Thread Library

Contents

Introduction

Multitasking operating systems allow multiple tasks, or processes, to execute—seemingly concurrently—on a single processor. However, the overhead necessary to maintain a full process, both in space and in time, can make using multiple processes fairly expensive. Threads are lightweight versions of processes. Threads inhabit a common address space and share common global data within a single application. This allows threads to consume fewer resources than a full-fledged process might consume, and makes switching between threads significantly faster than switching between processes.

A typical Macintosh application might use threads to allow the user to continue to work in an application while the same application is simultaneously performing some lengthy operation. For instance, an application like the Finder might create a new thread every time the user copies files. The copy would execute concurrently while the user continues to use the Finder to perform further copy operations, or to do any other task within the Finder. As another example, a word processing application might create a thread when a user asks it to check the spelling of a long document. While the word processor is locating all misspelled words, the user could continue to edit other documents.

Such capabilities are possible even without using threads. For instance, one could create a task, with an associated entry point function, that is called on null events. The task would execute some small chunk of work, save its current state, and return to its caller. The next time a null event is received, the caller would call the task, which would restore its saved state, perform another small chunk of work, etc. However, such alternative approaches can be very cumbersome to implement. In a thread, the state of a task can be maintained on a completely separate stack, and there is no need to unroll the function call stack in order to allow the application to continue to execute. Thus, threads can lead to a much simpler style of programming that supports simulated concurrency on a single processor machine[1].

Description

Thread Library is a free library, for use by Macintosh software developers, that implements cooperative multiple thread execution within a single application. Thread Library

- does not require any extensions;
- should work with all Macintosh models (from the Plus on up);
- works with system 7.x and with system 6.x under Finder or MultiFinder;
- runs in native mode on 680x0 and PowerPC based Macintoshes;
- compiles into a small library of 3 to 8 kilobytes;
- works with the Symantec, Metrowerks, and MPW compilers.

---

[1] Threads can also be used in multiprocessor computers, where they can be used to provide true concurrency.

The 680x0 version of Thread Library is 2-3 times faster than Apple's Thread Manager. The PowerPC version of Thread Library is about 2 times faster than Apple's Thread Manager.

Every thread has its own stack, and there are no restrictions on the objects that can be allocated on a thread's stack. All other global application data are shared by threads. Context switches are very efficient since they involve only a few operations to save the current thread's state, followed by a context switch to the new thread, and a few instructions to restore the new thread's state.

This distribution includes:

- complete source code in C;
- detailed documentation in Microsoft Word 5.0 format;
- prebuilt libraries for 68K and PowerPC programs, including debug versions of the libraries;
- a new interface that mimics Apple's Thread Manager;
- a simple test application that demonstrates how threads may be used;
- an application that compares the speed of Thread Library with the speed of Apple's Thread Manager;
- Metrowerks CodeWarrior project files;
- AppleScript program for automatic builds.

Contents of this Distribution

This distribution consists of several folders, containing documentation, demonstration programs, and source code for Thread Library.

Documentation.....Contains files that provide documentation about Thread Library.

- Distribution.....Documents the terms under which Thread Library may be used and distributed.
- Thread Library Documentation.....This document.
- Version History .....Documents the changes made to each version.

Executables .....Contains executable versions of the demonstration applications.

Interfaces.....Header files for Thread Library.

- ThreadLibrary.h .....Declarations and definitions for using Thread Library. You need to include this header in any file that makes use of Thread Library.
- ThreadLibraryManager.h.....Interface to Thread Library that mimics Apple's Thread Manager.
- ThreadLibraryStubs.h .....Macros that stub out all of the calls to Thread Library with null statements or expressions. You can use this header while debugging to temporarily remove Thread Library from your application.

Libraries.....Compiled libraries for Thread Library.

- ThreadLibrary-68K.o.....Compiled M68K MPW library file for Thread Library. You need to link this library with your application in order to use Thread Library.
- ThreadLibrary-68K-Debug.o.....Compiled M68K MPW library file for Thread Library, with all debug code enabled. You should use this library while debugging your application.
- ThreadLibrary-PPC.o .....Compiled PowerPC library. You need to link this library with your application in order to use Thread Library.
- ThreadLibrary-PPC-Debug.o.....Compiled PowerPC library, with all debug code enabled. You should use this library while debugging your application.
- ThreadLibrary-PPC.lib.....Compiled PowerPC shared library. You may link this library with your application in order to use Thread Library.
- ThreadLibrary-PPC-Dbg.lib .....Compiled PowerPC shared library, with all debug code enabled. You may use this library while debugging your application.

Scripts.....Contains Apple scripts that automate the process of compiling Thread Library.

- Build.as.....Builds all of the components of Thread Library and then removes the object code.

Source .....Source code and project files for Thread Library and the demonstration applications.

- Demos.....Source code for the demonstration applications.
  - ThreadsTest.....Source code for the ThreadsTest application, which is used for testing Thread Library.
  - ThreadsTimed .....Source code for the ThreadsTimed application, which is used for comparing the execution speed of Thread Library with the execution speed of Thread Manager.
- Libraries.....Source code and project files for the compiled libraries.
  - ThreadLibrary.c .....The actual implementation of Thread Library.
  - ThreadLibraryManager.c.....Compatability layer that mimics Apple's Thread Manager using Thread Library.
  - ThreadLibraryPrefix.h.....Prefix file for Metrowerks CodeWarrior project files.
  - regppc.s .....Assembly language routines for the PowerPC version of Thread Library.

Using Thread Library

You should read this document before trying to use Thread Library in your application. There are some important issues in using threads that you should be aware of, otherwise your application may crash. For instance, exception handling and profiling may require changes in order to work correctly with threads. If you are upgrading from a previous version of Thread Library, you should read the sections in the file "Version History" for all newer versions. This will alert you to any important changes to the interfaces to Thread Library.

This document contains comments extracted from the source code that describe the functional interface to Thread Library and notes on how to use the library. The file "ThreadsTest.c" contains the source code for the ThreadsTest application; you can look at it to see how threads are used in a simple application. You can also examine the file "ThreadsTimed.c", which contains the source code to the ThreadsTimed application. Should you need to examine the source code to Thread Library, you can look in the file "ThreadLibrary.c".

Before you use Thread Library, you should run the ThreadsTest application (in the "Executables" folder). The test application displays several identical dialogs, each containing four lines of interest. The first two lines contain two counters, each incremented in its own thread. The dialog is updated once a second by a third thread; the third line shows the number of ticks elapsed between updates, and should be close to 60 ticks; the fourth line shows the number of seconds remaining until the test ends. If the test of Thread Library does not run correctly, then you should disable all extensions by holding down the shift key while starting up your Macintosh and try running the test application again. If the application still does not run correctly, then you may have discovered a bug in Thread Library.

Minimal Use

Thread Library provides two interfaces. One is Thread Library's regular interface, which has no resemblance to Apple's Thread Manager. The other interface is called Thread Library Manager, and provides a largely functionally equivalent interface to Thread Manager. If you are adding Thread Library to an application that already makes use of Thread Manager, you can use the Thread Library Manager interface, which is described in the section “Thread Library Manager”. This section describes the regular interface to Thread Library.

You do not need to use all of the functions in Thread Library in order to use threads in your application. At a minimum, you will need to use the functions ThreadBeginMain, ThreadBegin, ThreadEnd, and ThreadYield. You may also want to use the functions ThreadYieldInterval, ThreadMain, ThreadActive, and ThreadEndAll. In addition, you will need to include the file "ThreadLibrary.h" in every file from which you make calls to Thread Library.

A minimal application might be structured as follows (this is untested sample code):

```
Boolean gQuitting; /* true when exiting application */
Boolean gThreadsAvailable; /* true if can create threads */

static void EventLoop(void)
{
  gQuitting = false;
  while (! gQuitting) {
```

```
    GetAndHandleOneEvent();
    ThreadYield(0);
  }
}

static void InitializeThreads(void)
{
  gThreadsAvailable = true; /* assume we have threads */
  (void) ThreadBeginMain(NULL, NULL, NULL);
  if (ThreadError() != THREAD_ERROR_NONE) {
    /*  handle the error--you can still run your application,
    but you cannot create any threads */
    gThreadsAvailable = false;
  }
}

void main(void)
{
  ThreadType mainThread;

  InitializeManagers();
  InitializeThreads();
  EventLoop();
  ThreadEndAll();
}
```

At some point in your application, you might create a new thread:

```
...
ThreadType thread;
if (gThreadsAvailable) {
  thread = ThreadBegin(MyThreadEntryPoint, NULL, NULL, NULL, 0);
  if (ThreadError() != THREAD_ERROR_NONE) {
    /* handle the error */
  }
}
...

void MyThreadEntryPoint(ThreadDataType data)
{
  Boolean done;

  done = false;
  while (! done) {
    /* do a chunk of work and yield to other threads */
    ThreadYield(0);
  }
}
```

You will need to use additional Thread Library functions to support such features as exception stacks (if your application uses exceptions); custom suspend, resume, and termination callbacks; custom memory allocation callbacks; iteration over threads; custom scheduling; termination of threads; and other facilities provided by Thread Library.

