



INSIDE MACINTOSH

Thread Manager



August 25, 1999
Technical Publications
© 1999 Apple Computer, Inc.



Apple Computer, Inc.

© 1995 Apple Computer, Inc.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., except to make a backup copy of any documentation provided on CD-ROM.

The Apple logo is a trademark of Apple Computer, Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book.

Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for typographical errors.

Apple Computer, Inc.

1 Infinite Loop

Cupertino, CA 95014

408-996-1010

Apple, the Apple logo, and Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Adobe, Acrobat, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

Helvetica and Palatino are registered trademarks of Linotype-Hell AG and/or its subsidiaries.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

PowerPC is a trademark of Information Business Machines Corporation, used under license therefrom.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures, Tables, and Listings 5

Chapter 1 Thread Manager 7

Introduction to Threads	8
About the Thread Manager	9
Scheduling	11
The Main Thread	13
Custom Scheduler	13
Default Saved Thread Context	15
Custom Context-Switching Function	16
Thread Stacks	17
Creating and Disposing of Threads	18
Using the Thread Manager	19
Determining Attributes of the Thread Manager	19
Creating and Allocating a Thread	20
Creating a Pool of Threads	22
Allocating a Thread	24
Turning Scheduling Off	28
Working With Stacks	29
Creating Dialog Boxes That Yield	31
Passing Input and Output Parameters to a New Thread	33
Using Threads With I/O	36
Thread Manager Reference	44
Data Types	44
Gestalt Selector and Response Bits	44
The Thread State	45
The Thread Task Reference	46
The Thread Type	46
The Thread ID	47
Thread Options	48
The Scheduler Information Structure	49
Thread Manager Functions	49

Creating and Getting Information About Thread Pools	50
Creating and Disposing of Threads	56
Getting Information About Specific Threads	60
Scheduling Threads	64
Preventing Scheduling	69
Getting Information and Scheduling Threads During Interrupts	72
Installing Custom Scheduling, Switching, Terminating, and Debugging Functions	77
Application-Defined Routines	85
Summary of the Thread Manager	92
C Summary	92
Constants and Data Types	92
Thread Manager functions	93
Pascal Summary	96
Constants and Data Types	96
Thread Manager Functions	98
Assembly Language Information	100
Result Codes	101

Glossary	103
----------	-----

Index	105
-------	-----

Figures, Tables, and Listings

Chapter 1 Thread Manager 7

Figure 1-1	Relationship of Thread Manager and Process Manager	10
Figure 1-2	Thread scheduling model	12
Figure 1-3	Thread Manager default and custom scheduling mechanisms	14
Figure 1-4	Custom context-switching function	17
Figure 1-5	Using a completion routine to wake up a thread making an asynchronous I/O call	37
Figure 1-6	Using two threads to handle an asynchronous I/O call	38
Table 1-1	Registers in the 680x0 default thread context	15
Table 1-2	Registers in the PowerPC default thread context	16
Listing 1-1	Setting up the main thread	20
Listing 1-2	Creating a thread pool	23
Listing 1-3	Allocating threads	24
Listing 1-4	Using the <code>gPhilo</code> structure in a subroutine	27
Listing 1-5	Marking a critical section of code	28
Listing 1-6	Increasing the size of the main thread's stack area	30
Listing 1-7	Determining and increasing the stack size of a thread	31
Listing 1-8	Creating a dialog box that yields	31
Listing 1-9	Passing data between threads	33
Listing 1-10	Making an asynchronous I/O call with two threads	39

Thread Manager

This chapter describes how you can use the Thread Manager to provide threads, or multiple points of execution, in an application. You can think of the Thread Manager as an enhancement to the Process Manager, which still governs how applications work together in the Macintosh multitasking environment. Therefore, you should already be familiar with the concepts in *Inside Macintosh: Processes* and *Inside Macintosh: Memory* before reading this chapter.

Read this chapter if you are interested in developing an application with more than one thread (called a *threaded application* in this document). If your application uses no Thread Manager functions, the Process Manager treats it as a single-threaded application (called a *nonthreaded application* in this document). The Process Manager does call the Thread Manager at launch time to create the main thread for the application, but it does this transparently and in no way affects the performance of your application.

This chapter begins by describing the advantages of using threads within an application context. It describes the scheduling model that the Thread manager provides, the context information that the Thread Manager saves when it switches one thread out and another one in, and it describes thread stacks. It then shows how to

- create threads and thread pools and set them up to run
- turn scheduling on and off
- work with stacks
- create dialog boxes that yield control to other threads
- pass information between threads
- install custom scheduling and context-switching functions
- use threads to make asynchronous I/O calls

Introduction to Threads

Threads, also known as *lightweight tasks*, are a way to develop concurrency, or multiple points of execution, within a particular context, such as in an operating system or application. The Thread Manager offers threads for use within an application context only. It does not provide threads to be used on a systemwide basis.

Threads offer a new and better way to structure applications for simplicity, efficiency, and responsiveness. With multiple points of execution, you can do things such as

- separate the user interface from time-consuming tasks to guarantee responsiveness to the user
- place a modal dialog box in one thread and a function to process data or perform calculations in a different thread so that your application can continue working rather than sitting and waiting while a user decides which choice to make in the dialog box
- simplify your code by placing each element of a simulation in a separate thread
- increase the efficiency of your application by eliminating many VBL and Time Manager tasks.

Although you can already do many of the things that threads enable you to do, the implementation without threads can be difficult and inelegant. For example, with null events at idle time you can write idle-processing procedures that bring a measure of concurrency to your application. However, threads offer many advantages not available with other methods of achieving concurrency in an application program.

A major benefit of using threads is that you can enhance the logical structure of your program. Using threads is in many ways like adding an object layer to your program. For example, one way to write a traffic simulation program is to create a separate thread to control each element of the simulation—that is, the traffic signals and the individual cars. You could think of each thread as an object with particular capabilities. In any case, programs with threads are easier to write and much easier to understand than programs that achieve

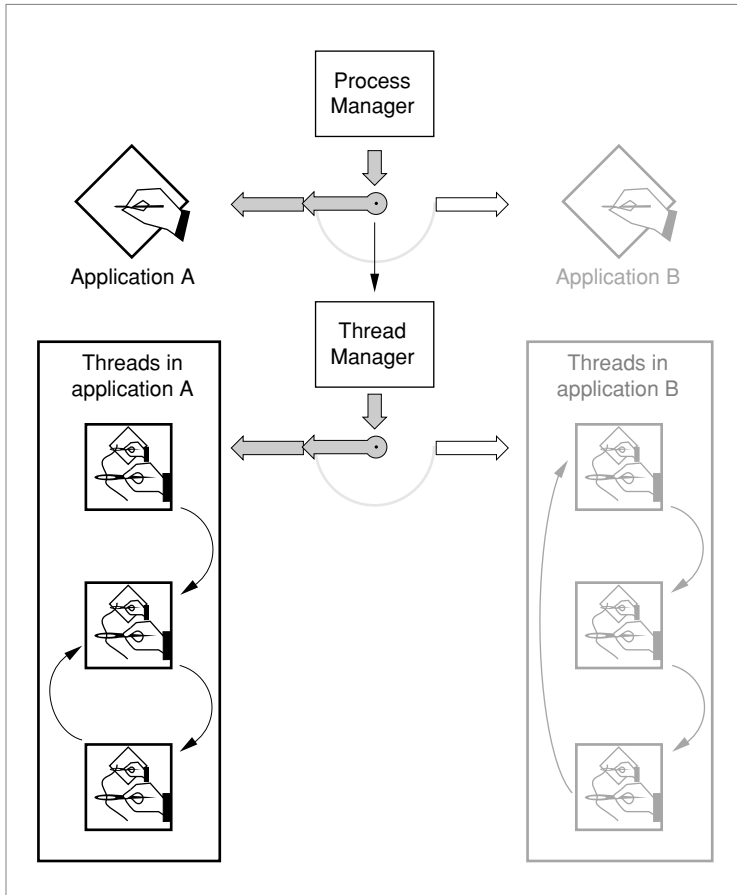
Thread Manager

concurrency in a roundabout fashion, such as using idle-processing procedures or state machines.

A thread consists of application code and the processor state or context to execute it. The **thread context** consists of a register set, a program counter, and a stack. Each thread shares the address space, file access paths, and other system resources of the application process in which it runs. Therefore, when the Thread Manager switches control from one thread to another, the amount of context information it must save is relatively small and the switch is much faster than that between application processes.

About the Thread Manager

The Thread Manager manages threads within an application context. It provides routines to create, get information about, schedule, and dispose of threads. The Process Manager, on the other hand, is responsible for switching the context between various application processes within the Macintosh multitasking environment. Figure 1-1 illustrates the relationship between the Thread Manager and the Process Manager.

Figure 1-1 Relationship of Thread Manager and Process Manager

Threads in an application are available to run only when the Process Manager schedules the application to run. For example, when the Process Manager switches in application A, its threads can run. The Thread Manager saves the thread context information each time it switches out one thread and schedules a different one to run. When the Process Manager switches in application B, the threads in application A are no longer available to run and the Thread Manager now manages the threads in application B.

Scheduling

The Thread Manager provides a single, cooperative method of scheduling threads. In cooperative scheduling, a thread must explicitly yield control to give other threads an opportunity to run.

Previously, the Thread Manager supported preemptive scheduling as well as cooperative scheduling but currently only cooperative scheduling is supported.

The situation for threads within an application is similar to that of applications in a multitasking environment. Every application must have periodic yielding calls that allow the Process Manager to schedule other applications as necessary—for example, when a user presses the mouse button to select another application to run. Likewise, every thread within an application must make regular yield calls to allow other threads to run. The Thread Manager provides the following functions to yield control to other threads:

- `YieldToAnyThread`, which yields control to the next thread available to run
- `YieldToThread`, which yields control to a specific thread.
- `SetThreadState`, which you can use to change the state of the current thread from running to ready or stopped. When you do so, you either specify a new thread to run or let the Thread Manager schedule the next available thread.

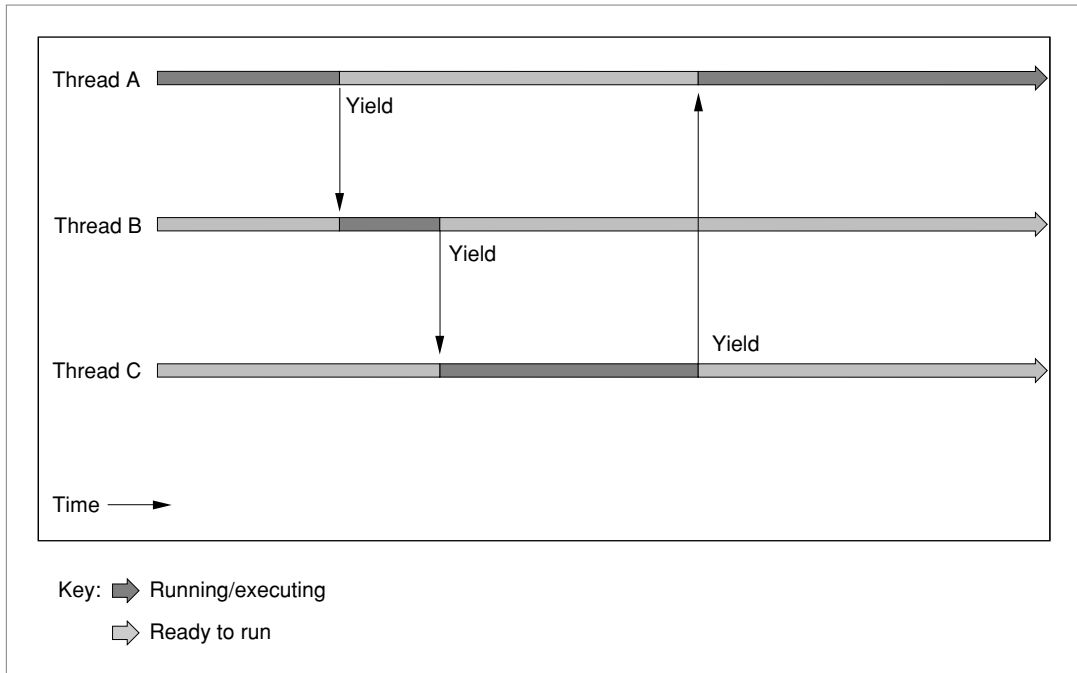
As you can see from the three calls that yield control from the current thread, there are two ways to determine the next thread to run. One way is for you to specify a particular thread to run next; the other way is to allow the Thread Manager to choose the next available thread to run. (An available thread is one that is marked ready to run—an unavailable thread is one that is marked stopped.) The Thread Manager queues up all of the threads that are ready to run, and, when a nonspecific yield occurs, it executes the next available thread. When a thread finishes executing, it moves to the back of the queue if it is still ready to run, or, if it is marked as stopped, the Thread Manager removes it from the queue of available threads.

Note

The previous paragraph describes the default Thread Manager scheduling mechanism. You can also define a custom scheduler for your application that works in conjunction with the default scheduling mechanism to determine the next thread to run. See “Custom Scheduler” (page 13) for more information about creating a custom scheduler for your application.

Figure 1-2 shows the default Thread Manager scheduling model.

Figure 1-2 Thread scheduling model



Because threads yield control under explicit conditions, they have access to all Toolbox and Operating System routines. They allow you to do anything that you can currently do in an application without threads, such as allocate memory, perform file I/O, perform QuickDraw operations, and so on.

For situations in which you are concerned about the integrity of your data, the Thread Manager provides a pair of functions, `ThreadBeginCritical` and `ThreadEndCritical`, that enable you to mark a section of code as critical, turning scheduling off. With scheduling off, the Thread Manager does not allow any threads to be scheduled until scheduling is turned back on; that is, all yield and other scheduling functions are ignored until the code exits the critical section. See “Turning Scheduling Off” beginning on page 1-28 for information on how and when to mark sections of code as critical.

The Main Thread

When the Process Manager launches your application, it creates and runs a special thread, called the **main thread** or application thread. The `main` function is the entry point to this thread and to the application. The main thread has some characteristics that distinguish it from other threads. It is the only thread that has a preallocated stack—the stacks for threads that your application creates reside in separate areas of the heap. The main thread is the only thread from which you properly can extend the application heap. Therefore you should call `MaxAppZone` from the main thread immediately after your application launches, or at least before any other threads run.

Another characteristic of the main thread is that the Thread Manager assumes the main thread handles event processing. Therefore, whenever an operating-system event occurs, the Thread Manager schedules the main thread at the next scheduling opportunity, no matter where the main thread happens to be in the scheduling queue.

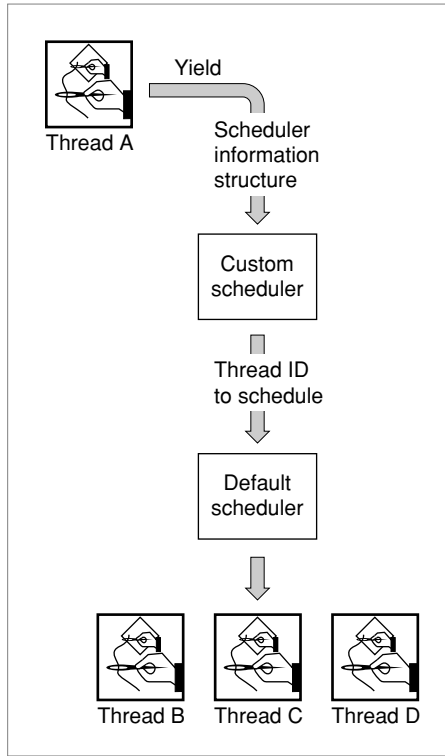
Note

After an operating-system event, the Thread Manager schedules the main thread at the next opportunity unless you have specified a particular thread to run. In other words, if you call a function such as `YieldToAnyThread` to cause the rescheduling, the Thread Manager runs the main thread. If, however, you call a function such as `YieldToThread` and specify a particular thread to run, the Thread Manager schedules that thread rather than the main thread even after the occurrence of an operating-system event. ♦

To guarantee responsiveness to users, you should put all your event handling in the main thread. For the same reason, it is highly recommended that you never put the main thread in the stopped state.

Custom Scheduler

The Thread Manager allows you to install a custom scheduling function that works in conjunction with the Thread Manager default scheduling mechanism. You install the custom scheduling function with the `SetThreadScheduler` function. Figure 1-3 shows how the custom scheduler works with the default Thread Manager scheduling mechanism.

Figure 1-3 Thread Manager default and custom scheduling mechanisms

When a yield or other Thread Manager call triggers a reschedule, the Thread Manager calls the custom scheduling function and passes it a scheduler information structure. This structure has four fields; the first contains the size of the structure and allows for expansion in the future. The next two fields are thread IDs that identify the current thread and the thread that the application has selected to run next. The final field was to identify a cooperative thread that was interrupted by a preemptive thread. However, because it no longer supports preemptive threads, the Thread Manager always passes the `kNoThreadID` constant for this field.

The custom scheduling function can use this information to determine which thread to schedule next. It returns to the default scheduling mechanism the

Thread Manager

thread ID of the next thread to schedule and the Thread Manager does the actual scheduling.

Default Saved Thread Context

When the Thread Manager switches the context between one thread and another, it saves a default context, which consists of the CPU registers, the floating-point (FPU) registers (if any), and the location of the context information.

The thread context resides on a thread's stack and the Thread Manager saves the location of this context when it switches contexts between threads. The A5 register (GPR1 on the PowerPC) for each thread contains a pointer to the application's global data world. When it switches contexts, the Thread Manager initially sets A5 (GPR1) and the MMU mode to the same values as those in the main thread. In this way all threads can share in the application's global data world.

Table 1-1 shows the registers that the Thread Manager saves for a 680x0 application.

