SUN/266.1

Starlink Project
Starlink User Note 266.1

D.S. Berry

12th September 2011

THR
A Thread Management Library
Version 1.0
Programmer's Manual

Abstract

THR provides high level utility functions for creating and using pools of persistent execution thread.

Contents

1 Introduction

This library contains functions that can be used to create and use pools of persistent worker threads. It is wrapper around various functions in the pthread library.

Note, functions in this library cannot be used from Fortran.

This library currently includes:

(1) Wrappers for the basic pthreads functions, that add inherited status handling. These include:

- thrMutexInit: Initialise a mutex
- thrCondInit: Initialise a condition variable
- thrThreadCreate: Create a thread
- thrMutexLock: Lock a mutex
- thrMutexUnlock: Unlock a mutex
- thrCondBroadcast: Broadcast a condition
- thrCondSignal: Signal a condition
- thrCondWait: Wait for a condition

(2) A set of functions that maintains a pool of threads ready for use. Each thread in the pool is described as a "worker" and the whole pool is described as a "workforce". The idea is that a task is split into separate jobs, and all jobs are performed in parallel by the workers in the workforce. Once a workforce has been told about all the jobs within a task (using thrAddJob), the calling thread waits until all the jobs have been completed.

A Function Descriptions

thrAddJob
Add a job to the list of jobs to be performed by a given workforce

Description:

This function adds a job to the list of jobs to be performed by the workforce. The job will start immediately if a worker thread is available to execute the job, and any jobs specified in the " wait_on" list have completed. Otherwise, it will start as soon as a worker thread becomes available and all the " wait_on" jobs have completed. Jobs are not necessarily started in the order in which they are added to the workforce.

Invocation:

```
int thrAddJob( ThrWorkForce *workforce, int flags, void *data, void (*func)( void *,
int * ), int nwait_on, const int *wait_on, int *status )
```

Arguments:

workforce

Pointer to the workforce. If NULL is supplied, the job is executed immediately in the current thread by calling " func" , and the " flags" and " checker" arguments are ignored.

flags

Flags controlling how the job behaves. See " Job Control Flags:" below.

data

An arbitrary data pointer that will be passed to the worker allocated to perform this job. If the THR__FREE_JOBDATA flag is set (see " flags") the pointer will be freed automatically by the function registered using thrFreeFun when the job completes.

func

A pointer to a function that the worker will invoke to do the job. This function takes two arguments; 1) the supplied " data" pointer, and 2) an inherited status pointer. It returns void.

nwait_on

The number of values supplied in the " wait_on" array. If zero, the " wait_on" pointer will be ignored.

wait_on

An array of integer identifiers for previously created jobs. The length of this array is given by " nwait_on" . No attempt will be made to start the new job until all the jobs specified in this array have completed. If NULL is supplied, or if " nwait_on" is zero, the new job will be started as soon as a worker thread becomes available.

status

Pointer to the inherited status value.

Returned Value:

A positive integer identifier for the job. Zero if an error occurs.

Job Control Flags :

- THR__REPORT_JOB: Indicates that this job is to be included in the list of jobs for which thrJobWait will wait.

- THR__FREE_JOBDATA: Indicates that the supplied pointer to the job data ("data") is to be freed when the job completes. This is performed by passing the supplied "data" pointer to the user-supplied function registered using thrFreeFun (astFree is used if no function is registered). Note, thrFreeFun is called from within the worker thread.

thrBeginJobContext
Starts a new job context

Description:

This function indicates that all jobs created before the subsequent matching call to thrEndJobContext should be grouped together. This affects the behaviour of functions thrWait and thrJobWait.

Invocation:

```
void thrBeginJobContext( ThrWorkForce *workforce, int *status )
```

Arguments:

workforce

Pointer to the workforce performing the jobs. If NULL is supplied, this function returns without action.

status

Pointer to the inherited status value.

thrCondBroadcast
A wrapper for pthread_cond_broadcast

Description:

This function broadcasts a condition to all threads, unblocking all threads that are blocked on the condition variable.

Invocation:

```
void thrCondBroadcast( pthread_cond_t *cond, int *status )
```

Arguments:

cond

Pointer to the condition variable.

status

Pointer to the inherited status value.

Notes:

- This function attempts to execute even if an error has already occurred, although no further error will be reported if this function should then subsequently fail.

thrCondInit
A wrapper for pthread_cond_init

Description:
This function initialises a condition variable using default attributes.

