dotat.at / git / picoro.git / blob

Remove C99 bool.

[picoro.git] / picoro.c

```
 1 /*
 2  * picoro - minimal coroutines for C.
 3  * Written by Tony Finch <dot@dotat.at>
 4  * http://creativecommons.org/publicdomain/zero/1.0/
 5  */
 6
 7 #include <assert.h>
 8 #include <setjmp.h>
 9 #include <stdlib.h>
10
11 #include "picoro.h"
12
13 /*
14  * Each coroutine has a jmp_buf to hold its context when suspended.
15  *
16  * There are lists of running and idle coroutines.
17  *
18  * The coroutine at the head of the running list has the CPU, and all
19  * others are suspended inside resume(). The "first" coro object holds
20  * the context for the program's initial stack and also ensures that
21  * all externally-visible list elements have non-NULL next pointers.
22  * (The "first" coroutine isn't exposed to the caller.)
23  *
24  * The idle list contains coroutines that are suspended in
25  * coroutine_main(). After initialization it is never NULL except
26  * briefly while coroutine_main() forks a new idle coroutine.
27  */
28 static struct coro {
29         struct coro *next;
30         jmp_buf state;
31 } first, *running = &first, *idle;
32
33 /*
34  * A coroutine can be passed to resume() if
35  * it is not on the running or idle lists.
36  */
37 int resumable(coro c) {
38         return(c != NULL && c->next == NULL);
39 }
40
41 /*
42  * Add a coroutine to a list and return the previous head of the list.
43  */
44 static void push(coro *list, coro c) {
45         c->next = *list;
46         *list = c;
47 }
48
49 /*
50  * Remove a coroutine from a list and return it.
51  */
52 static coro pop(coro *list) {
53         coro c = *list;
54         *list = c->next;
55         c->next = NULL;
56         return(c);
57 }
58
59 /*
60  * Pass a value and control from one coroutine to another.
61  * The current coroutine's state is saved in "me" and the
62  * target coroutine is at the head of the "running" list.
```

```
 */
static void *pass(coro me, void *arg) {
        static void *saved;
        saved = arg;
        if(!setjmp(me->state))
                longjmp(running->state, 1);
        return(saved);
}

void *resume(coro c, void *arg) {
        assert(resumable(c));
        push(&running, c);
        return(pass(c->next, arg));
}

void *yield(void *arg) {
        return(pass(pop(&running), arg));
}

/* Declare for mutual recursion. */
void coroutine_start(void), coroutine_main(void*);

/*
 * The coroutine constructor function.
 *
 * On the first invocation there are no idle coroutines, so fork the
 * first one, which will immediately yield back to us after becoming
 * idle. When there are idle coroutines, we pass one the function
 * pointer and return the activated coroutine's address.
 */
coro coroutine(void *fun(void *arg)) {
        if(idle == NULL && !setjmp(running->state))
                coroutine_start();
        return(resume(pop(&idle), fun));
}

/*
 * The main loop for a coroutine is responsible for managing the "idle" list.
 *
 * When we start the idle list is empty, so we put ourself on it to
 * ensure it remains non-NULL. Then we immediately suspend ourself
 * waiting for the first function we are to run. (The head of the
 * running list is the coroutine that forked us.) We pass the stack
 * pointer to prevent it from being optimised away. The first time we
 * are called we will return to the fork in the coroutine()
 * constructor function (above); on subsequent calls we will resume
 * the parent coroutine_main(). In both cases the passed value is
 * lost when pass() longjmp()s to the forking setjmp().
 *
 * When we are resumed, the idle list is empty again, so we fork
 * another coroutine. When the child coroutine_main() passes control
 * back to us, we drop into our main loop.
 *
 * We are now head of the running list with a function to call. We
 * immediately yield a pointer to our context object so our creator
 * can identify us. The creator can then resume us at which point we
 * pass the argument to the function to start executing.
 *
 * When the function returns, we move ourself from the running list to
 * the idle list, before passing the result back to the resumer. (This
 * is just like yield() except for adding the coroutine to the idle
 * list.) We can then only be resumed by the coroutine() constructor
 * function which will put us back on the running list and pass us a
 * new function to call.
 *
 * We do not declare coroutine_main() static to try to stop it being inlined.
 *
 * The conversion between the function pointer and a void pointer is not
 * allowed by ANSI C but we do it anyway.
 */
void coroutine_main(void *ret) {
        void *(*fun)(void *arg);
        struct coro me;
        push(&idle, &me);
```

```
137         fun = pass(&me, ret);
138         if(!setjmp(running->state))
139                 coroutine_start();
140         for(;;) {
141                 ret = fun(yield(&me));
142                 push(&idle, pop(&running));
143                 fun = pass(&me, ret);
144         }
145 }
146
147 /*
148  * Allocate space for the current stack to grow before creating the
149  * initial stack frame for the next coroutine.
150  */
151 void coroutine_start(void) {
152         char stack[16 * 1024];
153         coroutine_main(stack);
154 }
155
156 /* eof */
```

Various little pure C coroutine implementations.

Atom RSS