

compare switch with ifelse

```
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /data/switchifelse02...done.
(gdb) disassemble function_switch3
Dump of assembler code for function function_switch3(int):
0x0000000000400683 <+3>: sub    $0x1,%edi    将传入参数减1, 方便后续比较和索引。
0x0000000000400685 <+5>: xor    %eax,%eax
0x0000000000400688 <+8>: cmp    $0x2,%edi
0x000000000040068a <+10>: je     0x400701 <function_switch3(int)+17>    Compare edi寄存器的值, 由于这个是switch 1、2、3, 所以这里和2比较, 如果大则跳到<function_switch3(int)+17> (地址0x400691)
0x0000000000400691 <+17>: mov    0x4009c0(,%rdi,4),%eax    访问链表
0x0000000000400691 <+17>: repz retq
End of assembler dump.
(gdb) disassemble function_switch4
Dump of assembler code for function function_switch4(int):
0x00000000004006f0 <+0>: sub    $0x1,%edi
0x00000000004006f3 <+3>: xor    %eax,%eax
0x00000000004006f5 <+5>: cmp    $0x3,%edi
0x00000000004006f8 <+8>: ja     0x400701 <function_switch4(int)+17>
0x00000000004006fa <+10>: mov    0x4009c0(,%rdi,4),%eax
0x0000000000400701 <+17>: repz retq
End of assembler dump.
(gdb) disassemble function_switch3_jump
Dump of assembler code for function function_switch3_jump(int):
0x00000000004006a0 <+0>: cmp    $0xeb,%edi
0x00000000004006a6 <+6>: mov    $0xc8,%eax
0x00000000004006ab <+11>: je     0x4006ca <function_switch3_jump(int)+42>
0x00000000004006ad <+13>: cmp    $0x149,%edi
0x00000000004006b1 <+19>: mov    $0xb8,%ax
0x00000000004006b7 <+25>: je     0x4006ca <function_switch3_jump(int)+42>
0x00000000004006b9 <+27>: xor    %ax,%ax
0x00000000004006bc <+30>: mov    $0xa,%edx
0x00000000004006c1 <+37>: cmp    $0x9c,%edi
0x00000000004006c7 <+43>: cmovbe %edx,%eax
0x00000000004006ca <+42>: repz retq
End of assembler dump.
(gdb) disassemble function_switch4_jump
Dump of assembler code for function function_switch4_jump(int):
0x0000000000400710 <+0>: cmp    $0xdd,%edi
0x0000000000400716 <+6>: je     0x400748 <function_switch4_jump(int)+56>
0x0000000000400718 <+8>: jle    0x400738 <function_switch4_jump(int)+40>
0x000000000040071a <+10>: cmp    $0x180,%edi
0x0000000000400720 <+16>: mov    $0xbb,%eax
0x0000000000400725 <+21>: je     0x400733 <function_switch4_jump(int)+35>
0x0000000000400727 <+23>: cmp    $0x19b,%edi
0x000000000040072d <+29>: mov    $0x9c4,%ax
0x0000000000400731 <+35>: jne    0x400742 <function_switch4_jump(int)+50>
0x0000000000400733 <+37>: repz retq
0x0000000000400735 <+39>: nopl    (%rax)
0x0000000000400738 <+40>: cmp    $0x7c,%edi
0x000000000040073b <+43>: mov    $0xa,%eax
0x0000000000400740 <+48>: je     0x400733 <function_switch4_jump(int)+35>
0x0000000000400742 <+50>: xor    %eax,%eax
0x0000000000400744 <+52>: retq
0x0000000000400745 <+53>: nopl    (%rax)
0x0000000000400748 <+56>: mov    $0xc8,%eax
0x000000000040074d <+61>: retq
End of assembler dump.
(gdb)

Samples: 45 of event |branch-misses', Event count (approx.): 254979
Overhead Command Shared Object Symbol
85.24% switch02 libc-2.17.so [k] __random_r
2.94% switch02 libc-2.17.so [k] __random
5.42% switch02 switch02 [k] main
1.03% switch02 switch02 [k] random0plt
0.20% switch02 libstdc++.so.6.0.19 [k] dynamic_cast
0.14% switch02 [kernel.kallsyms] [k] filemap_map_pages
0.13% switch02 libc-2.17.so [k] __memcmp_sse2
0.00% perf [kernel.kallsyms] [k] x86_pmu_enable

Samples: 85 of event |branch-misses', Event count (approx.): 827477
Overhead Command Shared Object Symbol
91.02% ifelse02 libc-2.17.so [k] __random_r
5.15% ifelse02 ifelse02 [k] main
2.59% ifelse02 ifelse02 [k] function_ifelse8
0.05% ifelse02 libc-2.17.so [k] __random
0.04% ifelse02 libc-2.17.so [k] __newlocale
0.04% ifelse02 libstdc++.so.6.0.19 [k] std::locale::id:::_M_id
0.00% perf [kernel.kallsyms] [k] x86_pmu_enable
```

