Coroutines in C

by [Simon Tatham]

Introduction

Structuring a large program is always a difficult job. One of the particular problems that often comes up is this: if you have a piece of code producing data, and another piece of code consuming it, which should be the caller and which should be the callee?

Here is a very simple piece of run-length decompression code, and an equally simple piece of parser code:

```
/* Decompression code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (c == 0xFF) {
        len = getchar();
        c = getchar();
        while (len--)
            emit(c);
    } else
        emit(c);
}
emit(EOF);
```

```
/* Parser code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (isalpha(c)) {
        do {
            add_to_token(c);
            c = getchar();
        } while (isalpha(c));
        got_token(WORD);
    }
    add_to_token(c);
    got_token(PUNCT);
}
```

Each of these code fragments is very simple, and easy to read and understand. One produces a character at a time by calling `emit()`; the other consumes a character at a time by calling `getchar()`. If only the calls to `emit()` and the calls to `getchar()` could be made to feed data to each other, it would be simple to connect the two fragments together so that the output from the decompressor went straight to the parser.

In many modern operating systems, you could do this using pipes between two processes or two threads. `emit()` in the decompressor writes to a pipe, and `getchar()` in the parser reads from the other end of the same pipe. Simple and robust, but also heavyweight and not portable. Typically you don't want to have to divide your program into threads for a task this simple.

In this article I offer a creative solution to this sort of structure problem.

Rewriting

The conventional answer is to rewrite one of the ends of the communication channel so that it's a function that can be called. Here's an example of what that might mean for each of the example fragments.

```
int decompressor(void) {
    static int repchar;
    static int replen;
    if (replen > 0) {
        replen--;
        return repchar;
    }
    c = getchar();
    if (c == EOF)
        return EOF;
    if (c == 0xFF) {
        replen = getchar();
        repchar = getchar();
        replen--;
        return repchar;
    } else
        return c;
}
```

```
void parser(int c) {
    static enum {
        START, IN_WORD
    } state;
    switch (state) {
        case IN_WORD:
        if (isalpha(c)) {
            add_to_token(c);
            return;
        }
        got_token(WORD);
        state = START;
        /* fall through */

        case START:
        add_to_token(c);
        if (isalpha(c))
            state = IN_WORD;
        else
            got_token(PUNCT);
        break;
    }
}
```

Of course you don't have to rewrite both of them; just one will do. If you rewrite the decompressor in the form shown, so that it returns one character every time it's called, then the original parser code can replace calls to `getchar()` with calls to `decompressor()`, and the program will be happy. Conversely, if you rewrite the parser in the form shown, so that it is called once for every input character, then the original decompression code can call `parser()` instead of `emit()` with no problems. You would only want to rewrite *both* functions as callees if you were a glutton for punishment.

And that's the point, really. Both these rewritten functions are thoroughly ugly compared to their originals. Both of the processes taking place here are easier to read when written as a caller, not as a callee. Try to deduce the grammar recognised by the parser, or the compressed data format understood by the decompressor, just by reading the code, and you will find that both the originals are clear and both the rewritten forms are less clear. It would be much nicer if we didn't have to turn either piece of code inside out.

Knuth's coroutines

In *The Art of Computer Programming*, Donald Knuth presents a solution to this sort of problem. His answer is to throw away the stack concept completely. Stop thinking of one process as the caller and the other as the callee, and start thinking of them as cooperating equals.

In practical terms: replace the traditional "call" primitive with a slightly different one. The new "call" will save the return value somewhere other than on the stack, and will then jump to a location specified in another saved return value. So each time the decompressor emits another character, it saves its program counter and jumps to the last known location within the parser - and each time the parser *needs* another character, it saves its own program counter and jumps to the location saved by the decompressor. Control shuttles back and forth between the two routines exactly as often as necessary.

This is very nice in theory, but in practice you can only do it in assembly language, because no commonly used high level language supports the coroutine call primitive. Languages like C depend utterly on their stack-based structure, so whenever control passes from any function to any other, one must be the caller and the other must be the callee. So if you want to write portable code, this technique is at least as impractical as the Unix pipe solution.

Stack-based coroutines

So what we would *really* like is the ability to mimic Knuth's coroutine call primitive in C. We must accept that in reality, at the C level, one function will be caller and the other will be callee. In the caller, we have no problem; we code the original algorithm, pretty much exactly as written, and whenever it has (or needs) a character it calls the other function.

The callee has all the problems. For our callee, we want a function which has a "return and continue" operation: return from the function, and next time it is called, resume control from just after the *return* statement. For example, we would like to be able to write a function that says

```
int function(void) {
    int i;
    for (i = 0; i < 10; i++)
        return i;   /* won't work, but wouldn't it be nice */
}
```

and have ten successive calls to the function return the numbers 0 through 9.

