Deadlock

*Encouraged by your victory against the armored dragons, you move on to the Refactor Lands, a critical territory where the Deadlock Empire is always trying to recruit more towns to convert to their paradigms. Even now, they are moving a great army to this territory in the hope of intimidating developers to accept their teachings. This must not be allowed to happen.*

A '[deadlock](http://en.wikipedia.org/wiki/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.

Thread 0

**Monitor.Enter(mutex);**

Monitor.Enter(mutex2);

*critical\_section*();

Monitor.Exit(mutex);

Monitor.Exit(mutex2);

Thread 1

**Monitor.Enter(mutex2);**

Monitor.Enter(mutex);

*critical\_section*();

Monitor.Exit(mutex2);

Monitor.Exit(mutex);

Tutorial 2: Non-Atomic Instructions

Many statements are not atomic and are actually composed of several "minor" statements. Whenever such a statement is the active instruction, you can "expand" it to be able to step through with more precision. Follow the path outlined below, watch changes in the global state and learn how this works.  
  
First, click Expand in the first thread to reveal what an assignment consists of.  
Then, click Step in the first thread to evaluate the expression, but not twice! We will still need the old value of 'a' (zero) in the second thread!  
Then, click Step in the second thread to move to the assignment statement and Expand it. Again, click Step to read the expression into a thread-local variable.  
The order of steps does not matter after this. You should be able to get into the critical section with both threads.  
  
If you miss the right order, just click Reset level and try again. You can also Undo all your actions.

Thread 0

**a = a + 1;**

if (a == 1) {

*critical\_section*();

}

Thread 1

**// Expand the following instruction:**

a = a + 1;

if (a == 1) {

*critical\_section*();

}

Boolean Flags Are Enough For Everyone

*The day finally came. The Deadlock Empire opened its gates and from them surged massive amounts of soldiers, loyal servants of the evil Parallel Wizard. The Wizard has many strengths - his armies are fast, and he can do a lot of stuff that we can't. But now he set out to conquer the world, and we cannot have that.  
  
You are our best****Scheduler****, commander! We have fewer troops and simpler ones, so we will need your help. Already two armies of the Deadlock Empire are approaching our border keeps. They are poorly equipped and poorly trained, however. You might be able to desync them and break their morale.*

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.

First Army

**while (true) {**

while (flag != false) {

;

}

flag = true;

*critical\_section*();

flag = false;

}

Second Army

**while (true) {**

while (flag != false) {

;

}

flag = true;

*critical\_section*();

flag = false;

}

Insufficient Lock

*The Deadlock Empire strikes again, and in force!  
  
Their dragons still have critical sections where they are weak, but this time, they have brought armored locks to hide them from us. Our artillery is not powerful enough to punch through this armor or to defeat the dragons without exploiting the critical sections. It falls upon you, Scheduler, to reveal to us a way to access the critical sections, even under the armored locks.*

*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 objects (including any classes) can act as locks and can be locked and unlocked via the [System.Threading.Monitor](https://msdn.microsoft.com/en-us/library/system.threading.monitor.aspx" \t "_blank) 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# Monitors 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](https://msdn.microsoft.com/en-us/library/c5kehkcz.aspx) 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?

Thread 0

**while (true) {**

Monitor.Enter(mutex);

i = i + 2;

*critical\_section*();

if (i == 5) {

*Debug.Assert(false)*;

}

Monitor.Exit(mutex);

}

Thread 1

**while (true) {**

Monitor.Enter(mutex);

i = i - 1;

*critical\_section*();

Monitor.Exit(mutex);

}

A More Complex Thread

You look up the Refactor Lands hill at the lone flag that shows who controls this important territory. You climb fast - you must reach it first. Unfortunately, that won't happen - not one, not two, but three enemy armies are closing in on the hill and they will all reach the flag before you do. You must do something about this, stop them somehow, if you are to claim this territory and save the poor people of Refactor Lands the trouble of changing paradigms.

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.

Thread 0

while (true) {

if (Monitor.TryEnter(mutex)) {

Monitor.Enter(mutex3);

Monitor.Enter(mutex);

critical\_section();

Monitor.Exit(mutex);

Monitor.Enter(mutex2);

flag = false;

Monitor.Exit(mutex2);

Monitor.Exit(mutex3);

} else {

Monitor.Enter(mutex2);

flag = true;

Monitor.Exit(mutex2);

}

}

Thread 1

while (true) {

if (flag) {

Monitor.Enter(mutex2);

Monitor.Enter(mutex);

flag = false;

critical\_section();

Monitor.Exit(mutex);

Monitor.Enter(mutex2);

} else {

Monitor.Enter(mutex);

flag = false;

Monitor.Exit(mutex);

}

}

The Barrier

*Soldiers of the Deadlock Empire let out a mighty cheer as a new device rolls out from their factories. It is a giant armored****wall****, covered in spikes and it is now rolling on its mighty wheels towards your troops, casting fireballs from its magical engines. You would do well to destroy this new weapon before it crushes your armies.*

The [Barrier class](https://msdn.microsoft.com/en-us/library/system.threading.barrier) 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.

