Peterson Locking Algorithm Report

Author: Charles, Liu. Derek, Zhang.

Charles, Liu: [charlesliu.cn.bj@gmail.com](mailto:charlesliu.cn.bj@gmail.com) 安远国际

Derek, Zhang: [anonymous@anonymous.com](mailto:anonymous@anonymous.com) 百度

* Peterson实现

#ifndef \_PETERSON\_LOCK\_H

#define \_PETERSON\_LOCK\_H

/\*

\* Description: implementing peterson's locking algorithm

\* File: peterson\_lock.h

\* Author: Charles, Liu.

\* Mailto: charlesliu.cn.bj@gmail.com

\*/

#include <pthread.h>

typedef struct {

volatile bool flag[2];

volatile int victim;

} peterson\_lock\_t;

void peterson\_lock\_init(peterson\_lock\_t &lock) {

lock.flag[0] = lock.flag[1] = false;

lock.victim = 0;

}

void peterson\_lock(peterson\_lock\_t &lock, int id) {

lock.flag[id] = true; // Mark as A

lock.victim = id; // Mark as B, A 和B的顺序必须正确，不然可能两个线程同时进入Critical Section

while (lock.flag[1 - id] && lock.victim == id);

}

void peterson\_unlock(peterson\_lock\_t &lock, int id) {

lock.flag[id] = false;

lock.victim = id;

}

#endif

关于Peterson Locking Algorithm：<https://en.wikipedia.org/wiki/Peterson%27s_algorithm>

这个实现有一个隐含缺陷，就是memory order的问题。

* Memory Order

## 这里有一篇文章讲到了Peterson Lock的memory order的问题，[Who ordered memory fences on an x86](https://bartoszmilewski.com/2008/11/05/who-ordered-memory-fences-on-an-x86/)?

更详细的关于Inter 64 Memory Order的资料，[Intel® 64 Architecture Memory Ordering White Paper](http://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf)

# C++ <atomic>，[std::memory\_order](http://en.cppreference.com/w/cpp/atomic/memory_order)

以上Peterson实现会出现一个指令重拍的问题，根据以上资料来看，store-load在CPU层面上是可以重排的，所以得加memory barrier。

编译时期 memory barrier： asm volatile (“” : : : “memory”)

cpu级别加上编译时期 memory barrier：asm volatile (“mfence” : : : “memory”)

# 参考Wiki：[Memory ordering](https://en.wikipedia.org/wiki/Memory_ordering)

加上 memory barrier之后

#ifndef \_PETERSON\_LOCK\_H

#define \_PETERSON\_LOCK\_H

/\*

\* Description: implementing peterson's locking algorithm

\* File: peterson\_lock.h

\* Author: Charles, Liu.

\* Mailto: charlesliu.cn.bj@gmail.com

\*/

#include <pthread.h>

typedef struct {

volatile bool flag[2];

volatile int victim;

} peterson\_lock\_t;

void peterson\_lock\_init(peterson\_lock\_t &lock) {

lock.flag[0] = lock.flag[1] = false;

lock.victim = 0;

}

void peterson\_lock(peterson\_lock\_t &lock, int id) {

lock.flag[id] = true;

lock.victim = id;

asm volatile ("mfence" : : : "memory");

while (lock.flag[1 - id] && lock.victim == id);

}

void peterson\_unlock(peterson\_lock\_t &lock, int id) {

lock.flag[id] = false;

lock.victim = id;

}

#endif

1. mfence

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes in program order the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction is globally visible. The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any SFENCE and LFENCE instructions, and any serializing instructions (such as the CPUID instruction).

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing.

The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). The PREFETCHh instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the MFENCE instruction is not ordered with respect to PREFETCHh instructions or any other speculative fetching mechanism (that is, data could be speculatively loaded into the cache just before, during, or after the execution of an MFENCE instruction).

1. lfence

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. This serializing operation guarantees that every load instruction that precedes in program order the LFENCE instruction is globally visible before any load instruction that follows the LFENCE instruction is globally visible. The LFENCE instruction is ordered with respect to load instructions, other LFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to store instructions or the SFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue and speculative reads. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The LFENCE instruction provides a performance-efficient way of insuring load ordering between routines that produce weakly-ordered results and routines that consume that data.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). The PREFETCHh instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the LFENCE instruction is not ordered with respect to PREFETCHh instructions or any other speculative fetching mechanism (that is, data could be speculative loaded into the cache just before, during, or after the execution of an LFENCE instruction).

