How does a "stack overflow" occur and how do you prevent it?

Asked 16 years, 4 months ago Modified 10 months ago

Viewed 158k times

150

How does a stack overflow occur and what are the ways to make sure it doesn't happen, or ways to prevent one?

memory stack-overflow

Share

Improve this question

Follow

edited Feb 15, 2023 at 12:24

Dalija Prasnikar ♦
28.5k ● 45 ● 94 ● 175

asked Aug 25, 2008 at 14:49

JasonMichael
2,573 ● 4 ● 26 ● 25

9 Answers

Sorted by: Highest score (default)

170

Stack

A stack, in this context, is the last in, first out buffer you place data while your program runs. Last in, first out (LIFO) means that the last thing you put in is always the first thing you get back out - if you push 2 items on the stack, 'A' and then 'B', then the first thing you pop off the stack will be 'B', and the next thing is 'A'.

When you call a function in your code, the next instruction after the function call is stored on the stack, and any storage space that might be overwritten by the function call. The function you call might use up more stack for its own local variables. When it's done, it frees up the local variable stack space it used, then returns to the previous function.

Stack overflow

A stack overflow is when you've used up more memory for the stack than your program was supposed to use. In embedded systems you might only have 256 bytes for the stack, and if each function takes up 32 bytes then you can only have function calls 8 deep - function 1 calls function 2 who calls function 3 who calls function 4 .... who calls function 8 who calls function 9, but function 9 overwrites memory outside the stack. This might overwrite memory, code, etc.

Many programmers make this mistake by calling function A that then calls function B, that then calls function C, that then calls function A. It might work most of the time, but just once the wrong input will cause it to go in that circle forever until the computer recognizes that the stack is overblown.

Recursive functions are also a cause for this, but if you're writing recursively (ie, your function calls itself) then you need to be aware of this and use static/global variables to prevent infinite recursion.

Generally, the OS and the programming language you're using manage the stack, and it's out of your hands. You should look at your call graph (a tree structure that shows from your main what each function calls) to see how deep your function calls go, and to detect cycles and recursion that are not intended. Intentional cycles and recursion need to be artificially checked to error out if they call each other too many times.

Beyond good programming practices, static and dynamic testing, there's not much you can do on these high level systems.

Embedded systems

In the embedded world, especially in high reliability code (automotive, aircraft, space) you do extensive code reviews and checking, but you also do the following:

- Disallow recursion and cycles - enforced by policy and testing
- Keep code and stack far apart (code in flash, stack in RAM, and never the twain shall meet)
- Place guard bands around the stack - empty area of memory that you fill with a magic number (usually a

software interrupt instruction, but there are many options here), and hundreds or thousands of times a second you look at the guard bands to make sure they haven't been overwritten.

- Use memory protection (ie, no execute on the stack, no read or write just outside the stack)
- Interrupts don't call secondary functions - they set flags, copy data, and let the application take care of processing it (otherwise you might get 8 deep in your function call tree, have an interrupt, and then go out another few functions inside the interrupt, causing the blowout). You have several call trees - one for the main processes, and one for each interrupt. If your interrupts can interrupt each other... well, there be dragons...

High-level languages and systems

But in high level languages run on operating systems:

- Reduce your local variable storage (local variables are stored on the stack - although compilers are pretty smart about this and will sometimes put big locals on the heap if your call tree is shallow)
- Avoid or strictly limit recursion
- Don't break your programs up too far into smaller and smaller functions - even without counting local variables each function call consumes as much as

64 bytes on the stack (32 bit processor, saving half the CPU registers, flags, etc)

- Keep your call tree shallow (similar to the above statement)

Web servers

It depends on the 'sandbox' you have whether you can control or even see the stack. Chances are good you can treat web servers as you would any other high level language and operating system - it's largely out of your hands, but check the language and server stack you're using. It **is** possible to blow the stack on your SQL server, for instance.

Share Improve this answer Follow

edited Feb 15, 2023 at 10:16

Cody Gray ♦
244k ● 52 ● 501 ● 581

answered Aug 25, 2008 at 23:55

Adam Davis
93.5k ● 60 ● 271 ● 333

15

A stack overflow in real code occurs very rarely. Most situations in which it occurs are recursions where the termination has been forgotten. It might however rarely occur in highly nested structures, e.g. particularly large XML documents. The only real help here is to refactor the code to use an explicit stack object instead of the call stack.

Share Improve this answer Follow

answered Aug 25, 2008 at 14:55

Konrad Rudolph
545k ● 139 ● 956 ● 1.2k

13

Most people will tell you that a stack overflow occurs with recursion without an exit path - while mostly true, if you work with big enough data structures, even a proper recursion exit path won't help you.

