i]);
10.    }
11.    for(i=0;i<num_thread;i++)
12.        pthread_join(id[i],NULL);
13.    pi = (double)4*(global_hit/total_loopcount);
14.    printf("pi = %.8lf\n",pi);
15. }
```

效能瓶頸 (使用perf top觀察)

Samples: 12M of event 'cycles:ppp', Event count (approx.): 214031732918

Overhead	Shared Object	Symbol
63.39%	libc-2.27.so	[.] __drand48_iterate
25.77%	libc-2.27.so	[.] __erand48_r
8.27%	pi_drand48_r	[.] thread
0.82%	pi_drand48_r	[.] drand48_r@plt
0.81%	libc-2.27.so	[.] drand48_r

關於process/thread id

- 🍏 對Linux而言，process和thread都是「task」，每個task都有獨立的id
- 🍏 於Linux中，主執行緒所屬的每個執行緒的tid代表的是指向struct pthread_t的指標

在Linux中的thread id

1. `#define _GNU_SOURCE`
2. `#include <sys/types.h>`
3. `#include <unistd.h>`
4. `#include <sys/syscall.h>`
5. `//libc不提供gettid函數，但如果要除錯（gdb）的話，必須拿到tid才可以attach`
6. `int gettid() {`
7. `return syscall(SYS_gettid);`
8. `}`

thread_print_id.c

```
11. void thread(void) {
12.     printf("pthread_t tid = %p\n", (void*)pthread_self());
13.     printf("tid = %d\n", gettid());
14.     while(1);
15. }
16.
17. int main(void) {
18.     pthread_t id1, id2;
19.     printf("my pid = %d\n", getpid());
20.     pthread_create(&id1, NULL, (void *) thread, NULL);
21.     pthread_create(&id2, NULL, (void *) thread, NULL);
22.     getchar();
23.     return (0);
24. }
```

執行結果

```
$ ./thread_print_id
my pid = 26653
pthread_t tid =
0x7fd634031700
tid = 26654
pthread_t tid =
0x7fd633830700
tid = 26655
```

```
$ ls /proc/26653/task/
26653  26654  26655
$ ps -L -p 26653
      PID     LWP  TTY
TIME  CMD
26653  26653 pts/17
00:00:00 thread_print_id
26653  26654 pts/17
00:08:34 thread_print_id
26653  26655 pts/17
00:08:34 thread_print_id
```

退出thread

1. 在thread中直接執行return，請注意回傳值的型態為void*
 - ✿如果回傳型態宣告為void也可以，但不可以回傳值
2. 使用void pthread_exit(void *retval);
 - ✿retval是回傳值
3. 使用int pthread_cancel(pthread_t thread);
 - ✿直接取消掉一個thread，或者「建議取消掉」一個thread
 - ✿int pthread_setcancelstate(int state, int *oldstate);
 - ✿int pthread_setcanceltype(int type, int *oldtype);
 - ✿後續再做介紹

pthread_detach()

🍏 `#include <pthread.h>`

🍏 `int pthread_detach(pthread_t thread);`

- 🍏 通常需要使用pthread_join釋放掉thread所使用的記憶體（如：stack、struct pthread）
- 🍏 當「回傳值」不重要，不需要join時，可以使用pthread_detach()
- 🍏 常用的方式：pthread_detach(pthread_self());

pthread_detach()

🍏 `#include <pthread.h>`

🍏 `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);`

🍏 `int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);`

🍏 在行程創建之初，設定attr

🍀 `int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*function)(void *), void *argument)`

🍏 請自行參考man pthread_attr_setdetachstate

輕量級的鎖

- 🍏 `int pthread_spin_init(pthread_spinlock_t *lock, int pshared);`
 - 🍏 `int pthread_spin_destroy(pthread_spinlock_t *lock);`
 - 🍏 `int pthread_spin_lock(pthread_spinlock_t *lock);`
 - 🍏 `int pthread_spin_unlock(pthread_spinlock_t *lock);`
- 🍏 使用自旋鎖 (spin-lock) 會利用一個密集的迴圈進行測試，適用於critical section很小的情況下

pthread_spin_lock

- 🍏 `#include <pthread.h>`
- 🍏 `int pthread_spin_init(pthread_spinlock_t *lock int pshared);`
- 🍏 `int pthread_spin_destroy(pthread_spinlock_t *lock);`
- 🍏 `int pthread_spin_lock(pthread_spinlock_t *lock);`
- 🍏 `int pthread_spin_trylock(pthread_spinlock_t *lock);`
- 🍏 `int pthread_spin_unlock(pthread_spinlock_t *lock);`
- 🍏 如果這個lock沒有要跨process使用，pthread_spin_init的pshared傳入PTHREAD_PROCESS_PRIVATE
- 🍏 pthread_spin用法和Mutex幾乎一模一樣，唯一的差別在於pthread_spin不會釋放CPU，也不會造成voluntary context switch（自願釋放CPU）

spinlock.c

```
1.  void thread(void) {
2.      int i;
3.      for (i=0; i<1000000000; i++) { //lock 1,000,000,000 次
4.          pthread_spin_lock(&spinlock);
5.          global+=1;
6.          pthread_spin_unlock(&spinlock);
7.      }
8.  }
9.  int main(void) {
10.     pthread_t id1, id2;
11.     pthread_spin_init(&spinlock, PTHREAD_PROCESS_PRIVATE);
12.     pthread_create(&id1, NULL, (void *) thread, NULL);
13.     pthread_create(&id2, NULL, (void *) thread, NULL);
14.     pthread_join(id1, NULL);
15.     pthread_join(id2, NULL);
16. }
```

spinlock.c

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #include <semaphore.h>
4.  #include <stdio.h>
5.
6.  int global=0;
7.  pthread_spinlock_t spinlock;
8.
9.  void thread(void) {
10.     int i;
11.     for (i=0; i<1000000; i++) {
12.         pthread_spin_lock(&spinlock);
13.         global+=1;
14.         pthread_spin_unlock(&spinlock);
15.     }
16. }
```

比較, lock 1,000,000,000次, 花的時間

	spinlock	mutex	semaphore	atomic
經過時間	42.37	124.90	311.20	41.72
USR	82.58	149.71	363.27	83.42
kernel	0	97.40	258.18	0
ctx-sw	2	8,074	170,820	2

spinlock vs. atomic

```
1.  //spinlock
2.  for (i=0; i<1000000000; i++) {
3.      pthread_spin_lock(&spinlock);
4.      global+=1;
5.      pthread_spin_unlock(&spinlock);
6.  }
7.  //atomic
8.  for (i=0; i<1000000000; i++)
9.      atomic_fetch_add(&global, 1);
```

反組譯 atomic.c

1. 0x0000000000400c80 <+0>: xor %eax,%eax
2. 0x0000000000400c82 <+2>: mov \$0x6040c4,%edx
3. 0x0000000000400c87 <+7>: mov \$0x1,%ecx
4. 0x0000000000400c8c <+12>: **lock** add %ecx,(%rdx)
5. 0x0000000000400c8f <+15>: inc %eax
6. 0x0000000000400c91 <+17>: cmp \$0x3b9aca00,%eax
7. 0x0000000000400c96 <+22>: jl 0x400c82 <thread+2>
8. 0x0000000000400c98 <+24>: retq
9. 0x0000000000400c99 <+25>: nopl 0x0(%rax)

Lock: x86的前綴字

- 🍏 The LOCK prefix ensures that the CPU has **exclusive ownership of the appropriate cache line** for the duration of the operation, and provides certain additional ordering guarantees. **This may be achieved by asserting a bus lock**, but the CPU will avoid this where possible. If the bus is locked then it is only for the duration of the locked instruction.
- 🍏 換句話說：在比較好的情況下，CPU會鎖定該變數所在的cache line，最差的情況下會鎖定bus。
- 🍏 在鎖bus的情況下，其他CPU、core不能存取記憶體

POSIX's spinlock

1. `do {`
2. `do {`
3. `atomic_spin_nop();//asm ("rep; nop")`
4. `val = atomic_load_relaxed(lock);`
5. `} while (val != 0);`
6. `} while (!atomic_compare_exchange_weak_acquire(lock, &val, 1));`
7. `/*rep; nop is indeed the same as the pause instruction. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.*/`

什麼是futex(fast user-space locking)

The futex() system call provides a method for waiting until a certain condition becomes true. It is typically used as a blocking construct in the context of shared-memory synchronization. When using futexes, the majority of the synchronization operations are performed in user space. A user-space program employs the futex() system call only when it is likely that the program has to block for a longer time until the condition becomes true. Other futex() operations can be used to wake any processes or threads waiting for a particular condition.

Spinlock一定比較快嗎？

- 🍏 不一定，當critical section很大時，spinlock的效能反而不如mutex及semaphore了

取得平衡

- 🍏 改成使用自適應鎖 (adaptive) , 先用自旋鎖, 如果鎖太久自動轉換為mutex。 (最早出現在SUN Solaris)
- 🍏 方法如下:
 1. `pthread_mutexattr_t attr;`
 2. `pthread_mutexattr_init(&attr);`
 3. `pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ADAPTIVE_NP);`
 4. `pthread_mutex_init(&mutex, &attr);`

mutex_apaptive.c

```
1.  void* thread(void* para) {
2.      int i;
3.      for (i=0; i<1000000; i++) {
4.          pthread_mutex_lock(&mutex);
5.          global+=1;
6.          pthread_mutex_unlock(&mutex);
7.      }
8.  }
9.  int main(void) {
10.     pthread_mutexattr_t attr;
11.     pthread_mutexattr_init(&attr);
12.     pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ADAPTIVE_NP);
13.     pthread_mutex_init(&mutex, &attr);
14. }
```

Mutex的其他選項

- 🍏 PTHREAD_MUTEX_TIMED_NP
 - ♣️ 預設值，先進先出
- 🍏 PTHREAD_MUTEX_RECURSIVE_NP
 - ♣️ 可以遞迴使用
- 🍏 PTHREAD_MUTEX_ERRORCHECK_NP
 - ♣️ 會幫忙檢查有沒有deadlock（重複上鎖）
- 🍏 PTHREAD_MUTEX_ADAPTIVE_NP
 - ♣️ 用spinlock做SMP的優化，unlock的時候不保證先進先出
 - ♣️ 速度最快的Mutex

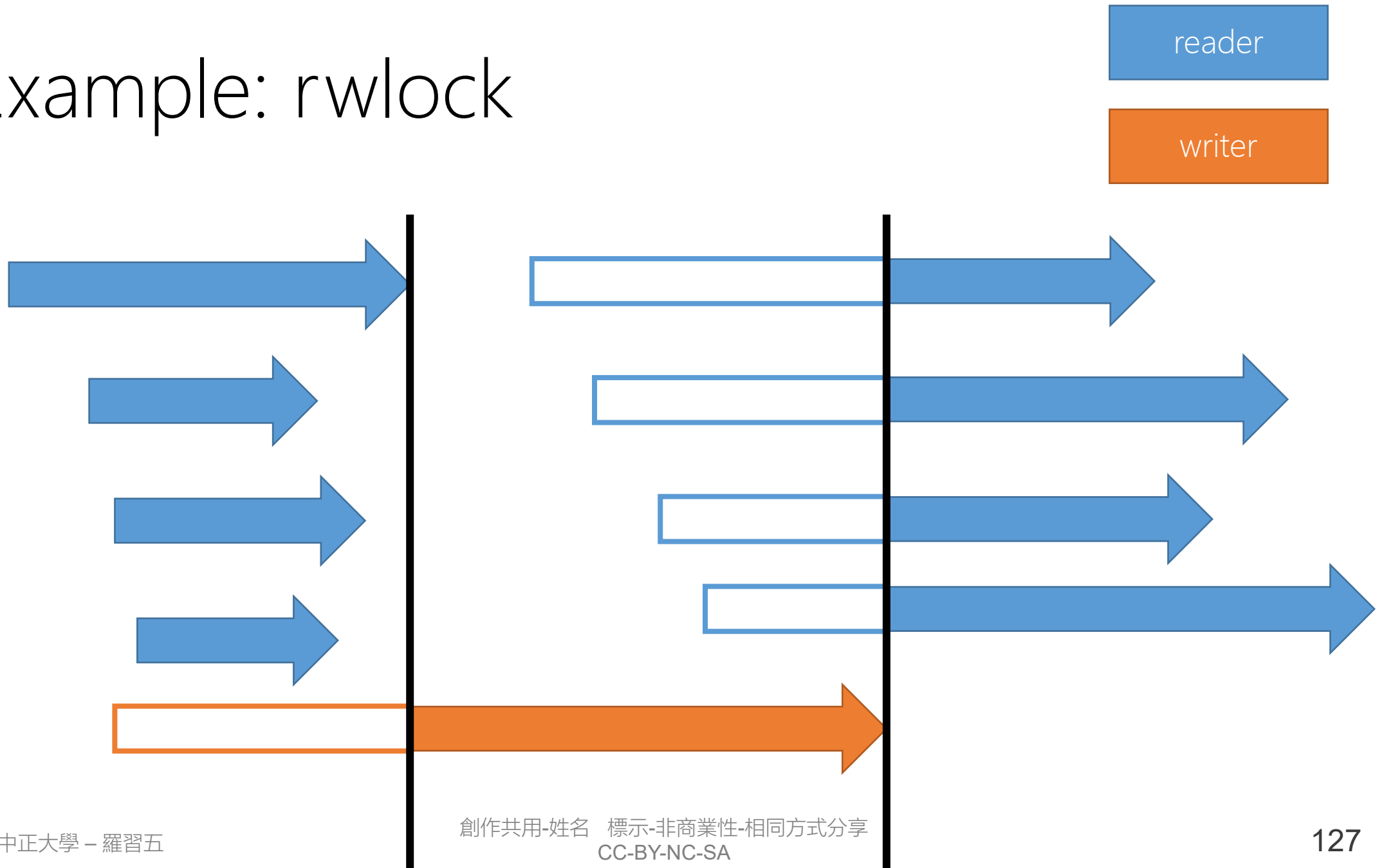
PTHREAD_MUTEX_RECURSIVE_NP 的用途

```
1. void a () {  
2.     mutexlock(this_mutex);  
3.     /*...*/  
4.     b();  
5.     /*...*/  
6.     mutexunlock(this_mutex);  
7. }  
8. void b() { //在b()中重複lock了 "this_mutex"  
9.     mutexlock(this_mutex);  
10.    /*...*/  
11.    mutexunlock(this_mutex);  
12. }
```