How can we implement this? Well, we can transfer control to an arbitrary point in the function using a `goto` statement. So if we use a state variable, we could do this:

```
int function(void) {
    static int i, state = 0;
    switch (state) {
        case 0: goto LABEL0;
        case 1: goto LABEL1;
    }
    LABEL0: /* start of function */
    for (i = 0; i < 10; i++) {
        state = 1; /* so we will come back to LABEL1 */
        return i;
```

```
        LABEL1:; /* resume control straight after the return */
    }
}
```

This method works. We have a set of labels at the points where we might need to resume control: one at the start, and one just after each `return` statement. We have a state variable, preserved between calls to the function, which tells us which label we need to resume control at next. Before any return, we update the state variable to point at the right label; after any call, we do a `switch` on the value of the variable to find out where to jump to.

It's still ugly, though. The worst part of it is that the set of labels must be maintained manually, and must be consistent between the function body and the initial `switch` statement. Every time we add a new return statement, we must invent a new label name and add it to the list in the `switch`; every time we remove a return statement, we must remove its corresponding label. We've just increased our maintenance workload by a factor of two.

Duff's device

The famous "Duff's device" in C makes use of the fact that a `case` statement is still legal within a sub-block of its matching `switch` statement. Tom Duff used this for an optimised output loop:

```
    switch (count % 8) {
        case 0:        do {  *to = *from++;
        case 7:              *to = *from++;
        case 6:              *to = *from++;
        case 5:              *to = *from++;
        case 4:              *to = *from++;
        case 3:              *to = *from++;
        case 2:              *to = *from++;
        case 1:              *to = *from++;
                    } while ((count -= 8) > 0);
    }
```

We can put it to a slightly different use in the coroutine trick. Instead of using a `switch` statement to decide which `goto` statement to execute, we can use the `switch` statement to perform the jump itself:

```
int function(void) {
    static int i, state = 0;
    switch (state) {
        case 0: /* start of function */
        for (i = 0; i < 10; i++) {
            state = 1; /* so we will come back to "case 1" */
            return i;
            case 1:; /* resume control straight after the return */
        }
```

```
    }
}
```

Now this is looking promising. All we have to do now is construct a few well chosen macros, and we can hide the gory details in something plausible-looking:

```
#define crBegin static int state=0; switch(state) { case 0:
#define crReturn(i,x) do { state=i; return x; case i:; } while (0)
#define crFinish }
int function(void) {
    static int i;
    crBegin;
    for (i = 0; i < 10; i++)
        crReturn(1, i);
    crFinish;
}
```

(note the use of `do ... while(0)` to ensure that `crReturn` does not need braces around it when it comes directly between `if` and `else`)

This is almost exactly what we wanted. We can use `crReturn` to return from the function in such a way that control at the next call resumes just after the return. Of course we must obey some ground rules (surround the function body with `crBegin` and `crFinish`; declare all local variables `static` if they need to be preserved across a `crReturn`; *never* put a `crReturn` within an explicit `switch` statement); but those do not limit us very much.

The only snag remaining is the first parameter to `crReturn`. Just as when we invented a new label in the previous section we had to avoid it colliding with existing label names, now we must ensure all our state parameters to `crReturn` are different. The consequences will be fairly benign - the compiler will catch it and not let it do horrible things at run time - but we still need to avoid doing it.

Even this can be solved. ANSI C provides the special macro name `__LINE__`, which expands to the current source line number. So we can rewrite `crReturn` as

```
#define crReturn(x) do { state=__LINE__; return x; \
                         case __LINE__:; } while (0)
```

and then we no longer have to worry about those state parameters at all, provided we obey a fourth ground rule (never put two `crReturn` statements on the same line).

Evaluation

So now we have this monstrosity, let's rewrite our original code fragments using it.

```
int decompressor(void) {
    static int c, len;
    crBegin;
    while (1) {
        c = getchar();
        if (c == EOF)
            break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--)
                crReturn(c);
        } else
            crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}
```

```
void parser(int c) {
    crBegin;
    while (1) {
        /* first char already in c */
        if (c == EOF)
            break;
        if (isalpha(c)) {
            do {
                add_to_token(c);
                crReturn( );
            } while (isalpha(c));
            got_token(WORD);
        }
        add_to_token(c);
        got_token(PUNCT);
        crReturn( );
    }
    crFinish;
}
```

We have rewritten both decompressor and parser as callees, with no need at all for the massive restructuring we had to do last time we did this. The structure of each function exactly mirrors the structure of its original form. A reader can deduce the grammar recognised by the parser, or the compressed data format used by the decompressor, far more easily than by reading the obscure state-machine code. The control flow is intuitive once you have wrapped your mind around the new format: when the decompressor has a character, it passes it back to the caller with `crReturn` and waits to be called again when another character is required. When the parser needs another character, it returns using `crReturn`, and waits to be called again with the new character in the parameter `c`.

There has been one small structural alteration to the code: `parser()` now has its `getchar()` (well, the corresponding `crReturn`) at the end of the loop instead of the start, because the first character is already in `c` when the function is entered. We could accept this small change in structure, or if we really felt strongly about it we could specify that `parser()` required an "initialisation" call before you could start feeding it characters.

