搜狗实验室技术交流文档

Vol4:2 C10K 与高性能程序续篇

摘要

本文介绍了如何利用流水线和一些锁的技巧提高服务器吞吐量，以及新兴的 Lock Free 技术。

Pipe Line

流水线是已经在大搜索中广泛应用的技术。流水线解决 4 个问题：充分利用多 CPU，尽量按照 FIFO 的原则处理用户请求，防止长请求阻塞短请求，提高磁盘 I/O 效率。

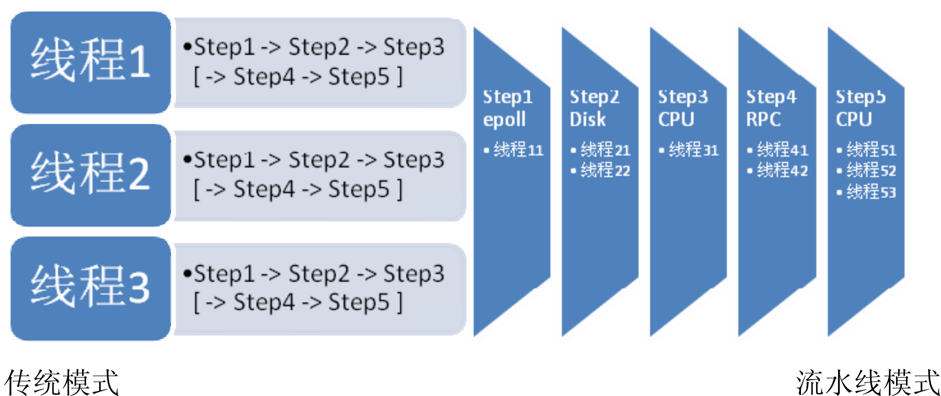
在第一篇技术通讯中讲到用 epoll 等技术在单线程中调度 socket I/O 才能达到最高的效率。单线程的 epoll 类方法有一个前提是对每个用户请求，都不能占用太多的 CPU 来处理应用逻辑。否则很容易出现一个 CPU 被占满，其它 CPU 空闲的情况。

总耗时较长且 CPU 和 I/O 交替进行的任务，如果是普通的每任务 1 线程，那么由于 OS 调度的随机性，对用户请求的响应很难做到先进先出。

用户请求的处理可以分为多个步骤，每个具体请求需要执行的步骤数目可能有差别。比如对 cache 的访问，命中和未命中就差别很大，未命中时需要等待后台服务的处理。每任务 1 线程的情况下，处理步骤较多的长请求会占据全部的工作线程。这时候本服务器往往是 IDLE 的，但是却没有空闲的线程处理那些步骤较少的短请求。

一次用户请求往往需要多次磁盘 I/O，普通程序都是串行的执行它们。这样操作系统失去了内部合并和重排序 I/O 的机会。

流水线技术把一次任务拆为多个步骤，每一步都有各自专门的线程来处理，每个步骤称为流水线的一级。拆分的原则是每一级都关心不同的资源：CPU，网络 I/O，磁盘 I/O，和其他服务的 I/O 等等。每一级流水线有自己的线程或者线程池，各级之间通过消息队列通信。



传统模式　　　　流水线模式

流水线继承了 epoll 类技术线程数不会膨胀的优点，同时通过步骤拆分充分的利用了多个 CPU。

线程数少了以后，由于消息队列的存在，FIFO 现在很容易保证。

流水线中，长请求只会占用流水线的后级，短请求仍然可以利用流水线的前集快速响应。这极大地提高了并发。

负责磁盘 I/O 的任务步骤现在可以很容易的用多个线程来并行发起 I/O。流水线前级把多个磁盘 I/O 请求打入 I/O 队列，I/O 线程处理完后再打入后级，后级每收集完全一组请求需要的数据后执行对应操作。

一些简单的同步技巧

对于一些简单的需求，例如全局统计、引用计数等等，可以归结为对整数的原子计算。可以使用一些轻量级的原子计算方法，比如 Win32 下的 Interloced 系列函数。Interloced 函数粗略的可以分为两族：一族以 InterlocedAdd 为代表，对应 C 语言中的 A += B 模式；一族以 InterlocedCompareAndExchange 为代表，提供了 CAS 操作（Compare – And - Swap）的能力。基于 CAS 操作，我们可以在一些更为复杂的需求上实现 Lock Free。Linux 下有 cmpxchg 函数实现 CAS（kernel-devel）。