更進階的lock機制, rwlock

- 🍏 `int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);`
- 🍏 `int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);`
- 🍏 `int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);`
- 🍏 `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`
- 🍏 `int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);`
- 🍏 pthread_rwlock_init中的attr通常填NULL
- 🍏 將lock依照使用的情況，區分為write lock和read lock，可以更進一步的平行化

Example: rwlock



範例：rwlock

待補充

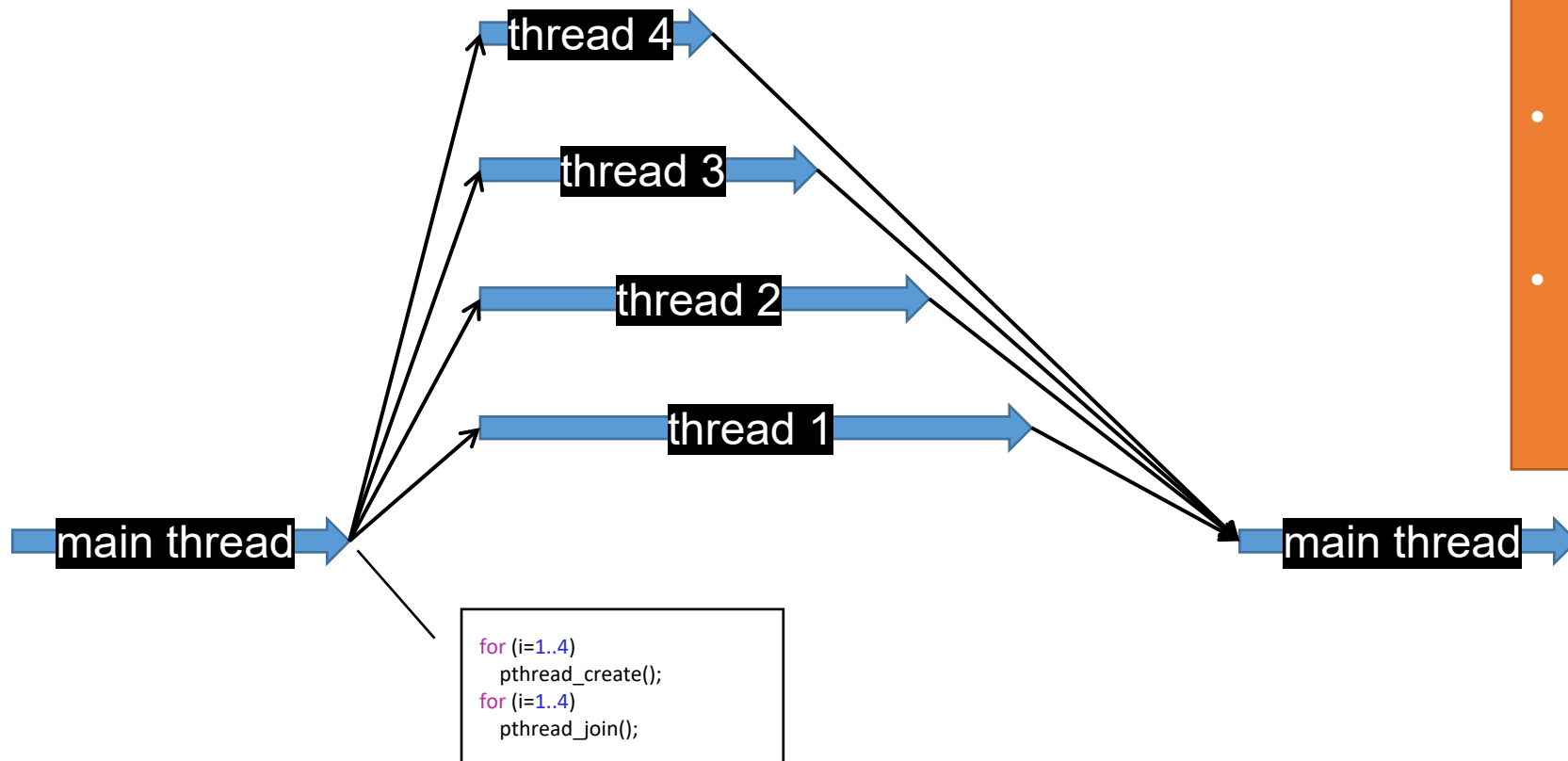
小結

- 🍏 瞭解Linux中thread在kernel space和user space各為何
- 🍏 除了會「睡覺」的mutex及semaphore以外，還有「不會睡覺」的spinlock
- 🍏 綜合mutex及spinlock的優點，創造出adaptive mutex
- 🍏 瞭解rwlock的設計動機



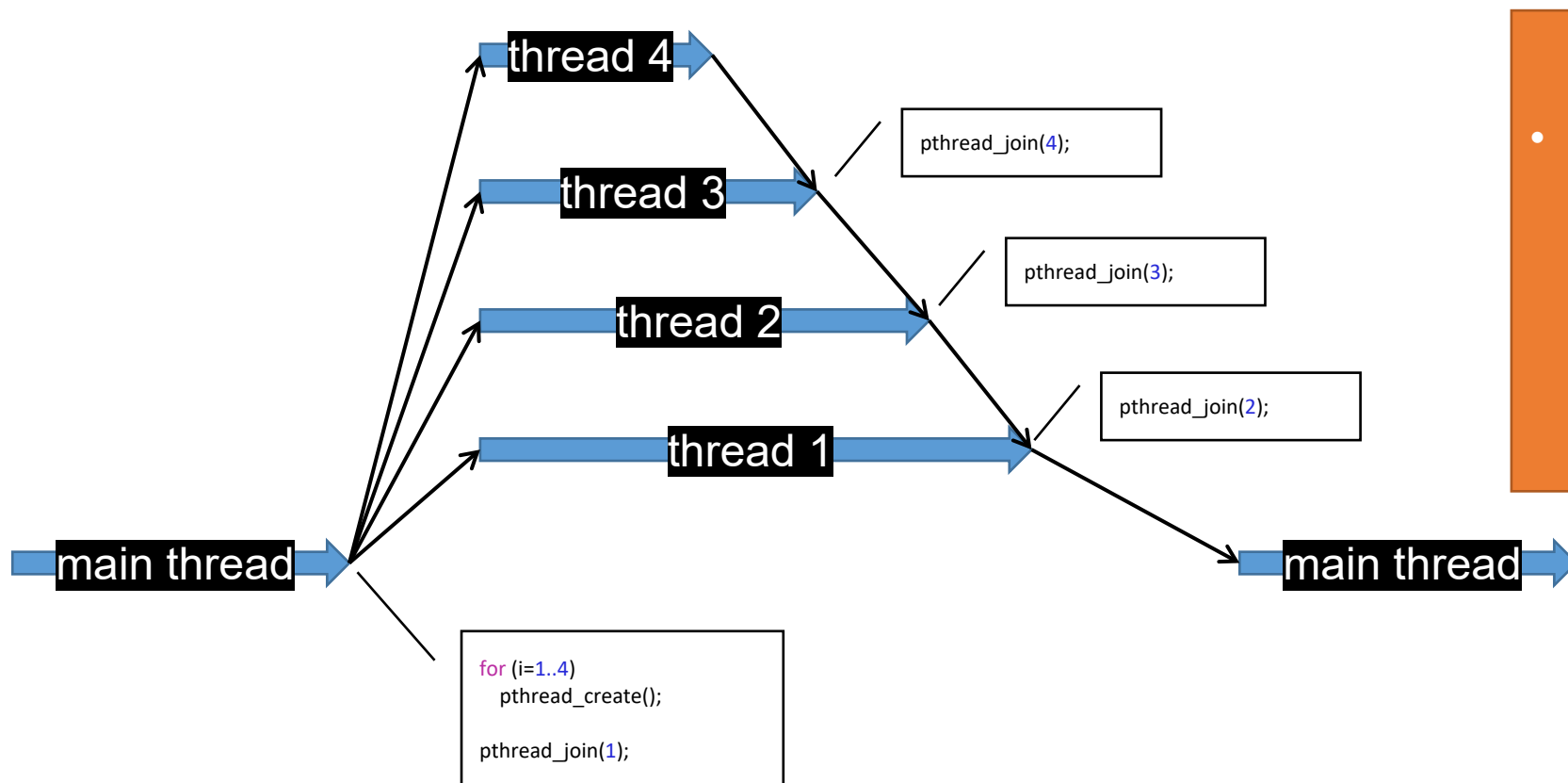
pthread小結

pthread小結 - create & join



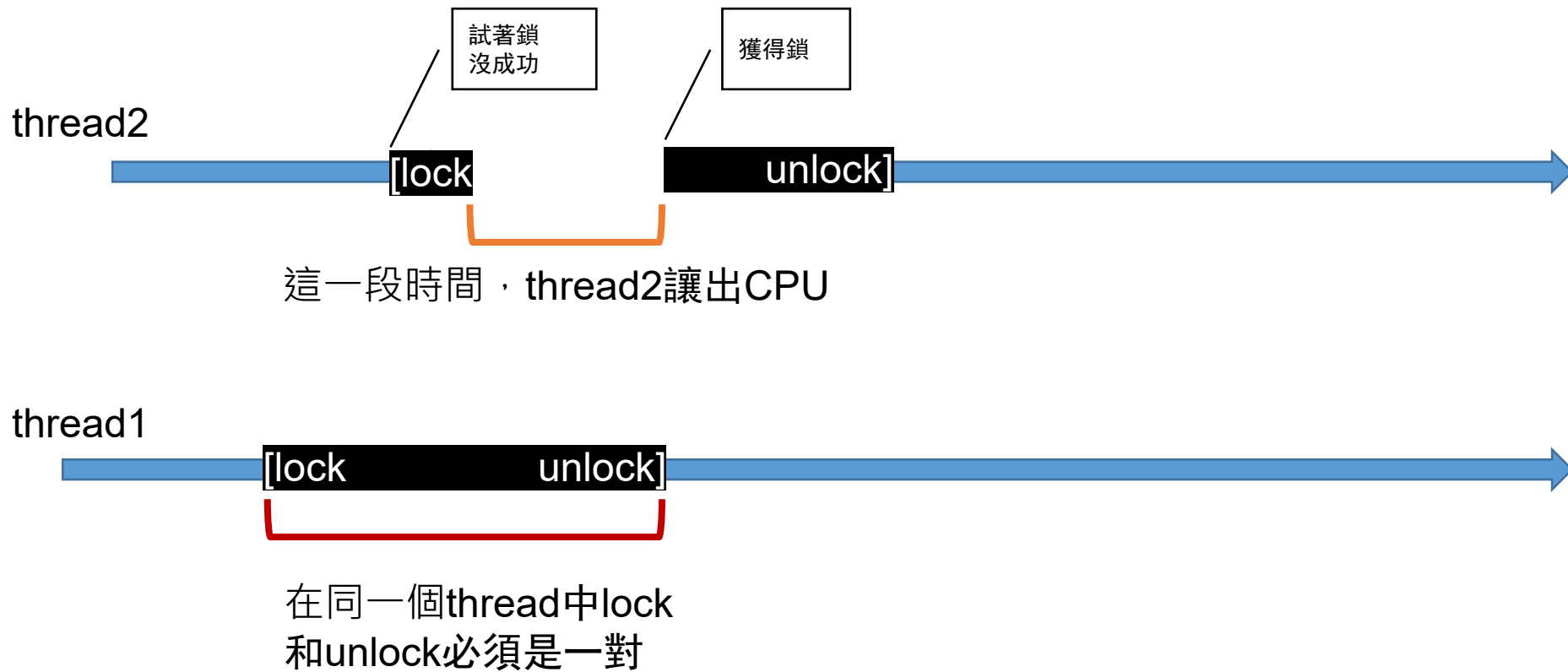
- 可以main thread等所有的thread執行完成
- 這個模式跟process的wait很像

