**高级操作系统论文阅读报告**

**姓名：施宇 学号：2017211384**

# Non-scalable locks are dangerous

## 论文背景

锁在操作系统中是很常见的一种同步机制。但是，一些锁由于设计问题，当有许多个核同时竞争锁保护的资源的时候，会导致performance严重下降，例如Linux内核中的ticket spin lock。这些锁被称为non-scalable lock。使用了non-scalable lock进行同步的程序，随着核的数目增多，一开始throughput会不断增加，但是核的数目增加到了到了某一个点的时候，throughput会突然急剧下降，甚至和单核的情况差不多(collapse)，图1给出一个例子。这篇文论利用一个马尔可夫模型从数学的角度来解释这种现象，由此说明non-scalable lock的现象不是硬件问题，而是锁的算法本身问题。进而说明了non-scalable lock对系统效率的危害。

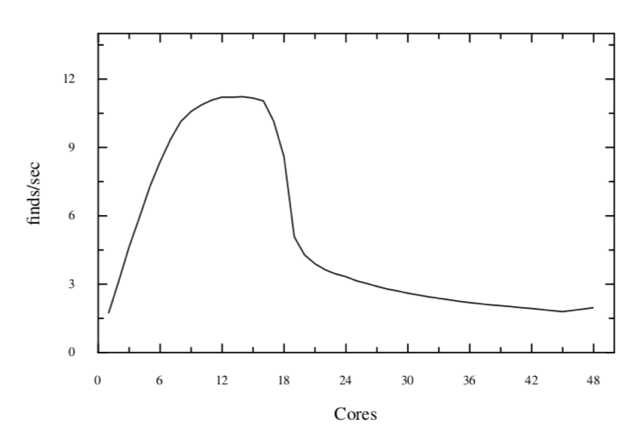


图1 Collapse现象

## 为什么一些锁的可扩展性差

在内核中，一些锁是在多个处理器核之间共享的，如果锁的状态被其中一个核所改变，由于cache coherence，可能会导致其他所有核cache当中的锁需要进行更新，这样的更新是串行的，因此代价随着核数目线性增长。ticket spin lock为例，图2是ticket spin lock的伪代码。

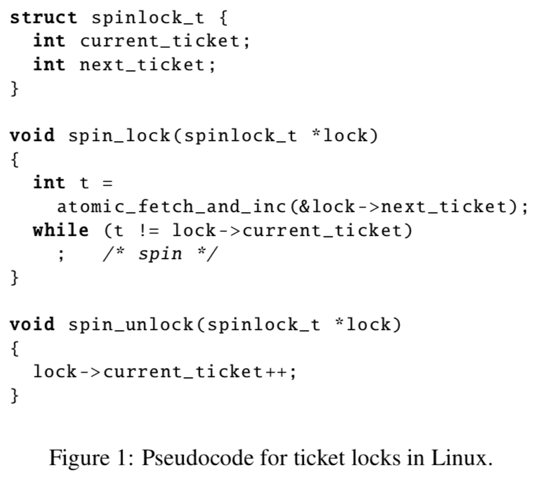


图2 ticket spin lock

Ticket spin lock维持了current\_ticket和next\_ticket两个计数值。当lock被释放的时候，current\_ticket会加一。进入临界区之前，首先要把next\_ticket的值并加一，记录在变量t当中。这个t的值相当于排队拿到的号码。只有当current\_ticket的值和t的值相等的时候，才说明轮到了当前的进程进入临界区。否则就一直处于忙等状态。

由于lock是被所有核共享的，并且每个核都在不断地读取lock的current\_ticket值，因此每次unlock的时候，对current\_ticket的更新必须要同步到其他所有核的cache中。这个代价随着核的数目线性增长。

## 利用Markov模型对Collapse现象进行建模

对于一个n核的系统，针对ticket spin lock建立一个有(n+1)个状态的Markov模型。如图3所示，每个状态表示当前有多少个核正在等待这个资源，即有多少个核正在被lock住。每个状态之间有对应的转移概率。

