基于RCU平衡树的可扩展地址空间

Scalable Address Spaces Using RCU Balanced Trees

背景介绍

读者-写者问题描述的是在多线程共享数据的场景下，其中一些线程要对数据进行修改，另一些线程只读取这些数据而不作修改。对这一问题常用的解决方法有两类，一类是对数据加读写锁，使得任何时刻同时对数据进行操作的只有若干读者或只有一个写者；另一类则是读-复制-更新（read-copy-update, RCU）的机制，写者并不直接修改数据，而是将要修改的部分复制一份新的，在最后只更新一个指针。RCU机制保证了读者的效率，但是并不是对于每一种数据结构和每一种应用场景都可以直接套用的，每种情况都需要具体的分析。

在操作系统中就存在这样一种应用场景，在页表和页目录中，操作系统会维护一个快速查找的数据结构，通常是平衡二叉树。在这棵二叉树上，内存映射（mmap/munmap）系统调用会新增/删除节点，而缺页的处理只会修改某个节点所映射的地址，而不改变树结构。在Linux内核中，这棵树上会加读写锁，使得在内存映射的同时页表不会被访问。这样的机制使得在内存映射的同时，同一进程的其它线程中所有的缺页处理都将被阻塞，造成可扩展性问题。

该论文设计了一种基于RCU的平衡二叉树，BONSAI树。对这种数据结构的只读查找可以完全可扩展地并行，从而使得缺页的处理不会被内存映射阻塞，提高了内存管理系统的可扩展性。

RCU数据结构

RCU基本原理

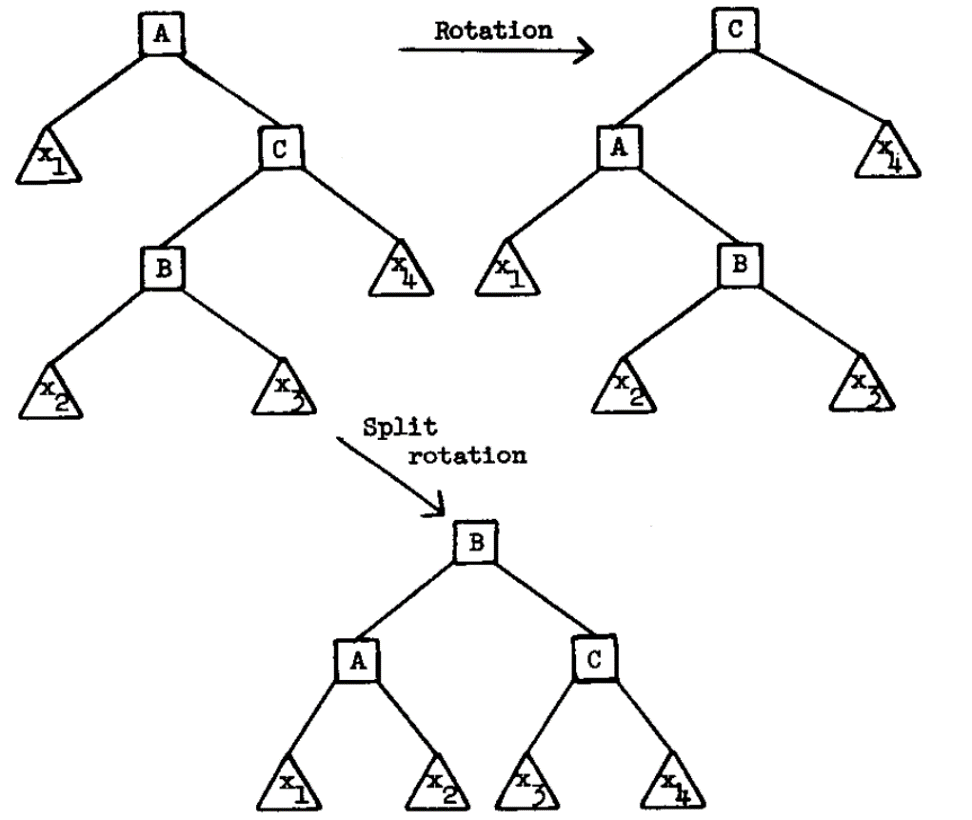
RCU的原理非常简单，在一些场景下，要求对一个数据结构的修改满足原子性，修改的中间状态不会被其它线程看到；如果对这个数据结构的修改只是修改一个指针，就很容易用无锁的方式实现。而实际的数据是在新分配的一块内存中创建其新版本，在完全创建好后再将指针改为指向新版本。此时可能还有其它线程在读取旧版本的数据，旧版本数据所占的内存还不能释放，只能加入到延迟释放表，在之后的某个时刻释放。

BONSAI树

Nievergelt和E.M.Reingold [1]提出的加权平衡树（weight-balanced tree / bounded-balanced tree）是一种平衡二叉树，在树的每个节点上存储了以该节点为根的子树节点数size。加权平衡树维护其平衡性质：对任意节点

其中是一个可调参数；越大，查找时间越长，插入和删除后的旋转次数越少；反之则查找时间越短，插入和删除后的旋转次数越多。可以证明，当

时，若插入或删除导致某个节点不满足上述性质，可以通过最多两次旋转使其恢复。假设节点的右子树太大，设为其右子节点，为的左子节点；若的右子树比左子树大，则左旋；若的左子树比右子树大，则先右旋，再左旋。而的左子树太大的情况则完全对称。



