How do you implement Coroutines in C++

Asked 16 years, 3 months ago Modified 6 months ago

Viewed 57k times

80

I doubt it can be done portably, but are there any solutions out there? I think it could be done by creating an alternate stack and reseting SP,BP, and IP on function entry, and having yield save IP and restore SP+BP. Destructors and exception safety seem tricky but solvable.

Has it been done? Is it impossible?

c++ coroutine c++17

Share

Improve this question

Follow

edited Jan 6, 2017 at 16:45

Atif

1,538 ● 17 ● 26

asked Sep 23, 2008 at 15:31

Mike Elkins

1,438 ● 1 ● 11 ● 13

1 Just wanted to point out that coroutines are possible in C++. Boost is one possibility. The other is the coroutine ratified as a technical spec with C++17. There are already two

compilers with support (VC14 & Clang), and the TS will likely make it into the language post C++17. See my answer for details. – Atif Aug 16, 2016 at 16:02

1 For C programmers, here is an article, Coroutines in C by Simon Tatham that has a number of approaches. chiark.greenend.org.uk/~sgtatham/coroutines.html some are more complex than others. – Richard Chambers Oct 28, 2016 at 15:01

20 Answers

Sorted by: Highest score (default)

100

Yes it **can be done** without a problem. All you need is a little assembly code to move the call stack to a newly allocated stack on the heap.

I would **look at the boost::coroutine library**.

The one thing that you should watch out for is a stack overflow. On most operating systems overflowing the stack will cause a segfault because virtual memory page is not mapped. However if you allocate the stack on the heap you don't get any guarantee. Just keep that in mind.

Share Improve this answer Follow

edited Feb 29, 2016 at 20:04
Nayuki
18.5k ●6 ●57 ●83

answered Sep 23, 2008 at 15:38
Ted
15.3k ●6 ●30 ●28

122 I think there should be a badge for being able to mention thze work "stackoverflow" in a valid technical context on SO! – Torsten Marek Sep 23, 2008 at 15:51

7 Here's a nice standard C++ solution that doesn't require involving Boost : akira.ruc.dk/~keld/research/COROUTINE – Dan Sep 26, 2008 at 15:59

If you allocate the stack on the heap, you can do the same thing as the real stack and put a guard page at the end (or start, since it usually grows backwards) that will also cause a segfault on small overflows. – Alex Celeste Feb 5, 2016 at 15:34

3 Just a note, though boost::coroutine is a great library, c++ coroutines are on track to becoming a core c++ feature post C++17. Currently defined in a tech spec, and reference implementations are in Visual Studio 2015, and Clang: wg21.link/p0057r2 – Atif Jun 22, 2016 at 18:52

The Coroutines in c++20 are not the corroutines the OP want because they are stackless. – Lothar Aug 27, 2017 at 5:21

19

On POSIX, you can use makecontext()/swapcontext() routines to portably switch execution contexts. On Windows, you can use the fiber API. Otherwise, all you need is a bit of glue assembly code that switches the machine context. I have implemented coroutines both with ASM (for AMD64) and with swapcontext(); neither is very hard.

Share Improve this answer Follow

answered Sep 23, 2008 at 16:05

zvrba
24.5k ● 3 ● 56 ● 65

18 Unfortunately `makecontext()` and its related functions have been marked obsolescent in the IEEE 1003.1 Posix Standard in 2001 (pubs.opengroup.org/onlinepubs/009695399/functions/…) and have been removed from that standard in 2008 (blog.fpmurphy.com/2009/01/ieee-std-10031-2008.html). Also with older pthread implementations these functions are known to break a lot of stuff, and since they are now non standard hardly anybody will care about breaking them again. – LiKao Feb 14, 2012 at 9:47

1 Coroutines are on track to becoming a language feature post c++17: wg21.link/p0057r2 – Atif Jun 22, 2016 at 18:51

1 The Coroutines in c++20 are not the corroutines the OP want because they are stackless. – Lothar Aug 27, 2017 at 5:21

14

For posterity,

Dmitry Vyukov's wondeful web site has a clever trick using ucontext and setjump to simulated coroutines in c++.

Also, Oliver Kowalke's context library was recently accepted into Boost, so hopefully we'll be seeing an updated version of boost.coroutine that works on x86_64 soon.

Share Improve this answer Follow

answered Feb 24, 2012 at 22:22

tgoodhart
3,266 ● 29 ● 39

12

There's no easy way to implement coroutine. Because coroutine itself is out of C/C++'s stack abstraction just like thread. So it cannot be supported without language level changes to support.