Table 1-1 Registers in the 680x0 default thread context

CPU registers	FPU registers
D0–D7	FPCR, FPSR, FPIAR
A0–A7	FP0–FP7
SR (including CCR)	FPU frame

For 680x0 applications, when you create or allocate a thread with the `NewThread` function, the Thread Manager provides an option that allows you to create a thread whose FPU registers are not to be saved. This allows faster context switches for threads that don't use the FPU registers.

For PowerPC applications, the Thread Manager always saves the FPU registers, regardless of any options you set because the PowerPC processor can use the FPU registers for optimizations.

Table 1-2 shows the registers that the Thread manager saves for a PowerPC application.

Table 1-2 Registers in the PowerPC default thread context

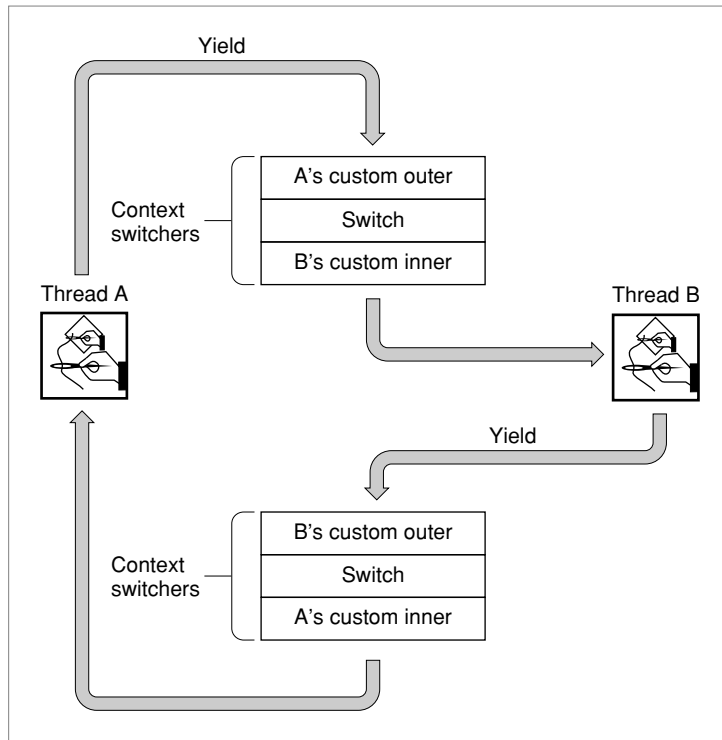
CPU registers	FPU registers	Machine registers
R0–R31	FP0–FP31	CTR, LR, PC
	FPSCR	CR, XER

Custom Context-Switching Function

The Thread Manager allows you to install a custom context-switching function to supplement the context information that the Thread Manager saves when it switches control between one thread and another. This section describes how to install a custom context-switching function and use it in conjunction with the default context-switching mechanism.

You install a custom context-switching function with the `SetThreadSwitcher` function. You assign a custom switching function separately to each thread. However, because you also pass a parameter containing thread specific information, you can define a single switching function to assign to all threads and use this parameter to pass specific information to each thread.

The Thread Manager calls the custom context-switching function, `MyThreadSwitchProc`, whenever the thread it is assigned to is scheduled. Because it is a ‘switcher inner’ it is called just before the code starts executing. The Thread Manager calls a ‘switcher outer’ custom switching function just after the code in the thread stops executing. Figure 1-4 shows when the Thread Manager calls each type of custom context-switching function.

Figure 1-4 Custom context-switching function

Thread Stacks

When the Process Manager launches a threaded application, it creates the main thread and sets up the stack for it just like it would for a nonthreaded application. You can expand this stack by calling the Memory Manager `SetAppLimit` function at the beginning of your application. See “Working With Stacks” (page 29) for an example of how to use this function.

For each subsequent thread that you create, the Thread Manager maintains a separate stack in your application *heap* area.

Note

Because the Thread Manager does not move stacks during a thread context switch, you can pass function parameters on the stack. ♦

You specify the stack size when you create a new thread with the `CreateThreadPool` or `NewThread` function. The stack must be large enough to handle saved thread context, normal application stack usage, interrupt handling routines, and CPU exceptions. You can specify a particular size in bytes or use the default size that the Thread Manager supplies for a thread. The default size is more than adequate for most threads.

Creating and Disposing of Threads

There are two ways to create threads with calls to the Thread Manager. One is to use the `NewThread` function to create a single thread. The other way is to call the `CreateThreadPool` function to create a pool of threads that you later allocate with the `NewThread` function. The advantage of the latter method is that you handle memory allocation up front before fragmentation occurs.

When you create or allocate a thread with the `NewThread` function, you specify, among other things, the stack size. You also identify the function that is the entry point to the thread and can pass it data if you wish. You can also allocate storage that you can use to store the thread result.

When a thread finishes executing its code, the Thread Manager automatically calls the `DisposeThread` function to clean up after the thread. The `DisposeThread` function either removes the thread entirely (the default for cooperative threads) or recycles the thread into the thread pool. You can call `DisposeThread` yourself if you want to recycle a cooperative thread into the thread pool.

The `DisposeThread` function passes a parameter back to the `NewThread` function that initially created and launched the thread. It places the information from this parameter in the storage that the `NewThread` function allocated when it first created the thread. You can use this parameter to pass the thread result back to the calling thread, if you wish. For example, if you call a function to perform a calculation or process data, you can use the `DisposeThread` function to pass the result back.

See “Passing Input and Output Parameters to a New Thread” beginning on page 1-33 for information on how to return data from a thread to the thread that launched it.

Using the Thread Manager

This section describes how you can take advantage of the Thread Manager to create threaded applications. It describes how to

- use the Gestalt Manager to determine if the Thread Manager is available and which features are supported
- create a thread pool and allocate and run threads
- turn off scheduling in critical sections of code
- create dialog boxes that leave an application free to do other work
- pass parameters between threads
- use threads with asynchronous I/O routines

Determining Attributes of the Thread Manager

To determine if the Thread Manager is available and which features are supported, call the `Gestalt` function with the selector `gestaltThreadMgrAttr`. The `Gestalt` function returns information by setting or clearing bits in the response parameter. The following constants define the bits currently used.

```
enum {
#define gestaltThreadMgrAttr      'thds'      /* Thread Manager attributes */
    gestaltThreadMgrPresent      = 0,
    gestaltSpecificMatchSupport  = 1,
    gestaltThreadsLibraryPresent = 2
};
```

`gestaltThreadMgrPresent`

This bit is set if the Thread Manager is present.

`gestaltSpecificMatchSupport`

This bit is set if the Thread Manager supports the allocation of threads based on an exact match with the requested stack size. If this bit is not set, the Thread Manager allocates threads based on the closest match to the requested stack size.

Thread Manager

`gestaltThreadsLibraryPresent`

This bit is set if the native version of the threads library has been loaded.

Creating and Allocating a Thread

This section shows you how to create a pool of threads, allocate a thread from that pool, and get this thread to run. The code samples in this section are adapted from a sample application that addresses the classic computer science dining philosophers problem. This application uses a separate thread to control the display and movement of each philosopher icon as the philosophers move, one by one, into a dining room, pick up a fork eat, and then leave the room.

When your application launches, the Process Manager automatically creates the main, or application, thread. You are responsible for creating any additional threads. The `main` function is the entry point to the main thread. One of the features of the main thread is that it is the only thread from which you can properly call the Memory Manager `MaxApplZone` function to expand your application heap to its limit. You must call `MaxApplZone` before any other threads run.

Listing 1-1 shows the `main` function for an application, which calls subroutines that perform initializations and create and allocate threads. It also shows the application's event loop.

Listing 1-1 Setting up the main thread

```
void main()
{
    WindowPtr    appWindow;
    MaxApplZone();                               /* Expand application heap */

    DoInitMac();                                 /* Standard Macintosh
                                                application
initialization */
    DoCreateTPool();                             /* Create a pool of threads
*/

    appWindow = DoInitRooms();                  /* Drawing initialization */
}
```

CHAPTER 1

Thread Manager

```
        DoInitPhilos(appWindow);                /* Initialize philosophers */
        DoSpawnThreads();                       /* Allocate new threads */
        MyEventLoop();                          /* Event handlers
    }

        ...
void MyEventLoop()
{
    EventRecord      my_evt;
    short           got_evt = 0;
    OSErr           anError;
    WindowPtr       win;

    while( !gDone )
    {

        got_evt = WaitNextEvent(everyEvent, &my_evt,
                                kSleepTicks, nil);

        if ( got_evt )
        {
            switch( my_evt.what )
            {
                /* Case statements for each event */
            }
        }
        else
        {
            /* Draw window */
        }

        anError = YieldToAnyThread();
        if ( anError )
            DoHandlerError ("Error in yielding from the main
                            thread",
                            anError, kFatal);
    }
    /* Shutdown routines */
        ...
}
```

Thread Manager

In Listing 1-1, the first thing the `main` function does, after declaring a window pointer variable, is to call the `MaxApplZone` function to extend the application heap. To be safe, this call comes before any of the initialization calls. It must come before any other thread runs in the application.

The `main` function calls two functions (`DoInitMac`, and `DoInitPhilos`) to perform various Macintosh and application-specific initializations. These functions are not of particular interest here, other than to show the order in which `main` calls them in relationship to `MaxApplZone`, so the code they contain is not shown here. The `DoCreateTPool` and `DoSpawnThreads` functions create a pool of threads and allocate threads from the pool, respectively. The next two sections show and describe the code for these functions.

Note that the `main` function contains the event loop for the application. While it is not required that the main thread handle all events, it is highly recommended that it do so. A characteristic of the `main` thread is that whenever an operating-system event is pending, the Thread Manager schedules the main thread at the next generic scheduling opportunity (that is, when a `yield` or other call causes a reschedule but does not specify a particular thread to schedule next), no matter where the main thread is in the scheduling queue. This characteristic of the main thread guarantees responsiveness to users if, as in the sample code, the main thread handles event-processing.

Listing 1-1 shows a skeletal view of `MyEventLoop`. It is a standard event loop, with a `while` loop and various `case` statements to handle the various possible Macintosh events. The main thread should make a `yield` call often enough to allow other threads an opportunity to run. Therefore, it calls the Thread Manager `YieldToAnyThread` function each time through the event loop.

Creating a Pool of Threads

The `DoCreateTPool` function in Listing 1-1 creates a pool of threads. While it isn't strictly necessary to create a pool of threads—you can create and allocate threads in one step with the `NewThread` function—there are advantages to doing so. For example, you can allocate all the memory for your threads up front before memory is used or fragmented. Listing 1-2 shows the code in `DoCreateTPool`, which creates a pool of threads.

Listing 1-2 Creating a thread pool

```

#define kNumOfPhilos          5 /* Number of philosopher icons*/
#define kDefaultStackSize    0 /* System determines stack size
*/

void DoCreateTPool()
{
    OSErr anError;

    /* Make a pool of threads for the philosophers */
    anError = CreateThreadPool(kCooperativeThread,
                              kNumOfPhilos, kDefaultStackSize);
    if ( anError )
        DoHandlerError ("\pProblem creating thread pool",
                        anError, kFatal);
}

```

The code in `DoCreateTPool` passes three parameters to the `CreateThreadPool` function. The first, `kCooperativeThread`, is a constant defined by the Thread Manager specifying that the threads to create are cooperative threads.

Note

Historically, the Thread Manager supported two types of threads, cooperative and preemptive but now only cooperative threads are supported. The `CreateThreadPool` function (and the `NewThread` function) still require that you specify the type of the thread, even though only one type is available.

The next parameter, `kNumOfPhilos`, is an application-defined constant that specifies the number of threads to create—in this case, five. The last parameter, `kDefaultStackSize`, specifies that Thread Manager use the default stack size for the five threads that it creates. You can specify the size in bytes if you don't wish to use the default size. The Thread Manager defines a default size that is probably larger than the minimum size that is required.

If there is a problem creating the threads, `DoCreateTPool` calls the error handling function and passes it the result code returned by `CreateThreadPool`. Note that if there is not enough memory to create all the specified threads, `CreateThreadPool` creates none and returns the `memFullErr` result code.

Allocating a Thread

Once an application has created a pool of threads, it can allocate them by calling the `NewThread` function. You specify to the `NewThread` function the type of thread and stack size to use, whether to use an existing thread or create a new one, the entry point function for the thread, data to pass to this function, and storage that the thread can use to return data, if any, when it terminates. The `NewThread` function allocates a thread from the pool (or creates a new one, depending on the options you choose) and returns the thread ID.

In Listing 1-1, the `main` function calls `DoSpawnThreads` to allocate threads from the thread pool. Listing 1-3 shows the code in `DoSpawnThreads` that creates a thread, the code for the thread entry point function, `DoPhiloActions`, and the data structure, `gPhilo`, for passing information to the entry point function.

Listing 1-3 Allocating threads

```
#define kNumOfPhilos          5  /* Number of philopher icons*/
#define kDefaultStackSize    0  /* System determines stack size
*/

/* Spawn each thread from the pool of newly created threads */
void DoSpawnThreads()
{
    OSErr anError;
    short index;

    for ( index = 0; index < kNumOfPhilos; index++ )
    {
        anError = NewThread(kCooperativeThread,
                           DoPhiloActions,
                           (void *)&(gPhilo[index]),
                           kDefaultStackSize,
                           kUsePremadeThread,
                           nil,
                           &(gPhilo[index].theThread));

        if ( anError )
            DoHandlerError("\pError in creating the New Thread
(DoSpawnThreads)", anError, kFatal);
    }
}
```

CHAPTER 1

Thread Manager

```
/* Global declarations */
#define kNumberOfIterations          1000    /*Number of iterations*/
...
typedef struct                      { /* Resource handles where it is and whether it
                                     has a fork */

    Rect        thinking_location, waiting_location, dining_location;
    Rect        current_location;
    short       left_fork, fork_state;
    ThreadID    theThread;

} philoRecord, *philoPtr;
philoRecord    gPhilo[kNumOfPhilos]; /* global declaration */
...
/* Thread entry function */
pascal void *DoPhiloActions(void *thisPhilo)
{
    short index;

    for (index = 0; index < kNumberOfIterations; index++ )
    {
        DoThinkForAwhile();
        DoGoToEat(thisPhilo);
        DoPickUpLeftFork(thisPhilo);
        DoPickUpRightFork(thisPhilo);
        DoEatForAwhile(thisPhilo);
        DoPutDownRightFork(thisPhilo);
        DoPutDownLeftFork(thisPhilo);
        GoToThink(thisPhilo);
    }
}
```

As just mentioned, the `NewThread` function can either create a new thread or allocate an existing one from the thread pool. If you scan the parameter list for `NewThread` in Listing 1-3, you see that `kUsePremade` is passed as the fifth parameter. This is one of five possible options you can pass in this parameter (you sum them together if you want to use more than one) and it indicates to allocate an existing thread from the thread pool. For a description of the other four options, see “Thread Options” (page 48).

Thread Manager

The first parameter to the `NewThread` function specifies that `NewThread` allocate a cooperative thread, and the fourth parameter (the stack size parameter) contains `kDefaultStackSize`, which specifies the default stack size. The thread pool that `DoCreateTPool` created in Listing 1-2 contains five threads and each of these uses the Thread Manager default stack size.

As you can see, the `DoSpawnThreads` function calls the `NewThread` function in a loop to allocate a number of threads. In this case, the index for the `for` loop is the constant `kNumOfPhilos`, which is set to 5. So `DoSpawnThreads` calls the `NewThread` function until it has allocated all five threads from the existing pool of threads. If there is a problem allocating the threads, `DoSpawnThreads` calls the error handling function and passes it the result code returned by `NewThread`.

The `NewThread` function uses the very last parameter to store the thread IDs of the newly created threads. At each iteration of the loop, it places the thread ID of the newly created thread in a field of the `gPhilo` structure. Actually, since this structure is indexed, each thread ID is stored in a separate index of the `gPhilo` structure.

The remaining three parameters set up the entry point to the thread. The second parameter points to `DoPhiloActions` as the entry point function. Since the loop in `DoSpawnThreads` creates five threads, `DoPhiloActions` is the entry point to each thread.

With the next parameter, `NewThread` points to a structure, `gPhilo`, that it passes to `DoPhiloActions`. This structure contains location information that is used for screen drawing and updates for each of the philosopher icons. It also contains the thread ID of each of the threads.

The `NewThread` function uses the second to last parameter to allocate storage for the function result from the new thread. Here it passes `nil` to indicate that there is no need to retrieve information from the newly created threads. See “Passing Input and Output Parameters to a New Thread” (page 33) for information on how to set up storage to return data from a thread that you create.

By default, `NewThread` marks each thread that it creates as ready to run. As soon as the application executes the `YieldToAnyThread` function in `MyEventLoop`, the Thread Manager begins executing the first of the new threads and the application executes the code in `DoPhiloActions`.

In `DoPhiloActions` you can see that `NewThread` passes in the `gPhilo` structure as the `thisPhilo` pointer, which `DoPhiloActions` passes on to each of its subroutines, beginning with `DoGoToEat`. These subroutines use this structure to

move the onscreen window icons from place to place and to “eat”. For example, Listing 1-4 shows the code for one of the subroutines, `DoEatForAwhile`.

Listing 1-4 Using the `gPhilo` structure in a subroutine

```
void DoEatForAwhile(philosPtr thisPhilo)
{
    short counter, timeToEat = Random() % kEatingTimeLimit;

    thisPhilo->current_location = thisPhilo->dining_location;
    for (counter = 0; counter < timeToEat; counter++)
        YieldToAnyThread()
    ;
}
```

The code for `DoEatForAwhile`, places the icon in the dining room for a random amount of time, then yields control to another thread. The code for the other subroutines called by `DoPhiloActions` in Listing 1-3 is not shown here but it is similar: it either moves the icon into a different room, makes it stay put for awhile, or performs an action, such as lifting a fork.

