28

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.

28.1 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:

```
1  lock_t mutex; // some globally-allocated lock 'mutex'
2  ...
3  lock(&mutex);
4  balance = balance + 1;
5  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

1

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.

28.2 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 (we again use our wrappers that check for errors upon lock and unlock):

```
1  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Pthread_mutex_lock(&lock); // wrapper; exits on failure
4  balance = balance + 1;
5  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).

28.3 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 provide 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.

28.4 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 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.

28.5 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:

```
1  void lock() {
2      DisableInterrupts();
3  }
4  void unlock() {
5      EnableInterrupts();
6  }
```

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 never regains control of the system, and there is only one recourse: restart the system. 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, turning off interrupts for extended periods of time can lead to interrupts becoming lost, which can lead to serious systems problems. Imagine, for example, if the CPU missed the fact that a disk device has finished a read request. How will the OS know to wake the process waiting for said read?

```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 -> lock is available, 1 -> held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)   // TEST the flag
10          ; // spin-wait (do nothing)
11      mutex->flag = 1;           // now SET it!
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

Figure 28.1: **First Attempt: A Simple Flag**

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 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.

28.6 A Failed Attempt: Just Using Loads/Stores

To move beyond interrupt-based techniques, we will have to rely on CPU hardware and the instructions it provides us to build a proper lock. Let's first try to build a simple lock by using a single flag variable. In this failed attempt, we'll see some of the basic ideas needed to build a lock, and (hopefully) see why just using a single variable and accessing it via normal loads and stores is insufficient.

In this first attempt (Figure 28.1), 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.

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, the code has two problems: one of correctness, and an-

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Figure 28.2: **Trace: No Mutual Exclusion**

other of performance. The correctness problem is simple to see once you get used to thinking about concurrent programming. Imagine the code interleaving in Figure 28.2; assume `flag=0` to begin.

As you can see from this interleaving, with timely (untimely?) interrupts, we can easily produce a case where *both* threads set the flag to 1 and both threads are thus able to enter the critical section. This behavior is what professionals call "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.

28.7 Building Working Spin Locks with Test-And-Set

Because disabling interrupts does not work on multiple processors, and because simple approaches using loads and stores (as shown above) don't work, 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 known as a **test-and-set** (or **atomic exchange**[1]) instruction. We define what the test-and-set instruction does via the following C code snippet:

```
1   int TestAndSet(int *old_ptr, int new) {
2       int old = *old_ptr; // fetch old value at old_ptr
3       *old_ptr = new;     // store 'new' into old_ptr
4       return old;         // return the old value
5   }
```

---
[1]Each architecture that supports test-and-set calls it by a different name. On SPARC it is called the load/store unsigned byte instruction (`ldstub`); on x86 it is the locked version of the atomic exchange (`xchg`).

ASIDE: DEKKER'S 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 algorithm** uses just loads and stores (assuming they are atomic with respect to each other, which was true on early hardware).

Dekker's approach was later refined by Peterson [P81]. 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 the code. What are the `flag` and `turn` variables used for?

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

void init() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}
void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while it's not your turn
}
void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

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 line of work 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 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...

```
1   typedef struct __lock_t {
2       int flag;
3   } lock_t;
4
5   void init(lock_t *lock) {
6       // 0: lock is available, 1: lock is held
7       lock->flag = 0;
8   }
9
10  void lock(lock_t *lock) {
11      while (TestAndSet(&lock->flag, 1) == 1)
12          ; // spin-wait (do nothing)
13  }
14
15  void unlock(lock_t *lock) {
16      lock->flag = 0;
17  }
```

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

What the test-and-set instruction does is as follows. It returns the old value pointed to by the old_ptr, and simultaneously updates said value to new. The key, of course, is that this sequence of operations is performed **atomically**. 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 28.3. Or better yet: figure it out first yourself!

Let's make sure we understand why this lock 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 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, TestAndSet() will return the old value at flag, which is 1 (because the lock is held), while simultaneously setting it to 1 again. 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 lock value) and **set** (of the new

TIP: THINK ABOUT CONCURRENCY AS A MALICIOUS SCHEDULER
From this example, you might get a sense of the approach you need to take to understand concurrent execution. What you should try 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. What a mean scheduler you are! Although the exact sequence of interrupts may be *improbable*, it is *possible*, and that is all we need to demonstrate that a particular approach does not work. It can be useful to think maliciously! (at least, sometimes)

value) a single atomic operation, we ensure that only one thread acquires the lock. And that's how to build a working 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.

28.8 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 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. Simple spin locks (as discussed thus far) are not fair and may 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 threads spread out across many CPUs.

For spin locks, in the single CPU case, performance overheads can be quite painful; imagine the case where the thread holding the lock is preempted 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, 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