pthread小結 – create & join

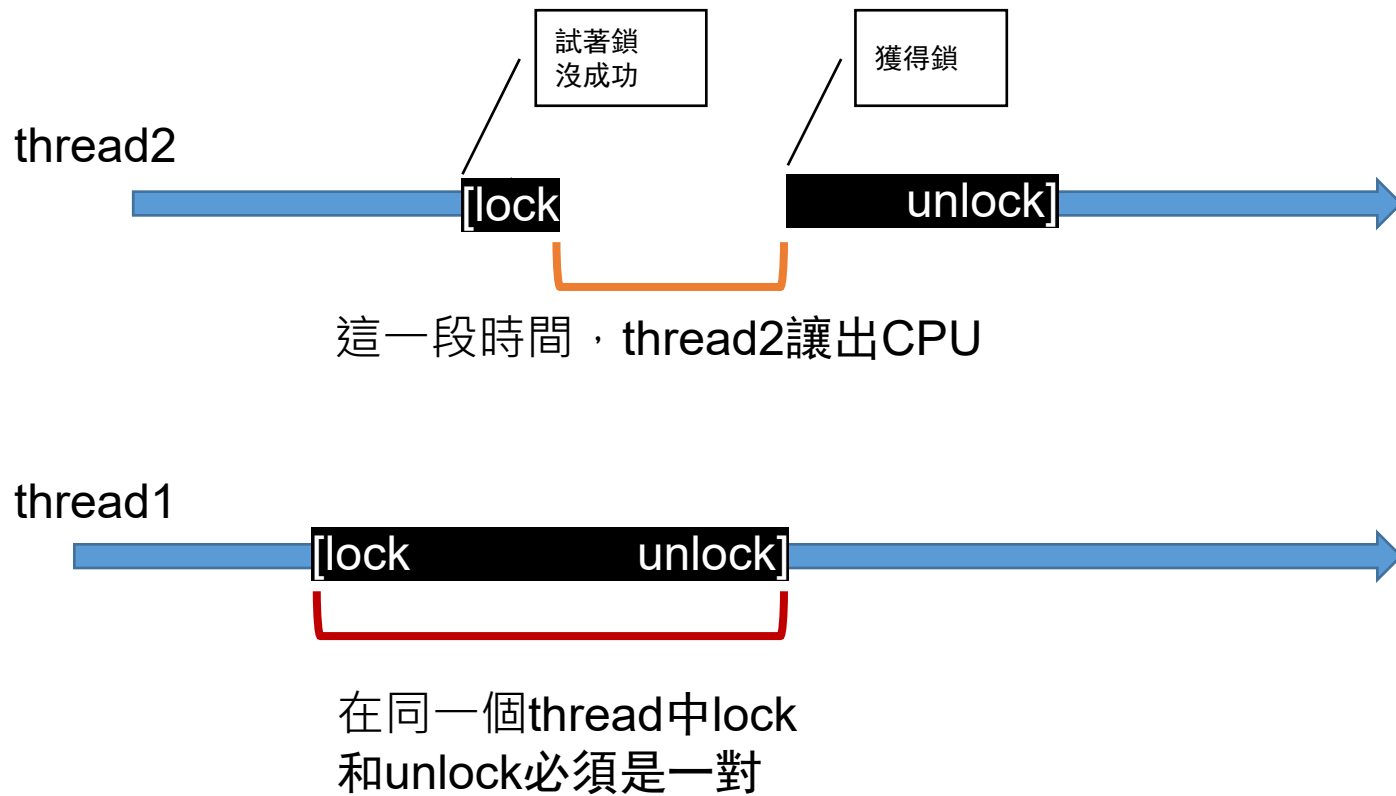


- 只要有「pthread_t pt_id」可以任意的做 join(pt_id)

semaphore & mutex – 「sleep waiting」

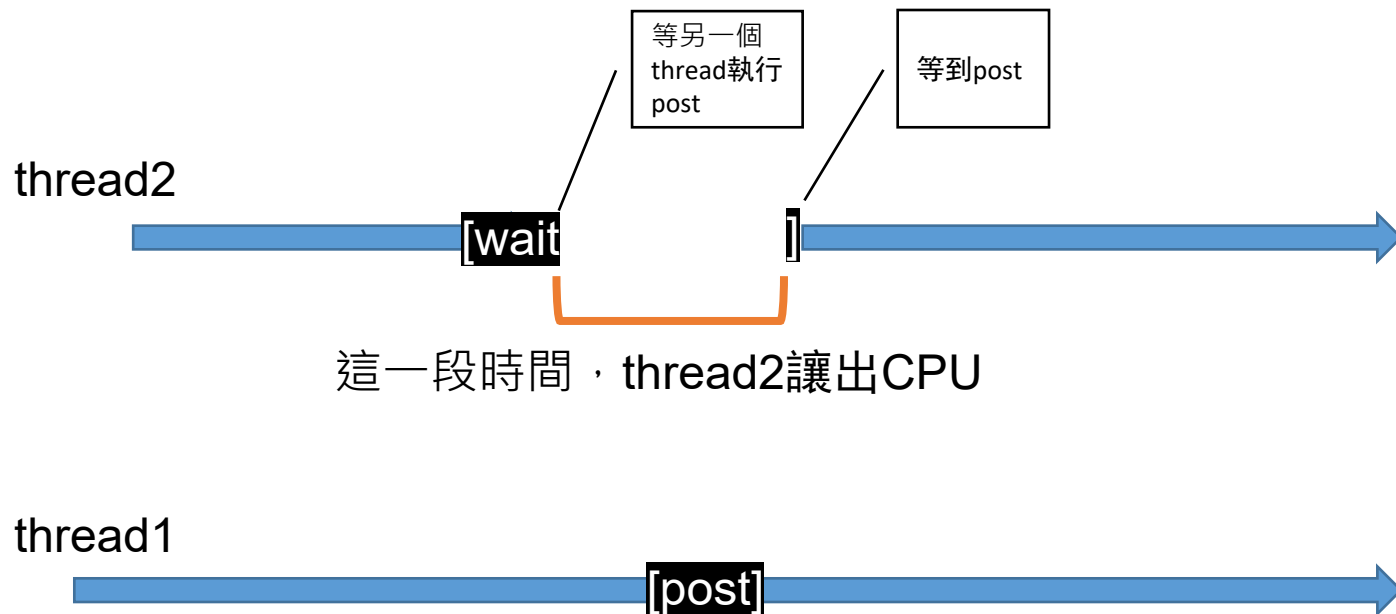


semaphore & mutex – 「sleep waiting」



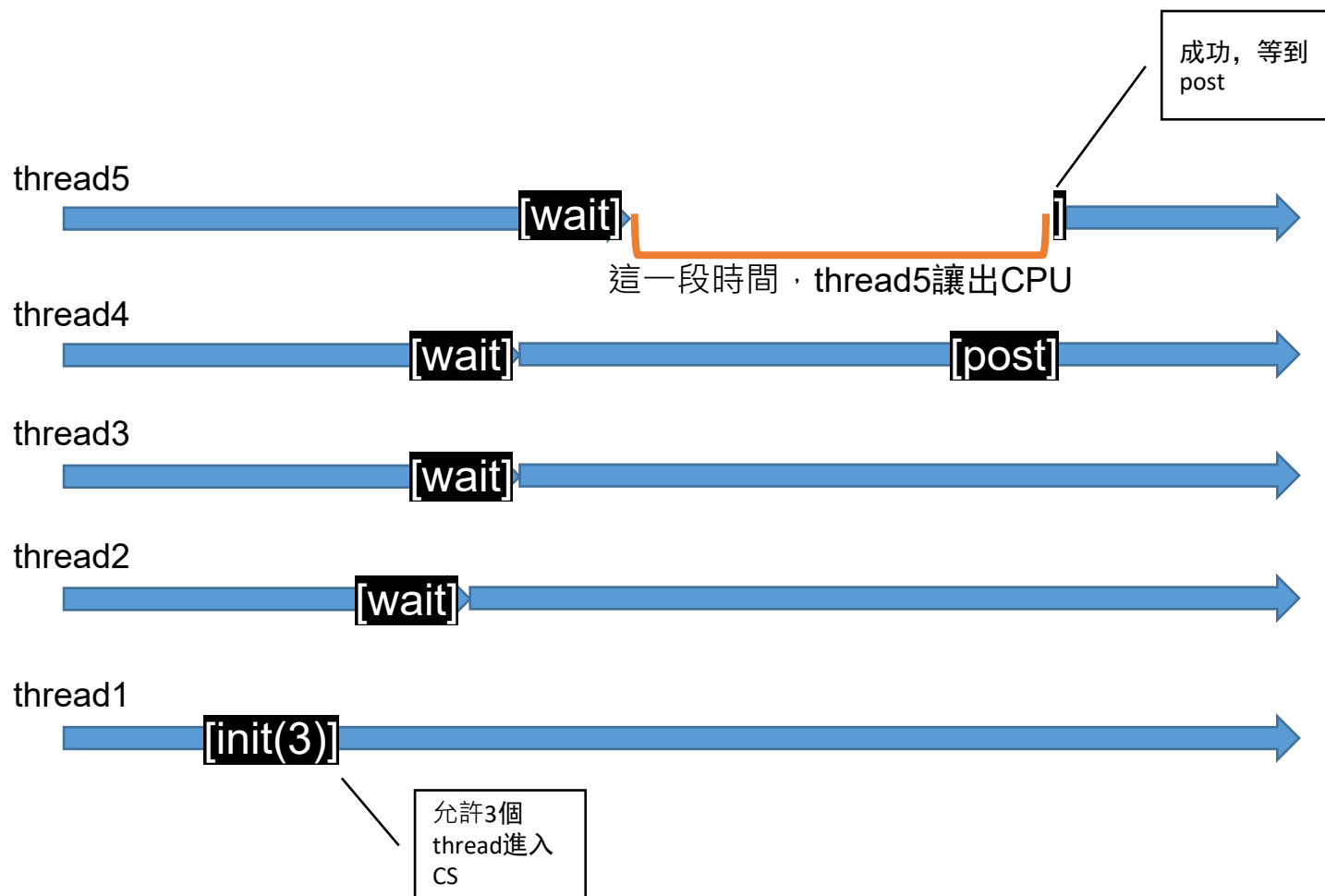
- mutex在一個thread裡面，必須成對，[lock...unlock]
- 如果lock不成功，該thread會讓出CPU給其他task執行

semaphore & mutex – 「sleep waiting」



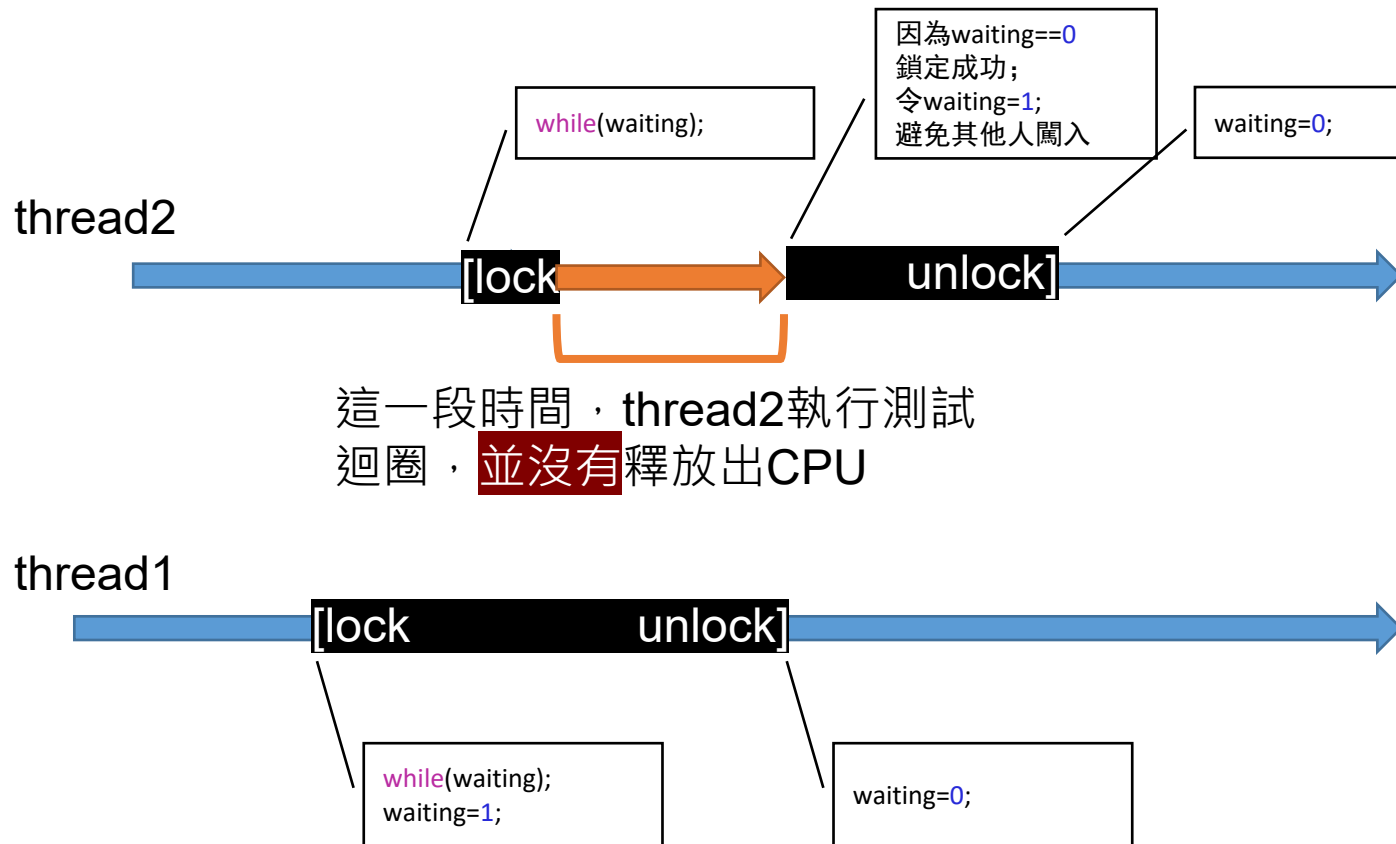
- Semaphore在一個thread中不一定要成對出現，例如左例：thread2 等 thread1執行 post
- wait期間，thread2會讓出CPU

semaphore & mutex – 「sleep waiting」





- Semaphore可以給初始值, 初始值可以是任何整數, 包含0
- 以左圖為例, `init(3)`代表可以允許3個thread進入CS, 第4個thread想進入CS時, 必須稍候 (釋放出CPU), 等有人離開 (`post`) 才可進入CS

spinlock - 「busy waiting」



- Spinlock在用法上很像mutex，其時做方法是用`while(waiting);`不斷的對`waiting`進行測試，直到`waiting==0`
- 如左圖，在測試期間，thread2並沒有釋放出CPU給其他人執行

比較

	context switch (釋放出CPU，切換到其他 process/thread處理)	while(waiting);
busy waiting (如： spinlock)		 隨著等待時間越長 overhead越大 適用於CS比較小的情況
sleep waiting (如： mutex、 semaphore)	 Context-switch的overhead是定值 因此適用於CS比較長的情況 或者要等比較久的情況	

