27

Locks

From the introduction to concurrency, we saw one of the fundamental problems in concurrent programming: we would like to execute a series of instructions atomically, but due to the presence of interrupts on a single processor (or multiple threads executing on multiple processors concurrently), we couldn't. In this chapter, we thus attack this problem directly, with the introduction of something referred to as a **lock**. Programmers annotate source code with locks, putting them around critical sections, and thus ensure that any such critical section executes as if it were a single atomic instruction.

27.1 Locks: The Basic Idea

As an example, assume our critical section looks like this, the canonical update of a shared variable:

```
balance = balance + 1;
```

Of course, other critical sections are possible, such as adding an element to a linked list or other more complex updates to shared structures, but we'll just keep to this simple example for now. To use a lock, we add some code around the critical section like this:

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

A lock is just a variable, and thus to use one, you must declare a **lock variable** of some kind (such as `mutex` above). This lock variable (or just "lock" for short) holds the state of the lock at any instant in time. It is either **available** (or **unlocked** or **free**) and thus no thread holds the lock, or **acquired** (or **locked** or **held**), and thus exactly one thread holds the lock and presumably is in a critical section. 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.

The semantics of the `lock()` and `unlock()` routines are simple. 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 then calls `lock()` on that same lock variable (`mutex` in this example), it will not return while the lock is held by another thread; in this way, other threads are prevented from entering the critical section while the first thread that holds the lock is in there.

Once the owner of the lock calls `unlock()`, the lock is now available (free) again. If no other threads are waiting for the lock (i.e., no other thread has called `lock()` and is stuck therein), the state of the lock is simply changed to free. If there are waiting threads (stuck in `lock()`), one of them will (eventually) notice (or be informed of) this change of the lock's state, acquire the lock, and enter the critical section.

Locks provide some minimal amount of control over scheduling to programmers. In general, we view threads as entities created by the programmer but scheduled by the OS, in any fashion that the OS chooses. Locks yield some of that control back to the programmer; 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.

27.2 Pthread Locks

The name that the POSIX library uses for a lock is a **mutex**, as it is used to provide **mutual exclusion** between threads, i.e., if one thread is in the critical section, it excludes the others from entering until it has completed the section. Thus, when you see the following POSIX threads code, you should understand that it is doing the same thing as above (note again that we use our own wrappers that check for errors upon lock and unlock):

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock);   // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

You might also notice here that the POSIX version passes a variable to lock and unlock, as we may be using *different* locks to protect different variables. Doing so can increase concurrency: instead of one big lock that is used any time any critical section is accessed (a **coarse-grained** locking strategy), one will 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).

27.3 Building A Lock

By now, you should have some understanding of how a lock works, from the perspective of a programmer. But how should we build a lock? What hardware support is needed? What OS support? It is this set of questions we address in the rest of this chapter.

The Crux: HOW TO BUILD A LOCK

How can we build an efficient lock? Efficient locks provided mutual exclusion at low cost, and also might attain a few other properties we discuss below. What hardware support is needed? What OS support?

To build a working lock, we will need some help from our old friend, the hardware, as well as our good pal, the OS. Over the years, a number of different hardware primitives have been added to the instruction sets of various computer architectures; while we won't study how these instructions are implemented (that, after all, is the topic of a computer architecture class), we will study how to use them in order to build a mutual exclusion primitive like a lock. We will also study how the OS gets involved to complete the picture and enable us to build a sophisticated locking library.

27.4 Evaluating Locks

Before building any locks, we should first understand what our goals are, and thus we ask how to evaluate the efficacy of a particular lock implementation. To evaluate whether a lock works (and works well), we should first establish some basic criteria. The first is whether the lock does its basic task, which is to provide **mutual exclusion**. Basically, does the lock work, preventing multiple threads from entering a critical section?

The second is **fairness**. Does each thread contending for the lock get a fair shot at acquiring it once it is free? Another way to look at this is by examining the more extreme case: 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. There are a few different cases that are worth considering here. One is the case of no contention; when a single thread is running and grabs and releases the lock, what is the overhead of doing so? Another is the case where multiple threads are contending for the lock on a single CPU; in this case, are there performance concerns? Finally, how does the lock perform when there are multiple CPUs involved, and threads on each contending for the lock? By comparing these different scenarios, we can better understand the performance impact of using various locking techniques, as described below.

