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

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、aotmic、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/...及2thread/...
- 進階的lock機制
  - lock-free queue
  - sequential lock
  - ticket lock
  - r/w spinlock

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
```

使用工具:“ perf ”以及“ vtune ”

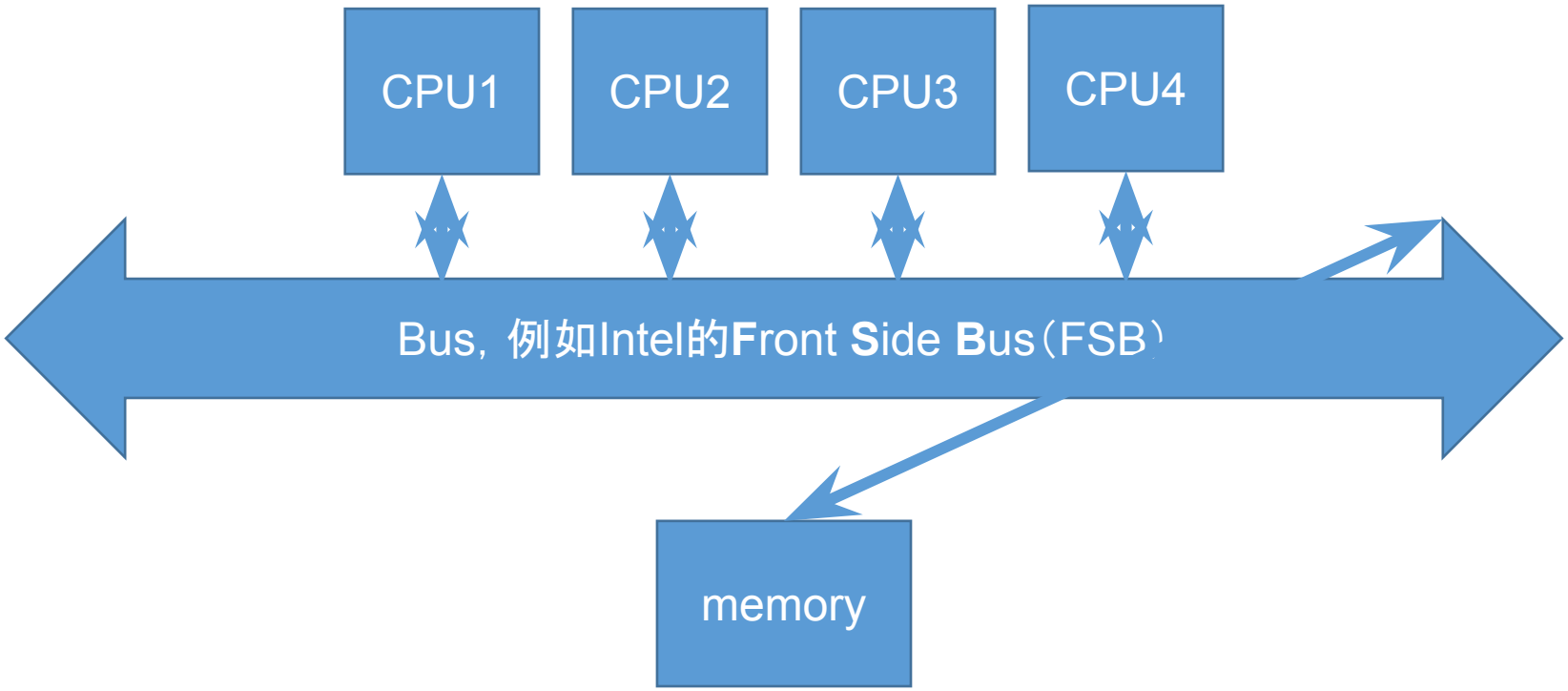
- perf及VTune都是使用CPU的 “performance monitor unit”
- 因為multi-thread的程式通常需要不同的core之間交換訊息，因此使用perf、VTune可以協助優化
- perf除了可以觀察CPU的效能，還可以觀察Linux kernel內部
- VTune給的「報告」比較人性化

在撰寫multithread程式之前

- 軟體要進行multi-threading必須靠硬體的幫忙(硬體提供什麼樣的執行環境？我們可以做什麼樣的假設？)
- 我們必須知道硬體所提供的功能和局限性
- 要正確的了解多執行緒執行的各種現象，必須對硬體有所了解
- 本單元簡述與multi-threading相關的硬體知識

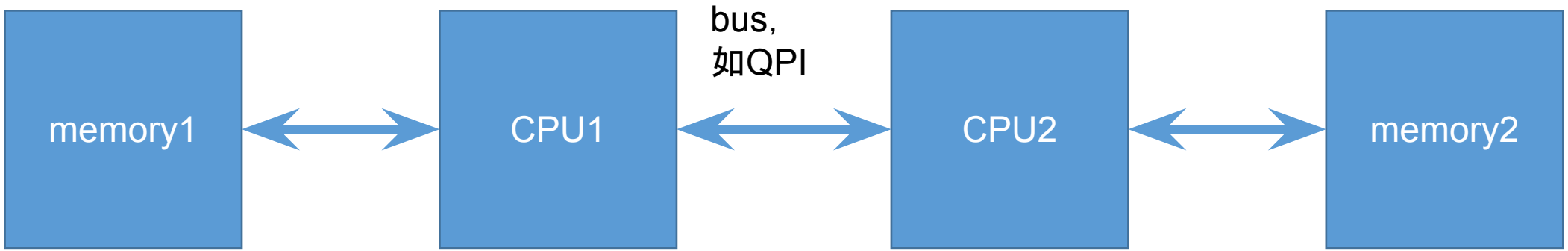
適合多執行緒的硬體

NUMA與UMA架構



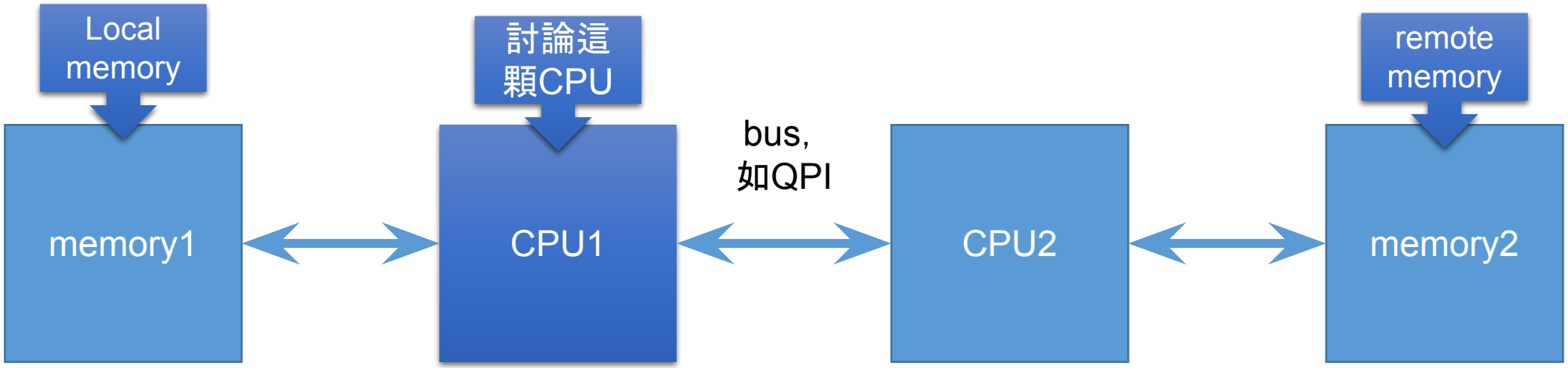
所有處理器都使用同一個記憶體，對程式設計師而言，記憶體的架構非常容易了解，但缺點是記憶體頻寬有限，不適用於「非常多」處理器

NUMA與UMA架構



QPI：Intel **Q**uick**P**ath **I**nterconnect

NUMA與UMA架構



QPI：Intel **Q**uick**P**ath **I**nterconnect

NUMA與UMA架構

latency