Double Check 已经在 Singleton 模式中被广泛使用，这里就不多介绍了。

对于大数据集的并发操作，为防止并发线程在同一处阻塞，可以考虑把大数据集拆分成多个小数据集，对每个小数据集单独加锁。这样既提高了并发度，又不会消耗太多的资源。

pthread_spinlock 可以用来在少量线程间同步一些非常短小的操作，例如引用计数。但是切忌，可能并发访问的线程不能过多，这些线程的优先级不能有差别。

Lock Free

OS 提供的 mutex 互斥体这样的线程/进程同步手段，从好的一面来说，只要互斥体是在锁状态，你就可以放心地进行任何操作，不用担心其它线程会闯进来搞坏你的共享数据。但是也有很多弊病，用 mutex 的粒度过大容易导致线程闲置，粒度过小又容易导致各种 dead lock/love lock。尤其是使用 spin lock 这种轻量级锁的时候，一旦各个线程的优先级不一致，几乎一定会出现优先级倒置带来的死锁。为此人们提出了 Wait-Free 等待无关和 Lock-Free 锁无关的概念。

一个“Wait-Free”的程序可以在有限步之内结束，而不管其它线程的相对速度如何。一个“Lock-Free”的程序能够确保执行它的所有线程中至少有一个能够继续往下执行。这便意味着有些线程可能会被任意地延迟，然而在每一步都至少有一个线程能够往下执行。因此这个系统作为一个整体总是在“前进”的，尽管有些线程的进度可能不如其它线程来得快。

基于锁的程序则无法提供上述的任何保证。一旦任何线程持有了某个互斥体并处于等待状态，那么其它任何想要获取同一互斥体的线程就只好站着干瞪眼；如果两个线程互相等待另一个线程所占有的 mutex，就只好僵死。优先级倒置、信号中断、异常中止都是多线程程序的噩梦，但是 Lock-Free 对它们是免疫的。

毫无疑问，落实到最终实现，Lock-Free 算法总要对应一些具体的原子操作，现在主要用到的就是 CAS（Compare – And - Swap）以及它的一些变种。

LL SC CAS DCAS 与 CAS2

LL 和 SC 是 lock free 理论领域研究的理想原语。

LL [addr], dst 从内存单元[addr]读取值到 dst。

SC value, [addr] 若自本线程上次从[addr] LL 以来，没有任何 SC 操作在[addr]上，则把[addr]的值设为 value

LL 和 SC 都是原子操作。实际上没有任何 CPU 直接实现了 SC 原语，一般我们用 CAS 类原语来部分模拟 SC。

CAS 原语负责比较某个内存地址处的 1 字长的内容与 1 个期望值，如果比较成功则将该内存地址处的内容替换为一个新值。这整个操作是原子的。现代 CPU 一般都支持 CAS 操作。常用来做指针替换和原子计算。

对应的 DCAS 原语比较 2 个内存地址处的 1 字长的内容与 2 个期望值，如果成功则将这 2 个内存地址处的内容替换为 2 个新值。这也是原子操作。经典的用法是同时处理指针和其引用计数。

现在实现 DCAS 的 CPU 还很少。作为一种折衷，CAS2 原语比较某个内存地址处的 2 字长的内容与 2 个期望值，如果比较成功则将该内存地址处的内容替换为 2 个新值。x86/x64 的 CPU 支持 CAS2 原语。

使用这些原语有一个基本前提，CPU 对 1 字长的内存数据简单存储是原子的。具体到 x86，"movl"这样的操作在地址已对齐的情况下是原子的，不会受到 SMP 的干扰。