27.5 Controlling Interrupts

One of the earliest solutions used to provide mutual exclusion was to disable interrupts for critical sections; this solution was invented for single-processor systems. The code would look like this:

```
void lock() {
     DisableInterrupts();
}
void unlock() {
     EnableInterrupts();
}
```

Assume we are running on such a single-processor system. By turning off interrupts (using some kind of special hardware instruction) 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 (again, via a hardware instruction) and thus the program proceeds as usual.

The main positive of this approach is its simplicity. You certainly don't have to scratch your head too hard to figure out why this works. Without interruption, a thread can be sure that the code it executes will execute and that no other thread will interfere with it.

The negatives, unfortunately, are many. First, this approach requires us to allow any calling thread to perform a *privileged* operation (turning interrupts on and off), and thus *trust* that this facility is not abused. As you already know, any time we are required to trust an arbitrary program, we are probably in trouble. Here, the trouble manifests in numerous ways: a greedy program could call `lock()` at the beginning of its execution and thus monopolize the processor; worse, an errant or malicious program could call `lock()` and go into an endless loop. In this latter case, the OS will never regain control of the system, and the only way to address the problem is to restart the system. Thus, using interrupt disabling as a general-purpose synchronization solution requires too much trust in applications.

Second, the approach does not work on multiprocessors. If multiple threads are running on different CPUs, and each try to enter the same critical section, it does not matter whether interrupts are disabled; threads will be able to run on other processors, and thus could enter the critical section. As multiprocessors are now commonplace, our general solution will have to do better than this.

Third, and probably least important, this approach can be inefficient. Compared to normal instruction execution, code that masks or unmasks interrupts tends to be executed slowly by modern CPUs.

For these reasons, turning off interrupts is only used in limited contexts as a mutual-exclusion primitive. For example, in some cases an operating system itself will sometimes use interrupt masking to guarantee atomicity when accessing its own data structures, or at least to prevent certain messy interrupt handling situations from arising. This usage makes sense, as the trust issue disappears inside the OS, which always trusts itself to perform privileged operations anyhow.

ASIDE: DEKKER AND PETERSON'S ALGORITHMS

In the 1960's, Dijkstra posed the concurrency problem to his friends, and one of them, a mathematician named Theodorus Jozef Dekker, came up with a solution [D68]. Unlike the solutions we discuss here, which use special hardware instructions and even OS support, Dekker's approach uses just loads and stores (assuming they are atomic with respect to each other).

Dekker's approach was later refined by Peterson [P81] (and thus "Peterson's algorithm"), shown here. Once again, just loads and stores are used, and the idea is to ensure that two threads never enter a critical section at the same time. Here is Peterson's algorithm (for two threads); see if you can understand it.

```
int flag[2];
int turn;

void init() {
    flag[0] = flag[1] = 0;     // 1->thread wants to grab lock
    turn = 0;                  // whose turn? (thread 0 or 1?)
}
void lock() {
    flag[self] = 1;            // self: thread ID of caller
    turn = 1 - self;           // make it other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait
}
void unlock() {
    flag[self] = 0;            // simply undo your intent
}
```

For some reason, developing locks that work without special hardware support became all the rage for a while, giving theory-types a lot of problems to work on. Of course, this all became quite useless when people realized it is much easier to assume a little hardware support (and indeed that support had been around from the very earliest days of multiprocessing). Further, algorithms like the ones above don't work on modern hardware (due to relaxed memory consistency models), thus making them even less useful than they were before. Yet more research relegated to the dustbin of history...

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Figure 27.1: First Attempt: A Simple Flag

27.6 Test And Set (Atomic Exchange)

Because disabling interrupts does not work on multiple processors, system designers started to invent hardware support for locking. The earliest multiprocessor systems, such as the Burroughs B5000 in the early 1960's [M82], had such support; today all systems provide this type of support, even for single CPU systems.

