# What is a race condition?

When writing multi-threaded applications, one of the most common problems experienced are race conditions.

My questions to the community are:

What is a race condition? How do you detect them? How do you handle them? Finally, how do you prevent them from occurring?

multithreading concurrency terminology race-condition

edited Jun 21 '16 at 20:02

TylerH
13.2k 8 43 61

asked Aug 29 '08 at 15:55

bmurphy1976
7,387 10 26 21

There is a great chapter in the Secure Programming for Linux HOWTO that describes what they are, and how to avoid them. – Craig H Aug 29 '08 at 15:59

2 I'd like to mention that - without specifying the language - most parts of this question cannot be answered properly, because in different languages, the definition, the consequences and the tools to prevent them might differ. – MikeMB Apr 21 '15 at 17:18

@MikeMB. Agreed, except when analyzing byte code execution, like it is done by Race Catcher (see this thread stackoverflow.com/a/29361427/1363844) we can address all those approximately 62 languages that compile to byte code (see en.wikipedia.org/wiki/List_of_JVM_languages) – Ben Aug 12 '16 at 5:49

## 17 Answers

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g:

```
if (x == 5) // The "Check"
{
   y = x * 2; // The "Act"

   // If another thread changed x in between "if (x == 5)" and "y = x * 2" above,
   // y will not be equal to 10.
}
```

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.

In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

```
// Obtain lock for x
if (x == 5)
{
   y = x * 2; // Now, nothing can change x until the lock is released.
              // Therefore y = 10
}
// release lock for x
```

edited Apr 7 '15 at 11:03 — Amit Joki — 43.1k 7 39 69

answered Aug 29 '08 at 16:05 — Lehane — 18k 10 42 50

61 What does the other thread do when it encounters the lock? Does it wait? Error? – Brian Ortiz Oct 19 '09 at 1:58

92 Yes, the other thread will have to wait until the lock is released before it can proceed. This makes it very important that the lock is released by the holding thread when it is finished with it. If it never releases it, then the other thread will wait indefinitely. – Lehane Oct 22 '09 at 9:01

2 @Ian In a multithreaded system there will always be times when resources need to be shared. To say that one approach is bad without giving an alternative just isn't productive. I'm always looking for ways to improve and if there is an alternative I will gladly research it and weigh the pro's and cons. – Despertar May 3 '12 at 5:16

2 @Despertar ...also, its not necessarily the case that resources will always need to be shared in a milti-threaded system. For example you might have an array where each element needs processing. You could possibly partition the array and have a thread for each partition and the threads can do their work completely independently of one another. – Ian Warburton May 3 '12 at 17:29

8 For a race to occur it's enough that a single thread attempts to change the shared data while rest of the threads can either read or change it. – SomeWittyUsername Nov 9 '12 at 16:13



A "race condition" exists when multithreaded (or otherwise parallel) code that would access a shared resource could do so in such a way as to cause unexpected results.

Take this example:

```
for ( int i = 0; i < 10000000; i++ )
{
   x = x + 1;
}
```

If you had 5 threads executing this code at once, the value of x WOULD NOT end up being 50,000,000. It would in fact vary with each run.

This is because, in order for each thread to increment the value of x, they have to do the following: (simplified, obviously)

```
Retrieve the value of x
Add 1 to this value
Store this value to x
```

Any thread can be at any step in this process at any time, and they can step on each other when a shared resource is involved. The state of x can be changed by another thread during the time between x is being read and when it is written back.

Let's say a thread retrieves the value of x, but hasn't stored it yet. Another thread can also retrieve the **same** value of x (because no thread has changed it yet) and then they would both be storing the **same** value (x+1) back in x!

Example:

```
Thread 1: reads x, value is 7
Thread 1: add 1 to x, value is now 8
Thread 2: reads x, value is 7
Thread 1: stores 8 in x
Thread 2: adds 1 to x, value is now 8
Thread 2: stores 8 in x
```

Race conditions can be avoided by employing some sort of **locking** mechanism before the code that accesses the shared resource:

```
for ( int i = 0; i < 10000000; i++ )
{
   //lock x
   x = x + 1;
```

```
    //unlock x
}
```

Here, the answer comes out as 50,000,000 every time.