Linking

To add threads to your application, you need to link the appropriate library file with your application. If you are developing a 68K application, you should link with the library "ThreadLibrary-68K-Debug.o" while debugging your application; in your final application, you should link with the library "ThreadLibrary-68K.o". If you are developing a PowerPC application, you should link with the library "ThreadLibrary-PPC-Debug.o" while debugging your application; in your final application, you should link with the library "ThreadLibrary-PPC.o".

Segmentation

An application that uses Thread Library may be segmented just like any other application. However, the segment containing Thread Library must be kept in memory so that context switches will operate correctly. Also, segments containing inactive threads must not be unloaded.

WARNING

The segment containing Thread Library must not be unloaded while there are any threads, and segments containing inactive threads must not be unloaded. ◆

Thread Serial Numbers

Every thread is assigned a unique serial number. Serial numbers are used to refer to threads, rather than using a pointer, since there is always the possiblity that a thread may have terminated before a pointer to a thread is used, which would make the pointer invalid. The specific assignment of serial numbers to threads is not defined by the interface, though every valid thread is guaranteed a non-zero serial number.

IMPORTANT

You should not assume that any thread will have a specific serial number. ◆

Time

All intervals and time periods in Thread Library use Macintosh ticks, which are measured in increments of 1/60th of a second, starting at zero when the Macintosh is turned on. For instance, a value of 60 passed to ThreadYield means that the current thread can yield the processor for up to 1 second.

Concurrency

Threads introduce issues of concurrency and access to shared data in a single application. In most reasonably engineered applications, it should not be too difficult to add threads for certain separable tasks. However, you must be aware of, and alert to, the possiblity of multiple threads attempting to access shared data. It may be necessary to provide basic locking mechanisms to ensure correct execution of your application. Thread Library does not provide locking mechanisms, but any good introductory book that discusses issues of concurrent access to shared resources (such as

books on operating systems or parallel computation) should provide the information you need.

Context

The context maintained for each thread consists of the values of a subset of the CPU's registers. Only registers accessible from user-mode programs are saved by Thread Library, and only registers that are nonvolatile (i.e., whose values must be preserved across function calls) are saved. The registers saved for the M68K version of Thread Library are d3-d7/a2-a7/fp4-fp7. The registers saved for the PowerPC version of Thread Library are GPR1, GPR2, GPR13-GPR31, FPR14-FPR31, and LR.

Detecting Errors

All errors are reported using the function ThreadError, which returns THREAD_ERROR_NONE (defined as zero) if the last function called in Thread Library completed successfully, otherwise it returns an error number. You should call this function after every call to a function in Thread Library.

Handling Errors with Exceptions

If your application uses exceptions to handle errors, you will need to add a custom context switching routine to your threads. Most implementations of exceptions work by modifying the program counter and other registers to restore the state of the application and to jump to an exception handling routine. The code needed to raise an exception typically keeps track of which exception handler to jump to in a global variable. A problem can occur, however, if the exception implementation attempts to jump to a routine that is part of an inactive thread.

For instance, in the following code, the macros TRY, CATCH, and ENDTRY set up an exception handler. The TRY macro indicates the statements to attempt to execute. If an exception occurs while executing these statements, execution jumps to the statements following the CATCH macro. The ENDTRY macro is used to terminate the exception handler. The function ThrowThreadError raises an exception if ThreadError returns any value other than THREAD_ERROR_NONE, and the function ThrowOSErr raises an exception if passed any value other than noErr.

```
void thread1(ThreadDataType data)
{
  Boolean done;

  TRY {
    (void) ThreadBegin(thread2, NULL, NULL, NULL, 0);
    ThrowThreadError();
    done = false;
    while (! done) {
      /* ... do something ... */
      ThreadYield(0);
    }
  } CATCH {
    cleanup();
  } ENDTRY;
```

```
}

void thread2(ThreadDataType data)
{
  while (! done_allocating_memory()) {
    if (! allocate_some_memory())
      ThrowOSErr(memFullErr);
    ThreadYield(0);
  }
}
```

Both thread1 and thread2 have their own private stack and CPU state. When thread2 raises an exception, the exception raising code will attempt to jump to the last exception handler specified. But the last exception handler was specified when thread1 was active. Since the exception raising code does not know about other threads or about Thread Library, it cannot properly switch contexts when the exception is raised. This situation will probably result in a mysterious crash. Since the problem will only occur under extraordinary circumstances (e.g., running out of memory), it will also be hard to reproduce and debug.

When you create a thread, you need to allocate memory to save the state of the exception handler and you need to install your own custom context switching routines. At a minimum you will need to provide suspend and resume callback functions for the thread. The suspend function must save a copy of the state of the exception handler, while the resume function must restore the state of the exception handler. You may also want to provide begin and end callback functions to allocate and dispose of the exception stack. The suspend, resume, begin, and end callback functions are described in more detail in the section “Application Defined Protocol Functions”.

WARNING

You must allocate a separate exception stack for each thread that uses exception handling. You must also use suspend and resume callback functions in order to save and restore the exception handling environment. ◆

You must also be careful to prevent exceptions from propagating beyond a thread's entry point. For instance, a thread's entry point could be written as follows.

```
void thread(ThreadDataType data)
{
  TRY {
    /* ... do stuff ... */
  } CATCH {
    /* ... cleanup ... */
    NOPROPAGATE;
  } ENDTRY;
}
```

When an exception is raised in the thread, it will be prevented from propagating beyond the thread's entry point by the NOPROPAGATE statement. If exceptions were allowed to propagate beyond the entry point to the thread, they would cause the application to behave in an undefined manner.

WARNING

Exceptions must not propagate beyond a thread's entry point. ◆

Debugging

When you first use Thread Library, you should disable all optimizations and should link with the library "ThreadLibrary-68K-Debug.o" or "ThreadLibrary-PPC-Debug.o". These libraries were compiled with all of the debug code enabled (the preprocessor symbol THREAD_DEBUG was defined as 1). Thread Library includes numerous assertions intended to catch run-time errors. Once you know that threads work with your application, you can enable compiler optimizations and test your application again to make sure it still runs.

The debug code will make Thread Library execute significantly slower than its optimal speed. Since the debug code does not alter the functional specification of Thread Library, you could ship a final version of your program with the debug code enabled, but the code that uses threads would run slower than it would if the the debug code were disabled. It is unfortunate that debug code must be disabled to achieve maximum performance. Personally, I would like to leave debug code enabled, though modifying it to return an error code or raise an exception instead of breaking into the debugger. To disable the debug code, you should link your application with one of the libraries "ThreadLibrary-68K.o" or "ThreadLibrary-PPC.o", which were compiled with the preprocessor symbol THREAD_DEBUG defined as 0.

A stack overflow will often result in a corrupted heap, since the stack is allocated as a nonrelocatable block in the heap (when using the default memory allocator) and overflow usually overwrites the block's header. For this reason, you may be able to detect stack overflow by enabling a heap-check option in a low-level debugger such as TMON or MacsBug.

You can use source-level debuggers to debug applications that use Thread Library. However, you should not attempt to step through the context switching code in Thread Library, as this may cause your debugger to crash. This problem is only likely to occur if you include the file "ThreadLibrary.c" in your application (rather than using one of the compiled libraries) and you also attempt to step into the context switching code.

WARNING

Using the THINK C or Metrowerks debuggers to trace through context switches may result in corruption of the application's heap followed by a crash. The problem occurs if you place a breakpoint or try to step too close to the register restoring code that accomplishes the context switch in the function ThreadActivatePtr in the file "ThreadLibrary.c". I have successfully used TMON Professional to trace through the context switches, and other low-level debuggers (like MacsBug) should also work. ◆

Profiling

Profiling an application that uses threads presents problems similar to those described for exception handling. In a nutshell, the problem is that profilers tend to maintain a single stack of function calls for the application. But a multi-threaded application will have more than one stack of function calls. This means that you will probably not be able to profile an application that uses threads unless you modify the profiler. For instance, the profiler supplied with THINK C versions 5.0.4 and 7.0.x will not work properly when threads are used.

I have modified the THINK C profiler to work with threads, but cannot redistribute the modified profiler due to Symantec's copyright in the original source code. In brief, THINK C's profiler maintains a stack of called functions. This stack must be swapped

for each thread by using custom suspend and resume callback functions. Custom begin and end callback functions can be used to allocate and dispose of the profiler's stack for the thread. The most important modifications to the THINK C profiler consisted of placing all of the global data maintained by the profiler into a single structure, and using a single global variable to point to the current profiler data. The begin and end callback functions for my threads allocate and deallocate a new profiler data structure for each thread, and the suspend and resume functions for my threads save and restore the value of the profiler's global variable.

WARNING

When profiling your code, you may need to allocate a separate profiling stack for each thread. You must also use suspend and resume callback functions to save and restore the profiler's stack. This may require making modifications to the profiler supplied with your development environment. ◆

Thread Library Manager

Apple's Thread Manager already provides a popular method of implementing threads. It might seem that you would have to decide, before even using threads, whether you would support Apple's Thread Manager or my Thread Library in your application. However, included with Thread Library is another small library called Thread Library Manager (TLM). TLM provides an interface to Thread Library that mimics the interface provided by Apple's Thread Manager. The interface to TLM is the same as the interface to version 2.0.1 of Apple's Thread Manager, except that the names of the routines in TLM all start with the letters "TLM". For instance, the routine TLMNewThread creates a new thread using Thread Library, while TLMYieldToAnyThread allows other threads to execute.