BONSAI树是在此基础上利用RCU机制设计的可并发查找的平衡二叉树。按照严格的RCU原理，对树做修改时能修改的只有指向根节点的指针。当插入一个节点时，需要将从插入位置到根的整条路径上的节点都复制一份；当删除一个节点时，若节点左右孩子都非空，则需要复制从该节点的后继到根的整条路径；否则需要复制从该节点到根的整条路径。

但是复制整条路径带来的开销还是非常大。实际上没有必要遵循严格的RCU原理。当对一棵树的修改只发生在某棵子树上时，只需要将从修改处到的根节点的路径，并更新的父节点上指向的指针即可。具体来说，插入、删除叶子节点和删除半叶子节点不需要任何复制，只需要修改插入或删除的节点的父节点上的指针。而删除一个有两棵非空子树的节点，需要复制从该节点到其后继的路径。

更新了指针之后，插入或删除的效果即对其他线程可见了。虽然此时还没有完成旋转，但二叉树的平衡性不会影响查找的正确性，此时的并发读访问已经可以正确地查找到结果了。之后可能需要进行一次旋转或两次旋转，修改其中两个或三个节点上的指针。同样的，旋转过程只需要复制这两个或三个节点，并更新父节点上的一个指针。

要注意到，BONSAI树支持一个写者与多个读者的并发执行，但并不支持多个写者的并发执行。当两个写者读取同一个旧版本的数据，并各自创建自己的一份复制和；那么最后更新指针后，读者所能看到的要么是，要么是，而不是正确的或。

虚拟内存空间管理

Linux

Linux的虚拟内存管理中，页表通过红黑树来索引，整棵树加了读写锁mmap\_sem，每个页表项和页目录项上还加了细粒度的锁。在内存映射操作时，必须获取mmap\_sem写锁；而在缺页处理时，必须获取mmap\_sem读锁和所缺失的页上的细粒度锁。这使得缺页处理的并行度较低，成为了程序可扩展性的障碍。为此，作者提出了三种优化方案。

缺失锁

注意到对树的修改并不会改动每一个节点，而是只改动其中一部分节点。对树中每个节点加读写锁。可以使缺页处理不必获取全局读写锁mmap\_sem，而在内存映射操作时，获取所有需要修改的节点上的写锁。这种方案本质上还是受限于锁的性能，只能获得较小的提升。

混合锁/RCU

在这种方案下，同样希望缺页处理不必获取全局读写锁。但是这样会在缺页处理和内存映射会产生数据竞争。有三种会产生数据竞争的情况，可以逐一检测，一旦在缺页处理时发现这样的数据竞争，则重试；在重试时，获取mmap\_sem读锁以确保第二次成功。

缺页处理，首先需要查找到对应的页。第一类数据竞争是在并发的读和修改时，使得查找失败。为了解决这类数据竞争，在查找失败时直接加速重试即可。

第二类数据竞争发生在munmap操作时。当munmap操作删除了树中一个节点时，有可能并发的缺页处理会访问到这个节点上；若该节点内存已被释放，则会读到无效数据。和RCU处理旧版本内存一样，这类数据竞争的解决方式就是延迟释放。

第三类冲突是在munmap操作时，一个页目录下的所有页都被回收后，对应的页目录项会被标记为已删除并清空。而当一个缺页处理程序看到该目录项时，就会误以为需要分配新的页表项和一个新的页，而它们之后再也不会被释放。为了解决这类数据竞争，在缺页处理程序准备分配页表项之前，需要检查该目录项是否被删除，且是否仍然包含所要映射的页。由于缺页处理程序和munmap都需要获取该页目录项的细粒度锁，所以要么缺页处理程序检查到错误并重试，要么munmap发现并释放了缺页处理程序所分配的页。

纯RCU

纯RCU的方式即用前面提到的BONSAI树来替换带锁的红黑树。前面提到，BONSAI树支持一个写者与多个读者的并发执行，但并不支持多个写者的并发执行。所以在内存映射时仍然需要获取锁，但是缺页处理不需要获取锁，所以相互之间以及与内存映射之间都可以并行。

实验

将修改后的Linux内核上运行在8个Intel E7-8870 (10-core)的机器上，并运行Metis、Psearchy和Dedup三个测试程序。纯RCU的性能分别是原版读写锁的3.4倍、1.8倍和1.7倍，混合锁/RCU 的性能分别是原版读写锁的2.7倍、1.7倍和1.6倍，而缺失锁相比原版读写锁的性能提升不明显。

结论

随着处理器核数的增加，内核的一些操作将成为多线程的虚存密集型应用的性能瓶颈。为了提升可扩展性，该论文提出了基于读-复制-更新的平衡二叉树，BONSAI树，来管理虚拟内存空间，使得缺页的损失成为与处理器核数无关的常数。该数据结构可应用在很多用平衡树来索引页表的系统，解决这些系统上虚存管理的可扩展性问题。

参考文献（原文除外）

[1] J. Nievergelt, E. M. Reingold. Binary Search Trees of Bounded Balance. SIAM Journal on Computing. 2: 33