**Chapter 28: Locks**

**28.1 Locks: The basic idea**

Locks are used to make sure a critical section is executed properly.

Text

Description automatically generated with low confidence

A lock is just a variable, and to use one, we must declare a **lock variable** of some kind. This lock holds the state the lock at any time, either **available** or **acquired**. We could store other information in the data type as well, such as which thread holds the lock, or a queue for ordering lock acquisition, but information like that is hidden from the user of the lock.

Calling the routine lock() tries to acquire the lock; if no other thread holds the lock (i.e., it is free), the thread will acquire the lock and enter the critical section; this thread is sometimes said to be the **owner** of the lock. If another thread calls lock() on the same lock, it will not return while the lock is held by another thread.

Once the owner calls unlock(), the lock is available again. One of the waiting threads will be noticed, acquire the lock and enter critical section.

By putting a lock around a section of code, the programmer can guarantee that no more than a single thread can ever be active within that code. Thus locks help transform the chaos that is traditional OS scheduling into a more controlled activity.

**28.2 Pthread Locks**

POSIX calls lock a **mutex** (mutual exclusion between threads).

Text

Description automatically generated with medium confidence

In the lock and unlock calls, we pass in a lock as we can use different locks to protect different variables. This can increase **concurrency**: instead of one big lock that is used any time any critical section is accessed (**coarse**-**grained** locking strategy), we often protect different data and data structures with different locks, thus allowing more threads to be in locked code at once (a more **fine-grained** approach).

**28.3 Building a lock**

To build a lock, we need the support from hardware and the OS

**28.4 Evaluating Locks**

The first is whether the lock does its basic task, which is to provide **mutual exclusion**.

The second is **fairness**. Does each thread contending for the lock get a fair shot at acquiring it once it is free? Does any thread contending for the lock starve while doing so, thus never obtaining it?

The final criterion is **performance**, specifically the time overheads added by using the lock.

**28.5 Controlling Interrupts**

The earliest approach was to disable interrupts for critical sections. This is only applicable to single-processor systems.

Text

Description automatically generated

By turning off interrupts before entering a critical section, we ensure that the code inside the critical section will not be interrupted, and thus will execute as if it were atomic. When we are finished, we re-enable interrupts.

The positive part of this approach is **simplicity**.

However, there are many disadvantages. The first is that this approach requires us to allow any calling thread to perform a privileged operation and trust that this facility is not abused (run infinite loop in critical section, so OS never regains control). Secondly, this approach does not work on multiprocessors. Thirdly, turning off interrupts for extended periods of time can lead to interrupts becoming lost, which can lead to serious systems problems. Finally, this approach can be inefficient.

**28.6 A Failed Attempt: Just Using Loads/Stores**

The first attempt is to build a simple lock by using a single flag variable. The idea is quite simple: use a simple variable (flag) to indicate whether some thread has possession of a lock. The first thread that enters the critical section will call lock(), which **tests** whether the flag is equal to 1 (in this case, it is not), and then sets the flag to 1 to indicate that the thread now **holds** the lock. When finished with the critical section, the thread calls unlock() and clears the flag, thus indicating that the lock is no longer held.

**Text, letter

Description automatically generated**

If another thread happens to call lock(), it simply **spin-wait** in the while loop until the lock is unlocked.

However, the code has two problems:

1. **Correctness**: with interrupts, there can be a case where both threads set the flag to 1 and both are able to enter critical section.

Text

Description automatically generated with medium confidence

1. **Performance**: the way we wait is an endless checking of the value of flag. Spin-waiting wastes time waiting for another thread to release a lock.

**28.7 Building Working Spin Locks with Test-And-Set**

The simplest bit of hardware support to understand is known as a **test-and-set** (or **atomic exchange**) instruction.

Text, letter

Description automatically generated

The test-and-set function returns the old value pointed to by the old\_ptr, and simultaneously updates said value to new (test the old value and set new value). The key is that this sequence of operations is performed **atomically**. This enables us to build a simple **spin lock.**

**Text, letter

Description automatically generated**

By making both the test and set a single atomic operation, we ensure that only one thread acquires the lock. **Spin lock** is the simplest type of lock to build, and simply spins, using CPU cycles, until the lock becomes available.