Invocation:

```
void thrCondInit( pthread_cond_t *cond, int *status )
```

Arguments:

cond
The condition variable to be initialised.

status
Pointer to the inherited status value.

thrCondSignal
A wrapper for pthread_cond_signal

Description:

This function signals a condition, unblocking at least one thread that is blocked on the condition variable.

Invocation:

```
void thrCondSignal( pthread_cond_t *cond, int *status )
```

Arguments:

cond

Pointer to the condition variable.

status

Pointer to the inherited status value.

Notes:

- This function attempts to execute even if an error has already occurred, although no further error will be reported if this function should then subsequently fail.

thrCondWait
A wrapper for pthread_cond_wait

Description:

This function blocks the calling thread until a condition is signalled or broadcast.

Invocation:

```
void thrCondWait( pthread_cond_t *cond, pthread_mutex_t *mutex, int *status )
```

Arguments:

cond

Pointer to the condition variable.

mutex

Pointer to the associated mutex.

status

Pointer to the inherited status value.

thrCreateWorkforce
Create a thread pool holding a specified number of threads

Description:

This function creates a new " workforce" - a pool of threads that can be used to execute tasks in parallel. Each task should be split into two or more jobs, and a description of each job should be given to the workforce using thrAddJob. As each job is added, any available worker thread claims the job and executes it. If all workers are busy, the jobs will be claimed by workers once they have completed their current jobs. Once all jobs have been added, thrWait should be called to wait until all jobs have ben completed.

The model used by this module is of a group of people (each person representing a thread), most of which are " workers" whose duty it is to collect jobs from a " job desk" and then go away and perform them, returning to the job desk to report completion of the job and to get a new job. The lists of available jobs, active jobs, etc, are kept at the job desk, and everyone must queue at the job desk to gain access to this information. In order to prevent confusion being caused by different people acccessing the job desk information at the same time, only the person at the head of the queue can access these lists (stored in the workforce structure in reality). This job desk queue is implemented using a mutex - a thread (person) joins the queue by attempting to lock the mutex. At this point the thread blocks (the person waits) until they have reached the head of the queue as indicated by the thrMutexLock call returning. [In fact the queue is not guaranteed to be first-in, first-out - some " queue jumping" may occur as determined by the thread scheduler - but that shouldn' t matter.]

Two condition variables are used; one is used to signal that new jobs have been placed on the job desk (any idle workers will respond to this signal by rejoining the job desk queue), and the other is used to signal that all jobs have been completed (i.e. no jobs waiting to be started and no active jobs). The person (thread) who added the jobs to the table will respond to this signal by waking up and continuing with whetever else it has to do (which may include submitting more jobs to the job desk).

Invocation:

```
ThrWorkForce *thrCreateWorkforce( int nworker, int *status )
```

Arguments:

nworker
The number of threads within the new thread pool. If zero, a NULL pointer will be returned without error.

status
Pointer to the inherited status value.

Returned Value:

A pointer to a structure describing the new thread pool. The

returned pool should be freed using thrDestroyWorkforce when

no longer needed.

thrDestroyWorkforce
Destroy a workforce

Description:

This function frees all resources used by a work force. This includes cancelling the worker threads, and freeing memory structures. The calling thread blocks until any busy workers have completed their jobs. The worker threads themselves are then terminated.

Invocation:

```
ThrWorkForce *thrDestroyWorkforce( ThrWorkForce *workforce )
```

Arguments:

workforce

Pointer to the workforce to be destroyed. If NULL is supplied, this function returns without action.

Returned Value:

A NULL pointer is returned.

thrEndJobContext
End the current job context

Description:

This function ends the job context started by the earlier matching call to thrBeginJobContext. Any remaining jobs belonging to the current job context are exported into the parent job context.

Invocation:

```
void thrEndJobContext( ThrWorkForce *workforce, int *status )
```

Arguments:

workforce

Pointer to the workforce performing the jobs. If NULL is supplied, this function returns without action.

status

Pointer to the inherited status value.

thrFreeFun
Register a function to delete a job data structure

Description:

This function can be used to register a function that will be called by the thr library to delete a job data structure once a job has completed. In this context, a " job data structure" is the data structure passed to thrAddJob when a job is submitted to the workforce. The registered function will be called to delete the job data structure only if the THR__FREE_JOBDATA flag is specified when the job was submitted to the workforce using thrAddJob. If no function is registered, the astFree function will be used by default. This is only appropriate if the data structure does not contain any dynamically allocated arrays or other resources that need to be released before freeing the structure.