When control moves to the next thread with the yield call, the same subroutines are executed as in the first thread, but they affect a different icon because the indexed data structure referenced by `thisPhilo` specifies five different icons in turn.

When control returns to the first thread in this sequence, it comes back to the statement in the `DoEatForAwhile` function after `YieldToAnyThread`, which was the last statement executed. Since this is the end of this subroutine, control goes back to `DoPhiloActions`, which then executes the next subroutine. This subroutine performs an action and then, since it also has a yield call, it yields to the next thread—the various threads continue to perform actions on the icon that they control while yielding to each thread in turn.

As you can see, the design of this application is such that the actions are controlled by one function, `DoPhiloActions`, and the icons are controlled by separate threads. The yield calls in each subroutine of `DoPhiloActions` produce the appearance of simultaneous movement of the different icons.

Turning Scheduling Off

In cases where you need to ensure data coherency, The Thread Manager provides a pair of functions, `ThreadBeginCritical` and `ThreadEndCritical` that disable scheduling temporarily by marking a section of code as critical. While the critical section of code is executing, no other threads can be scheduled; that is, the Thread Manager ignores all yield and other scheduling functions until the code exits the critical section.

Listing 1-5 shows a situation in which `ThreadBeginCritical` and `ThreadEndCritical` mark a section of code as critical.

Listing 1-5 Marking a critical section of code

```

Boolean batch = true
#define kNumOfPhilos          5 /* Number of icons to create*/
#define kNoCreationOptions    0 /* Use default options*/
#define kDefaultStackSize    0 /* System determines stack size
*/
...
void DoCreateThreads()
{
    OSErr anError;
    short index;
    if batch ThreadBeginCritical();
    for ( index = 0; index < kNumOfPhilos; index++ )
    {
        anError = NewThread(kCooperativeThread,
                           DoPhiloActions,
                           (void *)&(gPhilo[index]),
                           kDefaultStackSize,
                           kNoCreationOptions,
                           nil,
                           &(gPhilo[index].theThread);

        if ( anError )
            DoHandlerError("\pError in creating the New Thread
            (DoSpawnThreads)", anError, kFatal);
        YieldToAnyThread

```

Thread Manager

```
    }  
    if batch ThreadEndCritical();  
}
```

As you can see, the `DoCreateThreads` function calls the `NewThread` function in a loop to allocate a number of threads. In this case, the index for the `for` loop is the constant `kNumOfPhilos`, which is set to 5. So `DoCreateThreads` calls the `NewThread` function until it has created five new threads. If there is a problem allocating the threads, `DoCreateThreads` calls the error handling function and passes it the result code returned by `NewThread`.

In some cases you might want each newly created thread to run before the rest of the threads are created. However, in other cases, you might want `DoCreateThreads` to create all the threads before any of them runs. The Boolean variable `batch` and the `ThreadBeginCritical` and `ThreadEndCritical` functions enable you to control whether the threads begin running individually or together.

When `batch` is true, the code in the loop is marked as critical, so the Thread Manager ignores the `YieldToAnyThread` function. All the threads are created before any of them can run.

On the other hand, if `batch` is false, the loop is not marked as a critical section of code. The current thread yields control at the end of the loop, and since threads are created in the ready state, each newly created thread runs immediately after creation.

Working With Stacks

The main thread, which is created by the Process Manager when it launches an application, is the only thread whose stack resides in the application stack area—the stacks for threads that you create reside in the application heap area. The main thread's stack in a threaded application is identical to the stack in a nonthreaded application. Therefore, to increase the size of the main thread's stack in a threaded application, you can use the same Memory Manager commands that you would use in a nonthreaded application. Listing 1-6 shows how to do this.

Listing 1-6 Increasing the size of the main thread's stack area

```

OSErr IncreaseApplicationStack(Size incrementSize)
{
OSErr retCode;
    SetApplLimit((Ptr) ((unsigned long) GetApplLimit()
        incrementSize));
retCode=MemError();
if(retCode==noErr)
    MaxApplZone();

return retCode;
}

```

IMPORTANT

You call the function in Listing 1-6 only once at the beginning of your application. You must call it before any other threads in the application allocate memory. To be safe you should call it before any other threads run, because running another thread could trigger a call to the `LoadSeg` function (on a 680x0 machine only), which allocates memory and could grow the heap. ▲

For threads that you create in your application, the Thread Manager maintains a separate stack in the application heap area. You specify the stack size when you create a new thread with the `CreateThreadPool` or `NewThread` function. The stack must be large enough to handle saved thread context, normal application stack usage, interrupt handling routines, and CPU exceptions. You can specify a particular size in bytes or use the default size that the Thread Manager supplies for a thread. The default size, in most cases, is more than adequate for your needs.

You can call `GetDefaultThreadStackSize` to determine the default amount of space that the system allocates for threads.

If during testing you find that the stack size is inadequate for an individual thread, you can increase the amount of space for it when you create the thread with the `CreateThreadPool` or `NewThread` function. Listing 1-7 shows how to determine the current stack space for a particular thread and how to increase it.

Listing 1-7 Determining and increasing the stack size of a thread

```

OSErr IncreaseThreadStack(ThreadID testThread)
{
    anError = ThreadCurrentStackSpace(testThread, currentStackSize);
    anError = DisposeThread(testThread, 0, 0)
    anError = NewThread(kCooperativeThread,
                       DoSomething,
                       nil,
                       (currentStackSize) + 1000,
                       kNoCreationOptions,
                       nil,
                       testThread);
}

```

The `ThreadCurrentStackSpace` function returns, in the `currentStackSize` parameter, the amount of space available to the thread named `testThread`. Since you have already determined that this size is inadequate, you dispose of the thread by calling `DisposeThread`. Then `NewThread` creates a new thread. The third parameter specifies the stack space to allocate for this thread. In this case, the original amount is increased by a thousand bytes.

Creating Dialog Boxes That Yield

An easy thing to do with the Thread Manager is to free your application to do useful work in the background while waiting for a user to respond to a dialog box that is displayed on the screen. The way to do this is to handle the dialog box in the main thread and to put a call in the dialog's event filter function that yields control to other threads that can do useful work while the dialog box is displayed. Listing 1-8 shows the code to implement such a dialog box.

Listing 1-8 Creating a dialog box that yields

```

pascal boolean DoYieldFilter (DialogPtr theDialogPtr, EventRecord *theEvent,
                             short *theItemHit)
{
    /* Yield to whomever wants to run. */
    YieldToAnyThread();
    /* Call the standard filter procedure defined in Dialogs.h. */
}

```

Thread Manager

```

    return (MyStdFilterProc(theDialogPtr, theEvent, theItemHit));
}
/* The DoOKDialog function just handles a simple OK dialog box. */
void DoOKDialog (short dialogID)
{
    DialogPtr theDialog;
    short itemHit;
    GrafPtr savePort;
    OSErr theError;

    GetPort(&savePort);

    if ((theDialog = GetNewDialog(dialogID, NULL, (Ptr)-1)) != NULL)
    {
        SetPort(theDialog);
        ShowWindow(theDialog);
        do
        {
            ModalDialog(DoYieldFilter, &itemHit);
        } while (itemHit != okButton);
        DisposDialog(theDialog);
    } else
        DebugStr("\pCould not find dialog");
    SetPort(savePort);
}

```

In Listing 1-8, `DoOKDialog` is a function that handles an OK dialog box. It calls the Dialog Manager `ModalDialog` function to display the dialog box. The `ModalDialog` function calls an event filter procedure, `DoYieldFilter`. This procedure makes two calls; one to `YieldToAnyThread` and the other to `MyStdFilterProc`. The call to `YieldToAnyThread` enables your application to keep working while the dialog box is displayed. It yields control to any threads that are waiting to execute. Each waiting thread that executes in turn, of course, also has a yield call in it, so control eventually returns to `DoYieldFilter`.

When control returns, `DoYieldFilter` calls another event filter procedure, `MyStdFilterProc`. If no events have occurred, it simply returns to the `ModalDialog` function, which loops through again and calls the `DoYieldFilter` function, enabling the working threads to gain control again. If an event does occur, `MyStdFilterProc` handles it and returns the result to `ModalDialog`. When a user chooses the OK or Cancel button, `ModalDialog` exits the loop.

Thread Manager

Keep in mind that when an operating-system event occurs, the Thread Manager always returns control to the main thread at the first scheduling opportunity. This means that if there are several threads in your application doing background work while the dialog box is being displayed, at the first scheduling opportunity after an operating-system event occurs (and if the yield or other call causing the reschedule does not specify a particular thread to schedule next), the Thread Manager schedules the main thread no matter which threads are ahead of it in the scheduling queue. For this reason it is best to put event handling functions in the main thread.

Passing Input and Output Parameters to a New Thread

When you create a new thread, you can pass data to it by passing a parameter to the thread entry function. You can also retrieve data from the thread when it terminates. You set up the storage for this data when you create the thread.

Listing 1-9 shows how to pass data to a newly created thread and create the storage to hold the data returned by the new thread when it terminates.

Listing 1-9 Passing data between threads

```
#define kNoCreationOptions          0    /* Use the standard default
                                         creation options    */
#define kDefaultThreadStackSize    0    /* Use the default value*/

/* Define a structure */
struct ExampleRecord {
    long        someLongValue;
    short       someShortValue;
};
typedef struct ExampleRecord ExampleRecord;
typedef ExampleRecord *ExampleRecordPtr;

void MyParametersExample (void)
{
    ThreadID          tempThreadID;
    OSErr             err;
    long              myLong;
    short             myShort;
    Boolean           notDone = true;
```

CHAPTER 1

Thread Manager

```
ExampleRecordPtr      recordOutResult = nil;      /* Declare a variable to
output */              store new thread's
ExampleRecord          recordInParam;            /* Declare a variable to
pass                                                            data to a new thread */

/* Assign values to pass to a new thread */
recordInParam.someLongValue = 0x1FFF2EEE;
recordInParam.someShortValue = 0xABCD;

/* Create a new thread */
err = NewThread(kCooperativeThread,
                (ThreadEntryProcPtr)(MyExampleFunc),
                (void*)&recordInParam,
                kDefaultThreadStackSize,
                kNoCreationOptions,
                (void**)&recordOutResult,
                &tempThreadID);

if (err)
    DebugStr("\p Could not make coop thread 2");

while (notDone)
{
    YieldToAnyThread(); /* Other threads run. */

    if (recordOutResult != nil)
    {
        myLong = recordOutResult->someLongValue;      /* Store thread
output */
        myShort = recordOutResult->someShortValue;    /* Store thread
output */
        DoStuffWithParams(myLong, myShort);          /* Use thread
output */
        DisposePtr((Ptr)recordOutResult);            /* Remove storage
*/
        recordOutResult = nil;                        /* Neutralize
variable */
    }
}

/* Handle user events until quit time */
GoHandleEvents(&notDone);
```

Thread Manager

```

    }
    return; /* Done. */
}
/* Thread entry function */
pascal ExampleRecordPtr MyExampleFunc (ExampleRecordPtr inputRecordParam)
{
    ExampleRecordPtr    myRecordPtr;

    myRecordPtr = NewPtr(sizeof(ExampleRecord));
    myRecordPtr->someLongValue = inputRecordParam->someLongValue;
    myRecordPtr->someShortValue = inputRecordParam->someShortValue;

/* Do some calculations on the data and put the result in myRecordPtr */
    ...
    return (myRecordPtr);/* Must be the size of a void*. */
}

```

The first thing the code in Listing 1-9 does is to define some symbolic variables to make the code easier to read. When you create a thread with the `NewThread` function, you can specify some options that define the behavior of the thread, and you must specify a stack size for the thread. The two `#define` statements define variables that specify to use the default options and to use the default stack size.

The `ExampleRecord` structure defines a type of structure that later is used to pass a long and a short value to a new thread and then back again. The code creates the `ExampleRecord` type and also a pointer to it.

The `MyParametersExample` function performs the major work in this example. It first declares some variables, including `recordInParam` and `recordOutResult`. Note that `recordInParam`, which is used to pass data to a newly created thread, is declared as an `ExampleRecord` structure, and `recordOutResult`, which is used to store data returned from the new thread, is declared as a pointer to an `ExampleRecord` structure.

Next, `MyParametersExample` assigns hex values to the `someShortValue` and `someLongValue` fields of the `recordInParam` structure. It then uses the `NewThread` function to create a new thread. It specifies `MyExampleFunc` as the thread's entry function and passes it the `recordInParam` structure. It also specifies `recordOutResult` as the storage for any data returned from the new thread. Note that `NewThread` passes `recordInParam` as a pointer to a value and `recordOutResult` as a pointer to an address. As you recall, `recordInParam` is defined as an

Thread Manager

`ExampleRecord` structure and `recordOutResult` as a pointer to an `ExampleRecord` structure.

The `MyParametersExample` function then sets up a `while` loop to see if the newly created thread has returned any data yet. The `YieldToAnyThread` function guarantees that the newly created thread—and any other thread in the application—gets time to run. The variables `myLong` and `myShort` hold the data that the new thread returns. The `DoStuffWithParams` function, whose code is not shown here, passes in these variables and does some additional work on the data. The Memory Manager `DisposePtr` function frees the memory used by the `recordOutResult` structure. Note that the `while` loop also contains a function to handle user events.

The `MyExampleFunc` function is the entry point to the thread that the `MyParametersExample` function created with the `NewThread` function. It declares `myRecordPtr` as an `ExampleRecordPtr` and then uses the Memory Manager `NewPtr` function to allocate a block of memory for it that is the size of an `ExampleRecord` structure. It then passes the hex values from the `NewThread` function to the `someLongValue` and `someshortValue` fields of the structure pointed to by `myRecordPtr`.

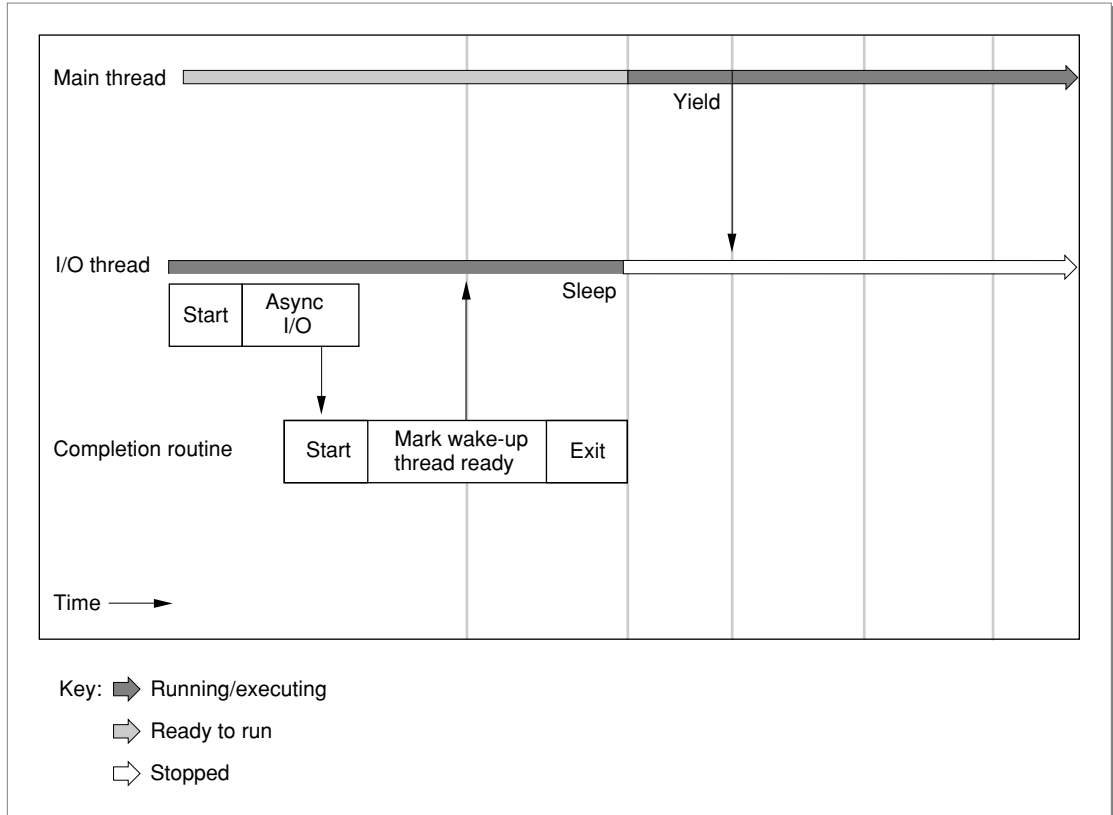
After doing some calculations on the hex values, `MyExampleFunc` returns the data to the storage allocated in the `MyParametersExample` function.

Using Threads With I/O

This section shows you one way to make an asynchronous I/O call from a threaded application. The straightforward way to do this is to create a separate thread that makes the I/O call and then puts itself to sleep so that other threads in the application can continue to work while the I/O request is being handled. You would also provide a completion routine that wakes up the stopped thread when the I/O task is complete.

