Coroutines in less than 20 lines of standard C - Tony Finch's blog — LiveJournal

Coroutines in less than 20 lines of standard C

« previous entry | next entry »
22nd Jan 2010 | 19:42

Here's a bit of evil hackery to brighten your weekend.

It's quite simple to switch between coroutines using standard C: use `setjmp()` to save the state of the current coroutine, and `longjmp()` to jump to the target coroutine. It's also useful to be able to transfer a bit of data while transferring control. The function `coto()` (short for "coroutine goto") does this. First it saves its argument somewhere the target coroutine can find it, then it calls `setjmp()` which saves the current coroutine's state and returns 0, so it goes on to `longjmp()` to the target coroutine, where `setjmp()` returns 1 so `coto()` returns the saved argument.

```
static void *coarg;

void *coto(jmp_buf here, jmp_buf there, void *arg) {
    coarg = arg;
    if (setjmp(here)) return(coarg);
    longjmp(there, 1);
}
```

The other thing you need to be able to do is create coroutines. This requires allocating space for the coroutine's stack, initializing a stack frame there, and jumping to it. This is usually considered impossible to do without special support from the operating system or the run-time system - for example, see this paper (postscript). However if we can assume the stack is laid out contiguously in memory and that variable length arrays are allocated on the stack, we can do it using pure standard C99. (If your compiler doesn't support variable length arrays, `alloca()` might be available as an alternative.)

Each coroutine's stack is a chunk of the memory area used for the normal process stack. To allocate a new chunk of stack, we add the chunk size to the current top of stack (which allows space for future function calls by the current topmost coroutine) to give us a target location for the new topmost coroutine's stack. We then need to move the stack pointer to this new location, which we can do by creating a variable length array of the appropriate length. We can use the address of a local variable to approximate the current stack pointer when calculating this length. Finally, a simple function call creates the coroutine's initial stack frame and jumps to it.

The function `cogo()` (short for "coroutine start") implements this. It also saves the current coroutine's state before starting the new one.

```
#define STACKDIR - // set to + for upwards and - for downwards
#define STACKSIZE (1<<12)

static char *tos; // top of stack

void *cogo(jmp_buf here, void (*fun)(void*), void *arg) {
    if (tos == NULL) tos = (char*)&arg;
    tos += STACKDIR STACKSIZE;
    char n[STACKDIR (tos - (char*)&arg)];
    coarg = n; // ensure optimizer keeps n
    if (setjmp(here)) return(coarg);
    fun(arg);
    abort();
}
```

Here's a little test program to demonstrate these functions in action.

```
#define MAXTHREAD 10000

static jmp_buf thread[MAXTHREAD];
static int count = 0;

static void comain(void *arg) {
    int *p = arg, i = *p;
    for (;;) {
            printf("coroutine %d at %p arg %p\n", i, (void*)&i, arg);
            int n = arc4random() % count;
            printf("jumping to %d\n", n);
```

```
                arg = coto(thread[i], thread[n], (char*)arg + 1);
        }
    }

    int main(void) {
        while (++count < MAXTHREAD) {
                printf("spawning %d\n", count);
                cogo(thread[0], comain, &count);
        }
        return 0;
    }
```

Permalink | Leave a comment |

---

from: ~~pauamma~~
date: 23rd Jan 2010 17:43 (UTC)
Permalink

Nice one. Another one (which doesn't make any assumptions on stack layout) is Simon Tatham's.

Reply | Thread

---

from: fanf
date: 23rd Jan 2010 22:14 (UTC)
Permalink

Yes, I should have linked to Simon's hack since it's quite well known. In fact the stack hack above came out of a discussion in the pub between me, Ian Jackson, and Simon Tatham. The advantage of the stack hack is you can jump between coroutines from nested function calls, whereas Simon's `case __LINE__` hack only allows the topmost function to yield.

I should also note that Eric Freudenthal has also implemented the stack hack, though not so concisely. He even inflicts it on his students as part of a course on operating systems!
http://robust.cs.utep.edu/freudent/courses/osWeb/labs/tthreads/

Reply | Parent | Thread

---

from: ~~pauamma~~
date: 24th Jan 2010 13:35 (UTC)
Permalink

> The advantage of the stack hack is you can jump between coroutines from nested function calls, whereas Simon's case __LINE__ hack only allows the topmost function to yield.

