Threads for the C Standard Library

WG14 Document: N1325
Date: 2008-08-02

Introduction

This document is a proposal for an approach to add threads to the C Standard library. As discussed in the WG14 meeting held in Delft in April of 2008. A thread in this document is a separate flow of execution within an application. On a multi-processor system threads can execute simultaneously on different processors. On a single-processor system and on a multi-processor system with fewer available processors than active threads two or more threads must share a processor. The details of switching a processor from one thread to another are handled by the operating system and are not covered in this document.

CONTENTS

FUNCTIONS

The `call_once` function

Synopsis

```
void call_once(once_flag *flag, void (*func)(void));
```

Description

The **`call_once`** function uses the **`once_flag`** pointed to by **`flag`** to ensure that **`func`** is called exactly once, the first time **`call_once`** is called with that value of **`flag`**.

Returns

The **`call_once`** function returns no value.

The `cnd_broadcast` function

Synopsis

```
int cnd_broadcast(cnd_t *cond);
```

Description

The **`cnd_broadcast`** function unblocks all of the threads that are blocked on the condition variable pointed to by **`cond`** at the time of the call. If no threads are blocked on the condition variable pointed to by **`cond`** at the time of the call, the function does nothing.

Returns

The **`cnd_broacast`** function returns:

- **`thrd_success`** – on success, or
- **`thrd_error`** – when the request could not be honored.

The `cnd_destroy` function

Synopsis

```
void cnd_destroy(cnd_t *cond);
```

Description

The **`cnd_destroy`** function releases all resources used by the condition variable pointed to by **`cond`**. The **`cnd_destroy`** function requires that no threads be blocked waiting for the condition variable pointed to by **`cond`**.

Returns

The **`cnd_destroy`** function returns no value.

The `cnd_init` function

Synopsis

```
int cnd_init(cnd_t *cond);
```

Description

The **`cnd_init`** function creates a condition variable. If it succeeds it sets the variable pointed to by **`cond`** to a value that uniquely identifies the newly created condition variable. A thread that calls **`cnd_wait`** on a newly created condition variable will block.

Returns

The **`cnd_init`** functions returns:

- **`thrd_success`** – on success, or
- **`thrd_nomem`** – no memory could be allocated for the newly created condition, or
- **`thrd_error`** – when the request could not be honored.

The `cnd_signal` function

Synopsis

```
int cnd_signal(cnd_t *cond);
```

Description

The **`cnd_signal`** function unblocks one of the threads that are blocked on the condition variable pointed to by **`cond`** at the time of the call. If no threads are blocked on the condition variable at the time of the call, the function does nothing and return success.

Returns

The **cnd_signal** function returns:

- **thrd_success** – on success or
- **thrd_error** – when request could not be honored.

The cnd_timedwait function

Synopsis

```
int cnd_timedwait(cnd_t *cond,
     mtx_t *mtx,
     const xtime *xt);
```

Description

The **cnd_timedwait** function atomically unlocks the **mutex mtx** and blocks until the condition variable pointed to by **cond** is signaled by a call to **cnd_signal** or to **cnd_broadcast**, or until after the time specified by the **xtime** object pointed to by **xt**. When the calling thread becomes unblocked it locks the variable pointed to by **mtx** before it returns. The **cnd_timedwait** function requires that the **mutex** pointed to by **mtx** be locked by the calling thread.

Returns

The **cnd_timedwait** function returns:

- **thrd_success** – upon success, or
- **thrd_timeout** – if time specified in the call was reached without acquiring the requested resource, or
- **thrd_error** – when the request could not be honored.

The cnd_wait function

Synopsis

```
int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

Description

The function atomically unlocks the mutex pointed to by **mtx** and blocks until the condition variable pointed to by **cond** is signaled by a call to **cnd_signal** or to **cnd_broadcast**. When the calling thread becomes unblocked it locks the mutex pointed to by **`mtx`** before it returns. If the mutex pointed to by **`mtx`** is not locked by the calling thread, the function **`cnd_wait`** will act as if the function **`abort()`** is called.

Returns

The **`cnd_wait`** function returns:

- **`thrd_success`** – on success or
- **`thrd_error`** – when the request could not be honored.

The `mtx_destroy` function

Synopsis

```
void mtx_destroy(mtx_t *mtx);
```

Description

The **`mtx_destroy`** function releases any resources used by the mutex pointed to by **`mtx`**. No threads can be blocked waiting for the mutex pointed to by **`mtx`**.

Returns

The **`mtx_destroy`** function returns no value.

The `mtx_init` function

Synopsis

```
int mtx_init(mtx_t *mtx, int type);
```

Description

The function creates a mutex object with properties indicated by **`type`**, which must have one of the six values:

- **`mtx_plain`** — for a simple non-recursive mutex
- **`mtx_timed`** — for a non-recursive mutex that supports timeout
- **`mtx_try`** — for a non-recursive mutex that supports test and return
- **`mtx_plain`** | **`mtx_recursive`** — for a simple recursive mutex
- **`mtx_timed`** | **`mtx_recursive`** — for a recursive mutex that supports timeout
- **`mtx_try`** | **`mtx_recursive`** — for a recursive mutex that supports test and return

If **mtx_init** function succeeds it sets the **mtx_t** pointed to by **mtx** to a value that uniquely identifies the newly created mutex.

Returns

The **mtx_init** function returns:

- **thrd_success** – on success, or
- **thrd_error** – when request could not be honored.

The mtx_lock function

Synopsis

