

**NAME**

sthread\_t – Shore Thread Abstract Base Class

**SYNOPSIS**

```
#include <sthread.h>

/* See sthread.h - too long to include here */
```

**DESCRIPTION**

The thread mechanism allows several threads of control to share the same address space. Each thread is represented by an instance of class **sthread\_t**. Once created, a thread is an independent entity with its own stack. A thread's execution begins with its **run** method and ends when **run** returns. A thread can also be made to end early by calling **end**, which forces a longjmp out of the **run** method.

A thread is created by allocating it from the heap with a call to **new** and it is started by calling its method **fork**. One can await a thread's completion by calling its method **wait**. The following code is an example from the Tcl-based storage manager test shell, **ssh**.

```
/*
 * ssh_smthread_t is derived from smthread_t, which is
 * derived from sthread_t.
 */
smthread_t *doit = new ssh_smthread_t(f_arg);
if (!doit) {
    /* error - out of memory */
}

w_rc_t rc = doit->fork();
if(rc) {
    /* fatal error */
}

w_rc_t rc = doit->wait();
if(rc) {
    /* fatal error */
}

delete doit;
```

In a C++ program, the sthread initialization code is built into the library such that it will execute before the **main** function. The initialization code is responsible for spawning a **main\_thread**, such that, when the initialization function returns, it returns in the context of the **main\_thread**. This ensures that the program executes in a threaded environment from the very beginning.

Class **sthread\_base\_t** is the base class for all sthread classes. It defines constants and enums used throughout the thread package.

Class **sthread\_named\_base\_t** inherits from **sthread\_base\_t** and adds a name string to the class. Its only purpose is to ease debugging by providing a name to sthread classes.

Class **sthread\_t** is an abstract base class that represents a thread of execution; it must be derived in order to be used, hence the protected constructor.

**Enumerations****enum status\_t**

A thread can be in one of the following states at any one time:

<b>t_defunct</b>	is dead
<b>t_ready</b>	is in the ready queue waiting to run
<b>t_running</b>	is running
<b>t_blocked</b>	is not ready to run

**enum priority\_t**

These are the thread priority levels in decreasing order:

<b>t_time_critical</b>	has highest priority
<b>t_regular</b>	has regular priority
<b>t_fixed_low</b>	has lower than regular priority
<b>t_idle_time</b>	only runs when system is idle

**Methods****sthread\_t(priority, block\_immediate, auto\_delete, name)**

The constructor creates a *priority* level thread. If *block\_immediate* is true, the thread will automatically be run sometime soon. Otherwise, the thread is blocked awaiting an explicit **unblock** call. If *auto\_delete* is true, the thread automatically deallocates (destroys) itself when it ends. Otherwise, the caller must deallocate the thread with **delete**. The *name* parameter is used for debugging purposes only.

The constructor is protected because **sthread\_t** is an abstract base class. Users should derive from **sthread\_t** and **provide** the virtual **run** method.

**~sthread\_t()**

The destructor deallocates the stack and other resources used by the thread.

**run()**

Method **run** is automatically started (by the thread switching code) when a thread begins execution. It is a pure virtual function that must be implemented in a derived class. The thread ends when

**static end()**

The **end** method ends the execution of the current thread by forcing a longjmp out of the **run** method.

**static block(timeout, list, caller)**

The **block** method makes the current thread dormant for at least *timeout* milliseconds. The thread can be awakened explicitly by an **unblock** call. The calling thread's tcb is inserted into { *list* }, and the *caller* string is saved for debugging purposes. Note that **block** only returns when the thread is unblocked (by another thread). Ordinarily, programs do not call **block** or **unblock**, since they are the basis for more powerful synchronization mechanisms: mutexes and condition variables.

**unblock(rc)**

The **unblock** method unblocks the thread with an the error *rc* and marks it as ready to continue running. The value of *rc* will be returned from the **block** method.

#### **static me()**

The **me** method returns a pointer to the current (running) thread.

#### **wait(timeout)**

The **wait** method waits for the thread to terminate. The method returns without error when the thread terminates within *timeout* milliseconds. Otherwise, a timeout error is returned.

#### **sleep(timeout)**

The **sleep** method causes the thread to halt execution for *timeout* milliseconds. Other threads continue running.

#### **yield()**

The **yield** method gives up the CPU so other threads can run. The current thread remains active and will be run again soon.

### **I/O Operations**

The thread package provides asynchronous I/O operations. Threads performing these operations will block, but the server process will not. The implementation was developed for operating systems that do not provide threads or asynchronous I/O. For each open file a process, is started. When a thread needs I/O on a file, the sthread library blocks the thread, forwards its request to and switches to another ready thread. When the I/O request is complemented, the process informs the sthread library, which in turn unblocks the original thread that requested the I/O.

**Sthread\_t** provides methods similar to Unix in handling file I/O. However, the file descriptors used by these methods are not interchangeable with that of Unix, i.e., the *fd* returned by **sthread\_t::open** can only be used with other methods in **sthread\_t** such as **sthread\_t::read**.

These I/O operations are closely integrated with buffer pool management. The storage manager buffer pool is located in memory shared with the processes. I/O requests must refer to locations in this shared memory.

Now that most popular operating systems provide threads and asynchronous I/O, the **sthread\_t** I/O operations should be re-implemented or even eliminated.

### **ERRORS**

See **errors(sthread)**

### **EXAMPLES**

#### **VERSION**

This manual page applies to Version 2.0 of the Shore Storage Manager.

### **SPONSORSHIP**

The Shore project is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract

STHREAD\_T(STHREAD)

STHREAD \_T(STHREAD)

DAAB07-91-C-Q518. Further funding for this work was provided by DARPA through Rome Research Laboratory Contract No. F30602-97-2-0247.

## **COPYRIGHT**

Copyright (c) 1994-1999, Computer Sciences Department, University of Wisconsin -- Madison. All Rights Reserved.

## **SEE ALSO**

**smthread\_t(ssm), smutex\_t(sthread), scond\_t(sthread), sevsem\_t(sthread), file\_handlers(sthread), intro(sthread).**