```
Measuring idle latencies (in ns)...
        Memory node
Socket       0       1
     0    67.5   125.2
     1   126.5    68.5
```

bandwidth

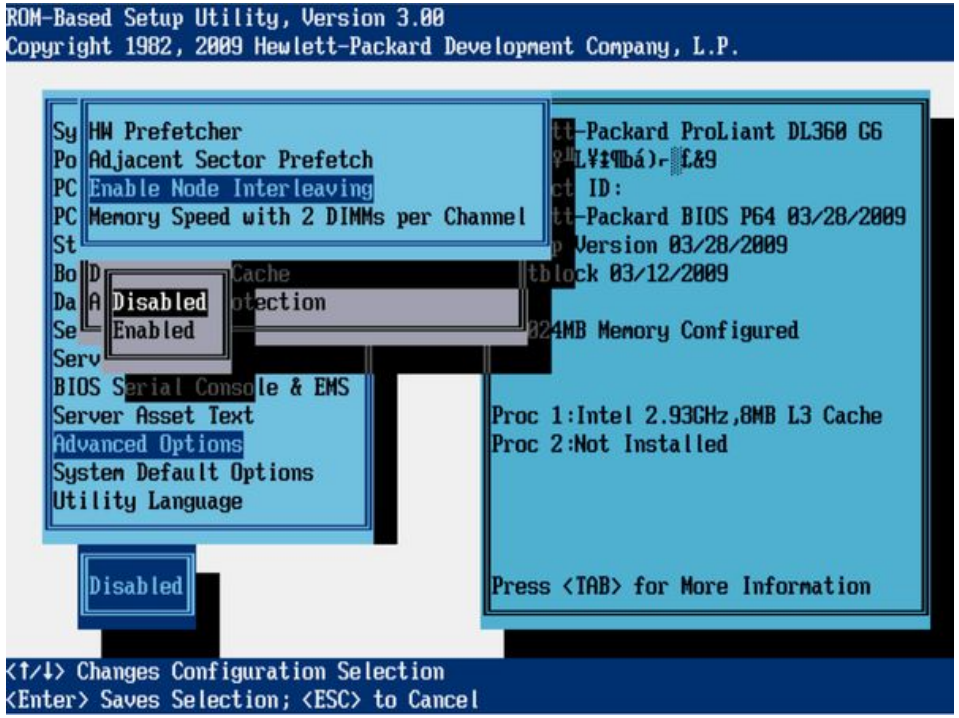
```
Measuring Memory Bandwidths between nodes within system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using Read-only traffic type
        Memory node
 Socket       0       1
     0  38477.8 19373.1
     1  19510.3 38610.4
```

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的存取方式。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）



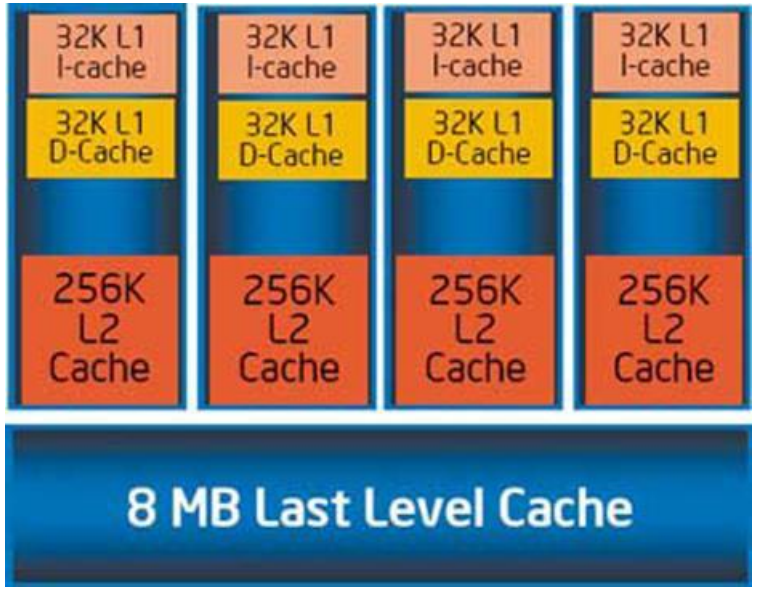
http://systemmanager.ru/c01609725.en/102428.htm

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)



https://konspekta.net/lektsiiorgimg/baza8/4374096372124.files/image006.jpg

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;
```

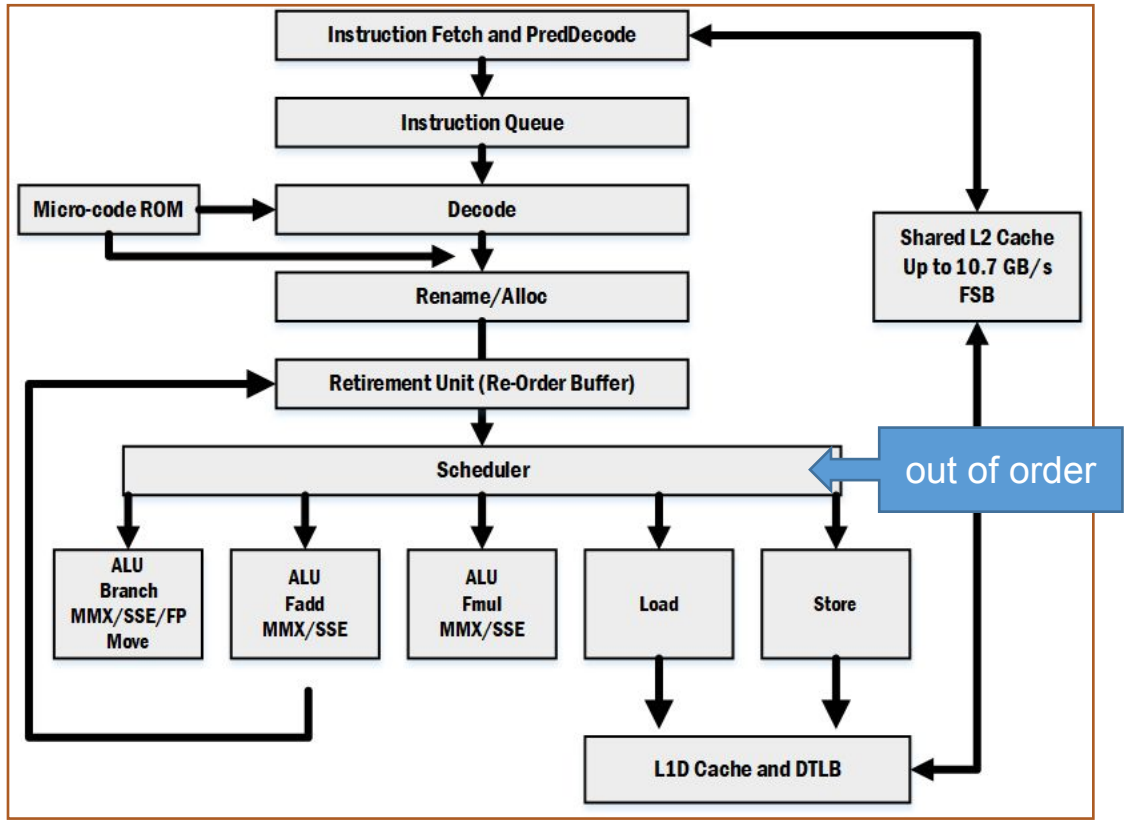
- 如果程式碼有相依性，那麼CPU會保證執行的順序。例如：

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：

```
A=1;
B=2;
```

範例二：

```
B=2;
A=1;
```

- 在mutlti-threading的情況下，如果一定要保證順序，必須使用 memory barrier(後面會介紹)