For more on locking, search for: mutex, semaphore, critical section, shared resource.

edited Nov 12 '15 at 19:31

IKavanagh
**4,814** 9 24 36

answered Aug 29 '08 at 17:01

privatehuff
**1,774** 1 9 7

See jakob.engbloms.se/archives/65 for an example of a program to test how oiften such things go bad... it really depends on the memory model of the machine you are running on. – jakobengblom2 Oct 12 '08 at 19:54

this the explanation i wanted thanks enthuware.com/forum/… – shareef Aug 22 '14 at 19:26

1 How can it get to 50 million if it has to stop at 10 million? – user4624979 Oct 26 '15 at 14:24

5 @nocomprende: By 5 threads executing the same code at a time, as described directly below the snippet... – Jon Skeet Nov 12 '15 at 18:47

3 @JonSkeet You are right, I confused the i and the x. Thank you. – user4624979 Nov 12 '15 at 18:57

---

> What is a Race Condition?

You are planning to go to a movie at 5 pm. You inquire about the availability of the tickets at 4 pm. The representative says that they are available. You relax and reach the ticket window 5 minutes before the show. I'm sure you can guess what happens: it's a full house. The problem here was in the duration between the check and the action. You inquired at 4 and acted at 5. In the meantime, someone else grabbed the tickets. That's a race condition - specifically a "check-then-act" scenario of race conditions.

> How do you detect them?

Religious code review, multi-threaded unit tests. There is no shortcut. There are few Eclipse plugin emerging on this, but nothing stable yet.

> How do you handle and prevent them?

The best thing would be to create side-effect free and stateless functions, use immutables as much as possible. But that is not always possible. So using java.util.concurrent.atomic, concurrent data structures, proper synchronization, and actor based concurrency will help.

The best resource for concurrency is JCIP. You can also get some more details on above explanation here.

edited Aug 5 '15 at 6:11

ekostadinov
**5,095** 2 17 36

answered Oct 4 '13 at 21:20

Vishal Shukla
**1,169** 8 12

Code reviews and unit tests are secondary to modeling the flow between your ears, and making less use of shared memory. – A-B-B Nov 25 '13 at 22:49

1 Very good example on "check-then-act" scenario +1. – Ad Infinitum Apr 27 at 8:58

I appreciated the real world example of a race condition – Tommy O Jun 6 at 17:45

2 Like the answer *thumbs up*. Solution is: you lock the tickets between 4-5 with mutex (mutual exception, c++). In real world it is called ticket reservation :) – Volt Aug 25 at 12:11

---

There is an important technical difference between race conditions and data races. Most answers seem to make the assumption that these terms are equivalent, but they are not.

A data race occurs when 2 instructions access the same memory location, at least one of these accesses is a write and there is no *happens before ordering* among these accesses. Now what constitutes a happens before ordering is subject to a lot of debate, but in general ulock-lock pairs on the same lock variable and wait-signal pairs on the same condition variable induce a happens-before order.

A race condition is a semantic error. It is a flaw that occurs in the timing or the ordering of events that leads to erroneous program *behavior*.

Many race conditions can be (and in fact are) caused by data races, but this is not necessary. As a matter of fact, data races and race conditions are neither the necessary, nor the sufficient

condition for one another. This blog post also explains the difference very well, with a simple bank transaction example. Here is another simple example that explains the difference.

Now that we nailed down the terminology, let us try to answer the original question.

Given that race conditions are semantic bugs, there is no general way of detecting them. This is because there is no way of having an automated oracle that can distinguish correct vs. incorrect program behavior in the general case. Race detection is an undecidable problem.

On the other hand, data races have a precise definition that does not necessarily relate to correctness, and therefore one can detect them. There are many flavors of data race detectors (static/dynamic data race detection, lockset-based data race detection, happens-before based data race detection, hybrid data race detection). A state of the art dynamic data race detector is ThreadSanitizer which works very well in practice.