Currently(C++11), all existing C++ coroutine implementations are all based on assembly level hacking which is hard to be safe and reliable crossing over platforms. To be reliable it needs to be standard, and handled by compilers rather than hacking.

There's a standard proposal - N3708 for this. Check it out if you're interested.

Share Improve this answer Follow

edited Dec 30, 2014 at 2:41

answered Oct 26, 2013 at 6:13

eonil
85.8k ● 89 ● 329 ● 529

2 The feature is now in a technical specification, slated for post C++17: wg21.link/p0057r2 – Atif Jun 22, 2016 at 18:49

7

You might be better off with an iterator than a coroutine if possible. That way you can keep calling `next()` to get the next value, but you can keep your state as member variables instead of local variables.

It might make things more maintainable. Another C++ developer might not immediately understand the coroutine whereas they might be more familiar with an iterator.

Share Improve this answer Follow

edited Dec 3, 2010 at 23:36

pts
87k ● 22 ● 115 ● 195

answered Sep 23, 2008 at 15:33

Steve g
2,489 ● 17 ● 17

6

For those who want to know how they may leverage Coroutines in a portable way in C++ ~~you will have to wait for C++17~~ the wait is over (see below)! The standards committee is working on the feature see the [N3722 paper]. To summarize the current draft of the paper, instead of Async and Await, the keywords will be resumable, and await.

Take a look at the experimental implementation in Visual Studio 2015 to play with Microsoft's experimental implementation. It doesn't look like clang has a implementation yet.

There is a good talk from Cppcon [Coroutines a negative overhead abstraction] outline the benefits of using Coroutines in C++ and how it affects simplicity and performance of the code.

At present we still have to use library implementations, but in the near future, we will have coroutines as a core C++ feature.

Update: Looks like the coroutine implementation is slated for C++20, but was released as a technical specification with C++17 (p0057r2). Visual C++, clang and gcc allow you to opt in using a compile time flag.

Share Improve this answer Follow

edited Nov 15, 2018 at 23:42

answered Feb 19, 2016 at 15:13

Atif
1,538 ● 17 ● 26

5

Does COROUTINE a portable C++ library for coroutine sequencing point you in the right direction? It seems like an elegant solution that has lasted the test of time.....it's 9 years old!

In the DOC folder is a pdf of the paper A Portable C++ Library for Coroutine Sequencing by Keld Helsgaun which describes the library and provides short examples using it.

[update] I'm actually making successful use of it myself. Curiosity got the better of me, so I looked into this solution, and found it was a good fit for a problem I've been working on for some time!

Share Improve this answer

Follow

edited Jan 20, 2016 at 21:36

Richard Chambers
17.5k ● 4 ● 91 ● 118

answered Sep 23, 2008 at 15:33

Dan
974 ● 7 ● 11

5

I dont think there are many full-blown, clean implementations in C++. One try that I like is [Adam Dunkels' protothread library](#).

See also [Protothreads: simplifying event-driven programming of memory-constrained embedded systems](#) in the ACM Digital Library and discussion in Wikipedia topic [Protothread](#),

Share Improve this answer

Follow

edited Oct 21, 2018 at 17:24

Richard Chambers
17.5k ● 4 ● 91 ● 118

answered Sep 23, 2008 at 15:37

yrp
4,575 ● 2 ● 26 ● 10

4

This is an old thread, but I would like to suggest a hack using Duff's device that is not os-dependent (as far as I remember):

[C coroutines using Duff's device](#)

And as an example, here is a telnet library I modified to use coroutines instead of fork/threads: Telnet cli library using coroutines

And since standard C prior to C99 is essentially a true subset of C++, this works well in C++ too.

Share Improve this answer Follow

answered Dec 4, 2015 at 15:48

Erik Alapää
2,673 ● 1 ● 17 ● 25

3

It is based on (cringe) macros, but the following site provides an easy-to-use generator implementation: http://www.codeproject.com/KB/cpp/cpp_generators.aspx

Share Improve this answer Follow

answered Nov 4, 2010 at 13:31

Mark
31 ● 1

3

A new library, **[Boost.Context]**, was released today with portable features for implementing coroutines.

Share Improve this answer Follow

edited Dec 30, 2014 at 2:12

Philipp Claßen
43.8k ● 36 ● 160 ● 247

answered Aug 21, 2012 at 15:34

Jeff Trull
1,323 ● 11 ● 18

2

I've come up with an implementation **without asm** code. The idea is to use the system's thread creating function to initialize stack and context, and use setjmp/longjmp to switch context. But It's not portable, see the tricky pthread version if you are interested.