Using TLM has several benefits over using Thread Library directly:

- you only have to know one set of interfaces to implement threads in any Macintosh application;
- if you already know how to use Thread Manager, you will not need to learn much additional information to use Thread Library;
- it is relatively easy to switch your code between use of Thread Manager and Thread Library.

Using Thread Library Manager

To use TLM you need to include the file "ThreadLibraryManager.h" in every source file that calls Thread Manager routines. This file contains macros that define the Thread Manager routine names to TLM routine names. You also need to link the appropriate Thread Library binary with you application.

For an example of how easy it is to use TLM instead of Thread Manager, I was able to add TLM to Apple's ThreadedSort sample program, which is distributed with Thread Manager 2.1, by:

1. adding the folder containing Thread Library to the access path for the ThreadedSort CodeWarrior project files;

2. adding the file "ThreadLibrary-PPC.o" to the PPC CodeWarrior project and the file "ThreadLibrary-68K.o" to the 68K CodeWarrior project.

3. adding the following line to the file "Sprocket.h":

```
#include "ThreadLibraryManager.h"
```

4. recompiling.

Once recompiled, the application ran without a glitch.

Differences From Thread Manager

You can read Apple's published documentation for a complete specification of Thread Manager. Since TLM is nearly identical to Thread Manager, there is no need to describe TLM in detail. The differences between Thread Manager and TLM are:

- TLM does not support preemptive threads. In this respect, TLM operates like the PowerPC version of Thread Manager. For instance, calling TLMNewThread with a thread style parameter of kPreemptiveThread returns the error threadProtocolError.
- TLM only saves registers that must be saved across function calls, while Thread Manager saves virtually all user-mode registers. This should not pose a problem, however, since TLM can only be used with cooperative threads, which always yield the processor by calling a function.
- TLM does not support thread pools. This is no great loss, since TLM does not support preemptive threads. The routine TLMCreateThreadPool always returns noErr, but does not create a thread pool. The routine TLMGetFreeThreadCount always returns a count of zero and a result of noErr. The routine TLMGetSpecificThreadCount always returns a count of zero and a result of noErr.
- The flags kUsePremadeThread, kCreateIfNeeded, kFPUNotNeeded, and kExactMatchThread in the options parameter to TLMNewThread are ignored.
- The state of the floating point registers are always saved, both on 68K based Macintoshes that have a floating point unit and on PowerPC based Macintoshes.

If you use TLM, then you should not call Thread Library directly. For instance, the IDs used to refer to threads in TLM are not the same as the serial numbers used to refer to threads in Thread Library.

Predicates

The descriptions of functions in this document use preconditions and postconditions expressed as mathematical predicates. The preconditions and postconditions are not a complete formal specification of the operation of the functions and their impact on the remainder of the application. A complete formal specification would require more extensive work and use of a formal specification language such as Z. The preconditions and postconditions do, however, provide a reasonable statement on the operation of the functions.

A function call can succeed if all of the predicates in its preconditions are true, and it will attempt to ensure that all of the predicates in its postconditions are true when it has finished executing. If a function is unable to ensure that its postconditions are true, then it will set an error indicator, which the caller can later access. An error indicator

may be set even if the function was able to return a value allowed by its postconditions; for instance, the error memFullErr might be set when the function ThreadBegin fails to create a new thread. Since any function in Thread Library may set the error indicator, the following postcondition is implicitely assumed for every function:

ThreadError'() = THREAD_ERROR_NONE ⇔ function was successful

where ThreadError'() is the value returned by calling ThreadError after the function has been executed.

Since there is no tool to verify the implementation against the predicates, it is quite possible that there are errors in the predicates. This is also the first time that I am using mathematical predicates to describe the interface to a library, and I may have inadvertantly made some mistakes in their use. Any ambiguity or contradiction in the description of a function and its predicates may be resolved by examining the source code for Thread Library.

Notation

The predicates use standard set notation, though occasional English statements are used where appropriate. Following are the few adaptations I made to simplify the syntax of the predicates.

- Multiple predicates are and'ed together. For instance, the predicates

  a = 1.

  b = 2.

  are equivalent to

  a = 1 ^ b = 2.

- Multiple predicates may be grouped together using square brackets. For instance,

  a = 1 ⇒ [b = 2. c = 3.].

- Function composition is defined as

  $f^n(x)$ =

  $f^n(x) = x$ if n = 0,

  $f^n(x) = f(f^{n-1}(x))$ if n > 0.

- Variables named in the parameter list to a function are accessible in the function's preconditions and postconditions. The character ' is used to refer to the value of an object (variable, set, or function) after the function has executed; the original value is indicated by omitting the ' character. For instance, the predicate

  ThreadCount'() = ThreadCount() + 1

  states that the value returned by ThreadCount after the function has executed is one greater than the value of ThreadCount before the function was executed. This notation is used in the specification language Z.

- The predefined variable result contains the result of a function, and is defined only within a function's postconditions. For instance, the predicate

  result = x^2

  could be used to describe the C language function

```
double square(double x) { return(x * x); }
```

The use of a special variable to hold the result of a function is derived from the language Eiffel.

Definitions

Several predefined constants and sets are used in the predicates.

The set of Boolean values is

$$Boolean \equiv \{ 0, 1 \}.$$

The set of possible serial numbers is

$$SN \equiv \{ n : integer \mid n \neq THREAD\_NONE \}.$$

The set of possible error numbers is

$$EN \equiv \{ n : integer \mid n \neq THREAD\_ERROR\_NONE \}.$$

The set of all serial numbers assigned to threads is

$$TAssigned = subset\ of\ SN$$

where the set TAssigned is initially empty, and elements are added to it by the thread creation functions and removed from it by the thread destruction functions.

The set of all serial numbers that have been assigned is

$$TAssignedAll = \ subset\ of\ SN$$

where the set TAssignedAll is initially empty, and elements are added to it by the thread creation functions. Elements are never removed from the set TAssignedAll. An immediate corollary is that TAssigned is a subset of TAssignedAll.

The variable TMemoryAvailable is defined as the amount of memory available for use by Thread Library. Notice that this value may differ from the amount of memory available for use by the application, which is denoted by AppMemoryAvailable.

It is possible to formally define some of the types used in Thread Library. For instance,

$$ThreadSizeType \equiv \{ n : integer \mid 0 \leq n \leq THREAD\_SIZE\_MAX \}$$

$$ThreadType \equiv (SN \cup \{ THREAD\_NONE \}) = \{ n: integer \mid -\infty < n < \infty \}$$

Such formal definitions, however, would only be necessary in a full specification language, and are therefore omitted from the predicates.

Note

The special types and sets defined in this section are not part of Thread Library, and are used only within this document. ◆

Functional Interface

This section describes the functional interface to Thread Library. Every function's description is divided into the sections name, syntax, parameters, returns, preconditions, postconditions, and description.

name The name of the function, in boldface text.

syntax	The syntax of the function (types of parameters and type of return value, if any).
parameters	Short English descriptions of the parameters to the function. This section may be omitted.
returns	Short English description of the value returned by the function. This section may be omitted.
preconditions	A set of mathematical predicates that specify the conditions that must be true for a call to the function to succeed.
postconditions	A set of mathematical predicates that specify the conditions that will be true following a successful call to the function.
description	An English description of the operation of the function.

The preconditions and postconditions may be skipped, as the syntax and description sections of the functions provide sufficient information for their use.

Error Handling

The functions described in this section provide support for detection of errors encountered while executing other functions in Thread Library.

ThreadError

```
ThreadErrorType ThreadError(void)
```

Preconditions

No special preconditions.

Postconditions

result ∈ (EN ∪ { THREAD_ERROR_NONE }).

result = THREAD_ERROR_NONE ⇔ last operation was successful.

Description

ThreadError returns THREAD_ERROR_NONE if the last function in Thread Library completed successfully, or an error number corresponding to the reason that the function failed. ThreadError does not itself modify the error code, so consecutive calls to ThreadError will return the same value, provided no intervening calls to other functions in Thread Library are made.

You should call ThreadError after every call to a function in ThreadLibrary. If an error code is returned, then the function was unable to fulfill your request and you should take appropriate actions to recover from the error or to report the error to the user.

Typical errors occur when:

- Thread Library is unable to acquire some resource, such as memory for a new thread;
- an invalid serial number is passed to a Thread Library function;
- one or more of the preconditions to a Thread Library function is violated by the caller, such as attempting to terminate the main thread before all other threads have been terminated (such error detection might only be available if debug code has been enabled).

ThreadErrorSet

```
void ThreadErrorSet(ThreadErrorType error)
```

Preconditions

error ∈ (EN ∪ { THREAD_ERROR_NONE }).

Postconditions

ThreadError'() = error.

Description

ThreadErrorSet sets the value that will be returned by a subsequent call to ThreadError. Interveninig calls to other functions in Thread Library may alter the value returned by ThreadError.

You usually will not need to call ThreadErrorSet. You might use this function, for instance, from a special purpose memory allocator in which you may need to indicate that memory could not be allocated and that Thread Library should abort the operation that attempted to allocate the memory.