false sharing

大部分时候业务逻辑代码的优化是大头，但false sharing同样重要，当业务逻辑优化到一定程度后，这一块会是性能瓶颈。

就游戏服务器来说，大部分场景我觉得做成多进程单线程才是最优解，能避免使用锁或者false sharing造成的性能浪费。当某个应用场景是CPU计算密集型的时候才适合使用多线程。

如下截图是一个简单的测试代码，在64位机器上测试false sharing带来的消耗。

未做填充时的代码以及输出截图：

```
#include <pthread.h>
#include <iostream>
#include <chrono>
using namespace std;

struct T
{
    int a;
    int b;
};

T global;

static void *thread_start1(void *arg)
{
    do
    {
        ++global.a;
    } while(global.a < 100000000);
    return nullptr;
}

static void* thread_start2(void* arg)
{
    do
    {
        ++global.b;
    } while(global.b < 100000000);
    return nullptr;
}

int main()
{
    global.a = 0;
    global.b = 0;
    auto start = std::chrono::steady_clock::now();
    pthread_t threadid1;
    pthread_t threadid2;
    pthread_create(&threadid1, nullptr, &thread_start1, nullptr);
    pthread_create(&threadid2, nullptr, &thread_start2, nullptr);

    pthread_join(threadid1, nullptr);
    pthread_join(threadid2, nullptr);
    auto end = std::chrono::steady_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "Elapsed time: " << elapsed.count() << " ms\n";
    return 0;
}
```

```
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing
Elapsed time: 2026 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing
Elapsed time: 1886 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing
Elapsed time: 2048 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing
Elapsed time: 1755 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing
Elapsed time: 2384 ms
```

做填充时的代码以及输出截图：

```
#include <pthread.h>
#include <iostream>
#include <chrono>
using namespace std;

struct T
{
    int a;
    int full[15];
    int b;
};

T global;

static void *thread_start1(void *arg)
{
    do
    {
        ++global.a;
    } while(global.a < 100000000);
    return nullptr;
}

static void* thread_start2(void* arg)
{
    do
    {
        ++global.b;
    } while(global.b < 100000000);
    return nullptr;
}

int main()
{
    global.a = 0;
    global.b = 0;
    auto start = std::chrono::steady_clock::now();
    pthread_t threadid1;
    pthread_t threadid2;
    pthread_create(&threadid1, nullptr, &thread_start1, nullptr);
    pthread_create(&threadid2, nullptr, &thread_start2, nullptr);

    pthread_join(threadid1, nullptr);
    pthread_join(threadid2, nullptr);
    auto end = std::chrono::steady_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "Elapsed time: " << elapsed.count() << " ms\n";
    return 0;
}
```

```
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing2
Elapsed time: 90 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing2
Elapsed time: 129 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing2
Elapsed time: 134 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing2
Elapsed time: 98 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$ ./falsesharing2
Elapsed time: 93 ms
jsfeipos@feipalibb2gs2:~/temp/falsesharing$
```

