If you thought you could do multi-threading, then play "The Deadlock Empire" games

This is about a Delphi specific web game focussing on the concurrency issues in multi-threading environments. Each round you play the machine that executes the example code and tries to let it crash in a multi-threading way. Be inventive, think like a bad guy and let the two threads in the sample code deadlock, end up in the same critical section or fail in a debugging assertion.

During the workshop we will play each round interactively: all attendees play the round followed by a short discussion. This is about collective learning, so the speaker will probably learn just as much as attendees do: teaching is learning and learning is teaching.

During the conference you will also get a link to the orginal C# version from which Jeroen derived the Delphi version.

Difficulty level: 2 (intermediate)

Game URLs

- Delphi (adapted by me, need to merge changes from the original)
  - Game: https://jpluimers.github.io/deadlockempire.github.io/
  - Source: https://github.com/jpluimers/deadlockempire.github.io
  - Delphi feature branch: https://github.com/jpluimers/deadlockempire.github.io/tree/feature/Delphi-language-and-Delphi-RTL
  - Deployment settings: https://github.com/jpluimers/deadlockempire.github.io/settings/pages (only for me)
- C#: (original version, still maintained)
  - Game: https://deadlockempire.github.io/
  - Source: https://deadlockempire.github.io/

The game

This is about a web game focussing on the concurrency issues in multi-threading environments.

The goal is to break the multi-threaded code where in each exercise you basically act as the `Scheduler` for two or more threads.

Each thread has:

- Lines (the game calls them `instructions`) with statements.
  - you can hover with your mouse over a line to get extra information
- A `Step` button will execute the line or (when expanded) an atomic operation.
- A `Expand` button that is enabled when a line can be split into atomic operations.

Each level has buttons at the top:

- `Undo` will undo the last `Step` (of either thread: it remembers the order of steps taken).
- `Reset level` will start over the level at the beginning.
- `Return to main menu` will reset the level then return to the main menu.
- if you have `won` a level a `View Congratulations` button to view your accomplishment for that level.

At the bottom of each level you will see the `global` static variables.

You `win` if either of these occur:

- both threads and up in a `critical_section();` at the same time.
- one thread executes `Debug.Assert(false);`.
- both threads deadlock each other.

You `loose` if

- both threads reach the end
- both threads loop eternally (not technically a loose, but a draw: you don't get anywhere)
- either thread reaches an `Environment.Exit()` call

If you win:

- The main menu will know you've finished the level and unlock access to the next level.
- A popup will show with three buttons:
  - `Back to the code` shows the last code state and reveals a button `View Congratulations`.
  - `Return to Main Menu` shows the main menu (wich a green checkmark in front of all `won` levels).
  - `Next challenge` goes to the next level..

References:

- My blog post If you thought you could do multi-threading, then play “The Deadlock Empire” games
- https://plus.google.com/u/0/107713441330014756065/posts/PW4aZCRWBJj (now defunct as it didn't make it into the Wayback Machine: thanks Alfabet for killing G+!)

Concepts per level

Tutorials

Tutorial 1: Interface

1. Remember you are the `Scheduler`.
2. Break the threads some way.
3. Here: ensure the both reach `critical_section();`

First, follow the tutorial hints and learn about the interface. You may refresh the page to view the hints again.

Then, step through the two programs until the active instructions of both threads are on the critical section line.

Tutorial 2: Non-Atomic Instructions

1. `Expand` button.
2. Atomic parts of a line.

Combine the power of both buttons `Expand` reveals more code that you can `Step` through individually.

It means not all code executes atomically.

Unsynchronized Code

Boolean Flags Are Enough For Everyone

1. Loops combined with
    - boolean variables,
    - non-atomic instuctions and
    - `critical_section();`.

If you miss the right order, just click `Reset level` and try again. You can also `Undo` all your actions.

If two threads enter a critical section at the same time, the program is not thread-safe and thus you win the challenge.

The `while` loop at the beginning is called a `guard` - it prevents execution from continuing into a critical section under certain conditions.

However, this here is a weak guard. After you pass it in one thread, if you stop at the right time, you will be able to pass it in the other thread, too.