Share Improve this answer Follow

edited Aug 14, 2014 at 3:34

answered Aug 12, 2014 at 17:34

rox
555 ● 1 ● 7 ● 16

1

https://github.com/tonbit/coroutine is C++11 single .h asymmetric coroutine implementation supporting resume/yield/await primitives and Channel model. It's implementing via ucontext / fiber, not depending on boost, running on linux/windows/macOS. It's a good starting point to learn implementing coroutine in c++.

Share Improve this answer Follow

answered Nov 16, 2016 at 1:58

atto_mu
11 ● 1

1

Based on macros as well (Duff's device, fully portable, see http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html) and inspired by the link posted by Mark, the following emulates co-processes collaborating using events as synchronization mechanism (slightly different model than the traditional co-routines/generator style)

```
// Coprocess.h
#pragma once
#include <vector>

class Coprocess {
  public:
    Coprocess() : line_(0) {}
    void start() { line_ =  0; run(); }
    void end()   { line_ = -1; on_end(); }
    virtual void run() = 0;
    virtual void on_end() {};
  protected:
    int line_;
};
```

```
class Event {
  public:
    Event() : curr_(0) {}

    void wait(Coprocess* p) { waiters_[curr_].push_bac

    void notify() {
        Waiters& old = waiters_[curr_];
        curr_ = 1 - curr_; // move to next ping/pong s
        waiters_[curr_].clear();
        for (Waiters::const_iterator I=old.begin(), E=
            (*I)->run();
    }
  private:
    typedef std::vector<Coprocess*> Waiters;
    int curr_;
    Waiters waiters_[2];
};

#define corun()   run() { switch(line_) { case 0:
#define cowait(e) line_=__LINE__; e.wait(this); return
#define coend     default:; }} void on_end()
```

An example of use:

```
// main.cpp
#include "Coprocess.h"
#include <iostream>

Event e;
long sum=0;

struct Fa : public Coprocess {
    int n, i;
    Fa(int x=1) : n(x) {}
    void corun() {
        std::cout << i << " starts\n";
        for (i=0; ; i+=n) {
            cowait(e);
            sum += i;
        }
    } coend {
            std::cout << n << " ended " << i << std::endl;
        }
};

int main() {
    // create 2 collaborating processes
    Fa f1(5);
    Fa f2(10);

    // start them
    f1.start();
    f2.start();
    for (int k=0; k<=100; k++) {
        e.notify();
    }
    // optional (only if need to restart them)
    f1.end();
    f2.end();

    f1.start(); // coprocesses can be restarted
    std::cout << "sum " << sum << "\n";
    return 0;
}
```

Share Improve this answer

Follow

answered Oct 7, 2019 at 16:12

a_coder

41 ● 3

1

Check out my implementation, it illustrates the asm hacking point, it works on x86, x86-64, aarch32 and aarch64:

https://github.com/user1095108/cr2/

Most of hand-rolled coroutine implementations are variants of the `setjmp/longjmp` pattern or the `ucontext` pattern. Since these work on a variety of architectures, the coroutine implementations themselves are widely portable, you just need to provide some basic assembly code.

Share Improve this answer Follow

edited Apr 9, 2022 at 7:27

answered Jan 5, 2017 at 8:29

user1095108
14.6k ● 10 ● 66 ● 122

0

[WvCont](#) is a part of [WvStreams](#) that implements so-called semi-coroutines. These are a little easier to handle than full-on coroutines: you call into it, and it yields back to the person who called it.

It's implemented using the more flexible WvTask, which supports full-on coroutines; you can find it in the same library.

Works on win32 and Linux, at least, and probably any other Unix system.

Share Improve this answer Follow

answered Sep 23, 2008 at 19:28

apenwarr
11k ● 6 ● 50 ● 59

On POSIX, you can use `makecontext()`/`swapcontext()` processes to logically switch execution context. On Windows you can use the fiber API. Otherwise, all you

0

need is a bit of glue assembly code to switch machine context. I have implemented coroutines with both ASM (for amd64) and with `swapcontext();` is not very difficult.

Share Improve this answer Follow

edited Mar 17, 2023 at 15:35
avariant
2,300 ● 5 ● 27 ● 34

answered Mar 13, 2023 at 4:04
tanhoctin
1 ● 2

0

C++ 20 comes with Coroutine support out of the box.

Libgolang also has golang like channels if you want to use common messaging patterns while using coroutines.

If you are using c++17 and don't want to use boost, you can also use std::transform cleverly

```
std::transform(execution_policy, // par, seq, par_unse
               inVec.begin(), inVec.end(),
               outVec.begin(),
               ElementOperation);
```

And each of your inVec vector elements can be a tuple that's processed by ElementOperation as method params.