内存序

本想对C++11几种内存序做些记录，但实际准备写的时候又觉得没必要，首先是不管开发还是实际线上运营都是在X86_64架构下，而且一般都是加锁。其次这块感觉也没什么难点需要特别记录的。之所以明确表明X86_64架构，是因为X86_64是TSO（强一致性内存模型），这种CPU只有Store Buffer，没有Invalid Queue。也就是说这种内存模型只有store load这种情况会出现乱序，也就在这种情况下如果需要保证一致性才需要加内存屏障。测试案例如下所示：

volatile关键字，表明变量是易变的，CPU每次都从主存上拿数据。

没有内存屏障代码截图以及测试结果如下图所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

volatile int x = 0, y = 0;
volatile int r1 = 0, r2 = 0;

void *thread1(void *arg) {
    x = 1;
    asm volatile("" ::: "memory"); //避免指令重排保证先执行x=1, 再执行r1=y
    //__sync_synchronize(); //内存屏障
    r1 = y;
    return NULL;
}

void *thread2(void *arg) {
    y = 1;
    asm volatile("" ::: "memory"); //避免指令重排, 保证先执行y=1, 再执行r2=x
    //__sync_synchronize(); //内存屏障
    r2 = x;
    return NULL;
}

int main() {
    int i, detected = 0;

    for (i = 0; i < 1000000; i++) {
        pthread_t t1, t2;

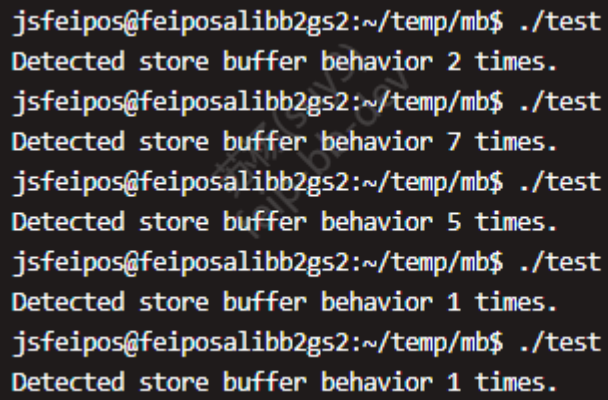
        x = 0; y = 0;
        r1 = 0; r2 = 0;

        pthread_create(&t1, NULL, thread1, NULL);
        pthread_create(&t2, NULL, thread2, NULL);

        pthread_join(t1, NULL);
        pthread_join(t2, NULL);

        if (r1 == 0 && r2 == 0) {
            detected++;
        }
    }

    printf("Detected store buffer behavior %d times.\n", detected);
    return 0;
}
```

A terminal window with a dark background and light-colored text. It shows five consecutive executions of a command to test for store buffer behavior. The results are: 2 times, 7 times, 5 times, 1 time, and 1 time. A faint watermark is visible in the background of the terminal.

```
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 2 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 7 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 5 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 1 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 1 times.
```

有内存屏障代码截图以及测试结果如下图所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

volatile int x = 0, y = 0;
volatile int r1 = 0, r2 = 0;

void *thread1(void *arg) {
    x = 1;
    //asm volatile("" ::: "memory"); //避免指令重排保证先执行x=1, 再执行r1=y
    __sync_synchronize(); //内存屏障
    r1 = y;
    return NULL;
}

void *thread2(void *arg) {
    y = 1;
    //asm volatile("" ::: "memory"); //避免指令重排, 保证先执行y=1, 再执行r2=x
    __sync_synchronize(); //内存屏障
    r2 = x;
    return NULL;
}

int main() {
    int i, detected = 0;

    for (i = 0; i < 1000000; i++) {
        pthread_t t1, t2;

        x = 0; y = 0;
        r1 = 0; r2 = 0;

        pthread_create(&t1, NULL, thread1, NULL);
        pthread_create(&t2, NULL, thread2, NULL);

        pthread_join(t1, NULL);
        pthread_join(t2, NULL);

        if (r1 == 0 && r2 == 0) {
            detected++;
        }
    }

    printf("Detected store buffer behavior %d times.\n", detected);
    return 0;
}
```



```
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 0 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 0 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 0 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 0 times.
jsfeipos@feiposalibb2gs2:~/temp/mb$ ./test
Detected store buffer behavior 0 times.
```

virtual table

```
class A
{
public:
    virtual void func1(){}
};
class B
{
public:
    virtual void func2(){}
};
class C : public A, public B
{
};
```