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

```
$ gcc -O3 memoryModel.c
/*使用gcc或icc*/
$ icc -O3 memroyModel.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    $0xfffffffffffffff80,%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
End of assembler dump.
```

指令順序對multi-thread的重要性

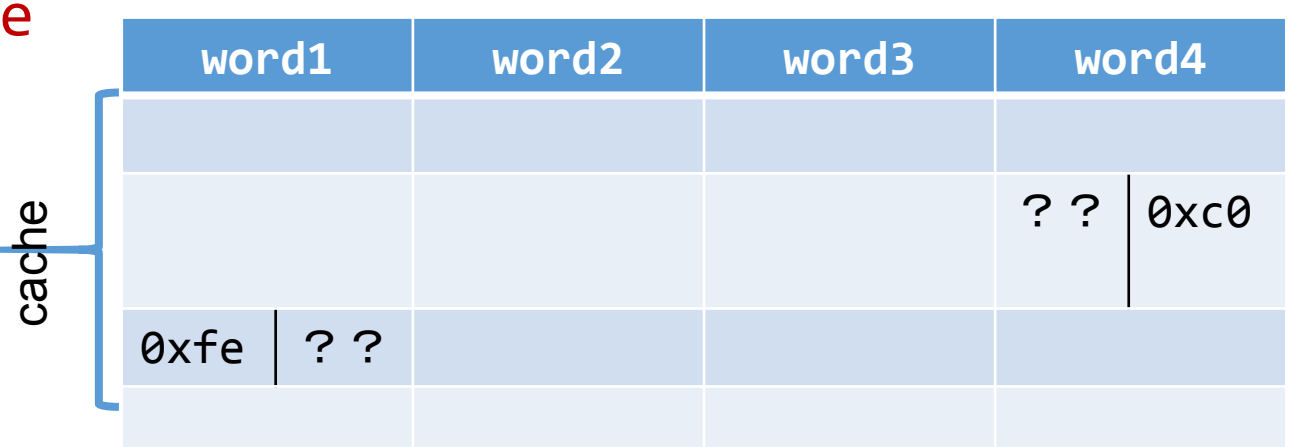
- 在許多multi-thread的程式中，都是透過設定變數，控制不同thread之間的交互關係
  - 執行的前後關係
  - 存取全域變數或全域資料結構的正確性
- 在相關的程式碼必須確認執行的順序的正確性
- 除了compiler會調動程式碼以外，CPU也會動態調動程式碼（OoO），因此必須使用編譯器、函數庫所提供的memory barrier確保關鍵程式碼的執行順序

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
- 「0xcofe」跨過二個cache line，因此CPU會

1. 先讀??0xco
2. 在讀0xfe??
3. 將??0xc0與0xfe??組合成0xc0fe

- 在mutithreading的情況下，如果要保證load、store是atomic operation，需要宣告為atomic_t

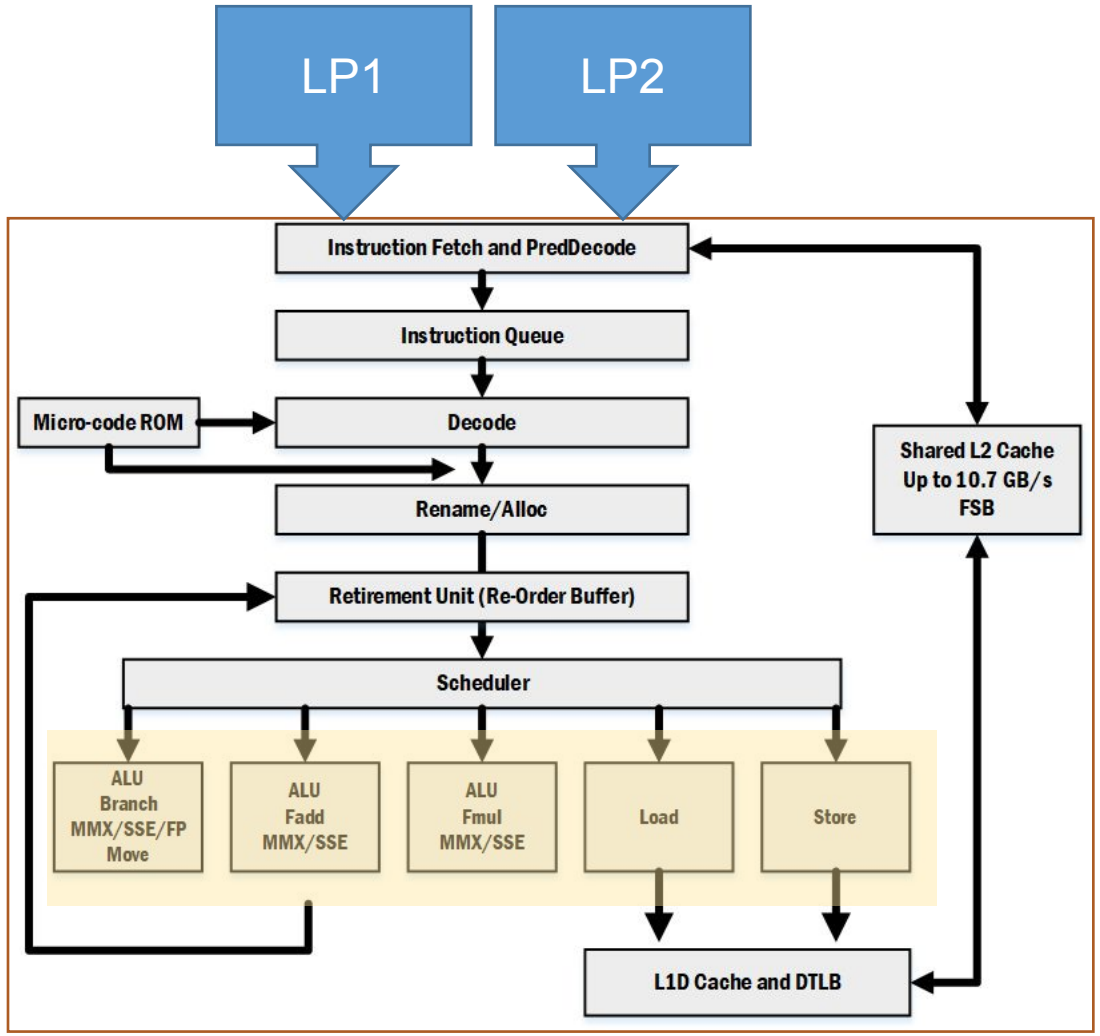


x86及x64的load & store

- 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

再論硬體架構

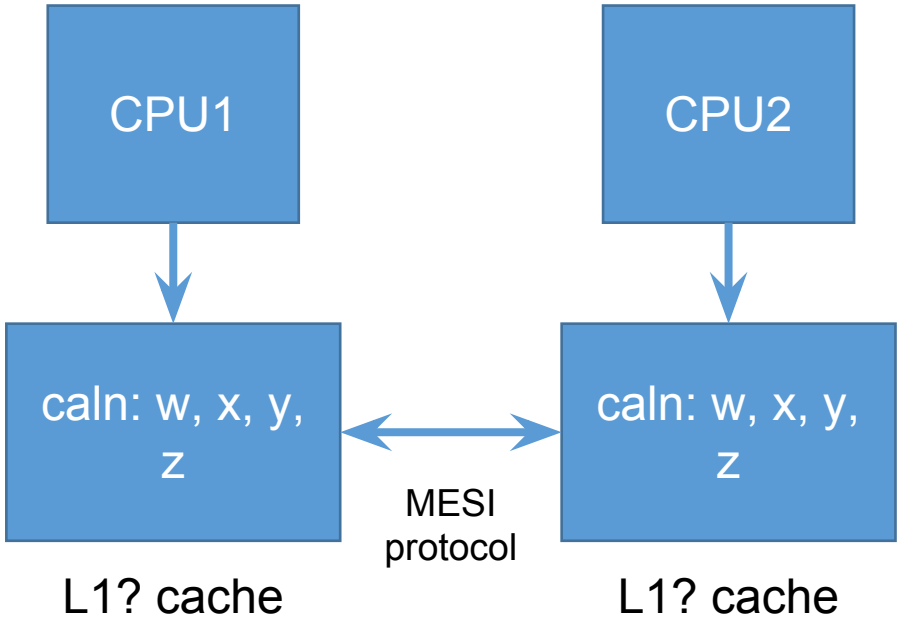
- 如右圖所示，以Intel core i7為例，一個core上可以執行2個logical processor（LP）
- 每一個LP可以執行獨立的程式（相同程式也可）
- 透過同時執行多個程式，可以讓CPU的執行單元（execution unit）的使用率提高（右圖黃色的部分），進而增加整體速度
- 如果LP1與LP2會互相惡性競爭資源，可能會讓速度變得更慢
  - 例如：LP1等LP2執行完成，但LP1不斷的偵測LP2是否完成工作，造成LP2一直被干擾
  - 解決方法：在spinlock中加入 “pause” 指令