<table>
<tr><th></th><th>Definition</th><th>Implementation</th></tr>
<tr><td>CAS</td><td><pre>bool CAS(intptr_t* addr, intptr_t oldv, intptr_t newv)
atomically {
    if ((*addr == oldv) {
        *addr = newv;
        return true;
    } else
        return false;
}</pre></td><td>x86: cmpxchg
Windows
InterlockedCompareAnd
Exchange
x64: cmpxchg8b
Windows
InterlockedCompareAnd
Exchange64</td></tr>
<tr><td>DCAS</td><td><pre>bool DCAS(intptr_t* addr1, intptr_t* addr2,
          intptr_t old1, intptr_t old2,
          intptr_t new1, intptr_t new2)
atomically {
    if ((*addr1 == old1) && (*addr2 == old2)) {
        *addr1 = new1; *addr2 = new2;
        return true;
    } else
        return false;
}</pre></td><td>N/A</td></tr>
<tr><td>CAS2</td><td><pre>bool CAS2(intptr_t (*addr)[2],
          intptr_t old1, intptr_t old2,
          intptr_t new1, intptr_t new2)
atomically {
    if (((*addr)[0] == old1) && ((*addr)[1] == old2)) {
        (*addr)[0] = new1; (*addr)[1] = new2;</pre></td><td>x86: cmpxchg8b
Windows:
InterlockedCompareAnd
Exchange64
x64: cmpxchg16b</td></tr>
</table>

```
            return true;
        } else
            return false;
    }
```

注意：SMP 环境下使用 comxchg 系列指令时，需要 lock 前缀。

Spin Lock

Spin Lock 是一种轻量级的同步方法，它和 mutex 具有近似的语义。但是当 lock 操作被阻塞时，并不是把自己挂到一个等待队列，而是死循环 CPU 空转等待其他线程放锁。这样节省了 OS 切换线程上下文的开销。如果确信需要同步的操作总是能在数条指令内完成，那么使用 Spin Lock 会比传统的 mutex lock 要快一个数量级。基于 CAS 原语设计很容易实现 spin lock

```
struct spinlock {
    int v;
}
```

```
void spin_init(spinlock* sl)
{
    sl->v = 1;
}
```

```
void spin_lock(volatile spinlock* sl)
{
    do { nop; } while (!CAS(&sl->v, 1, 0))
}
```

```
void spin_unlock(volatile spinlock* sl)
{
    sl->v = 1;
}
```

典型的，spin lock 可能被用于修改计数器，或者同步一些浮点运算。

pthread 提供了 spin lock，请参阅 pthread_spin_lock。

Lock-Free LIFO Stack

借助 CAS 可以很大程度上避免同步操作，先来看一个用 CAS 原语实现的简单 LIFO Stack。

```
struct cell {
    struct cell* next;
    struct value_t value;
}
```

```
struct lifo {
    volatile struct cell* top;
}
```

```
void lifo_push(volatile lifo*lf, cell* cl) {
    do {
        cl->next = lf->top;
    } while (!CAS(&lf->top, cl->next, cl))
}
```

```
cell* lifo_pop(volatile lifo*lf) {
    cell* head, *next;
    do {
        head = lf->top;
        if (head == NULL) break;
        next = head->next;
    } while (!CAS(&lf->top, head, next))
    return head;
}
```

容易看出 Lock Free 算法基于乐观假设，采用了 commit-retry 的模式。当同步冲突出现的机会很少时，这种乐观假设能带来较大的性能提高。

do {

备份旧数据

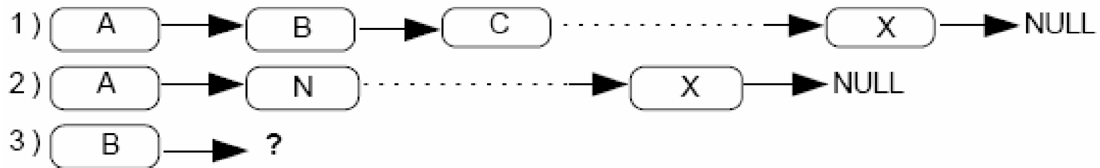
搜狐公司研发中心版权所有，仅供技术交流，转载请保留上述文字

基于旧数据构造新数据

} while (!CAS(内存地址，备份的旧数据，新数据))

