

NAME

sthread_t – Shore Thread Abstract Base Class

SYNOPSIS

```
#include <sthread.h>

class sthread_base_t : public w_base_t {
public:
    typedef uint4_t id_t;

    enum {
        max_thread    = 32,
        WAIT_IMMEDIATE = 0,
        // sthreads package recognizes 2 WAIT_* values:
        // == WAIT_IMMEDIATE
        // and != WAIT_IMMEDIATE.
        // If it's not WAIT_IMMEDIATE, it's assumed to be
        // a positive integer (milliseconds) used for the
        // select timeout.
        //
        // All other WAIT_* values other than WAIT_IMMEDIATE
        // are handled by sm layer.
        //
        // The user of the thread (e.g., sm) had better
        // convert timeouts that are negative values (WAIT_* below)
        // to something >= 0 before calling block().
        //
        WAIT_FOREVER    = -1,
        WAIT_ANY        = -2,
        WAIT_ALL         = -3,
        WAIT_SPECIFIED_BY_THREAD = -4, // used by lock manager
        WAIT_SPECIFIED_BY_XCT = -5, // used by lock manager
        stack_sz        = (1024 * 64)
    };

#include "st_error.h"

    enum {
        stOS = fcOS,
        stINTERNAL = fcINTERNAL,
        stNOTIMPLEMENTED = fcNOTIMPLEMENTED,
    };
};

#define NAME_ARRAY 64
class sthread_name_t {
public:
    void rename(const char *n1, const char *n2=0, const char *n3=0);
    NORET sthread_name_t();
    NORET ~sthread_name_t();
    char _name[NAME_ARRAY];
    NORET W_FASTNEW_CLASS_DECL;
```

```

};

class sthread_named_base_t: public sthread_base_t {
public:
    NORET          sthread_named_base_t(
        const char*      n1 = 0,
        const char*      n2 = 0,
        const char*      n3 = 0);
    NORET          ~sthread_named_base_t();

    void            rename(
        const char*      n1,
        const char*      n2 = 0,
        const char*      n3 = 0);

    const char*      name() const;
    void            unname();
};

/*
 * Thread Structure
 */
class sthread_t : public sthread_named_base_t {
public:
    enum status_t {
        t_defunct, // status before and after run()
        t_ready,   // status when entering run()
        t_running, // when me() is this thread
        t_blocked, // thread is blocked on anything
    };

    enum priority_t {
        t_time_critical = 3,
        t_regular       = 2,
        t_fixed_low     = 1,
        t_idle_time     = 0,
        max_priority    = t_time_critical,
        min_priority    = t_idle_time
    };

    void*          user; // user can use this
    const id_t      id;
    int             trace_level; // for debugging

    /*
     * Thread Control
     */
    static w_rc_t    block(
        int4_t        timeout = WAIT_FOREVER,
        sthread_list_t* list = 0,
        const char* const caller = 0);

```

```

static void                yield(bool doselect=false);
w_rc_t                    unblock(const w_rc_t& rc = *(w_rc_t*)0);
static void                end();
static void                dump(const char *, ostream &);

w_rc_t                    set_priority(priority_t priority);
priority_t                priority() const;
status_t                  status() const;
w_rc_t                    set_use_float(int);
void                      push_resource_alloc(const char* n);
void                      pop_resource_alloc();

/*
 * Asynchronous I/O ops
 */
static char*              set_bufsize(int size);
static w_rc_t             open(
    const char*            path,
    int                    flags,
    int                    mode,
    int&                   fd);
static w_rc_t             close(int fd);
static w_rc_t             writev(
    int                    fd,
    const iovec*           iov,
    size_t                 iovcnt);
static w_rc_t             write(
    int                    fd,
    const void*            buf,
    int                    n);
static w_rc_t             read(
    int                    fd,
    const void*            buf,
    int                    n);
static w_rc_t             readv(
    int                    fd,
    const iovec*           iov,
    size_t                 iovcnt);
static w_rc_t             lseek(
    int                    fd,
    off_t                  offset,
    int                    whence,
    off_t&                 ret);
static w_rc_t             fsync(int fd);

/*
 * Misc
 */
static pthread_t*         me() { return _me; }
void                      sleep(long timeout = WAIT_FOREVER);
w_rc_t                    wait(long timeout = WAIT_FOREVER);

```

```

static void                set_diskrw_name(const char* diskrw)
                           {_diskrw = diskrw;}

friend ostream&            operator<<(
    ostream&               o,
    const sthread_t&        t);

protected:
    NORET                  sthread_t(
        priority_t         priority = t_regular,
        bool               block_immediate = false,
        bool               auto_delete = false,
        const char*        name = 0);
    virtual NORET          ~sthread_t();

    virtual void            run() = 0;

};

```

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.

In a C++ program, the sthread initialization code is built into the library such 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:

t_defunct	is dead
t_ready	is in the ready queue waiting to run
t_running	is running
t_blocked	is not ready to run

enum priority_t

These are the thread priority levels in decreasing order:

t_time_critical	has highest priority
t_regular	has regular priority
t_fixed_low	has lower than regular priority
t_idle_time	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, `diskrw`, is started. When a thread needs I/O on a file, the `sthread` library blocks the thread, forwards its request to `diskrw` and switches to another ready thread. When the I/O request is complemented, the `diskrw` 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 `diskrw` 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 1.1 of the Shore software.

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 DAAB07-91-C-Q518.

COPYRIGHT

Copyright © 1994, 1995, 1996, 1997, Computer Sciences Department, University of WisconsinMadison. All Rights Reserved.

SEE ALSO

smthread_t(ssm), **smutex_t(sthread)**, **scond_t(sthread)**, **sevsem_t(sthread)**, **file_handlers(sthread)**, **intro(sthread)**.