The specified function, or astFree if no function is specified, is called from within the worker thread.

Invocation:

```
void *(*thrFreeFun( void *(*freejob)( void *, int *) ))( void *, int * )
```

Arguments:

freejob

Pointer to the function to be called to free a job data structure. It should take two arguments - a " void *" pointer to the structure to be freed and an " int *" pointer to the inherited status value. It should always return a NULL pointer. If NULL is supplied FOR the function pointer (or if this function has not been called), then astFree will be used to free job data structures.

Returned Value:

The pointer to the previously registered function, or NULL if no

function is currently registered.

thrGetJobData
Returns a job data pointer that was supplied when the job was created

Description:

This function returns the pointer that was supplied as argument " data" when thrAddJob was called to create the specified job.

Invocation:

```
void *thrGetJobData( int ijob, ThrWorkForce *workforce, int *status )
```

Arguments:

ijob

Identifier for the job.

workforce

Pointer to the workforce. NULL should be supplied if this function is called from within a job executing in a worker thread.

status

Pointer to the inherited status value.

Returned Value:

The pointer to the job data. NULL is returned if the job is not

found, but no error is reported.

thrGetJobs
Return a list of jobs in a given state

Description:

This function returns a list containing the identifiers for all job currently in the specified state. This is a snapshot at the moment this function is called. Jobs may have changed state by the time the calling function gets round to processing the returned list.

Invocation:

```
int *thrGetJobs( ThrWorkForce *workforce, int state, int *njob, int *status )
```

Arguments:

workforce

Pointer to the workforce. NULL should be supplied if this function is called from within a job executing in a worker thread.

state

An integer in which each bit is a boolean flag indicating if jobs in a particular state should be included in the returned list. The supplied value should be the union of one or more of the following values defined in header file " thr.h" :

THR__ACTIVE: active jobs that are currently running or halted THR__AVAILABLE: inactive jobs that have not yet been started but are available to run as soon as a worker becomes available. THR__FINISHED: inactive jobs that have finished running and are awaiting other jobs to finish before being freed. THR__WAITING: inactive jobs that are waiting for other jobs to finish before being started

njob

Pointer to an int in which to return the length of the returned list of job identifier.

status

Pointer to the inherited status value.

Returned Value:

A pointer to a newly allocated list of job identifier. Its length

is given by the value returned in " *njob" . It should be freed using

astFree when no longer needed. A NULL pointer will be returned if an

error occurrs.

thrGetNThread
Determine the number of threads to use

Description:

This function returns the number of worker threads to use when dividing a task up between multiple threads. Note, a value of " 1" means one worker thread in addition to the required manager thread that co-ordinates the workers (i.e. the main thread in which the application is started). The default value is the number of CPU cores available, but this can be over-ridden by setting the environment variable specified by the " env" argument to some other value. A value fo zero is returned if the app should run in a single thread without any worker threads.

Invocation:

```
int thrGetNThread( const char *env, int *status );
```

Arguments:

env = const char * (Given)

Pointer to the name of an environment variable which should be used to get the number of threads (e.g. " SMURF_THREADS").

status = int* (Given and Returned)

Pointer to inherited status.

Returned Value:

The number of threads to use. A value of 1 is returned if an error

occurs. A value of zero indicates that the application should run

in a single thread.

thrGetWorkforce
Return a pointer to a singleton workforce

Description:

Applications that may be run in a monolith environment such as ICL or ORAC-DR should normally use this function in preference to thrCreateWorkforce. Use of this function reduces the total number of threads that are started and killed within a monolith, thus reducing the associated overheads of CPU time and memory.

One the first invocation, this function invokes thrCreateWorkforce to create a new workforce with the requested number of threads. A pointer to this workforce is stored internally, and the same pointer is returned on each subsequent invocation of this function.

If the returned workforce is freed explicitly using thrDestroyWorkforce, then the next invocation of this function will create a new workforce again. For this reason, applications should not normally free the returned workforce explicitly. The resources associated with the workforce will be freed when the monolith process terminates.

Invocation:

```
ThrWorkForce *thrGetWorkforce( int nworker, int *status )
```

Arguments:

nworker

If this value is negative, a NULL pointer is returned if no workforce exists on entry, and a pointer to the existing workforce is returned otherwise. If " nworker" is zero, a NULL pointer is always returned (in which case the app should be run in a single thread without any workers). If " nworker" is positive, it will be ignored if a workforce already exists, and will be used to specify the number of worker threads in the new workforce otherwise.

status

Pointer to the inherited status value.