Thread 0

**while (true) {**

Interlocked.Increment(ref fireballCharge);

barrier.SignalAndWait();

if (fireballCharge < 2) {

*Debug.Assert(false)*;

}

*fireball()*;

}

Thread 1

**while (true) {**

Interlocked.Increment(ref fireballCharge);

barrier.SignalAndWait();

}

Thread 2

**while (true) {**

Interlocked.Increment(ref fireballCharge);

barrier.SignalAndWait();

barrier.SignalAndWait();

fireballCharge = 0;

}

Semaphores

*You behold the Factory Lands of the Deadlock Empire. You are almost in awe. Everything runs smoothly and efficiently, all factories producing new materials at the same time without unnecessary delays, everything manned by thousands of workers. Cooperation. But there are weaknesses - it may be efficient but it is unstable. A single mistake - anywhere - can bring entire factories down. You take one look at the fearsome mechanical dragons leaving the factories on numerous conveyor bolts and your determination is sealed. You will destroy this land.*

[Semaphores](https://msdn.microsoft.com/en-us/library/system.threading.semaphoreslim(v=vs.110).aspx) 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()](https://msdn.microsoft.com/en-us/library/dd270787(v=vs.110).aspx) 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()](https://msdn.microsoft.com/en-us/library/dd235727(v=vs.110).aspx) 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?

Thread 0

**while (true) {**

semaphore.Wait();

*critical\_section*();

semaphore.Release();

}

Thread 1

**while (true) {**

if (semaphore.Wait(500)) {

*critical\_section*();

semaphore.Release();

} else {

semaphore.Release();

}

}

Condition Variables

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.

Thread 0

**while (true) {**

Monitor.Enter(mutex);

if (queue.Count == 0) {

Monitor.Wait(mutex);

}

queue.Dequeue();

Monitor.Exit(mutex);

}

Thread 1

**while (true) {**

Monitor.Enter(mutex);

if (queue.Count == 0) {

Monitor.Wait(mutex);

}

queue.Dequeue();

Monitor.Exit(mutex);

}

Thread 2

**while (true) {**

Monitor.Enter(mutex);

queue.Enqueue(42);

Monitor.PulseAll(mutex);

Monitor.Exit(mutex);

}

Simple Counter - dragon

*The Parallel Wizard, leader of the Deadlock Empire, has unleashed the first Dragons upon you - these are terrifying creatures but for some reason, these two dragons appear to have critical weakspots specifically designed to be weak. Maybe you can exploit that, Scheduler.*

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.

Five-Headed Dragon

**while (true) {**

counter++;

if (counter == 5) {

*critical\_section*();

}

}

Three-Headed Dragon

**while (true) {**

counter++;

if (counter == 3) {

*critical\_section*();

}

}

;

Test & set

def t11():

  global x, d

  i = -2

  for i in range(LOOP):

      yield

      x=TARGET

      yield

      if x != TARGET:

          yield

          assert (x!=TARGET)

  yield END

# 4 < 2 + D < 5

# D = 2.5

def t12():

  global x, d

  for (i,di) in enumerate(d):

      yield

      x = i

  yield END

General Peterson

NUM\_PROCESSES = 4

levels = [0] \* NUM\_PROCESSES

last\_to\_enter = [None] \* (NUM\_PROCESSES - 1)

def peterson\_process(pid):

for level in range(NUM\_PROCESSES - 1):

levels[pid] = level

# Bug: Missing assignment to last\_to\_enter[level]

yield # Simulate context switch

while any(

(other != pid and levels[other] >= level and last\_to\_enter[level] == pid)

for other in range(NUM\_PROCESSES)

):

yield # Wait

# Critical section

print(f"Process {pid} entering critical section")

yield # Simulate critical section

print(f"Process {pid} leaving critical section")

levels[pid] = -1

yield # Simulate context switch

<https://copilot.microsoft.com/shares/TVg581eKgEoU1eRVpDzEz>

Below is a complete analysis of the 11 problems. For each one, I describe what the example does, what malfunction it causes, what the underlying programming mistake is, and then I summarize all the information by classifying each problem along two dimensions: its observed “Outcome” and its “Root Cause.” (Keep in mind that a single problem may have aspects that belong to more than one cell in our classification.)

**Detailed Explanations**

**1. Deadlock**

* **What it does:**  
  Two threads acquire two locks in opposite orders. Thread 0 enters by first locking mutex then mutex2, while Thread 1 locks mutex2 first and then mutex.
* **Effect / Malfunction:**  
  The threads end up waiting on each other indefinitely. Neither can proceed because each holds one lock the other needs.
* **Underlying Mistake:**  
  The inconsistent ordering of lock acquisition causes a circular wait—a textbook deadlock situation.

**2. Non‐Atomic Instructions**

* **What it does:**  
  Two threads each execute an operation like a = a + 1 (with the assignment expanded into multiple steps).
* **Effect / Malfunction:**  
  When the non‐atomic steps interleave, the shared variable may have an unexpected value. This can lead, for example, to both threads erroneously deciding that it’s their turn to enter a critical section.
* **Underlying Mistake:**  
  The failure to treat composite operations as atomic leads to interleaved updates (a race condition) and, consequently, incorrect computation.

**3. Boolean Flags Are Enough For Everyone**

* **What it does:**  
  Two threads use a simple guard: they wait in a loop until a boolean flag is false, then set it to true before entering the critical section.
* **Effect / Malfunction:**  
  If one thread passes the guard and does not immediately block, the other thread can also see the flag as false (or be scheduled at just the right moment) so that both end up in the critical section concurrently.
* **Underlying Mistake:**  
  Relying on a single boolean flag—without atomic test-and-set or memory barriers—is a weak guard that does not guarantee mutual exclusion.

**4. Insufficient Lock**

* **What it does:**  
  Two threads modify a shared variable i under a lock. Thread 0 increments i by 2 and then tests whether i == 5 (triggering an assertion), while Thread 1 decrements i by 1.
* **Effect / Malfunction:**  
  Even though both threads use the same lock, the invariant that “i must not equal 5” is violated under certain schedules (the assertion fails).
* **Underlying Mistake:**  
  Although a lock is used, the coordination (i.e. the arithmetic and the order of operations within the critical section) is flawed. The locking is “insufficient” because it does not prevent an unexpected combination of operations that leads to an invariant failure.