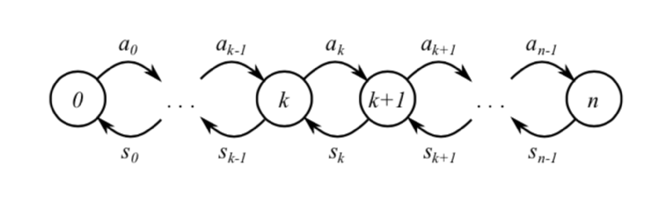


图3 表示正在竞争资源的核数目的Markov模型。

为了计算转移概率，论文做出了如下假设：

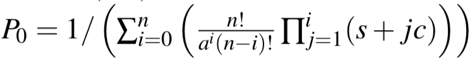
1. 每个核每隔a时间就会试图访问该资源一次。
2. 临界区的花费的平均时间为s。
3. 传输更新的lock状态到另一个核所需要的时间为c。

利用以上假设，可以计算出模型中的转移概率和。首先，由于每个核每隔a时间就会试图访问内存一次，在状态k，共有n-k个核处于空闲状态，因此在下一时刻增加一个试图访问资源的核的概率正比于，这就是从状态k转移到k+1的概率。当一个核进入临界区时，首先需要花费s的时间在临界区中，退出临界区后，需要把更新的锁的状态传递给其他正在等待的核的cache。由于这个传递是线性的，如果有k个核正在等待，则每个核收到更新的锁的状态需要的平均时间为。因此从一个核进入临界区，到下一个核进入临界区，大概需要的时间。因此，从状态k转移到状态k-1的概率正比于。

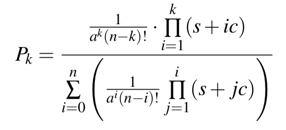
现在来估计每个状态的稳定概率。假设状态k的稳定概率为，则应该满足，由此可以解出



利用所有状态的概率加起来等于1，可以得到



最终得到状态k的概率



根据这个概率就可以计算出每一时刻有多少个核处于等待状态：

从而可以计算得到实际的speedup约为

在n为48的情况下，画出预测的speedup如图4所示。其中红线是利用模型的公示计算的，黑线是实际模拟出来的。每条曲线的临界区长度不同。可以看到两者非常吻合。

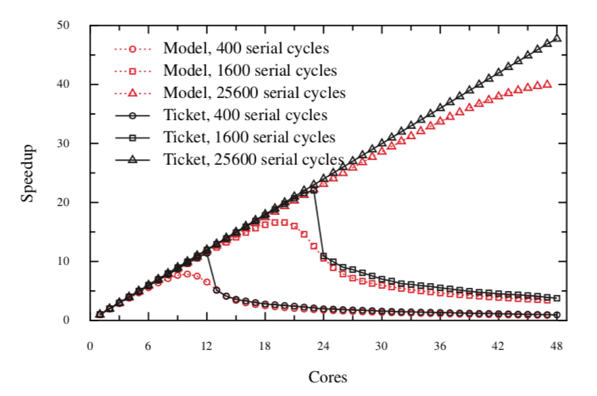


图4 模型的结果与实际吻合

从图4中我们观察到，只有当临界区很短的时候，才会出现collapse的现象。这从模型当中也可以得到解释，因为如果临界区的长度s很大，那么转移概率=中的ck相对s很小，因此核的数目k的影响可以忽略。相反，如果s很小，那么k的影响是巨大的。文章最后做了一些实验比较了scalable lock和non-scalable lock的性能，结果也与模型相符。

# Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks

上一篇论文已经证明了non-scalable lock的危害。这篇文论提出了一种几乎不需要memory barrier和原子操作的scalable读写锁。

## 论文背景

在操作系统中，有许多数据结构要求可以被多个读者和写者并行访问。为了保证正确性，需要确保在写者进行写操作的时候没有其他写者或读者在临界区内。读写锁(Reader-writer lock)，简称rwlock就是用来维护这一机制的。传统的rwlock在读者的common path（即正常的，非特殊情况的执行路径）中有一些原子操作，或者需要memory barrier。这些操作导致了读者在读者数目多的情况下，rwlock的可扩展性较差。这篇文章提出了一种新的算法prwlock，在TSO (total store ordering) 体系结构中避免了在读者的common path中的原子操作以及memory barrier，保证了对于读者数目的可扩展性。此外论文提出的算法还允许读者在critical section中进入睡眠状态。