Some options in this case:

- Breadth-first search
- Tail recursion, .Net-specific great blog post (sorry, 32-bit .Net)

Share Improve this answer Follow

answered Aug 25, 2008 at 15:00

Greg Hurlman
17.8k ● 6 ● 55 ● 87

Totally agreed. Even without direct recursion. – Maf Feb 27, 2023 at 12:49

8

Infinite recursion is a common way to get a stack overflow error. To prevent - always make sure there's an exit path that *will* be hit. :-)

Another way to get a stack overflow (in C/C++, at least) is to declare some enormous variable on the stack.

```
char hugeArray[100000000];
```

That'll do it.

Share Improve this answer

Follow

answered Aug 25, 2008 at 14:55

Matt Dillard
14.9k ● 7 ● 52 ● 61

1 What language are you using? In C, this will almost certainly result in a stack overflow. In C#, it won't because the array is allocated on the heap and not on the stack.See this question for an example of this being hit in practice: stackoverflow.com/questions/571945/… – Matt Dillard Feb 11, 2019 at 14:23

Then arent these other languages like C# making use of the stack? Can we say that some kind of “heap overflow” would occur then? Also please, let’s not forget that we don’t need infinite recursion to have a stack overflow as your example ilustrates clearly. I was about to ask how much memory the OS allocattes to each program? Is that the same for every program? – Maf Feb 27, 2023 at 12:52

8

Aside from the form of stack overflow that you get from a direct recursion (eg `Fibonacci(1000000)`), a more subtle form of it that I have experienced many times is an indirect recursion, where a function calls another function, which calls another, and then one of those functions calls the first one again.

This can commonly occur in functions that are called in response to events but which themselves may generate new events, for example:

```
void WindowSizeChanged(Size& newsize) {
  // override window size to constrain width
    newSize.width=200;
    ResizeWindow(newSize);
}
```

In this case the call to `ResizeWindow` may cause the `WindowSizeChanged()` callback to be triggered again, which calls `ResizeWindow` again, until you run out of stack. In situations like these you often need to defer responding to the event until the stack frame has returned, eg by posting a message.

Share Improve this answer Follow

answered May 22, 2013 at 11:52

the_mandrill

30.8k ● 6 ● 69 ● 93

6

Usually a stack overflow is the result of an infinite recursive call (given the usual amount of memory in standard computers nowadays).

When you make a call to a method, function or procedure the "standard" way or making the call consists on:

1. Pushing the return direction for the call into the stack(that's the next sentence after the call)
2. Usually the space for the return value get reserved into the stack

3. Pushing each parameter into the stack (the order diverges and depends on each compiler, also some of them are sometimes stored on the CPU registers for performance improvements)
4. Making the actual call.

So, usually this takes a few bytes depeding on the number and type of the parameters as well as the machine architecture.

You'll see then that if you start making recursive calls the stack will begin to grow. Now, stack is usually reserved in memory in such a way that it grows in opposite direction to the heap so, given a big number of calls without "coming back" the stack begins to get full.

Now, on older times stack overflow could occur simply because you exausted all available memory, just like that. With the virtual memory model (up to 4GB on a X86 system) that was out of the scope so usually, if you get an stack overflow error, look for an infinite recursive call.

Share Improve this answer Follow

answered Aug 25, 2008 at 15:01

Jorge Córdoba
52.1k ● 11 ● 82 ● 130

What do you mean by “the entire memory”? Also could you provide an example in which some function variable is stored in the CPU registers as you mentioned? – Maf Feb 27, 2023 at 12:48

6

I have recreated the stack overflow issue while getting a most common Fibonacci number i.e. 1, 1, 2, 3, 5..... so calculation for fib(1) = 1 or fib(3) = 2.. fib(n) = ??.

for n, let say we will interested - what if n = 100,000 then what will be the corresponding Fibonacci number ??

The one loop approach is as below -

```
package com.company.dynamicProgramming;

import java.math.BigInteger;

public class FibonacciByBigDecimal {

    public static void main(String ...args) {

        int n = 100000;
        BigInteger[] fibOfnS = new BigInteger[n +
1];

        System.out.println("fibonacci of "+ n + "
is : " + fibByLoop(n));
    }


    static BigInteger fibByLoop(int n){

        if(n==1 || n==2 ){
            return BigInteger.ONE;
        }

        BigInteger fib = BigInteger.ONE;
        BigInteger fip = BigInteger.ONE;


        for (int i = 3; i <= n; i++){

            BigInteger p = fib;
            fib = fib.add(fip);
            fip = p;
        }

        return fib;
    }

}
```

this quite straight forward and result is -

```
fibonacci of 100000 is :
2597406934722172416615503402127591541488048538651769
```