http://www.cnblogs.com/TaigaCon/p/7678394.html

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

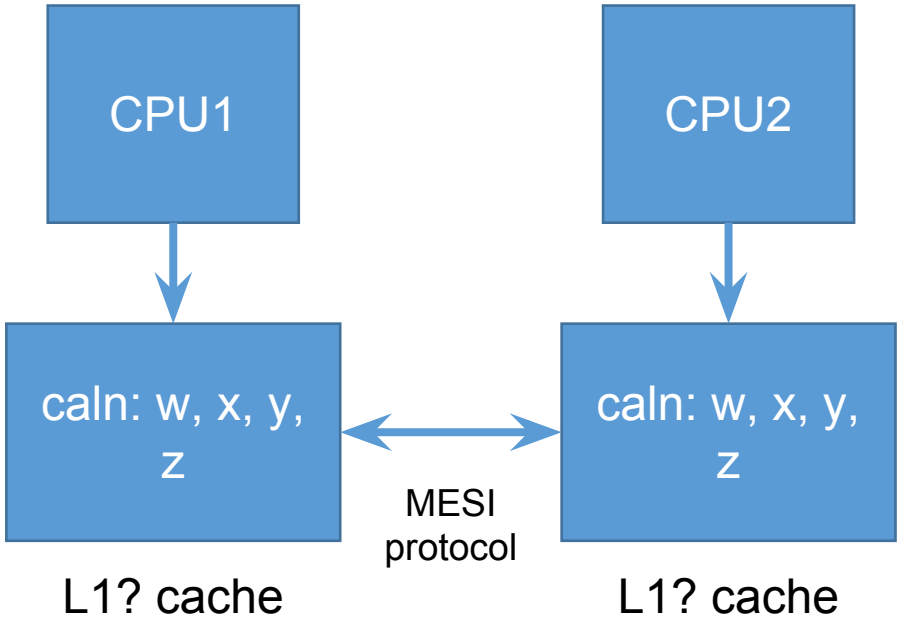
```
Samples: 51K of event 'mem-stores', Event count (approx.): 5563314379
Overhead  Shared Object                  Symbol
  98.79%  volatile                       [.] main
   0.10%  [kernel]                       [k] format_decode
   0.07%  [kernel]                       [k] vsnprintf
/*上面例子是perf top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. }
```

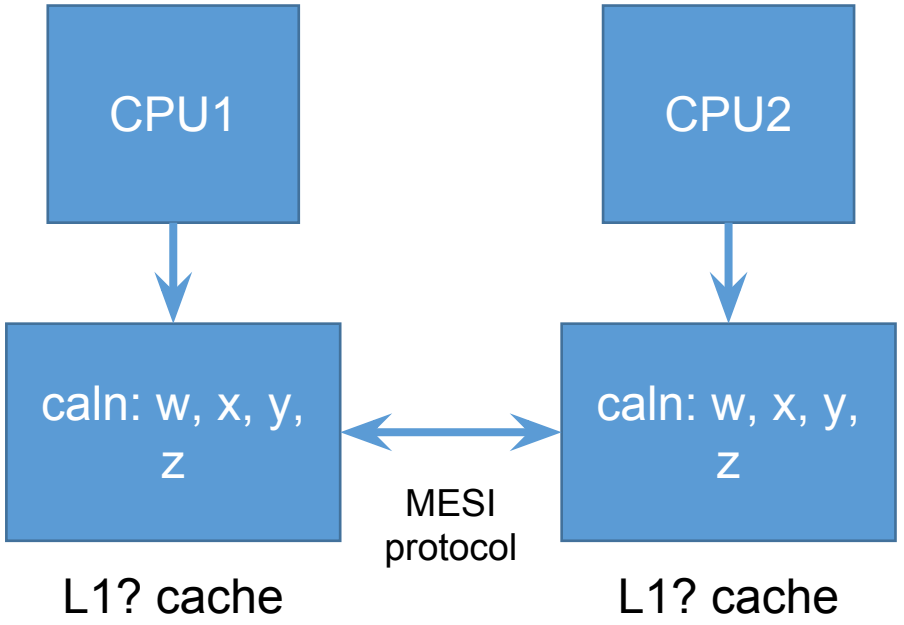
記憶體的一致性(Cache Coherence)

- 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(syncronization)
  - 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



Intel使用的記憶體模型

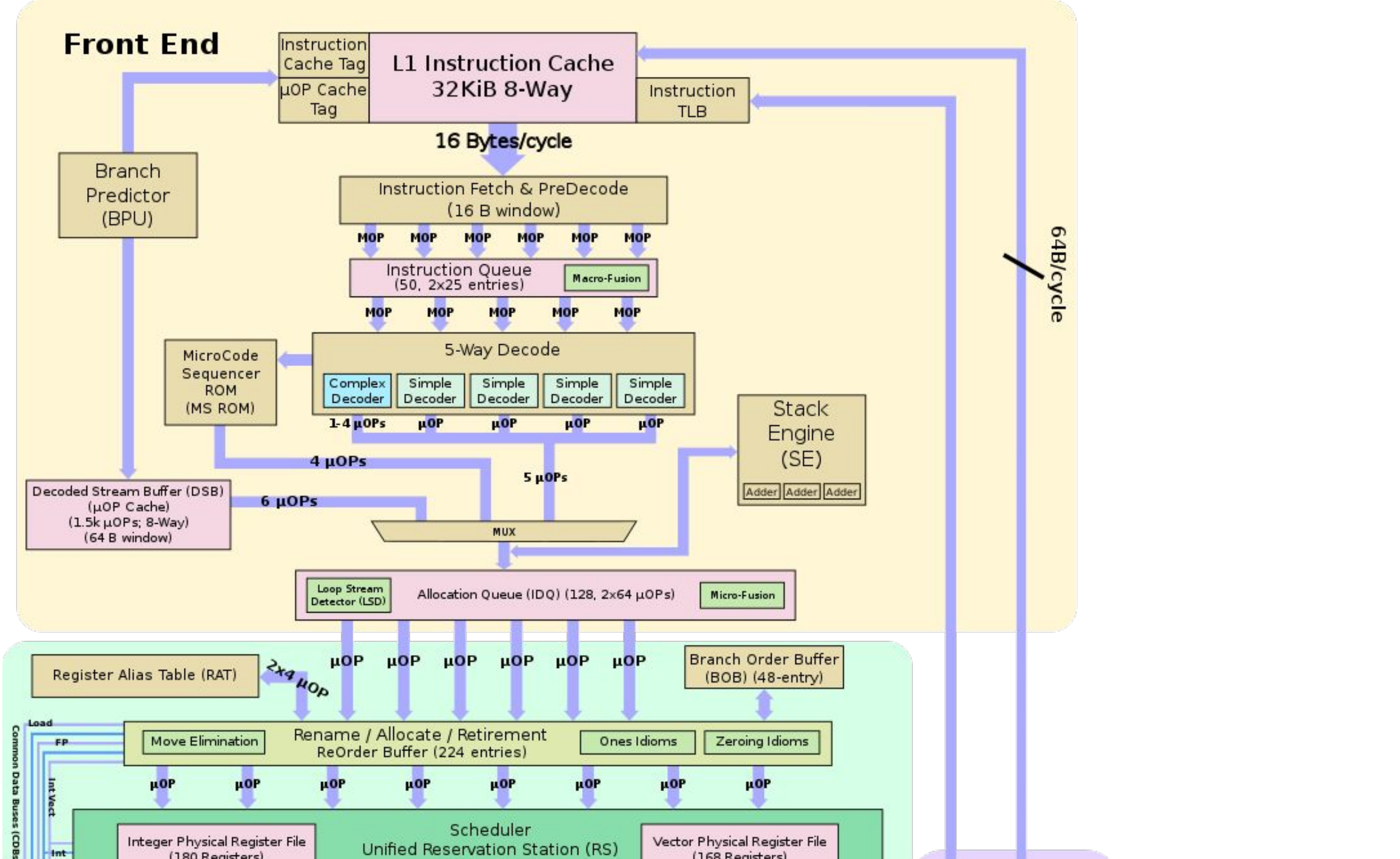
- Intel使用snooping記憶體同步（memory consistency）模型
- Snooping有多種實現的方法，Intel採用以write-invalidation為基礎的MESI
- 在Intel的PMU（performance monitor unit，perf及VTune的硬體基礎）文件上，使用xsnp代表snooping
- 將MESI分成M、E、S三個事件
- 因此在Intel的處理器上可以觀察Snooping和MESI相關事件，判斷資料同步事件發生的頻率