## 传统的读写锁

可以分为两类，第一类是基于mutex来进行实现的，每个读者（或者每个core）对应一个mutex，在读者进入临界区之前，必须获得改mutex。而对于每一个写者来说，要想进入临界区，需要获得所有的mutex。其过程如图5中的Brlock1所示。由于lock mutex的开销很高，因此这样的锁会导致写者的开销随着读者数目快速增长。例如，在一个64核的机器上，处理读者的那个核需要获得其他所有核的mutex。在Linux Kernel 2.6版本中就有这样的锁(brlock)。另一类读写锁使用了一个数组brlockArray来记录每个读者（每个核）的当前是否在临界区中，该数组由所有的读者和写者共享。读者在读之前只需要把brlockArray里面对应的位置设置成1，然后等待写者退出临界区。而写者在写之前，需要先获得writerLock

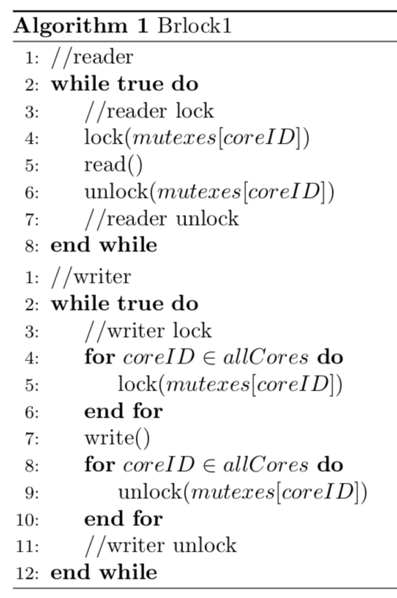
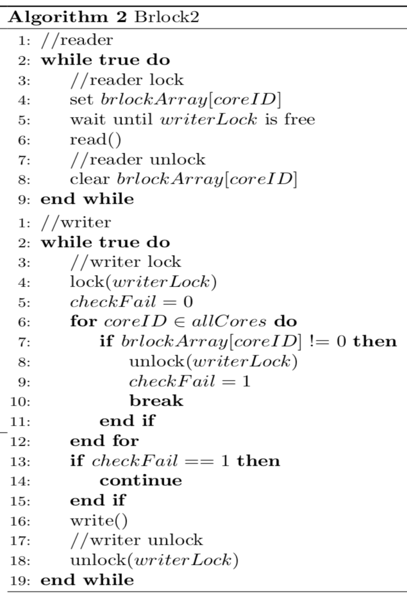
 

图5 两种传统的读写锁

然后检查每个写者的brlockArray里对应的位置是否被设置成1。如果检查失败，则放弃writerLock并重新尝试。其过程在图5的Brlock2中。相比于Brlock1，在Brlock2中读者不需要进行mutex操作，写者也只需要获得一个mutex。但是由于CPU的乱序执行，在读者set brlockArray[coreID]之后需要一个memory barrier。否则在brlockArray[coreID]的值被写入内存之前，写者可能已经进入了临界区，并且读者已经执行了read()当中的指令，这样就不能起到读写者互斥的作用。

综上，传统的brlock中，要么写者的开销巨大，要么读者当中需要memory barrier或者mutex等昂贵的操作。这导致brlock的效率在读者数目多的情况下很低。

## PRWLock

为了达到读者和写者互斥，在读者开始读之前需要知道写者是否在临界区内，同样的在写者开始写之前需要确保没有读者在临界区内。也就是读者写者需要告知对方自己的状态，而。在Brlock1算法中，这样的状态是通过mutex传递的。只要写者获得了所有的mutex，就可以知道没有读者在临界区中。在Brlock2算法中，这样的状态传递是通过writerLock和brlockArray共同实现的。在prwlock中，这样的状态记录在一个全局version变量里。一个写者在写操作之前都要先把这个全局version变量增加一。每个读者所在的核都有一个本地的version，读者在进入和退出临界区的时候都需要把本地的version更新到和全局的verison一致。此外，写者还有一个writer锁，用来记录是否有写者正在准备进入临界区。prwlock的基本思路如图6所示。

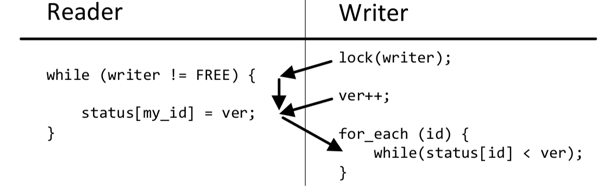


图6 利用version实现读写锁的思路

在图6中，写者开始写之前，首先需要获得writer锁。然后把version增加一。读者的本地version存储在status数组里。由于读者在进入和离开临界区的时候，都会把version更新到和全局的version一致，因此如果发现有读者的本地version落后于全局version，就说明该读者可能在临界区内，这个时候writer就会一直等待。