Thread Creation and Destruction

This section describes the functions that are used to create and destroy threads.

ThreadBegin

```
ThreadType ThreadBegin(ThreadProcType entry, ThreadProcType
suspend, ThreadProcType resume, ThreadDataType data,
ThreadSizeType stack_size)
```

Parameters

entry	a pointer to a function that is called to start executing the thread.
suspend	a pointer to a function called whenever the thread is suspended. You can use the suspend function to save application-defined context for the thread.
resume	a pointer to a function called whenever the thread is resumed. You can use the resume function to restore application-defined context for the thread.
data	passed to the entry, suspend, and resume functions and may contain a pointer to any application-defined data.
stack_size	specifies the size of the stack needed by the thread.

The entry, suspend, and resume callback functions are described in more detail in the section “Application Defined Protocol Functions”.

Returns

The serial number of a new thread, or THREAD_NONE if a new thread could not be created.

Preconditions

TAssignedAll ≠ ∅.

entry = valid entry function.

suspend ∈ { NULL , valid suspend function }.

resume ∈ { NULL , valid resume function }.

data ∈ { NULL , valid pointer }.

0 ≤ stack_size ≤ THREAD_SIZE_MAX.

Postconditions

result ∉ TAssignedAll.

result ∈ (SN ∪ { THREAD_NONE }).

result ≠ THREAD_NONE ⇔

[ThreadCount'() = ThreadCount() + 1.

TAssigned' = (TAssigned ∪ { result }).

TAssignedAll' = (TAssignedAll ∪ { result }).

$ThreadNext^{ThreadCount'()-1}(ThreadFirst'()) = result.$]

result ≠ THREAD_NONE ⇒

[ThreadEnabled(result) = true.

ThreadWakeTime(result) = 0.

ThreadProcBegin(result) = NULL.

ThreadProcEnd(result) = NULL.

ThreadProcEntry(result) = entry.

ThreadProcSuspend(result) = suspend.

ThreadProcResume(result) = resume.

ThreadData(result) = data.

ThreadStackSize(result) = stack_size.

TMemoryAvailable' ≤ TMemoryAvailable -

ThreadMemorySize(0, THREAD_TYPE_THREAD) -

ThreadMemorySize(stack_size, THREAD_TYPE_STACK).]

Description

ThreadBegin creates a new thread and returns the thread's serial number. You must create the main thread with ThreadBeginMain before you can call ThreadBegin.

ThreadBegin returns immediately after creating the new thread. The thread, however, is not executed immediately. It is added to the end of the queue of threads, and will be selected for execution by ThreadSchedule according to the scheduling algorithm. When the thread is executed, the function specified in the entry parameter is called. When the function has returned, the thread is removed from the queue of threads and its stack and any private storage allocated by ThreadBegin are discarded.

The requested stack size should be large enough to contain all function calls, local variables and parameters, and any operating system routines that may be called while the thread is active (including interrupt driven routines). If the requested stack size is zero, then the default stack size returned by ThreadStackDefault is used.

It is a good idea to set the stack size to at least the value returned by ThreadStackMinimum; otherwise, your application is likely to crash somewhere inside the operating system. If your thread crashes, try increasing the thread's stack size.

WARNING

Your application may crash if you do not specify a large enough size for a thread's stack. ◆

ThreadBeginMain

```
ThreadType ThreadBeginMain(ThreadProcType suspend, ThreadProcType
resume, ThreadDataType data)
```

suspend	a pointer to a function called whenever the thread is suspended. You can use the suspend function to save application-defined context for the thread.
resume	a pointer to a function called whenever the thread is resumed. You can use the resume function to restore application-defined context for the thread.
data	passed to the entry, suspend, and resume functions and may contain a pointer to any application-defined data.

The suspend and resume callback functions are described in more detail in the section “Application Defined Protocol Functions”.

Returns

The serial number of the main thread, or THREAD_NONE if the main thread could not be created.

Preconditions

TAssignedAll = ∅.

suspend ∈ { NULL , valid suspend function }.

resume ∈ { NULL , valid resume function }.

data ∈ { NULL , valid pointer }.

0 ≤ stack_size ≤ THREAD_SIZE_MAX.

Postconditions

result ∉ TAssignedAll.

result ∈ (SN ∪ { THREAD_NONE }).

result ≠ THREAD_NONE ⇔

[ThreadCount'() = 1.

TAssigned' = { result }.

TAssignedAll' = { result }.

ThreadMain'() = ThreadActive'() = ThreadFirst'() = result.]

result ≠ THREAD_NONE ⇒

[ThreadEnabled(result) = true.

ThreadWakeTime(result) = 0.

ThreadProcSuspend(result) = suspend.

ThreadProcResume(result) = resume.

ThreadData(result) = data.

TMemoryAvailable' ≤ TMemoryAvailable -

ThreadMemorySize(0, THREAD_TYPE_THREAD).]

Description

ThreadBeginMain creates the main application thread and returns the main thread's serial number. You must call this function before creating any other threads with ThreadBegin. You must also call MaxApplZone before calling this function. The resume, suspend, and data parameters have the same meanings as the parameters to ThreadBegin.

There are several important differences between the main thread and all subsequently created threads.

- The main thread is responsible for handling events sent to the application, and is therefore scheduled differently from other threads; see the function ThreadSchedule for details.
- The main thread uses the application's stack and context; no private stack is allocated for the main thread. Initially, there is therefore no need to change the context to start executing the thread, and no special entry point is required. But, like all other threads, the main thread's context will be saved whenever it is suspended to allow another thread to execute, and its context will be restored when it is resumed.
- While other threads do not begin executing until they are scheduled to execute, the main thread is designated as the active thread as soon as ThreadBeginMain returns.
- Since other threads have a special entry point, they are automatically disposed of when that entry point returns. The main thread, lacking any special entry point, must be disposed of by the application. You should call ThreadEnd, passing it the thread returned by ThreadBeginMain or ThreadMain, before exiting your application.

ThreadEnd

```
void ThreadEnd(ThreadType thread)
```

Preconditions

thread ∈ (TAssigned ∪ { THREAD_NONE }).

Postconditions

thread ∉ TAssigned'.

∀ n: integer • 0 ≤ n < ThreadCount'() ⇒ ThreadNext'^{n}(ThreadFirst'()) ≠ thread.

thread ≠ THREAD_NONE ⇒

[ThreadCount'() = ThreadCount() - 1.

TAssigned' = (TAssigned \ { thread }).

thread = ThreadMain() ⇒

[TMemoryAvailable' ≥

TMemoryAvailable + ThreadMemorySize(0, THREAD_TYPE_THREAD).

ThreadMain'() = ThreadActive'() = ThreadFirst'() = THREAD_NONE.]

thread ≠ ThreadMain() ⇒

[thread = ThreadActive() ⇒ ThreadActive'() = ThreadSchedule'().

TMemoryAvailable' ≥ TMemoryAvailable +

ThreadMemorySize(0, THREAD_TYPE_THREAD) +

ThreadMemorySize(ThreadStackSize(thread), THREAD_TYPE_STACK).]]

Description

ThreadEnd removes the thread from the queue of threads and disposes of the memory allocated for the thread. If the thread is the active thread, then it is deactivated and the next scheduled thread is activated. The main thread cannot be disposed of until all other threads have been discarded. ThreadEnd is called automatically when the entry point of a thread returns, so there is often no need to explicitely call ThreadEnd.

ThreadEndAll

```
void ThreadEndAll(void)
```

Preconditions

ThreadActive() = ThreadMain().

Postconditions

TAssigned' = ∅.

ThreadCount'() = 0.

ThreadMain'() = ThreadActive'() = ThreadFirst'() = THREAD_NONE.

Description

ThreadEndAll disposes of all threads, including the main thread. ThreadEndAll is useful when your application is terminating and you want to dispose of any threads that may still exist. ThreadEndAll can be called only from within the main thread.

Scheduling

The functions described in this section control the scheduling and activation of threads. The three functions ThreadSchedule, ThreadActivate, and ThreadYield handle the scheduling and context switching of threads. These functions will likely be executed the most often of any of the functions in Thread Library, and therefore will have the greatest impact on its efficiency. If you find Thread Library's context switches too slow, you could improve the efficiency of these functions.

ThreadYield

```
void ThreadYield(ThreadTicksType sleep)
```

Preconditions

TAssigned ≠ ∅.

0 ≤ sleep ≤ THREAD_TICKS_MAX.

Postconditions

$0 \leq \mid \text{ThreadWakeTime'(ThreadActive())} - (t + \text{sleep} + \delta) \mid \leq \varepsilon$;

$t \equiv$ time at which ThreadYield was called;

$\delta \equiv$ delay to execute ThreadWakeTimeSet.

Description

ThreadYield activates the next scheduled thread as determined by ThreadSchedule. The sleep parameter has the same meaning as the parameter to ThreadSleepIntervalSet.

Note

See the note for ThreadActivate. ◆

ThreadYieldInterval

```
ThreadTicksType ThreadYieldInterval(void)
```

Preconditions

ThreadCount() > 0.

Postconditions

$0 \leq$ result $\leq$ THREAD_TICKS_MAX.

