NAME

co_thread_init, co_thread_cleanup, co_create, co_call, co_resume, co_delete, co_exit_to, co_exit, co_current, co_get_data, co_set_data – C coroutine management

SYNOPSIS

#include <pcl.h>

int co_thread_init(void);

void co_thread_cleanup(void);
coroutine_t co_create(void **func*, **void ****data*, **void ****stack*, **int** *stacksize***);**
void co_delete(coroutine_t *co***);**
void co_call(coroutine_t *co***);**
void co_resume(void);
void co_exit_to(coroutine_t *co***);**
void co_exit(void);
coroutine_t co_current(void);
void *co_get_data(coroutine_t *co***);**
void *co_set_data(coroutine_t *co*, **void ****data***);**

Link with *-lpthread* if you are using a multi-thread version of **PCL**.

DESCRIPTION

The **Portable Coroutine Library (PCL)** implements the low level functionality for coroutines. For a definition of the term **coroutine** see *The Art of Computer Programming* by *Donald E. Knuth*. Coroutines are a very simple cooperative multitasking environment where the switch from one task to another is done explicitly by a function call. Coroutines are a lot faster than processes or threads switch, since there is no OS kernel involvement for the operation. This document defines an API for the low level handling of coroutines i.e. creating and deleting coroutines and switching between them. Higher level functionality (scheduler, etc.) is not covered.

Functions

The following functions are defined:

int co_thread_init(void);

If the **PCL** library is built in multi-thread mode, and if multi threads are actually used, this function should be called before calling any **PCL** function. If the **PCL** library is built in multi-thread mode, but it is used only from one thread (the main one, likely), then it is possible to avoid to call **co_thread_init**(). Returns 0 in case of success, or an negative error code in case of error.

void co_thread_cleanup(void);

If the **PCL** library is built in multi-thread mode, and if multi threads are actually used, this function should be called before the thread exits, or whenever the thread decides it won't call the **PCL** functions anymore. A failure in calling **co_thread_cleanup**() will result in resource leakage by the calling application.

coroutine_t co_create(void **func*, **void ****data*, **void ****stack*, **int** *stacksize***);**

This function creates a new coroutine. *func* is the entry point of the coroutine. It will be called with one arg, a **void ***, which holds the data passed through the *data* parameter. If *func* terminates, the associated coroutine is deleted. *stack* is the base of the stack this coroutine will use and *stacksize* its size in bytes. You may pass a **NULL** pointer for *stack* in which case the memory will be allocated by **co_create** itself. Both, *stack* and *stacksize* are aligned to system requirements. A *stacksize* of less then 4096 bytes will be rejected. You have to make sure, that the stack is large enough for your coroutine and possible signal handlers (see below). The stack will not grow! (Exception: the main coroutine uses the standard system stack which may still grow) On success, a handle (**coroutine_t**) for a new coroutine is returned, otherwise **NULL**.

void co_delete(coroutine_t *co***);**

This function deletes the given coroutine *co*. If the stack for this coroutine was allocated by **co_create** it will be freed. After a coroutine handle was passed to **co_delete** it is invalid and may not be used any more. It is invalid for a coroutine to delete itself with this function.

void co_call(coroutine_t *co***);**

This function passes execution to the given coroutine *co*. The first time the coroutine is executed, its entry point *func* is called, and the *data* parameter used during the call to **co_create** is passed to *func*. The current coroutine is suspended until another one restarts it with a **co_call** or **co_resume** call. Calling oneself returns immediately.

void co_resume(void);

This function passes execution back to the coroutine which either initially started this one or restarted it after a prior **co_resume**.

void co_exit_to(coroutine_t *co***);**

This function does the same a **co_delete(co_current**()) followed by a **co_call** would do. That is, it deletes itself and then passes execution to another coroutine *co*.

void co_exit(void);

This function does the same a **co_delete(co_current**()) followed by a **co_resume** would do. That is, it deletes itself and then passes execution back to the coroutine which either initially started this one or restarted it after a prior **co_resume**.

coroutine_t co_current(void);

This function returns the currently running coroutine.

void *co_get_data(coroutine_t *co***);**

This function returns the data associated with the *co*
coroutine. The data associated with a coroutine is the *data* parameter passed to **co_create**().

void *co_set_data(coroutine_t *co***, void ***data***);**

Sets the *data* associated with the *co* coroutine, and returns the previously associated data.

Notes

Some interactions with other parts of the system are covered here.

Threads

If the **PCL** library has been built in multi-thread mode, then it is possible to use it in multi-thread software. A thread should call **co_thread_init**() before using the **PCL** APIs, and call **co_thread_cleanup**() before exiting, or when it has done using the **PCL** APIs.
WARNING: For no reason should two different threads run the same coroutine at the same time.

Signals First, a signal handler is not defined to run in any specific coroutine. The only way to leave the signal handler is by a return statement.

Second, the signal handler may run with the stack of any coroutine, even with the stack of library internal coroutines which have an undefined stack size (just enough to perform a kernel call). Using and alternate stack for signal processing (see **sigaltstack**(2)) is recommended!

Conclusion: avoid signals like a plague. The only thing you may do reliable is setting some global variables and return. Simple kernel calls may work too, but nowadays it's pretty hairy to tell, which function really is a kernel call. (Btw, all this applies to normal C programs, too. The coroutines just add one more problem)

setjmp/longjmp

The use of **setjmp**(2)/**longjmp**(2) is limited to jumping inside one coroutine. Never try to jump from one coroutine to another with **longjmp**(2).

DIAGNOSTICS

Some fatal errors are caught by the library. If one occurs, a short message is written to file descriptor 2 (stderr) and a segmentation violation is generated.

[PCL]: Cannot delete itself

A coroutine has called **co_delete** with it's own handle.

[PCL]: Resume to deleted coroutine

A coroutine has deleted itself with **co_exit** or **co_exit_to** and the coroutine that was activated by the exit tried a **co_resume**.

[PCL]: Stale coroutine called

Someone tried to active a coroutine that has already been deleted. This error is only detected, if the stack of the deleted coroutine is still resident in memory.

[PCL]: Context switch failed

Low level error generated by the library in case a context switch between two coroutines failes.

SEE ALSO

Original **coroutine** library at **http://www.goron.de/˜froese/coro/coro.html** . GNU Pth library at **http://www.gnu.org/software/pth/** .

AUTHOR

Developed by Davide Libenzi < **davidel@xmailserver.org** >.
Ideas and man page base source taken by the coroutine library developed by E. Toernig < **froese@gmx.de** >.
Also some code and ideas comes from the GNU Pth library available at **http://www.gnu.org/software/pth/** .

BUGS

There are no known bugs. But, this library is still in development even if it results very stable and pretty much ready for production use.

Bug reports and comments to Davide Libenzi < **davidel@xmailserver.org** >.