範例程式： signal-wait_mutex.c

```
1.  void thread1(void) {
2.      //喚醒thead2//post
3.      int ret = pthread_mutex_unlock(&mutex);
4.      if (ret != 0) {
5.          errno=ret; //蠻特別的，要自己設定errno
6.          perror("unlock");
7.      }
8.  }
9.  void thread2(void) {
10.     int ret = pthread_mutex_lock(&mutex);
11.     ret = pthread_mutex_lock(&mutex);
12.     if (ret != 0) { //thread1等待thread2//wait
13.         errno=ret; //蠻特別的，要自己設定errno
14.         perror("lock");
15.     }
16. }
```

```
16. int main(void) {
17.     pthread_t id1, id2;
18.     pthread_mutexattr_t attr;
19.     pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK);
20.     //如果傳的第二個參數是NULL，程式會卡死
21.     pthread_mutex_init(&mutex, &attr);
22.     pthread_create(&id1, NULL, (void *) thread1, NULL);
23.     pthread_create(&id2, NULL, (void *) thread2, NULL);
24.     pthread_join(id1, NULL);
25.     pthread_join(id2, NULL);
26. }
```

執行結果

```
$ ./signal-wait_mutex  
unlock: Operation not permitted  
lock: Resource deadlock avoided
```

範例程式： signal-wait_semaphore.c

```
1.  void thread1(void) {
2.      printf("thread1: sleep 5 sec.\n");
3.      for (int i=0; i<5; i++){
4.          fprintf(stderr, "%d ", i);
5.          sleep(1);
6.      }
7.      printf("5\n");
8.      printf("thread1: post a signal.\n");
9.      int ret = sem_post(&semaphores); //喚醒thread2//post
10. }
11. void thread2(void) {
12.     printf("thread2: wait for thread1...\n");
13.     int ret = sem_wait(&semaphores);
14.     printf("thread2: continue\n");
15. }

16. int main(void) {
17.     pthread_t id1, id2;
18.     sem_init(&semaphores, 0, 0);
19.     pthread_create(&id1, NULL, (void *) thread1, NULL);
20.     pthread_create(&id2, NULL, (void *) thread2, NULL);
21.     pthread_join(id1, NULL);
22.     pthread_join(id2, NULL);
23. }
```

執行結果

```
$ ./timedetail ./signal-wait_semaphore
```

```
thread1: sleep 10 sec.
```

```
0 thread2: wait for thread1...
```

```
1 2 3 4 5 6 7 8 9 10
```

```
thread1: post a signal.
```

```
thread2: continue
```

```
經過時間: 10.3609753s
```

```
CPU花在執行程式的時間: 0.002329s
```

```
CPU於usr mode執行此程式所花的時間: 0.002329s
```

```
CPU於kr1 mode執行此程式所花的時間: 0.000000s
```

```
Page fault, 但沒有造成I/O: 77
```

```
Page fault, 並且觸發I/O: 0
```

```
自願性的context switch: 14
```

```
非自願性的context switch: 0
```

範例程式： signal-wait_spinlock.c

```
1.  void thread1(void) {
2.      pthread_spin_lock(&spinlock);
3.      printf("thread1: sleep 10 sec.\n");
4.      for (int i=0; i<10; i++){
5.          fprintf(stderr, "%d ", i);
6.          sleep(1);
7.      }
8.      printf("10\n");
9.      printf("thread1: post a signal.\n");
10.     pthread_spin_unlock(&spinlock); //喚醒thead2//post
11. }
12. void thread2(void) {
13.     printf("thread2: wait for thread1...\n");
14.     pthread_spin_lock(&spinlock);
15.     printf("thread2: continue\n");
16.     pthread_spin_unlock(&spinlock);
17. }

18. int main(void) {
19.     pthread_t id1, id2;
20.     pthread_spin_init(&spinlock, PTHREAD_PROCESS_PRIVATE);
21.     pthread_create(&id1, NULL, (void *) thread1, NULL);
22.     pthread_create(&id2, NULL, (void *) thread2, NULL);
23.     pthread_join(id1, NULL);
24.     pthread_join(id2, NULL);
25. }
```

執行結果

```
$ ./timedetail ./signal-wait_spinlock
thread1: sleep 10 sec.
0 thread2: wait for thread1...
1 2 3 4 5 6 7 8 9 10
thread1: post a signal.
thread2: continue
```

經過時間:	10.2764267s	10.3609753s
CPU花在執行程式的時間:	10.002828s	0.002329s
CPU於usr mode執行此程式所花的時間:	10.002828s	0.002329s
CPU於kr1 mode執行此程式所花的時間:	0.000000s	0.000000s
Page fault，但沒有造成I/O:	79	77
Page fault，並且觸發I/O:	0	0
自願性的context switch:	13	14
非自願性的context switch:	13	0

semaphore

adaptive mutex

p和q競爭mutex, p想要這個mutex (即lock) , 討論p的情況

🍎 如果mutex未上鎖, p獲得鎖

🍎 如果上鎖

♣️ q在另外一顆處理器, 且q在OS的waiting queue (例如: 正在讀資料) , 則p進入sleep的狀態等待mutex (即context-switch)

♣️ q在另外一顆處理器, 且q不在waiting queue (表示q正在運算) , 則p採用busy waiting的方式等待mutex

♣️ q和p在同一顆處理器上, 則p進入sleep的狀態等待mutex (即context-switch)

signal-wait_adaptive_mutex.c

```
1.  void p(void* para) {
2.      for (int i=0; i< 100; i++) {
3.          pthread_mutex_lock(&mutex);
4.          sem_post(&semaphore1);
5.          if(gotoSleep) usleep(1);
6.          pthread_mutex_unlock(&mutex);
7.          sem_wait(&semaphore2);
8.      } }
9.  void q(void* para) {
10.     for(int i=0; i< 100; i++) {
11.         sem_wait(&semaphore1);
12.         pthread_mutex_lock(&mutex);
13.         pthread_mutex_unlock(&mutex);
14.         sem_post(&semaphore2);
15.     } }

16. int main(int argc, char** argv) {
17.     sem_init(&semaphore1, 0, 0);
18.     sem_init(&semaphore2, 0, 0);
19.     pthread_mutexattr_settype(&attr,
20.         PTHREAD_MUTEX_ADAPTIVE_NP);
21.     pthread_mutex_init(&mutex, &attr);
22.     pthread_create(&id1, NULL, (void *) p, NULL);
23.     pthread_create(&id2, NULL, (void *) q, NULL);
24. }
```

執行結果

沒有nanosleep()

```
$ ./timedetail ./signal-wait_adptive_mutex
經過時間: 0.5033064s
CPU花在執行程式的時間: 0.005729s
CPU於usr mode執行此程式所花的時間: 0.005729s
CPU於krl mode執行此程式所花的時間: 0.000000s
Page fault, 但沒有造成I/O: 74
Page fault, 並且觸發I/O: 0
自願性的context switch: 201
非自願性的context switch: 0
```

有nanosleep

```
./timedetail ./signal-wait_adptive_mutex sleep
經過時間: 0.10627179s
CPU花在執行程式的時間: 0.008580s
CPU於usr mode執行此程式所花的時間: 0.008580s
CPU於krl mode執行此程式所花的時間: 0.000000s
Page fault, 但沒有造成I/O: 72
Page fault, 並且觸發I/O: 0
自願性的context switch: 400
非自願性的context switch: 0
```

結果討論

- 🍏 使用adaptive mutex時，如果p執行nanosleep，那麼q就會做context-switch，因此context-switch的數量為400
- 🍏 如果p沒有執行nanosleep，那麼q就不會做context-switch，因此context-switch數量為201（比較少）

關於thread id

1. `pthread_t ptid;`
2. `int tid;`
3. `/*獲得方式*/`
4. `ptid = pthread_create(...);`
5. `tid = gettid() { return syscall(SYS_gettid);}`

使用方式：

- 🍏 與pthread相關的函數使用ptid（例如：join()）
- 🍏 與Linux相關的函數使用tid（例如：gdb）

Thread local variable

- 🍏 宣告變數時，如果加上 `__thread`，那麼這個compiler會對每一個thread宣告這個變數
- 🍏 通常用於全域變數
- 🍏 實作方式：在Linux x64中，每一個thread的gs/fs (32/64) 暫存器指向thread local storage
- 🍏 範例：

```
__thread int thread_local = 0;
```

__thread.c

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #include <semaphore.h>
4.
5.  volatile int global=0;
6.  pthread_mutex_t mutex;
7.  __thread int thread_local = 0;
8.
9.  void thread(void) {
10.     int i;
11.     for (i=0; i<1000000000; i++)
12.         thread_local+=1;
13.     printf("@thread_local = %p\n", &thread_local);
14.
15.     pthread_mutex_lock(&mutex);
16.     global+=thread_local;
17.     pthread_mutex_unlock(&mutex);
18. }
```

__thread.c

```
17.  int main(void) {
18.      pthread_t id1, id2;
19.      pthread_mutex_init(&mutex, NULL);
20.      pthread_create(&id1, NULL, (void *) thread, NULL);
21.      pthread_create(&id2, NULL, (void *) thread, NULL);
22.      pthread_join(id1, NULL);
23.      pthread_join(id2, NULL);
24.      printf("1000000000+1000000000 = %d\n", global);
25.      return (0);
26. }
```


pthread的重點

- 🍏 仔細了解算出pi的程式碼
- 🍏 想一下如何更快速的算出pi



本學期上到這裡

取消執行緒的執行

🍏 `int pthread_cancel(pthread_t thread);`

🍏 會立即返回 (non-blocking)

🍏 向目標執行緒送出SIGCANCEL

🍀 SIGCANCEL可以ignore

取消執行緒的執行

🍏 `int pthread_setcancelstate(int state, int *oldstate);`

🍀 `PTHREAD_CANCEL_ENABLE`

🍇 允許 `pthread_cancel`

🍀 `PTHREAD_CANCEL_DISABLE`

🍇 忽略 `pthread_cancel`

🍏 `int pthread_setcanceltype(int type, int *oldtype);`

🍀 `PTHREAD_CANCEL_DEFERRED`

🍇 在適當的時間點，例如某些函數（如：`pthread_testcancel`）檢查是否取消

🍀 `PTHREAD_CANCEL_ASYNCHRONOUS`

🍇 立即取消該執行緒的執行（十分危險的做法）

戰場的清理

🍏 `void pthread_cleanup_push(void (*routine)(void *), void *arg);`

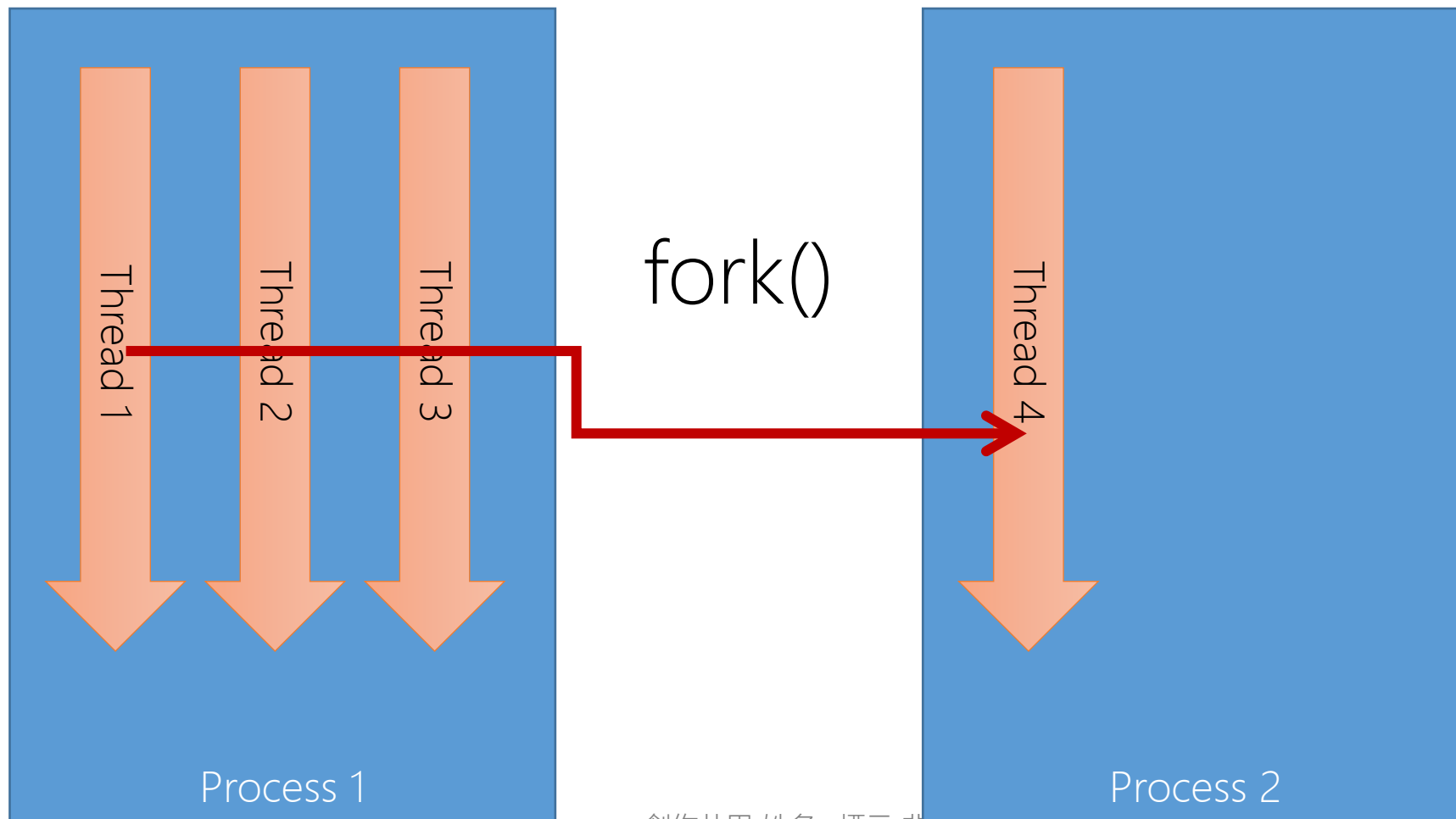
🍏 `void pthread_cleanup_pop(int execute);`

🍏 呼叫push將routine放入堆疊中，執行緒執行結束時會呼叫該routine

🍏 呼叫pop時，如果參數不為零，則會執行pop出來的函數

🍏 詳情請見： `man pthread_cleanup_push`

thread與fork



thread與fork

- 🍏 多執行緒的process，執行fork以後，新的process內只有「該執行fork的thread」
- 🍏 依照上述語意，執行結果通常「很奇怪」，因此請不要在multi-thread process中執行fork

thread & false sharing

- 🍏 二個以上的執行緒，存取不同的變數。而這些變數卻在於同一個cache line上。
- 🍏 換句話說：表面上存取不同變數，實體上卻是存取同一個資源 (cache line)
- 🍏 這種現象稱之為false sharing或者是ping-pong

pingpong.c

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #include <semaphore.h>
4.
   /*a和b大小只有4個byte，二者合計8個byte*/
5.  struct DS {
6.      int a;
7.      int b;
8.  };
9.
   void thread(void* local) {
10.     int* _local=(int*) local;
11.     int i;
12.     for (i=0; i<1000000000; i++)
13.         *_local+=1;
14. }
```