在64位linux平台上，C的大小毋庸置疑是16个字节。两个虚函数指针，但这两个虚函数指针指向的是哪呢？网上有不少讲这块的博客，基本上都说是指向两个虚函数表。那问题来了，是哪两个虚函数表？类A和类B的虚函数表？那类C的虚函数表呢？测试结果如下所示：

```
(gdb) p c
$1 = {<A> = {_vptr.A = 0x555555557cc0 <vtable for C+16>}, <B> = {_vptr.B = 0x555555557cd8 <vtable for C+40>}, <No data fields>}}
(gdb) x /32xg 0x555555557cb0
0x555555557cb0 <_ZTV1C>: 0x0000000000000000 0x0000555555557d48
0x555555557cc0 <_ZTV1C+16>: 0x0000555555555268 0xfffffffffffffffff8
0x555555557cd0 <_ZTV1C+32>: 0x0000555555557d48 0x0000555555555278
0x555555557ce0 <_ZTV1B>: 0x0000000000000000 0x0000555555557d80
0x555555557cf0 <_ZTV1B+16>: 0x0000555555555278 0x0000000000000000
0x555555557d00 <_ZTV1A+8>: 0x0000555555557d90 0x0000555555555268
0x555555557d10 <_ZTI1D>: 0x00007ffff7faad58 0x0000555555556005
0x555555557d20 <_ZTI1D+16>: 0x0000000200000000 0x0000555555557d90
0x555555557d30 <_ZTI1D+32>: 0x0000000000000002 0x0000555555557d80
0x555555557d40 <_ZTI1D+48>: 0x0000000000000002 0x00007ffff7faad58
0x555555557d50 <_ZTI1C+8>: 0x0000555555556008 0x0000000200000000
0x555555557d60 <_ZTI1C+24>: 0x0000555555557d90 0x0000000000000002
0x555555557d70 <_ZTI1C+40>: 0x0000555555557d80 0x0000000000000002
0x555555557d80 <_ZTI1B>: 0x00007ffff7faa008 0x000055555555600b
0x555555557d90 <_ZTI1A>: 0x00007ffff7faa008 0x000055555555600e
0x555555557da0: 0x0000000000000001 0x0000000000000001
```

先说结论，C的两个函数指针都是指向的vtable for C，只是指向的位置不一样。A的虚函数表起始地址是0x555555557cf0，B的虚函数表起始地址是0x555555557ce0，C的虚函数表起始地址是0x555555557cb0。可以看到C的虚函数表中具体存了哪些数据，首先在0x555555557cb8的值是0x0000555555557d48，0x0000555555557d48也可以从表中得出结论是C的typeinfo信息。然后是0x0000555555555268，该地址的反汇

编代码如下所示：

```
(gdb) disassemble 0x000055555555268
Dump of assembler code for function A::func1():
0x000055555555268 <+0>:    endbr64
0x00005555555526c <+4>:    push    %rbp
0x00005555555526d <+5>:    mov     %rsp,%rbp
0x000055555555270 <+8>:    mov     %rdi,-0x8(%rbp)
0x000055555555274 <+12>:   nop
0x000055555555275 <+13>:   pop     %rbp
0x000055555555276 <+14>:   retq
End of assembler dump.
```

然后又是0x0000555555557d48，最后是0x000055555555278，反汇编代码如下所示：

```
(gdb) disassemble 0x000055555555278
Dump of assembler code for function B::func2():
0x000055555555278 <+0>:    endbr64
0x00005555555527c <+4>:    push    %rbp
0x00005555555527d <+5>:    mov     %rsp,%rbp
0x000055555555280 <+8>:    mov     %rdi,-0x8(%rbp)
0x000055555555284 <+12>:   nop
0x000055555555285 <+13>:   pop     %rbp
0x000055555555286 <+14>:   retq
End of assembler dump.
```

由于C没有重写func1()和func2()，所以C里的函数地址和A、B的是一样的。综上所述，从逻辑上看是指向两个不同的虚函数表。从物理视角看是指向同一个虚函数表区域的不同部分。