这个思路当中存在一个问题：读者离开临界区之后可能再也不会进入临界区，从而本地verison得不到更新。为了处理这种情况，在写者发现有读者的version落后于最新的version时，会主动发出一个Inter Process Interrupt (IPI)来强制更新该读者的本地version。论文当中用了一个bounded staleness的argument，即只要写者更新了version，其他读者在较短时间内都能看到并更新最新的version，从而需要IPI的情况极少，因此IPI的开销很低。

另一个问题是，如果读者在临界区中休眠，那么无法接收到IPI。如果读者离开临界区以后，再也不访问临界区，本地version就总是得不到更新，也会导致写者无休止地等待下去。为了解决这个问题，prwlock引入了一个active计数器，一个写者在进入睡眠状态的时候会把自己标记为ACTIVE状态，然后把active计数器加一。离开临界区的时候，如果发现自己处于ACTIVE状态，再把active计数器减一。写者在进入临界区之前会检查active计数器是否为0，如果不为0则等待处于ACTIVE状态的读者退出临界区。

增加了IPI和active计数器的完整版prwlock算法如图7所示。ScheduleOut是读者在临界区内部进入睡眠状态之前调用的。Report是读者收到IPI之后执行的操作。

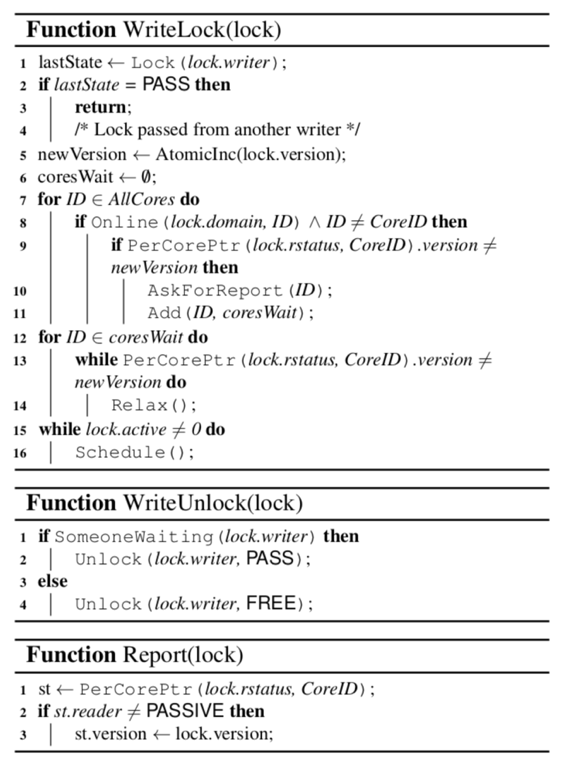
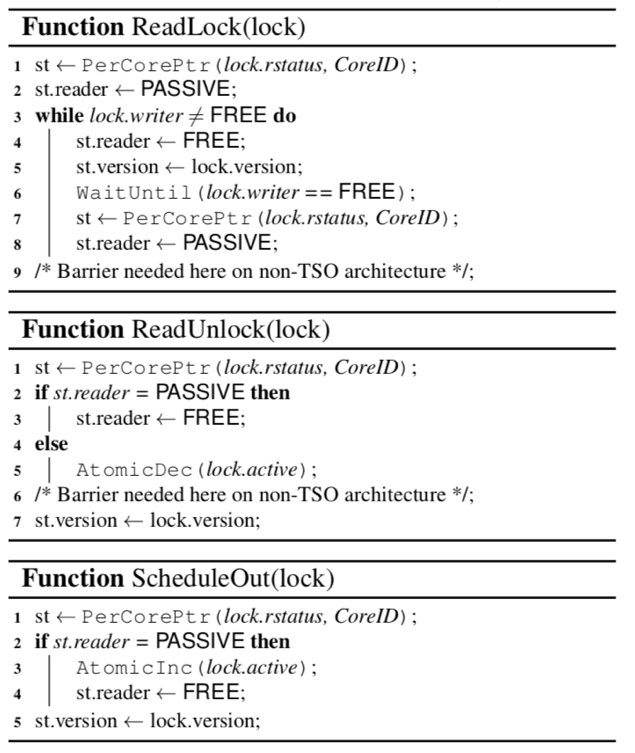