Share Improve this answer Follow

edited Jun 11 at 9:29

answered Jun 11 at 9:23

arviman
5,255 ● 45 ● 48

-2

You should always consider using threads instead; especially in modern hardware. If you have work that can be logically separated in Co-routines, using threads means the work might actually be done concurrently, by separate execution units (processor cores).

But, maybe you do want to use coroutines, perhaps because you have an well tested algorithm that has already been written and tested that way, or because you are porting code written that way.

If you work within Windows, you should take a look at fibers. Fibers will give you a coroutine-like framework with support from the OS.

I am not familiar with other OS's to recommend alternatives there.

Share Improve this answer Follow

answered Sep 23, 2008 at 15:58

Euro Micelli
33.9k ● 8 ● 53 ● 72

7 I disagree about blindly favoring threads over fibers. In an optimal system the number of threads trying to run is equal to then number of cores (more if hyper threading). In a solution with LOTS of threads(100s), a thread pool running fibers can be far more efficient. – deft_code Oct 15, 2009 at 14:24

1 Thanks for commenting. However, I didn't say "blindly favor"; I said "always consider". I agree that there are circumstances where fibers or other coroutine methodologies might be more appropriate, but they can be even harder to "get right" than threads -- and that's saying a lot. Basically, I'm suggesting that for most circumstances you should use threads by default, unless you can convince yourself that there are good reasons to go for something else. – Euro Micelli Oct 27, 2009 at 2:25

5 Threading means locking, whereas coroutines are naturally executed in order. Boom half your work is already done for you. Threads are good if you want to compute multiple heavy algorithms in parallel, I can't really think of any other reason to use them. I guess if there's some blocking API that has no non-blocking mode? – Longpoke Nov 23, 2009 at 15:39

2 @Longpoke, I don't agree that half your work is done for you with coroutines. You trade thinking about locks for having to split your algorithm. Different things. Sometimes coroutines are better, sometimes not. That's what i meant by *consider threading*. – Euro Micelli Jul 26, 2010 at 0:26

@Longpoke I don't agree with the OP that you should always use threads. However, threads have one benefit: many "non-blocking" API's in linux still block for actual files, see this reference. I'd love to be proven wrong. There's also this stack overflow answer. That makes the case for threads stronger. There's still the `aio` library in the SO answer. – vitiral Jun 15, 2022 at 15:30

-3

I've tried to implement coroutines myself using C++11 and threads:

```
#include <iostream>
#include <thread>
```

```
class InterruptedException : public std::exception {
};

class AsyncThread {
public:
    AsyncThread() {
        std::unique_lock<std::mutex> lock(mutex);
        thread.reset(new std::thread(std::bind(&AsyncT
        conditionVar.wait(lock); // wait for the threa
    }
    ~AsyncThread() {
        {
            std::lock_guard<std::mutex> _(mutex);
            quit = true;
        }
        conditionVar.notify_all();
        thread->join();
    }
    void run() {
        try {
            yield();
            for (int i = 0; i < 7; ++i) {
                std::cout << i << std::endl;
                yield();
            }
        } catch (InterruptedException& e) {
            return;
        }
        std::lock_guard<std::mutex> lock(mutex);
        quit = true;
        conditionVar.notify_all();
    }
    void yield() {
        std::unique_lock<std::mutex> lock(mutex);
        conditionVar.notify_all();
        conditionVar.wait(lock);
        if (quit) {
            throw InterruptedException();
        }
    }
    void step() {
        std::unique_lock<std::mutex> lock(mutex);
        if (!quit) {
            conditionVar.notify_all();
            conditionVar.wait(lock);
        }
    }
private:
    std::unique_ptr<std::thread> thread;
    std::condition_variable conditionVar;
    std::mutex mutex;
    bool quit = false;
};

int main() {
    AsyncThread asyncThread;
    for (int i = 0; i < 3; ++i) {
        std::cout << "main: " << i << std::endl;
        asyncThread.step();
    }
}
```

Share Improve this answer

Follow

answered May 27, 2012 at 18:10

jhasse

2,583 ● 1 ● 32 ● 43

Isn't this just a producer-consumer implementation?
– André Caron Aug 14, 2012 at 17:00

---

3 You lost me when you said thread. Coroutines shouldn't need threads. – Atif Jun 22, 2016 at 18:50

---

2 Yeah this was a long time ago when I didn't really understand what coroutines were ;) – jhasse Jun 29, 2016 at 12:54