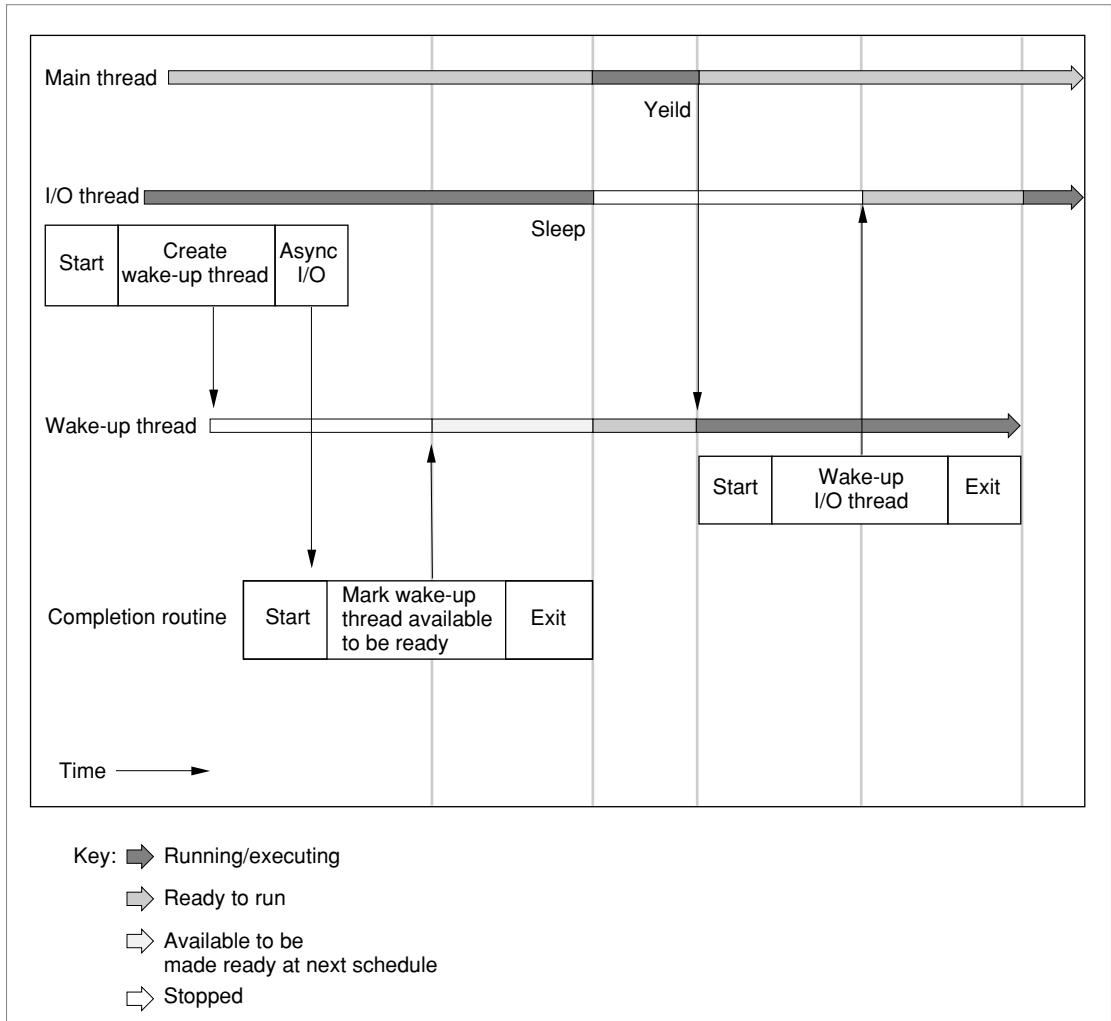
Figure 1-5 shows the problem with this approach. It is possible for the completion routine to execute before the thread puts itself in the stopped state. If this happens, the completion routine returns without doing anything because the thread is still running when the completion routine attempts to wake it up. Then the thread puts itself in the stopped state and stays there forever waiting for a completion routine that has already finished executing.

Figure 1-5 Using a completion routine to wake up a thread making an asynchronous I/O call



One solution to this problem is to create two threads, one to make the I/O call and the other to wake up the first thread. Figure 1-6 illustrates this process.

Figure 1-6 Using two threads to handle an asynchronous I/O call



As you can see in the figure, the thread making the I/O call creates a second thread (the wake-up thread) that is in the stopped state. The purpose of the completion routine is to start the wake-up thread. It doesn't actually make the thread ready to run but marks it as available to be ready to run. At the next

Thread Manager

scheduling opportunity, the wake-up thread is set to the ready-to-run state. At the following scheduling opportunity, if it is at the top of the queue, it begins to run and can wake up the I/O thread.

This scheme is guaranteed to work because there is no way that the I/O thread can still be awake when the completion routine attempts to wake it up. The wake-up thread can run only after the I/O thread has put itself to sleep. Listing 1-10 shows the code to implement this process.

Listing 1-10 Making an asynchronous I/O call with two threads

```

/* Set up parameter block */
struct ExtendedParamBlk {
    /* PB must be first so that the file system can get the data. */
    ParamBlockRec    pb;
    ThreadTaskRef    theAppTask;
    ThreadID         theThread;
};

typedef struct ExtendedParamBlk ExtendedParamBlk;
typedef ExtendedParamBlk *ExtendedParamBlkPtr;
/* Routine prototypes. */
pascal void    MyIOExampleThread (void);
pascal void DoWakeUpThread (ThreadID threadToWake);
void MyCompletionRoutine (void);
/* Completion routines are called with register A0 pointing to */
/* the parameter block. */
pascal ExtendedParamBlkPtr GetPBPtr(void) = {0x2E88};
                                     /* move.l a0, (sp) */

/* A routine in the main thread that creates a thread to make an I/O call */
void DoKickOffAnIOThread (void)
{
    ThreadID    newCoopID;
    OSErr      theError;

    theError = NewThread(kCooperativeThread,
                        (ThreadEntryProcPtr)MyIOExampleThread,
                        nil,
                        kDefaultThreadStackSize,
                        kNoCreationOptions,

```

Thread Manager

```

                                nil,
                                &newCoopID);
    if (theError)
        DebugStr("\p Could not make cooperative I/O thread");
    /* Return and let the I/O thread do its thing! */
}

/* The entry point for the code to make the I/O call */
pascal void MyIOExampleThread (void)
{
    ThreadID          wakeupThreadID, meThreadID;
    ThreadTaskRef     theAppRef;

    ExtendedParamBlk  myAsyncPB;
    OSErr             theError, theIOResult;

    /* Get the ID of MyIOExampleThread. */
    theError = MacGetCurrentThread(&meThreadID);
    if (theError != noErr)
        DebugStr("\pFailed to get the current thread ID");
    /* Get the application's task reference. */
    theError = GetThreadCurrentTaskRef(&theAppRef);
    if (theError != noErr)
        DebugStr("\Could not get our task ref");
    /* Create a wake-up thread. */
    theError = NewThread(kCooperativeThread,
                        (ThreadEntryProcPtr)DoWakeUpThread,
                        (void*)meThreadID,
                        kDefaultThreadStackSize,
                        kNewSuspend,
                        nil,
                        &wakeupThreadID);

    if (theError != noErr)
        DebugStr("\pFailed to create a cooperative thread");

    /* Prepare for and make the I/O call */
    myAsyncPB.pb.ioParam.ioCompletion = (ProcPtr)MyCompletionRoutine;
    myAsyncPB.pb.ioParam.ioResult = 0; /* Initialize the result. */
    myAsyncPB.pb.ioParam.ioNamePtr = nil; /* No name used here. */
    myAsyncPB.pb.ioParam.ioVRefNum = -1; /* The boot drive. */
    myAsyncPB.theThread = wakeupThreadID;
}

```

Thread Manager

```

myAsyncPB.theAppTask = theAppRef;
PBFFlushVol((ParmBlkPtr)&myAsyncPB, async);

/* Put I/O thread to sleep */
theError = SetThreadState(kCurrentThreadID, kStoppedThreadState,
                          kNoThreadID);

if (theError != noErr)
    DebugStr ("\pFailed to put current thread to sleep");

/* Get the result of the I/O operation */
theIOResult = myAsyncPB.pb.ioParam.ioResult;
    . . .
}
void MyCompletionRoutine (void)
{
    ExtendedParamBlkPtr    myAsyncPBPtr;
    ThreadTaskRef          theAppTaskRef;
    ThreadID               theThreadID;
    ThreadState            theThreadState;
    OSErr                  theError;
    /* Get the parameter block. */
    myAsyncPBPtr = GetPBPtr();
    /* Get the data. */
    theAppTaskRef = myAsyncPBPtr->theAppTask;
    theThreadID = myAsyncPBPtr->theThread;

    /* See if the thread is stopped yet - just to be sure. */
    theError = GetThreadStateGivenTaskRef(theAppTaskRef, theThreadID,
                                          &theThreadState);

    /* If we can get the thread state, go for it! */
    if (theError == noErr)
    {
        /* If it's not stopped, something is wrong. */
        if (theThreadState != kStoppedThreadState)
            DebugStr ("\pWake-up thread is in the wrong state!");
        /* Should be sleeping, mark it for wake up! */
        else
            SetThreadReadyGivenTaskRef(theAppTaskRef, theThreadID);
    }
}

```

Thread Manager

```

/* The wake up thread wakes up the I/O thread */
pascal void DoWakeUpThread (ThreadID threadToWake)
{
    OSErr      theError;

    theError = SetThreadState(threadToWake, kReadyThreadState,
                              kNoThreadID);

    if (theError != noErr)
        DebugStr("\pFailed to wake our thread");
    /* We've done our deed, so just return quietly and let it run. */
}

```

The code in Listing 1-10 is long but can be broken up into discreet parts. The first thing it does is to set up the parameter block. The extended parameter block holds the parameter block for use by the file system and has fields to hold the thread task reference and thread ID of the wake-up thread for use by the completion routine. After the parameter block declaration are prototypes for the entry functions to the I/O thread and the wake-up thread, and for the completion routine that marks the wake-up thread as ready. The inline routine `GetPBPtr` retrieves the address of the parameter block for the completion routine from register A0.

The `DoKickOffThread` function uses the `NewThread` function to create the cooperative wake-up thread. The entry point to this thread is the `MyIOExampleThread` function.

The `MyIOExampleThread` function does several things. It uses `MacGetCurrentThread` to get and store its thread ID. Next it gets the thread task reference for the application. The completion routine needs the thread task reference to make any Thread Manager calls to a thread in this application context because during execution of the completion routine, there is no guarantee as to which application is the current context. Then the `MyIOExampleThread` function creates the wake-up thread with the `NewThread` function. It specifies the `DoWakeUp` function as the entry point to the routine and passes its own thread ID as a parameter to this function. Note that the `kNewSuspend` option creates the new thread in the stopped state.

Next, the `MyIOExampleThread` function prepares for the I/O call by setting up the address of the completion routine and the extended data the completion routine requires, including the thread ID of the wake-up thread and the thread task reference for the current application. The actual I/O call is an asynchronous file system command.

Thread Manager

The last thing `MyIOExampleThread` does is to call `SetThreadState` to put itself in the stopped state. It passes the `kNoThreadID` constant as the last parameter to indicate that the Thread Manager should schedule the next available thread, rather than any particular thread.

IMPORTANT

It is always important to keep the main thread in the ready or running state, and the current example shows one of the reasons why. If the main thread has stopped itself, there may be no threads running at all after the current thread stops itself. The completion routine will return and mark the wake-up thread as available, but without a rescheduling call from the main thread or some other thread, the wake-up thread will remain marked as available but never ready or running. ▲

When the asynchronous I/O call completes, it calls the completion routine to indicate that it has finished. The completion routine retrieves the parameter block and gets the thread task reference for the application and the thread ID of the wake-up thread. For good measure, it uses the `GetThreadStateGivenTaskRef` function to verify that the wake-up thread is indeed stopped. It passes the thread task reference and the thread ID of the wake-up thread to this function. It then marks the wake-up thread as ready at the next reschedule with the `SetThreadReadyGivenTaskRef`. Again, it passes the thread task reference and the thread ID of the wake-up thread to this function.

At the next reschedule, the wake-up thread is made ready to run and eventually it begins executing. The entry point to this thread is the `DoWakeUpThread` function, which is passed the thread ID of the thread to wake—in this case, the thread ID of the I/O thread. The `DoWakeUpThread` function calls `SetThreadState` to change the state of the I/O thread from stopped to ready.

Thread Manager Reference

This section describes the data types and functions that are specific to the Thread Manager.

Data Types

This section describes the data types that the Thread Manager uses. These include data types to

- determine if the Thread Manager is available and which features are supported
- identify the state of a thread
- identify the application context from an interrupt or completion routine when your application is not guaranteed to be the current context
- specify the type of thread
- specify the ID of a thread
- specify options for the creation of a thread
- pass information to a custom scheduling function

Gestalt Selector and Response Bits

To determine if the Thread Manager is available and which features are supported, call the `Gestalt` function with the selector `gestaltThreadMgrAttr`. The `Gestalt` function returns information by setting or clearing bits in the `response` parameter. The following constants define the bits currently used.

```
enum {
#define gestaltThreadMgrAttr      'thds'      /* Thread Manager attributes */
    gestaltThreadMgrPresent      = 0,
    gestaltSpecificMatchSupport  = 1,
    gestaltThreadsLibraryPresent = 2
};
```

Thread Manager

Constant Descriptions

`gestaltThreadMgrPresent`

This bit is set if the Thread Manager is present.

`gestaltSpecificMatchSupport`

This bit is set if the Thread Manager supports the allocation of threads based on an exact match with the requested stack size. If this bit is not set, the Thread Manager allocates threads based on the closest match to the requested stack size.

`gestaltThreadsLibraryPresent`

This bit is set if the native version of the threads library has been loaded.

The Thread State

The Thread Manager provides various functions, such as `GetThreadState` and `SetThreadState`, to get and set information about the state of a thread. These functions use the `ThreadState` data type to get and set thread state information.

```
typedef unsigned short      ThreadState;
```

There are three possible values for the thread state:

```
#define kReadyThreadState      ((ThreadState) 0)
#define kStoppedThreadState    ((ThreadState) 1)
#define kRunningThreadState    ((ThreadState) 2)
```

Constant descriptions

`kReadyThreadState`

The thread is ready to run.

`kStoppedThreadState`

The thread is stopped and not ready to run.

`kRunningThreadState`

The thread is running.

The Thread Task Reference

In certain cases, such as during execution of an interrupt routine, your application is not guaranteed to be the current process. Since threads are defined within an application context, it follows that in cases such as these, you cannot get or set information about any particular threads in your application unless you have a way of identifying the application context. The thread task reference gives you a way of doing this.

You can obtain the thread task reference by calling `GetCurrentThreadTaskRef` at a time when you know your application is the current context. Later, during execution of an interrupt routine, you can use the thread task reference to identify your application. For example, you can pass the thread task reference to functions such as `GetThreadStateGivenTaskRef` and `SetThreadReadyGivenTaskRef` in an interrupt routine to get and set information about the state of particular threads in your application.

The `ThreadTaskRef` data type defines the thread task reference.

```
typedef void* ThreadTaskRef;
```

The Thread Type

Historically, the Thread Manager defined two types of threads to run in an application context: cooperative and preemptive, but now it supports only cooperative threads.

Although the Thread Manager only supports a single type of thread, many Thread Manager functions (for historical reasons) require you to use the thread type to specify the type of the thread.

The `ThreadStyle` data type specifies the type of a thread.

```
typedef unsigned long ThreadStyle;
```

Because there is only one type of thread (cooperative) the thread type accepts a single value:

```
#define kCooperativeThread (1<<0)
```

The Thread ID

The Thread Manager assigns a thread ID to each thread that you create or allocate with the `NewThread` function. The thread ID uniquely identifies a thread within an application context. You can use the thread ID in functions that schedule execution of a particular thread, dispose of a thread, and get and set information about a thread; for example, you pass the thread ID to functions such as `YieldToThread`, `DisposeThread`, and `GetThreadState`.

The `ThreadID` data type defines the thread ID.

```
typedef unsigned long          ThreadID;
```

The Thread Manager defines the following three constants that you can use in addition to the specific thread IDs that the `NewThread` function returns:

```
#define kNoThreadID           ((ThreadID) 0)
#define kCurrentThreadID     ((ThreadID) 1)
#define kApplicationThreadID ((ThreadID) 2)
```

Constant descriptions

`kNoThreadID` Indicates no thread; for example, you can use a function such as `SetThreadState` to put the current thread in the stopped state and pass `kNoThreadID` to indicate that you don't care which thread runs next.

`kCurrentThreadID` Identifies the currently executing thread.

`kApplicationThreadID` Identifies the main application thread; this is the cooperative thread that the Thread Manager creates at launch time. You cannot dispose of this thread. All applications—even those that are not aware of the Thread Manager—have one main application thread. The Thread Manager assumes that the main application thread is responsible for event gathering; when operating-system event occurs, the Thread Manager schedules the main application thread as the next thread to execute.

Thread Options

When you create or allocate a new thread with the `NewThread` function, you can specify thread options that define certain characteristics of the thread. The `ThreadOptions` data type defines the thread options.

```
typedef unsigned long          ThreadOptions;
```

To specify more than one option, you sum them together and pass them as a single parameter to the `NewThread` function.

```
#define kNewSuspend             (1<<0)
#define kUsePremadeThread      (1<<1)
#define kCreateIfNeeded        (1<<2)
#define kFPUNotNeeded          (1<<3)
#define kExactMatchThread      (1<<4)
```

Constant descriptions

<code>kNewSuspend</code>	Begin a new thread in the stopped state.
<code>kUsePremadeThread</code>	Use a thread from the existing supply.
<code>kCreateIfNeeded</code>	Create a new thread if one with the proper style and stack size requirements does not exist.
<code>kFPUNotNeeded</code>	Do not save the FPU context. This saves time when switching contexts. Note, however, that for PowerPC threads, the Thread Manager always saves the FPU registers regardless of how you set this option. Because the PowerPC microprocessor uses the FPU registers for optimizations, they could contain necessary information.
<code>kExactMatchThread</code>	Allocate a thread from the pool only if it exactly matches the stack-size request. Without this option, a thread is allocated that best fits the request—that is, a thread whose stack is greater than or equal to the requested size.