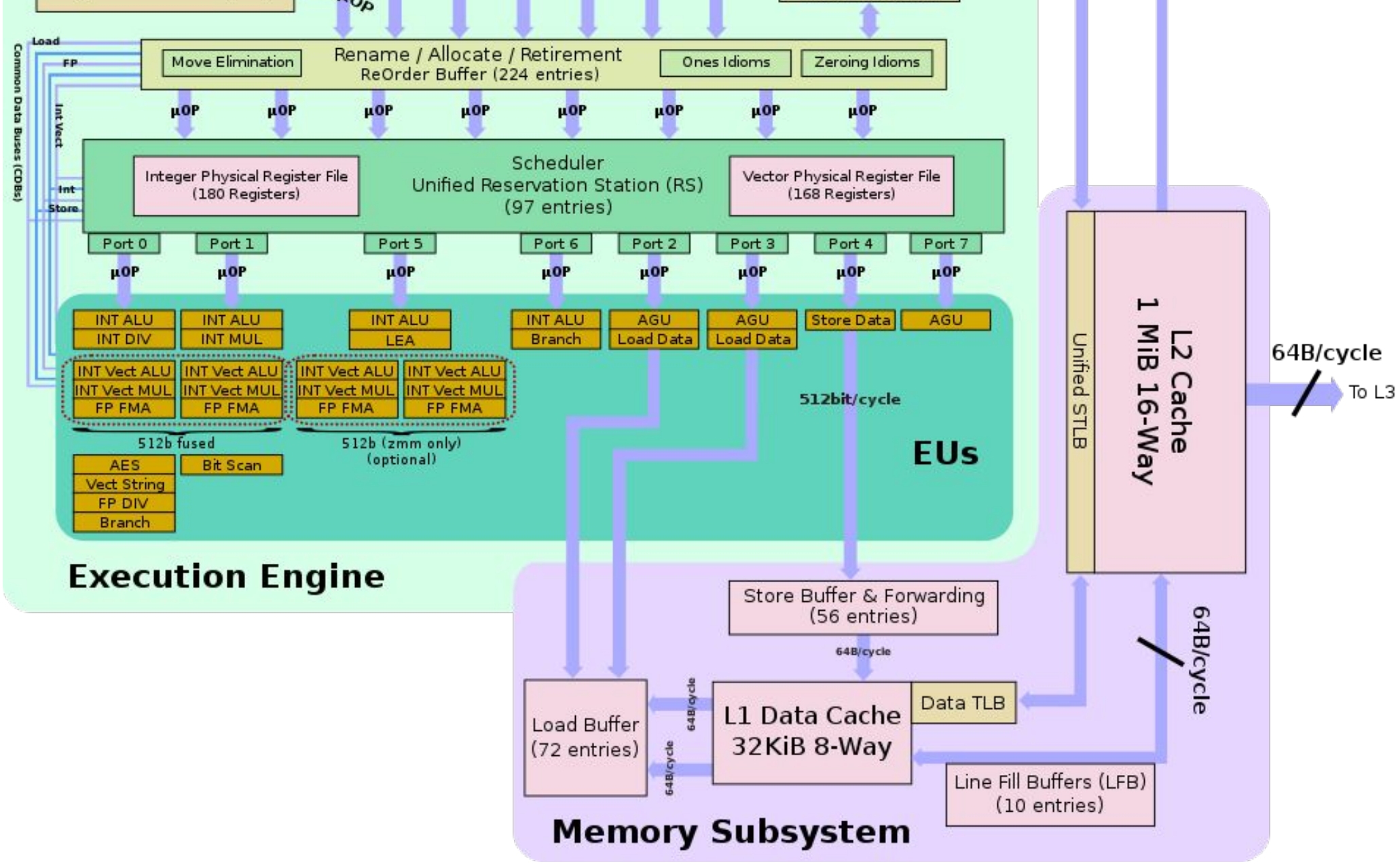
Memory Ordering Machine Clears

- Intel的PMU能支援Memory Ordering Machine Clears，事件名稱為 “machine_clears.memory_ordering”
- Intel的處理器支援“out-of-order load”，而 out-of-order會造成 “Memory Ordering Machine Clears”
- 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

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
  - 類比：二個人同時改一個文件檔，這個文件會被反覆的複寫，造成錯誤，應該一個人改完以後，再換另外一個人改
- 如果是簡單型別，可以使用atomic_t
- 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
  - 例如：atomic_compare_and_exchange


Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP MOP MOP MOP MOP MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MOP MOP MOP MOP MOP
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Simple Decoder
1-4 µOPs µOP µOP µOP µOP
4 µOPs
5 µOPs
6 µOPs
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
MUX
Stack Engine (SE)
Adder Adder Adder
64B/cycle
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
Register Alias Table (RAT)
2x4 µOP
µOP µOP µOP µOP µOP µOP
Branch Order Buffer (BOB) (48-entry)
Load
FP
Int Vect
Int
Common Data Buses (CDBs)
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
µOP µOP µOP µOP µOP µOP µOP µOP
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS)
Vector Physical Register File (168 Registers)



https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)#Memory_Protection_Extension_.28MPX.29

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）
- 如果是稀疏的table是否有其它種表示方法

- PMU中有很多事件是供給compiler工程師分析程式碼的品質，一般工程師不需要特別注意

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 instruciton）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 operatiion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	66,834	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