**5. A More Complex Thread**

* **What it does:**  
  Two threads combine several locks (including a recursive lock on the same object), a try-lock call, and the use of a shared flag to determine different lock acquisition paths.
* **Effect / Malfunction:**  
  The overall complexity creates room for errors such as forgotten releases, inconsistent lock ordering, or misinterpreted flag conditions. This may lead to deadlocks and/or race conditions.
* **Underlying Mistake:**  
  Overcomplicating the locking protocol—with inconsistent ordering, reentrant locking, and conditional lock acquisition—results in a fragile synchronization design.

**6. The Barrier**

* **What it does:**  
  Threads (three in this example) synchronize their progress by calling a barrier’s SignalAndWait() method. However, the barrier is designed for only two participants.
* **Effect / Malfunction:**  
  Because the barrier expects exactly two threads, some threads may wait forever (deadlock) or signal conditions (like an assertion checking that fireballCharge is at least 2) are violated.
* **Underlying Mistake:**  
  The synchronization primitive is misconfigured: using a barrier with the wrong number of participants means that the intended coordination is not achieved.

**7. Semaphores**

* **What it does:**  
  Two threads use a semaphore to control access to a critical section. Thread 0 follows the standard wait–critical section–release flow. Thread 1, however, uses a time‑limited wait and then calls release regardless of success.
* **Effect / Malfunction:**  
  Releasing a semaphore that was never successfully acquired corrupts its count. This error can allow multiple threads to enter the critical section, breaking mutual exclusion.
* **Underlying Mistake:**  
  The logic error of calling Release() without having secured the resource (i.e. releasing without acquisition) upsets the semaphore’s intended behavior.

**8. Condition Variables**

* **What it does:**  
  Two consumer threads check a shared queue: if it is empty, they call [Monitor.Wait](https://Monitor.Wait)(mutex). A producer thread enqueues an element and signals all waiting threads using [Monitor.PulseAll](https://Monitor.PulseAll)(mutex).
* **Effect / Malfunction:**  
  Because the consumers use an if statement (rather than a while loop) when checking the condition, a spurious wake-up or simultaneous wake by multiple consumers can lead to attempting to dequeue from an empty queue.
* **Underlying Mistake:**  
  The misuse of condition variables—failing to recheck the condition in a loop—allows a race between consumers, which can result in runtime errors.

**9. Simple Counter – Dragon**

* **What it does:**  
  Two “dragon” threads each increment a shared counter and, based on different threshold conditions (one enters the critical section at a count of 5, the other at 3), try to trigger an event.
* **Effect / Malfunction:**  
  Without any synchronization around the counter, the updates interleave unpredictably. The threads may enter the critical section at unintended times or even simultaneously.
* **Underlying Mistake:**  
  The shared counter is not protected by locks (or another synchronization mechanism), so its unsynchronized updates produce a race condition.

**10. Test & Set**

* **What it does:**  
  A generator-based simulation models a test‑and‑set operation where one thread sets a shared variable x to a target value and another thread may interleave and change x. The yields simulate context switches.
* **Effect / Malfunction:**  
  If the operations on x are not atomic, the expected invariant (that x remains equal to the target after being set) can be violated, triggering an assertion failure.
* **Underlying Mistake:**  
  The test‑and‑set operation is not implemented atomically. Simulated context switches reveal that without true atomicity, the mechanism fails.

**11. General Peterson**

* **What it does:**  
  An implementation of Peterson’s algorithm (generalized for 4 processes) is provided. It uses arrays for process “levels” and a last\_to\_enter array to determine ordering.
* **Effect / Malfunction:**  
  Because the code omits the assignment to last\_to\_enter[level] (noted as a bug in the comment), the mutual exclusion guarantee is broken. Multiple processes may enter the critical section concurrently.
* **Underlying Mistake:**  
  A small but critical error in the algorithm’s implementation—the missing update to last\_to\_enter—destroys the primary synchronization invariant.

**Classification Table**

Below is a summary table that lists each problem (by its title and number) along with its observed outcome(s) and the underlying root cause(s). (Problems that have more than one “Outcome” or “Root Cause” list all applicable items.)

|  |  |  |
| --- | --- | --- |
| **Problem** | **Outcome(s)** | **Root Cause(s)** |
| **1. Deadlock** | • Deadlock (program hangs indefinitely) | • Inconsistent lock acquisition order |
| **2. Non-Atomic Instructions** | • Race condition<br>• Incorrect computation | • Non-atomic composite operations |
| **3. Boolean Flags Are Enough For Everyone** | • Mutual exclusion violation (simultaneous critical section entry) | • Inadequate guard (weak boolean flag without proper atomic check) |
| **4. Insufficient Lock** | • Invariant violation (unexpected assertion failure) | • Flawed coordination inside the lock (improper update sequencing) |
| **5. A More Complex Thread** | • Potential deadlock<br>• Race condition | • Overcomplicated/inconsistent lock management (erratic ordering/release) |
| **6. The Barrier** | • Synchronization failure<br>• Deadlock (or assertion failure) | • Misconfigured synchronization (using the wrong participant count) |
| **7. Semaphores** | • Semaphore count corruption<br>• Mutual exclusion breakdown | • Incorrect semaphore usage (releasing without a successful wait) |
| **8. Condition Variables** | • Runtime error (e.g. empty dequeue)<br>• Incorrect synchronization | • Misuse of condition variables (using “if” instead of “while” for re-check) |
| **9. Simple Counter – Dragon** | • Race condition (unsynchronized critical section entry) | • Lack of synchronization on a shared counter |
| **10. Test & Set** | • Assertion failure (inconsistent state of shared variable) | • Non-atomic test‑and‑set simulation (improper handling of context switches) |
| **11. General Peterson** | • Mutual exclusion violation (concurrent entry by multiple processes) | • Algorithmic error in synchronization (omitted assignment to last\_to\_enter) |