**28.8 Evaluating Spin Locks**

1. Correctness: yes, the spin lock allows a single thread to enter the critical section at a time.
2. Fairness: spin locks don’t provide any fairness guarantees. Indeed, a thread spinning may spin forever, under contention.
3. Performance: performance overheads can be quite painful on single CPU case because each of the threads will spin for the duration of a time slice before giving up the CPU, which is a waster. On multiple CPUs, spin locks work reasonably well because spinning to wait for a lock held on another processor does not waste many cycles.

**28.9 Compare-And-Swap**

Text

Description automatically generated

The basic idea is for compare-and-swap to test whether the value at the address specified by ptr is equal to expected; if so, update the memory location pointed to by ptr with the new value. If not, do nothing. In either case, return the original value at that memory location, allowing the code calling to know whether it succeeded or not.

With this, we can build a lock similar to test-and-set:

Text

Description automatically generated with medium confidence

The rest of the code is the same.

Compare-and-swap is more powerful than test-and-set. We will use this in topics such as **lock-free synchronization**.

**28.10 Load-Linked and Store-Conditional**

The **load-linked** and **store-conditional** instructions can be used in tandem to build locks and other concurrent structures.

Text

Description automatically generated

The load-linked operates much like a typical load instruction, and simply fetches a value from memory and places it in a register. The key difference comes with the store-conditional, which only succeeds if no intervening store to the address has taken place. In the case of success, the store-conditional returns 1 and updates the value at ptr to value; if it fails, the value at ptr is not updated and 0 is returned.

The lock and unlock are defined as followed:

Text, letter

Description automatically generated

**28.11 Fetch-And-Add**

Fetch-and-add instruction atomically increments a value while returning the old value at a particular address.

Text

Description automatically generated

We will use this instruction to build ticket lock.

Text, letter

Description automatically generated

Instead of a single value, this solution uses a ticket and turn variable in combination to build a lock. The basic operation is pretty simple: when a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value; that value is now considered this thread’s “turn” (myturn). The globally shared lock->turn is then used to determine which thread’s turn it is; when (myturn == turn) for a given thread, it is that thread’s turn to enter the critical section.

**28.12 Too Much Spinning: What Now?**

Sometime it is inefficient because we waste an entire time slice doing nothing but checking a value

**28.13 A Simple Approach: Just Yield, Baby**

Our first attempt is to give up the CPU to another thread.

Text, letter

Description automatically generated

In this approach, we assume an operating system primitive yield(). Yield essentially de-schedules itself. Our yield method works quite well. However, the cost of context switch is still substantial.

Worse, we have not tackled the starvation problem at all. A thread may get caught in an endless yield loop while other threads repeatedly enter and exit the critical section.

**28.14 Using Queues: Sleeping Instead Of Spinning**

The spinning and yielding methods have potential waste and no prevention of starvation.

Thus, we must explicitly exert some control over which thread next gets to acquire the lock after the current holder releases it. To do this, we will need a little more OS support, as well as a queue to keep track of which threads are waiting to acquire the lock.

For simplicity, we will use the support provided by Solaris, in terms of two calls: park() to put a calling thread to sleep, and unpark(threadID) to wake a particular thread as designated by threadID.

Text

Description automatically generated

First, we combine the old test-and-set idea with an explicit queue of lock waiters to make a more efficient lock. Second, we use a queue to help control who gets the lock next and thus avoid starvation.

This approach does not avoid spin-waiting entirely. A thread might be interrupted while acquiring or releasing the lock, and thus cause other threads to spin-wait for this one to run again, but the time spent spinning is quite limited.

This method can still cause race condition with wrong timing (wakeup/waiting race). We solve this by modifying the lock:

A picture containing text, watch

Description automatically generated

**28.15 Different OS, Different Support**

Linus provides a **futex** provides more in-kernel functionality.

**28.16 Two-Phase Locks**

A **two-phase lock** realizes that spinning can be useful, particularly if the lock is about to be released. In the first phase, the lock spins for a while, hoping that it can acquire the lock. However, if the lock is not acquired during the first spin phase, a second phase is entered, where the caller is put to sleep, and only woken up when the lock becomes free later.

This is a hybrid approach combining two ideas into one.