Simple Counter

1. Loops combined with non-boolean variables, non-atomic instuctions and `critical_section();`.

Here also you must make both threads enter the critical section.

If you'd like to reset the counter, use the orange `Reset level` button on the right.

Confused Counter

1. Loops combined with non-boolean variables, non-atomic instuctions, compound conditions and `critical_section();`.
2. `Debug.Assert(false);`

Could it be that some instructions are hidden from sight?

Most instructions are *not* atomic. That means that context may switch during the instruction's execution. For assignments, for example, it means that the expression may be read into registers of a thread, but then context may switch and when the thread receives priority again, it won't read the expression again, it will simply write the register into the left-hand variable.

To win this level, you must *execute* the *failure instruction* . It represents a point in the program that should never be executed normally.

Locks

Insufficient Lock

1. Monitor class
    - you can use any object instance as mutex, then call these methods
        - `Monitor.Enter(mutex);` to aqcuire exclusive access to the mutex
        - `Monitor.Exit(mutex);` to release exclusive access to the mutex
        - a thread having exclusive access can call `Enter` multiple times has to call `Exit` the same number of times

Locks (or *mutexes*, from *mut*ual *ex*clusion) disallow more threads from running some code at the same time. At any point in time, a lock is either *locked* (or *held*) by one thread, or it's *unlocked*. Locks have two basic operations: *locking* and *unlocking*.

When thread X tries to *lock* an unlocked lock L, the lock is granted to X and nobody else can lock L again until X *unlocks* it. On the other hand, if L is already held by another thread Y, X cannot obtain the lock L. You can usually choose what happens then: common options are "*block until Y releases L and retry*", "*try to lock L immediately, fail if L is locked*", and "*wait for Y to unlock L, but give up after a timeout period*".

You might wonder why do we need the "*block until Y releases L and retry*" option: you could accomplish something similar by using "*try to lock immediately*" in a loop: `while (!locked) { if (TryLock(obj) { break; } }`. (This pattern is called a *spinlock*.) The problem with this loop is that it *actively waits*. If you let this loop run for 1 second without letting the thread that holds the lock progress, it will just keep reentering the loop, without any hope of progress until the lock is released (by another thread). Basically, the computer just burns CPU cycles when it runs this code. The option that blocks until we manage to grab the lock tells the

runtime: "We can't go on until this other thread releases this lock, so don't even schedule us until that happens."

If more threads are waiting on the same lock, one of them will lock it when it unlocks, but you can't make any assumptions about which one will win.

In C# (unlike, for example, C++), there is no designated type for locks. Instead, all `object`s (including any classes) can act as locks and can be locked and unlocked via the System.Threading.Monitor static class.

For our purposes, we will only need `Monitor.Enter(obj)`, which locks `obj` (or waits until it unlocks and retries) and `Monitor.Exit(obj)`, which unlocks it. C# `Monitor`s are a cross between *locks* and *condition variables* (which you shall conquer later).

Finally, a lot of C# code just uses a simple pattern of locking provided by the `lock` statement, which handles most common cases nicely and is easier than fiddling with `Monitor`. `lock (obj) { ... }` is translated by the compiler to a `Monitor.Enter`/`Exit` pair on `obj`. As a bonus, it also properly handles exceptions. It's easy to forget this when you use `Monitor` directly: if anything between `Enter` and `Exit` throws an exception and the lock is not released by an exception handler (e.g., in a `finally` block), it forever remains locked by the thread which threw the exception, which is not good.

Have a look at the documentation for the lock statement if you'd like to know more.

Play with these two threads. See what happens when you lock the lock in one thread and try to lock it again in another one?

Deadlock

1. Combining multiple `mutex` instances.

A 'deadlock' is a scenario where all threads in the program wait for each other to release some resource (usually locks). None of them is willing to concede a resource before the other ones and thus the program is stuck - forever waiting for locks which will never be released. In this game, if you cause a deadlock to occur, you win the challenge immediately.

A More Complex Thread

1. `Monitor.TryEnter()`

This may appear difficult at first. There's a lot of locks, a boolean flag and critical sections. The code is not very readable and an error could be anywhere. In fact, it wouldn't surprise us if you found a solution to this challenge different from what we thought of when creating it. You should definitely write more concise and understandable code than this.