Hmm, yes. OTOH, unless I misread the code badly (which is possible - my C is slightly rusty and threading libraries make my head hurt), neither your implementation nor Eric Freudenthal's allow returning a coroutine from a function (since the coroutine's stack is deallocated upon return from cogo or , which makes it hard to create generator functions or lazy evaluators. I wonder whether that could be fixed easily. Hmm, I can't think of one that doesn't involve using the heal to store the coroutine's stack, which doesn't feel very portable to me.

Reply | Parent | Thread

---

from: fanf
date: 24th Jan 2010 19:48 (UTC)
Permalink

Forget your normal model of stack allocation :-)

When a function returns, its frame is deallocated only in the sense that it is liable to being overwritten - the memory isn't given back to the OS. So if we make sure we don't overwrite it, the frame can hang around as long as we want.

The purpose of the array `n` in `cogo()` is to provide space for the current stack to grow, without overwriting the stack frame created by `cogo()`'s call to `fun(arg)`. Actually, it does a bit more than that because if the current coroutine is not the topmost chunk on the stack, the array has to span all the more recently created coroutines' stack chunks.

For a slightly more developed coroutine library (50ish lines of code and 150ish lines of comments) have a look at http://dotat.at/prog/picoro/. It has its own data type and a constructor function and everything!

Edited at 2010-01-24 07:49 pm (UTC)

Reply | Parent | Thread

(Deleted comment)

---

from: fanf
date: 1st Feb 2010 12:20 (UTC)
Permalink

Yes, but that requires system calls, and probably ends up being much more long-winded :-)

(I know that on BSD, setjmp and longjmp make system calls to set the signal mask, so one should use _setjmp and _longjmp instead.)

You don't have to put all the coroutine stacks in chunks of the main stack: you can instead malloc() them, and use the same stack pointer arithmetic trick (either a variable length array or a call to alloca) to make the first switch to the new stack :-)

Reply | Parent | Thread

---

from: *anonymous*
date: 1st Jul 2015 14:26 (UTC)
Permalink

“and use the same stack pointer arithmetic trick (either a variable length array or a call to alloca) to make the first switch to the new stack”

Hey,
I picked up your picoro lib, its awesome! Im trying to lift stack size limits by switching stack to malloc(). Could you elaborate on this? Code is complex enough so i easily loose track of what should be happening there. I tried obvious thing of replacing stack array with dynamically allocated chunk of memory but something is missing and im not entirely sure what. Besides there probably should be some stack switching back? Also could you give me some hints on stack pointer manipulation tricks? All i know is unportable inline asm.

Reply | Parent | Thread

---

from: fanf
date: 1st Jul 2015 15:44 (UTC)
Permalink

Moving the stack pointer into a malloced area is tricky :-)

Look at coroutine1() in http://dotat.at/cgi/git?p=picoro.git;a=blob;f=picoro.c;hb=smaller

```
void coroutine1(void) {
    char dummy[16*1024];
    coroutine2(dummy);
}
```

The dummy array saves some stack for the parent coroutine and moves the stack pointer to the child coroutine.

To use malloc() you need something like this (untested)

```
char *newstack = malloc(STACKSIZE);
char *newtop = newstack + STACKSIZE;
char *mytop = (char*)&mytop;
/* alloca(n) is equivalent to sp -= n if your stack grows down as usual */
alloca(mytop - newtop);
```

Reply | Parent | Thread

---

from: *anonymous*
date: 5th Jul 2015 10:56 (UTC)
Permalink

Thank you, that works.

Reply | Parent | Thread

Question on Stack Allocation

from: *anonymous*
date: 24th Feb 2015 17:42 (UTC)
Permalink

Did you set the Stack size based on coto argument size?

Reply | Thread

Re: Question on Stack Allocation

from: fanf
date: 24th Feb 2015 18:27 (UTC)
Permalink

No, the stack chunk size was just a guess. 4KB is about the smallest plausible choice, though it does not allow much call depth or local data. And if you overflow it the crashes will be even harder to debug than usual!

Reply | Parent | Thread

Leave a comment