Description

ThreadYieldInterval returns the maximum time until the next call to ThreadYield. The interval is computed by subtracting the current time from each thread's wake time, giving the amount of time that each thread can remain inactive. The minimum of these times gives the maximum amount of time until the next call to ThreadYield. The wake time of the current thread is ignored, since the thread is already active. You can use the returned value to help determine the maximum sleep value to pass to WaitNextEvent.

ThreadSchedule

```
ThreadType ThreadSchedule(void)
```

Preconditions

TAssigned $\neq \varnothing$.

Postconditions

result $\in$ TAssigned.

ThreadEnabled(result) = true.

result $\neq$ ThreadMain() $\Rightarrow$ ThreadWakeTime(result) $\leq t$;

$t \equiv$ time at which ThreadSchedule returns.

Description

ThreadSchedule returns the next thread to activate. Threads are maintained in a queue and are scheduled in a round-robin fashion. Starting with the active thread, the queue of threads is searched for the next enabled thread whose wake time has arrived. The first such thread found is returned.

In addition to the round-robin scheduling shared with all threads, the main thread will also be activated if any events are pending in the event queue. The application can

then immediately handle the events, allowing the application to remain responsive to user actions such as mouse clicks. The main thread will also be activated if no other threads are scheduled for activation, which allows the application to either continue with its main processing or to call WaitNextEvent and sleep until a thread needs to be activated or some other task or event needs to be handled.

ThreadActivate

```
void ThreadActivate(ThreadType thread)
```

Preconditions

thread ∈ TAssigned.

ThreadEnabled(thread) = true.

Postconditions

$ThreadNext^{ThreadCount()-1}(ThreadFirst'()) = ThreadActive() = thread$

Description

ThreadActivate deactivates the currently active thread and makes the specified thread the active thread. The activated thread is moved to the end of the queue of threads.

Note

ThreadActivate does not return to its caller until the activated thread yields the processor and the caller's thread is reactivated. Also, any number of additional calls to functions in Thread Library may be made before ThreadActivate returns to its caller; these function calls could change data shared amongst all threads, such as TAssigned. In fact, the function ThreadActivate might never even return to its caller. This means that the set of postconditions given above is incomplete. ◆

ThreadEnabled

```
Boolean ThreadEnabled(ThreadType thread)
```

Preconditions

thread ∈ TAssigned.

Postconditions

result ∈ Boolean.

Description

ThreadEnabled returns true if the thread is enabled. An enabled thread is eligible for execution by the scheduler, while a disabled thread is not scheduled for execution. Threads are enabled when they are created, and remain enabled unless you explicitely disable them with ThreadEnabledSet.

ThreadEnabledSet

```
void ThreadEnabledSet(ThreadType thread, Boolean enabled)
```

Preconditions

ThreadMain() ∈ TAssigned

thread ∈ (TAssigned \ { ThreadMain() }).

enabled ∈ Boolean.

Postconditions

ThreadEnabled'(thread) = enabled.

Description

ThreadEnabledSet enables or disables the thread. A disabled thread is not eligible for scheduling by ThreadSchedule. If called for the active thread, the thread remains active until the next call to ThreadYield or ThreadActivate. You cannot disable the main thread, since there must always be a default thread that can be activated when no other threads are available.

ThreadWakeTime

```
ThreadTicksType ThreadWakeTime(ThreadType thread)
```

Preconditions

thread ∈ TAssigned.

Postconditions

0 ≤ result ≤ THREAD_TICKS_MAX.

Description

ThreadWakeTime returns the time when the specified thread will become eligible for scheduling. The wake time is typically determined indirectly by the value passed to ThreadYield.

ThreadWakeTimeSet

```
void ThreadWakeTimeSet(ThreadType thread, ThreadTicksType wake)
```

Parameters

sleep specifies when the thread will become eligible for scheduling.

Preconditions

thread ∈ TAssigned.

0 ≤ wake ≤ THREAD_TICKS_MAX.

Postconditions

ThreadWakeTime'(thread) = wake.

Description

ThreadWakeTimeSet sets the time when the specified thread will become eligible for scheduling. The wake time is typically determined indirectly by the value passed to ThreadYield.

ThreadSleepSet

```
void ThreadSleepSet(ThreadType thread, ThreadTicksType sleep)
```

Description

This is a synonym for the function ThreadSleepIntervalSet. This function is included for compatability with prior versions of Thread Library.

ThreadSleepIntervalSet

```
void ThreadSleepIntervalSet(ThreadType thread, ThreadTicksType
sleep)
```

Parameters

sleep	specifies the maximum amount of time that the thread can remain inactive.

Preconditions

thread ∈ TAssigned.

0 ≤ sleep ≤ THREAD_TICKS_MAX.

Postconditions

0 ≤ | ThreadWakeTime'(thread) - (t + sleep + δ) | ≤ ε;

t ≡ time at which ThreadSleepIntervalSet was called;

δ ≡ delay to execute ThreadWakeTimeSet.

Description

ThreadSleepIntervalSet sets the amount of time that the specified thread will remain inactive. The larger the sleep value, the more time is available for execution of other threads. When called from the main thread, you can pass a sleep parameter equal to the maximum interval between null events; if no null events are needed, you can pass a sleep value of THREAD_TICKS_MAX. The main thread will continue to receive processing time whenever an event is pending and when no other threads are scheduled (see ThreadSchedule). If the thread is already active, the sleep time specified will be used when the thread is inactive and is thus eligible for scheduling by ThreadSchedule. ThreadSleepIntervalSet is normally called by ThreadYield, but you may need to use it if you call ThreadSchedule or ThreadActivate.

ThreadSleepIntervalSet is a convenient way to call ThreadWakeSet when you want to delay execution of a thread for some amount of time. For instance, the following call to ThreadSleepIntervalSet,

```
ThreadSleepIntervalSet(sleep)
```

is approximately equivalent to

```
ThreadWakeSet(time + sleep)
```

The only difference of note is that the calculation `time + sleep` made in ThreadSleepIntervalSet takes care to avoid arithmetic overflow, which could result, for instance, from a sleep value of THREAD_TICKS_MAX.

Accessing the Queue of Threads

All threads created by the application are kept in a circular queue of threads. A circular queue is used to provide fair round-robin scheduling to all threads. You can use the functions described in this section to access all of the threads your application has created. Using these functions, and the function ThreadActivate, you could, for instance, implement your own scheduler to replace the functions ThreadSchedule and ThreadYield.

An invariant relating ThreadCount, ThreadFirst, ThreadNext, and TAssigned is

∀ t: thread • t ∈ TAssigned ⇔

∃ n: integer • 0 ≤ n < ThreadCount() ⇒ ThreadNext^{n}(ThreadFirst()) = t.

ThreadCount

```
long ThreadCount(void)
```

Preconditions

No special preconditions.

Postconditions

$0 \leq$ result

result = $0 \Leftrightarrow$ TAssigned = $\emptyset$.

Description

ThreadCount returns the number of threads in the queue, which is equivalent to the number of elements in the set TAssigned.

ThreadMain

```
ThreadType ThreadMain(void)
```

Preconditions

No special preconditions.

Postconditions

result $\in$ (TAssigned $\cup$ { THREAD_NONE }).

result = THREAD_NONE $\Leftrightarrow$ TAssigned = $\emptyset$.

Description

ThreadMain returns the main thread, or THREAD_NONE if there are no threads.

ThreadActive

```
ThreadType ThreadActive(void)
```

Preconditions

No special preconditions.

Postconditions

result $\in$ (TAssigned $\cup$ { THREAD_NONE }).

result = THREAD_NONE $\Leftrightarrow$ TAssigned = $\emptyset$.

Description

ThreadActive returns the currently active thread, or THREAD_NONE if there are no threads.

ThreadFirst

```
ThreadType ThreadFirst(void)
```

Preconditions

No special preconditions.

Postconditions

result $\in$ (TAssigned $\cup$ { THREAD_NONE }).

result = THREAD_NONE $\Leftrightarrow$ TAssigned = $\emptyset$.

Description

ThreadFirst returns the first thread in the queue of threads, or THREAD_NONE if there are no threads.

ThreadNext

```
ThreadType ThreadNext(ThreadType thread)
```

Preconditions

thread ∈ (TAssigned ∪ { THREAD_NONE }).

Postconditions

result ∈ (TAssigned ∪ { THREAD_NONE }).

result = THREAD_NONE ⇔ thread = THREAD_NONE.

result = thread ⇔ ThreadCount() ≤ 1.

Description

ThreadNext returns the next thread in the circular queue of threads, or THREAD_NONE if there are no threads.

Since the queue is circular, and since ThreadCount gives the number of items in the queue, it is always true that

$\text{ThreadNext}^{\text{ThreadCount}()}(\text{thread}) = \text{thread}$.

Information About the Stack

The functions described in this section can be used to help determine the size of a new thread's stack and provide information about the space available in an existing thread's stack.

ThreadStackMinimum

```
ThreadSizeType ThreadStackMinimum(void)
```

Preconditions

No special preconditions.

Postconditions

0 ≤ result ≤ ThreadStackDefault().

Description

ThreadStackMinimum returns the recommended minimum stack size for a thread. Thread Library does not enforce a lower limit on the stack size, but it is a good idea to allow at least the number of bytes returned by ThreadStackMinimum for a thread's stack.

ThreadStackDefault

```
ThreadSizeType ThreadStackDefault(void)
```

Preconditions

No special preconditions.