图7 prwlock算法

# Scalable Address Spaces Using RCU Balanced Trees

## 论文背景

多线程通过shared memory的方式进行并行非常高效。维护线程所需要的开销很小，并且同过shared memory可以直接在多个线程之间进行通信，比起进程之间用message passing进行通信同步高效许多。但是多线程并行存在一个问题，就是他们共享了一些内核中的数据结构，例如用来记录进程虚拟地址空间的平衡树，以及进程的页表。各个线程都有可能触发page fault，以及把一块虚拟地址空间映射到一个文件(mmap操作)，或者取消映射(mummap)，这些操作涉及到线程共享的平衡树和页表等数据结构，因此并行起来比较困难，这就导致了多线程性能可扩展性的一个瓶颈。

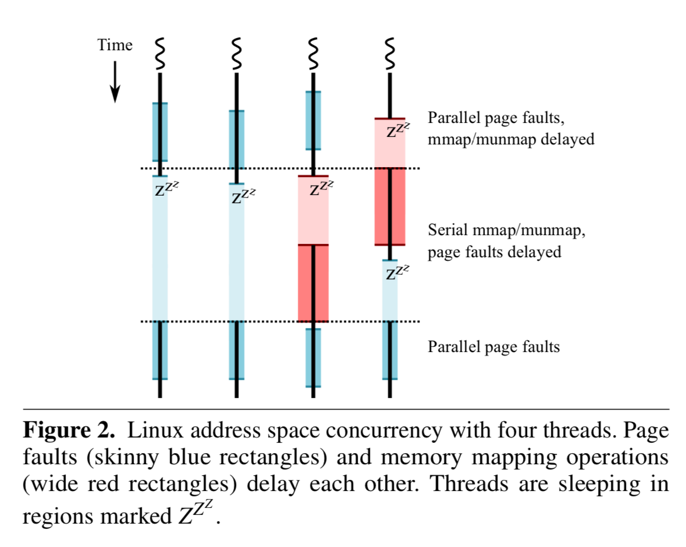


图8 多线程的page fault, mmap, munmap无法完全并行

使用RCU来允许读和写操作同时进行是Linux内核中增强可扩展性的常用方法。RCU是Linux kernel的一种锁机制，它允许读者和写者同时操作。写者首先拷贝一份数据，然后在备份数据上写，等到读者完成读操作之后，再把指向原数据的指针指向备份数据。但是，RCU只直接支持对单个指针的操作。而平衡树的旋转要想保证正确性，需要几个操作对读者而言都是原子的。在图9中，一个旋转操作需要改变3个指针。如果这3个改变不是原子的，假设一个读者想要访问Z，如果对a的右孩子指针的改变发生在读者获得a指针，并决定访问a的右孩子之后，就会导致读者最终访问到了Y。

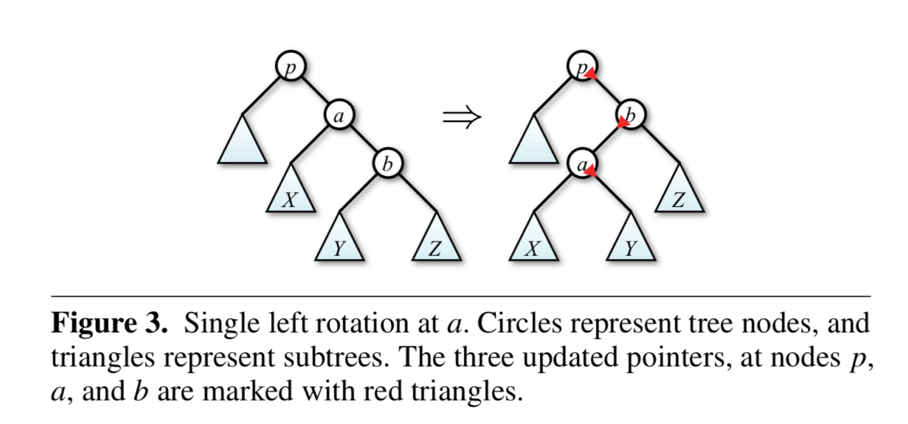


图9 旋转操作需要同时改变多个指针

这篇论文设计了一个高效的平衡树（BONSAI tree），对他的旋转操作可以用RCU来保证原子性，从而允许读和写同时进行。用这样的平衡树管理地址空间，就可以让page fault和mmap操作并行执行，从而提高多线程的可扩展性。

