51.8.3 Controlling access: lock()

The lock() function takes a variable (or subroutine, but we'll get to that later) and puts a lock on it. No other thread may lock the variable until the locking thread exits the innermost block containing the lock. Using lock() is straightforward:

```
use Thread qw(async);
$a = 4;
$thr1 = async {
    $foo = 12;
    {
        lock ($a); # Block until we get access to $a
        $b = $a;
        $a = $b * $foo;
    }
    print "\$foo was $foo\n";
};
$thr2 = async {
    $bar = 7;
    {
        lock ($a); # Block until we can get access to $a
        $c = $a;
        $a = $c * $bar;
    }
    print "\$bar was $bar\n";
};
$thr1->join;
$thr2->join;
print "\$a is $a\n";
```

lock() blocks the thread until the variable being locked is available. When lock() returns, your thread can be sure that no other thread can lock that variable until the innermost block containing the lock exits.

It's important to note that locks don't prevent access to the variable in question, only lock attempts. This is in keeping with Perl's longstanding tradition of courteous programming, and the advisory file locking that flock() gives you. Locked subroutines behave differently, however. We'll cover that later in the article.

You may lock arrays and hashes as well as scalars. Locking an array, though, will not block subsequent locks on array elements, just lock attempts on the array itself.

Finally, locks are recursive, which means it's okay for a thread to lock a variable more than once. The lock will last until the outermost lock() on the variable goes out of scope.

51.8.4 Thread Pitfall: Deadlocks

Locks are a handy tool to synchronize access to data. Using them properly is the key to safe shared data. Unfortunately, locks aren't without their dangers. Consider the following code:

```
use Thread qw(async yield);
$a = 4;
$b = "foo";
async {
    lock($a);
    yield;
    sleep 20;
    lock ($b);
};
async {
    lock($b);
```