```
1   int CompareAndSwap(int *ptr, int expected, int new) {
2       int original = *ptr;
3       if (original == expected)
4           *ptr = new;
5       return original;
6   }
```

Figure 28.4: **Compare-and-swap**

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 effective.

28.9 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 28.4.

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, 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:

```
1   void lock(lock_t *lock) {
2       while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3           ; // spin
4   }
```

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, the code sequence (from [S05]) might be useful[2].

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

---

[2]github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-locks

the future when we briefly delve into topics such as **lock-free synchro-nization** [H91]. However, if we just build a simple spin lock with it, its behavior is identical to the spin lock we analyzed above.

28.10 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 28.5. Alpha, PowerPC, and ARM provide similar instructions [W09].

The load-linked operates much like a typical load instruction, and sim-ply fetches a value from memory and places it in a register. The key differ-ence comes with the store-conditional, which only succeeds (and updates the value stored at the address just load-linked from) 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.

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 28.6.

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 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,

```
1   int LoadLinked(int *ptr) {
2       return *ptr;
3   }
4
5   int StoreConditional(int *ptr, int value) {
6       if (no update to *ptr since LL to this addr) {
7           *ptr = value;
8           return 1; // success!
9       } else {
10          return 0; // failed to update
11      }
12  }
```

Figure 28.5: **Load-linked And Store-conditional**

```
1   void lock(lock_t *lock) {
2       while (1) {
3           while (LoadLinked(&lock->flag) == 1)
4               ; // spin until it's zero
5           if (StoreConditional(&lock->flag, 1) == 1)
6               return; // if set-to-1 was success: done
7                       // otherwise: try again
8       }
9   }
10
11  void unlock(lock_t *lock) {
12      lock->flag = 0;
13  }
```

Figure 28.6: **Using LL/SC To Build A Lock**

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 sug-
gested 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!

```
1   void lock(lock_t *lock) {
2       while (LoadLinked(&lock->flag) ||
3           !StoreConditional(&lock->flag, 1))
4           ; // spin
5   }
```

28.11 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 partic-
ular address. The C pseudocode for the fetch-and-add instruction looks
like this:

```
1   int FetchAndAdd(int *ptr) {
2       int old = *ptr;
3       *ptr = old + 1;
4       return old;
5   }
```

TIP: LESS CODE IS BETTER CODE (LAUER'S LAW)
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.

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 is found in Figure 28.7 (page 14).

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.

28.12 Too Much Spinning: What Now?

Our 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

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn   = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15  }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19  }
```

Figure 28.7: **Ticket Locks**

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.

28.13 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 interrupted (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. As Al Davis might say, "just yield, baby!" [D91]. Figure 28.8 (page 15) shows the approach.

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```
Figure 28.8: **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. A thread can be in one of three states (running, ready, or blocked); yield is simply a system call that moves the caller from the **running** state to the **ready** state, and thus promotes another thread to running. Thus, the yielding thread essentially **deschedules** itself.

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, this approach does not address starvation. 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 starvation directly.