pingpong.c

```
15. int main(void) {
16.     pthread_t id1, id2;
17.     struct DS ds;
18.     ds.a=0; ds.b=0;
19.     printf("sizeof(DS)=%d\n",(int)sizeof(ds));
20.     pthread_create(&id1,NULL,(void *) thread,&(ds.a));
21.     pthread_create(&id2,NULL,(void *) thread,&(ds.b));
22.     pthread_join(id1,NULL);
23.     pthread_join(id2,NULL);
24.     return (0);
25. }
```

執行結果

```
$ getconf LEVEL1_DCACHE_LINESIZE  
64  
$ time ./pingpong  
sizeof(DS)=8  
  
real  0m5.293s  
user  0m10.560s  
sys   0m0.000s
```

pingpong_aligned.c

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #include <semaphore.h>
4.
5.  struct DS {
6.      __attribute__((aligned(64))) int a;
7.      __attribute__((aligned(64))) int b;
8.  };
9.
10. void thread(void* local) {
11.     int* _local=(int*) local;
12.     int i;
13.     for (i=0; i<1000000000; i++)
14.         *_local+=1;
15. }
```

pingpong_aligned.c

```
14. int main(void) {
15.     pthread_t id1, id2;
16.     struct DS ds;
17.     ds.a=0; ds.b=0;
18.     printf("sizeof(DS)=%d\n",(int)sizeof(ds));
19.     pthread_create(&id1,NULL,(void *) thread,&(ds.a));
20.     pthread_create(&id2,NULL,(void *) thread,&(ds.b));
21.     pthread_join(id1,NULL);
22.     pthread_join(id2,NULL);
23.     return (0);
24. }
```

執行結果

pingpong

```
$ time ./pingpong  
sizeof(DS)=8
```

```
real 0m5.293s  
user 0m10.560s  
sys  0m0.000s
```

pingpong_aligned

```
$ time ./pingpong_aligned  
sizeof(DS)=128
```

```
real 0m2.350s  
user 0m4.688s  
sys  0m0.000s
```

比2job的2.131
秒慢

2job vs. pingpong_aligned

2job

```
//local+=1;  
addl    $0x1, -0x8(%rbp)
```

pingpong_aligned

```
mov     -0x8(%rbp), %rax  
mov     (%rax), %eax  
lea     0x1(%rax), %edx  
mov     -0x8(%rbp), %rax  
mov     %edx, (%rax)
```

小小結論

🍏 對齊過後的程式碼真的蠻快的（小輸2job），但很浪費記憶體空間，在最關鍵的時候使用。

C11 – alignas (pingpong_alignedas.c)

```
1. #include <stdio.h>
2. #include <pthread.h>
3. #include <semaphore.h>
4. #include <stdalign.h>
5. /*c11專用語法*/
6. struct DS {
7.     int alignas(64) a;
8.     int alignas(64) b;
9. };
```

執行結果

```
shiwulo@vm:~/sp/ch12$ time ./pingpong
sizeof(DS)=8
real    0m7.593s
user    0m15.168s
sys     0m0.004s
shiwulo@vm:~/sp/ch12$ time ./pingpong_aligned
sizeof(DS)=128
real    0m2.358s
user    0m4.704s
sys     0m0.000s
shiwulo@vm:~/sp/ch12$ time ./pingpong_alignas
sizeof(DS)=128
real    0m2.373s
user    0m4.728s
sys     0m0.000s
```

反組譯的結果

🍏 就這個程式來說，反組譯以後`pingpong_aligned`和`pingpong_alignas`的組合語言是一致的

C11 – aligned_alloc

🍏 `void *aligned_alloc(size_t alignment, size_t size);`

🍏 alignment: 跟alignment的倍數作對齊

🍏 size: 需要分配的大小

原子運算

- 🍏 某些基本的運算單元（如：int）可以宣告為Atomic_
- 🍏 透過定義在<stdatomic.h>的operator，可以對一些基本的運算單元（如：int進行簡單操作）
- 🍏 請注意：一連串的「原子運算」不會形成「原子運算」

C11 – stdatomic.h

```
1. #include <stdio.h>
2. #include <pthread.h>
3. #include <stdatomic.h>
4. atomic_int global=0;
5. void thread(void) {
6.     int i;
7.     for (i=0; i<10000000000; i++)
8.         atomic_fetch_add(&global, 1);
9. }
```

執行結果

```
$ $ time ./atomic
1000000+1000000 = 2000000000
real    0m43.646s
user    1m27.272s
sys     0m0.000s
$gdb atomic
(gdb) disas /m thread
Dump of assembler code for function thread:
...
10                atomic_fetch_add(&global, 1);
    0x0000000000400703 <+13>:    lock addl $0x1,0x200949(%rip)        # 0x601054 <global>
...
```



2jobs的秒數為2.131s

```
gcc -c -g -Wa,-a,-ad atomic.c > atomic.asm
```

```
10:atomic.c      ****          atomic_fetch_add(&global, 1);  
28              .loc 1 10 0 discriminator 3  
29 000d F0830500  lock addl    $1, global(%rip)
```


小小結論

🍏 硬體的同步化機制（例如：Intel's lock）還是不如手動平行化的速度

小結

- 🍏 瞭解thread local variable
- 🍏 瞭解fork和cancel在thread中的作用，並且知道「盡量不要」使用這二種機制
- 🍏 瞭解false sharing及解決方法
- 🍏 一些基本型別（如：int）有hardware solution（即：atomic operation）

進階Lock機制



綜觀

1. lock-free queue

- ✿ 利用x86上，load和store是atomic operation所設計的concurrent queue
- ✿ 只允許一個producer、一個consumer

2. sequential lock

- ✿ Writer擁有無窮高的優先權（這少見，一般來說都是reader優先權高）
- ✿ 如果reader被writer打斷，reader要重做（redo）

3. ticket lock

- ✿ 具有FIFO功能的spinlock，與spinlock相較複雜度並沒有增加很多

4. r/w spinlock

- ✿ 同上一樣，使用票卷的概念實現，writer一次拿走全部的票卷，reader一次拿一張票券
- ✿ 隱含的，reader的優先權比writer高



(1.) lock-free的 concurrent queue

只支援單個producer、consumer

spin-lock設計的基本技巧

- 🍏 「檢查-進入」這樣的方式是不對的，因為在檢查和進入之間可能會有其他thread同時做「檢查」，其結果是可能多個thread同時進入
- 🍏 常見的spin-lock的寫法是「改變lock 的狀態-檢查-確認是否lock成功」
 - ♣️ 因為先改變lock的狀態，因此其他thread就算同時要進入critical section也都會先做這個動作
 - ♣️ 如果lock失敗，所有thread都重做一次（即：spin），總有一個可以成功
- 🍏 另一個常見的做法是將「檢查和鎖住」寫成一個atomic operation，如：
swap()

回顧：lockfreeQueue.c

```
1.  volatile int in=0, out=0;
2.  void put() {
3.      static int item=1; //用於除錯，放入buffer的資料都是嚴格遞增的數列
4.      while ((in+1)%bufsize == out) ; //busy waiting
5.      buffer[in]=item++; //將資料放入
6.      //這裡應該要加入memory fence，確保增加buffer的狀態（in++）後，get()真的會讀到資料
7.      in = (in + 1)%bufsize; //下一次要放入的位置
8.  }
9.  void get() {
10.     int tmpItem; //讀取的數字暫時放在tmpItem
11.     while (in == out) ; // busy waiting
12.     tmpItem=buffer[out]; //讀出buffer的東西
13.     out = (out + 1)%bufsize; //下一次要拿取的位置
14. }
```

回顧：執行結果

```
$ ./time_detail ./lockfree_buf
```

經過時間：	0.699771642s	2.923294185s
CPU花在執行程式的時間：	1.397830s	5.812117s
CPU於usr mode執行此程式所花的時間：	1.393825s	3.717052s
CPU於krl mode執行此程式所花的時間：	0.004005s	2.095065s
Page fault, 但沒有造成I/O：	76	76
Page fault, 並且觸發I/O：	0	0
自願性的context switch：	2	36898
非自願性的context switch：	1	57

回顧：執行結果分析

程式的正確性：

- ✿ 假設只有一個producer，一個consumer。in, out這二個變數宣告為volatile，確保每次的寫入，真的寫入到memory (/cache)。
- ✿ 基於上述的假設，只有一個thread（即：producer）會對in做寫入。同樣的原理，只有一個thread（即：consumer）會對out做寫入。
- ✿ 由於設定完in, out以後就是function return，因此未使用memory barrier。

執行效率：

- ✿ 由於buffer_sem只允許一個producer、consumer，因此與buffer_sem比較。
- ✿ 採用lockfree的方法比semaphore快上4.17倍

producer, consumer對in, out的存取

	producer	consumer
in	寫入及讀取：程式碼如下 <code>in = (in + 1)%bufsize;</code>	讀取：程式碼如下 <code>while (in == out) ;</code>
out	讀取：程式碼如下 <code>while ((in+1)%bufsize == out) ;</code>	寫入及讀取：程式碼如下 <code>out = (out + 1)%bufsize;</code>

- 這一個程式在producer (put)、consumer (get) 不會有race condition, 原因是：共用變數, in及out, 不會有二個以上的thread, 對其做寫入
- 基於上述理由, 這個方法只適用於一個producer、一個consumer。
- 舉例：如果有二個producer, 那麼in這個變數就可能有2個thread (因為有二個producer) 對其修



(2.) sequential lock

讓reader付出成本的reader-writer solution
這個版本的seqlock只可以有一個writer

演算法概念

共用變數：version，初始值為0

writer

version++; //lock, version變成奇數
開始讀寫
讀寫完應該加入memory fence
確保reader看到的是完整的發佈
version++; //unlock, version變成偶數

reader

redo:
localversion=version;
if (locaversion == 奇數)
 //writer正在寫
 goto redo;
開始讀取
if(localVersion != version)
 //讀取的過程中, writer寫入,
 //造成version改變
 goto redo;

這一段程式碼相當於lock，但lock成功不代表一定讀到正確的值

這一段程式碼相當於unlock，unlock之前，必須確保讀取的值是正確的

seqlock.c

```
1. void seq_wrt_lock() { atomic_fetch_add(&g_version, 1); }
2. void seq_wrt_unlock() { atomic_fetch_add(&g_version, 1); }
3. void rd_thread(void* para) {
4.     while(1) { //reader的主迴圈
5.         //下面程式碼是為了要讀取資料，如果讀取的過程當中，writer更新了資料，這時候reader就重新讀
6.         while (1) { { //這個迴圈是為了要redo，相當於寫：label redo;
7.             int local_version;
8.             //有writer正在critical section，再試一次
9.             if ((local_version=atomic_load(&g_version))%2==1)
10.                continue;
11.            //底下這個for loop要置換成讀取資料的程式碼
12.            for (int i=0; i< 100; i++); //模擬讀取資料
13.            //判斷要不要重新讀取
14.            if (local_version == atomic_load(&g_version))
15.                break; //版本編號一樣，表示讀取過程當中，writer沒有更新資料，不用重讀
16.            else continue; //版本編號不一樣，讀取過程中writer更新資料，reader重新讀一次
17.        } }
```


是否可以將atomic operation都拿掉？

- 🍏 二邊共用的變數是version，只有writer會更改version，而且系統當中只有一個writer
 - ♣️ 應該可以，但要確保二個version++依序執行，資料的修改「匡在」這二個version++之間，可以使用memory fence
- 🍏 如果系統當中有多個writer，writer之間必須額外使用spinlock確保一次只有一個writer進入critical section
 - ♣️ 在這個情況下，version應該要用atomic operation進行操作

執行結果

```
$ ./timedetail ./seqlock
```

建立 3 個 reader threads, 1 個 writer thread

reader在CS的數量 3

reader的redo的數量 26895/sec

reader的noredo的數量 9360/sec

writer的寫入數量 17213/sec



(3.) ticket lock

具有FIFO功能的spinlock，其overhead與spinlock差不多

演算法概念

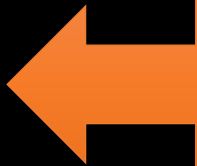
- 🍏 每一個人想進入CS前先抽一張號碼牌
- 🍏 然後等「叫號」，如果叫到的號碼跟自己的號碼牌一樣，就進入CS
- 🍏 離開CS的時候，做「叫號」，即讓下一個號碼的人進入CS

ticketlock.c