The simplest bit of hardware support to understand is what is known as a **test-and-set instruction**, also known as **atomic exchange**. To understand how test-and-set works, let's first try to build a simple lock without it. In this failed attempt, we use a simple flag variable to denote whether the lock is held or not.

In this first attempt (Figure 27.1), the idea is quite simple: use a simple variable 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.

If another thread happens to call `lock()` while that first thread is in the critical section, it will simply **spin-wait** in the while loop for that thread to call `unlock()` and clear the flag. Once that first thread does so, the waiting thread will fall out of the while loop, set the flag to 1 for itself, and proceed into the critical section.

Unfortunately, this piece of code has two problems: one of correctness, and another of performance. The correctness problem is simple to see once you get used to thinking about concurrent programming. Imagine the code interleaving as seen below (assume we start in the state `flag=0`).

```
          Thread 0                                       Thread 1

call lock()
while (flag == 1) // flag=0->continue
[INTERRUPT, SWITCH TO THREAD 1]
                                             call lock()
                                             while (flag == 1) // flag=0 ...
                                             flag = 1; // set flag to 1
                                             [INTERRUPT, SWITCH TO THREAD 0]
flag = 1; // set flag to 1 (too!)
```

As you can see from this interleaving, with timely (untimely?) interrupts, we can easily produce a case where *both* threads set their flags to 1 and both threads are thus able to enter the critical section. This is bad! We have obviously failed to provide the most basic requirement: providing mutual exclusion.

The performance problem, which we will address more later on, is the fact that the way a thread waits to acquire a lock that is already held: it endlessly checks the value of flag, a technique known as **spin-waiting**. Spin-waiting wastes time waiting for another thread to release a lock. The waste is exceptionally high on a uniprocessor, where the thread that the waiter is waiting for cannot even run (at least, until a context switch occurs)! Thus, as we move forward and develop more sophisticated solutions, we should also consider ways to avoid this kind of waste.

DESIGN TIP: THINKING ABOUT CONCURRENCY

What we also get from this example is a sense of the approach we need to take when trying to understand concurrent execution. What you are really trying to do is to pretend you are a **malicious scheduler**, one that interrupts threads at the most inopportune of times in order to foil their feeble attempts at building synchronization primitives. Although the exact sequence of interrupts may be *improbable*, it is *possible*, and that is all we need to show to demonstrate that a particular approach does not work.

27.7 Building A Working Spin Lock

While the idea behind the example above is a good one, it is not possible to implement without some support from the hardware. Fortunately, some systems provide an instruction to support the creation of simple locks based on this concept. This more powerful instruction has different names – on SPARC, it is the load/store unsigned byte instruction (`ldstub`), whereas on x86, it is the atomic exchange instruction (`xchg`) – but basically does the same thing across platforms, and is usually generally referred to as **test-and-set**. We define what the test-and-set instruction does with the following C code snippet:

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr; // fetch old value at ptr
    *ptr = new;     // store 'new' into ptr
    return old;     // return the old value
}
```

What the test-and-set instruction does is as follows. It returns the old value pointed to by the `ptr`, and simultaneously updates said value to `new`. The key, of course, is that this sequence of operations is performed **atomically**[1]. The reason it is called "test and set" is that it enables you to "test" the old value (which is what is returned) while simultaneously "setting" the memory location to a new value; as it turns out, this slightly more powerful instruction is enough to build a simple **spin lock**, as we now examine in Figure 27.2.

Let's make sure we understand why this works. Imagine first the case where a thread calls `lock()` and no other thread currently holds the lock; thus, `flag` should be 0. When the thread then calls `TestAndSet(flag, 1)`, the routine will return the old value of `flag`, which is 0; thus, the calling thread, which is *testing* the value of flag, will not get caught spinning in the while loop and will acquire the lock. The thread will also atomically *set* the value to 1, thus indicating that the lock is now held. When the thread is finished with its critical section, it calls `unlock()` to set the flag back to zero.

The second case we can imagine arises when one thread already has the lock held (i.e., `flag` is 1). In this case, this thread will call `lock()` and then call `TestAndSet(flag, 1)` as well. This time, however, `TestAndSet()` will return the old value at flag, which is 1 (because the lock is held), while simultaneously setting it to 1 again.

---

[1] How does the hardware do this? Take a hardware class and find out!

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 27.2: A Simple Spin Lock using Test-and-set

As long as the lock is held by another thread, `TestAndSet()` will repeatedly return 1, and thus this thread will spin and spin until the lock is finally released. When the flag is finally set to 0 by some other thread, this thread will call `TestAndSet()` again, which will now return 0 while atomically setting the value to 1 and thus acquire the lock and enter the critical section.

By making both the **test** (of the old value of the lock) and **set** (of the new value) a single atomic operation, we can ensure that only one thread acquires the lock. Thus we can build a successful mutual exclusion primitive!

You may also now understand why this type of lock is usually referred to as a **spin lock**. It is the simplest type of lock to build, and simply spins, using CPU cycles, until the lock becomes available. To work correctly on a single processor, it requires a **preemptive scheduler** (i.e., one that will interrupt a thread via a timer, in order to run a different thread, from time to time). Without preemption, spin locks don't make much sense on a single CPU, as a thread spinning on a CPU will never relinquish it.

27.8 Evaluating Spin Locks

Given our basic spin lock, we can now evaluate how effective it is along our previously described axes. The most important aspect of a lock is correctness: does it provide mutual exclusion? The answer here is obviously yes: the spin lock only allows a single thread to enter the critical section at a time. Thus, we have a correct lock.

The next axis is fairness. How fair is a spin lock to a waiting thread? Can you guarantee that a waiting thread will ever enter the critical section? The answer here, unfortunately, is bad news: spin locks don't provide any fairness guarantees. Indeed, a thread spinning may spin forever, under contention. Thus, spin locks are not fair and may indeed lead to starvation.

The final axis is performance. What are the costs of using a spin lock? To analyze this more carefully, we suggest thinking about a few different cases. In the first, imagine threads competing for the lock on a single processor; in the second, consider the threads as spread out across many processors.

For spin locks, in the single CPU case, performance overheads can be quite painful; imagine the case where the thread holding the lock is pre-empted within a critical section. The scheduler might then run every other thread (imagine there are $N - 1$ others), each of which tries to acquire the lock. In this case, each of those threads will spin for the duration of a time slice before giving up the CPU, which is quite a waste of CPU cycles.

However, on multiple CPUs, spin locks work reasonably well (if the number of threads roughly equals the number of CPUs). The thinking goes as follows: imagine Thread A on CPU 1 and Thread B on CPU 2, both contending for a lock. If Thread A (CPU 1) grabs the lock, and then Thread B tries to, B will spin (on CPU 2). However, presumably the critical section is short, and thus soon the lock becomes available, and is acquired by Thread B. Spinning to wait for a lock held on another processor doesn't waste many cycles in this case, and thus can be quite effective.

27.9 Compare-And-Swap

Another hardware primitive that some systems provide is known as the **compare-and-swap** instruction (as it is called on SPARC, for example), or **compare-and-exchange** (as it called on x86). The C pseudocode for this single instruction is found in Figure 27.3.

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 actual value at that memory

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Figure 27.3: Compare-and-swap

location, thus allowing the code calling compare-and-swap to know whether it succeeded or not.

With the compare-and-swap instruction, we can build a lock in a manner quite similar to that with test-and-set. For example, we could just replace the `lock()` routine above with the following:

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

The rest of the code is the same as the test-and-set example above. This code works quite similarly; it simply checks if the flag is 0 and if so, atomically swaps in a 1 thus acquiring the lock. Threads that try to acquire the lock while it is held will get stuck spinning until the lock is finally released.

If you want to see how to really make a C-callable x86-version of compare-and-swap, this code sequence might be useful (stolen from here [S05]):

```
char CompareAndSwap(int *ptr, int old, int new) {
  unsigned char ret;