Even so, you might use this advice: In C#, locks can be locked recursively. For example, a thread can *lock* (via *Monitor.Enter*) a single object multiple times. In order to release the lock on that object and permit other threads to lock it, *all* of the locks must be released, i.e. the method *Monitor.Exit* must be called the same number of times as *Monitor.Enter*.

You did not encounter Monitor.TryEnter() yet. It does exactly what it says on the tin: it tries to lock a lock if possible. If the lock is unlocked, it locks it and returns `true`. Otherwise, the lock remains locked by its owner and the method returns `false`.

Manual Reset Event

1. System.Threading.ManualResetEventSlim class
    - initial state `non-signaled`
    - `Wait` blocks on `non-signaled` waiting until `signaled`
    - `Set` makes state `signaled`
    - `Reset` makes state `non-signaled`

The System.Threading.ManualResetEventSlim (which supersedes the older ManualResetEvent that is maintained for backwards compatibility only) is an object with a single `boolean` flag and three methods - `Wait`, `Set` and `Reset`. The `Wait()` method blocks **until** the "reset event" is "signaled" (it starts out nonsignaled). Whether the event is signaled or not is set manually by the programmer using the methods `Set()` and `Reset()`.

For example, you might want to block a thread until some long computation finishes on another thread. To do this, you might create a `ManualResetEvent` named `computationFinished` and call `computationFinished.Wait()` on the first thread and `computationFinished.Set()` on the computing thread after the computation is complete.

> BUGFIX (February 9, 2016): Previously, the `.Wait()` errorneously blocked when the reset event was signaled (the reverse of what it does in C#). This has been fixed.

Countdown Event

1. CountdownEvent class
    - initial state is `non-signaled` and a `Count` value indicating the number of `Signal` calls that need to be made to set state to `signaled`
    - `Signal` decrements `Count` until it reaches `0` at which moment the state becomes `signaled`

The CountdownEvent class has an internal counter and is initialized with a number. Its .Signal() method atomically decrements the counter. Its .Wait() method blocks the calling thread until the counter reaches zero. You can use this primitive to wait until all threads finished their work if you know the size of the work, for example. Its advantage compared to the Barrier is that you can wait without signalling, and that you can signal multiple times from the same thread.

However, this could also be a vulnerability if handled improperly, as you can see in this level.

Countdown Event Revisited

1. `CountdownEvent` revisited.

This is now much simpler, no? This CountdownEvent is going to be a breeze for you, Scheduler.

The Barrier

1. Barrier class

The Barrier class is quite safe when used correctly, though it must have been difficult to create correctly for the developers of the .NET framework. The Barrier has a fixed *number of participants* - in this case, **two**. It has only one useful method - `.SignalAndWait()` that blocks until all participants reach it. Then, all participant threads are let through the barrier and the barrier resets.

Semaphores

Semaphores

- SemaphoreSlim

Semaphores limit the number of threads that can access a resource at the same time. In C#, they are implemented by the SemaphoreSlim class. You can imagine a semaphore as a stack of coins. When a thread wants to access the resource protected by the semaphore, it needs to take a coin. Once it's done, it returns the coin to the stack.

To take a coin, you can call the .Wait() method on the semaphore. If there are no coins on the stack, the method waits until someone returns a coin. If you don't want to wait forever, you can pass it how long should it wait, in milliseconds. In that case, `.Wait()` will return a boolean indicating whether it obtained a coin. The .Release() method adds a coin on the stack. Normally, you would call `.Release()` only after a `.Wait()` - you would take a coin, do something while you have it, and then give it back. However, you can also call `.Release()` while you don't have any coins yourself. If you let *Thread 1* run, you will see it do this: if it can't find a coin within 500 milliseconds, it will create a new one.

The two threads below try to use a semaphore to ensure they don't enter the critical section at the same time. Can you figure out what are they doing wrong?

Producer-Consumer

- Queue

In producer-consumer scenarios, one thread produces some items that another thread consumes. For example, one thread could accept work requests from a user, and another thread could take outstanding requests and fulfill them.

Even through the producer-consumer problem might look trivial, it has some subtle complexity to it. For example, what if the consumer needs a lot of time to consume one item, while the producer produces items as fast as it can? We could run out of memory.