Postconditions

ThreadStackMinimum() ≤ result ≤ THREAD_SIZE_MAX.

Description

ThreadStackDefault returns the default stack size for a thread. This is the amount of stack space reserved for a thread if a zero stack size is passed to ThreadBegin.

ThreadStackSize

```
ThreadSizeType ThreadStackSize(ThreadType thread);
```

Preconditions

thread ∈ TAssigned.

Postconditions

0 ≤ result ≤ THREAD_SIZE_MAX.

Description

ThreadStackSize returns the amount of space, in bytes, allocated for the thread's stack. For the main thread, this is the space allocated by the Process Manager for the application's stack (and possibly adjusted by the application). For all other threads, this is the same as the stack_size parameter passed to ThreadBegin when the thread was created.

ThreadStackSpace

```
ThreadSizeType ThreadStackSpace(ThreadType thread)
```

Preconditions

thread ∈ TAssigned.

Postconditions

0 ≤ result ≤ ThreadStackSize(thread).

Description

ThreadStackSpace returns the amount of stack space remaining in the specified thread. There are at least the returned number of bytes between the thread's stack pointer and the bottom of the thread's stack, though slightly more space may be available to the application due to overhead from Thread Library.

IMPORTANT

The trap StackSpace will return incorrect results if called from any thread other than the main thread. Likewise, using the low-memory globals ApplLimit, HeapEnd, or CurStackBase to determine the bounds of a thread's stack will produce incorrect results when used outside of the main thread. Instead of calling StackSpace, use ThreadStackSpace to determine the amount of free stack space in a thread. ◆

Application Defined Data

The functions described in this section provide access to the application-defined data for threads. You can use a thread's application-defined data pointer to store a pointer to any data that your application may want to access. The data pointer is especially useful for providing contextual information to callback functions, such as suspend and resume callbacks, that you might specify for a thread.

ThreadData

```
ThreadDataType ThreadData(ThreadType thread)
```

Preconditions

thread ∈ TAssigned.

Postconditions

result ∈ { NULL , valid pointer }.

Description

ThreadData returns the application-defined data pointer of the thread.

ThreadDataSet

```
void ThreadDataSet(ThreadType thread, ThreadDataType data)
```

Preconditions

thread ∈ TAssigned.

data ∈ { NULL , valid pointer }.

Postconditions

ThreadData'(thread) = data.

Description

ThreadDataSet sets the application-defined data pointer of the thread.

Accessing Application Defined Memory Allocation Functions

This section describes the functions you can use to access the application-defined memory allocation functions. The memory allocation functions are described in more detail in the section “Application Defined Memory Allocation Functions”.

ThreadProcAllocate

```
ThreadProcAllocateType ThreadProcAllocate(void)
```

Preconditions

No special preconditions.

Postconditions

result ∈ { NULL, address of memory allocation function }.

Description

ThreadProcAllocate returns the function used to allocate memory, or NULL if the default function is being used.

ThreadProcAllocateSet

```
void ThreadProcAllocateSet(ThreadProcAllocateType alloc)
```

Parameters

alloc pointer to application-defined memory allocation function.

Preconditions

alloc ∈ { NULL, address of memory allocation function }.

Postconditions

ThreadProcAllocate'() = alloc.

Description

ThreadProcAllocateSet sets the function used to allocate memory. If you specify NULL, then the default function will be used.

ThreadProcDispose

```
ThreadProcDisposeType ThreadProcDispose(void)
```

Preconditions

No special preconditions.

Postconditions

result ∈ { NULL, address of memory disposal function }.

Description

ThreadProcDispose returns the function used to dispose of memory, or NULL if the default function is being used.

ThreadProcDisposeSet

```
void ThreadProcDisposeSet(ThreadProcDisposeType dispose)
```

Parameters

dispose pointer to application-defined memory disposal function.

Preconditions

dispose∈ { NULL, address of memory disposal function }.

Postconditions

ThreadProcDispose'() = dispose.

Description

ThreadProcDisposeSet sets the function used to dispose of memory. If you specify NULL, then the default function will be used.

Accessing Application Defined Protocol Functions

This section describes the functions that provide access to the application-defined protocol functions for a thread. The protocol functions are described in more detail in the section “Application Defined Protocol Functions”.

ThreadProcBegin

```
ThreadProcBeginEndType ThreadProcBegin(ThreadType thread)
```

Preconditions

ThreadMain() ∈ TAssigned

thread ∈ (TAssigned \ { ThreadMain() }).

Postconditions

result ∈ { NULL, address of begin function }.

Description

ThreadProcBegin returns the begin function for the thread.

ThreadProcBeginSet

```
void ThreadProcBeginSet(ThreadType thread, ThreadProcBeginEndType
begin)
```

Preconditions

ThreadMain() ∈ TAssigned

thread ∈ (TAssigned \ { ThreadMain() }).

begin ∈ { NULL, address of begin function }.

Postconditions

ThreadProcBegin'(thread) = begin.

Description

ThreadProcBeginSet sets the begin function for the thread.

ThreadProcEnd

```
ThreadProcBeginEndType ThreadProcEnd(ThreadType thread)
```

Preconditions

ThreadMain() ∈ TAssigned

thread ∈ (TAssigned \ { ThreadMain() }).

Postconditions

result ∈ { NULL, address of end function }.

Description

ThreadProcEnd returns the end function for the thread.

ThreadProcEndSet

```
void ThreadProcEndSet(ThreadType thread, ThreadProcBeginEndType
end)
```

Preconditions

ThreadMain() ∈ TAssigned

thread ∈ (TAssigned \ { ThreadMain() }).

end ∈ { NULL, address of end function }.

Postconditions

ThreadProcEnd'(thread) = end.

Description

ThreadProcEndSet sets the end function for the thread.

ThreadProcResume

```
ThreadProcType ThreadProcResume(ThreadType thread)
```

Preconditions

thread ∈ TAssigned.

Postconditions

result ∈ { NULL, address of resume function }.

Description

ThreadProcResume returns the resume function for the thread.

ThreadProcResumeSet

```
void ThreadProcResumeSet(ThreadType thread, ThreadProcType resume)
```

Preconditions

thread ∈ TAssigned.

resume ∈ { NULL, address of resume function }.

Postconditions

ThreadProcResume'(thread) = resume.

Description

ThreadProcResumeSet sets the resume function for the thread.

ThreadProcSuspend

```
ThreadProcType ThreadProcSuspend(ThreadType thread)
```

Preconditions

thread ∈ TAssigned.

Postconditions

result ∈ { NULL, address of suspend function }.

Description

ThreadProcSuspend returns the suspend function for the thread.

ThreadProcSuspendSet

```
void ThreadProcSuspendSet(ThreadType thread, ThreadProcType
suspend)
```

Preconditions

thread ∈ TAssigned.

suspend ∈ { NULL, address of suspend function }.

Postconditions

ThreadProcSuspend'(thread) = suspend.

Description

ThreadProcSuspendSet sets the suspend function for the thread.

ThreadProcEntry

```
ThreadProcType ThreadProcEntry(ThreadType thread)
```

Preconditions

ThreadMain() ∈ TAssigned

thread ∈ (TAssigned \ { ThreadMain() }).

Postconditions

result ≠ NULL

Description

ThreadProcEntry returns the entry function for the thread.

ThreadProcEntrySet

```
void ThreadProcEntrySet(ThreadType thread, ThreadProcType entry)
```

Preconditions

thread ∈ (TAssigned \ { ThreadMain() }).

entry = address of entry function.

Postconditions

ThreadProcEntry'(thread) = entry.

Description

ThreadProcEntrySet sets the entry function for the thread.

Application Defined Protocol Functions

The application can install functions that are called at specific times during a thread's lifetime. Each thread can have its own set of unique functions, which can be unrelated to the functions installed for any other thread. The functions are: begin, end, resume, suspend, and entry. Of these functions, the main thread uses only the resume and suspend functions. All other threads can use all of the functions, though only the entry function is required.

Call Sequence of Protocol Functions

Over the lifetime of a thread, the following sequence of function calls is made:

```
begin
resume
entry
(suspend, resume)0..N
suspend
end
```

First, the begin function is called. Then, the resume function is called once. Next, the entry function is called. While the entry function is executing, any number of calls to the suspend and resume functions may be made, depending on how many times the thread is activated and deactivated. When the thread is terminated, the suspend function is called once if the thread was active when it was terminated. Finally, the end function is called.

There is one special case that will occur when a thread is created and subsequently destroyed without ever having been activated. Since the thread was never activated, and therefore never started executing, there is no time at which the begin, resume, suspend, or entry functions could be called. In this case, only the end function will be called. For instance,

```
...
ThreadType thread = ThreadBegin(thread_entry, thread_suspend,
thread_resume, thread_data, thread_stack_size);
ThreadProcBeginSet(thread, thread_begin);
ThreadProcEndSet(thread, thread_end);
ThreadEnd(thread);
...
```

In this sequence of calls, the function thread_end will be the only function that is called by Thread Library. Depending on your point of view, this "feature" may be considered a bug in the thread protocol. I have not decided whether this should be considered a feature of, or an error in, the protocol.