The Scheduler Information Structure

You can, if you wish, use the `SetThreadScheduler` function to install a custom scheduling function to work in conjunction with the default Thread Manager scheduling mechanism. The Thread Manager uses the scheduler information structure to pass information to the custom scheduling function that allows it to decide which thread, if any, to schedule next.

```
struct SchedulerInfoRec {
    unsigned long      InfoRecSize;
    ThreadID          CurrentThreadID;
    ThreadID          SuggestedThreadID;
    ThreadID          InterruptedCoopThreadID;
};
typedef struct SchedulerInfoRec SchedulerInfoRec;
typedef SchedulerInfoRec *SchedulerInfoRecPtr;
```

Field descriptions

<code>InfoRecSize</code>	The size of the structure.
<code>CurrentThreadID</code>	The thread ID of the current thread.
<code>SuggestedThreadID</code>	The thread ID of the thread that the application has suggested to run.
<code>InterruptedCoopThreadID</code>	Historically, the thread ID of a preempted cooperative thread if a cooperative thread has been interrupted and has not yet resumed execution. Because it no longer supports preemptive threads, the Thread Manager always passes the constant <code>kNoThreadID</code> to indicate that there is no thread that has been interrupted.

Thread Manager Functions

You can use Thread Manager functions to perform the following tasks:

- create and get information about pools of threads
- create and delete individual threads
- get information about individual threads

Thread Manager

- schedule threads
- disable scheduling
- get information about and schedule threads from interrupt code
- install custom scheduler, context switcher, termination, and debugging functions

Creating and Getting Information About Thread Pools

This section describes functions that allow you to create a pool of threads and to get information about the threads, such as the number of threads of a particular stack size that are available or the default stack requirement for a thread.

CreateThreadPool

You can use the `CreateThreadPool` function to create a pool of threads for your application.

```
pascal OSErr CreateThreadPool(ThreadStyle threadStyle,
                             short numToCreate, Size stackSize);
```

`threadStyle` The type of thread to create for this set of threads in the pool. Cooperative is the only type that you can specify. Historically, the Thread Manger supported two types of threads, preemptive and cooperative. However, due to severe limitations on their use, the Thread Manager no longer supports preemptive threads.

`numToCreate` The number of threads to create for the pool.

`stackSize` The stack size for this set of threads in the pool. This stack must be large enough to handle saved thread context, normal application stack usage, interrupt handling routines, and CPU exceptions. Specify a stack size of 0 to request the Thread Manager's default stack size for the specified type of thread.

Thread Manager

DESCRIPTION

The `CreateThreadPool` function creates the specified number of threads with the specified stack requirements. It places the threads that it creates into a pool for use by your application.

When you call `CreateThreadPool`, if the Thread Manager is unable to create all the threads that you specify, it doesn't create any at all and returns the `memFullErr` result code.

The threads in the pool are indistinguishable except by stack size. That is, you cannot refer to them individually. When you want to use a thread to execute some code in your application, you allocate a thread of a specific size from the pool using the `NewThread` function. The `NewThread` function assigns a thread ID to the thread and specifies the function that is the entry point to the thread.

Note that it is not strictly necessary to create a pool of threads before allocating a thread. If you wish, you can use the `NewThread` function to create and allocate a thread in one step. The advantage of using `CreateThreadPool` is that you can allocate memory for threads early in your application's execution before memory is used or fragmented.

IMPORTANT

Before making any calls to `CreateThreadPool`, be certain that you first have called the Memory Manager function `MaxApp1Zone` to extend the application heap to its limit. You must call `MaxApp1Zone` from the main application thread before any other threads in your application run. ▲

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0501</code>

RESULT CODES

<code>noErr</code>	0	Specified threads were created and are available
<code>paramErr</code>	-50	Unknown thread type; or specified a preemptive thread without architecture support
<code>memFullErr</code>	-108	Insufficient memory to create the thread structures

SEE ALSO

To allocate a thread from the pool created with `CreateThreadPool`, use the `NewThread` function described on (page 56).

To determine how many threads in the pool are available for allocation, use the `GetFreeThreadCount` function described on (page 52). To determine how many threads of a particular stack size are available, use the `GetSpecificFreeThreadCount` function described on (page 53).

GetFreeThreadCount

You can use the `GetFreeThreadCount` function to determine how many threads are available to be allocated in a thread pool.

```
pascal OSErr GetFreeThreadCount(ThreadStyle threadStyle,
                                short *freeCount);
```

`threadStyle`

The type of thread to get information about. Cooperative is the only type that you can specify. Historically, the Thread Manger supported two types of threads, preemptive and cooperative, but the Thread Manager no longer supports preemptive threads.

`freeCount`

A pointer to the number of threads available to be allocated.

DESCRIPTION

The `GetFreeThreadCount` function determines how many threads are available to be allocated. The number of threads in the pool varies throughout execution of your application. Calls to `CreateThreadPool` add threads to the pool and calls to `NewThread`, when an existing thread is allocated, reduce the number of threads. You also add threads to the pool when you dispose of a thread with the `DisposeThread` function and specify that the thread be recycled.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0402</code>

RESULT CODES

<code>noErr</code>	0	The number of available threads was returned
<code>paramErr</code>	-50	Unknown thread type; or specified a preemptive thread without architecture support

SEE ALSO

Use the `GetSpecificFreeThreadCount` function (described next) to determine how many threads of a particular *stack size* are available.

GetSpecificFreeThreadCount

You can use the `GetSpecificFreeThreadCount` function to determine how many threads of a specified stack size are available to be allocated in a thread pool.

```
pascal OSErr GetSpecificFreeThreadCount
    (ThreadStyle threadStyle,
     Size stackSize, short *freeCount);
```

`threadStyle`

The type of thread to get information about. Cooperative is the only type that you can specify. Historically, the Thread Manger supported two types of threads, preemptive and cooperative, but the Thread Manager no longer supports preemptive threads.

`stackSize`

The stack size of the threads to get information about.

`freeCount`

A pointer to the number of threads of the specified stack size available to be allocated.

DESCRIPTION

The `GetSpecificFreeThreadCount` function determines how many threads with a stack size equal to or greater than the specified size are available to be allocated. Use this function instead of `GetFreeThreadCount` when you are interested not simply in the total number of available threads but when you want to know the number of available threads of a specified stack size as well.

The number of threads in the pool varies throughout execution of your application. Calls to `CreateThreadPool` add threads to the pool and calls to `NewThread`, when an existing thread is allocated, reduce the number of threads. You also add threads to the pool when you dispose of a thread with the `DisposeThread` function and specify that the thread be recycled.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0615</code>

RESULT CODES

<code>noErr</code>	0	The number of available threads of the specified style and stack size was returned
<code>paramErr</code>	-50	Unknown thread type; or specified a preemptive thread without architecture support

SEE ALSO

To determine how many threads of *any* stack size are available, use the `GetFreeThreadCount` function ((page 52)).

GetDefaultThreadStackSize

You can use the `GetDefaultThreadStackSize` function to determine the default stack size required by a thread.

```
pascal OSErr GetDefaultThreadStackSize(ThreadStyle threadStyle,
                                       Size *stackSize);
```

Thread Manager

<code>threadStyle</code>	The type of thread to get information about. Cooperative is the only type that you can specify. Historically, the Thread Manger supported two types of threads, preemptive and cooperative, but the Thread Manager no longer supports preemptive threads.
<code>stackSize</code>	A pointer to the default stack size (in bytes) returned by the Thread Manager. The <code>GetDefaultThreadStackSize</code> function places this value in the variable that you pass to it. When you create a thread pool or an individual thread, this is the stack size that the Thread Manager allocates when you specify the default size.

DESCRIPTION

The `GetDefaultThreadStackSize` function returns, in the `stackSize` parameter, the default stack size required by a thread in your application. The Thread Manager determines the default stack size.

Keep in mind that the default stack size is not an absolute value that you must use but is a rough estimate.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0413</code>

RESULT CODES

<code>noErr</code>	0	The proper default stack size was returned for the specified style of thread
<code>paramErr</code>	-50	Unknown thread type; or specified a preemptive thread without architecture support

SEE ALSO

To determine how much stack space is available for a particular thread, use the `ThreadCurrentStackSpace` function described on (page 61).

Creating and Disposing of Threads

This section describes functions that allow you to create or allocate threads and to dispose of them when the code they contain has finished executing.

NewThread

You can use the `NewThread` function to create or allocate a thread with particular characteristics.

```
pascal OSErr NewThread(ThreadStyle threadStyle,
                      ThreadEntryProcPtr threadEntry,
                      void *threadParam,
                      Size stackSize,
                      ThreadOptions options,
                      void **threadResult,
                      ThreadID *threadMade);
```

`threadStyle`

The type of thread to create. Cooperative is the only type that you can specify. Historically, the Thread Manager supported two types of threads, preemptive and cooperative, but the Thread Manager no longer supports preemptive threads.

`threadEntry` A pointer to the thread entry function.

`threadParam`

A pointer to a value that the Thread Manager passes as a parameter to the thread entry function. Specify `nil` if you are passing no information.

`stackSize`

The stack size (in bytes) to allocate for this thread. This stack must be large enough to handle saved thread context, normal application stack usage, interrupt handling routines, and CPU exceptions. Specify a stack size of 0 (zero) to request the Thread Manager's default stack size.

`options`

Options that define characteristics of the new thread. See the `ThreadOptions` data type ((page 48)) for details on the options. You sum the options together to create a single `options` parameter.

Thread Manager

`threadResult`

A pointer to the address of a location to hold the function result that is returned by the `DisposeThread` function when the thread terminates. Specify `nil` for this parameter if you are not interested in the function result.

`threadMade`

A pointer to the thread ID of the newly created or allocated thread that the `NewThread` function returns through this parameter. If there is an error, `NewThread` sets the value of `threadMade` to `kNoThreadID`.

DESCRIPTION

The `NewThread` function creates a new thread or allocates one from the existing pool of threads. It returns a thread ID that you can use in other Thread Manager functions to identify the thread. If you want to allocate a thread from the pool of threads, specify the `kUsePremadeThread` option of the `options` parameter. Otherwise, `NewThread` creates a new thread.

When you request a thread from the existing pool, the Thread Manager allocates one that best fits your specified stack size. If you specify the `kExactMatchThread` option of the `options` parameter, the Thread Manager allocates a thread whose stack exactly matches your stack-size requirement or, if it can't allocate one because no such thread exists, it returns the `threadTooManyReqsErr` result code.

IMPORTANT

Before making any calls to `NewThread`, be certain that you first have called the Memory Manager function `MaxApp1Zone` to extend the application heap to its limit. You must call `MaxApp1Zone` from the main application thread before any other threads in your application run. ▲

When you call the `NewThread` function, you pass, as the `threadEntry` parameter, a pointer to the name of the entry function to the thread. When the newly created thread runs initially, it begins by executing this function.

You can use the `threadParam` parameter to pass thread-specific information to a newly created or allocated thread. In the data structure pointed to by this parameter, you could place something like A5 information or the address of a window to update. You could also use this parameter to specify a place for a thread's local storage.

IMPORTANT

Be sure to create the storage for the `threadResult` parameter in a place that is guaranteed to be available when the thread terminates—for example, in an application global variable or in a local variable of the application’s main function (the main thread, by definition, cannot be disposed of so it is always available). Do not create the storage in a local variable of a subroutine that completes before the thread terminates or the storage will become invalid. ▲

SPECIAL CONSIDERATIONS

Do not pass a routine descriptor as the `threadEntry` parameter to the `NewThread` function. For all Thread Manager functions that pass a procedure pointer, such as this one, you must pass the address of the routine, not the address of a routine descriptor. You are required to use routine descriptors when you write code in the PowerPC instruction set that passes a routine’s address to code that might be in the 680x0 instruction set. However, the threads in your application must all be in the same instruction set— 680x0 or PowerPC. Therefore, the routine identified by the `threadEntry` parameter is by definition in the same instruction set as the `NewThread` function and a routine descriptor is not required.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0E03</code>

RESULT CODES

<code>noErr</code>	0	Specified thread was made or allocated
<code>paramErr</code>	-50	Unknown thread type; or specified a preemptive thread without architecture support
<code>memFullErr</code>	-108	Insufficient memory to create the thread structures
<code>threadTooManyReqsErr</code>	-617	No matching thread structures available

SEE ALSO

To dispose of a thread, use the `DisposeThread` function described next.

See the description of the `ThreadOptions` data type on (page 48) for details on the characteristics you can specify in the `options` parameter.

For more information about the thread entry function, see the `myThreadEntry` function described on (page 85).

DisposeThread

You can use the `DisposeThread` function to delete a thread when it finishes executing.

```
pascal OSErr DisposeThread(ThreadID threadToDump,
                          void *threadResult,
                          Boolean recycleThread);
```

`threadToDump`

The thread ID of the thread to delete.

`threadResult`

A pointer to the thread's result. The `DisposeThread` function returns this result to an address which you originally specify with the `threadResult` parameter of the `NewThread` function when you create or allocate the thread. Pass a value of `nil` (0) if you are not interested in returning a function result.

`recycleThread`

A Boolean value that specifies whether to return the thread to the allocation pool or to remove it entirely. Specify `False` (0) to dispose of the thread entirely and `True` (1) to return it to the thread pool.

DESCRIPTION

When a thread finishes executing, the Thread Manager automatically calls `DisposeThread` to delete it. Therefore, the only reason for you to explicitly call `DisposeThread` is to recycle a terminating thread. To do so, set the `recycleThread` parameter to `True` (1). The Thread Manager clears out the thread's internal data

Thread Manager

structure, resets it, and puts the thread in the thread pool where it can be used again as necessary.

The `DisposeThread` function returns the thread's function result in the `threadResult` parameter. You allocate the storage for the thread result when you create or allocate a thread with the `NewThread` function. See "Passing Input and Output Parameters to a New Thread" beginning on page 1-33 for an example of how to set up storage for the thread result when you create a new thread.

IMPORTANT

You cannot explicitly dispose of the main application thread. If you attempt to do so, `DisposeThread` returns the `threadProtocolErr` result code. ▲

When your application terminates, the Thread Manager calls `DisposeThread` to terminate any active threads. It terminates stopped and ready threads first but in no special order. It terminates the currently running thread last. This thread should always be the main application thread.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0504</code>

RESULT CODES

<code>noErr</code>	0	Specified thread was disposed of
<code>threadNotFoundErr</code>	-618	No thread with the specified thread ID
<code>threadProtocolErr</code>	-619	The thread specified in <code>threadToDump</code> was the application thread

SEE ALSO

To install a callback function to do special cleanup when a thread terminates, use the `SetThreadTerminator` function described on (page 81).

Getting Information About Specific Threads

This section describes functions that allow you to get information about a specific thread.

ThreadCurrentStackSize

You can use the `ThreadCurrentStackSize` function to determine the amount of stack space that is available for any thread in your application.

```
pascal OSErr ThreadCurrentStackSize(ThreadID thread,
                                   unsigned long *freeStack);
```

<code>thread</code>	The thread ID of the thread about which you want information.
<code>freeStack</code>	A pointer to the amount of stack space (in bytes) that is available to the specified thread. The <code>ThreadCurrentStackSize</code> function returns this information.

DESCRIPTION

The `ThreadCurrentStackSize` function returns the amount of stack space (in bytes) that is available for a specified thread.

This function is primarily useful during debugging since you determine the maximum amount of stack space you need for any particular thread before you ship your application. However, if your application calls a recursive function that could call itself many times, you might want to use `ThreadCurrentStackSize` to keep track of the stack space and take appropriate action if it becomes too low.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0414</code>

RESULT CODES

<code>noErr</code>	0	Amount of stack space available in the thread was returned
<code>threadNotFoundErr</code>	-618	No thread with the specified thread ID

SEE ALSO

To determine the default size that the Thread Manager assigns to threads use the `GetDefaultThreadStackSize` function described on (page 54).

For information on how to optimize memory use in a threaded application, see the section “Thread Stacks” (page 17).

MacGetCurrentThread

You can use the `MacGetCurrentThread` function to obtain the thread ID of the currently executing thread.

```
pascal OSErr MacGetCurrentThread (ThreadID *currentThreadID);

currentThreadID
```

A pointer to the thread ID of the current thread. The `MacGetCurrentThread` function returns this information.

DESCRIPTION

The `MacGetCurrentThread` function retrieves the thread ID of the currently executing thread and places it in the `currentThreadID` parameter. You can use the thread ID in functions such as `GetThreadState` and `SetThreadState` to get and set the state of a thread.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0206</code>

RESULT CODES

<code>noErr</code>	0	ID of the current thread was returned
<code>threadNotFoundErr</code>	-618	No current thread

SEE ALSO

The `ThreadID` data type is described on (page 47).

GetThreadState

You can use the `GetThreadState` function to obtain the state of any thread.

```
pascal OSErr GetThreadState(ThreadID threadToGet,
                             ThreadState *threadState);
```

`threadToGet`

The thread ID of the thread about which you want information.

`threadState`

A pointer to a `ThreadState` data structure in which `GetThreadState` places the information about the state of the specified thread.

DESCRIPTION

The `GetThreadState` function returns the state of the specified thread. A thread can be in one of three states: ready to execute (`kThreadReadyState`), stopped (`kStoppedThreadState`), or executing (`kRunningThreadState`).

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0407</code>

RESULT CODES

<code>noErr</code>	0	State of the specified thread was returned
<code>threadNotFoundErr</code>	-619	No thread with the specified thread ID

SEE ALSO

To change the state of a specified thread, use `SetThreadState` described on (page 67).

The `ThreadState` data type is described on (page 45).

Scheduling Threads

This section describes functions that allow you to control the execution of threads.