**Analysis and Justification**

* **Outcomes:**  
  The “outcome” of each problem reflects the observable error or behavior during execution. For example, a deadlock (Problem 1) comes from circular waiting, while a race condition (Problems 2 and 9) results from unsynchronized access. In some cases (Problems 4, 10) an invariant is violated (e.g., an assertion fails) because the shared resource ends up in an unexpected state.
* **Root Causes:**  
  The “root cause” focuses on the underlying programming mistake:
  + **Incorrect Lock Ordering:** Seen in Problem 1, where the locks are acquired in opposite orders.
  + **Non-Atomic Operations:** Problem 2 and Problem 10 illustrate that operations broken into multiple steps can be interleaved.
  + **Inadequate Guarding:** Problem 3 shows that using a simple boolean flag without atomicity may allow simultaneous critical section access.
  + **Overcomplicated or Flawed Locking Schemes:** Problems 4 and 5 demonstrate that even with locks, the design of the coordination can be wrong if the update sequence or ordering is not managed correctly.
  + **Misconfigured or Incorrectly Used Synchronization Primitives:** Problems 6, 7, and 8 arise because the primitives (barriers, semaphores, condition variables) are used in ways that do not conform to their safe usage patterns.
  + **Algorithmic Implementation Errors:** In Problem 11, a missing assignment in a well‐known synchronization algorithm (generalized Peterson) invalidates the mutual exclusion guarantee.

This classification highlights that while the observable behavior (the “outcome”) is often a symptom—deadlock, race, assertion failure—the true source of the problem comes from an incorrect understanding or use of synchronization constructs (the “root cause”). Understanding both dimensions is vital for designing robust concurrent code.

I hope this comprehensive analysis and the classification table help you better understand these concurrency issues and the mistakes that lead to them! If you’d like to discuss any of these problems further or explore remediation strategies for each, let’s dive into that next.

I've incorporated an analysis that justifies the classification choices in the table. Below, I'll explain the reasoning behind each assignment, showing how each problem fits into its respective **Outcome** and **Root Cause** categories.

**Analysis and Justification**

**Outcome Dimension: Observable Effect of the Problem**

Each problem exhibits a particular failure mode, such as **deadlock**, **race condition**, or **incorrect computation**. These observable errors arise because of flawed synchronization mechanisms. Here’s how each outcome was determined:

* **Deadlock** → Problems 1, 5, and 6
  + Deadlock occurs when threads wait indefinitely for a resource held by another, with no progress possible.
  + *Problem 1 (Deadlock)* shows a classic example of circular waiting due to inconsistent lock ordering.
  + *Problem 5 (Complex Thread)* contains multiple locking paths that can lead to deadlock when locks are acquired in conflicting orders.
  + *Problem 6 (Barrier)* results in a form of synchronization deadlock when too many participants are incorrectly included, blocking thread progression.
* **Race Condition** → Problems 2, 3, 9, and 10
  + Race conditions occur when multiple threads access shared resources without proper synchronization, leading to unpredictable behavior.
  + *Problem 2 (Non-Atomic Instructions)* illustrates race conditions by expanding an assignment into multiple steps that interleave improperly.
  + *Problem 3 (Boolean Flags Are Enough)* allows mutual exclusion failure due to weak flag guarding.
  + *Problem 9 (Simple Counter – Dragon)* demonstrates unprotected access to a shared counter where interleaving updates lead to incorrect behavior.
  + *Problem 10 (Test & Set)* shows atomicity failure: without proper test-and-set handling, threads can see inconsistent values of a shared resource.
* **Incorrect Computation** → Problems 2, 4, and 10
  + These problems produce unintended or incorrect results due to improper synchronization or interleaved operations.
  + *Problem 2 (Non-Atomic Instructions)* causes incorrect values due to unprotected assignment operations.
  + *Problem 4 (Insufficient Lock)* allows an assertion failure, meaning an invariant violation occurs due to unexpected sequencing inside locked sections.
  + *Problem 10 (Test & Set)* leads to an unexpected failure when atomicity assumptions are broken.
* **Mutual Exclusion Violation** → Problems 3, 7, 8, and 11
  + Mutual exclusion violations occur when multiple threads access a critical section simultaneously due to broken synchronization.
  + *Problem 3 (Boolean Flags Are Enough)* permits simultaneous entry because its guarding mechanism is too weak.
  + *Problem 7 (Semaphores)* allows improper semaphore use, corrupting its count and breaking its intended role in controlling access.
  + *Problem 8 (Condition Variables)* leads to incorrect queue access, as improperly handled signaling allows unexpected execution order.
  + *Problem 11 (General Peterson)* incorrectly tracks process ordering, breaking mutual exclusion guarantees.

**Root Cause Dimension: Underlying Mistake**

Now, let’s analyze what **mistakes in code design** led to each problem.

