

Locks: Introduction

From the last note, we saw that we had a fundamental problem in concurrent programming: we would like to execute a series of instructions atomically, but due to the presence of interrupts, we couldn't. In this note, we thus attack the problem of how to provide what is called a **lock**: basically, what we are trying to do is put some code around these critical sections and thus enable them to appear to execute as if they were a single atomic instruction.

As an example, assume our critical section was code that looked like this:

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

We would then add some code around it to achieve the desired effect:

```
lock();  
balance = balance + 1;  
unlock();
```

Note that sometimes a lock is called a **mutex**, as it is used to provide mutual exclusion. Thus, when you see the following POSIX threads code, you should understand that it is doing the same thing as above:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&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 increases 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).

DESIGN TIP: LOCK GRANULARITY

One of the key aspects in designing multi-threaded data structures is how you decide to protect them under concurrent access. One big lock only allows one thread to access the data structure at a time, and thus limits concurrency (the **coarse-grained approach**). Having more locks allows more threads to access the data structure, but is usually more complex (the **fine-grained approach**). A typical example is a hash table, where you can have one lock for the entire table or perhaps one lock per hash bucket. Thus, when thinking about multi-thread safe data structures, the first thing you should think about is whether you need concurrent access for the sake of performance. If you do, you'll need to think about more sophisticated, fine-grained locking approaches.

19.1 Requirements

Before building some synchronization primitives (such as locks), we first list three requirements that we'd like for our solution:

- **Mutual exclusion:** Obviously, we'd like to ensure that

only one process at a time can enter a critical section. Thus, our most basic requirement is that our solution does so.

- **Deadlock freedom:** When using locks and other synchronization primitives, we are basically allowing one thread to enter a critical section while preventing others from doing so. Thus, when other threads run and try to enter a critical, they may be forced to **wait**. In more complex locking scenarios, we may accidentally cause all threads to wait, thus prohibiting the program from making progress. We call such a situation **deadlock**, and we'll be discussing it in some detail later on. Any solution we have should not lead to deadlock when used properly; thus, our solution should ensure that threads can make **progress**.
- **Starvation free:** Because we have multiple threads potentially trying to enter a critical section, and many of them wait, what our ideal solution would guarantee is that eventually, all of the waiters will enter the critical section. We sometimes call this requirement **bounded wait**, as we would like to ensure that a waiting thread only needs to wait for a bounded, finite amount of time before entering a critical section.

(add some references here to the early papers that lay out these properties)

Beyond these requirements, we of course will be aware of the usual things. For example, simplicity is good; the primitives that we develop should be easy to use. Performance is also paramount; paying a high cost to enter and leave critical sections will defeat one of the purposes of using threads, which is higher performance through parallelism.

19.2 First Try: Turning Off Interrupts

Given that interrupts were the problem, we could attack them directly in our solution. Thus, here is one way to implement `lock()` and `unlock()`:

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

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

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

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, we could get in trouble in a few 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.

Second, the approach does not work on multiprocessors. Each CPU typically has its own interrupt mask and thus turning off one will not prevent a thread running on another processor from entering the critical section.

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

For these reasons, turning off interrupts is only used in limited contexts as a mutual-exclusion primitive. For example, in a single CPU system, the operating system itself will sometimes

use interrupt masking to guarantee atomicity when accessing its own data structures. This usage makes sense, as the OS always trusts itself to perform privileged operations anyhow.

19.3 Next Try: Developing Peterson's Algorithm

We next develop an approach to build a locking primitive but without resorting to the interrupt disabling of above. This approach was developed by Gary Peterson of the University of Rochester in 1981, as a generalization of a solution posed by Dekker [D68] in earlier years. The problem itself was formulated by Dijkstra [D01].

The neat thing about this approach is that it does not assume much of the hardware. Specifically, it assumes that loads and stores are atomic (even across processors) and that they execute in order. No special synchronization instructions are required; later, we will add some new hardware instructions to help us with synchronization.

19.4 First Attempt: Software Test-And-Set

We now start to sketch out key pieces of Peterson's approach. The first component is a **flag** variable. Assume in this example, we only have two threads that may (or may not) enter our critical section. We thus add a flag which can (hopefully) be used to test whether or not a thread is in the critical section.