28.14 Using Queues: Sleeping Instead Of Spinning

The real problem with some previous approaches (other than the ticket lock) is that they leave too much to chance. The scheduler determines which thread runs next; if the scheduler makes a bad choice, a thread that runs 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.

```
1   typedef struct __lock_t {
2       int flag;
3       int guard;
4       queue_t *q;
5   } lock_t;
6
7   void lock_init(lock_t *m) {
8       m->flag  = 0;
9       m->guard = 0;
10      queue_init(m->q);
11  }
12
13  void lock(lock_t *m) {
14      while (TestAndSet(&m->guard, 1) == 1)
15          ; //acquire guard lock by spinning
16      if (m->flag == 0) {
17          m->flag = 1; // lock is acquired
18          m->guard = 0;
19      } else {
20          queue_add(m->q, gettid());
21          m->guard = 0;
22          park();
23      }
24  }
25
26  void unlock(lock_t *m) {
27      while (TestAndSet(&m->guard, 1) == 1)
28          ; //acquire guard lock by spinning
29      if (queue_empty(m->q))
30          m->flag = 0; // let go of lock; no one wants it
31      else
32          unpark(queue_remove(m->q)); // hold lock
33                                      // (for next thread!)
34      m->guard = 0;
35  }
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

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. 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 28.9 to understand one possible use of such primitives.

ASIDE: MORE REASON TO AVOID SPINNING: PRIORITY INVERSION

One good reason to avoid spin locks is performance: as described in the main text, if a thread is interrupted while holding a lock, other threads that use spin locks will spend a large amount of CPU time just waiting for the lock to become available. However, it turns out there is another interesting reason to avoid spin locks on some systems: correctness. The problem to be wary of is known as **priority inversion**, which unfortunately is an intergalactic scourge, occurring on Earth [M15] and Mars [R97]!

Let's assume there are two threads in a system. Thread 2 (T2) has a high scheduling priority, and Thread 1 (T1) has lower priority. In this example, let's assume that the CPU scheduler will always run T2 over T1, if indeed both are runnable; T1 only runs when T2 is not able to do so (e.g., when T2 is blocked on I/O).

Now, the problem. Assume T2 is blocked for some reason. So T1 runs, grabs a spin lock, and enters a critical section. T2 now becomes unblocked (perhaps because an I/O completed), and the CPU scheduler immediately schedules it (thus descheduling T1). T2 now tries to acquire the lock, and because it can't (T1 holds the lock), it just keeps spinning. Because the lock is a spin lock, T2 spins forever, and the system is hung.

Just avoiding the use of spin locks, unfortunately, does not avoid the problem of inversion (alas). Imagine three threads, T1, T2, and T3, with T3 at the highest priority, and T1 the lowest. Imagine now that T1 grabs a lock. T3 then starts, and because it is higher priority than T1, runs immediately (preempting T1). T3 tries to acquire the lock that T1 holds, but gets stuck waiting, because T1 still holds it. If T2 starts to run, it will have higher priority than T1, and thus it will run. T3, which is higher priority than T2, is stuck waiting for T1, which may never run now that T2 is running. Isn't it sad that the mighty T3 can't run, while lowly T2 controls the CPU? Having high priority just ain't what it used to be.

You can address the priority inversion problem in a number of ways. In the specific case where spin locks cause the problem, you can avoid using spin locks (described more below). More generally, a higher-priority thread waiting for a lower-priority thread can temporarily boost the lower thread's priority, thus enabling it to run and overcoming the inversion, a technique known as **priority inheritance**. A last solution is simplest: ensure all threads have the same priority.

We do a couple of interesting things in this example. 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.

You might notice how the guard is used (Figure 28.9, page 16), 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 reasonable.

You might also observe 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()` function 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 further detect that the flag does not get set back to 0 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 re-
turning 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), a problem sometimes called the **wakeup/waiting race**.

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 sleep-
ing. The code modification, inside of `lock()`, is quite small:

```
1    queue_add(m->q, gettid());
2    setpark(); // new code
3    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 de-
queue the running thread.