* **Incorrect Lock Ordering** → Problems 1, 5
  + Deadlock situations arise primarily due to inconsistent lock acquisition order.
  + *Problem 1 (Deadlock)* directly demonstrates this with conflicting lock order between two threads.
  + *Problem 5 (Complex Thread)* exacerbates this by introducing recursive locks, conditional locking, and inconsistent acquisition.
* **Non-Atomic Composite Operations** → Problems 2, 10
  + Interleaved, multi-step operations can introduce incorrect behavior when not treated atomically.
  + *Problem 2 (Non-Atomic Instructions)* expands an assignment into steps that interleave incorrectly.
  + *Problem 10 (Test & Set)* assumes atomicity but fails, causing assertion errors.
* **Inadequate Guarding / Weak Synchronization** → Problems 3, 8
  + Guard mechanisms that don’t properly enforce mutual exclusion allow unintended concurrency.
  + *Problem 3 (Boolean Flags Are Enough)* relies on a boolean guard that doesn’t prevent simultaneous entry.
  + *Problem 8 (Condition Variables)* incorrectly uses an if check instead of while, leading to unexpected execution sequences.
* **Flawed Coordination Inside a Lock** → Problems 4, 5
  + Even when using locks, improper coordination within the critical section may lead to unexpected results.
  + *Problem 4 (Insufficient Lock)* shows invariant failure because its updates aren’t managed properly within the lock.
  + *Problem 5 (Complex Thread)* applies inconsistent locking across multiple threads, resulting in fragile synchronization.
* **Misconfigured Synchronization Primitive** → Problems 6, 7
  + Using synchronization primitives incorrectly results in deadlocks or incorrect mutual exclusion behavior.
  + *Problem 6 (Barrier)* sets up a barrier for an incorrect number of participants, making progress impossible.
  + *Problem 7 (Semaphores)* releases a semaphore improperly, corrupting its count.
* **Algorithmic Implementation Errors** → Problems 11
  + Errors in well-known algorithms directly impact mutual exclusion guarantees.
  + *Problem 11 (General Peterson)* fails because the key tracking variable (last\_to\_enter[level]) isn’t properly assigned.

**Table Representation of Classification**

With this understanding, here’s the structured classification table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Root Cause → Outcome** | **Deadlock** | **Race Condition** | **Incorrect Computation** | **Mutual Exclusion Violation** |
| **Incorrect Lock Ordering** | 1. Deadlock, 5. Complex Thread | — | — | — |
| **Non-Atomic Composite Operations** | — | 1. Non-Atomic Instructions, 10. Test & Set | 1. Non-Atomic Instructions, 10. Test & Set | — |
| **Inadequate Guarding / Weak Sync** | — | 1. Boolean Flags Are Enough | — | 1. Boolean Flags Are Enough, 8. Condition Variables |
| **Flawed Coordination Inside a Lock** | — | — | 1. Insufficient Lock | — |
| **Misconfigured Synchronization Primitive** | 1. Barrier | 1. Semaphores | — | 1. Semaphores |
| **Algorithmic Implementation Error** | — | — | — | 1. General Peterson |

**Final Justification of Assignments**

* **Each issue is placed based on its observed failure** and **what caused that failure in the code design**.
* Some problems have **multiple causes** (e.g., *Problem 5* involves both incorrect lock ordering and flawed coordination).
* Some problems have **multiple outcomes** (e.g., *Problem 2* leads to **race conditions** and **incorrect computation**).
* The classification ensures **granularity**—each cell identifies specific issues contributing to concurrency failures.

By structuring this analysis with both **symptoms (outcomes)** and **root causes**, it provides a comprehensive understanding of how concurrency errors arise and how they can be prevented.

Let me know if you'd like adjustments or deeper dives into any specific problem! 🚀

Below is a structured analysis of the eleven concurrency bugs from the attached document, classified along the two requested dimensions—**Outcome** and **Root Cause**—with detailed explanations for each. Problems may appear in multiple cells if they exhibit more than one outcome or cause.

**1. Structured Explanations**

For each problem I provide:

1. **What it does**
2. **Effect (Outcome)**
3. **Underlying mistake (Root Cause)**
4. **Other characteristics**

**1. Deadlock (Problem 1)**

**What it does**  
Two threads each acquire two locks in opposite order:

// Thread 0

Enter(mutex) → Enter(mutex2) → CS → Exit(mutex) → Exit(mutex2)

// Thread 1

Enter(mutex2) → Enter(mutex) → CS → Exit(mutex2) → Exit(mutex)

**Effect**  
Threads can each hold one lock and wait forever for the other ⇒ **deadlock**.  
**Root Cause**  
Circular wait due to inconsistent lock acquisition order (lock-ordering violation).  
**Other**  
Classic “lock inversion” scenario in multithreaded code .

**2. Non-Atomic Instructions (Problem 2)**

**What it does**  
Two threads do a = a + 1; if (a == 1) CS. When expanded into multiple micro-steps, interleaving can let both enter CS.  
**Effect**  
Both threads enter CS ⇒ **race condition / incorrect computation**.  
**Root Cause**  
Assuming a compound statement is atomic; in reality it’s (read, increment, write), violating atomicity.  
**Other**  
Illustrates need for atomic reads/writes or explicit locking .

**3. Guard Violations (Weak Flag Guard) (Problem 3)**

**What it does**  
Two threads share a single boolean flag. Each does:

while(flag != false) { }

flag = true;

CS;

flag = false;

**Effect**  
By stepping between guard and flag-set, both can pass ⇒ **race condition / mutual-exclusion violation**.  
**Root Cause**  
Weak guard: checking flag and setting it are non-atomic, allowing interleaving.  
**Other**  
Demonstrates need for atomic test-and-set or proper lock .

**4. Insufficient Lock (Problem 4)**

**What it does**  
One thread holds a lock only around the critical section but not around the update of i that triggers an assert.

Enter(mutex); i += 2; CS; if(i==5) assert; Exit(mutex);

**Effect**  
i can be modified concurrently ⇒ **incorrect computation** or even assertion failure.  
**Root Cause**  
Lock does not cover all shared-state accesses (insufficient critical region).  
**Other**  
All accesses to shared data must be within the same lock .

**5. Spinlock vs. Blocking (Problem 5)**

**What it does**  
Demonstrates why a busy-wait spinlock (try-lock in a loop) is problematic versus Monitor.Enter (blocking).  
**Effect**  
High CPU usage; potential livelock if holder never runs.  
**Root Cause**  
Active waiting (spinlock) instead of blocking wait; non-cooperative scheduling.  
**Other**  
Not a mutual-exclusion bug per se, but a performance/correctness pitfall .

