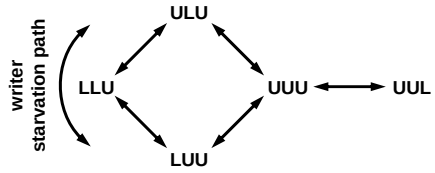
Synchronization Concepts

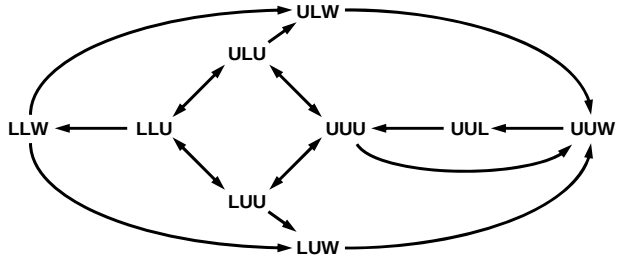
Although we've gone over these ideas already in class, I thought that it might be useful to make a focused case study of a few synchronization issues that arise in the implementation of a relatively simple mechanism. For this purpose, we'll use the reader-writer spin locks defined in the Linux 2.4 kernel, but will build them up step by step to illustrate the scenarios that particular aspects of the design are intended to avoid.

Reader-writer spin locks, or R/W locks, are designed to allow an arbitrary number of reader threads to operate concurrently, but to provide mutual exclusion between a writer thread and threads of any other type. That is, when a write lock is held by some thread, no other threads can hold either read or write locks. When a read lock is held, any other threads can hold read locks as well, but no thread can hold a write lock. R/W locks are useful when read-only operations on a data structure are frequent, as these cannot interfere with one another, whereas the occasional update to the data structure, a write, must be done without concurrent execution of other operations.

Let's start with an idealized case before we think about implementation. We'll consider only two threads that repeatedly request and use read locks, and one thread that repeatedly requests and uses a write lock. Let's call the read threads R and S, and the write thread W. Each of these threads can be in either the unlocked (U) or locked (L) state, and simply moves back and forth between the two states as the program goes forward. Using the tuple <R,S,W> to denote the global state, we want something like the following diagram:



The graph is divided by the UUU state: one can only go from read locks to write locks by passing through a state in which no locks are held. The region to the right of the UUU state represents a thread holding a write lock. Additional writer threads simply add unrelated loops for each thread. The region to the left of the UUU state represents threads holding one or more reader locks. The left region generalizes to a hypercube when more reader threads are added, but already one can see that if reader threads are not somehow restricted when a writer waits, writer starvation can occur. The Linux kernel's R/W spin locks do not address this issue, as they are not meant to be used in a manner in which readers are expected to be continuously active. R/W lock implementations can be defined to favor readers (as shown), to favor writers (no read locks granted while a writer is waiting), or to favor whichever type does not currently hold a lock. As an example, one favors writers by creating a wait state (W) for the write lock and not allowing read locks to be acquired when any thread is in state W. An expansion of our example appears below. Note that an implementation that prioritizes write locks can similarly starve readers in a system with more than one write thread.



Now we're ready to begin discussing the implementation of R/W locks in the Linux kernel. The x86 implementation on which we focus is based on the fetch-and-add (FAA) synchronization primitive available in later versions of the x86 ISA.

Recall that synchronization primitives execute atomically with respect to other instructions (not all instructions, however, necessarily execute atomically with respect to synchronization primitives! For example, the load and store implied by an x86 ADD instruction with a memory operand might be split by an FAA to the same location). Pseudo-code for FAA can be written as follows:

```
int32_t FAA (int32_t* addr, int32_t amt)
{
    int32_t value;
    value = *addr;
    *addr = value + amt;
    return value;
}
```

In order to use FAA for an R/W lock, we assume that the number of threads can be bounded by some large integer. Linux uses 2^24. The lock value starts at 2^24. Readers decrement the lock value by 1 and check for results above 0, while writers decrement the lock value by 2^24 and check for a return value of exactly 2^24. Note that the 32-bit design also implicitly assumes that fewer than 2^8 writers will try the lock simultaneously.