YieldToAnyThread

You can use the `YieldToAnyThread` function to relinquish the current thread's control.

```
pascal OSErr YieldToAnyThread(void);
```

DESCRIPTION

The `YieldToAnyThread` function invokes the Thread Manager's scheduling mechanism. The current thread relinquishes control and the Thread Manager schedules the next available thread.

The current thread is suspended in the ready state and awaits rescheduling when the CPU is available. When the suspended thread is scheduled again, `YieldToAnyThread` regains control and returns to the function that called it.

If you have installed a custom scheduler, the Thread Manager passes it the thread ID of the suspended thread.

In each thread you must make one or more strategically placed calls to relinquish control to another thread. You can either make this yield call or another yield call such as `YieldToThread`; or you can make a call such as `SetThreadState` to explicitly change the state of the thread.

SPECIAL CONSIDERATIONS

Threads must yield in the CPU addressing mode (24-bit or 32-bit) in which the application was launched.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$303C</code>

RESULT CODES

<code>noErr</code>	0	Current thread has yielded
<code>threadProtocolErr</code>	-619	Current thread is in a critical section

SEE ALSO

To relinquish control to a specific thread, use the `YieldToThread` function described next.

To change the state of a specified thread, use the `SetThreadState` function described on (page 67).

For more information on how the Thread Manager schedules threads, see “Scheduling” beginning on page 1-11.

YieldToThread

You can use the `YieldToThread` function to relinquish the current thread’s control to a particular thread.

```
pascal OSErr YieldToThread(ThreadID suggestedThread);
```

```
suggestedThread
```

The ID of the thread to yield control to.

DESCRIPTION

The `YieldToThread` function invokes the Thread Manager’s scheduling mechanism. The current thread relinquishes control and passes the thread ID of a thread for the Thread Manager to schedule. The Thread Manager schedules this thread if it is available. Otherwise, the Thread Manager schedules the next available thread.

Thread Manager

The current thread is suspended in the ready state and awaits rescheduling when the CPU is available. When the suspended thread is scheduled again, `YieldToThread` regains control and returns to the function that called it.

If you have installed a custom scheduler, the Thread Manager passes it the thread ID of the suspended thread.

In each thread you must make one or more strategically placed calls to relinquish control to another thread. You can either make this yield call or another yield call such as `YieldToAnyThread`; or you can make a call such as `SetThreadState` to explicitly change the state of the thread.

SPECIAL CONSIDERATIONS

Threads must yield in the CPU addressing mode (24-bit or 32-bit) in which the application was launched.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0205</code>

RESULT CODES

<code>noErr</code>	0	Current thread has yielded and is now running again
<code>threadNotFoundErr</code>	-618	No thread with the specified thread ID or the suggested thread is not in a ready state
<code>threadProtocolErr</code>	-619	Current thread is in a critical section

SEE ALSO

To relinquish control without naming a specific thread, use the `YieldToAnyThread` function described on (page 64).

To change the state of a specified thread, use the `SetThreadState` function described on next.

For more information on how the Thread Manager schedules threads, see “Scheduling” beginning on page 1-11.

SetThreadState

You can use the `SetThreadState` function to change the state of any thread.

```
pascal OSErr SetThreadState(ThreadID threadToSet,
                            ThreadState newState,
                            ThreadID suggestedThread);
```

`threadToSet`

The thread ID of the thread whose state is to be changed.

`newState`

The new state for the thread. You can specify `ready to execute` (`kThreadReadyState`), `stopped` (`kStoppedThreadState`), or `executing` (`kRunningThreadState`).

`suggestedThread`

The thread ID of the next thread to run. You specify this thread if you are changing the state of the currently executing thread to stopped or ready to run. Pass `kNoThreadID` if you do not want to specify a particular thread to run next. In this case, the Thread Manager schedules the next available thread to run.

DESCRIPTION

The `SetThreadState` function changes the state of the specified thread. The effect of `SetThreadState` depends on whether the thread you specify for changing is the currently executing thread or another thread. If you specify the current thread and thus change the state to stopped or ready, `SetThreadState` invokes the Thread Manager scheduling mechanism. The current thread relinquishes control (it is put in the state you specify, stopped or ready) and the Thread Manager schedules the thread that you specify with the `suggestedThread` parameter. If this thread is unavailable for running, or if you passed `kNoThreadID`, the Thread Manager schedules the next available thread.

If you change the state of the current thread to ready, the Thread Manager suspends it awaiting availability of the CPU. When it is rescheduled, `SetThreadState` regains control and returns to the function that called it.

If you have installed a custom scheduler, the Thread Manager passes it the thread ID of the suspended thread.

If you specify a thread other than the currently executing thread, no rescheduling occurs. If you change the state from ready to stopped, the thread is

Thread Manager

removed from the scheduling queue. The Thread Manager does not schedule this thread for execution again until you change its state to ready. On the other hand, if you change the state from stopped to ready, you have in effect put the thread in the scheduling queue, and the Thread Manager gives it CPU time as soon as it reaches the top of the scheduling queue.

SPECIAL CONSIDERATIONS

Threads must yield in the CPU addressing mode (24-bit or 32-bit) in which the application was launched.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0508</code>

RESULT CODES

<code>noErr</code>	0	Thread was put in the specified state; if this was the current thread, it is now running again
<code>threadNotFoundErr</code>	-618	No thread with the specified thread ID, or the suggested thread is not in a ready state
<code>threadProtocolErr</code>	-619	Specified thread is in a critical section, or the new state that was specified is an invalid state

SEE ALSO

To obtain the state of any thread, use the `GetThreadState` function described on (page 63).

To relinquish control to the next available thread, use the `YieldToAnyThread` function described on (page 64). To relinquish control to a specific thread, use the `YieldToThread` function described on (page 65).

The `ThreadState` data structure is described on (page 45).

For more information on how the Thread Manager schedules threads, see “Scheduling” beginning on page 1-11.

To set the state of the current thread before it exits a critical section of code, use the `SetThreadStateEndCritical` function described on (page 71).

Preventing Scheduling

This section describes functions that allow you to turn scheduling off and back on again.

ThreadBeginCritical

You can use the `ThreadBeginCritical` function to indicate that the thread is entering a critical code section.

```
pascal OSErr ThreadBeginCritical(void);
```

DESCRIPTION

The `ThreadBeginCritical` function disables scheduling by marking the beginning of a section of critical code. That is, no other threads in the current application can run—even if the current thread yields control—until the current thread exits the critical section (by calling the `ThreadEndCritical` function). Disabling scheduling allows the currently executing function to look at or change shared or global data safely. You can nest critical sections within a thread.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$000B</code>

RESULT CODES

<code>noErr</code>	0	Current thread can now execute critical section
--------------------	---	---

SEE ALSO

To mark the end of a critical code section and turn scheduling back on, use the `ThreadEndCritical` function (described next). If you also need to set the state of the current thread before scheduling is turned back on, use the `SetThreadStateEndCritical` function (described on (page 71)).

ThreadEndCritical

You can use the `ThreadEndCritical` function to indicate that the thread is leaving a critical code section.

```
pascal OSErr ThreadEndCritical(void);
```

DESCRIPTION

The `ThreadEndCritical` function turns scheduling back on by indicating that the thread is exiting a critical section of code. All scheduling operations are now available to the application.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$000C</code>

RESULT CODES

<code>noErr</code>	0	Current thread is now out of most nested critical section
<code>threadProtocolErr</code>	-619	Current thread is not in a critical section

SEE ALSO

Use the `ThreadBeginCritical` function (described on (page 69)) to mark the beginning of a critical code section and turn scheduling off.

If you need to set the state of the current thread before scheduling is turned back on, use the `SetThreadStateEndCritical` function described next.

SetThreadStateEndCritical

You can use the `SetThreadStateEndCritical` function to change the state of the current thread and exit that thread's critical section at the same time.

```
pascal OSErr SetThreadStateEndCritical(ThreadID threadToSet,
                                       ThreadState newState,
                                       ThreadID suggestedThread);
```

`threadToSet`

The thread ID of the thread whose state is to be changed.

`newState`

The new state for the thread. You can specify `ready to execute` (`kThreadReadyState`), `stopped` (`kStoppedThreadState`) or `executing` (`kRunningThreadState`).

`suggestedThread`

The thread ID of the next thread to run. You specify this thread if you are changing the state of the currently executing thread to stopped or ready to run. Pass `kNoThreadID` if you do not want to specify a particular thread to run next. In this case, the Thread Manager schedules the next available thread to run.

DESCRIPTION

The `SetThreadStateEndCritical` function does in one step the same thing that `ThreadEndCritical` and `SetThreadState` function do in two steps; that is, change the state of the thread and exit the thread's critical section.

Note

Historically, the primary purpose of the `SetThreadStateEndCritical` function was to close the scheduling window at the end of a critical section. A preemptive thread that was waiting while the critical section of code was executing could begin executing before you changed the state of the current thread to stopped with the `SetThreadState` function. Obviously, because the Thread Manager no longer supports preemptive threads, this function is no longer necessary to close the scheduling window, but you can still use it to change the state of a thread and exit a critical section in one step instead of two.

Thread Manager

When you change the state of the currently executing thread, the Thread Manager schedules the thread you specify with the `suggestedThread` parameter. If this thread is unavailable or if you pass `kNoThreadID`, the Thread Manager schedules the next available thread.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0512</code>

RESULT CODES

<code>noErr</code>	0	Thread was put in the specified state; if this was the current thread, it is now running again
<code>threadNotFoundErr</code>	-618	No thread with the specified thread ID, or the suggested thread is not in a ready state
<code>threadProtocolErr</code>	-619	Current thread is not in a critical section, or the new state that was specified is an invalid state

SEE ALSO

To mark a section of code as critical, use the `ThreadBeginCritical` described on (page 69) and the `ThreadEndCritical` function described on (page 70).

To change the state of any thread, use the `SetThreadState` function described on (page 67).

For more information on how the Thread Manager schedules threads, see “Scheduling” beginning on page 1-11.

Getting Information and Scheduling Threads During Interrupts

This section describes functions that allow you to get information about threads and to schedule threads at times when your application is not necessarily the current process, such as during the execution of interrupt code.

GetThreadCurrentTaskRef

You can use the `GetThreadCurrentTaskRef` function to obtain a thread task reference.

```
pascal OSErr GetThreadCurrentTaskRef (ThreadTaskRef *threadTRef);
```

`threadTRef` A pointer to a thread task reference, which the `GetThreadCurrentTaskRef` function returns.

DESCRIPTION

The `GetThreadCurrentTaskRef` function returns a thread task reference. The thread task reference is somewhat of a misnomer because it identifies your application context, not a particular thread. Identifying your application context is necessary in situations where you aren't guaranteed that your application is the current context—such as during the execution of an interrupt routine. In such cases, you need both the thread ID to identify the thread and the thread task reference to identify the application context.

After you obtain the thread task reference, you can use it in the `GetThreadStateGivenTaskRef` and `SetThreadReadyGivenTaskRef` functions to get and set information about specific threads in your application at times when you are not guaranteed that your application is the current context.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$020E</code>

RESULT CODES

<code>noErr</code>	0	Thread task reference was returned
--------------------	---	------------------------------------

SEE ALSO

To get information about a thread when your application is not the current process, use the `GetThreadStateGivenTaskRef` function described next.

To change the state of a thread from stopped to ready when your application is not the current process, use the `SetThreadReadyGivenTaskRef` function described on (page 75).

The `ThreadTaskRef` data type is described on (page 46).

GetThreadStateGivenTaskRef

You can use the `GetThreadStateGivenTaskRef` function to obtain the state of a thread when your application is not necessarily the current process—for example, during execution of an interrupt routine.

```
pascal OSErr GetThreadStateGivenTaskRef(ThreadTaskRef threadTRef,
                                         ThreadID threadToGet,
                                         ThreadState *threadState);
```

`threadTRef` The thread task reference of the application containing the thread whose state you want to determine.

`threadToGet` The thread ID of the thread whose state you want to determine.

`threadState` A pointer to a thread state variable in which the function places the state of the specified thread.

DESCRIPTION

The `GetThreadStateGivenTaskRef` function returns the state of the specified thread. You can use `GetThreadStateGivenTaskRef` at times when you aren't guaranteed that your application is the current context, such as during execution of an interrupt routine. In such cases you must identify the thread task reference (the application context) as well as the thread ID.

You obtain the thread task reference for your application with the `GetCurrentTaskRef` function.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$060F</code>

RESULT CODES

<code>noErr</code>	0	State of the specified thread was returned
<code>threadNotFoundErr</code>	-618	No thread with the specified thread ID and thread task reference
<code>threadProtocolErr</code>	-619	Specified thread task reference is invalid

SEE ALSO

To determine the thread task reference (application context) for your application, use the `GetThreadCurrentTaskRef` function described on (page 73).

To change the state of a thread from stopped to ready when your application is not the current process, use the `SetThreadReadyGivenTaskRef` function described next.

The `ThreadTaskRef` data type is described on (page 46).

SetThreadReadyGivenTaskRef

You can use the `SetThreadReadyGivenTaskRef` function to change the state of a thread from stopped to ready when your application is not the current process.

```
pascal OSErr SetThreadReadyGivenTaskRef(ThreadTaskRef threadTRef,
                                         ThreadID threadToSet);
```

`threadTRef` The thread task reference of the application containing the thread whose state you want to change.

`threadToSet` The thread ID of the thread whose state you want to change.

DESCRIPTION

The `SetThreadReadyGivenTaskRef` function changes the state of a thread from stopped to ready to execute. In other words, when you mark a thread as ready to run with this function, the Thread Manager does not put it immediately into the scheduling queue but does so the next time it reschedules threads.

You can use `SetThreadStateGivenTaskRef` at times when you aren't guaranteed that your application is the current context, such as during execution of an interrupt routine. In such cases you must identify the thread task reference (the application context) as well as the thread ID.

You obtain the thread task reference for your application with the `GetThreadCurrentTaskRef` function.

IMPORTANT

The `SetThreadReadyGivenTaskRef` function allows you to do one thing only—change a thread from stopped to ready to execute. You cannot change the state of an executing thread to ready or stopped, nor can you change the state of a ready thread to executing or stopped with this call. ▲

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0410</code>

RESULT CODES

<code>noErr</code>	0	Specified thread was marked as ready
<code>threadNotFoundErr</code>	-618	No thread with the specified thread ID and thread task reference
<code>threadProtocolErr</code>	-619	Caller attempted to mark a thread ready that is not in the stopped state, or the specified thread task reference is invalid

SEE ALSO

To obtain the thread task reference of your application, use the `GetThreadCurrentTaskRef` function described on (page 73).

Thread Manager

To determine the state of a thread when your application is not the current process, use the `GetThreadStateGivenTaskRef` function described on (page 74).

See “Using Threads With I/O” beginning on page 1-36 for an example of using the `SetThreadReadyGivenTaskRef` function to wake up a thread from a completion routine.

Installing Custom Scheduling, Switching, Terminating, and Debugging Functions

This section describes functions that enable you to install custom functions that are called whenever a thread is scheduled or terminates or when the context switches. There is also a function for installing debugging functions that the Thread Manager calls whenever it creates, schedules, or disposes of a thread.

SetThreadScheduler

You can use the `SetThreadScheduler` function to install a custom scheduling function (custom scheduler).

```
pascal OSErr SetThreadScheduler
              (ThreadSchedulerProcPtr threadScheduler);
```

`threadScheduler`

A pointer to a custom scheduler. Specify `nil` if you want to remove an installed custom scheduler and use the default Thread Manager scheduling mechanism.

DESCRIPTION

The `SetThreadScheduler` function installs a custom scheduler that runs in conjunction with the default Thread Manager scheduling mechanism. The Thread Manager uses a scheduler information structure ((page 49)) to pass the custom scheduler the ID of the current thread and the ID of the thread that the Thread Manager has scheduled to run next.

A custom scheduler should return to the Thread Manager the ID of the thread that it determines to schedule. If it does not determine a particular thread to schedule, it should return the constant `kNoThreadID` and the Thread Manager default scheduling mechanism schedules the next thread.

Thread Manager

If you already have a custom scheduler installed when you call `SetThreadScheduler`, it replaces the old one with a new one. If you want to remove your custom scheduler and return to using the default Thread Manager scheduling mechanism, call `SetThreadScheduler` and specify a value of `nil` for the parameter.

IMPORTANT

The `SetThreadScheduler` function automatically disables scheduling to avoid any reentrancy problems with the custom scheduling function. Therefore, in your custom scheduling function, you should make no `yield` calls or other calls that would cause scheduling to occur. ▲

SPECIAL CONSIDERATIONS

Do not pass a routine descriptor as the `threadScheduler` parameter to the `SetThreadScheduler` function. As with all Thread Manager functions that pass a procedure pointer, you must pass the address of the routine, not the address of a routine descriptor. You are required to use routine descriptors when you write code in the PowerPC instruction set that passes a routine's address to code that might be in the 680x0 instruction set. However, the threads in your application must all be in the same instruction set—680x0 or PowerPC. Therefore, the routine identified by the `threadScheduler` parameter is by definition in the same instruction set as the `SetThreadScheduler` function and a routine descriptor is not required.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0209</code>

RESULT CODES

<code>noErr</code>	0	Specified scheduler was installed
--------------------	---	-----------------------------------

SEE ALSO

For more information on the custom scheduling function, see the `MyThreadScheduler` function on (page 86) and “Custom Scheduler” (page 13).

For more information on how the Thread Manager schedules threads, see “Scheduling” beginning on page 1-11.

SetThreadSwitcher

You can use the `SetThreadSwitcher` function to install a custom context-switching function for any thread.

```
pascal OSErr SetThreadSwitcher (ThreadID thread,
                               ThreadSwitchProcPtr threadSwitcher,
                               void *switchProcParam, Boolean inOrOut);
```

`thread` The thread ID of the thread to associate with a context-switching function.

