« 开发笔记(27)：公式计算机 | 返回首页 | 星际争霸2编辑器的初接触 »

并发问题 bug 小记

今天解决了一个遗留多时的 bug，想明白以前出现过的许多诡异的问题的本质原因。心情非常好。

简单来说，今天的一个新需求导致了我重读过去的代码，发现了一个设计缺陷。这个设计缺陷会在特定条件下给系统的一个模块带来极大的压力。不经意间给它做了一次高强度的压力测试。通过这个压力测试，我完善了之前并发代码中写的不太合理的部分。

一两句话说不清楚，所以写篇 blog 记录一下。

最开始是在上周，阿楠同学发现：在用机器人对我们的服务器做压力测试时的一个异常状况：机器人都在线的时候，CPU 占用率不算特别高。但是一旦所以机器人都被关闭，系统空跑时，CPU 占用率反而飚升上去。但是，经过我们一整天的调试分析，却没有发现有任何线程死锁的情况。换句话说，不太像 bug 导致的。

但是，一旦出现这种情况，新的玩家反而登陆不进去。所以这个问题是不可以放任不管的。

后来，我们发现，只要把 skynet 的工作线程数降低到 CPU 的总数左右，问题就似乎被解决了。不会有 CPU 飚升的情况，后续用户也可以登陆系统。

虽然问题得到了解决，但我一直没想明白原因何在，心里一直有点不爽。

待到开发过了一段落，我又想回头来分析这个问题。发现，在 CPU 占用率很高的时候，大量 CPU 都消耗在对全局消息队列进出的操作上了。看起来是工作线程空转造成的。这个时候，如果设置了上百个工作线程，消息队列中有没有太多消息的话，处理消息队列的 spin lock 的碰撞就恶化了。那些真正有工作的线程有很大几率拿不到锁。

我的全局消息队列处理的业务很简单，就是存放着系统所有服务的二级消息队列。每个工作线程都是平等的，从中取得一个二级消息队列，处理完其中的一个消息，然后将其压回去。

这时我发现，其实，这个全局队列完全可以实现成无锁的结构，这样就不会再有锁碰撞的问题了。

原来的锁最重要的用途是在全局消息队列容量不够时，保护重新分配内存的过程不被干扰。但实际上，全局消息队列的预容量大于系统中服务体的数量的话，是永远够用的。我设置了单台机器支撑的服务体数量上限为 64K 个，那么消息队列预分配 64K 个单元，就无需动态调整。

昨天我着手实现无锁版的循环队列，原本以为也就是几行代码的事情，但事实上隐藏的坑挺深。

当线程特别多时，任何一个线程都可能暂时被饿死。那么，即使我们原子的移动了循环队列的指针，也无法保证立刻就从队列头部弹出数据。这时，纵然队列容量大到了 64K，而队列中的数据只有几个，也无法完全避免队列的回绕，头尾指针碰撞。

为了保证在进队列操作的时序。我们在原子递增队列尾指针后，还需要额外一个标记位指示数据已经被完整写入队列，这样才能让出队列线程取得正确的数据。

一开始，我让读队列线程忙等写队列线程完成写入标记。我原本觉得写队列线程递增队列尾指针和写入完成标记间只有一两条机器指令，所以忙等是完全没有问题的。但是我错了，再极端情况下（队列中数据很少，并发线程非常多），也会造成问题。

后来的解决方法是，修改了出队列 api 的语义。原来的语义是，当队列为空的时候，返回一个 NULL，否则就一定从队列头部取出一个数据来。修改后的语义是：返回 NULL 表示取队列失败，这个失败可能是因为队列空，也可能是遇到了竞争。

我们这个队列仅在一处使用，在使用环境上看，修改这个 api 语义是完全成立的，修改后完全解决了我前面的问题，并极大的简化了并发队列处理的代码。

有兴趣的同学可以看看代码。最前面 30 行操作 globalmq 的代码既是。

由于简化了代码，使得我上周的问题更清晰的展现出来。但问题的解决并非这么直接。

今天， mike 同学要求在底层增加一个接口，可以把一个消息同时发给许多目标。虽然我之前实现了组播服务，但 make 同学不希望额外维护一个组对象，而是希望每次主动提交一个目标地址列表。

