ostringstream多线程下性能问题分析

2017-11-06

c++ stream 性能 多线程

最近在某 CPU 比较密集的高并发项目中，加了一段看起来影响不大的代码，上线后 CPU 占用率从 10% 上涨到 11%，觉得在预期内，于是没有太在意。到晚上 8 点左右访问高峰时期，CPU 占用率突然发生雪崩，暴涨至 50% 以上，大量请求失败。赶紧回滚代码，检查原因。

最终定位在一个看起来很简单的函数中，函数的功能是将几个整数拼成一个字符串，使用 `std::ostringstream` 生成字符串。这个类在多线环境下性能似乎特别差，为了验证这种推测，写了一个测试程序：

```
#include <sstream>
#include <string>
#include <stdio.h>
#include <sys/time.h>

std::string use_snprintf(int a) {
    char buf[64];
    snprintf(buf, sizeof(buf), "%d", a);
    return buf;
}

std::string use_stringstream(int a) {
    std::ostringstream oss;
    oss << a;
    return oss.str();
}

const int LOOPS = 1000000;

void *thread(void *p) {
    std::string (*foo)(int) = (std::string (*)(int))p;
    for (int i = 0; i < LOOPS; ++i)
        foo(i + 1);
    return p;
}
```

```
    for (int i = 0; i < threads; ++i)
        pthread_create(&tids[i], nullptr, thread, (void *)foo);
    for (int i = 0; i < threads; ++i)
        pthread_join(tids[i], nullptr);
    delete[] tids;

    gettimeofday(&end, nullptr);

    return (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) * 1e-6;
}

void test_with_threads(int threads) {
    printf("%d threads:\n", threads);
    double time_snprintf = run_with_threads(threads, use_snprintf);
    double time_stringstream = run_with_threads(threads, use_stringstream);
    printf("snprintf:        %f\n", time_snprintf);
    printf("stringstream:    %f\n", time_stringstream);
    printf("stream/snprintf: %f\n", time_stringstream / time_snprintf);
    printf("\n");
}

int main(int argc, char **argv) {
    if (argc > 1) {
        test_with_threads(atoi(argv[1]));
    } else {
        test_with_threads(1);
        test_with_threads(2);
        test_with_threads(4);
        test_with_threads(10);
        test_with_threads(20);
    }
}
```

在我个人的四核电脑上，运行结果如下：

```
1 threads:
snprintf:        0.088347
stringstream:    0.369389
stream/snprintf: 4.181115

2 threads:
snprintf:        0.082788
stringstream:    0.865383
stream/snprintf: 10.453000

4 threads:
snprintf:        0.085714
stringstream:    1.529721
stream/snprintf: 17.846804
```

```
20 threads:
snprintf:        0.428851
stringstream:    7.649718
stream/snprintf: 17.837706
```

可见，单线程时，`ostringstream` 的效率比 `snprintf` 差，但也没有到数量级的差距，而线程数增加后，`ostringstream` 的效率下降非常快。这时候，我们已经可以合理推测是 `ostringstream` 使用了线程锁所致了。

使用 perf 跑一把看看，前几名是：

```
Samples: 25K of event 'cycles:u', Event count (approx.): 21735936055
Overhead  Command    Shared Object        Symbol
  16.15%  bench.out  libstdc++.so.6.0.21  [.] std::locale::operator=
  16.04%  bench.out  libstdc++.so.6.0.21  [.] std::locale::locale
  15.06%  bench.out  libstdc++.so.6.0.21  [.] std::locale::~locale
   6.72%  bench.out  libstdc++.so.6.0.21  [.] std::has_facet<std::ctype<char> >
   6.44%  bench.out  libstdc++.so.6.0.21  [.] __dynamic_cast
   3.93%  bench.out  libstdc++.so.6.0.21  [.] std::use_facet<std::ctype<char> >
```

可以看到前三名都与 `std::locale` 有关，而且我们知道 `stringstream` 是依赖 locale 的。查阅 libstdc++ 源代码：

```
locale::locale(const locale& __other) throw()
: _M_impl(__other._M_impl)
{ _M_impl->_M_add_reference(); }

locale::~locale() throw()
{ _M_impl->_M_remove_reference(); }

const locale&
locale::operator=(const locale& __other) throw()
{
  __other._M_impl->_M_add_reference();
  _M_impl->_M_remove_reference();
  _M_impl = __other._M_impl;
  return *this;
}
```

引用计数！

继续往下查，`_M_impl` 的引用计数实现的核心就是简单的加一、减一原子操作。原子操作的效率是比普通读写低一些，但也不至于那么夸张，我只能想到 cache thrashing 一种解释。（Cache thrashing：

c++
nginx
性能 多线程
python http
cors epoll
protobuf
stream linux
xshell 游戏
google test
gtest stl
c++11 ssh

```
                0000003d8dcc13c0 <std::locale::~locale()@@GLIBCXX_3.4>:
  0.65          cmpq   $0x0,vtable for std::messages_byname<wchar_t
                push   %rbx
                mov    (%rdi),%rbx
  0.59        ↓ je     40
                mov    $0xffffffff,%eax
                lock   xadd   %eax,(%rbx)
 98.42  17:     cmp    $0x1,%eax
              ↓ jne    38
                test   %rbx,%rbx
              ↓ je     38
                mov    %rbx,%rdi
              → callq  std::locale::_Impl::~_Impl()@plt
                mov    %rbx,%rdi
                pop    %rbx
              ↓ jmpq   ffffffffffccec0
                nop
  0.21  38:     pop    %rbx
  0.13        ← retq
                nop
        40:     mov    (%rbx),%eax
                lea    -0x1(%rax),%edx
                mov    %edx,(%rbx)
              ↑ jmp    17
```

这上面显示 98.42% 的 CPU 时间耗在 `cmp $0x1,%eax` 上，这没有道理，于是看上一条，是 `lock xadd`，果然，原子操作！确认了！（由于 CPU 流水线的原因，采样结果偏差一条指令是非常正常的事情。）

以上实验都是在四核电脑上做的，尚且如此明显，在服务器上几十个核争抢一把锁，那就更严重得多，发生雪崩也就好理解了。

上一篇 下一篇