`threadSwitcher` A pointer to the context-switching function.

`switchProcParam` A pointer to a thread-specific parameter that you pass to the context-switching function.

`inOrOut` A Boolean value that indicates whether the Thread Manager calls the context-switching function when the specified thread switches in (`True`) or when it is switched out by another thread (`False`).

DESCRIPTION

The `SetThreadSwitcher` function installs a custom context-switching function that is associated with a specified thread. The custom switching function allows you to save context information in addition to the default context information that the Thread Manager automatically saves when it switches contexts. The default context information consists of the CPU registers, the FPU registers (if any), and the location of the thread’s context.

You must actually define two context-switching functions, one for leaving a thread and another for entering a thread. When leaving a thread, you call the outer context-switching function to save additional context information. When reentering a thread, you call the inner context-switching function to restore the

Thread Manager

extra information that was saved on exit. Use the `inOrOut` parameter of the `SetThreadSwitcher` function to specify which type of context-switching function is being installed.

You can pass a different `switchProcParam` parameter to each thread, which allows you to write a single, application-wide custom switching function and then pass any thread-specific information when the Thread Manager calls the switching function for that thread.

IMPORTANT

The `SetThreadSwitcher` function automatically disables scheduling to avoid any reentrancy problems with the custom switching function. Therefore, in the custom switching function, you should make no yield calls or other calls that would cause scheduling to occur. ▲

SPECIAL CONSIDERATIONS

Do not pass a routine descriptor as the `threadSwitcher` parameter to the `SetThreadSwitcher` function. As with all Thread Manager functions that pass a procedure pointer, you must pass the address of the routine, not the address of a routine descriptor. You are required to use routine descriptors when you write code in the PowerPC instruction set that passes a routine's address to code that might be in the 680x0 instruction set. However, the threads in your application must all be in the same instruction set—680x0 or PowerPC. Therefore, the routine identified by the `threadSwitcher` parameter is by definition in the same instruction set as the `SetThreadSwitcher` function and a routine descriptor is not required.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$070A</code>

RESULT CODES

<code>noErr</code>	0	Specified thread switcher was installed
<code>threadNotFoundErr</code>	-618	No thread with the specified thread ID

SEE ALSO

For more information on the custom context-switching function, see the `MyThreadSwitch` function on (page 87) and “Custom Context-Switching Function” (page 16).

For information about the default context that the Thread Manager saves, see “Default Saved Thread Context” (page 15).

SetThreadTerminator

You can use the `SetThreadTerminator` function to install a custom thread-termination function for any thread.

```
pascal OSErr SetThreadTerminator (ThreadID thread,
                                ThreadTerminationProcPtr threadTerminator,
                                void *terminationProcParam);
```

`thread` The thread ID of the thread to associate with the thread-termination function.

`threadTerminator` A pointer to the thread-termination function.

`terminationProcParam` A pointer to a thread-specific parameter that you pass to the thread-termination function.

DESCRIPTION

The Thread Manager calls the custom termination function whenever the specified thread completes execution of its code or when you manually dispose of the thread with the `DisposeThread` function.

You can pass a different `terminationProcParam` parameter to each thread, which allows you to write a single, application-wide custom thread-termination function and then pass any thread-specific information when the Thread Manager calls the termination function for that thread.

SPECIAL CONSIDERATIONS

Do not pass a routine descriptor as the `threadTerminator` parameter to the `SetThreadTerminator` function. As with all Thread Manager functions that pass a procedure pointer, you must pass the address of the routine, not the address of a routine descriptor. You are required to use routine descriptors when you write code in the PowerPC instruction set that passes a routine's address to code that might be in the 680x0 instruction set. However, the threads in your application must all be in the same instruction set—680x0 or PowerPC. Therefore, the routine identified by the `threadTerminator` parameter is by definition in the same instruction set as the `SetThreadTerminator` function and a routine descriptor is not required.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$0611</code>

RESULT CODES

<code>noErr</code>	0	Specified thread terminator was installed
<code>threadNotFoundErr</code>	-618	No thread with the specified thread ID

SEE ALSO

To manually dispose of a thread, use the `DisposeThread` function described on (page 59).

For more information on the custom thread-termination function, see the `MyThreadTermination` function on (page 88).

For more information on what the Thread Manager does when a thread terminates, see “Creating and Disposing of Threads” (page 18).

SetDebuggerNotificationProcs

You can use the `SetDebuggerNotificationProcs` function to install functions that notify the debugger when a thread is created, disposed of, or scheduled. You generally use this function only during development of an application.

```
pascal OSErr SetDebuggerNotificationProcs
    (DebuggerNewThreadProcPtr notifyNewThread,
     DebuggerDisposeThreadProcPtr notifyDisposeThread,
     DebuggerThreadSchedulerProcPtr
     notifyThreadScheduler);
```

`notifyNewThread`

A pointer to the callback function that notifies the debugger when a thread is created.

`notifyDisposeThread`

A pointer to the callback function that notifies the debugger when a thread is disposed of. This function is called whether you manually dispose of a thread with the `DisposeThread` function or if a thread disposes of itself automatically when it returns from its highest level of code.

`notifyThreadScheduler`

A pointer to the callback function that notifies the debugger when a thread is scheduled.

DESCRIPTION

The `SetDebuggerNotificationProcs` function provides debugging support in a threaded application by letting you know when any thread is created, disposed of, or scheduled. The `SetDebuggerNotificationProcs` function installs three separate callback functions that return the thread ID of a newly created thread, the thread ID of a newly disposed of thread, and the thread ID of a newly scheduled thread.

Note

The `SetDebuggerNotificationProcs` function always installs all three of the debugging functions. You cannot set only one or two of these functions, nor can you chain them together. These restrictions ensure that the function that calls `SetDebuggerNotificationProcs` owns all three of the debugging functions. If you want to prevent one or two of these debugging functions from being called, you can do so by setting them to `nil`. ♦

The Thread Manager calls the disposing-notification callback function whether you manually dispose of a thread with the `DisposeThread` function or if a thread disposes of itself automatically when it returns from its highest level of code.

To guarantee that the debugger is getting an accurate view of scheduling, the Thread Manager doesn't call the scheduling-notification callback function until both the generic Thread Manager scheduler and any custom thread scheduler have decided on a thread to schedule.

SPECIAL CONSIDERATIONS

Do not pass a routine descriptor as any of the parameters to the `SetDebuggerNotificationProcs` function. As with all Thread Manager functions that pass a procedure pointer, you must pass the address of the routine, not the address of a routine descriptor. You are required to use routine descriptors when you write code in the PowerPC instruction set that passes a routine's address to code that might be in the 680x0 instruction set. However, the threads in your application must all be in the same instruction set—680x0 or PowerPC. Therefore, the routines identified by the parameters in this function are by definition in the same instruction set as the `SetDebuggerNotificationProcs` function and a routine descriptor is not required.

ASSEMBLY-LANGUAGE INFORMATION

Trap macro	Selector
<code>_ThreadDispatch</code>	<code>\$060D</code>

RESULT CODES

<code>noErr</code>	0	Debugger procedures were installed
--------------------	---	------------------------------------

SEE ALSO

To create or allocate a new thread, use the `NewThread` function described on (page 56).

To dispose of a thread, use the `DisposeThread` function described on (page 59).

To schedule a thread, you can use a yield function such as `YieldToAnyThread` ((page 64)) or `YieldToThread` ((page 65)) or a function to change the state of a thread, such as `SetThreadState` ((page 67)).

Application-Defined Routines

This section describes the function that you must provide as the entry point for any thread that your application creates and it describes all the custom functions that you can install, such as custom scheduling or context-switching functions.

MyThreadEntry

You must provide a `MyThreadEntry` function as the entry point to any thread that you create in your application.

```
typedef pascal void* MyThreadEntry(void *threadParam);
```

```
threadParam
```

A pointer to a `void` data structure passed to this function by the `NewThread` function.

DESCRIPTION

The `MyThreadEntry` function is the entry point to a new thread. When you create or allocate a new thread with the `NewThread` function, you pass the name of this entry function. You also pass a parameter that the Thread Manager passes on to the `MyThreadEntry` function. You can use this parameter to pass thread-specific information to the newly created or allocated thread. For example, you could pass something like A5 information or the address of a window to update. Or you use this parameter to specify local storage for a thread that other threads could access.

Thread Manager

When the code in a thread finishes executing, the Thread Manager automatically calls the `DisposeThread` function to dispose of the thread. The `MyThreadEntry` function passes its function result to `DisposeThread`. The `DisposeThread` function passes this result back to the `NewThread` function that called `MyThreadEntry` to begin with.

This mechanism allows you to spawn a thread that does some work and then continue with your original thread. When the spawned thread is finished doing its work—for example a calculation—it returns the result to the original thread.

SEE ALSO

See “Passing Input and Output Parameters to a New Thread” beginning on page 1-33 for an example of passing input and output parameters between one thread and the thread entry function in a newly created thread.

MyThreadScheduler

You may provide a custom scheduling function, `MyThreadScheduler`, to supplement the Thread Manager default scheduling mechanism.

```
typedef pascal ThreadID MyThreadScheduler(SchedulerInfoRecPtr
                                         schedulerInfo);
```

`schedulerInfo`

A pointer to the scheduler information record that the Thread Manager uses to pass information to `MyThreadScheduler`.

DESCRIPTION

You use the `SetThreadScheduler` function to install the `MyThreadScheduler` custom scheduling function (custom scheduler). The `MyThreadScheduler` function does not supplant the Thread Manager scheduling mechanism but rather works in conjunction with it.

Whenever scheduling occurs, the Thread Manager passes a scheduler information structure to `MyThreadScheduler`. Among other information, the scheduler information structure contains the thread ID of the current thread and the thread ID of the thread that the application has scheduled to run next.

Thread Manager

The `MyThreadScheduler` function returns to the Thread Manager the thread ID of the thread that it has chosen to schedule and the Thread Manager does the actual scheduling. If `MyThreadScheduler` decides not to schedule a thread, it returns the constant `kNoThreadID` and the Thread Manager default scheduling mechanism schedules the next thread.

IMPORTANT

When the `SetThreadScheduler` function installs the custom scheduler, it automatically disables scheduling to avoid any reentrancy problems. Therefore, in the custom scheduler, you should make no yield calls or other calls that would cause scheduling to occur.

SEE ALSO

See “The Scheduler Information Structure” (page 49) for more information on how the Thread Manager passes information to `MyThreadScheduler`.

For more information on how the Thread Manager schedules threads to run, see “Scheduling” beginning on page 1-11.

MyThreadSwitch

You may provide a custom switching function, `MyThreadSwitch`, to add to the thread context information that the Thread Manager saves and restores.

```
typedef pascal void MyThreadSwitch(ThreadID threadBeingSwitched,
                                   void *switchProcParam);
```

`threadBeingSwitched`

The thread ID of the thread whose context is being switched.

`switchProcParam`

A pointer to a void that the `SetThreadSwitcher` function passes to `MyThreadSwitch`.

DESCRIPTION

You use the `SetThreadSwitcher` function to install the `MyThreadSwitch` custom context-switching function. The custom switching function allows you to save and restore context information in addition to the default context information that the Thread Manager automatically saves and restores when it switches contexts. You must actually define two context-switching functions, one for leaving a thread and another for entering a thread. When leaving a thread, you call the outer context-switching function to save additional context information. When reentering a thread, you call the inner context-switching function to restore the extra information that was saved on exit.

The default context information consists of the CPU registers, the FPU registers (if any), and the location of the thread's context.

IMPORTANT

When the `SetThreadSwitcher` function installs the custom switching function, it automatically disables scheduling to avoid any reentrancy problems. Therefore, in the custom switching function, you should make no yield calls or other calls that would cause scheduling to occur. ▲

SEE ALSO

For more information on the thread context that the Thread Manager automatically saves, see "Default Saved Thread Context" beginning on page 1-15.

For more information about using a custom context-switching function, see "Custom Context-Switching Function" (page 16).

MyThreadTermination

You may provide a custom termination function, `MyThreadTermination`, to do additional cleanup when the code in a thread finishes executing.

```
typedef pascal void MyThreadTermination(ThreadID threadTerminated,  
                                       void *terminationProcParam);
```

Thread Manager

`threadTerminated`

The thread ID of the thread being disposed of.

`terminationProcParam`

A pointer to a void data structure that the `SetThreadTerminator` function passes to `MyThreadTermination`.

DESCRIPTION

You use the `SetThreadTerminator` function to install the `MyThreadTermination` custom termination function. The custom termination function allows you to do additional cleanup when the code in a thread finishes executing or when you call the `DisposeThread` function to manually dispose of a thread.

SEE ALSO

To dispose of a thread, use the `DisposeThread` function described on (page 59).

For more information on what the Thread Manager does when a thread terminates, see “Creating and Disposing of Threads” (page 18).

MyDebuggerNewThread

You may provide a debugging callback function, `MyDebuggerNewThread`, that the Thread Manager calls whenever it creates a new thread.

```
typedef pascal void MyDebuggerNewThread(ThreadID threadCreated);
```

`threadCreated`

The thread ID of the thread being created.

DESCRIPTION

The `MyDebuggerNewThread` function is one of three debugging functions that you can install with the `SetDebuggerNotificationProcs` function. The Thread Manager calls `MyDebuggerNewThread` whenever an application creates or allocates a new thread with the `NewThread` function. The Thread Manager does not call `MyDebuggerNewThread` when an application creates a thread pool with the `CreateThreadPool` function.

SEE ALSO

To create a new thread, use the `NewThread` function described on (page 56).

MyDebuggerDisposeThread

You may provide a debugging callback function, `MyDebuggerDisposeThread`, that the Thread Manager calls whenever it disposes of a thread.

```
typedef pascal void MyDebuggerDisposeThread(ThreadID threadDeleted);
```

```
threadDeleted
```

The thread ID of the thread being disposed of.

DESCRIPTION

The `MyDebuggerDisposeThread` function is one of three debugging functions that you can install with the `SetDebuggerNotificationProcs` function. The Thread Manager calls `MyDebuggerDisposeThread` whenever an application disposes of a thread. The thread manager calls this debugging function whether you manually call `DisposeThread` to dispose of a thread or if a thread finishes executing its code and the Thread Manager automatically disposes of it.

SEE ALSO

To dispose of a thread, use the `DisposeThread` function described on (page 59).

MyDebuggerThreadScheduler

You may provide a debugging callback function, `MyDebuggerThreadScheduler`, that the Thread Manager calls whenever a thread is scheduled.

```
typedef pascal ThreadID MyDebuggerThreadScheduler
    (SchedulerInfoRecPtr schedulerInfo);
```

Thread Manager

`schedulerInfo`

A pointer to a scheduler information structure that the `SetDebuggerNotificationProcs` passes to the `MyDebuggerThreadScheduler` function. Among other information, the scheduler information structure contains the ID of the current thread and the ID of the thread that the Thread Manager has scheduled to run next.

DESCRIPTION

The `MyDebuggerThreadScheduler` function is one of three debugging functions that you can install with the `SetDebuggerNotificationProcs` function. The Thread Manager calls `MyDebuggerThreadScheduler` whenever an application schedules a new thread to run. The `MyDebuggerThreadScheduler` function gets the last look at the thread being scheduled—that is, the Thread Manager calls this function after the Thread Manager default scheduling mechanism and a custom scheduler, if you have installed one, decide on the next thread to schedule.

If you wish, you can use this debugging callback function to schedule a different thread than that chosen by the Thread Manager and any custom scheduling function. The `MyDebuggerThreadScheduler` returns the thread ID of the next thread to schedule. The `MyDebuggerThreadScheduler` can specify `kNoThreadID` for the thread ID if you do not want to change the decision of the Thread Manager default scheduler or a custom scheduler.

SEE ALSO

To schedule a thread, use functions such as `YieldToAnyThread` (described on (page 64)), `YieldToThread` (described on (page 65)), and `SetThreadState` (described on (page 67)).

The scheduler information structure is described on (page 49).