The FAA primitive always changes the value at the specified address, but obviously a lock cannot always be acquired in this manner. In some cases, the return value from FAA must indicate that lock acquisition has failed, in which case the thread must try again. One simple method is for the lock code to add the same amount back into the lock value and to try again until it succeeds. For example:

```
void read_lock (rwlock_t* lock)
{
    while (1) {
        if (0 < FAA (lock, -1)) {return;}
        // thread is in failure state (F)
        FAA (lock, 1);
    }
}
```

Note that, when a write lock is held, the lock value is never above 0, and when a read lock is held, the lock value is always below 2^24.

The write lock code is similar:

```
void write_lock (rwlock_t* lock)
{
    while (1) {
        if (0x1000000 == FAA (lock, -0x1000000))
            {return;}
        // thread is in failure state (F)
        FAA (lock, 0x1000000);
    }
}
```

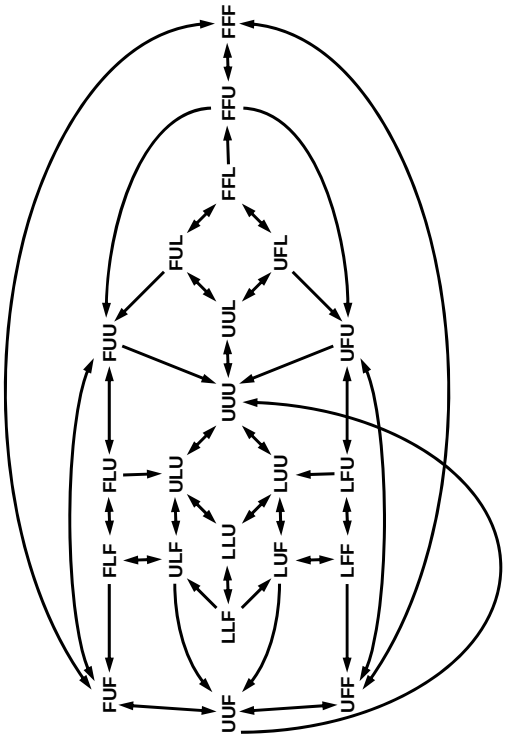
Note that we have added another logical state to each of our reader and writer threads. The failure state (F) occurs when a lock attempt is made but fails. After the second FAA in both operations, the thread returns to the U state and tries again to reach the L state.

The problem with the approach shown—which is not, by the way, the full version of the Linux code—is that it admits livelock, in which a number of threads constantly move between their own private states, but none of them ever actually obtains a lock.

In terms of our original three threads, such an event can happen as follows. First, R obtains a lock, then W tries but fails, then S tries but fails (the lock value has been lowered below 0 by W), and finally R releases its lock. The state is now UFF. Note that neither S nor W can obtain a lock while the other is in state F. A possible cycle is then UFF, UFU, UFF, UUF, and back to UFF.

For our purposes, think of the difference between a deadlock and a livelock as follows: in a deadlock scenario, progress is guaranteed to be impossible; in a livelock scenario, complete lack of progress is possible. Livelocks can be unstable and appear as performance problems, or can be stable and appear to be deadlocks. Livelocks in general also allow threads to perform work that is subsequently discarded, making it difficult to identify them by looking for small loops in state machines.

A full state diagram appears on the next page.



The problem of livelock can be solved here by waiting until a lock attempt has a chance of succeeding before trying it again. In other words, until the lock value reaches a sufficiently high value, there is no point in decrementing it, since any attempt is practically doomed to failure. The two operations now appear as follows (as they do logically in Linux, although the actual code is spread across several files in a mix of preprocessor macros and x86 assembly code):

```
void read_lock (rwlock_t* lock)
{
    while (1) {
        if (0 < FAA (lock, -1)) {return;}
        // thread is in failure (F) state
        FAA (lock, 1);
        while (0 >= *lock);
    }
}


void write_lock (rwlock_t* lock)
{
    while (1) {
        if (0x1000000 == FAA (lock, -0x1000000))
            {return;}
        // thread is in failure (F) state
        FAA (lock, 0x1000000);
        while (0x1000000 > *lock);
    }
}
```

The added loops effectively prevent transitions from U to F whenever a thread was in state F rather than L prior to its current state U. This “memory,” of course, must be represented as another state (say, V), which makes our transition diagram too large and complicated for this kind of note.

If you think back to our earlier example, however (UFF, UFU, UFF, UUF, and back to UFF), we find that instead the second state becomes UFV, and recognize immediately that UFV cannot return to UFF. Instead, we eventually reach UVV, at which point both threads try again, and one is guaranteed to succeed.