Semaphores are useful for producer-consumer problems. Remember the "coin stack" analogy: each item in the queue is represented by a coin; when the producer produces a new one, it adds a coin, and when the consumer consumes an item, it removes a coin.

In this challenge, your goal is cause an exception to be raised.

Producer-Consumer (variant)

- thread safety: most operations are not atomic

For this challenge, it will be useful to know that most library methods are not thread-safe and if two threads enter an unsafe method on the same object simultaneously, strange things may happen. Things that result in your victory.

Condition Variables

Condition Variables

1. Monitor class members Monitor.Wait() and Monitor.PulseAll()
    - there is more than just `Enter()` and `Exit()` there is also waiting and resuming with these methods:
        - `Monitor.Wait(mutex);` releases exclusive access to the mutex until it receives a `Pulse` or `PulseAll` signal then re
        - `Monitor.Wait(mutex);` get exclusive access to the mutex
        - `Monitor.PulseAll(mutex);` to release exclusive access to the mutex

Condition variables are, unfortunately, still a rather difficult topic. We won't even try to get you a confusing story here, they're just hard. Try. If you fail, skip.

The Final Stretch

Dragonfire

- more and more concepts are coming together
    - end up at `critical_section();` in both threads
    - ensure all `Wait` conditions are satisfied in time

To win this challenge, you must prevent the dragon from breathing fire.

Triple Danger

- watch that the level starts with `System.Collections.Generic.Queue<EnergyBurst>energyBursts [number of enqueued items: 3];`
- remember when queues throw errors?

You must eliminate this new threat soon or all will be lost - but how can you win?

Boss Fight

- everything comes together
  - non-atomic behaviour
  - lock/unlock
  - wait/resume

You reluctantly cast the spell that splits your spirit in two. And then you both enter the fortress...

Explanation of each level

These are extractions of what you see after completing each level.

You can use them as quidance to solve a level which you are struggling with.

Be sure not to use too many of these hints: only use them when you get stuck, even after a good night of sleep.

Tutorials

Tutorial 1: Interface

1. `Thread 0`: click `Step` until at `critical_section();`.
2. `Thread 1`: click `Step` until at `critical_section();`.

Race condition: both threads in the in same `critical_section();`.

Accomplishment:

> Well done, player! There are a few more important things you must learn before you will be able to play **The Deadlock Empire** on your own. Please move to the next tutorial challenge to learn about them.

Tutorial 2: Non-Atomic Instructions

1. `Thread 0`: click `Expand` to see the atomic instructions of the first line.
2. `Thread 0`: click `Step` until at `a = temp;`. You don't see the value of `temp`, but it's 1. `a` still has a value of `0`.
3. `Thread 1`: click `Step` until at `critical_section();`.
4. `Thread 0`: click `Step` until at `critical_section();`.

Race condition: both threads in the in same `critical_section();`.

At step 3., there is no need to press `Expand` like in `Thread 0` as there `temp` already has the value `1` needed in both of the `if` statements.

Accomplishment:

> Congratulations! You have completed the tutorial! You may now either proceed with the first non-tutorial level (very easy) or choose a level from the main menu yourself. We hope you will have fun

> playing this game!

Unsynchronized Code

Boolean Flags Are Enough For Everyone

1. `Thread 0`: click `Step` until at `flag = true;`.
2. `Thread 1`: click `Step` until at `flag = true;`.
3. `Thread 0`: click `Step` until at `critical_section();`.
4. `Thread 1`: click `Step` until at `critical_section();`.

Race condition: both threads in the in same `critical_section();`.

Accomplishment:

> Congratulations!

Simple Counter

1. `Thread 1`: click `Step` until at `critical_section();` and `System.Int32counter=3;`.
2. `Thread 0`: click `Step` until at `critical_section();` and `System.Int32counter=5;`.

Race condition: both threads in the in same `critical_section();`.

Accomplishment:

> As you have seen previously, once you pass a test, such as an integer comparison, you don't care about what other threads do to the operands - you have already passed the test and may continue to the critical section. To fix this program, locks would be needed.