28.15 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 a **futex** which is similar to the Solaris in-
terface but provides more in-kernel functionality. Specifically, each futex
has associated with it a specific physical memory location, as well as a

```
1   void mutex_lock (int *mutex) {
2     int v;
3     // Bit 31 was clear, we got the mutex (fastpath)
4     if (atomic_bit_test_set (mutex, 31) == 0)
5       return;
6     atomic_increment (mutex);
7     while (1) {
8         if (atomic_bit_test_set (mutex, 31) == 0) {
9             atomic_decrement (mutex);
10            return;
11        }
12        // Have to waitFirst to make sure futex value
13        // we are monitoring is negative (locked).
14        v = *mutex;
15        if (v >= 0)
16          continue;
17        futex_wait (mutex, v);
18     }
19  }
20
21  void mutex_unlock (int *mutex) {
22    // Adding 0x80000000 to counter results in 0 if and
23    // only if there are not other interested threads
24    if (atomic_add_zero (mutex, 0x80000000))
25      return;
26
27    // There are other threads waiting for this mutex,
28    // wake one of them up.
29    futex_wake (mutex);
30  }
```

Figure 28.10: **Linux-based Futex Locks**

per-futex 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 the address 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 calls in a Linux mutex is shown in Figure 28.10 (page 19).

This code snippet from lowlevellock.h in the nptl library (part of the gnu libc library) [L09] is interesting for a few reasons. First, 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).

Second, the code snippet shows how to optimize for the common case,

specifically when there is no contention for the lock; 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 understand how it works. Do it and become a master of Linux locking, or at least somebody who listens when a book tells you to do something [3].

28.16 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 **futex** support 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.

28.17 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 code bases if you want to see more details; they are a fascinating read [L09, S09]. Also see David et al.'s excellent work for a comparison of locking strategies on modern multiprocessors [D+13].

---

[3]Like buy a print copy of OSTEP! Even though the book is available for free online, wouldn't you just love a hard cover for your desk? Or, better yet, ten copies to share with friends and family? And maybe one extra copy to throw at an enemy? (the book *is* heavy, and thus chucking it is surprisingly effective)

References

[D91] "Just Win, Baby: Al Davis and His Raiders" by Glenn Dickey. Harcourt, 1991. *The book about Al Davis and his famous quote. Or, we suppose, the book is more about Al Davis and the Raiders, and not so much the quote. To be clear: we are not recommending this book, we just needed a citation.*

[D+13] "Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask" by Tudor David, Rachid Guerraoui, Vasileios Trigonakis. SOSP '13, Nemacolin Woodlands Resort, Pennsylvania, November 2013. *An excellent paper comparing many different ways to build locks using hardware primitives. Great to see how many ideas work on modern hardware.*

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

[H93] "MIPS R4000 Microprocessor User's Manual" by Joe Heinrich. Prentice-Hall, June 1993. Available: http://cag.csail.mit.edu/raw/documents/R4400_Uman_book_Ed2.pdf. *The old MIPS user's manual. Download it while it still exists.*

[H91] "Wait-free Synchronization" by Maurice Herlihy. ACM TOPLAS, Volume 13: 1, January 1991. *A landmark paper introducing a different approach to building concurrent data structures. Because of the complexity involved, some of these ideas have been slow to gain acceptance in deployment.*

[L81] "Observations on the Development of an Operating System" by Hugh Lauer. SOSP '81, Pacific Grove, California, December 1981. *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)" by Many authors.. Available here: `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?" by A. Mayer. 1982. Available: `www.ajwm.net/amayer/papers/B5000.html`. *"It (RDLK) is an indivisible operation which reads from and writes into a memory location." RDLK is thus test-and-set! Dave Dahm created spin locks ("Buzz Locks") and a two-phase lock called "Dahm Locks."*

[M15] "OSSpinLock Is Unsafe" by J. McCall. `mjtsai.com/blog/2015/12/16/osspinlock-is-unsafe`. *Calling OSSpinLock on a Mac is unsafe when using threads of different priorities – you might spin forever! So be careful, Mac fanatics, even your mighty system can be less than perfect...*