```
int mtx_lock(mtx_t *mtx);
```

Description

The function blocks until it locks the mutex pointed to by **mtx**. If the mutex is non-recursive it shall not be locked by the calling thread.

Returns

The **mtx_lock** function returns:

- **thrd_success** – on success, or
- **thrd_busy** – resource requested is already in use, or
- **thrd_error** – when the request could not be honored.

The mtx_timedlock function

Synopsis

```
int mtx_timedlock(mtx_t *mtx, const xtime *xt);
```

Description

The **mtx_timedlock** function blocks until it locks the mutex pointed to by **mtx** or until the time specified by the xtime object **xt** has passed. The mutex pointed to by **mtx** shall be of type:

- **mtx_timed** or
- **mtx_timed** | **mtx_recursive**.

Returns

The **mtx_timedlock** function returns:

- **thrd_success** – on success, or

- **thrd_busy** – resource requested is already in use, or
- **thrd_timeout** – if time specified was reached without aquiring the requested resource, or
- **thrd_error** – when the request could not be honored.

The `mtx_trylock` function

Synopsis

```
int mtx_trylock(mtx_t *mtx);
```

Description

The **mtx_trylock** function attempts to lock the mutex pointed to by **mtx**. If the mutex is already locked the function returns without blocking. The mutex pointed to by **mtx** shall be of type:

- **mtx_try**, or
- **mtx_try** | **mtx_recursive**, or
- **mtx_timed**, or
- **tmx_timed** | **mtx_recursive**.

Returns

The **mtx_trylock** function returns:

- **thrd_success** – on success, or
- **thrd_busy** – resources requested is already in use, or
- **thrd_error** – when the request could no be honored.

The `mtx_unlock` function

Synopsis

```
int mtx_unlock(mtx_t *mtx);
```

Description

The **mtx_unlock** function unlocks the mutex pointed to by **mtx**. The mutex pointed to by **mtx** shall be locked by the calling thread.

Returns

The **mtx_unlock** function returns:

- **thrd_success** – on success or
- **thrd_error** – when the request could no be honored.

The `thrd_abort` function

Synopsis

```
void thrd_abort(const char *msg);
```

Description

The **`thrd_abort`** function writes the characters pointed to by **`msg`** to the standard error then calls **`abort()`**.

Returns

The **`thrd_abort`** function returns no value.

The `thrd_create` function

Synopsis

```
int thrd_create(thrd_t *thr, thrd_start_t func,
     void *arg);
```

Description

The **`thrd_create`** function creates a new thread executing **`func(arg)`**. If the **`thrd_create`** function succeeds it sets the thread **`thr`** to a value that uniquely identifies the newly created thread. The function does not return until the new thread has begun execution.

Returns

The **`thrd_create`** functions returns:

- **`thrd_success`** – on success, or
- **`thrd_nomem`** – no memory could be allocated for the thread requested, or
- **`thrd_error`** – when request could not be honored.

The `thrd_current` function

Synopsis

```
thrd_t thrd_current(void);
```

Description

The **`thrd_current`** function identifies the thread that called it.

Returns

The **`thrd_current`** function returns a value that uniquely identifies the thread that called it.

The `thrd_detach` function

Synopsis

```
int thrd_detach(thrd_t thr);
```

Description

The **`thrd_detach`** function tells the operating system to dispose of any resources allocated to the thread identified by **`thr`** when that thread terminates. The value of the thread identified by **`thr`** value shall not have been set by a call to **`thrd_join`** or **`thrd_detach`**.

Returns

The **`thrd_detach`** function returns:

- **`thrd_success`** – on success or
- **`thrd_error`** – when the request could no be honored.

The `thrd_equal` function

Synopsis

```
int thrd_equal(thrd_t thr0, thrd_t thr1);
```

Description

The **`thrd_equal`** function will determine whether the thread identified by **`thr0`** refers to the thread identified by **`thr1`**.

Returns

The **`thrd_equal`** function returns zero if the thread **`thr0`** and the thread **`thr1`** refer to different threads. Otherwise **`thrd_equal`** returns a non-zero value.

The `thrd_exit` function

Synopsis

```
void thrd_exit(int res);
```

Description

The **`thrd_exit`** function terminates execution of the calling thread and sets its result code to **`res`**.

Returns

The **`thrd_exit`** function returns no value.

The `thrd_join` function

Synopsis

```
int thrd_join(thrd_t thr, int *res);
```

Description

The **`thrd_join`** function communicates to the operating system that all resources allocated to the thread identified by **`thr`** should be terminated and all resources allocated freed and blocks until that thread has terminated. If the parameter **`res`** is not a null pointer it stores the thread's result code in the integer pointed to by **`res`**. The value of the thread identified by **`thr`** value shall not have been set by a call to **`thrd_join`** or **`thrd_detach`**.

Returns

The **`thrd_join`** function returns:

- **`thrd_success`** – on success or
- **`thrd_error`** – when request could no be honored.

The `thrd_sleep` function

Synopsis