Confused Counter

1. `Thread 0`: click `Step` until at `first++;`.
2. `Thread 0`: click `Expand`.
3. `Thread 0`: click `Step` until at `first = temp;`.
4. `Thread 1`: click `Step` until the thread finishes.
5. `Thread 0`: click `Step` until after `Debug.Assert(false);`.

Assert fired.

Accomplishment:

> And yet again the Wizard's tactics have been foiled! Hurray for simplicity!

Locks

Insufficient Lock

1. `Thread 1`: click `Step` until at `}` so `i` has value `-1`.
2. `Thread 0`: click `Step` until after `Debug.Assert(false);` as the value of `i` will go have the sequence `-1`, `1`, `3`, `5`.

Assert fired.

Accomplishment:

> Locks are the most commonly used synchronization primitive. It is very useful to know them.

Deadlock

1. `Thread 1`: click `Step` until at first `Monitor.Enter(mutex);` so it has `mutex2` locked.
2. `Thread 0`: click `Step` until at first `Monitor.Enter(mutex2);`.

Deadlock: both `mutex` and `mutex2` are locked in a cross-thread fashion.

Solution from programming point of view: lock mutexes in the same order in each thread.

Accomplishment:

> This was the most simple deadlock scenario - two threads mutually waiting for each other, because each was stuck on a different lock. Congratulations all the same for solving it!
>
> To avoid deadlocks in your programs, be very dilligent whenever you grab multiple locks. If two threads need the same locks, locking and unlocking them in the same order in both threads is one way to avoid deadlocking the two threads.

A More Complex Thread

1. `Thread 1`: click `Step` until at `Monitor.Exit(mutex);` in `else` clause so `mutex` is locked.
2. `Thread 0`: click `Step` until at `Monitor.Exit(mutex2);` in `else` clause so `flag` has value `true` and `mutex2` is locked.
3. `Thread 1`: click `Step` until at `Monitor.Enter(mutex2);` in `if` clause.
4. `Thread 0`: click `Step` until at `Monitor.Enter(mutex2);` in `if` clause so `mutex` is locked.
5. `Thread 1`: click `Step` until at `Monitor.Enter(mutex);` in `if` clause so `mutex2` is locked.

Deadlock: both `mutex` and `mutex2` are locked in a cross-thread fashion.

Accomplishment:

> The Monitor.TryEnter() method, if successful, also locks the mutex and in C#, objects can be locked recursively. In order for a lock to be released, it must be *exited* the same number of times it was *entered*. In this game, you saw that there is no matching `Monitor.Exit()` call to the `.TryEnter()` call and thus the first thread was able to lock the object, recursively, many times, making it impossible for the second thread to lock it.

Manual Reset Event

1. `Thread 0`: click `Step` until at `sync.Wait();` as sync is `not signaled`.
2. `Thread 1`: click `Step` until at `}` so `sync` becomes `signaled` (and `counter` has value `2`).
3. `Thread 0`: click `Step` until at `if (counter % 2 == 1) {`.
4. `Thread 1`: click `Step` until at second `counter++;` so `counter` has value `3`.
5. `Thread 0`: click `Step` until after `Debug.Assert(false);` because `(counter % 2 == 1` now is `true`.

Solution from programming point of view: ensure "status transfer" variables between threads are either:

- properly locked during write and read operations
- atomically accessed from write and read threads using for instance the interlocked class or InterlockedIncrement method.

Accomplishment:

> You've done well. Using `ManualResetEventSlim` is trickier if you use both `Set()` and `Reset()` rather than only `Set()`.

Countdown Event

1. `Thread 0`: click `Expand`.
2. `Thread 0`: click `Step` until at `progress = temp;`.
3. `Thread 1`: click `Step` until at `if (progress >= 30) {` and `progress` has value `30`.
4. `Thread 0`: click `Step` until after `event.Wait();` and `progress` has value `20`.
5. `Thread 1`: click `Step` until after `event.Wait();` and `event` still needs `1` `more`Signal`` calls.

Deadlock: both threads wait for `event` to get state `signaled`.

Accomplishment:

> Yes! When using the CountdownEvent, you must make extra sure that you are not leaving yourself open to deadlocks - the `.Wait()` calls will block indefinitely if not enough `.Signal()` calls have been made. Suppose you use the `CountdownEvent` for loading data. If one of threads fails to load data and somehow crashes, therefore not signalling, the program will be blocked and you won't be able to terminate the waiting threads.