我开始为组播服务增加新接口。

这时，我发现原来的代码写的有些过于复杂了。这个复杂性来至于优化。

在 skynet 系统中，大都不直接引用服务对象，而是记录一个数字 handle 。等到要发送消息时，再从 handle 转为一个 C 对象。这个查询转换有一定的代价（hash 查询），但可以保证 C 对象的生命期管理容易实现。

事实上，这个简化方案，在复杂的并发环境中也坑过我一次。可见前面一篇 blog 。

没想到这次的问题还是和这里有关。

我为了优化组播过程，缓存了 C 对象，相对应的，对这些保持的 C 对象指针做了引用计数加一。等到它们离组的时候再减少。

另外，分组并不是用 hash set 实现的，离组操作代价比较高。所以又是在组播过程中成批处理的。也就是说，当一个对象退出，并不能保证它即使的从分组中拿掉，那么其引用记数就无法正确的减少到 0 。

而之前我在删除这些对象附着的消息队列时，采用的策略是，先删除对象，再将其消息队列标记成可删除，并把消息队列压入全局队列，让全局队列分发函数去检查标记并真正删除。

到现在，我就完全明白了上周问题的成因：当用户退出，系统删除了它对应的 handle ，并试图删除对应的消息队列。而对象被某个组播分组引用，阻止了退出流程，迟迟未能标记消息队列为可删。然后工作线程就反复的做退出队列，重回队列的操作，浪费了大量的 CPU 。同时，在这种情况下，全局消息队列的冲突变得非常严重，早先的 spinlock 未能合理的处理这种不健康状态，导致了整个系统性能的下降。

云风 提交于 October 12, 2012 02:10 PM | 固定链接

COMMENTS

好多东西看不懂

Posted by: AUHS | (15) June 15, 2014 12:51 PM

好多东西看不懂

Posted by: AUHS | (14) June 15, 2014 12:50 PM

我不知道该说点啥

Posted by: dndxsys | (13) April 21, 2014 01:32 PM

你的DEFAULT_QUEUE_SIZE设定为64
global_queue的head和tail类型都是uint32_t最大值为0xFFFFFFFF
而(0xFFFFFFFF+1)%64刚好为0
即uint32_t溢出后刚好模为0
因为看你文章刚好需要64K
所以想问DEFAULT_QUEUE_SIZE是你设计为64还是碰巧？＝＝
DEFAULT_QUEUE_SIZE的值需要刚好与(0xFFFFFFFF+1)模为零的吧
不然溢出后似乎结果会不正确

Posted by: wandefou | (12) December 25, 2012 03:26 PM

路过，留下点什么

Posted by: dndxsys | (11) November 12, 2012 11:35 AM

@Cloud

看起来是不需要的。

另：

```
struct message_queue *
skynet_globalmq_pop() {
struct global_queue *q = Q;
uint32_t head = q->head;
uint32_t head_ptr = GP(head);
if (head_ptr == GP(q->tail)) {
return NULL;
}

if(!q->flag[head_ptr]) {
return NULL;
}

struct message_queue * mq = q->queue[head_ptr];
if (!__sync_bool_compare_and_swap(&q->head, head, head+1)) {
return NULL;
}
q->flag[head_ptr] = false;

return mq;
}
```

这里q->flag[head_ptr]=false也有延迟。如果获取队列的线程自己处理这个队列的消息话，这个不会引发一致性问题，然而，如果这个队列后来被交接给别的线程处理，则会引发一致性问题，因为那个接收的线程可能很块处理完消息，然后又有很快把消息队列放如global q, 那么你最初获取这个队列的线程对这个q->flag[head_ptr] = false执行就太滞后了，覆盖了那个线程设置为true的结果。不过在X86上可能不会发生，因为写总是顺序的。当把队列交付给别的线程时，总会发生内存写。
而你已经写了flag为false.

看来这个情况，在你的程序里不会发生。

Posted by: David Xu | (10) October 16, 2012 01:49 PM

@David Xu

我在

q->flag[tail] = true;

之后加了一句

__sync_synchronize();

之前似乎出现过读不到这个标记的情况.

不过

if(!q->flag[head_ptr])

之前加

__sync_synchronize();

应该没有特别的必要.