Now another approach I have applied is through Divide and Concur via recursion

i.e. Fib(n) = fib(n-1) + Fib(n-2) and then further recursion for n-1 & n-2.....till 2 & 1. which is programmed as -

```
package com.company.dynamicProgramming;

import java.math.BigInteger;

public class FibonacciByBigDecimal {

    public static void main(String ...args) {

        int n = 100000;
        BigInteger[] fibOfnS = new BigInteger[n +
1];

        System.out.println("fibonacci of "+ n + "
is : " + fibByDivCon(n, fibOfnS));

    }


    static BigInteger fibByDivCon(int n,
BigInteger[] fibOfnS){
        if(fibOfnS[n]!=null){
            return fibOfnS[n];
        }

        if (n == 1 || n== 2){
            fibOfnS[n] = BigInteger.ONE;
            return BigInteger.ONE;
        }

        // creates 2 further entries in stack
        BigInteger fibOfn = fibByDivCon(n-1,
fibOfnS).add( fibByDivCon(n-2, fibOfnS)) ;

        fibOfnS[n] = fibOfn;

        return fibOfn;
```

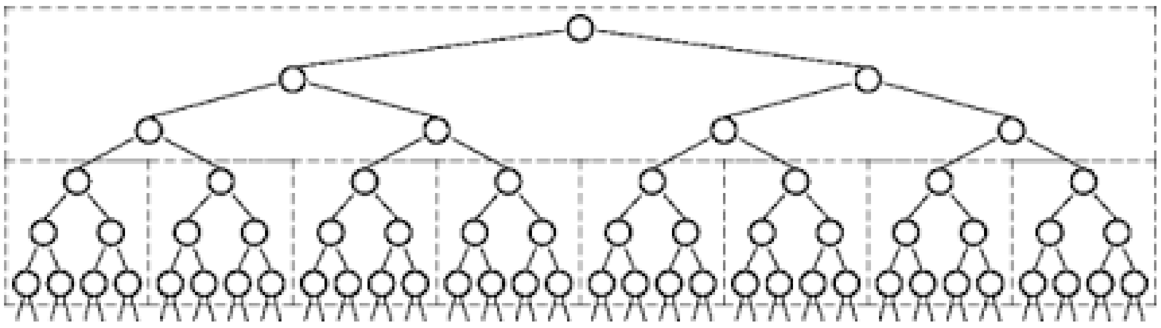
When i ran the code for n = 100,000 the result is as below -

```
Exception in thread "main"
java.lang.StackOverflowError
    at
com.company.dynamicProgramming.FibonacciByBigDecimal
    at
com.company.dynamicProgramming.FibonacciByBigDecimal
    at
com.company.dynamicProgramming.FibonacciByBigDecimal
```

Above you can see the StackOverflowError is created. Now the reason for this is too many recursion as -

```
        // creates 2 further entries in stack
        BigInteger fibOfn = fibByDivCon(n-1,
fibOfnS).add( fibByDivCon(n-2, fibOfnS)) ;
```

So each entry in stack create 2 more entries and so on... which is represented as -



Eventually so many entries will be created that system is unable to handle in the stack and StackOverflowError thrown.

For Prevention : For Above example perspective

1. Avoid using recursion approach or reduce/limit the recursion by again one level division like if n is too large then split the n so that system can handle with in its limit.
2. Use other approach, like the loop approach I have used in 1st code sample. (I am not at all intended to degrade Divide & Concur or Recursion as they are legendary approaches in many most famous algorithms.. my intention is to limit or stay away from recursion if I suspect stack overflow issues)

Share Improve this answer

Follow

edited Feb 15, 2023 at 14:31

General Grievance

4,977 ● 37 ● 36 ● 54

answered Mar 25, 2020 at 9:32

atul sachan

617 ● 6 ● 10

2

Considering this was tagged with "hacking", I suspect the "stack overflow" he's referring to is a call stack overflow, rather than a higher level stack overflow such as those referenced in most other answers here. It doesn't really apply to any managed or interpreted environments such as .NET, Java, Python, Perl, PHP, etc, which web apps are typically written in, so your only risk is the web server itself, which is probably written in C or C++.

Check out this thread:

https://stackoverflow.com/questions/7308/what-is-a-good-starting-point-for-learning-buffer-overflow

Share Improve this answer Follow

edited May 23, 2017 at 12:34
Community Bot
1 ● 1

answered Aug 25, 2008 at 23:04
Steve M
10.6k ● 12 ● 53 ● 63

1 Why do you say that?? Didn’t you see the example above with Java? Do you think these very High level languages don’t have a stack assigned to their programs by the OS?
– Maf Feb 27, 2023 at 12:43

Stack overflow occurs when your program uses up the entire stack. The most common way this happens is when

1

your program has a recursive function which calls itself forever. Every new call to the recursive function takes more stack until eventually your program uses up the entire stack.

Share Improve this answer Follow

answered Feb 13, 2023 at 5:45

zast99

37 ● 1

2 This answer does not add anything new that previous answers didn't cover. Please don't post such answers. – Dalija Prasnikar ♦ Feb 15, 2023 at 11:13