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L1 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    13     4    14    15     5    16     6
17    18    19     7    20     8    21    22    23     9    24    25
10    26    11    27    12    28    13    14    29    15    16    30
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*
- argumentt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:       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

semaphore

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。

使用semaphore設計concurrent queue

只支援單個producer、consumer

用semaphore設計queue：buffer_sem.c

```
1.  #define bufsize 10
2.  long buffer[bufsize];
3.  int in=0, out=0;    //新進的資料放在 buffer[in]，讀取資料從 buffer[out]
4.  sem_t notFull, notEmpty;    //使用semaphore判斷buffer的狀態
5.  void put() {
6.      static int item=1;  //用於除錯，放入 buffer的資料都是嚴格遞增的數列
7.      sem_wait(&notFull); //等待buffer有空間
8.      //printf("put %d\n", item);
9.      buffer[in]=item++;  //將資料放入
10.     in++; in=in%bufsize;    //規劃下筆資料應該擺放的地點
11.     sem_post(&notEmpty);    //放入資料了，所以就不是 empty
12. }
13. void get() {
14.     int tmpItem;    //讀取的數字暫時放在 tmpItem
15.     sem_wait(&notEmpty);    //等待buffer有東西
16.     tmpItem=buffer[out];    //讀出buffer的東西
17.     out++; out=out%bufsize; //規劃下次要提取資料的地方
18.     sem_post(&notFull); //拿出資料了，因此一定不是 full
19. }
```

```
20.     void producer(void* name) {     //生產資料的執行緒
21.         for (int i=0; i<10000000; i++)
22.             put();
23.     }
24.     void consumer(void* name) {     //消化資料的執行緒
25.         for (int i=0; i<10000000; i++)
26.             get();
27.     }
28.     int main(void) {
29.         pthread_t id1, id2, id3, id4;
30.         sem_init(&notFull, 0, bufsize);     //剛開始全部是空，因此有 bufsize的空間可以放
31.         sem_init(&notEmpty, 0, 0);          //剛開始沒有東西放在 buffer內
32.         pthread_create(&id1,NULL,(void *) producer,NULL);
33.         //pthread_create(&id2,NULL,(void *) producer,NULL); //只支援一個producer，拿掉這個註解會發生錯誤
34.         pthread_create(&id3,NULL,(void *) consumer,NULL);
35.         //pthread_create(&id4,NULL,(void *) consumer,NULL); //只支援一個consumer，拿掉這個註解會發生錯誤
36.         pthread_join(id1,NULL);
37.         //pthread_join(id2,NULL);
38.         pthread_join(id3,NULL);
39.         //pthread_join(id4,NULL);
40.     }
```

執行結果

```
$ ./time_detail ./buffer_sem


經過時間:                    2.923294185s
CPU花在執行程式的時間:          5.812117s
CPU於usr mode執行此程式所花的時間:    3.717052s
CPU於krl mode執行此程式所花的時間:    2.095065s
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:        36898
非自願性的context switch:      57
```

執行結果分析

以buffer_sem而言，總共有2個thread在執行

- 執行時間為：2.9秒
- 所消耗的CPU time：5.8秒
- 換句話說平行度為2（達到最高）
- 但浪費掉的額外開銷也很大，於krl中花費了2.1秒，usr為3.7秒。buffer_sem並沒有做任何的I/O，semaphore光是kernel的部分就造成36%的overhead。
- Context-switch共發生了36898次

為什麼buffer_sem.c只支援一個生產者、一個消費者

- 以put為例，如果buffer是空的，那麼可以有多個producer可以進入，這些producer會同時修改buffer和in，造成資料錯誤
- 因為我們沒有用任何保護機制，保護多個producer同時修改buffer和in

```
1.      void put() {
2.          static int item=1;  //用於除錯，放入buffer的資料都是嚴格遞增的數列
3.          sem_wait(&notFull); //等待buffer有空間
4.          //printf("put %d\n", item);
5.          buffer[in]=item++;  //將資料放入，所有producer都可以修改，造成錯誤
6.          in++; in=in%bufsize;//規劃下筆資料應該擺放的地點。所有producer都可以修改in
7.          sem_post(&notEmpty);    //放入資料了，所以就不是empty
8.      }
```

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krl 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

使用semaphore與mutex設計concurrent queue

支援複數個producer、consumer

多個producer多個consumer buffer_sem_mutex.c

```
1.  void put() {
2.      sem_wait(&notFull); //等待buffer有空間
3.      pthread_mutex_lock(&putMutex);  //限制一次一個producer進入
4.      buffer[in]=item++;  //將資料放入
5.      in++; in=in%bufsize;    //規劃下筆資料應該擺放的地點
6.      pthread_mutex_unlock(&putMutex);
7.      sem_post(&notEmpty);    //放入資料了，所以就不是empty
8.  }
9.  void get() {
10.     int tmpItem;    //讀取的數字暫時放在tmpItem
11.     sem_wait(&notEmpty);    //等待buffer有東西
12.     pthread_mutex_lock(&getMutex);  //限制一次一個consumer進入
13.     tmpItem=buffer[out];    //讀出buffer的東西
14.     out++; out=out%bufsize; //規劃下次要提取資料的地方
15.     pthread_mutex_unlock(&getMutex);
16.     sem_post(&notFull); //拿出資料了，因此一定不是full
17. }
```

執行結果

```
$ ./time_detail ./buffer_sem_mutex

經過時間:                    19.881834501s
CPU花在執行程式的時間:          37.851506s
CPU於usr mode執行此程式所花的時間:    16.387817s
CPU於krl mode執行此程式所花的時間:    21.463689s
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:       2462675
非自願性的context switch:     9831
```

buffer_sem

```
2.923294185s
5.812117s
3.717052s
2.095065s
76
0
36898
57
```

執行結果分析

以buffer_sem_mutex而言，總共有4個thread在執行

- 執行時間為：19.9秒
- 所消耗的CPU time：37.9秒
- 換句話說平行度為1.9 (理論最高值為4，1.9表示雖然有二個producer，但只有一個在put內，另外一個在外邊等。consumer也是相同情況）
- 浪費掉的額外開銷也很大，於kernel中花費了21.5秒。請注意，buffer_sem並沒有做任何的I/O，因此kernel的開銷幾乎都是semaphore
- Context-switch共發生了2462675次

semaphore & mutex的綜合討論

- 截至目前為止我們看到這二種機制的overhead都很大，這是因為我們處理的資料量很小，相對來說鎖定機制佔的比重就很高
- 這二種機制都會觸發context-switch，因此如果處理的資料量很大、處理時間很長，context-switch可以避免busy-waiting（後面會討論）的問題
- 如果系統中有多個程序在跑，與其busy-waiting，不如將CPU讓出來（即context-switch）給其他程序跑

lock-free的 concurrent queue

只支援單個producer、consumer

lockfree_buf.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.          in = (in + 1)%bufsize;  //下一次要放入的位置
7.  }
8.  void get() {
9.          int tmpItem;  //讀取的數字暫時放在tmpItem
10.         while (in == out) ;  // busy waiting
11.         tmpItem=buffer[out]; //讀出buffer的東西
12.         out = (out + 1)%bufsize;   //下一次要拿取的位置
13. }
```

IA-32及IA-64的記憶體讀寫

如果一般C語言的基礎型別（如：char、short、long、long long），對齊該型別的大小，則load和store為atomic operation

執行結果

```
$ ./time_detail ./lockfree_buf

經過時間:                    0.699771642s
CPU花在執行程式的時間:          1.397830s
CPU於usr mode執行此程式所花的時間: 1.393825s
CPU於krl mode執行此程式所花的時間: 0.004005s
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:       2
非自願性的context switch:     1
```

buffer_sem

```
2.923294185s
5.812117s
3.717052s
2.095065s
76
0
36898
57
```

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

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	4.2(				2.131s
user	4.2(				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」

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

thread_print_id.c

```
1.  #define _GNU_SOURCE
2.  #include <stdio.h>
3.  #include <pthread.h>
4.  #include <semaphore.h>
5.  #include <sys/types.h>
6.  #include <unistd.h>
7.  #include <sys/syscall.h>
8.
    int gettid() {
9.      return syscall(SYS_gettid);
10. }
```

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.
     int main(void) {
17.      pthread_t id1, id2;
18.      printf("my pid = %d\n", getpid());
19.      pthread_create(&id1,NULL,(void *) thread,NULL);
20.      pthread_create(&id2,NULL,(void *) thread,NULL);
21.      getchar();
22.      return (0);
23.  }
```