```
1.  struct ticketlock_t {
2.      volatile atomic_int next_ticket; //一定要加上volatile, 否則-O3會出錯, 初始值為0
3.      //一定要加上volatile, 否則-O3會出錯, 這一個不用是atomic_int, 因為執行時它已經在CS, 初始值為0
4.      volatile int now_serving;
5.  };
6.  void ticketLock_acquire(volatile atomic_int* next_ticket, volatile int* now_serving){
7.      int my_ticket;
8.      //抽號碼牌, 使用atomic operation, 避免二個人抽到同一張號碼
9.      my_ticket = atomic_fetch_add(next_ticket, 1);
10.     while(*now_serving != my_ticket); //等叫號, 如果叫到的號碼 (now_serving) 與自己一樣就進入CS
11. }
12. void ticketLock_release(volatile int *now_serving) {
13.     ++*now_serving; //離開CS前, 將「叫號」+1, 讓下一個人可以進來
14.     //如果要讓其他thread看到更新的資料結構, 這裡要加入memory fence
15. }
```

執行結果

```
$ ./ticketlock
thread_0 wait for entering CS
thread_1 in CS
thread_3 wait for entering CS
thread_2 in CS
thread_1 wait for entering CS
thread_2 wait for entering CS
thread_0 in CS
thread_3 in CS
thread_0 wait for entering CS
thread_1 in CS
thread_3 wait for entering CS
thread_2 in CS
thread_1 wait for entering CS
thread_2 wait for entering CS
```



可以觀察到，
的確是按照
FIFO順序進入
CS



(4.) rw-spinlock

如果系統中reader的數量非常多，writer可能會starvation

演算法概念

- 🍏 系統中有很多張門票（例如：1000張）
 - ♣️ writer要進入CS，一次要拿到1000張，這確保了其他writer、reader都不能再進入CS
 - ♣️ reader要進入CS，一次拿一張門票
 - 🍇 因為已經拿了一張門票，因此writer不可能拿到全部的門票
 - 🍇 系統有很多張門票，而每一個reader只要一張門票，因此可以有多個reader同時進入CS
- 🍏 在實作的時候不要把「最大門票」設定的太大（例如：不可以設定成INT_MAX），因為在實作的時候，writer會先試著拿門票，這個步驟會將現有門票減「最大門票」值，如果多個writer同時進入CS，可能會造成underflow

rw-spinlock.c

```
1. void init_rwlock() {
2.     atomic_store(&nTicket, maxTicket);
3. }
4. void wrt_lock() {
5.     while(1) {
6.         atomic_fetch_sub(&nTicket, maxTicket); //先改變鎖的狀態，避免race condition
7.
8.         int ret=atomic_load(&nTicket);
9.         if (ret == 0) return;
10.        else atomic_fetch_add(&nTicket, maxTicket); //lock沒有成功 (ret<0) , 代表有writer或reader正在CS，將門票加回去
11.    }
12. }
13. void rd_lock() {
14.     while(1) {
15.         atomic_fetch_sub(&nTicket, 1); //先拿一張門票
16.         int ret = atomic_load(&nTicket);
17.         if (ret > 0) return; //拿走一張以後，只要ret>0代表： 1. 自己是第一個進入CS的reader；或 2. 已經有reader在裡面
18.         else atomic_fetch_add(&nTicket, 1); //如果拿一張門票以後，門票總數<0，代表有writer正在CS
19.     }
20. }
21. void rd_unlock() { atomic_fetch_add(&nTicket, 1); } //還門票
22. void wrt_unlock() { atomic_fetch_add(&nTicket, maxTicket); } //還門票
```

執行結果

```
$ ./rwspinlock
```

```
建立4個reader
```

```
建立4個writer
```

```
按下Enter鍵繼續
```

```
nTicket = -2003      /*有3個reader在CS，有2個writer嘗試進入CS*/
```

```
total number of reader entering CS = 184937/sec
```

```
total number of writer entering CS = 29118/sec
```

```
parallel level of read = 1.944711 /*reader的平行度為1.944，代表只要有reader在CS，通常是1.944個reader在CS*/
```

```
nTicket = -1004
```

```
total number of reader entering CS = 187014/sec
```

```
total number of writer entering CS = 29400/sec
```

```
parallel level of read = 1.944365
```

```
nTicket = 998        /*有2個reader在CS，因此門票剩下998張*/
```

```
total number of reader entering CS = 187033/sec
```

```
total number of writer entering CS = 29383/sec
```

```
parallel level of read = 1.942579
```


討論

- 🍎 sequential lock、ticket lock、r/w spinlock，在程式碼裡面都有spinlock的影子
- 🍎 如果拿到lock的task剛好被Linux kernel踢出CPU，那麼其他人會等很久，而且是做一個無聊的測試迴圈
- 🍎 如前面所說，測試迴圈只會影響該顆core，不會造成額外的memory traffic
 - ♣️ 例如：執行完成的thread可以執行spinlock測試其他thread是否執行到特定段落
 - ♣️ 例如：如果需要等待n個thread，可以使用長度為n的陣列，每一個元素對應該thread是否執行到的特定段落，已經執行完成的thread測試該陣列所有的thread是否將其對應的flag設定為1。（避免使用atomic operation）
- 🍎 測試迴圈內最好加入：`asm("pause")`
 - ♣️ C11、C++11沒有提供pause類似的函數，因此要寫組合語言
- 🍎 指標型別在C11、C++11中也可以是atomic load、store，因此可以善用指標做資料的更新

討論

- 🍏 如果想要在lock不成功的時候，觸發context switch，可以使用futex，futex可以支援“priority inheritance”
- 🍏 例如：POSIX並沒有提供具有FIFO功能的rw-semaphore，如果要達到這個功能，只能自己撰寫
- 🍏 了解這些演算法的特型以後，如果沒有做特殊的修改，盡量上網找已經通過時間驗證的程式碼



memory order

x86 memory ordering

“A **strong hardware memory model** is one in which every machine instruction comes **implicitly with acquire and release** semantics. As a result, when one CPU core performs a sequence of writes, every other CPU core sees those values change in the same order that they were written.”

“**x86 still keeps its memory interactions in-order, so in a multicore environment**, we can still consider it strongly-ordered.”

在x86下考慮C11 & C++11的記憶體模型

- 🍏 Intel x86的硬體「幾乎」滿足memory_order_seq_cst
- 🍏 但compiler不一定滿足memory_order_seq_cst

C11、C++11中memory_order的定義

```
#include <stdatomic.h>
```

```
typedef enum memory_order {
```

```
    memory_order_relaxed,  
    memory_order_consume,  
    memory_order_acquire,  
    memory_order_release,  
    memory_order_acq_rel,  
    memory_order_seq_cst
```

```
} memory_order;
```

```
/*example*/
```

```
atomic_load_explicit( const volatile A* obj, memory_order order );
```

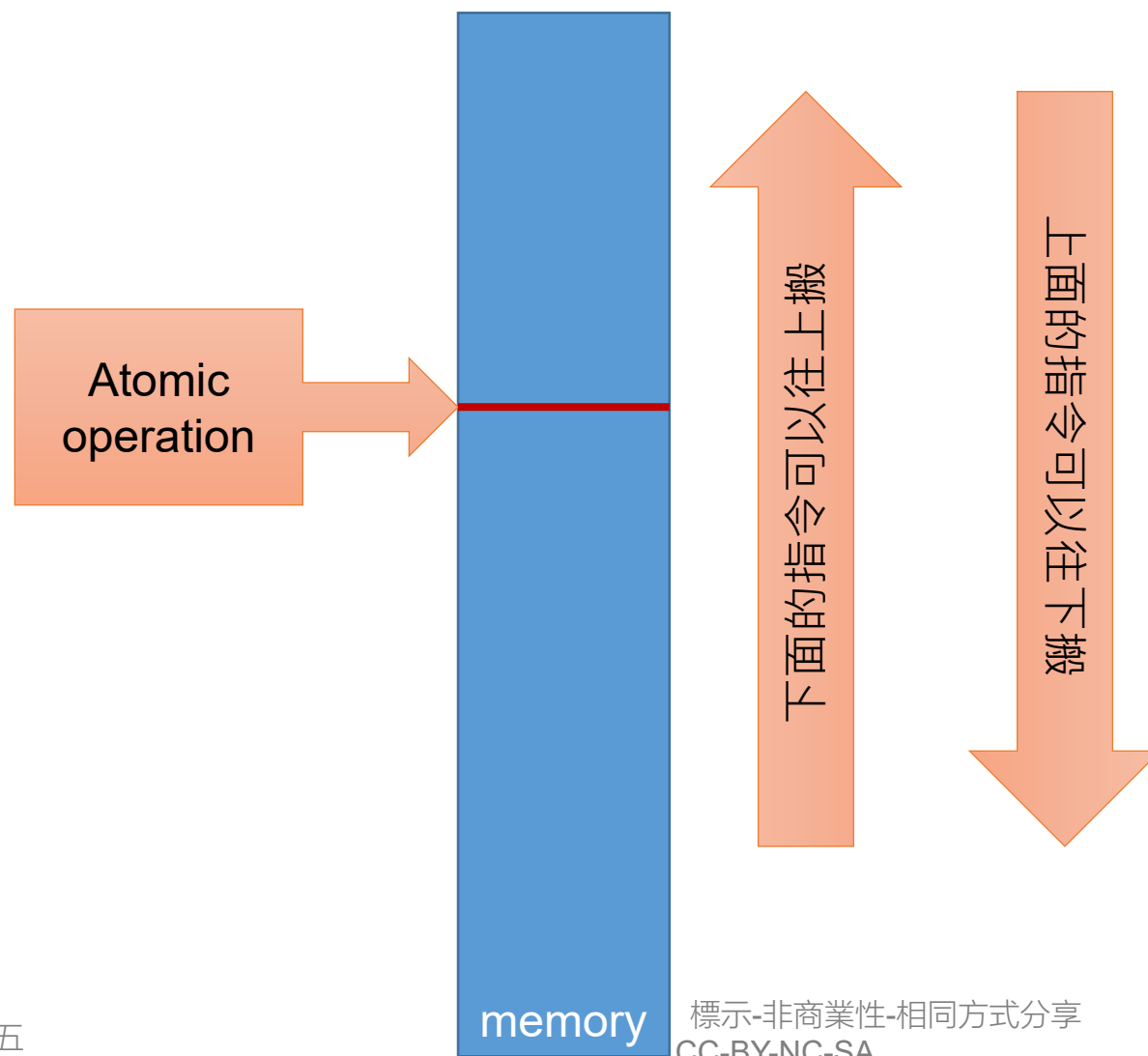
```
atomic_fetch_add_explicit( volatile A* obj, M arg, memory_order order );
```