1. sfence

Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes in program order the SFENCE instruction is globally visible before any store instruction that follows the SFENCE instruction is globally visible. The SFENCE instruction is ordered with respect store instructions, other SFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions or the LFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of insuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

对于以上的实现，想想这个问题，如果

Thread0: Thread1:

lock.flag[0] = true;

lock.victim = 0; lock.flag[1] = true;

lock.victim = 1;

mfence;

如果这个时候Thread0的lock.victim = 0对于Thread1可见，那么会进入Critical Section

mfence;

如果这个时候Thread1的lock.victim = 1对于

Thread0可见，那么会进入Critical Section

但是这种情况是不存在的，参考上文mfence标注的红色语句。还有就是为什么lfence和sfence对于Peterson Lock无效，因为Peterson Lock是一个Store Load重排的问题，所以只能用mfence。顺便提一下，Intel IA64系列的所有cpu都是strict order的，只存在Store Load重排的问题，所以lfence和sfence对于Intel IA64系列来说是无意义的。

* 关于volatile和GCC优化

去掉volatile，不加优化，运行正常，*void peterson\_lock(peterson\_lock\_t &lock, int id)*的汇编结果如下

0000000000400638 <\_Z13peterson\_lockR15peterson\_lock\_ti>:

400638: 55 push rbp

400639: 48 89 e5 mov rbp,rsp

40063c: 48 89 7d f8 mov QWORD PTR [rbp-0x8],rdi

400640: 89 75 f4 mov DWORD PTR [rbp-0xc],esi

400643: 48 8b 55 f8 mov rdx,QWORD PTR [rbp-0x8]

400647: 8b 45 f4 mov eax,DWORD PTR [rbp-0xc]

40064a: 48 98 cdqe

40064c: c6 04 02 01 mov BYTE PTR [rdx+rax\*1],0x1

400650: 48 8b 45 f8 mov rax,QWORD PTR [rbp-0x8]

400654: 8b 55 f4 mov edx,DWORD PTR [rbp-0xc]

400657: 89 50 04 mov DWORD PTR [rax+0x4],edx

40065a: 0f ae f0 mfence

40065d: 90 nop

40065e: b8 01 00 00 00 mov eax,0x1

400663: 2b 45 f4 sub eax,DWORD PTR [rbp-0xc]

400666: 48 8b 55 f8 mov rdx,QWORD PTR [rbp-0x8]

40066a: 48 98 cdqe

40066c: 0f b6 04 02 movzx eax,BYTE PTR [rdx+rax\*1]

400670: 84 c0 test al,al

400672: 74 0c je 400680 <\_Z13peterson\_lockR15peterson\_lock\_ti+0x48>

400674: 48 8b 45 f8 mov rax,QWORD PTR [rbp-0x8]

400678: 8b 40 04 mov eax,DWORD PTR [rax+0x4]

40067b: 3b 45 f4 cmp eax,DWORD PTR [rbp-0xc]

40067e: 74 de je 40065e <\_Z13peterson\_lockR15peterson\_lock\_ti+0x26>

400680: 5d pop rbp

400681: c3 ret

看图中红色标记的字段，根据gdb -tui a.out 调试的结果，这四句汇编顺序对应

lock.flag[id] = true;

lock.victim = id;

并没有编译时期的指令重排

加上-O2重新编译的结果如下

00000000004007a0 <\_Z13peterson\_lockR15peterson\_lock\_ti>:

4007a0: 48 63 c6 movsxd rax,esi

4007a3: c6 04 07 01 mov BYTE PTR [rdi+rax\*1],0x1

4007a7: 89 77 04 mov DWORD PTR [rdi+0x4],esi

4007aa: 0f ae f0 mfence

4007ad: b8 01 00 00 00 mov eax,0x1

4007b2: 29 f0 sub eax,esi

4007b4: 48 98 cdqe

4007b6: 80 3c 07 00 cmp BYTE PTR [rdi+rax\*1],0x0

4007ba: 75 04 jne 4007c0 <\_Z13peterson\_lockR15peterson\_lock\_ti+0x20>

4007bc: f3 c3 repz ret

4007be: 66 90 xchg ax,ax

4007c0: 39 77 04 cmp DWORD PTR [rdi+0x4],esi

4007c3: 75 f7 jne 4007bc <\_Z13peterson\_lockR15peterson\_lock\_ti+0x1c>

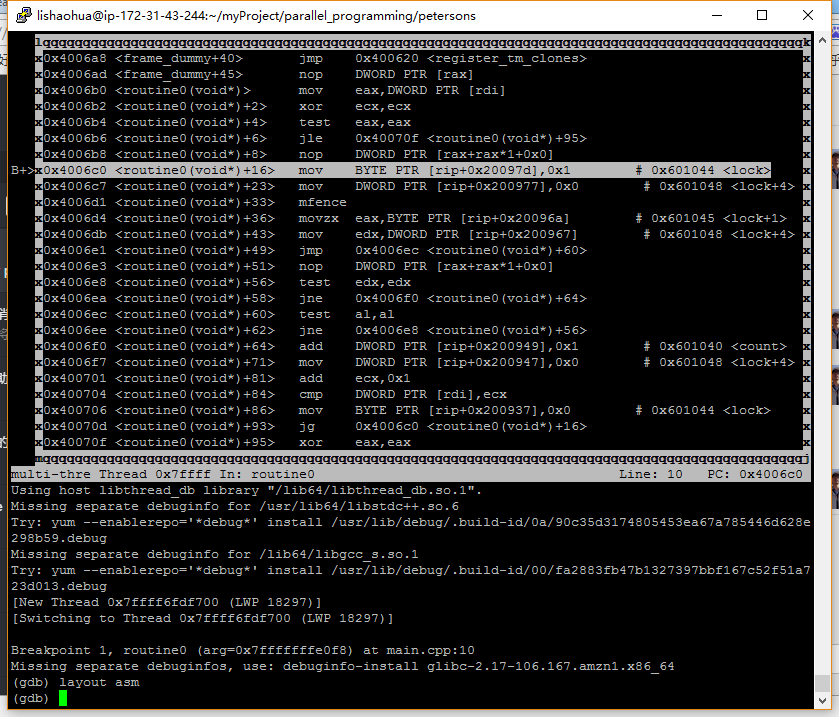
4007c5: 39 77 04 cmp DWORD PTR [rdi+0x4],esi

4007c8: 74 f6 je 4007c0 <\_Z13peterson\_lockR15peterson\_lock\_ti+0x20>

4007ca: eb f0 jmp 4007bc <\_Z13peterson\_lockR15peterson\_lock\_ti+0x1c>

4007cc: 0f 1f 40 00 nop DWORD PTR [rax+0x0]

以上是objdump的结果，但是在gdb -tui的时候，b peterson\_lock，运行并没有进入peterson\_lock函数，而是在routine函数里，说明-O2直接将peterson\_lock inline掉了。



直接执行以上红色标注出的语句，也就是victim的值是从esi里得的，这样Thread0和Thread1都会进入死锁状态。

然后-O3编译，再看汇编结果如下

00000000004007a0 <\_Z13peterson\_lockR15peterson\_lock\_ti>:

4007a0: 48 63 c6 movsxd rax,esi

4007a3: c6 04 07 01 mov BYTE PTR [rdi+rax\*1],0x1

4007a7: 89 77 04 mov DWORD PTR [rdi+0x4],esi

4007aa: 0f ae f0 mfence

4007ad: b8 01 00 00 00 mov eax,0x1

4007b2: 29 f0 sub eax,esi

4007b4: 48 98 cdqe

4007b6: 80 3c 07 00 cmp BYTE PTR [rdi+rax\*1],0x0

4007ba: 74 05 je 4007c1 <\_Z13peterson\_lockR15peterson\_lock\_ti+0x21>

4007bc: 3b 77 04 cmp esi,DWORD PTR [rdi+0x4]

4007bf: 74 07 je 4007c8 <\_Z13peterson\_lockR15peterson\_lock\_ti+0x28>

4007c1: f3 c3 repz ret

4007c3: 0f 1f 44 00 00 nop DWORD PTR [rax+rax\*1+0x0]

4007c8: eb fe jmp 4007c8 <\_Z13peterson\_lockR15peterson\_lock\_ti+0x28>

4007ca: 66 0f 1f 44 00 00 nop WORD PTR [rax+rax\*1+0x0]

相比-O2版本objdump的结果并没有多大改变，但是红色标注的块还是一样的，从寄存器读值存lock.victim

gdb -tui结果和-O2版本相同，运行结果都是死锁。

那么volatile在这里的作用就很明显了：避免从寄存器读取数据。

volatile在parallel programming里的作用属于hardware级别的，避免cpu将volatile变量的内容存进寄存器，必须直接存进cache或者memory，所以以上程序加上volatile之后才不会有死锁问题。