## BONSAI tree算法

算法的基本设计思路比较简单。在进行旋转的时候，对每一个要改变的指针，都直接创建一个新的指针，而保持久的指针不动。最后对旧的指针然后调用rcu\_free()，这样在所有读者退出临界区之后，这些旧指针才会被回收，从而确保了读者访问的安全性。

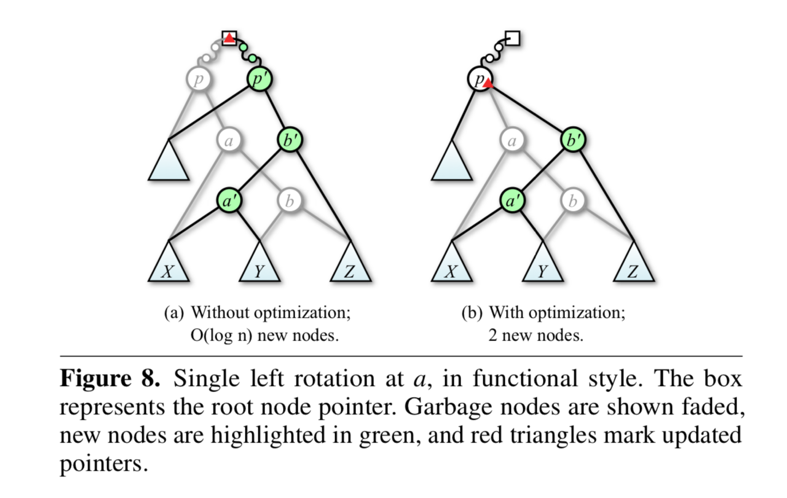
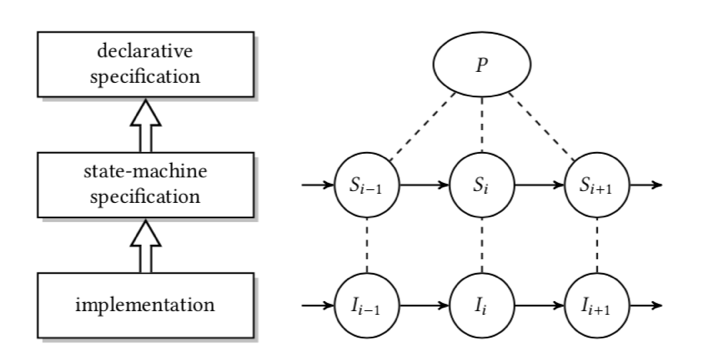


图10 只重新创建必要的指针

如果每一个路径上的指针全部都要重新创建，每次旋转操作都会带来O(log n)的内存分配和回收开销。在图10中，如果节点p的右孩子b经过旋转操作的b’已经平衡了，那么只需要把p节点的右孩子指针重新赋值为b’即可，无需创建一个新的p’节点。通过这样的优化操作可以把空间的分配和回收开销降到常数级别。

# Hyperkernel: Push-Button Verification of an OS Kernel

论文提出了一种基于Z3验证器设计可证明正确性的操作系统内核的方法。首先，把内核的状态抽象成一个python类。然后用python写内核接口的stat-machine specification，规定系统调用前后内核状态的变化。然后用c进行底层的实现，把实现编译成LLVM IR。再把LLVM IR中的变量和specification中的关联起来。通过展开LLVM IR找到所有的执行路径，然后利用Z3验证器验证在所有可能路径下实现的系统调用对系统状态的影响是否都符合state-machine specification的规定。此外，开发者还可以通过写高级的declarative specification来对state-machine specification的正确性进行检验。