Posted by: Cloud | (9) October 16, 2012 10:35 AM

Lock-less的代码是很难做对的，特别是对于memory order以及可见性是很难掌控而且各种体系结构之间也有区别，移植性方面存在不确定。
例如，在X86上读是可以乱序执行的，写是顺序的，但是也要在编译器不重新安排指令顺序的情况下。
static void

```
skynet_globalmq_push(struct message_queue * queue) {
struct global_queue *q= Q;

uint32_t tail = GP(__sync_fetch_and_add(&q->tail,1));
assert(!q->flag[tail]);
q->queue[tail] = queue;
__sync_synchronize();
q->flag[tail] = true;
}
```

象q->flag[tail]=true就不保证别的CPU是立刻可见的，它可能被CPU内部的write buffer queue缓冲。除非你在后面跟上一句write memory barrier，迫使write buffer queue被刷出去。否则很可能存在写延迟，导致一致性问题或者效率问题。

```
struct message_queue *
skynet_globalmq_pop() {
struct global_queue *q = Q;
uint32_t head = q->head;
uint32_t head_ptr = GP(head);
if (head_ptr == GP(q->tail)) {
return NULL;
}

if(!q->flag[head_ptr]) {
return NULL;
}
```

象这种语句，CPU不保证读的顺序，它很可能在读出q->tail之前先读出q->flag[head_ptr]。除非你在中间插入read memory barrier.

没有等待的忙轮询式SPINLOCK也可能导致低效率和消耗过度的电能。用户态SPINLOCK通常被实行成spin几百个循环，然后进内核等待。spin循环中还插入电能节省代码，X86有pause指令，执行一条pause指令可让CPU暂时下降电能的消耗。更加积极的SPINLOCK的实现，还实行指数级的退避，当LOCK是上锁的时候，线程以指数方式退避，防止CACHE LINE在各CPU之间不停地PING-PONG，cmpset指令会导致PING-PING，无辜消耗电能和占用CPU之间的通讯带宽。
不阻塞的SPINLOCK， 如果锁的拥有者被抢先，那么别的线程将无限制地spin，最糟糕的情况下，会消耗很多的CPU时间。例如拥有者和需要锁的线程在同一个CPU上运行，当后者抢先了前者时，可能需要一个调度的时间片来降低自己的动态优先级，让前者机会跑起来，然后释放锁拱后者使用。

无论是Linux还是FreeBSD都可以在初始化pthread_mutex时设置是否为spin lock，他们的内部实现都是spin多少个周期，然后进入睡眠。

Linux和FreeBSD都不是公平锁的mutex，公平锁是先进先出的，而 Linux和FreeBSD 不保证这种次序。
需要实现这样的锁时，可以使用他们的mutex，然后自己建立先进先出队列，这样可以把不公平性降到最低。

Posted by: David Xu | (8) October 16, 2012 09:54 AM

并发问题 真是头疼

Posted by: cx | (7) October 14, 2012 10:54 PM

也要看是做什么的线程，保证不阻塞的情况下才是这样。
>
在有限处理器核心的机器上，开太多线程是降低了处理器效率，应考虑线程池，保证处理器核心数和活跃线程数基本一致最好，实在没必要开成百线程

Posted by: Anonymous | (6) October 14, 2012 07:41 AM

呵呵,我并不同意 ccc 的看法,现在已经出问题了,就要解决问题,而现在是解决问题的同时重构了下,这些事情还是不要太晚做的好.

Posted by: lite3 | (5) October 13, 2012 09:56 AM

优化与否是一个不能用教条来判定的问题。

组播这个特性就是优化的产物，否则把需要发给多个对象的消息用循环依次发送出去即可。

如果组播这个特性必须存在，那说明它是用来解决一个性能热点的。其实现就必须被很好的优化。否则整个模块就不需要被实现。

Posted by: cloud | (4) October 13, 2012 09:42 AM

程序设计第一原则：能不优化时就尽量不要优化。

你看看你没事就优化，结果增加了工作量。而且看你的描述，这个优化还增加了模块之间的耦合度。

Posted by: ccc | (3) October 13, 2012 08:08 AM

在有限处理器核心的机器上，开太多线程是降低了处理器效率，应考虑线程池，保证处理器核心数和活跃线程数基本一致最好，实在没必要开成百线程

Posted by: nori/twinkling | (2) October 12, 2012 06:01 PM

这个地方如果用互斥锁，性能上会差多少？

Posted by: lo | (1) October 12, 2012 05:02 PM

POST A COMMENT

非这个主题相关的留言请到：留言本

名字：

Email 地址：

为了验证您是人类，请将六加一的结果（阿拉伯数字七）填写在下面：

URL:

记住我的信息？

留言：
（不欢迎在留言中粘贴程序代码）

提交