You may notice some redundancy in the sequence in which the callbacks are executed. For instance, the begin function is always called before the entry function, yet both functions are called exactly once, and so either function can be used for special initializations required by the thread. Yet the begin function is clearly symmetrical to the end function, and the end function is certainly required for proper use of Thread Library for those threads that need a special termination function. The choice of call sequences was partially constrained by the need to maintain reasonable compatability with prior versions of Thread Library. The decision as to the types of functions called and when they would be called was based on the completeness and symmetry of the current solution.

Main Thread

The main thread can use only the resume and suspend functions. The purpose of the call to ThreadBeginMain is to initialize Thread Library, and to store information about the main thread so that it can be activated and deactived by Thread Library. The begin, end, and entry functions are never used, since the main thread starts executing before Thread Library is ever called, and continues to execute even after disposing of all threads (including the main thread).

Environment When Protocol Functions are Called

When any of the functions other than the begin and end functions is called, the stack is set to the stack of the thread for which the function was installed, and the active thread is set to that thread. Thus, the functions can access the thread into which they were installed by calling ThreadActive. Unlike the other functions, however, the begin and end functions might be called while a different thread is active, and may therefore not access their thread by calling ThreadActive and may not make assumptions about the contents of the thread's stack. This restriction is necessary since the begin function may be called when the thread is created from within a separate thread, and the thread may be terminated by a call to ThreadEnd that is executed from a separate thread.

Functional Interface of Protocol Functions

This section describes the functional interface that the application-defined protocol functions must use.

Begin

```
void begin(ThreadType thread, ThreadDataType data);
```

Parameters

thread	thread for which the function is called;
data	pointer to thread's application-defined data.

Preconditions

ThreadMain() ∈ TAssigned.

thread ∈ (TAssigned \ { ThreadMain() }).

data = ThreadData(thread).

Postconditions

No special postconditions.

Description

Called once before the thread starts executing. You can use this function to do any special initialization of the thread.

The begin function is optional and may be set to NULL.

End

```
void end(ThreadType thread, ThreadDataType data);
```

Parameters

thread	thread for which the function is called;
data	pointer to thread's application-defined data.

Preconditions

ThreadMain() ∈ TAssigned.

thread ∈ (TAssigned \ { ThreadMain() }).

data = ThreadData(thread).

Postconditions

No special postconditions.

Description

Called whenever ThreadEnd is called for the thread, whether explicitely by your application, or implicitely by Thread Library when the thread's entry point returns. You can use this function to do any special termination of the thread, such as disposing of memory allocated for the thread. This function is especially useful if a thread is terminated with ThreadEnd while it is not the active thread. By specifying an end function for the thread, the application can dispose of any memory it allocated.

The end function is optional and may be set to NULL.

Resume

```
void resume(ThreadDataType data);
```

Parameters

data	pointer to thread's application-defined data.

Preconditions

ThreadActive() ∈ TAssigned.

data = ThreadData(ThreadActive()).

Postconditions

No special postconditions.

Description

Called whenever the thread is activated and once before the entry point is executed. You can access the thread for which the resume function was called by calling ThreadActive. You can use this function to do any special configuration of your thread that must occur whenever it is activated. For instance, if you are using exceptions to handle errors, you might need to swap the exception environment to the exception environment for the thread.

The resume function is optional and may be set to NULL.

Suspend

```
void suspend(ThreadDataType data);
```

Parameters

data pointer to thread's application-defined data.

Preconditions

ThreadActive() ∈ TAssigned.

data = ThreadData(ThreadActive()).

Postconditions

No special postconditions.

Description

Called whenever the thread is suspended and once after the entry point of the thread has returned. The suspend function is called exactly the same number of times as the resume function. You can access the thread for which the suspend function was called by calling ThreadActive. You can use this function to undo any special configuration of your thread that you did in the resume function.

The suspend function is optional and may be set to NULL.

Entry

```
entry(ThreadDataType data);
```

Parameters

data pointer to thread's application-defined data.

Preconditions

ThreadActive() ∈ TAssigned.

data = ThreadData(ThreadActive()).

Postconditions

No special postconditions.

Description

This function is the entry point for the thread. When your thread is first activated, the entry function is called. When the entry function returns, the thread is terminated by a call to ThreadEnd. A thread can also be terminated at any time (whether or not it is the active thread) by a direct call to ThreadEnd. You can access the thread for which the entry function is called by calling ThreadActive.

The entry function is required.

Memory Allocation

Thread Library provides two types of memory allocation, which it uses to dynamically allocate and dispose of pointers to blocks of memory.

If you do not specify a memory allocation method, then the default allocator uses the Memory Manager calls NewPtr and DisposePtr to allocate and dispose of memory. This method of memory allocation is sufficient for most users.

The application can also specify its own memory allocator. The application's memory allocator can be used to further tune the performance of Thread Library and can be used to integrate the blocks of memory used by Thread Library with the memory allocation scheme used by the rest of your application. This method of allocation is more complex than the default method, and requires additional work to implement, but also provides the greatest flexibility.

Allocation Algorithm

When Thread Library needs to allocate memory, it first check for an application-defined memory allocation function. If a function was provided by the application, then that function is used. If the function sets the error code using ThreadErrorSet to any value other than THREAD_ERROR_NONE, then Thread Library assumes that the memory allocation request failed. If the function returns NULL, but does not use ThreadErrorSet to set the error code, then Thread Library will attempt to allocate the memory using the default allocation method. Thus, it is generally sufficient for the memory allocation function to return NULL if sufficient memory was not available. If the memory allocation failed because of some other reason, or if you do not want Thread Library to allocate the memory using a different method, you can call ThreadErrorSet with an appropriate error code.

If the application did not specify a memory allocation function, or if the memory allocation callback returned NULL (but did not set the error code) then Thread Library attempts to allocate the memory using NewPtr. If NewPtr returns NULL, then Thread Library sets the error code to the result of MemError, or to memFullErr if MemError returns the value noErr.

Thread Library stores sufficient information with each block of memory it allocates to enable it to dispose of the memory using the appropriate disposal function. Thus, memory allocated by an application-defined allocation function is disposed of using the application-defined disposal function while memory allocated by NewPtr is disposed of using DisposePtr.

Application Defined Memory Allocation Functions

The application can install functions that are called when Thread Library needs to allocate or deallocate a pointer to a block of memory. You can use your own memory allocation functions to support better memory management in the context of your own application. For instance, you could maintain a pool of memory blocks to provide fast creation and destruction of threads. It is the responsibility of the application to ensure that the correct memory disposal function is installed with the corresponding memory allocation function. Functions for accessing the application-defined memory allocation functions of a thread are described in the section “Access to Application Defined Memory Allocation Functions”.

Functional Interface to Allocation Functions

This section describes the functional interface to the application-defined memory allocation functions.

Allocate

```
void *allocate(ThreadSizeType size, ThreadTypeType type);
```

Parameters

size number of bytes to allocate;

type type of memory to allocate.

Preconditions

0 ≤ size ≤ THREAD_SIZE_MAX.

type ∈ { THREAD_TYPE_UNSPECIFIED,
THREAD_TYPE_THREAD,
THREAD_TYPE_STACK }.

Postconditions

result ∈ { NULL, pointer to memory }.

result ≠ NULL ⇔

[TMemoryAvailable' ≤ TMemoryAvailable - size.

Memory pointed to by result is in the application's heap and is available for use by Thread Library.]

Description

Called whenever Thread Library needs to allocate a block of memory. Must return a block of memory compatible with the standard ANSI C library function malloc. Blocks allocated with the allocate function should be contained within the current application's heap zone. It is not, however, necessary that the blocks be allocated using NewPtr. The type parameter specifies the type of memory being allocated. In the future, it may be necessary to place additional constraints on certain types of memory blocks used by Thread Library; the type parameter provides for such enhancement.

Dispose

```
void dispose(void *p, ThreadSizeType size, ThreadTypeType type);
```

Parameters

p	pointer to block of memory previously allocated with a corresponding allocation function;
size	number of bytes to which p points;
type	type of memory to which p points.

Preconditions

p∈ { NULL, pointer returned by a prior call to allocate }.

0 ≤ size ≤ THREAD_SIZE_MAX.

type ∈ { THREAD_TYPE_UNSPECIFIED,
THREAD_TYPE_THREAD,
THREAD_TYPE_STACK }.

Postconditions

p ≠ NULL ⇔ TMemoryAvailable' ≥ TMemoryAvailable + size.

p ≠ NULL ⇒ Memory pointed to by p is no longer available for use by Thread Library.

Description

Called whenever Thread Library needs to dispose of a block of memory that was allocated with a corresponding allocation function. The size and type parameters have the same values as the values passed to the corresponding memory allocation function. The dispose function must behave in a manner compatible with the standard ANSI C library function free.

Apple's Thread Manager

Thread Manager is an implementation of threads provided by Apple. Thread Manager is now bundled as part of system 7.5, though previously it was distributed as a separate extension. Thread Manager is an integral part of MacOS, and is supported by Apple. Since it includes hooks for debuggers, some debuggers are now thread-aware and can help you debug threaded applications. Thread Manager for M68K Macintoshes provides preemptive threads, but the PowerPC version does not provide preemption. You can find more documentation about Thread Manager in any Apple distribution of it (e.g., the SDK or ETO CD-ROMs).