  // Note that sete sets a 'byte' not the word
  __asm__ __volatile__ (
        "  lock\n"
        "  cmpxchgl %2,%1\n"
        "  sete %0\n"
        : "=q" (ret), "=m" (*ptr)
        : "r" (new), "m" (*ptr), "a" (old)
        : "memory");
  return ret;
}
```

Finally, as you may have sensed, compare-and-swap is a more powerful instruction than test-and-set. We will make some use of

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

Figure 27.4: Load-linked and Store-conditional

this power in the future when we briefly delve into **wait-free synchronization** [H91]. However, if we just build a simple spin lock with it, its behavior is identical to the spin lock we analyzed above.

27.10 Load-Linked and Store-Conditional

Some platforms provide a pair of instructions that work in concert to help build critical sections. On the MIPS architecture [H93], for example, the **load-linked** and **store-conditional** instructions can be used in tandem to build locks and other concurrent structures. The C pseudocode for these instructions is as found in Figure 27.4. Alpha, PowerPC, and ARM provide similar instructions [W09].

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 (and updates the value stored at the address just load-linked from) if no intermittent 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.

As a challenge to yourself, try thinking about how to build a lock using load-linked and store-conditional. Then, when you are finished, look at the code below which provides one simple solution. Do it! The solution is in Figure 27.5.

The `lock()` code is the only interesting piece. First, a thread spins waiting for the flag to be set to 0 (and thus indicate the lock is not held). Once so, the thread tries to acquire the lock via the store-conditional; if it succeeds, the thread has atomically changed the flag's value to 1 and thus can proceed into the critical section.

Note how failure of the store-conditional might arise. One thread

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 27.5: Using LL/SC To Build A Lock

calls `lock()` and executes the load-linked, returning 0 as the lock is not held. Before it can attempt the store-conditional, it is interrupted and another thread enters the lock code, also executing the load-linked instruction, and also getting a 0 and continuing. At this point, two threads have each executed the load-linked and each are about to attempt the store-conditional. The key feature of these instructions is that only one of these threads will succeed in updating the flag to 1 and thus acquire the lock; the second thread to attempt the store-conditional will fail (because the other thread updated the value of flag between its load-linked and store-conditional) and thus have to try to acquire the lock again.

In class a few years ago, undergraduate student David Capel suggested a more concise form of the above, for those of you who enjoy short-circuiting boolean conditionals. See if you can figure out why it is equivalent. It certainly is shorter!