```
/*下列二者意思相同*/
```

```
atomic_fetch_add( volatile A* obj, M arg );
```

```
atomic_fetch_add_explicit( volatile A* obj, M arg, memory_order_seq_cst );
```

relaxed

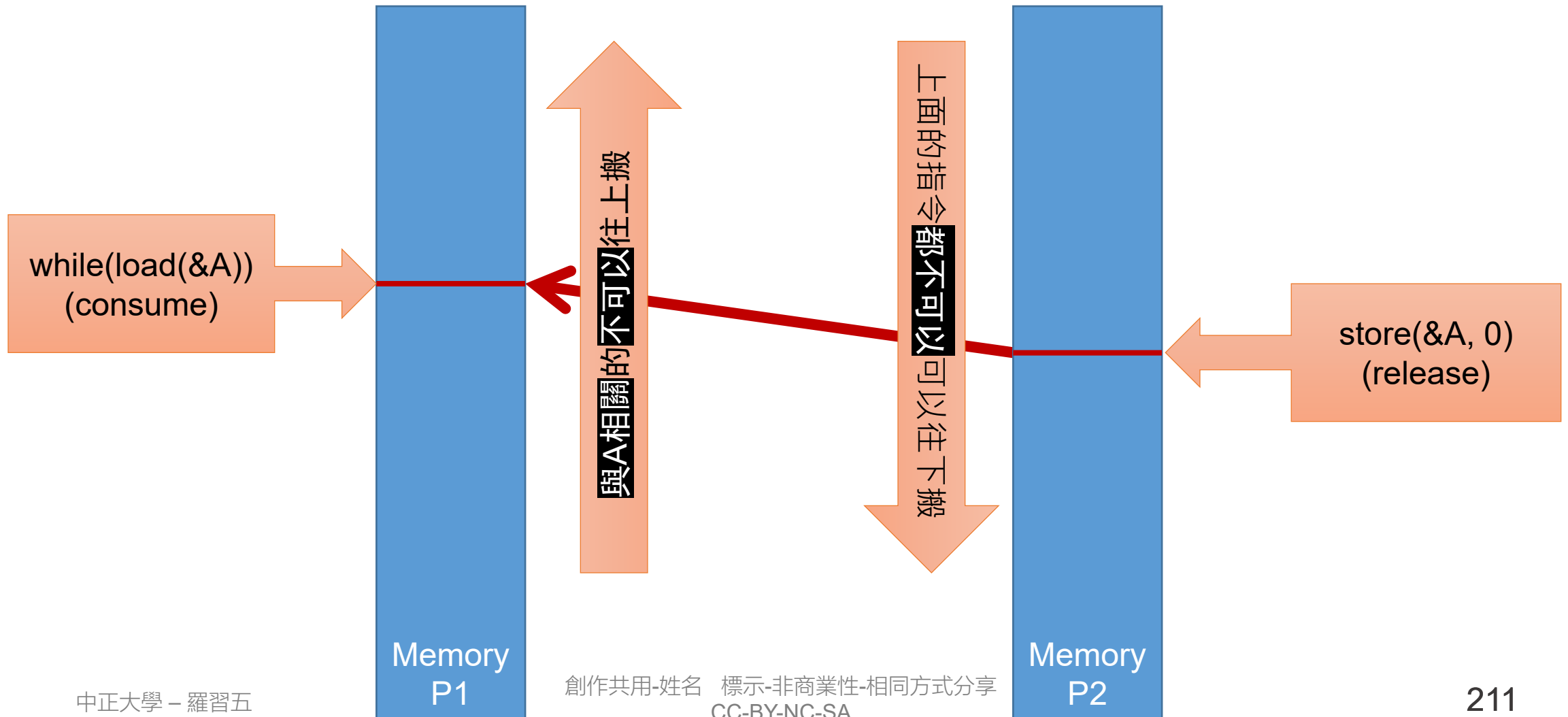


- 因此related只確保該指令是atomic operation

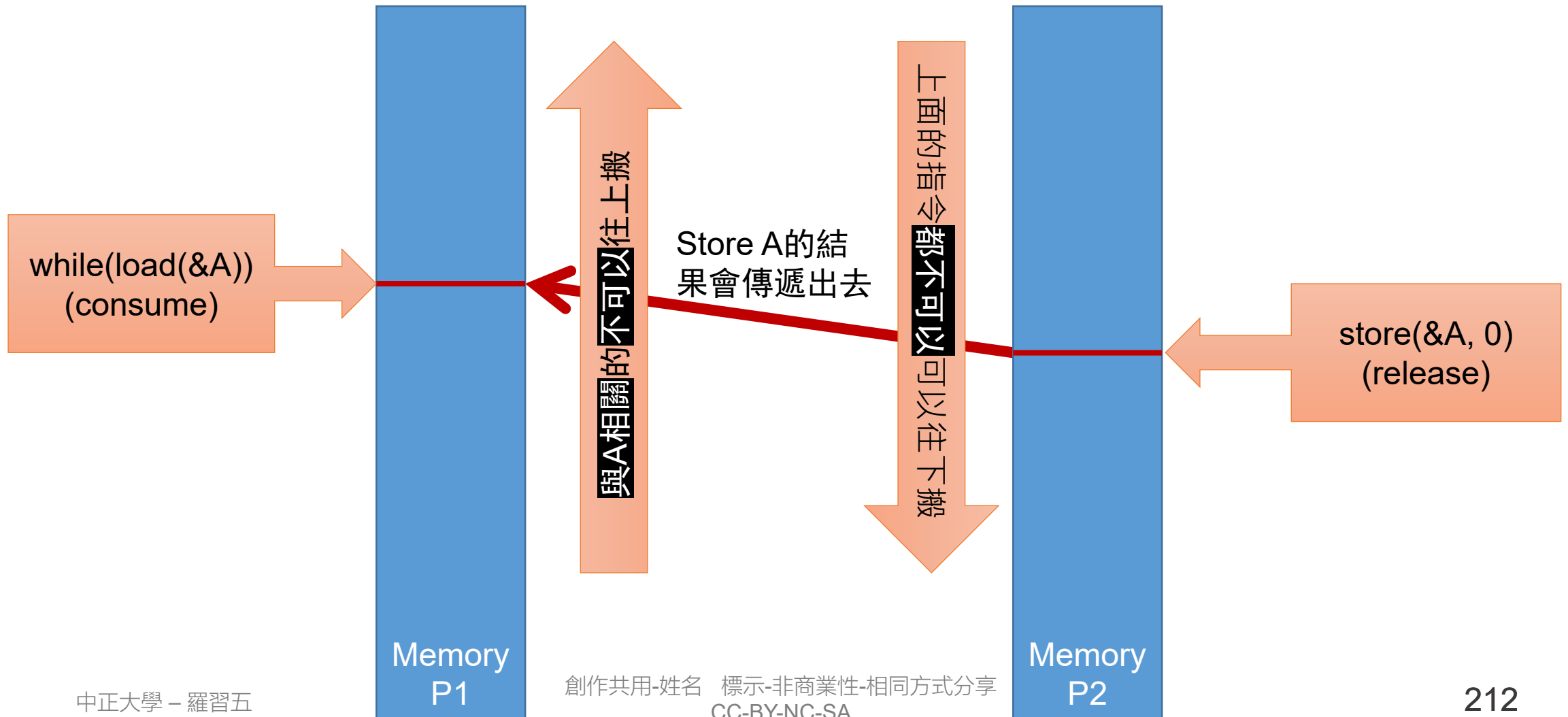
relaxed

- 🍏 應用情境：
- 🍀 如果一連串的atomic operation，頭幾個可以使用relaxed，最後一個再使用有memory fence的指令

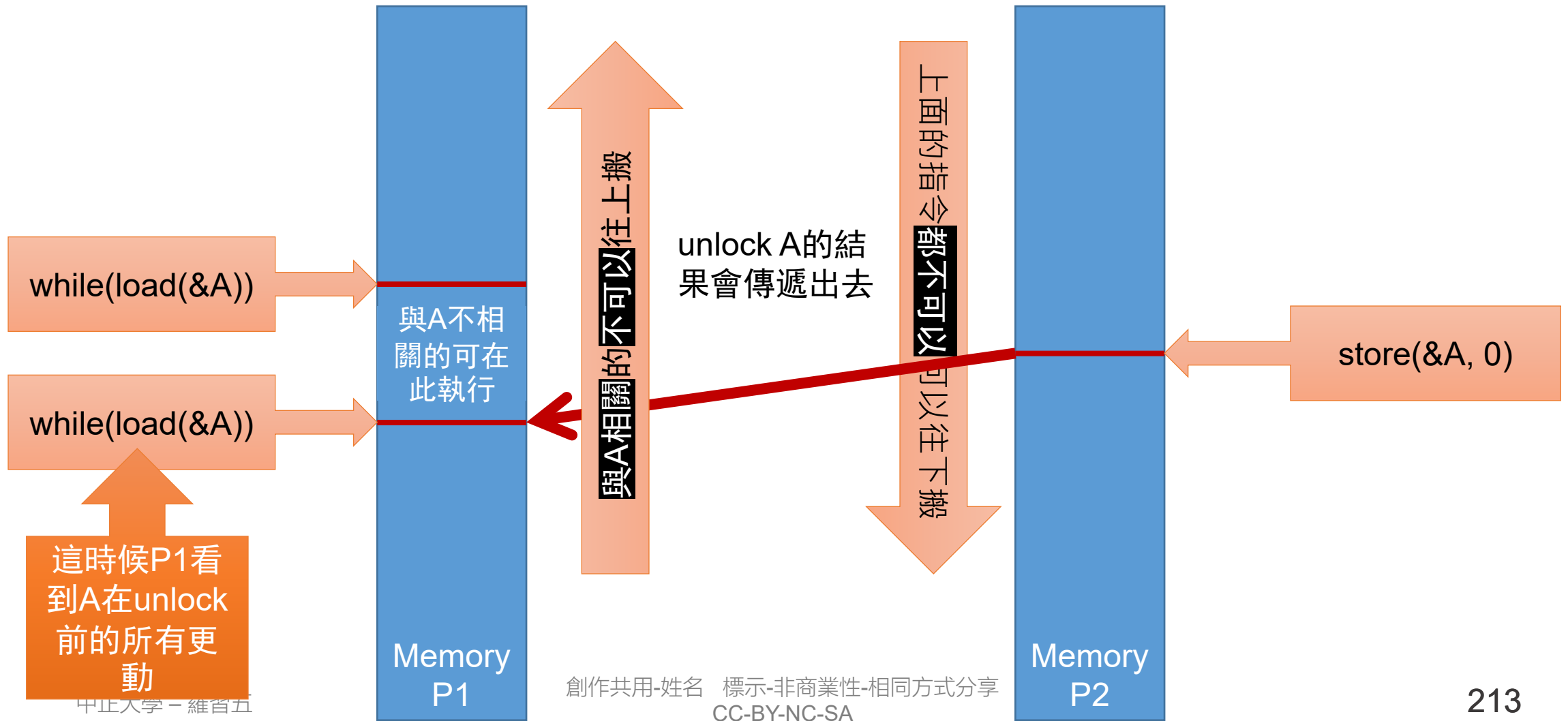
consume & release



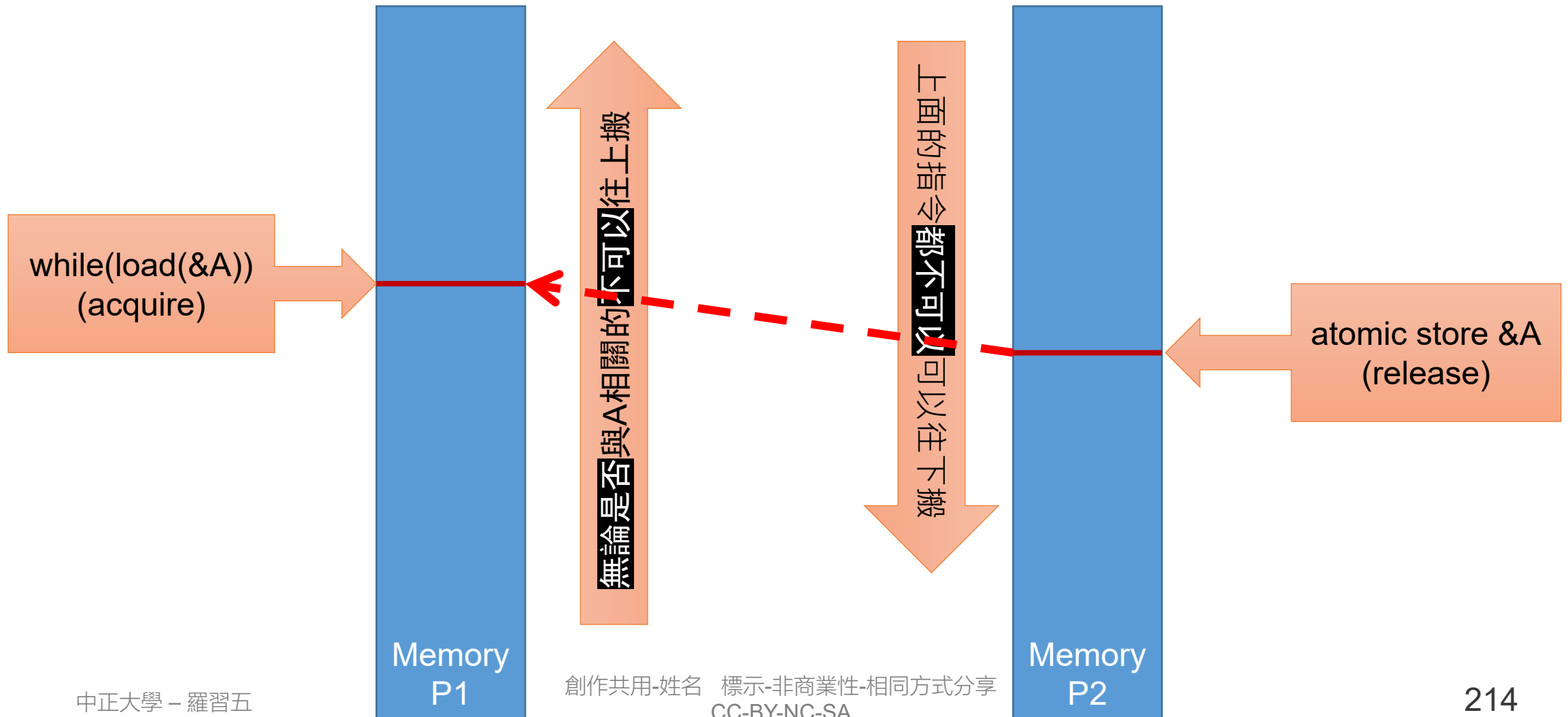
consume & release



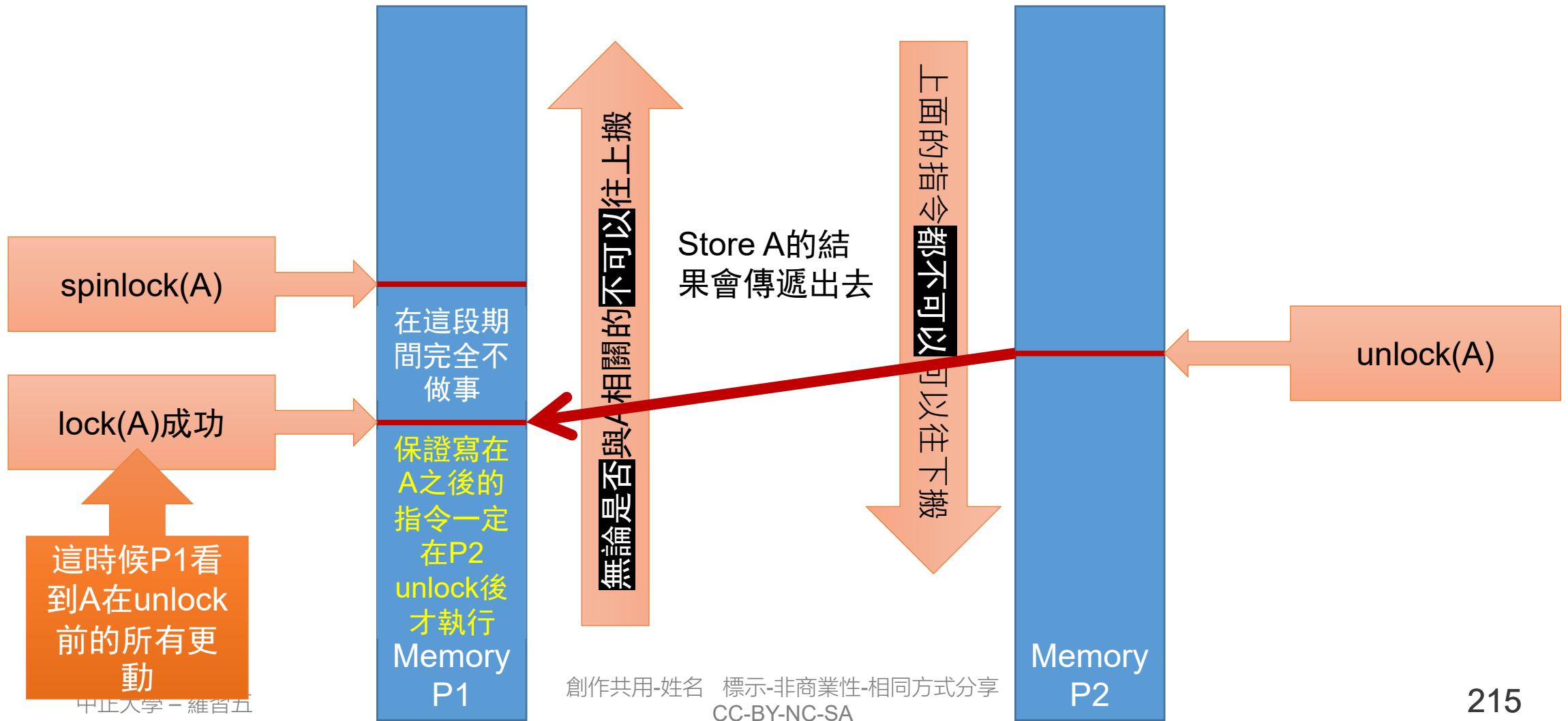
consume & release, 應用



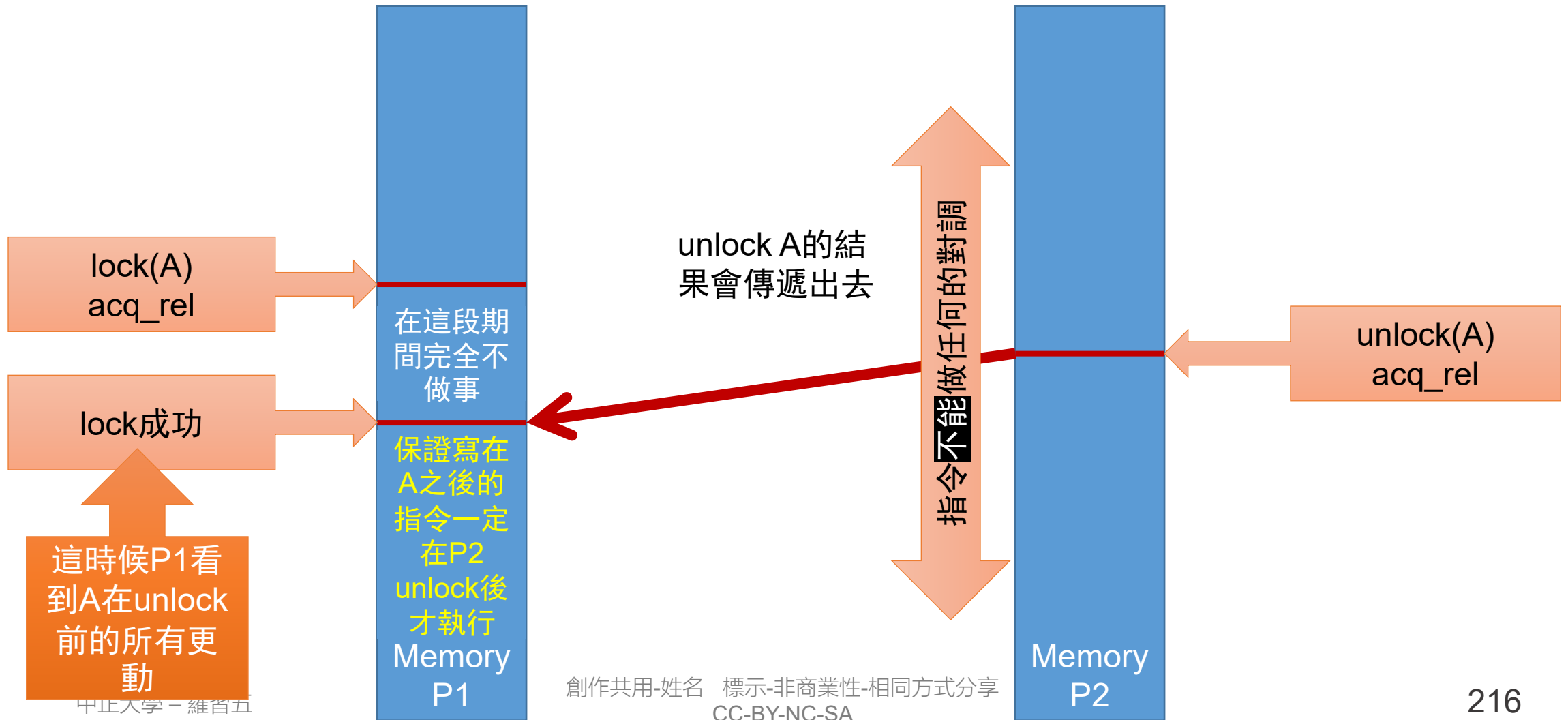
acquire & release



acquire & release, 應用



acq_rel (不是acquire & release)



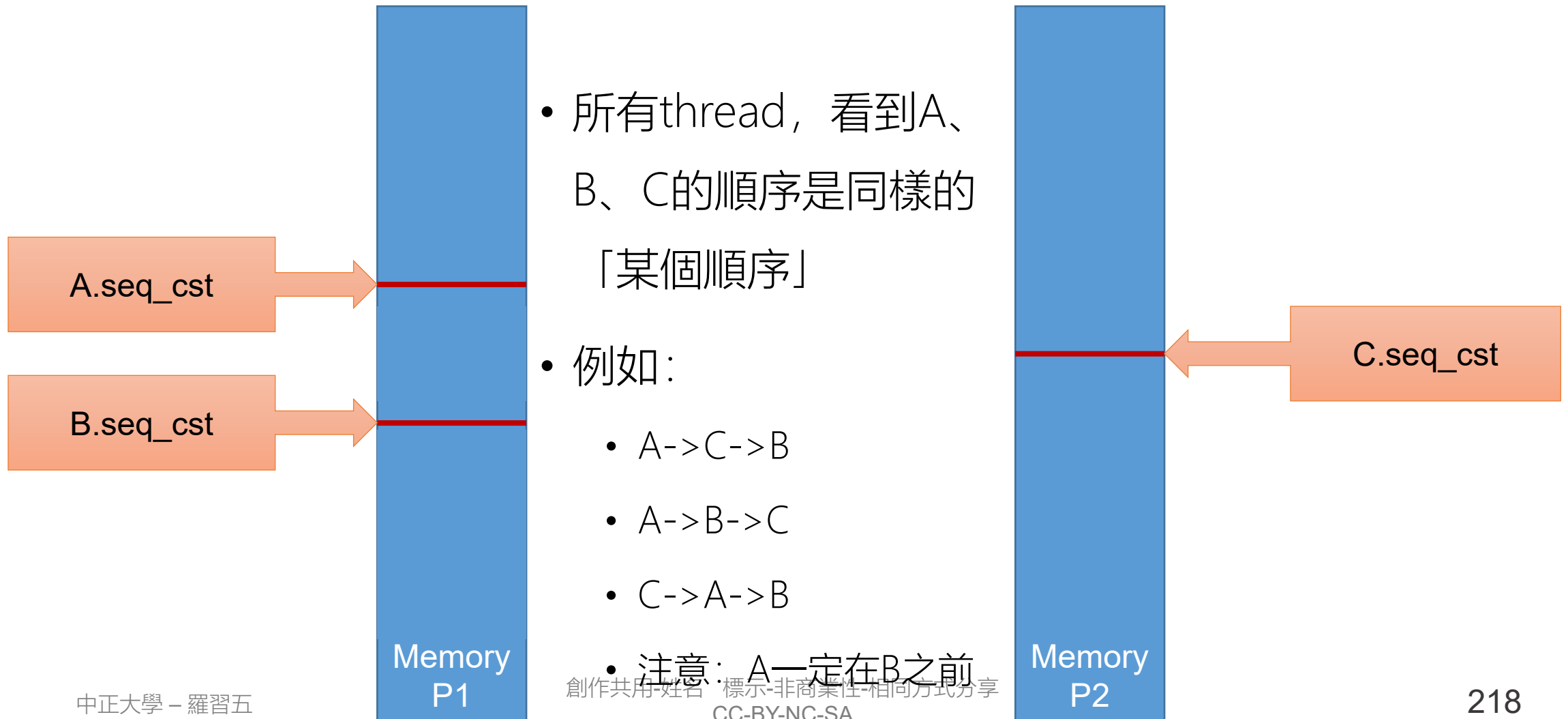
seq_cst: 最強的memory ordering

- 🍏 具有acq_rel的所有特性，並且所有宣告為acq_rel的指令一定不同時執行，一定是依照某個順序執行

須滿足下列條件

- 🍏 **Program order:** each process issues a memory request ordered by its program.
- 🍏 **Write atomicity:** memory requests are serviced based on the order of a single FIFO queue.