Countdown Event Revisited

1. `Thread 0`: click `Step` until at `progress = progress + 20;`.
2. `Thread 0`: click `Expand.`.
3. `Thread 0`: click `Step` until at `progress = temp;`.
4. `Thread 1`: click `Step` until at `event.Wait();` and `1` more `Signal`.
5. `Thread 1`: click `Step` until at second time `event.Signal();` where `event` has state `set`.

Crash: tried to `Signal` a `CountdownEvent` that has already had `Count` calls to `Signal` which would force the countdown timer below zero.

> Note there are more ways to make this level crash.

A solution would be to use atomic change operations on the `progress` variable.

Accomplishment:

> The high-level synchronization primitives such as CountdownEvent are very safe and throw exceptions whenever something bad happens. For example, as you have just seen, it is impossible to signal if the event counter is already at zero. Good job!

The Barrier

1. `Thread 0`: click `Step` until after `barrier.SignalAndWait();`.
2. `Thread 2`: click `Step` until after second `barrier.SignalAndWait();`.
3. `Thread 1`: click `Step` until after `barrier.SignalAndWait();` opening the `barrier` for `Thread 2`.
4. `Thread 2`: click `Step` until after `fireballCharge = 0;`.
5. `Thread 0`: click `Step` until after `Debug.Assert(false);`.

Accomplishment:

> It is highly recommended that you set the participant count to exactly the number of threads using the barrier in any real-world code.

Semaphores

Semaphores

1. `Thread 0`: click `Step` until at `ss.Wait();`.
2. `Thread 1`: click `Step` until after `ss.Release();` in the `else` clause.
3. `Thread 0`: click `Step` until at `critical_section();`.
4. `Thread 1`: click `Step` until at `critical_section();` (takes 2 iterations of the loop).

Race condition: both threads in the in same `critical_section();`.

Solution: Don't make a thread `Release` a `Semaphore` it didn't acquire.

Accomplishment:

> A few factories stopped but the Parallel Wizard is hard at work repairing them. You must move on and act quickly to capitalize on this.

Producer-Consumer

1. `Thread 1`: click `Step` until at `queue.Enqueue(new Dragon());`.
2. `Thread 0`: click `Step` until at `queue.Dequeue();`.

Accomplishment:

> Admittedly, this was not an extremely difficult producer-consumer pattern to exploit but you performed quite well nonetheless.

Producer-Consumer (variant)

1. `Thread 0`: click `Step` until at `queue.Enqueue(new Golem());`.
2. `Thread 0`: click `Expand` to see how the `Enqueue` is composed of atomic instructions.
3. `Thread 0`: click `Step` until at `queue returns to a consistent state.`.
4. `Thread 1`: click `Step` until after `queue.Dequeue();`.

Crash: tried to `Dequeue` from a `queue` that is in an invalid state.

Solution: lock the `queue` when performing non-atomic operations.

Accomplishment:

> Are you ready for the next challenge?

Condition Variables

Condition Variables

Try to `Dequeue` when there are no items.

This involves `Thread 0` and `Thread 1` to get into `Monitor.Wait(mutex);` at the same time and one performing the `Dequeue` before the other.

1. `Thread 0`: click `Step` until at `wait until woken up` (after the `Monitor.Wait(mutex);` auto-expanded).
2. `Thread 1`: click `Step` until at `wait until woken up` (after the `Monitor.Wait(mutex);` auto-expanded).
3. `Thread 2`: click `Step` until after `Monitor.PulseAll(mutex);` and one item is in the `queue`.
4. `Thread 0`: click `Step` until at `Monitor.Enter(mutex);` inside `Monitor.Wait(mutex);`
5. `Thread 1`: click `Step` until at `Monitor.Enter(mutex);` inside `Monitor.Wait(mutex);`
6. `Thread 2`: click `Step` until after `Monitor.Exit(mutex);`
7. `Thread 0`: click `Step` until after `Monitor.Exit(mutex);` so the `queue` is empty and the mutex is free.
8. `Thread 1`: click `Step` until after `queue.Dequeue();`

Crash: tried to `Dequeue` from a queue that is empty.