**6. Condition Variable Misuse (Problem 6)**

**What it does**  
Two consumers Wait(mutex) when queue empty; one producer PulseAll(mutex) after enqueue.  
**Effect**  
Consumers may wake spuriously or miss signals ⇒ possible **deadlock** (threads stuck) or **missed data**.  
**Root Cause**  
Using if instead of while around Wait; not re-checking condition on wakeup.  
**Other**  
Must use a loop to guard against missed or spurious wakeups .

**7. Simple Counters (Five- vs. Three-Headed) (Problem 7)**

**What it does**  
Two loops increment counter and trigger CS at different thresholds.  
**Effect**  
Both can enter CS if interleaved (race on counter) ⇒ **race condition**.  
**Root Cause**  
Non-atomic increment + check; missing synchronization.  
**Other**  
Counter++ must be atomic or protected by lock .

**8. Test-and-Set Bug (Problem 8)**

**What it does**  
Generator-based pseudo code does:

x = TARGET

if x != TARGET: assert

But interleaving can violate the expectation.  
**Effect**  
Assertion may fire incorrectly ⇒ **incorrect computation**.  
**Root Cause**  
Assuming assignment and subsequent read are atomic; ignoring memory reordering.  
**Other**  
Demonstrates memory‐model/race pitfalls .

**9. Generalized Peterson (Problem 9)**

**What it does**  
N-process Peterson missing the last\_to\_enter assignment.  
**Effect**  
Multiple processes can enter CS ⇒ **mutual-exclusion violation**.  
**Root Cause**  
Incorrect tie-breaking due to missing/incorrect update in choosing protocol.  
**Other**  
Algorithmic bug in synchronization protocol .

**10. Barrier Misuse (Problem 10)**

**What it does**  
Three threads increment fireballCharge and call SignalAndWait(). The third thread calls SignalAndWait() twice before resetting.  
**Effect**  
Barrier usage imbalance ⇒ potential **deadlock** or assertion failure if participants don’t match.  
**Root Cause**  
Incorrect number of barrier signals/waits (protocol violation).  
**Other**  
Barrier participant count and usage must align .

**11. Semaphore Misuse (Problem 11)**

**What it does**  
Thread 0 does Wait(); CS; Release();  
Thread 1 does if (Wait(500)) { CS; Release(); } else { Release(); }  
**Effect**  
Thread 1 can call Release() without a matching Wait() ⇒ semaphore count corrupted, can allow multiple CS ⇒ **mutual-exclusion violation**.  
**Root Cause**  
Release without ownership; incorrect semaphore protocol.  
**Other**  
Semaphores require one-to-one Wait/Release pairing .

**2. Classification Table**

|  | **Guard Violation** | **Non-Atomic Action** | **Lock-Ordering Violation** | **Insufficient Lock** | **Protocol Violation** | **Spinlock Misuse** |
| --- | --- | --- | --- | --- | --- | --- |
| **Deadlock** |  |  | Problem 1 |  |  |  |
| **Race Condition / Wrong Result** | Problem 3, 7 | Problem 2, 8 |  | Problem 4 | Problem 11 |  |
| **Deadlock / Hang** |  |  |  |  | Problem 6, 10 | Problem 5 |

* A problem may appear in multiple rows if it has more than one outcome (e.g., both a race and potential deadlock).
* “Protocol Violation” covers misuse of semaphores, barriers, Peterson tie-break, etc.

**3. Classification Analysis**

* **Deadlock (1, 6, 10)** arises when threads wait forever—either via lock inversion (1), missed condition-variable signals (6), or barrier misuse (10).
* **Race Conditions** happen when shared state is accessed without proper protection: non-atomic arithmetic (2, 7, 8), insufficient lock coverage (4), or semaphore misuse (11).
* **Guard Violations** in Problem 3 allow both threads to slip into CS.
* **Spinlock Misuse** (5) isn’t a correctness violation of mutual exclusion, but a performance and livelock risk.
* **Protocol Violations** (6, 10, 11, 9) break the contract of higher‐level synchronization primitives.

This classification ensures each bug is tracked by both what it **does** (the symptom) and why it **happens** (the root cause).

Below is a structured analysis of the 11 concurrency “bugs” from your document, organized by the two requested dimensions—**Outcome** and **Root Cause**—followed by a classification table and a short discussion of our choices.

**1. Deadlock**

**What it does:** Two threads each acquire two locks in opposite orders.  
**Effect:** Both threads end up waiting forever—classic deadlock.  
**Root mistake:** **Lock‐order inversion** (circular wait).  
**Other notes:** No progress can be made once each holds its first lock.

**2. Non-Atomic Instructions**

**What it does:** Two threads each do a = a + 1; if (a==1) …. Expanded, the read/add/write aren’t atomic.  
**Effect:** Both can enter the critical section despite the guard, because each reads the old a before the other writes. This is a **mutual-exclusion violation** leading to two simultaneous entries.  
**Root mistake:** **Non-atomic action** (shared variable update broken into multiple steps).  
**Other notes:** Illustrates need to either lock or use atomic read-modify-write.

**3. Boolean Flag Guard (“weak guard”)**

**What it does:** Two identical loops spin on while(flag!=false); flag=true; … flag=false.  
**Effect:** Under the right interleaving both threads break out and both enter critical section—**mutual-exclusion violation**.  
**Root mistake:** **Guard violation** (the test/assignment sequence isn’t protected, so two threads can pass the guard).  
**Other notes:** Simple “test-and-set” problem.