As before, of course, we don't have to rewrite both routines using the coroutine macros. One will suffice; the other can be its caller.

We have achieved what we set out to achieve: a portable ANSI C means of passing data between a producer and a consumer without the need to rewrite one as an explicit state machine. We have done this by combining the C preprocessor with a little-used feature of the `switch` statement to create an *implicit* state machine.

Coding Standards

Of course, this trick violates every coding standard in the book. Try doing this in your company's code and you will probably be subject to a stern

telling off if not disciplinary action! You have embedded unmatched braces in macros, used `case` within sub-blocks, and as for the `crReturn` macro with its terrifyingly disruptive contents . . . It's a wonder you haven't been fired on the spot for such irresponsible coding practice. You should be ashamed of yourself.

I would claim that the coding standards are at fault here. The examples I've shown in this article are not very long, not very complicated, and still just about comprehensible when rewritten as state machines. But as the functions get longer, the degree of rewriting required becomes greater and the loss of clarity becomes much, much worse.

Consider. A function built of small blocks of the form

```
case STATE1:
/* perform some activity */
if (condition) state = STATE2; else state = STATE3;
```

is not very different, to a reader, from a function built of small blocks of the form

```
LABEL1:
/* perform some activity */
if (condition) goto LABEL2; else goto LABEL3;
```

One is caller and the other is callee, true, but the visual structure of the functions are the same, and the insights they provide into their underlying algorithms are exactly as small as each other. The same people who would fire you for using my coroutine macros would fire you just as loudly for building a function out of small blocks connected by `goto` statements! And this time they would be right, because laying out a function like that obscures the structure of the algorithm horribly.

Coding standards aim for clarity. By hiding vital things like `switch`, `return` and `case` statements inside "obfuscating" macros, the coding standards would claim you have obscured the syntactic structure of the program, and violated the requirement for clarity. But you have done so in the cause of revealing the *algorithmic* structure of the program, which is far more likely to be what the reader wants to know!

Any coding standard which insists on syntactic clarity at the expense of algorithmic clarity should be rewritten. If your employer fires you for using this trick, tell them that repeatedly as the security staff drag you out of the building.

Refinements and Code

In a serious application, this toy coroutine implementation is unlikely to be useful, because it relies on `static` variables and so it fails to be re-entrant or multi-threadable. Ideally, in a real application, you would want to be able to call the same function in several different contexts, and at each call in a given context, have control resume just after the last return in the same context.

This is easily enough done. We arrange an extra function parameter, which is a pointer to a context structure; we declare all our local state, and our

coroutine state variable, as elements of that structure.

It's a little bit ugly, because suddenly you have to use `ctx->i` as a loop counter where you would previously just have used `i`; virtually all your serious variables become elements of the coroutine context structure. But it removes the problems with re-entrancy, and still hasn't impacted the *structure* of the routine.

(Of course, if C only had Pascal's `with` statement, we could arrange for the macros to make this layer of indirection truly transparent as well. A pity. Still, at least C++ users can manage this by having their coroutine be a class member, and keeping all its local variables in the class so that the scoping is implicit.)

Included here is a C header file that implements this coroutine trick as a set of pre-defined macros. There are two sets of macros defined in the file, prefixed `scr` and `ccr`. The `scr` macros are the simple form of the technique, for when you can get away with using `static` variables; the `ccr` macros provide the advanced re-entrant form. Full documentation is given in a comment in the header file itself.

Note that Visual C++ version 6 doesn't like this coroutine trick, because its default debug state (Program Database for Edit and Continue) does something strange to the `__LINE__` macro. To compile a coroutine-using program with VC++ 6, you must turn off Edit and Continue. (In the project settings, go to the "C/C++" tab, category "General", setting "Debug info". Select any option *other* than "Program Database for Edit and Continue".)

(The header file is MIT-licensed, so you can use it in anything you like without restriction. If you do find something the MIT licence doesn't permit you to do, mail me, and I'll probably give you explicit permission to do it anyway.)

Follow this link for `coroutine.h`.

Thanks for reading. Share and enjoy!

References

- Donald Knuth, *The Art of Computer Programming*, Volume 1. Addison-Wesley, ISBN 0-201-89683-4. Section 1.4.2 describes coroutines in the "pure" form.
- http://www.lysator.liu.se/c/duffs-device.html is Tom Duff's own discussion of Duff's device. Note, right at the bottom, a hint that Duff might also have independently invented this coroutine trick or something very like it.

  Update, 2005-03-07: Tom Duff confirms this in a blog comment. The "revolting way to use switches to implement interrupt driven state machines" of which he speaks in his original email is indeed the same trick as I describe here.

- PuTTY is a Win32 Telnet and SSH client. The SSH protocol code contains real-life use of this coroutine trick. As far as I know, this is the worst piece of C hackery ever seen in serious production code.

Id

Please send comments and criticism to anakin@pobox.com.