舉例：seq_cst



舉例：seq_cst

- 前提：
 - a()、b()、c()、d()都是一個thread
 - x, y, z是共用變數
- 如果對c()和d()而言，他們看到的x, y更改順序是一致的，那麼z就一定不會是0。
- 除了seq_cst以外，其他方法無法保證a()、b()所產生的結果是某一個特定順序。
 - 因為a()和b()在不同thread，並且a()和b()沒有任何關係，因此不在其他記憶體順序的規範中
- 舉例：
 - 如果c()看到的順序是a()再b())
 - 而d()看到的順序是b()再a())
 - 這樣就不是seq_cst

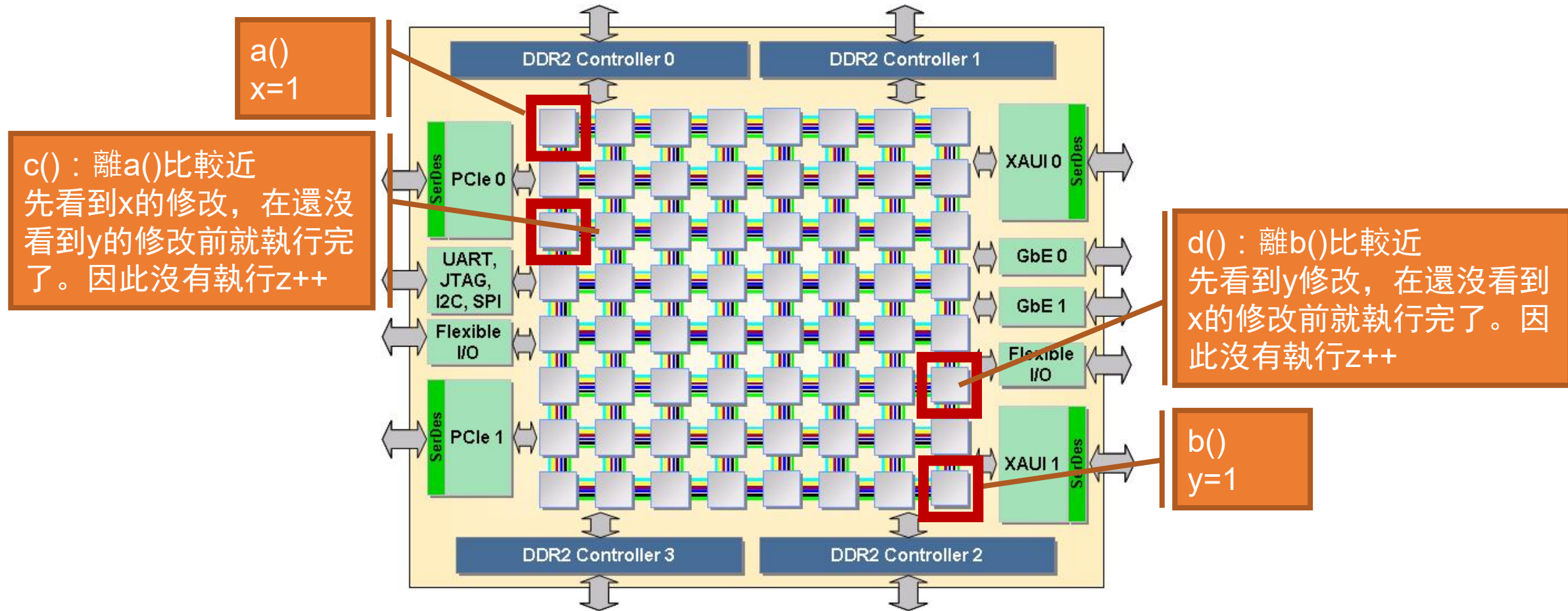
```
1.  int x= 0, y= 0, z=0;
2.  a() { x= 1; }
3.  b() { y= 1; }
4.  //如果先執行a(), 並且於b()還未執行之前, 那麼z「不會」加一
5.  c() {
6.      while (x == 1); //等x
7.      if (y) z++;
8.  }
9.  //如果先執行b(), 並且a()還未執行之前, 那麼z「不會」加一
10. d() {
11.     while (y == 1); //等y
12.     if (x) z++;
13. }
14. creat_4threads(a, b, c, d);
15. join_4thread();
16. //如果memory ordering使用seq_cst,
17. //那麼要不c()和d()要不看到a先執行, 或者看到b先執行
18. //假設先看到a()執行, b()後執行, 那麼d()「一定」會令z++
19. //如果b()先執行, c()一定會另z++, 所以z一定不是0
```

程式碼簡化自：

https://en.cppreference.com/w/cpp/atomic/memory_order

seq_cst: 在CPU架構上的舉例

不符合seq_cst的多核心架構



seq_cst: 在CPU架構上的舉例

不符合seq_cst的多核心架構