Handling data races in general requires some programming discipline to induce happens-before edges between accesses to shared data (either during development, or once they are detected using the above mentioned tools). this can be done through locks, condition variables, semaphores, etc. However, one can also employ different programming paradigms like message passing (instead of shared memory) that avoid data races by construction.

edited May 23 at 10:31

Community ♦
1 1

answered Aug 29 '13 at 8:45

Baris Kasikci
1,161 10 8

---

A sort-of-canonical definition is "*when two threads access the same location in memory at the same time, and at least one of the accesses is a write*." In the situation the "reader" thread may get the old value or the new value, depending on which thread "wins the race." This is not always a bug—in fact, some really hairy low-level algorithms do this on purpose—but it should generally be avoided. @Steve Gury give's a good example of when it might be a problem.

answered Aug 29 '08 at 16:21

Chris Conway
36.4k 35 110 143

3 Could you please give an example of how race conditions can be useful? Googling didn't help. – Alex V. Dec 11 '13 at 14:47

2 @Alex V. At this point, I have no idea what I was talking about. I think this may have been a reference to lock-free programming, but it's not really accurate to say that depends on race conditions, per se. – Chris Conway Dec 12 '13 at 15:31

---

A race condition is a kind of bug, that happens only with certain temporal conditions.

Example: Imagine you have two threads, A and B.

In Thread A:

```
if( object.a != 0 )
    object.avg = total / object.a
```

In Thread B:

```
object.a = 0
```

If thread A is preempted just after having check that object.a is not null, B will do `a = 0` , and when thread A will gain the processor, it will do a "divide by zero".

This bug only happen when thread A is preempted just after the if statement, it's very rare, but it can happen.

edited May 16 '13 at 5:08

Blorgbeard
67.7k 37 175 232

answered Aug 29 '08 at 16:03

Steve Gury
9,241 6 29 40

---

Race conditions occur in multi-threaded applications or multi-process systems. A race condition, at its most basic, is anything that makes the assumption that two things not in the same thread or process will happen in a particular order, without taking steps to ensure that they do. This happens commonly when two threads are passing messages by setting and checking member variables of a class both can access. There's almost always a race condition when one thread calls sleep to give another thread time to finish a task (unless that sleep is in a loop, with some checking mechanism).

Tools for preventing race conditions are dependent on the language and OS, but some comon ones are mutexes, critical sections, and signals. Mutexes are good when you want to make sure you're the only one doing something. Signals are good when you want to make sure someone else has finished doing something. Minimizing shared resources can also help prevent unexpected behaviors

Detecting race conditions can be difficult, but there are a couple signs. Code which relies heavily on sleeps is prone to race conditions, so first check for calls to sleep in the affected code. Adding particularly long sleeps can also be used for debugging to try and force a particular order of events. This can be useful for reproducing the behavior, seeing if you can make it disappear by changing the timing of things, and for testing solutions put in place. The sleeps should be removed after debugging.

The signature sign that one has a race condition though, is if there's an issue that only occurs intermittently on some machines. Common bugs would be crashes and deadlocks. With logging, you should be able to find the affected area and work back from there.

answered Aug 29 '08 at 16:12

tsellon
**1,372** 4 19 29

---

Microsoft actually have published a really detailed article on this matter of race conditions and deadlocks. The most summarized abstract from it would be the title paragraph:

> A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.

answered Sep 14 '12 at 8:00

Konstantin Dinev
**21.9k** 10 43 77

---

A race condition is a situation on concurrent programming where two concurrent threads or processes and the resulting final state depends on who gets the resource first.

answered Aug 29 '08 at 16:07

Jorge Córdoba
**28.2k** 8 61 112

---

A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly.