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

spinlock.c

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

spinlock.c

```
15.     int main(void) {
16.         pthread_t id1, id2;
17.         pthread_spin_init(&spinlock, PTHREAD_PROCESS_PRIVATE);
18.         pthread_create(&id1,NULL,(void *) thread,NULL);
19.         pthread_create(&id2,NULL,(void *) thread,NULL);
20.         pthread_join(id1,NULL);
21.         pthread_join(id2,NULL);
22.         printf("1000000+1000000 = %d\n", global);
23.         return (0);
24.     }
```

比較

	spinlock	mutex	semphore
real	39.748s	217.471s	289.102s
user	78.440s	225.196s	333.272s
sys	0.000s	88.820s	231.756s

為何比較快(strace)

spinlock

% time	seconds	usecs/call	calls	errors	syscall
0.00	0.000000	0	2		read
0.00	0.000000	0	1		write
0.00	0.000000	0	3		open
0.00	0.000000	0	3		close
0.00	0.000000	0	4		fstat
0.00	0.000000	0	13		mmap
0.00	0.000000	0	8		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		rt_sigprocmask
0.00	0.000000	0	4	4	access
0.00	0.000000	0	2		clone
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		getrlimit
0.00	0.000000	0	1		arch_prctl
0.00	**0.000000**	**0**	**1**		**futex**
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		set_robust_list
100.00	0.000000		53	4	total

mutex

% time	seconds	usecs/call	calls	errors	syscall
100.00	**0.044000**	**22000**	**2**		**futex**
0.00	0.000000	0	2		read
0.00	0.000000	0	1		write
0.00	0.000000	0	3		open
0.00	0.000000	0	3		close
0.00	0.000000	0	4		fstat
0.00	0.000000	0	13		mmap
0.00	0.000000	0	8		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		rt_sigprocmask
0.00	0.000000	0	4	4	access
0.00	0.000000	0	2		clone
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		getrlimit
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		set_robust_list
100.00	0.044000		54	4	total

mutex在futex這個system call上花了0.044秒，spinlock花了0秒

為何比較快(strace)

semaphore

% time	seconds	usecs/call	calls	errors	syscall
100.00	0.137686	68843	2		futex
0.00	0.000000	0	2		read
0.00	0.000000	0	1		write
0.00	0.000000	0	3		open
0.00	0.000000	0	3		close
0.00	0.000000	0	4		fstat
0.00	0.000000	0	13		mmap
0.00	0.000000	0	8		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		rt_sigprocmask
0.00	0.000000	0	4	4	access
0.00	0.000000	0	2		clone
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		getrlimit
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		set_robust_list
100.00	0.137686		54	4	total

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cortical section很大時，spinlock的效能反而不如mutex及semaphore了
- 回家作業...

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。(撒尿牛丸)(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);
```

Adaptive mutex的想法

mutex_adaptive.c

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

mutex_adaptive.c

```
15.  int main(void) {
16.      pthread_t id1, id2;
17.      pthread_mutexattr_t attr;
18.      pthread_mutexattr_init(&attr);
19.      pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ADAPTIVE_NP);
20.      pthread_mutex_init(&mutex, &attr);
21.      pthread_create(&id1,NULL,(void *) thread,NULL);
22.      pthread_create(&id2,NULL,(void *) thread,NULL);
23.      pthread_join(id1,NULL);
24.      pthread_join(id2,NULL);
25.      printf("1000000+1000000 = %d\n", global);
26.      return (0);
27.  }
```

再用strace看一次

spinlock:0.000

adaptive_mutex: 0.004

mutex: 0.044

Semaphore: 0.137686

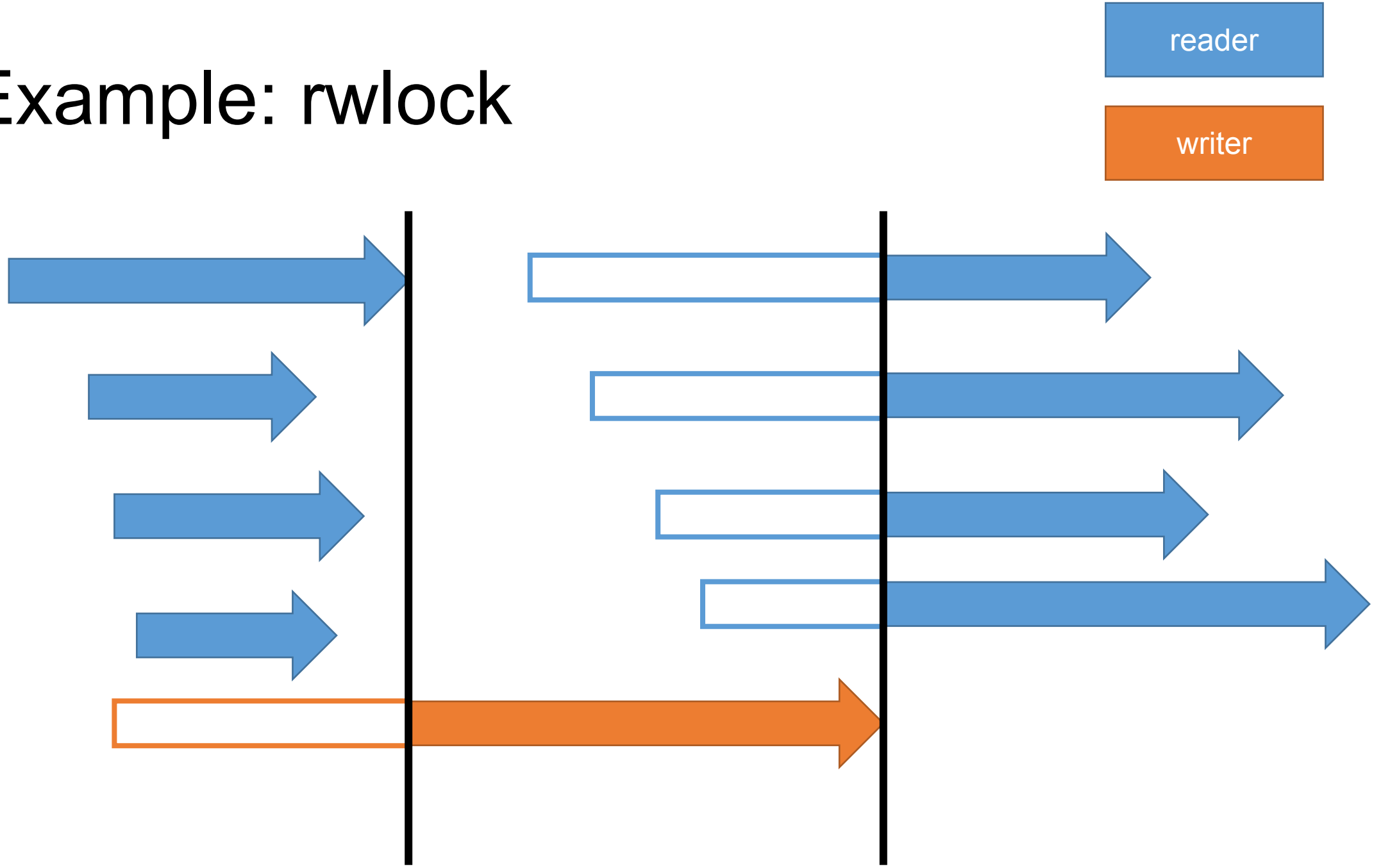
% time	seconds	usecs/call	calls	errors	syscall
100.00	0.004000	2000	2		futex
0.00	0.000000	0	2		read
0.00	0.000000	0	1		write
0.00	0.000000	0	3		open
0.00	0.000000	0	3		close
0.00	0.000000	0	4		fstat
0.00	0.000000	0	13		mmap
0.00	0.000000	0	8		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		rt_sigprocmask
0.00	0.000000	0	4	4	access
0.00	0.000000	0	2		clone
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		getrlimit
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		set_robust_list
100.00	0.004000		54	4	total

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，可以更近一步的平行化

Example: rwlock



範例：rwlock

下學期OS課介紹

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

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

__thread.c

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

__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.     }
```