ABA Problem

如果不涉及到内存回收，上一小节的程序已经相当完善(比如说在有 GC 的环境中)。一旦涉及到显式的内存管理，就会遇到经典的 ABA 问题。



1) lifo_push 被抢先之前的链表
2) 抢先结束后的链表
3) lifo_push 的结果

如上图所示。如果执行 lifo_pop 的线程，在“head = lf->top;”后被其它线程的 pop 抢先，经过多次 push/pop 后返回到原线程时可能出现原有的链表头在释放后又被重新分配的情形。也就是说 lf->top 的值从 A 变成 B 再变成 A 这一过程发生在“head = lf->top;”和 CAS 之间，这时 lf->top->next 已经不等于原来保存的 next，CAS 没有办法检测这种变化（这正是 CAS 比 SC 弱的地方）。

LIIFO with Sequence Number

为避免 ABA 问题，我们用一个顺序号 lf->pop_count 来保护 pop 操作，如果 pop 操作被其它的 pop 抢先，就 retry 一下。此时对 lf->top 和 lf->pop_count 的修改应该在同一个原子操作中进行——CAS2 的用武之地。

```
struct cell {
    struct cell* next;
    struct value_t value;
}
```

```
struct lifo {
    volatile struct cell* top;
    volatile int pop_count;
}
```

```
void lifo_push(volatile lifo*lf, cell* cl) {
    do {
        cl->next = lf->top;
    } while (!CAS(&lf->top, cl->next, cl))
}
```

```
cell* lifo_pop(volatile lifo*lf) {
    cell* head, *next;
    int oc;
    do {
        head = lf->top; oc = lf->pop_count;
        if (head == NULL) break;
        next = head->next;
    } while (!CAS2(lf, head, oc, next, oc+1))
    return head;
}
```

封装好的 ifo 可以用来做 memory pool 的 free list，这样很容易就得到了一个 lock free 的 memory pool。

x8632bit 下对 CAS 和 CAS2 的实现

Windows 下直接用 InterlockedCompareAndExchange。Linux 下麻烦一点，这里给出 linux 下的实现。

```
uint32_t CAS(uint32_t *ptr, uint32_t old, uint32_t new)
{
    bool ret;
    __asm__ __volatile__("lock cmpxchgl %1,%2 \n\t sete %%al"
                 : "=a"(ret)
                 : "r"(new), "m"(*ptr), "0"(old)
                 : "memory");
     return ret;
}
```

```
bool CAS2 (volatile void * ptr, uint32_t old1, uint32_t old2, uint32_t new1, uint32_t new2)
{
    bool ret;
    __asm__ __volatile__ ("lock cmpxchg8b (%1) \n\t sete %%al"
        :   "=a"(ret)
        :   "D"(ptr), "a"(old1), "d"(old2), "b"(new1), "c"(new2)
        :   "memory");
    return ret;
}
```

进阶阅读

- Lock-Free Techniques for Concurrent Access to Shared Objects
  http://www.grame.fr/pub/fober-JIM2002.pdf
- Hazard pointers: safe memory reclamation for lock-free objects
  http://researchweb.watson.ibm.com/people/m/michael/ieeetpds-2004.pdf
- 一个 Write-Rarely-Read-Many Map
  http://blog.csdn.net/chinajxw/archive/2006/03/08/618850.aspx

这一篇文章的 Update 函数只能单线程使用，否则可能访问一个悬空指针。需要在 Update 中也使用 Hazard 才可以并行 Update。当然 Write-Rarely 一般是不会并行 Update 的。

- SUN 的 Lock-Free Reference Counting
  http://research.sun.com/people/moir/Papers/SPAA04.pdf
- Lock-Free Linked Lists Using Compare-and-Swap
  http://citeseer.ist.psu.edu/cache/papers/cs/1067/ftp:zSzzSzftp.cs.rpi.eduzSzpubzSzvaloisjzSzpodc95.pdf/valois95lockfree.pdf
- transactional memory 简介
  http://www.newsmth.net/att.php?p.272.21110.693.pdf
- GCC-Inline-Assembly-HOWTO
  http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html
- Intel® 64 and IA-32 Architectures Software Developer's Manuals
  http://www.intel.com/products/processor/manuals/index.htm