Solution: use proper condition variables that encapsulate the condition and the monitor in one atomic operation.

Condition variable code for C# is at http://stackoverflow.com/questions/15657637/condition-variables-c-net and Delphi has built in ones in http://docwiki.embarcadero.com/Libraries/en/System.SyncObjs

A good explanation of using `PulseAll` versus `Pulse` is at http://stackoverflow.com/questions/6327278/monitor-wait-condition-variable/6331306#6331306

Accomplishment:

> Your skill is unmatched, Master Scheduler! Truly no program is safe before you

The Final Stretch

Dragonfire

You need the `fireball` to have at least a `counter` of `3` as the `Thread 0` needs three `fireball.Wait()` calls to end up in the `critical_section();`.

1. `Thread 0`: click `Step` until at `c = c + 1;` so the value of `c` is `-1`.
2. `Thread 1`: click `Step` until at `critical_section();` and `fireball` has a `counter` of `3`.
3. `Thread 0`: click `Step` until at `critical_section();`.

Race condition: both threads in the in same `critical_section();`.

Solution: ensure variables shared by threads are locked over all changes.

Accomplishment:

> *But as you march on the Wizard's citadel, a catastrophe happens! Another dragon appeared on the horizon, and on its back, a fearsome sorcerer. Can you defeat them in the next challenge? You must - because if you don't, all that is simple in the world will soon exist no more.*

Triple Danger

1. `Thread 2`: click `Step` until at `Monitor.Enter(conduit);`.
2. `Thread 1`: click `Step` until after `Monitor.Exit(conduit);` and `energyBursts` is empty.
3. `Thread 2`: click `Step` until at `energyBursts.Dequeue();`.

Crash: tried to `Dequeue` from a queue that is empty.

Solution: perform the `Count` check within the `Monitor.Enter()` lock.

Accomplishment:

> *And now, the time has come to take the battle to the enemy - to fight the Parallel Wizard in his own land, in his very lair!*

Boss Fight

Observing:

- `Thread 1` resets `darkness` and `evil` values to `0` at the end of each loop iteration
- `Thread 1` provides `fortress.Release();` in each loop iteration
  - `Thread 0` requires `fortress` to have a `counter` of at least `2`
- `Thread 0` can only enter the `if` on a different condition than `Thread 1` can which means you have to fiddle with `Expand` so that both conditions are met:
  - `Thread 0`: `darkness != 2` and `evil != 2` (easiest: `darkness == 1` and `evil == 1`)
  - `Thread 1`: `darkness != 2` and `evil = 2` (easiest: `darkness == 1` and `evil == 2`)

Steps:

1. `Thread 1`: click `Step` until at `darkness++;` for the 3rd time so `fortress` has a `counter` of `2`.
2. `Thread 1`: click `Expand` so `darkness++` gets expanded.
3. `Thread 1`: click `Step` until at `darkness = temp;` so `darkness` still has a value `0`.
4. `Thread 0`: click `Step` until at `if (fortress.Wait(500)) {` with `darkness` having a value `1` and `evil` having a value `1`.
5. `Thread 1`: click `Step` until at `Monitor.Enter(sanctum);` with `darkness` having a value `1` and `evil` having a value `2`.
6. `Thread 0`: click `Step` until at `Monitor.Enter(sanctum);` with `fortress` having a `counter` of `0` (because of two `fortress.Wait()` calls
7. `Thread 0`: click `Step` until at `Monitor.Wait(sanctum);` which automatically expands, releases the lock and ends up at `wait until woken up`.
8. `Thread 1`: click `Step` until at `critical_section();` so `sanctum` is not locked any more.
9. `Thread 0`: click `Step` until at `critical_section();`.

Crash: Two threads were in a critical section at the same time.