**4. Insufficient Lock (Assert in Critical Section)**

**What it does:** Two threads update i under one lock; one asserts i!=5.  
**Effect:** Depending on interleaving, i can reach 5—**incorrect computation** (assert fires).  
**Root mistake:** **Logic error inside lock**—the invariant check doesn’t match the update pattern.  
**Other notes:** Not a synchronization failure per se, but a misuse of critical‐section logic.

**5. Complex TryEnter / Flag Mix**

**What it does:** One thread uses TryEnter and nested locks based on a flag; the other mixes flag and locks.  
**Effect:** Can lead to both deadlock and mutual‐exclusion violation depending on scheduling.  
**Root mistake:** **Mixed protocol**: inconsistent use of TryEnter, nested locks, and flag.  
**Other notes:** Illustrates danger of combining multiple sync strategies.

**6. Barrier Misuse**

**What it does:** Three threads increment a shared counter and use a 2-party barrier.  
**Effect:** Timing may allow the Debug.Assert to fire (**incorrect computation**), or threads block indefinitely (**deadlock**) if barrier parties mismatch.  
**Root mistake:** **Wrong participant count / misuse of barrier semantics**.  
**Other notes:** Barrier resets once all parties arrive; the third thread calls it twice, throwing everything off.

**7. Semaphore Misuse**

**What it does:** Thread 0 does Wait(); CS; Release(); Thread 1 does if (Wait(500)) CS; Release();.  
**Effect:** If thread 1 times out, it still calls Release(), growing the semaphore’s count—**incorrect resource accounting** (leads to multiple simultaneous entries).  
**Root mistake:** **Improper Release on timeout**.  
**Other notes:** Demonstrates need to pair each successful Wait with exactly one Release.

**8. Condition-Variable Loop vs. if**

**What it does:** Two consumers use if(queue empty) Wait(); Dequeue(), producer does Enqueue(); PulseAll();.  
**Effect:** Spurious wakeups or missed pulses can lead to **deadlock** (consumer waits forever) or **exceptions** (dequeue on empty).  
**Root mistake:** **Use of if instead of while** around Wait.  
**Other notes:** Standard CV pattern requires a loop.

**9. Simple Counter Dragons**

**What it does:** Two loops increment a shared counter, one fires on counter==5, the other on ==3.  
**Effect:** Multiple entries to their “critical section” or none at all, depending on interleaving—**incorrect computation** and **mutual-exclusion violation**.  
**Root mistake:** **Unsynchronized shared counter** (race condition).  
**Other notes:** No locks at all.

**10. Test-and-Set Demo**

**What it does:** Thread 11 writes x=TARGET then asserts x==TARGET; thread 12 sets x=i.  
**Effect:** Depending on timing, thread 11’s assert can fail—**incorrect computation**.  
**Root mistake:** **Non-atomic write + data race**.  
**Other notes:** Echoes the non-atomic action bug.

**11. Generalized Peterson (Missing assignment)**

**What it does:** N-thread Bakery‐style Peterson: sets levels[pid]=k, but **omits** last\_to\_enter[k]=pid.  
**Effect:** Multiple threads can proceed out of order—**mutual-exclusion violation**.  
**Root mistake:** **Missing assignment** in the protocol.  
**Other notes:** Variant of Peterson’s classical correctness condition.

**Classification Table**

|  | **Guard Violation** | **Non-atomic Action** | **Lock-order Inversion** | **Missing Protocol Step** | **Mixed Protocols / Logic Error** |
| --- | --- | --- | --- | --- | --- |
| **Deadlock** | — | — | 1: Deadlock | 5: Complex TryEnter mix6: Barrier misuse (participant mismatch) |  |
| **Mutual-Exclusion Violation** | 3: Boolean Flag Guard8: CV if→while misuse | 2: Non-atomic Instructions9: Unsynced counter10: Test-and-Set demo | — | 11: Generalized Peterson (missing assignment) | 5: Complex TryEnter mix (inconsistent ordering) |
| **Incorrect Computation** | — | 2: Non-atomic Instructions10: Test-and-Set demo | — | 7: Semaphore misuse (extra Release)4: Insufficient Lock (assert i==5) | 6: Barrier misuse (wrong reset logic) |

* **Cells contain problem numbers**; some appear multiple times.
* **Empty cells (—)** indicate no examples classified under that combination.

**Classification Rationale**

* **Deadlock** arises when threads form a circular wait on locks (Lock-order Inversion, mixed protocols, barrier count mismatch).
* **Mutual-Exclusion Violation** occurs when the critical-section guard fails (weak boolean guard, wrong CV usage), when updates aren’t atomic, or when a protocol step is missing (Bakery/Peterson).
* **Incorrect Computation** captures data races (non-atomic increments, extra semaphore releases) and logic errors inside locks.

Each problem’s root cause is derived from the protocol mistake:

* **Guard Violation** for any flawed while(flag) or if(…)Wait().
* **Non-atomic Action** for assignments split into read/compute/write without protection.
* **Lock-order Inversion** where two locks are acquired in opposite orders.
* **Missing Protocol Step** when a required assignment or re-check is omitted.
* **Mixed Protocols/Logic Error** when synchronization constructs are combined badly or invariants mis-specified.

This completes the requested two-dimensional classification along with explanations for each bug and their placement in the table.

Below is a structured treatment of the 11 concurrency “challenges” in your document (). We first analyze each one, then show a 2-dimensional classification table (Outcome × Root Cause), and finally discuss our choices.

**1. Deadlock**

**What it does:** Two threads acquire two locks in opposite order.  
**Effect (Outcome):** Deadlock—both threads block forever.  
**Root Cause:** Circular wait on two mutexes (lock‐order inversion).  
**Characteristics:**

* Classic hold–and–wait.
* Neither thread ever releases its first lock after taking the second.