In computer memory or storage, a race condition may occur if commands to read and write a large amount of data are received at almost the same instant, and the machine attempts to overwrite some or all of the old data while that old data is still being read. The result may be one or more of the following: a computer crash, an "illegal operation," notification and shutdown of the program, errors reading the old data, or errors writing the new data.

answered Apr 13 '12 at 11:29

dilbag koundal
**49** 2

---

Here is the classical Bank Account Balance example which will help newbies to understand Threads in Java easily w.r.t. race conditions:

```
public class BankAccount {

/**
 * @param args
 */
int accountNumber;
double accountBalance;

public synchronized boolean Deposit(double amount){
    double newAccountBalance=0;
    if(amount<=0){
        return false;
```

```
    }
    else {
        newAccountBalance = accountBalance+amount;
        accountBalance=newAccountBalance;
        return true;
    }

}
public synchronized boolean Withdraw(double amount){
    double newAccountBalance=0;
    if(amount>accountBalance){
        return false;
    }
    else{
        newAccountBalance = accountBalance-amount;
        accountBalance=newAccountBalance;
        return true;
    }
}

public static void main(String[] args) {
    // TODO Auto-generated method stub
    BankAccount b = new BankAccount();
    b.accountBalance=2000;
    System.out.println(b.Withdraw(3000));

}
```

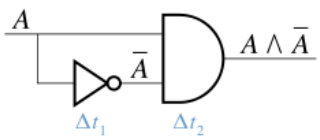
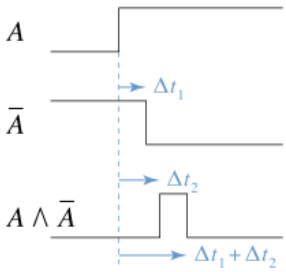
edited May 16 '13 at 5:12 — Blorgbeard — 67.7k 37 175 232

answered Nov 22 '11 at 5:35 — realPK — 420 6 15

---

Race condition is not only related with software but also related with hardware too. Actually the term initially was coined by the hardware industry.

According to wikipedia:

> The term originates with the idea of **two signals racing each other** to **influence the output first**.
>
> Race condition in a logic circuit:
>
> 
> 
>
> 
> 

Software industry took this term without modification, which makes it a little bit difficult to understand.

You need to do some replacement to map it to the software world:

- "two signals" => "two threads"/"two processes"
- "influence the output" => "influence some shared state"

So race condition in software industry means "two threads"/"two processes" racing each other to "influence some shared state", and the final result of the shared state will depend on some subtle timing difference, which could be caused by some specific thread/process launching order, thread/process scheduling, etc.

edited Sep 10 at 7:44

answered Aug 4 at 3:57 — nybon — 2,850 5 30 47

---

Ok thats 4 questions. one by one answer is as under....

> What is a race condition?

It occurs when the output and/or result of the process is critically dependent on the sequence or timing of other events i.e. e.g. 2 signals are racing to change the output first.

> How do you detect them?

It leads to error which is difficult to localize.

> How do you handle them?

Use Semaphores

And finally,

> How do you prevent them from occurring?

One way to avoid race condition is using locking mechanism for resources. but locking resources can lead to deadlocks. which has to be dealt with.