Note

Thread Library has no connection with Thread Manager or with ThreadsLib (a library provided by Apple for use with Thread Manager). ◆

Threads are properly considered a low-level operating system service. The Thread Manager is part of MacOS and is likely to be maintained and upgraded by Apple so as to be compatible with future versions of the operating system and with future hardware platforms. As such, Apple's Thread Manager provides the best supported method to implement threads under MacOS.

Thread Library is not intended to compete with Thread Manager. I wrote Thread Library to see how hard it would be to implement, to gain experience in low-level OS hacking, and to provide a free version of clean source code so that others could see how threads might be implemented in MacOS.

If you intend to use Thread Library in your applications, I recommend that you wrap your calls to Thread Library in a compatibility layer, such as that provided by the ThreadLibraryManager library provided with Thread Library. This layer would allow you to switch to Thread Manager should the need arise. For software that must run under system 6, or on systems where Thread Manager is not installed (as determined by the appropriate Gestalt selector), your compatability layer could automatically switch to Thread Library instead of Thread Manager. Even if you do not intend to use Thread Manager, you may still need to place a layer between your application and Thread Library. For instance, I use a layer of code—with nearly identical names and semantics to a subset of the code in Thread Library—to allow my applications to save and restore exception and profiler stacks, and to use my own custom memory allocator.

WARNING

It is not possible to use both Thread Manager and Thread Library at the same time within a single application. Using both Thread Manager and Thread Library in the same application will result in undefined behavior. ◆

Today, Thread Library offers fewer advantages over Apple's Thread Manager than it might once have had. While personally I may think that Thread Library's functional interface is cleaner and more logically structured, that is not the most compelling reason for its use in production software (if only because it is possible to place a wrapper around Thread Manager to improve its interface). Thread Library is faster than Thread Manager, and runs in native mode on the PowerPC. Also, Thread Library is compatible with system 6, which may be important for some applications, though the share of the market of Macintoshes using system 6 is shrinking. Finally, at the time I wrote Thread Library, Apple was not bundling Thread Manager with the operating system, and required a higher licensing fee ($200 versus $50 today), so Thread Library provides an option for those writing freeware and shareware applications, and who might therefore not have the resources to license Thread Manager.

Known and Potential Problems

This section lists any known bugs, incompatibilities, or suspected problems that you might encounter when using Thread Library.

Handling Events

Only the main thread may make calls to WaitNextEvent, GetNextEvent, and EventAvail. Calling WaitNextEvent or GetNextEvent from within any other thread may cause the application to crash. Calling EventAvail from within any other thread may result in odd behavior, though not necessarily a crash. I find it useful to retrieve all events using a single function which includes a simple assertion, such as "assert(ThreadActive() == ThreadMain())", to ensure that it is called only from within the main thread. All other threads should call ThreadYield and allow the main thread to retrieve events. As mentioned in the description of the scheduling algorithm, the main thread is scheduled whenever there are any pending events, so pending events will be handled as soon as ThreadYield is called.

Virtual Memory

It is possible that allocating a thread's stack in virtual memory may cause the operating system to crash if it attempts to execute a task on the thread's stack while the stack is swapped to disk. I am not sufficiently familiar with virtual memory to determine if this would indeed be a problem. Further analysis and testing are required.

Thread Manager

It is not possible to use both Thread Manager and Thread Library at the same time within a single application. Using both Thread Manager and Thread Library in the same application will result in undefined behavior.

Toolbox

For all threads other than the main thread, some Macintosh Toolbox routines may not work correctly if the stack is not between the region of memory defined by the low-memory globals CurStackBase and ApplLimit. Possibly prohibited are some QuickDraw calls, but I do not actually know which Toolbox routines will fail. Some simple tests I ran created a dialog with a progress bar; created, opened, read and wrote files; created a resource file and added resources to it; allocated memory; and did various other operations, all successfully and without problems. Since the main thread uses the application's stack, there are no restrictions on the Toolbox routines that the main thread may call. I am interested in whether you encounter (or do not encounter) limitations to Toolbox calls, and would like to know under what conditions the limitations arise.

SuperClock

SuperClock!, written by Steve Christensen, is a popular control panel that displays a clock in the menu bar. A version derived from SuperClock! is now a standard part of system 7.5. SuperClock! 4.0.4 will not update its numerals every second when EventAvail is called from within a thread other than the main thread. SuperClock! will still update the timer animation when you run the stopwatch, and will update the numerals less frequently. This happens both in my Thread Library and in Apple's Thread Manager. You usually will not notice this effect since threads usually do not call EventAvail. I am not sure if this is a bug in threads or in SuperClock!. At any rate, this appears to be a minor cosmetic problem and it should not interfere with the operation of an application that uses threads.

Future Operating Systems

Implementing threads requires, by necessity, some nonportable source code. For instance, the register swapping code depends on the specific instruction set and architecture of the computer and operating system. The main system-specific dependencies in Thread Library are:

- the instructions used to save and restore the registers;
- the stack grows down from high-memory to low-memory;

- any block of memory allocated from within an application's heap may be used for a thread's stack;
- several low-memory globals are accessed and modified by Thread Library; these low-memory globals might not be available in future versions of MacOS.

There are doubtless additional nonportable assumptions that I have made in writing Thread Library. The source code will have to be modifiedIf any of the assumptions that I relied upon becomes invalid.

To Do

Macintosh ticks are coarse time units. It would be better to measure times in microseconds rather than ticks. To avoid complicating the interface or slowing down the code with runtime checks to determine if intervals should be measured in microseconds or in ticks, it would probably be best to use a conditional compilation directive to control Thread Library's behavior.

It is possible to add preemptive threads to Thread Library. Preemptive threads have a limited utility, however, since they must be executed at interrupt time, precluding the use of most of the Macintosh toolbox. Someday, if there is demand for preemptive threads, I may add this feature. (I actually started trying to implement this, but it is tricky, so it may take a while, by which time Apple's Thread Manager may provide preemptive threads on both M68K and PowerPC Macintoshes.)

Some low-memory globals may not be available under A/UX (most notably, the Ticks low-memory global). Adding a runtime check for A/UX to determine if the Ticks low-memory global is available could slow down access to the variable, so it may be better to include a conditional compilation option.

Credits

Some ideas on how to use setjmp/longjmp to swap stacks were adapted from the source for Task Manager v2.2.1 by Michael Hecht <Michael_Hecht@mac.sas.com>, available at the info-mac archives and various other sites.

Special thanks to Peter Lewis <peter.lewis@info.curtin.edu.au>, who did a detailed review of Thread Library and made numerous suggestions to successive versions, including using serial numbers to refer to all threads, using a sentinel value in the stack-sniffer VBL task, and improving the scheduling of threads. Also, though it took me several months to decide to add it, it was his suggestion to add the ability to enable and disable individual threads.

Thanks to Metrowerks for allowing me to distribute a modified version of "setjmp.s" as the file "regppc.s".

Thanks also to all of the following people for helping me make Thread Library a better product.

Anton Rang <rang@icicle.winternet.mpls.mn.us> responded to my query on Comp.sys.mac.programmer on how to disable the stack sniffer VBL task. (Several other people also responded, but Anton Rang's reply was the first to arrive.)

Daniel Sears <sears@netcom.com> reported some problems with Thread Library, including a conflict with SuperClock!, and tried out updated versions I emailed to him.

Matthew Xavier Mora <mxmora@unix.sri.com> suggested the SetPort call in TestThreads and helped debug an update problem in ThreadsTest.

Barry Kirsch <bkirsch@NADC.NADC.NAVY.MIL> reported a problem with compiling Thread Library using THINK C's "MacHeaders" precompiled header. He also noted that, when using Thread Library 1.0d4, the Finder's About window would display incorrect information about the memory utilization of the application; this prompted me to remove the call to EventAvail and replace it with a different method to test for events which would not produce this problem. He also prodded me into adding the functionality of ThreadEndAll.

Finally, thanks are due to anyone I may have forgotten to mention above, and who helped me create or improve Thread Library.

Bibliography

Giering, Ted, and Mueller, Frank, et. al. “Pthreads”, 1993.

From the README file: “Pthreads is a prototype implementation of POSIX 1003.4a, Draft 6. It is a C-language library that supports multiple threads of control within a single process.” Pthreads is free. Though Pthreads was written for Unix systems, it may be possible to port it to MacOS. I downloaded version 2.3 from the Internet, though I have misplaced the ftp address. The authors provide the email address pthreads-bugs@ada.cs.fsu.edu, to which inquiries may be directed.

Lewis, Ted G., and El-Rewini, Hesham, “Introduction to Parallel Computing”, Prentice Hall, 1992.

Introduces parallel processing and issues that arise when programming concurrent software. Also provides information on basic locking mechanisms for software, such as test-and-set and queue locks.

Meyer, Bertrand, “Object-Oriented Software Construction”, Prentice Hall, 1988.

A good introduction to object-oriented techniques, and especially to Meyer's excellent object-oriented language Eiffel. The Eiffel programming language includes reasonably good runtime facilities for testing predicates.

Potter, Ben, et. al. “An Introduction to Formal Specification and Z”, Prentice Hall, 1991.

Introduces formal specification. The notation I used for the predicates in the preconditions and postconditions is based on information in this book.