**2. Non-Atomic Instructions**

**What it does:** Two threads each do a = a + 1; if (a == 1) critical\_section(); but stepping through expands the read–modify–write into separate steps.  
**Effect:** Both threads can enter the critical section (race).  
**Root Cause:** Non-atomic action (read–modify–write is not protected).  
**Characteristics:**

* “Lost update” and wrong guard evaluation.
* Demonstrates need for atomicity or locks.

**3. Boolean Flag Guard**

**What it does:** Two threads use the same Boolean flag as a guard before entering a CS.  
**Effect:** Both threads can pass the guard and enter the CS (race).  
**Root Cause:** Weak guard (no per-thread flag, shared variable).  
**Characteristics:**

* Guard violation: the simple while(flag!=false) is insufficient.
* Shows need for per-thread intent variables or stronger sync.

**4. Insufficient Lock**

**What it does:** Both threads lock a single mutex but check i==5 only under lock.  
**Effect:** No deadlock, but possible wrong assertion if scheduling interleaves badly.  
**Root Cause:** Lock covers only some operations—assert may race.  
**Characteristics:**

* Demonstrates importance of protecting all shared reads/writes.

**5. More Complex Thread**

**What it does:** A convoluted mix of TryEnter, Enter, multiple flags and mutexes.  
**Effect:** Starvation or deadlock depending on flag state and recursive locks.  
**Root Cause:** Broken logic & recursive lock misuse (wrong release count).  
**Characteristics:**

* Highlights pitfalls of mixing TryEnter + multiple locks + flag.

**6. Barrier**

**What it does:** Three threads increment a counter then call barrier.SignalAndWait(). One thread signals twice and resets counter.  
**Effect:** Assertion failure or deadlock if threads miss a barrier phase.  
**Root Cause:** Incorrect use of barrier (uneven signal count).  
**Characteristics:**

* Barrier phase mismatch.
* One thread passes barrier twice, leaving others stuck.

**7. Semaphores**

**What it does:** Thread 0 does Wait(); CS; Release(); Thread 1 does Wait(timeout) and on timeout also calls Release().  
**Effect:** Semaphore count increases uncontrollably → multiple threads enter CS.  
**Root Cause:** Incorrect Release() on timeout (releases without holding).  
**Characteristics:**

* Violates semaphore invariant.
* Leads to “buildup” of permits and race in CS.

**8. Condition Variables**

**What it does:** Two consumers Wait when queue empty; one producer PulseAll after enqueue.  
**Effect:** Possible missed wakeup or spurious wait (deadlock).  
**Root Cause:** Use of if rather than while around Wait (lost wakeup).  
**Characteristics:**

* Guard must be re-checked after wake.
* Typical “lost wakeup” pitfall.

**9. Simple Counter Dragons**

**What it does:** Two threads increment a shared counter and enter CS at different target values (5 and 3).  
**Effect:** Whichever hits first gets CS; the other may never reach exactly its value (starvation or both get CS unexpectedly).  
**Root Cause:** Race on shared counter without sync.  
**Characteristics:**

* Demonstrates that non-atomic counter updates cause unpredictable outcomes.

**10. Test-and-Set Bug**

**What it does:** One thread writes x = TARGET after a yield, then checks if x != TARGET and asserts; another writes x = i.  
**Effect:** Depending on interleaving, x can be overwritten before check → assert fires (broken mutual exclusion).  
**Root Cause:** Non-atomic test-and-set and wrong instruction ordering.  
**Characteristics:**

* Shows need for atomic test-and-set or proper fences.

**11. Generalized Peterson (Missing Assignment)**

**What it does:** N-process Peterson’s with levels[i] = level but **never** updates last\_to\_enter[level].  
**Effect:** Multiple processes can proceed to CS (violation).  
**Root Cause:** Missing assignment (algorithm incomplete).  
**Characteristics:**

* Illustrates how a single omitted statement breaks correctness.

**Classification Table**

|  | **Guard Violation** | **Non-atomic Action** | **Lock-Order Inversion** | **Missing Sync Operation** | **Barrier Mismatch** |
| --- | --- | --- | --- | --- | --- |
| **Deadlock** |  |  | Deadlock (1) |  |  |
| **Race / CS violation** | Boolean Flag (3)Condition Var (8)Test-and-Set (10) | Non-Atomic (2)Counter (9) |  | General Peterson (11)Semaphores (7) |  |
| **Starvation** |  |  |  | More Complex Thread (5) |  |
| **Deadlock / Hang** |  |  |  |  | Barrier (6) |

* **(1)** Deadlock
* **(2)** Non-Atomic Instructions
* **(3)** Boolean Flag Guard
* **(5)** More Complex Thread
* **(6)** Barrier
* **(7)** Semaphores
* **(8)** Condition Variables
* **(9)** Simple Counter Dragons
* **(10)** Test-and-Set Bug
* **(11)** Generalized Peterson

**Classification Rationale**

* **Guard Violation:** problems where the only protection is a guard loop (while) that can be bypassed or lost (3, 8, 10).
* **Non-atomic Action:** examples where a high-level statement expands into multiple steps on shared data (2, 9).
* **Lock-Order Inversion:** classic deadlock due to inconsistent lock acquisition order (1).
* **Missing Sync Operation:** cases where a required algorithm step was omitted or misused (11’s missing assignment; 7’s extra Release).
* **Barrier Mismatch:** a specialized synchronization (Barrier) used incorrectly, causing phase mismatch (6).
* **Semaphores (7)** appear under **Race/CS violation** because misuse breaks the invariant and allows multiple entries.
* **More Complex Thread (5)** can cause **starvation** when recursive or conditional locking logic unfairly favors one thread.

This two‐dimensional view highlights both *what* goes wrong at runtime (deadlock, race, starvation) and *why* (guard flaw, atomicity breach, missing step, etc.).