Note that in all of these examples, we will always write three pieces of pseudocode: an `init()` code to set any variables we need to their desired initial values, and of course `lock()` and `unlock()`. Also note that we will not write this code using a lock variable (as in the `pthread` example above); it is a straightforward exercise to add such a variable to the code snippets if so desired.

In this first attempt (Figure 19.1), the idea is quite simple: use a simple variable to indicate whether some thread has 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),

```
void init() {
    // 0 indicates that lock is available, 1 that it is held by a thread
    flag = 0;
}

void lock() {
    while (flag == 1)
        ; // spin-wait (do nothing)
    flag = 1;
}

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

Figure 19.1: First Attempt: A Single Flag

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 in Figure 19.2 (assume we start in the state `flag=0`).

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

Thread 0	Thread1
<pre> call lock() while (flag == 1) // flag=0 -> continue [INTERRUPT, SWITCH TO THREAD 1] flag = 1; // also sets flag to 1(!) </pre>	<pre> call lock() while (flag == 1) // flag=0 ... flag = 1; // set flag to 1 [INTERRUPT, SWITCH TO THREAD 0] </pre>

Figure 19.2: Why Attempt 1 Fails

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! Thus, as we move forward and develop more sophisticated solutions, we should also consider ways to avoid this kind of waste.

HOW TO THINK 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.

19.5 Second Try: Per-Thread Flags

Our next try will avoid this problem by adding a single flag per thread. The code for this attempt is found in Figure 19.3.

```
void init() {
    // 1 indicates that the thread wants to enter the critical section
    flag[0] = flag[1] = 0;
}

void lock() {
    flag[self] = 1;
    while (flag[1-self] == 1)
        ; // spin-wait
}

void unlock() {
    flag[self] = 0;
}
```

Figure 19.3: Try 2: Per-Thread Flags

A small note about the code: we now will still assume that there are only two threads that may enter this code. We will further assume that each has access to some kind of thread identification number which we will call `self`. For thread 0, `self=0`, and for thread 1, `self=1`. To get access to the other thread, one simply should access `1-self` (i.e., because for thread 1, `1-self` is 0, and for thread 0, `1-self` is 1).

This approach tries to get around the test-then-set approach by first setting one's own flag and then testing the other flag (i.e., set-then-test). Thus, it provides mutual exclusion: there is no way for two threads to enter the critical section. Excellent!

However, it has a different problem: deadlock. Not excellent. Check out the interleaving in Figure 19.4.

As you can see, it is possible for thread 0 to set its own flag and an interrupt to occur *before* the test of thread 1's flag. In that case, thread 1 sets its own flag and then tests to see if thread 0 has set its flag, which it already has! Thus, thread 1 spins waiting for thread 0 to release the lock. Unfortunately, when thread 0 runs (eventually, after the time slice is up and the timer interrupt goes off), it too hits the while loop, tests the other thread's flag, finds that it is set as well, and also spins. Thus, we have a **deadlock**; both processes spin indefinitely.

Thread 0	Thread1
<pre> call lock() flag[0] = 1; [INTERRUPT, SWITCH TO THREAD 1] while (flag[1] == 1) ; // spin forever </pre>	<pre> flag[1] = 1; while (flag[0] == 1) ; // spin forever ... [INTERRUPT, SWITCH TO THREAD 0] </pre>

Figure 19.4: Why Try 2 Fails

19.6 Third Try: Whose Turn Is It?

We will now try a different approach to getting into the critical section, using something we will call a **turn** variable. The basic idea here is to use a single variable to determine whose turn it is to enter the critical section. Because setting the turn to 1 or 0 is atomic, we should be able to use the turn to make sure only one thread gets into the critical section. The code for this approach is found in Figure 19.5.