[MS91] "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors" by John M. Mellor-Crummey and M. L. Scott. ACM TOCS, Volume 9, Issue 1, February 1991. *An excellent and thorough survey on different locking algorithms. However, no operating systems support is used, just fancy hardware instructions.*

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

[R97] "What Really Happened on Mars?" by Glenn E. Reeves. Available on our site at: `https://www.ostep.org/Citations/mars.html`. *A description of priority inversion on Mars Pathfinder. Concurrent code correctness matters, especially in space!*

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

[S09] "OpenSolaris Thread Library" by Sun.. Code: `src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libc/port/threads/synch.c`. *Pretty interesting, although who knows what will happen now that Oracle owns Sun. Thanks to Mike Swift for the pointer.*

[W09] "Load-Link, Store-Conditional" by Many authors. `en.wikipedia.org/wiki/Load-Link/Store-Conditional`. *Can you believe we referenced Wikipedia? But, we found the information there and it felt wrong not to. Further, it was useful, listing 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). Actually Wikipedia is pretty amazing, so don't be so harsh, OK?*

[WG00] "The SPARC Architecture Manual: Version 9" by D. Weaver, T. Germond. SPARC International, 2000. `http://www.sparc.org/standards/SPARCV9.pdf`. *See the website: developers.sun.com/solaris/articles/atomic_sparc/ for more on atomics.*

Homework (Simulation)

This program, `x86.py`, allows you to see how different thread inter-leavings either cause or avoid race conditions. See the README for details on how the program works and answer the questions below.

Questions

1. Examine `flag.s`. This code "implements" locking with a single memory flag. Can you understand the assembly?

2. When you run with the defaults, does `flag.s` work? Use the `-M` and `-R` flags to trace variables and registers (and turn on `-c` to see their values). Can you predict what value will end up in `flag`?

3. Change the value of the register `%bx` with the `-a` flag (e.g., `-a bx=2,bx=2` if you are running just two threads). What does the code do? How does it change your answer for the question above?

4. Set `bx` to a high value for each thread, and then use the `-i` flag to generate different interrupt frequencies; what values lead to a bad outcomes? Which lead to good outcomes?

5. Now let's look at the program `test-and-set.s`. First, try to understand the code, which uses the `xchg` instruction to build a simple locking primitive. How is the lock acquire written? How about lock release?

6. Now run the code, changing the value of the interrupt interval (`-i`) again, and making sure to loop for a number of times. Does the code always work as expected? Does it sometimes lead to an inefficient use of the CPU? How could you quantify that?

7. Use the `-P` flag to generate specific tests of the locking code. For example, run a schedule that grabs the lock in the first thread, but then tries to acquire it in the second. Does the right thing happen? What else should you test?

8. Now let's look at the code in `peterson.s`, which implements Peterson's algorithm (mentioned in a sidebar in the text). Study the code and see if you can make sense of it.

9. Now run the code with different values of `-i`. What kinds of different behavior do you see? Make sure to set the thread IDs appropriately (using `-a bx=0,bx=1` for example) as the code assumes it.

10. Can you control the scheduling (with the `-P` flag) to "prove" that the code works? What are the different cases you should show hold? Think about mutual exclusion and deadlock avoidance.

11. Now study the code for the ticket lock in `ticket.s`. Does it match the code in the chapter? Then run with the following flags: `-a bx=1000,bx=1000` (causing each thread to loop through the critical section 1000 times). Watch what happens; do the threads spend much time spin-waiting for the lock?

12. How does the code behave as you add more threads?

13. Now examine `yield.s`, in which a `yield` instruction enables one thread to yield control of the CPU (realistically, this would be an OS primitive, but for the simplicity, we assume an instruction does the task). Find a scenario where `test-and-set.s` wastes cycles spinning, but `yield.s` does not. How many instructions are saved? In what scenarios do these savings arise?

14. Finally, examine `test-and-test-and-set.s`. What does this lock do? What kind of savings does it introduce as compared to `test-and-set.s`?