```
void lock(lock_t *lock) {
  while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
    ; // spin
}
```

27.11 Fetch-And-Add

One final hardware primitive is the **fetch-and-add** instruction, which atomically increments a value while returning the old value at a particular address. The C pseudocode for the fetch-and-add instruction looks like this:

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

In this example, we'll use fetch-and-add to build a more interesting **ticket lock**, as introduced by Mellor-Crummey and Scott [MS91]. The lock and unlock code looks like what you see in Figure 27.6.

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. Unlock is accomplished simply by incrementing the turn such that the next waiting thread (if there is one) can now enter the critical section.

Note one important difference with this solution versus our previous attempts: it ensures progress for all threads. Once a thread is assigned its ticket value, it will be scheduled at some point in the future (once those in front of it have passed through the critical section and released the lock). In our previous attempts, no such guarantee existed; a thread spinning on test-and-set (for example) could spin forever even as other threads acquire and release the lock.

CODING TIP: LESS CODE IS BETTER CODE

Programmers tend to brag about how much code they wrote to do something. Doing so is fundamentally broken. What one should brag about, rather, is how *little* code one wrote to accomplish a given task. Short, concise code is always preferred; it is likely easier to understand and has fewer bugs. As Hugh Lauer said, when discussing the construction of the Pilot operating system: "If the same people had twice as much time, they could produce as good of a system in half the code." [L81] We'll call this **Lauer's Law**, and it is well worth remembering. So next time you're bragging about how much code you wrote to finish the assignment, think again, or better yet, go back, rewrite, and make the code as clear and concise as possible.

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

Figure 27.6: Ticket Locks

27.12 Summary: So Much Spinning

Our simple hardware-based locks are simple (only a few lines of code) and they work (you could even prove that if you'd like to, by writing some code), which are two excellent properties of any system or code. However, in some cases, these solutions can be quite inefficient. Imagine you are running two threads on a single processor. Now imagine that one thread (thread 0) is in a critical section and thus has a lock held, and unfortunately gets interrupted. The second thread (thread 1) now tries to acquire the lock, but finds that it is held. Thus, it begins to spin. And spin. Then it spins some more. And finally, a timer interrupt goes off, thread 0 is run again, which releases the lock, and finally (the next time it runs, say), thread 1 won't have to spin so much and will be able to acquire the lock. Thus, any time a thread gets caught spinning in a situation like this, it wastes an entire time slice doing nothing but checking a value that isn't going to change! The problem gets worse with N threads contending for a lock; $N - 1$ time slices may be wasted in a similar manner, simply spinning and waiting for a single thread to release the lock. And thus, our next problem:

THE CRUX: HOW TO AVOID SPINNING

How can we develop a lock that doesn't needlessly waste time spinning on the CPU?

Hardware support alone cannot solve the problem. We'll need OS support too! Let's now figure out just how that might work.

27.13 A Simple Approach: Just Yield, Baby

Hardware support got us pretty far: working locks, and even (as with the case of the ticket lock) fairness in lock acquisition. However, we still have a problem: what to do when a context switch occurs in a critical section, and threads start to spin endlessly, waiting for the interrupt (lock-holding) thread to be run again?

Our first try is a simple and friendly approach: when you are going to spin, instead give up the CPU to another thread. Or, as Al Davis might say, "just yield, baby!" [D91]. Figure 27.7 presents the approach.

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Figure 27.7: Lock with test-and-set and yield

In this approach, we assume an operating system primitive `yield()` which a thread can call when it wants to give up the CPU and let another thread run. Because a thread can be in one of three states (running, ready, or blocked), you can think of this as an OS system call that moves the caller from the **running** state to the **ready** state, and thus promotes another thread to running.

Think about the example with two threads on one CPU; in this case, our yield-based approach works quite well. If a thread happens to call `lock()` and find a lock held, it will simply yield the CPU, and thus the other thread will run and finish its critical section. In this simple case, the yielding approach works well.

Let us now consider the case where there are many threads (say 100) contending for a lock repeatedly. In this case, if one thread acquires the lock and is preempted before releasing it, the other 99 will each call `lock()`, find the lock held, and yield the CPU. Assuming some kind of round-robin scheduler, each of the 99 will execute this run-and-yield pattern before the thread holding the lock gets to run again. While better than our spinning approach (which would waste 99 time slices spinning), this approach is still costly; the cost of a context switch can be substantial, and there is thus plenty of waste.

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. We clearly will need an approach that addresses this problem directly.

27.14 Using Queues: Sleeping Instead Of Spinning

The real problem with our previous approaches is that they leave too much to chance. The scheduler determines which thread runs next; if the scheduler makes a bad choice, a thread runs that must either spin waiting for the lock (our first approach), or yield the CPU immediately (our second approach). Either way, there is potential for waste and no prevention of starvation.

Thus, we must explicitly exert some control over who gets to acquire the lock next 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 enter 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`. These two routines can be used in tandem to build a lock that puts a caller to sleep if it tries to acquire a held lock and wakes it when the lock is free. Let's look at the code in Figure 27.8 to understand one possible use of such primitives.

We do a couple of interesting things in this example. First, we

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Figure 27.8: Lock with Queues, test-and-set, yield, and wakeup

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.

You might notice how the guard is used, basically as a spin-lock around the flag and queue manipulations the lock is using. This approach thus doesn't 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. However, the time spent spinning is quite limited (just a few instructions inside the lock and unlock code, instead of the user-defined critical section), and thus this approach may be a reasonable one.

Second, you might notice that in `lock()`, when a thread can not acquire the lock (it is already held), we are careful to add ourselves to a queue (by calling the `gettid()` call to get the thread ID of the current thread), set guard to 0, and yield the CPU. A question for the reader: What would happen if the release of the guard lock came *after* the `park()`, and not before? Hint: something bad.

You might also notice the interesting fact that the flag does not get set back to 0 when when another thread gets woken up. Why is this? Well, it is not an error, but rather a necessity! When a thread is woken up, it will be as if it is returning from `park()`; however, it does not hold the guard at that point in the code and thus cannot even try to set the flag to 1. Thus, we just pass the lock directly from the thread releasing the lock to the next thread acquiring it; flag is not set to 0 in-between.

Finally, you might notice the perceived race condition in the solution, just before the call to `park()`. With just the wrong timing, a thread will be about to park, assuming that it should sleep until the lock is no longer held. A switch at that time to another thread (say, a thread holding the lock) could lead to trouble, for example, if that thread then released the lock. The subsequent park by the first thread would then sleep forever (potentially). This problem is sometimes called the **wakeup/waiting race**; to avoid it, we need to do some extra work.

Solaris solves this problem by adding a third system call: `setpark()`. By calling this routine, a thread can indicate it is *about to* park. If it then happens to be interrupted and another thread calls unpark before park is actually called, the subsequent park returns immediately instead of sleeping. The code modification, inside of `lock()`, is quite small:

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
```