```
void thrd_sleep(const xtime *xt);
```

Description

The **`thrd_sleep`** function suspends execution of the calling thread until after the time specified by the `xtime` object pointed to by **`xt`**.

Returns

The **thrd_sleep** function returns no value.

The thrd_yield function

Synopsis

```
void thrd_yield(void);
```

Description

The **thrd_yield** function permits other threads to run even if the current thread would ordinarily continue to run.

Returns

The **thrd_yield** function returns no value.

The tss_create function

Synopsis

```
int tss_create(tss_t *key, tss_dtor_t dtor);
```

Description

The **tss_create** function creates a thread-specific storage pointer with destructor **dtor**, which may be null.

Returns

If the **tss_create** function is successful it sets the thread-specific storage pointed to by **key** to a value that uniquely identifies the newly created pointer and returns **thrd_success**, else a **thrd_error** is returned and the thread-specific storage pointed to by **key** is set to an undefined value.

The tss_delete function

Synopsis

```
void tss_delete(tss_t key);
```

Description

The function releases any resources used by the thread-specific storage pointer **`key`**.

Returns

The **`tss_delete`** function returns no value.

The `tss_get` function

Synopsis

```
void *tss_get(tss_t key);
```

Description

The **`tss_get`** function returns the value for the current thread held in the thread-specific storage pointer identified by **`key`**.

Returns

The **`tss_get`** function returns the value for the current thread if successful, else a 0.

The `tss_set` function

Synopsis

```
int tss_set(tss_t key, void *val);
```

Description

The **`tss_set`** function sets the value for the current thread held in the thread-specific storage pointer identified by **`key`** to **`val`**.

Returns

The **`tss_set`** function returns:

- **`thrd_success`** – on success or
- **`thrd_error`** – when request could no be honored.

The `xtime_get` function

Synopsis

```
int xtime_get(xtime *xt, int base);
```

Description

The **xtime_get** function sets the **xtime** object pointed to by **xt** to hold the current time based on the time base **base**.

Returns

If the **xtime_get** function is successful it returns the non-zero value base, which must be TIME_UTC; otherwise it returns 0[1].

[1] Although an **xtime** object describes times with nanosecond resolution the actual resolution in an **xtime** object is system dependent.

TYPES

cnd_t

```
typedef o-type cnd_t;
```

The type is an object type *o-type* that holds an identifier for a condition variable.

thrd_t

```
typedef o-type thrd_t;
```

The type is an object type *o-type* that holds an identifier for a thread.

tss_t

```
typedef o-type tss_t;
```

The type is an object type *o-type* that holds an identifier for a thread-specific storage pointer.

mtx_t

```
typedef o-type mtx_t;
```

The type is an object type *o-type* that holds an identifier for a mutex.

tss_dtor_t

```
typedef void (*tss_dtor_t)(void*);
```

The type is the function type for a destructor for a thread-specific storage pointer.

thrd_start_t

```
typedef int (*thrd_start_t)(void*);
```

The type is the function type that is passed to **thrd_create** to create a new thread.

once_flag

```
typedef o-type once_flag;
```

The type is an object type *o-type* that holds a flag for use by call_once.

mtx_plain

`enum { mtx_plain = ..... };`

The compile-time constant is passed to **`mtx_init`** to create a mutex object that supports neither timeout nor test and return.

mtx_recursive

`enum { mtx_recursive = ..... };`

The compile-time constant is passed to **`mtx_init`** to create a mutex object that supports recursive locking.

mtx_timed

`enum { mtx_timed = ..... };`

The compile-time constant is passed to **`mtx_init`** to create a mutex object that supports timeout.

mtx_try

`enum { mtx_try = ..... };`

The compile-time constant is passed to **`mtx_init`** to create a mutex object that supports test and return.

RETURN CODES

thrd_timedout

enum { thrd_timedout = ... };

The compile-time constant is returned by a timed wait function to indicate that the time specified in the call was reached without acquiring the requested resource.

thrd_success

enum { thrd_success = ... };

The compile-time constant is returned by a function to indicate that the requested operation succeeded.

thrd_busy

enum { thrd_busy = ... };

The compile-time constant is returned by a function to indicate that the requested operation failed because a resource requested by a test and return function is already in use.

thrd_error

enum { thrd_error = ... };

The compile-time constant is returned by a function to indicate that the requested operation failed.

thrd_nomem

enum { thrd_nomem = ... };

The compile-time constant is returned by a function to indicate that the requested operation failed because it was unable to allocate memory.

MACROS

ONCE_FLAG_INIT

`#define ONCE_FLAG_INIT <`*`object initializer`*`>`

The macro yields a value that can be used to initialize an object of type **once_flag**.

TSS_DTOR_ITERATIONS

`#define TSS_DTOR_ITERATIONS <`*`integer constant expression`*`>`

The macro yields the maximum number of times that destructors will be called when a thread terminates.