```

void init() {
    turn = 0;
}

void lock() {
    // wait for my turn (or rather, for it NOT to be the other thread's turn)
    while (turn == (1 - self))
        ; // spin-wait
}

void unlock() {
    // now I am done, so I will make it the other thread's turn
    turn = 1 - self;
}

```

Figure 19.5: Try 3: A Turn Variable

This approach also provides mutual exclusion, because the

turn is set atomically (i.e., the C statement that sets `turn = 1 - self` boils down to a single store without worry of a race condition), and the test is quite simple: just wait for it to be your turn. In this way, we achieve mutual exclusion.

Unfortunately, the turn variable has a problem. While it will work well when there are two threads alternating which thread gets to enter the critical section, it does *not* work when one thread tries to enter the critical section twice in a row; see Figure 19.6. Note that Thread 1 doesn't do anything in this example (hence, the problem).

```

                                Thread 0                                Thread1

call lock()
while (turn == (1 - self)) // it doesn't, so continue
... // do whatever it is you do in critical section
call unlock()
turn = 1 - self; // sets turn to 1

call lock()
while (turn == (1 - self)) // it does, alas
    ; // spin forever!

```

Figure 19.6: Why Try 3 Fails

As you can see from the diagram, although thread 0 can acquire the lock once, the next time it tries, the turn is set to 1, and thus thread 0 waits forever for its turn to come. Clearly, a desirable solution will allow a critical section to be entered twice in a row by the same thread.

19.7 Finally: Peterson's Algorithm

We now have all the ingredients to assemble Peterson's algorithm. The idea is simple: combine the per-thread flags to indicate intent to enter the lock, and use the turn variable to determine which thread should enter in the rare case that both wish to enter at the same time. The code is found in Figure 19.7.

To understand why this approach works, let us go through

```
void init() {
    flag[0] = flag[1] = 0; // 1 -> thread wants to acquire lock (intent)
    turn = 0; // whose turn is it? (thread 0 or thread 1?)
}

void lock() {
    flag[self] = 1;
    turn = 1 - self; // be generous: make it the other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while other thread has intent
        // AND it is other thread's turn
}

void unlock() {
    flag[self] = 0; // simply undo your intent
}
```

Figure 19.7: Try 4, a.k.a. Peterson's Algorithm

a few interesting cases, examining where previous approaches failed and why this one will succeed. First, let us assume we just have one of our two threads repeatedly entering the critical section. With just the turn variable, we saw this was a problem before, as the turn was sometimes set to the other thread and thus required threads to alternate in acquiring the lock. Here, it seems like it could be worse, as the calling thread always sets the turn variable to the other thread's turn. Fortunately, we have the intent flag to help us out (Figure 19.8).

Thus, we can see that the problem with turn variables is addressed with the addition of per-thread flags. What about mutual exclusion? See if you can prove to yourself that Peterson's does what we need it to do, assuming (again) that loads and stores to memory occur in order and are atomic.

19.8 The Problem With Peterson's

Peterson's algorithm is a great way to start thinking about multi-threaded programming. Unfortunately, it has a few problems that prevent us from using it. First, spin-waiting can be

```
Thread 0                                     Thread1

call lock()
flag[0] = 1;
turn = 1; // set it to other thread's turn
while ((flag[1] == 1) && (turn == 1))
// does not spin, because other thread has no intent (flag[1] = 0)
...
critical section
...
flag[0] = 0;

// next call to lock by thread 0 will proceed the same way
```

Figure 19.8: Repeated Calls to Lock Work Fine

highly wasteful, as a thread can spend a great deal of CPU time waiting for another thread to release a lock. Secondly, for reasons that are outside the scope of this document, Peterson's algorithm actually does not actually work on modern out-of-order processors (turns out modern CPUs do some crazy things behind the scenes to improve performance, including funky models of how memory gets updated across processors and out-of-order execution within a single processor). Thus, we are going to need some help from the hardware and the OS itself to get locking to work properly. Wow, that was a lot of work for a non-working solution.

References

[D68] "Cooperating sequential processes"

Edsger W. Dijkstra, 1968

Available: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>

[D01] "My recollections of operating system design"

E.W. Dijkstra

April, 2001. Circulated privately

Available: <http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF>

[P81] "Myths About the Mutual Exclusion Problem"

G.L. Peterson

Information Processing Letters. 12(3) 1981, 115–116