A different solution could pass the guard into the kernel. In that case, the kernel could take precautions to atomically release the lock and dequeue the running thread.

```
void mutex_lock (int *mutex) {
  unsigned int v;
  /* Bit 31 was clear, we got the mutex (this is the fastpath)  */
  if (atomic_bit_test_set (mutex, 31) == 0)
    return;
  atomic_increment (mutex);
  while (1) {
      if (atomic_bit_test_set (mutex, 31) == 0) {
          atomic_decrement (mutex);
          return;
      }
      /* We have to wait now. First make sure the futex value
         we are monitoring is truly negative (i.e. locked). */
      v = *mutex;
      if (v >= 0)
        continue;
      futex_wait (mutex, v);
  }
}

void mutex_unlock (int *mutex) {
  /* Adding 0x80000000 to the counter results in 0 if and only if
    there are not other interested threads */
  if (atomic_add_zero (mutex, 0x80000000))
    return;

  /* There are other threads waiting for this mutex,
     wake one of them up.  */
  futex_wake (mutex);
```

Figure 27.9: Linux-based Futex Locks

27.15 Different OS, Different Support

We have thus far seen one type of support that an OS can provide in order to build a more efficient lock in a thread library. Other OS's provide similar support; the details vary.

For example, Linux provides something called a **futex** which is similar to the Solaris interface but provides a bit more in-kernel functionality. Specifically, each futex has associated with it a specific physical memory location; associated with each such memory location is an in-kernel queue. Callers can use futex calls (described below) to sleep and wake as need be.

Specifically, two calls are available. The call to `futex_wait(address, expected)` puts the calling thread to sleep, assuming the value at `address` is equal to `expected`. If it is *not* equal, the call returns immediately. The call to the routine `futex_wake(address)` wakes one thread that is waiting on the queue. The usage of these in Linux is as found in 27.9.

This code snippet from `lowlevellock.h` in the nptl library (part of the gnu libc library) [L09] is pretty interesting. Basically, it uses a single integer to track both whether the lock is held or not (the high bit of the integer) and the number of waiters on the lock (all the other bits). Thus, if the lock is negative, it is held (because the high bit is set and that bit determines the sign of the integer). The code is also interesting because it shows how to optimize for the common case where there is no contention: with only one thread acquiring and releasing a lock, very little work is done (the atomic bit test-and-set to lock and an atomic add to release the lock). See if you can puzzle through the rest of this "real-world" lock to see how it works.

27.16 Two-Phase Locks

One final note: the Linux approach has the flavor of an old approach that has been used on and off for years, going at least as far back to Dahm Locks in the early 1960's [M82], and is now referred to as a **two-phase lock**. A two-phase lock realizes that spinning can be useful, particularly if the lock is about to be released. So 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. The Linux lock above is a form of such a lock, but it only spins once; a generalization of this could spin in a loop for a fixed amount of time before using the **futex** support in the kernel to sleep.

Two-phase locks are yet another instance of a **hybrid** approach, where combining two good ideas may indeed yield a better one. Of course, whether it does depends strongly on many things, including the hardware environment, number of threads, and other workload details. As always, making a single general-purpose lock, good for all possible use cases, is quite a challenge.