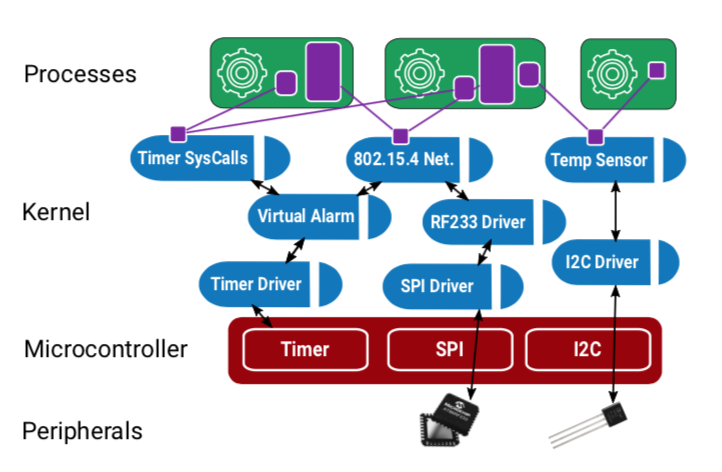


对于Hyperkernel的一些看法：

1. 所有的接口都只能是有限的，因此很多操作要丢给用户空间。例如由于回收内存的操作往往需要循环，因此hyperkernel要求用户通过多次系统调用不断地显式指定想要回收的page，这样给导致系统的易用性大大降低。
2. 只有state-machine specification是正确的，才能保证最后系统的正确性。而state-machine specification中需要手动编写的各种执行路径一定要考虑全最终实现的所有可能执行情况。即便有declarative specification，也不能保证state-machine specification一定符合设计者的初衷。

# Multiprogramming a 64 kB Computer Safely and Efficiently

过去的嵌入式系统一般都只处理特定的任务，然而近期出现了一下需要多任务并行、可动态装载的嵌入式应用，例如运动手表中需要同时运行记录、计时等功能。因此设计支持并发的嵌入式操作系统就成为了一个关键问题。论文提出了一种在内存极小的嵌入式处理器中的并行操作系统设计。由于嵌入式处理器没有内存管理单元，无法使用虚拟内存技术，因此如何实现进程间的隔离是一个挑战。



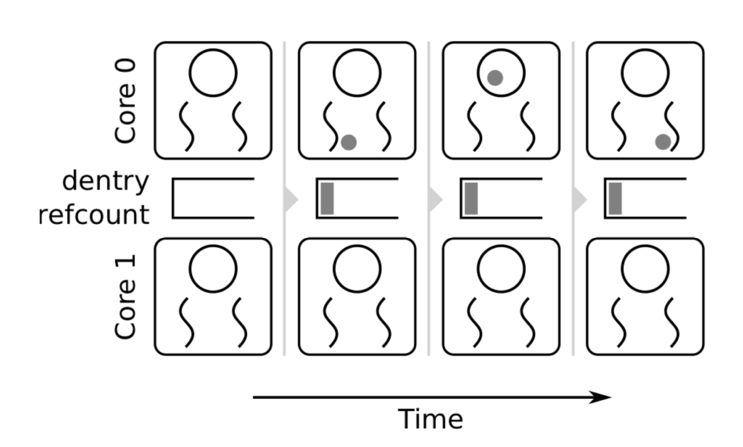
文章提出了Tock操作系统架构。其中与硬件直接交互的内核部分全部使用类型安全的Rust语言来设计。在Rust中数组越界等问题都可以被类型系统捕获，因此使用Rust编写内核部分极大提高了系统的可靠性。而一般的进程可以用任何语言进行编写，通过系统调用与内核进行交互。

Tock利用了嵌入式处理器的内存保护单元实现了进程间的隔离。首先，Tock把内核的堆空间划分成grants，进程通过系统调用来访问grants。Grants可以动态地分配给各个进程。由于对grant的访问是由Rust的类型系统来保障的，因此保证了不会发生越界的情况。

# An Analysis of Linux Scalability to Many Cores

论文分析了Linux在48核机器上的可扩展性，确定内核代码中影响可扩展性的bottleneck。在Linux中，有许多在被多核共享的计数器，同样地由于硬件的coherence需要，如果有许多核更新这些计数器，他们在核之间的传递就会变成一个严重的bottleneck。利用Sloopy counter减少对全局共享计数器的访问。

下图说明了sloopy counter的一个例子，首先core 0上的第一个线程把全局计数器减一。然后当这个线程需要对全局计数器做加操作时，不是直接往全局计数器上增加1，而是



往core0的本地计数器上加一。当core0上的另一个线程要对全局计数器减1时，先检查本地计数器是否大于等于1，如果是，则直接从本地计数器减一。因此loopy counter保证了所有本地计数器的值加上全局计数器，等于不使用loopy counter时的全局计数器值。这样做在保证原来的语义的情况下，减少了对全局计数器的访问，从而提高了系统的可扩展性。