edited Nov 14 '14 at 14:46
RubberDuck
6,596 2 29 58

answered Nov 14 '14 at 13:43
Adnan Qureshi
11 2

---

Try this basic example for better understanding of race condition:

```
    public class ThreadRaceCondition {

    /**
     * @param args
     * @throws InterruptedException
     */
    public static void main(String[] args) throws InterruptedException {
        Account myAccount = new Account(22222222);

        // Expected deposit: 250
        for (int i = 0; i < 50; i++) {
            Transaction t = new Transaction(myAccount,
                    Transaction.TransactionType.DEPOSIT, 5.00);
            t.start();
        }

        // Expected withdrawal: 50
        for (int i = 0; i < 50; i++) {
            Transaction t = new Transaction(myAccount,
                    Transaction.TransactionType.WITHDRAW, 1.00);
            t.start();

        }

        // Temporary sleep to ensure all threads are completed. Don't use in
        // realworld :-)
        Thread.sleep(1000);
        // Expected account balance is 200
        System.out.println("Final Account Balance: "
                + myAccount.getAccountBalance());

    }

}

class Transaction extends Thread {

    public static enum TransactionType {
        DEPOSIT(1), WITHDRAW(2);

        private int value;

        private TransactionType(int value) {
            this.value = value;
        }

        public int getValue() {
            return value;
        }
    };

    private TransactionType transactionType;
    private Account account;
    private double amount;

    /*
```

```
     * If transactionType == 1, deposit else if transactionType == 2 withdraw
     */
    public Transaction(Account account, TransactionType transactionType,
            double amount) {
        this.transactionType = transactionType;
        this.account = account;
        this.amount = amount;
    }

    public void run() {
        switch (this.transactionType) {
        case DEPOSIT:
            deposit();
            printBalance();
            break;
        case WITHDRAW:
            withdraw();
            printBalance();
            break;
        default:
            System.out.println("NOT A VALID TRANSACTION");
        }
        ;
    }

    public void deposit() {
        this.account.deposit(this.amount);
    }

    public void withdraw() {
        this.account.withdraw(amount);
    }

    public void printBalance() {
        System.out.println(Thread.currentThread().getName()
                + " : TransactionType: " + this.transactionType + ", Amount: "
                + this.amount);
        System.out.println("Account Balance: "
                + this.account.getAccountBalance());
    }
}

class Account {
    private int accountNumber;
    private double accountBalance;

    public int getAccountNumber() {
        return accountNumber;
    }

    public double getAccountBalance() {
        return accountBalance;
    }

    public Account(int accountNumber) {
        this.accountNumber = accountNumber;
    }

    // If this method is not synchronized, you will see race condition on
    // Remove syncronized keyword to see race condition
    public synchronized boolean deposit(double amount) {
        if (amount < 0) {
            return false;
        } else {
            accountBalance = accountBalance + amount;
            return true;
        }
    }

    // If this method is not synchronized, you will see race condition on
    // Remove syncronized keyword to see race condition
    public synchronized boolean withdraw(double amount) {
        if (amount > accountBalance) {
            return false;
        } else {
            accountBalance = accountBalance - amount;
            return true;
        }
    }
}
```

answered May 31 '13 at 15:51

Morsu

**1** 1

---

You don't always want to discard a race condition. If you have a flag which can be read and written by multiple threads, and this flag is set to 'done' by one thread so that other thread stop processing when flag is set to 'done', you don't want that "race condition" to be eliminated. In fact, this one can be referred to as a benign race condition.

However, using a tool for detection of race condition, it will be spotted as a harmful race condition.

More details on race condition here, http://msdn.microsoft.com/en-us/magazine/cc546569.aspx.

answered Sep 7 '14 at 9:11

octoback
**11.4k** 24 89 152

What language is your answer based on? – MikeMB Apr 21 '15 at 17:14

Frankly it seems to me that if you have race conditions *per se*, you are not architecting your code in a tightly-controlled manner. Which, while it may not be an issue in your theoretical case, is evidence of larger issues with the way you design & develop software. Expect to face painful race condition bugs sooner or later. – Arcane Engineer Jul 13 '16 at 12:57

---

Consider an operation which has to display the count as soon as the count gets incremented. ie., as soon as **CounterThread** increments the value **DisplayThread** needs to display the recently updated value.

```
int i = 0;
```

Output

```
CounterThread -> i = 1
DisplayThread -> i = 1
CounterThread -> i = 2
CounterThread -> i = 3
CounterThread -> i = 4
DisplayThread -> i = 4
```

Here **CounterThread** gets the lock frequently and updates the value before **DisplayThread** displays it. Here exists a Race condition. Race Condition can be solved by using Synchronzation

edited Jul 15 '15 at 8:06

answered Jul 15 '15 at 8:00

bharanitharan
**1,007** 4 17 27

---

You can *prevent race condition*, if you use "Atomic" classes. The reason is just the thread don't separate operation get and set, example is below:

```
AtomicInteger ai = new AtomicInteger(2);
ai.getAndAdd(5);
```

As a result, you will have 7 in link "ai". Although you did two actions, but the both operation confirm the same thread and no one other thread will interfere to this, that means no race conditions!

edited Aug 19 at 18:19

Vishal Shukla
**1,169** 8 12

answered Aug 12 at 14:48

Aleksei Moshkov
**29** 10

---

**protected** by Hans Passant May 22 '15 at 16:45

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?