27.17 Summary

The above approach shows how real locks are built these days: some hardware support (in the form of a more powerful instruction) plus some operating system support (e.g., in the form of `park()` and `unpark()` primitives on Solaris, or **futex** on Linux). Of course, the details differ, and the exact code to perform such locking is usually highly tuned. Check out the Solaris or Linux open source code bases if you want to see more details; they are a fascinating read [L09, S09].

References

[D91] "Just Win, Baby: Al Davis and His Raiders"
Glenn Dickey, Harcourt 1991
There is even an undoubtedly bad book about Al Davis and his famous "just win" quote. Or, I guess, the book is more about Al Davis and the Raiders, and maybe not just the quote. Read the book to find out?

[D68] "Cooperating sequential processes"
Edsger W. Dijkstra, 1968
Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF
One of the early seminal papers in the area. Discusses how Dijkstra posed the original concurrency problem, and Dekker's solution.

[H93] "MIPS R4000 Microprocessor User's Manual".
Joe Heinrich, Prentice-Hall, June 1993
Available: http://cag.csail.mit.edu/raw/
documents/R4400_Uman_book_Ed2.pdf

[H91] "Wait-free Synchronization"
Maurice Herlihy
ACM Transactions on Programming Languages and Systems (TOPLAS)
Volume 13, Issue 1, January 1991
A landmark paper introducing a different approach to building concurrent data structures. However, because of the complexity involved, many of these ideas have been slow to gain acceptance in deployed systems.

[L81] "Observations on the Development of an Operating System"
Hugh Lauer
SOSP '81
A must-read retrospective about the development of the Pilot OS, an early PC operating system. Fun and full of insights.

[L09] "glibc 2.9 (include Linux pthreads implementation)"
Available: http://ftp.gnu.org/gnu/glibc/
In particular, take a look at the nptl subdirectory where you will find most of the pthread support in Linux today.

[M82] "The Architecture of the Burroughs B5000
20 Years Later and Still Ahead of the Times?"
Alastair J.W. Mayer, 1982
www.ajwm.net/amayer/papers/B5000.html
From the paper: "One particularly useful instruction is the RDLK (read-lock). It is an indivisible operation which reads from and writes into a memory location." RDLK is thus an early test-and-set primitive, if not the earliest. Some credit here goes to an engineer named Dave Dahm, who apparently invented a number of these things for the Burroughs systems, including a form of spin locks (called "Buzz Locks" as well as a two-phase lock eponymously called "Dahm Locks.")

[MS91] "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors"
John M. Mellor-Crummey and M. L. Scott
ACM TOCS, February 1991
An excellent survey on different locking algorithms. However, no OS support is used, just fancy hardware instructions.

[P81] "Myths About the Mutual Exclusion Problem"
G.L. Peterson
Information Processing Letters. 12(3) 1981, 115–116
Peterson's algorithm introduced here.

[S05] "Guide to porting from Solaris to Linux on x86"
Ajay Sood, April 29, 2005
Available: `http://www.ibm.com/developerworks/linux/library/l-solar/`

[S09] "OpenSolaris Thread Library"
Available: http://src.opensolaris.org/source/xref/onnv/onnv-gate/
usr/src/lib/libc/port/threads/synch.c
This is also pretty interesting to look at, though who knows what will happen to it now that Oracle owns Sun. Thanks to Mike Swift for the pointer to the code.

[W09] "Load-Link, Store-Conditional"
Wikipedia entry on said topic, as of October 22, 2009
`http://en.wikipedia.org/wiki/Load-Link/Store-Conditional`
Can you believe I referenced wikipedia? Pretty shabby. But, I found the information there first, and it felt funny not to cite it. Further, they even listed the instructions for the different architectures: `ldl_l/stl_c` and `ldq_l/stq_c` (Alpha), `lwarx/stwcx` (PowerPC), `ll/sc` (MIPS), and `ldrex/strex` (ARM version 6 and above).

[WG00] "The SPARC Architecture Manual: Version 9"
David L. Weaver and Tom Germond, September 2000
SPARC International, San Jose, California
Available: `http://www.sparc.org/standards/SPARCV9.pdf`
Also see: `http://developers.sun.com/solaris/articles/atomic_sparc/` for some more details on Sparc atomic operations.