執行結果

__thread

```
$time ./__thread
@thread_local =
0x7fc6d8cb06fc
@thread_local =
0x7fc6d94b16fc
1000000000+1000000000 =
2000000000

real 0m2.143s
user 0m4.272s
sys  0m0.000s
```

2job

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

real 0m2.132s
user 0m4.248s
sys  0m0.000s
```

執行結果

__thread

```
12          thread_local+=1;
   0x0000000000400847 <+17>:  mov
%fs:0xfffffffffffffffc,%eax
   0x000000000040084f <+25>:  add
$0x1,%eax
   0x0000000000400852 <+28>:  mov
%eax,%fs:0xfffffffffffffffc
```

2job

```
12          local+=1;
   0x000000000040080e <+24>:  addl
$0x1,-0x8(%rbp)
```

奇怪的編譯？

- Controls whether TLS variables may be accessed with offsets from the TLS segment register (**%gs for 32-bit, %fs for 64-bit**), or whether the thread base pointer must be added.
- gcc使用fs暫存器當thread local storage的基底值，相關的變數再做偏移，因此編譯出的程式碼比較慢

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）

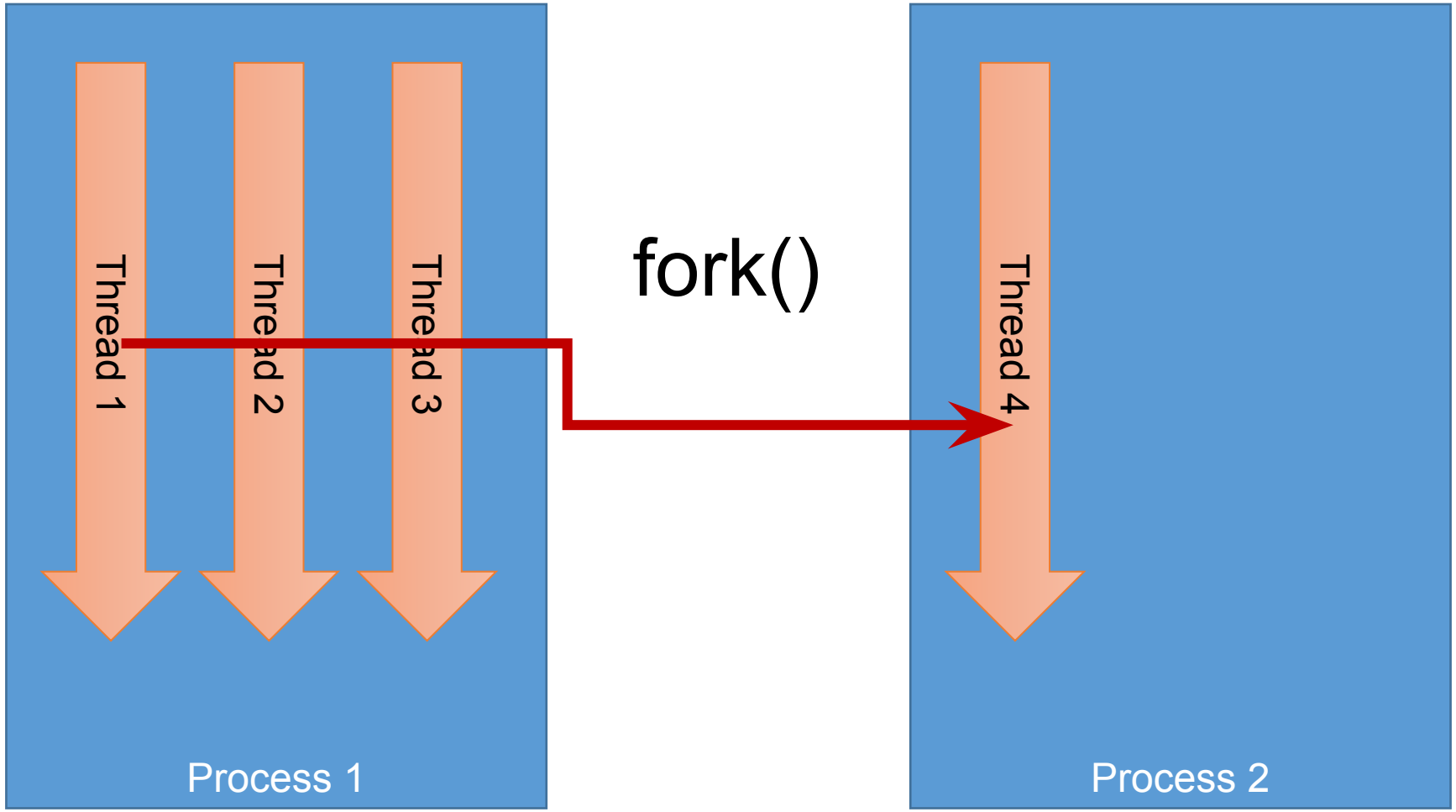
戰場的清理

- 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.
    struct DS {
5.      __attribute__((aligned(64))) int a;
6.      __attribute__((aligned(64))) int b;
7.  };
8.
    void thread(void* local) {
9.      int* _local=(int*) local;
10.     int i;
11.     for (i=0; i<1000000000; i++)
12.         *_local+=1;
13. }
```

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

結果比較（回合數：1,000,000,000）

	base	nosync（錯誤）	semaphore	mutex	volatile（錯誤，平台相依）	2job（理論上應該是最快）
real	**4.244s**	**5.371s**	**289.102s**	**217.471s**	**5.525s**	**2.131s**
user	**4.240s**	**10.520s**	**333.272s**	**225.196s**	**11.024s**	**4.248s**
sys	**0.000s**	**0.000s**	**231.756s**	**88.820s**	**0.000s**	**0.000s**

2job vs. pingpong_aligned

2job

```
12          local+=1;
   0x000000000040080e <+24>:  addl
$0x1,-0x8(%rbp)
```

pingpong_aligned

```
0x000000000040070f <+25>: mov
-0x8(%rbp),%rax
   0x0000000000400713 <+29>:  mov
(%rax),%eax
   0x0000000000400715 <+31>:  lea
0x1(%rax),%edx
   0x0000000000400718 <+34>:  mov
-0x8(%rbp),%rax
   0x000000000040071c <+38>:  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専用語法*/
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
sys0m0.004s
shiwulo@vm:~/sp/ch12$ time ./pingpong_aligned
sizeof(DS)=128
real   0m2.358s
user   0m4.704s
sys0m0.000s
shiwulo@vm:~/sp/ch12$ time ./pingpong_alignas
sizeof(DS)=128
real   0m2.373s
user   0m4.728s
sys0m0.000s
```

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<1000000000; i++)
8.          atomic_fetch_add(&global, 1);
9.  }
```

執行結果

```
$ $ time ./atomic
1000000+1000000 = 2000000000
real0m43.646s
user1m27.272s
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）

作業

- 寫一支程式利用「蒙地卡羅方法」計算pi
  - 演算法：https://goo.gl/BXlUZB
- 可以多執行緒，平行運算，計算出pi
- 執行方法
  - ./pi ####
  - ####為總共要執行幾次迴圈（亂數）
  - 執行完成後，於螢幕上印出pi的值到小數點以下第八位

Conditional wait

```
1.  int pthread_cond_destroy(pthread_cond_t *cond);
2.  int pthread_cond_init(pthread_cond_t *restrict cond,
3.  const pthread_condattr_t *restrict attr);
4.  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
5.
    int pthread_cond_wait(pthread_cond_t *restrict cond,
6.  pthread_mutex_t *restrict mutex);
7.  int pthread_cond_signal(pthread_cond_t *cond);
```

8. 要等待mutex所保護的資料結構發生變化，呼叫pthread_cond_wait。
9. 當mutex所保護的資料結構發生變化，呼叫pthread_cond_signal。

10. 下學期OS課會有更詳細的介紹