Accomplishment:

> **Congratulations!**
>
> *In the end...* **victory***!*
>
> *The Parallel Wizard is destroyed and his fortress crumbles at your feet. You have won. Never again will programmers over the world have to endure the difficulty of correct multithreaded programming because in defeating the Parallel Wizard, you have banished concurrency. The world will be as it was decades ago, with computer running at a reasonable speed and in the right order, as prescribed by the wise programmers.*
>
> *'Although,' you wonder, 'the tricks I used were somewhat useful... and I did feel quite a bit faster when parallelized. Perhaps there is something to this whole parallelism thing.'*
>
> *Indeed, perhaps there is, commander. Perhaps parallelism is useful, after all, Master Scheduler. The points you make are valid and maybe you should not be so quick to dismiss the advantages of parallelism and faster execution. After all, with the skills you gained fighting The Deadlock Empire, don't you think that you have become...*
>
> *...an even greater* **`Parallel Wizard`***?*
>
> Thank you, dear Scheduler, for playing The Deadlock Empire. We hope you had as much fun playing this game as we had making it. Concurrency programming is hard but it's also beautiful in a way and the world can always use more people learned in its ways. You are to be congratulated for making it this far. We are looking forward to the new software or games you will create using your knowledge of multithreading.
>
> You mastered all the lessons of The Deadlock Empire. Thank you for playing! Any thoughts about the game or ideas for improvement? We'd like to hear those! Just fill out this form.

Notes

Notes

Part	Locks: A More Complex Thread
Objective	1. follow guidance steps 2. break the code
Hint	• have the threads deadlock each other
Steps	Follow the guidance, then step threads: 1. Thread 1: step until the second `Monitor.Exit(mutex);` (now `flag` has value `true`) 2. Thread 0: step until the second `Monitor.Exit(mutex2);` 3. Thread 1: step until the first `Monitor.Enter(mutex2);` 4. Thread 0: step until the first `Monitor.Enter(mutex);` 5. Thread 1: step until the first `Monitor.Enter(mutex);` 6. Thread 0: step until the first `Monitor.Enter(mutex2);`
Learned	• Concurrency issues sometime take multiple logic-flows to occur • The order of locking matters again
Notes	There is second issue that won't win the level, but nonetheless: 1. Thread 0: step until the first `Monitor.Enter(mutex3);` 2. Thread 1: step until the first `Monitor.Enter(mutex);` Since Thread 0 never exits `mutex`, Thread 1 effectively hangs and Thread 0 loops.

Notes

```
+------------+-----------------------------------------------------------------
-------------------------------------------+
| Part       | Locks: A More Complex Thread
+============+=================================================================
===========================================+
| Objective  | 1. follow guidance steps
|            | 2. break the code
+------------+-----------------------------------------------------------------
-------------------------------------------+
```

```
| Hint       | - have the threads deadlock each other
+------------+-------------------------------------------------------------------
-----------------------------------------+
| Steps      | Follow the guidance, then step threads:
|            |   1. Thread 1: step until the second ``Monitor.Exit(mutex);`` (now
``flag`` has value ``true``)
|            |   2. Thread 0: step until the second ``Monitor.Exit(mutex2);``
|            |   3. Thread 1: step until the first ``Monitor.Enter(mutex2);``
|            |   4. Thread 0: step until the first ``Monitor.Enter(mutex);``
|            |   5. Thread 1: step until the first ``Monitor.Enter(mutex);``
|            |   6. Thread 0: step until the first ``Monitor.Enter(mutex2);``
+------------+-------------------------------------------------------------------
-----------------------------------------+
| Learned    | - Concurrency issues sometime take multiple logic-flows to occur
|            | - The order of locking matters again
+------------+-------------------------------------------------------------------
-----------------------------------------+
| Notes      | There is second issue that won't win the level, but nonetheless:
|            |   1. Thread 0: step until the first ``Monitor.Enter(mutex3);``
|            |   2. Thread 1: step until the first ``Monitor.Enter(mutex);``
|            | Since Thread 0 never exits ``mutex``, Thread 1 effectively hangs
and Thread 0 loops.
+------------+-------------------------------------------------------------------
-----------------------------------------+
```

TODO

1. Find references and send them updates when the C# changes have been re-merged into the Delphi repository
2. Resurrect my Conferences repository which currently gives git encoding errors (yup, newer versions of git are more strict on encoding, making it impossible to check out repositories that old versionf of git managed to put invalid encoded characters in: nice!)
3. Merge this repository into the Conferences repository
4. In each example show the code blocks as well.