Returned Value:

A pointer to a workforce. The returned pointer should not usually be

freed explicitly (e.g. with thrDestroyWorkforce).

thrHaltJob
Halt a running job until other jobs have completed

Description:

This function is intended to be called by a worker thread during the execution of a job. It blocks the current thread until a specified list of other jobs have finished, at which time the current thread resumes.

Invocation:

```
void thrHaltJob( ThrWorkForce *workforce, int njob, int *job_list, int *status )
```

Arguments:

workforce
Pointer to the workforce. NULL should be supplied if this function is called from within a job executing in a worker thread.

njob
The number of job identifiers in the " job_list" array.

job_list
A list of job identifiers. The current thread blocks until all the listed jobs have finished. Any identifiers in this list that refer to jobs that have already finished are ignored.

status
Pointer to the inherited status value.

thrJobWait
Wait for the next job to completed

Description:

Each consecutive call to this function return the integer identifier for a completed job, in the order in which they are completed. If all completed jobs have already been reported, then this function blocks until the next job is completed.

Note, only jobs which had the THR__REPORT_JOB flag set when calling thrAddJob and were created within the current job context (see thrBeginJobContext) are included in the list of returned jobs.

Invocation:

```
int thrJobWait( ThrWorkForce *workforce, int *status )
```

Arguments:

workforce

Pointer to the workforce. If NULL is supplied, this function exits immediately, returning a value of zero.

status

Pointer to the inherited status value.

Returned Value:

The integer identifier for the completed job. This can be compared

with the job identifiers returned by thrAddJob to determine which

job has finished. A value of -1 is returned if the workforce has no

no remining jobs.

Notes:

- This function attempts to execute even if an error has already occurred.

thrMutexInit
A wrapper for pthread_mutex_init

Description:

This function initialises a mutex using default attributes.

Invocation:

```
void thrMutexInit( pthread_mutex_t *mutex, int *status )
```

Arguments:

mutex

The mutex to be initialised.

status

Pointer to the inherited status value.

thrMutexLock
A wrapper for pthread_mutex_lock

Description:
This function locks a mutex.

Invocation:

```
void thrMutexLock( pthread_mutex_t *mutex, int *status )
```

Arguments:

mutex
Pointer to the mutex.

status
Pointer to the inherited status value.

thrMutexUnlock
A wrapper for pthread_mutex_unlock

Description:

This function unlocks a mutex.

Invocation:

```
void thrMutexUnlock( pthread_mutex_t *mutex, int *status )
```

Arguments:

mutex

Pointer to the mutex.

status

Pointer to the inherited status value.

Notes:

- This function attempts to execute even if an error has already occurred, although no further error will be reported if this function should then subsequently fail.

thrThreadCreate
A wrapper for pthread_create

Description:
This function creates a new thread using default attributes.

Invocation:

```
void thrThreadCreate( pthread_t *thread, void *(*start_routine)(void*),
```

Arguments:

thread
Pointer to the pthread structure to initialise.

start_routine
Pointer to the routine to run in the new thread.

arg Pointer to be passed to the start routine.

status
Pointer to the inherited status value.

thrThreadData
Returns a KeyMap that can be used to hold thread-specific global data

Description:

This function returns a pointer to an AST KeyMap that is associated with the running thread (each thread has a separate KeyMap). The KeyMap can be used to store values that need to be passed between functions within a thread, or that need to be retained between invocations.

Invocation:

```
AstKeyMap *thrThreadData( int *status );
```

Arguments:

status

Pointer to the inherited status value.

Notes:

- The returned Keymap, plus any data still in it, is released when the thread terminates.
- This function attempts to execute even if an error has already occurred.

thrWait
Wait for a workforce to have completed all its jobs

Description:

This function blocks the calling thread until all jobs within the current job context (see thrBeginJob-Context) have been completed.

A side effect of this function is to empty the list of jobs waiting to be reported by thrJobWait. Upon exit from this function, all jobs waiting to be reported via thrJobWait will be considered to have been reported (again, this only affects jobs within the current job context).

Invocation:

```
thrWait( ThrWorkForce *workforce, int *status )
```

Arguments:

workforce

Pointer to the workforce. If NULL is supplied, this function returns immediately.

status

Pointer to the inherited status value.

Notes:

- This function attempts to execute even if an error has already occurred.