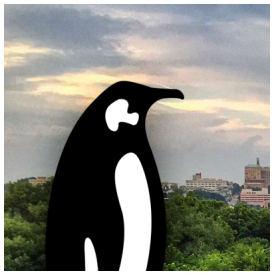

Contribution ID: **71** Type: **not specified**

Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLocks

Wednesday, 15 November 2023 17:15 (45 minutes)

Today's high-performance applications heavily rely on various synchronization mechanisms, such as locks. While locks ensure mutual exclusion of shared data, their design impacts application scalability. Locks, as used in practice, move the lock-guarded shared data to the core holding it, which leads to shared data transfer among cores. This design adds unavoidable critical path latency leading to performance scalability issues. Meanwhile, some locks avoid this shared data movement by localizing the access to shared data on one core, and shipping the critical section to that specific core. However, such locks require modifying applications to explicitly package the critical section, which makes it virtually infeasible for complicated applications with large code bases, such as the Linux kernel, which comprises 28M LoC with more than 180k static lock call sites.

We propose transparent delegation, in which a waiter automatically encodes its critical section information on its stack and notifies the combiner (lock holder). The combiner executes the shipped critical section on the waiter's behalf using a lightweight context switch. Using transparent delegation, we design a family of locking protocols, called TCLocks, that requires zero modification to applications'logic. We have implemented a spinlock, a delegation-based blocking lock, and a phase-based reader-writer lock. TCLocks also handles nested locking and out-of-order (OOO) unlocking, which is heavily used in the Linux kernel. While booting the kernel, we measure around ~80k nested locking calls and ~20k OOO calls.

Naively using transparent delegation breaks the assumptions of Linux kernel code about the stability of access to per-CPU variables under special execution contexts like interrupt handlers, non-preemptible or non-migratable contexts. One solution is to switch the *gs* register for x86 architecture to provide per-CPU variables of the waiter's CPU. However, it will lead to subtle bugs in the combiner phase unless we annotate parts of the kernel code which require access to the combiner's per-CPU variables, such as the scheduler, RCU, etc. We adopt a conservative approach of disabling combining for such execution contexts and fallback to *qspinlock* (Linux kernel spinlock).

The evaluation after replacing spinlock, mutex, rwlock and rwsem in the Linux kernel shows that TCLocks provide up to 5.2× performance improvement compared with recent locking algorithms. Detailed information about achieving transparent delegation for different types of locks can be found in our OSDI'23 paper and our implementation.

Enabling transparent delegation for the kernel still requires answering several questions like how does resource accounting works or how does delegation work with the *current* macro? Since the combiner thread executes the waiter's critical section, correct CPU/memory accounting must be done for the combiner and waiter thread. Similar to per-CPU variables, the Linux kernel uses the *current* macro for multiple purposes, including resource accounting for cgroups, permission checks using credentials, thread scheduling etc. The *current* macro has to be handled correctly within the combining phase otherwise it can lead to subtle bugs. We plan to discuss possible solutions to these problems to enable transparent delegation for the kernel.

Primary author: GUPTA, Vishal

Presenter: GUPTA, Vishal

Session Classification: LPC Refereed Track

Track Classification: LPC Refereed Track