Summary of the Thread Manager

C Summary

Constants and Data Types

```
enum {
#define gestaltThreadMgrAttr      'thds'      /* Gestalt selectors */
    gestaltThreadMgrPresent      = 0,        /* Thread Manager attributes */
    gestaltSpecificMatchSupport  = 1,        /* Thread Manager is present */
    match                         /* Thread Manager supports exact
                                   creation option */
    gestaltThreadsLibraryPresent = 2        /* Threads library (native version)
                                           has been loaded */
};

/* Thread states */
typedef unsigned short           ThreadState;
#define kReadyThreadState      ((ThreadState) 0)    /* thread is ready to
run */
#define kStoppedThreadState    ((ThreadState) 1)    /* thread is stopped
and
not ready to run */
#define kRunningThreadState    ((ThreadState) 2)    /* thread is running */

/* Thread environment characteristics */
typedef void*                   ThreadTaskRef;

/* Thread characteristics */
typedef unsigned long           ThreadStyle;
#define kCooperativeThread     (1<<0)             /* cooperative thread
*/

/* Thread identifiers */
typedef unsigned long           ThreadID;
#define kNoThreadID           ((ThreadID) 0)      /* no specific thread
*/
```

Thread Manager

```

*/
#define kCurrentThreadID          ((ThreadID) 1)          /* current thread */
#define kApplicationThreadID      ((ThreadID) 2)          /* main thread */

/* Options when creating a thread */
typedef unsigned long             ThreadOptions;
#define kNewSuspend                (1<<0)              /* begin a new thread as
stopped */
#define kUsePremadeThread          (1<<1)              /* use a premade thread */
#define kCreateIfNeeded            (1<<2)              /* create a new thread if one
with
required size doesn't exist

*/
#define kFPUNotNeeded              (1<<3)              /* don't save FPU context */
#define kExactMatchThread          (1<<4)              /* use a thread only if it
exactly
matches size request */

/* Information supplied to the custom scheduler */
struct SchedulerInfoRec {
    unsigned long                 InfoRecSize;          /* size of the structure */
    ThreadID                      CurrentThreadID;     /* current thread */
    ThreadID                      SuggestedThreadID;   /* suggested thread to run next
*/
    ThreadID                      InterruptedCoopThreadID; /* previously a preempted
thread; now use kNoThreadID

*/
};
typedef struct SchedulerInfoRec SchedulerInfoRec;
typedef SchedulerInfoRec *SchedulerInfoRecPtr;

```

Thread Manager functions

Creating and Getting Information About Thread Pools

```

pascal OSErr CreateThreadPool(ThreadStyle threadStyle, short numToCreate,
                             Size stackSize);

```

```

pascal OSErr GetFreeThreadCount(ThreadStyle threadStyle, short *freeCount);

```

Thread Manager

```

pascal OSErr GetSpecificFreeThreadCount(ThreadStyle threadStyle,
                                         Size stackSize, short *freeCount );

pascal OSErr GetDefaultThreadStackSize(ThreadStyle threadStyle,
                                         Size *stackSize);

```

Creating and Disposing of Threads

```

pascal OSErr NewThread(ThreadStyle threadStyle,
                      ThreadEntryProcPtr threadEntry,
                      void *threadParam,
                      Size stackSize,
                      ThreadOptions options,
                      void **threadResult,
                      ThreadID *threadMade);

pascal OSErr DisposeThread(ThreadID threadToDump, void *threadResult,
                           Boolean recycleThread);

```

Getting Information About Specific Threads

```

pascal OSErr ThreadCurrentStackSpace(ThreadID thread,
                                     unsigned long *freeStack);

pascal OSErr MacGetCurrentThread (ThreadID *currentThreadID);

pascal OSErr GetThreadState(ThreadID threadToGet, ThreadState *threadState);

```

Scheduling Threads

```

pascal OSErr YieldToAnyThread(void);

pascal OSErr YieldToThread(ThreadID suggestedThread);

pascal OSErr SetThreadState(ThreadID threadToSet, ThreadState newState,
                            ThreadID suggestedThread);

```

Preventing Scheduling

```

pascal OSErr ThreadBeginCritical(void);

pascal OSErr ThreadEndCritical(void);

```

```
pascal OSErr SetThreadStateEndCritical (ThreadID threadToSet,
                                       ThreadState newState,
                                       ThreadID suggestedThread);
```

Getting Information and Scheduling Threads During Interrupts

```
pascal OSErr GetThreadCurrentTaskRef (ThreadTaskRef *threadTRef);

pascal OSErr GetThreadStateGivenTaskRef (ThreadTaskRef threadTRef,
                                       ThreadID threadToGet,
                                       ThreadState *threadState);

pascal OSErr SetThreadReadyGivenTaskRef(ThreadTaskRef threadTRef,
                                       ThreadID threadToSet);
```

Installing Custom Scheduling, Switching, Terminating and Debugging Functions

```
pascal OSErr SetThreadScheduler(ThreadSchedulerProcPtr threadScheduler);

pascal OSErr SetThreadSwitcher(ThreadID thread,
                               ThreadSwitchProcPtr threadSwitcher,
                               void *switchProcParam, Boolean inOrOut);

pascal OSErr SetThreadTerminator(ThreadID thread,
                                  ThreadTerminationProcPtr threadTerminator,
                                  void *terminationProcParam);

pascal OSErr SetDebuggerNotificationProcs
    (DebuggerNewThreadProcPtr notifyNewThread,
     DebuggerDisposeThreadProcPtr notifyDisposeThread,
     DebuggerThreadSchedulerProcPtr notifyThreadScheduler);
```

Application-Defined Routines

```
typedef pascal void* MyThreadEntry(void *threadParam);

typedef pascal ThreadID MyThreadScheduler(SchedulerInfoRecPtr schedulerInfo);

typedef pascal void MyThreadSwitch(ThreadID threadBeingSwitched,
                                   void *switchProcParam);

typedef pascal void MyThreadTermination(ThreadID threadTerminated,
                                       void *terminationProcParam);
```

```

typedef pascal void MyDebuggerNewThread(ThreadID threadCreated);
typedef pascal void MyDebuggerDisposeThread(ThreadID threadDeleted);
typedef pascal ThreadID MyDebuggerThreadScheduler
                                (SchedulerInfoRecPtr
schedulerInfo);

```

Pascal Summary

Constants and Data Types

```
{ Gestalt selectors }
```

```
CONST
```

```

    gestaltThreadMgrAttr          = 'thds'; { Thread Manager attributes }
    gestaltThreadMgrPresent       = 0;     { Thread Manager is present }
    gestaltSpecificMatchSupport   = 1;     { Thread Manager supports exact
                                           match creation option }
    gestaltThreadsLibraryPresent  = 2;     { The Threads library (native
version)
                                           has been loaded }

```

```
{ Thread states }
```

```
TYPE
```

```
    ThreadState = INTEGER;
```

```
CONST
```

```

    kReadyThreadState      = 0;     { thread is ready to run }
    kStoppedThreadState    = 1;     { thread is stopped and not ready to run }
    kRunningThreadState    = 2;     { thread is running }

```

```
{ Thread environment characteristics }
```

```
TYPE
```

```
    ThreadTaskRef = Ptr;
```

CHAPTER 1

Thread Manager

```
{ Thread characteristics }
TYPE
    ThreadStyle = LONGINT;
CONST
    kCooperativeThread          = 1;          { cooperative thread }

{ Thread identifiers }
TYPE
    ThreadID = LONGINT;
CONST
    kNoThreadID                = 0;          { no specific thread }
    kCurrentThreadID           = 1;          { current thread }
    kApplicationThreadID       = 2;          { main thread }

{ Options when creating a thread }
TYPE
    ThreadOptions = LONGINT;
CONST
    kNewSuspend                = 1;          { begin a new thread as stopped }
    kUsePremadeThread           = 2;          { use a premade thread }
    kCreateIfNeeded             = 4;          { create a new thread if one with
                                             required size doesn't exist }
    kFPUNotNeeded               = 8;          { don't save FPU context }
    kExactMatchThread           = 16;         { create a thread only if it exactly
                                             matches size request }

{ Information supplied to the custom scheduler }

TYPE
SchedulerInfoRecPtr= ^SchedulerInfoRec;
SchedulerInfoRec= RECORD
    InfoRecSize:                LONGINT;      { size of the structure }
    CurrentThreadID:            ThreadID;     { current thread }
    SuggestedThreadID:          ThreadID;     { suggested thread to run next }
}
    InterruptedCoopThreadID:    ThreadID;     { previously a preempted
thread;                               now use kNoThreadID }

END;
```

Thread Manager Functions

Creating and Getting Information About Thread Pools

```

FUNCTION CreateThreadPool(threadStyle: ThreadStyle; numToCreate: INTEGER;
                        stackSize: Size):OSError;

FUNCTION GetFreeThreadCount(threadStyle: ThreadStyle;
                            VAR freeCount: INTEGER):OSError;

FUNCTION GetSpecificFreeThreadCount(threadStyle: ThreadStyle;
                                    stackSize: Size;
                                    VAR freeCount: INTEGER):OSError;

FUNCTION GetDefaultThreadStackSize(threadStyle: ThreadStyle;
                                    VAR stackSize: Size):OSError;

```

Creating and Disposing of Threads

```

FUNCTION NewThread(threadStyle: ThreadStyle;
                  threadEntry: ThreadEntryProcPtr;
                  threadParam: LONGINT;
                  stackSize: Size;
                  options: ThreadOptions;
                  threadResult: LongIntPtr;
                  VAR threadMade: ThreadID):OSError;

FUNCTION DisposeThread(threadToDump: ThreadID; threadResult: LONGINT;
                      recycleThread: BOOLEAN):OSError;

```

Getting Information About Specific Threads

```

FUNCTION ThreadCurrentStackSpace(thread: ThreadID;
                                VAR freeStack: LONGINT):OSError;

FUNCTION MacGetCurrentThread (VAR currentThreadID: ThreadID):OSError;

FUNCTION GetThreadState(threadToGet: ThreadID;
                        VAR threadState: ThreadState):OSError;

```

Scheduling Threads

```

FUNCTION YieldToAnyThread:OSErr;

FUNCTION YieldToThread(suggestedThread: ThreadID):OSErr;

FUNCTION SetThreadState(threadToSet: ThreadID; newState: ThreadState;
                        suggestedThread: ThreadID):OSErr;

```

Preventing Scheduling

```

FUNCTION ThreadBeginCritical:OSErr;

FUNCTION ThreadEndCritical:OSErr;

FUNCTION SetThreadStateEndCritical(threadToSet: ThreadID;
                                    newState: ThreadState;
                                    suggestedThread: ThreadID):OSErr;

```

Getting Information and Scheduling Threads During Interrupts

```

FUNCTION GetThreadCurrentTaskRef ( VAR threadTRef: ThreadTaskRef ):OSErr;

FUNCTION GetThreadStateGivenTaskRef(threadTRef: ThreadTaskRef;
                                     threadToGet: ThreadID;
                                     VAR threadState: ThreadState ):OSErr;

FUNCTION SetThreadReadyGivenTaskRef(threadTRef: ThreadTaskRef;
                                     threadToSet: ThreadID ):OSErr;

```

Installing Custom Scheduling, Switching, Terminating, and Debugging Functions

```

FUNCTION SetThreadScheduler(threadScheduler: ThreadSchedulerProcPtr):OSErr;

FUNCTION SetThreadSwitcher(thread: ThreadID;
                           threadSwitcher: ThreadSwitchProcPtr;
                           switchProcParam: LONGINT; inOrOut: BOOLEAN):OSErr;

FUNCTION SetThreadTerminator(thread: ThreadID;
                             threadTerminator: ThreadTerminationProcPtr;
                             terminationProcParam: LONGINT):OSErr;

```

Thread Manager

```

FUNCTION SetDebuggerNotificationProcs
    (notifyNewThread: DebuggerNewThreadProcPtr;
     notifyDisposeThread: DebuggerDisposeThreadProcPtr;
     notifyThreadScheduler: DebuggerThreadSchedulerProcPtr ):OSErr;

```

Application-Defined Functions

```

FUNCTION MyThreadEntry(threadParam: LONGINT): LONGINT; }

FUNCTION MyThreadScheduler(schedulerInfo: SchedulerInfoRec): ThreadID;

PROCEDURE MyThreadSwitch(threadBeingSwitched: ThreadID;
                          switchProcParam: LONGINT);

PROCEDURE MyThreadTermination(threadTerminated: ThreadID;
                               terminationProcParam: LONGINT);

PROCEDURE MyDebuggerNewThread(threadCreated: ThreadID);

PROCEDURE MyDebuggerDisposeThread(threadDeleted: ThreadID);

FUNCTION MyDebuggerThreadScheduler
    (schedulerInfo: SchedulerInfoRec): ThreadID;

```

Assembly Language Information

Trap Macros Requiring Routine Selectors

_ThreadDispatch

Selector	Routine
0x000B	ThreadBeginCritical
0x000C	ThreadEndCritical
0x0205	YieldToThread
0x0206	MacGetCurrentThread
0x0209	SetThreadScheduler
0x020E	GetThreadCurrentTaskRef
0x303C	YieldToAnyThread

Thread Manager

Selector	Routine
0x0402	GetFreeThreadCount
0x0407	GetThreadState
0x0410	SetThreadReadyGivenTaskRef
0x0413	GetDefaultThreadStackSize
0x0414	ThreadCurrentStackSize
0x0501	CreateThreadPool
0x0504	DisposeThread
0x0508	SetThreadState
0x0512	SetThreadStateEndCritical
0x060D	SetDebuggerNotificationProcs
0x060F	GetThreadStateGivenTaskRef
0x0611	SetThreadTerminator
0x0615	GetSpecificFreeThreadCount
0x070A	SetThreadSwitcher
0x0E03	NewThread

Result Codes

Thread Manager functions can return the following errors. Functions may also return standard Macintosh result codes such as `noErr` (0, no error) and `memFullErr` (memory full).

<code>threadTooManyReqsErr</code>	-617	No matching thread structures available
<code>threadNotFoundErr</code>	-618	No thread available with the specified Thread ID
<code>threadProtocolErr</code>	-619	Attempted an invalid operation, such as changing the state of a thread that is in a critical section of code

Glossary

application thread See **main thread**.

critical section of code A section of code in which scheduling is disabled and the current thread cannot yield control to another thread.

concurrency Having multiple, simultaneous points of execution within an application.

context see **thread context**.

cooperative thread A thread that uses the Operating System and Toolbox and hence cannot be arbitrarily interrupted. A cooperative thread explicitly indicates when it is giving up CPU time. The Thread Manager supports only cooperative threads. Compare **preemptive thread**.

lightweight task A synonym for *thread*.

main thread The entry point to the application. It is a cooperative thread and typically handles all event processing. It is also called the *application thread*. For applications that do not explicitly use threads, the Thread Manager defines a single main thread.

multithreaded application See **threaded application**.

nonthreaded application An application that has a single point of execution.

preemptive thread A thread that does not use the Operating System and Toolbox and hence can be interrupted or gain control of

the CPU at almost any time. The Thread Manager does not support preemptive threads. Compare **cooperative thread**.

ready thread A thread that is available for scheduling.

single-threaded application See **nonthreaded application**.

stopped thread A thread that is unavailable for scheduling.

thread The smallest amount of processor context state necessary to encapsulate a computation; a thread consists of a register set, a program counter, and a stack. Threads enable concurrency within an application context. A thread is sometimes called a *lightweight task*.

thread context Information the Thread Manager maintains about a thread. It includes a register set, program counter, and stack.

thread pool A pool of threads that you create for later allocation.

Thread Manager The part of the Macintosh Operating System that provides multiple points of execution within an application context by managing the scheduling, execution, and termination of threads.

threaded application An application that has multiple points of execution.

yield Give up control of the CPU to another thread.

Index

Numerals

680x0 Macintosh applications
 default thread context 15

A, B

asynchronous I/O
 using threads with 36–43

C

completion routines
 for Thread Manager routines 43
context. *See* thread context
CreateThreadPool function 50–52
 example of use 23
 specifying stack size with 30
critical code sections
 defined 12
 ending 28–29, 70–72
 starting 28–29, 69
custom context-switching function
 defined 87–88
 installing 79–81
custom debugging functions
 defined 89–91
 installing 83–85
custom scheduling function
 about 13
 defined 86–87
 installing 77–79
custom termination function
 defined 88–89
 installing 81–82

D

debugger disposing function 90
debugger notification functions
 defined 89–90
 installing 83–85
default scheduling mechanism 13
default stack size 54–55
dialog boxes
 yielding control from 31–33
DisposeThread function 59–60

E

entry point function. *See* thread entry function
events
 using main thread to handle 33

F

floating-point registers. *See* FPU registers.
FPU registers
 saving 15

G, H

Gestalt Manager
 using to determine attributes of Thread
 Manager 19, 44
GetCurrentThread function *See*
 MacGetCurrentThread function
GetDefaultThreadStackSize function 54–55
GetFreeThreadCount function 52–53
GetSpecificFreeThreadCount function 53–54

GetThreadCurrentTaskRef function 73–74
 GetThreadState function 63
 GetThreadStateGivenTaskRef function 74–75
 using in I/O completion routine 43

I – L

interrupt routines
 referring to threads from 46, 72–77
 I/O
 using threads with 36–43

M

MacGetCurrentThread function 42, 62
 main thread
 calling MaxApplZone in 20–22, 51
 keeping ready or running 43
 using to handle events 33
 MaxApplZone function
 calling in threaded applications 20–22, 51
 MyDebuggerDisposeThread function 90
 MyDebuggerNewThread function 89–90
 MyDebuggerThreadScheduler function 90–91
 MyThreadEntry function 85–86
 MyThreadScheduler function 86–87
 MyThreadSwitch function 87–88
 MyThreadTermination function 88–89

N, O

NewThread function 56–59
 allocating threads from pool with 24–27
 specifying stack size with 30

P, Q

PowerPC applications
 default thread context 15
 Process Manager
 relationship to Thread Manager 29

R

routine descriptors
 warning about 58, 78, 80, 82, 84

S

SchedulerInfoRec 49
 scheduler information structure
 defined 49
 using with custom scheduler 14
 scheduling
 See also custom scheduling function
 changing thread state 67–68, 75–77
 turning off 12, 28–29, 69
 turning on 70, 71–72
 yielding 64–65
 yielding to particular thread 65–66
 scheduling threads 11–18
 SetDebuggerNotificationProcs function 83–85
 SetThreadReadyGivenTaskRef function 75–77
 SetThreadScheduler function 13, 77–79
 SetThreadStateEndCritical function 71–72
 SetThreadState function 43, 67–68
 SetThreadStateGivenTaskRef function
 using in I/O completion routine 43
 SetThreadSwitcher function 16, 79–81
 SetThreadTerminator function 81–82
 680x0 Macintosh applications
 default thread context 15

stacks, for threads
 amount available, determining 61–62
 default size 18, 30, 54–55
 introduced 17
 overflowing 30
 size of 51
 size of, specifying 18, 29
 specifying 29

T – X

ThreadBeginCritical function 28, 69
 thread context
 default saved 15
 defined 9
 saving custom information 16, 79–81
 ThreadCurrentStackSpace function 61–62
 ThreadEndCritical function 28, 70
 thread entry function
 defined 85–86
 how to specify 26
 using 33–36
 thread ID
 defined 47
 obtaining 62
 ThreadID data type 47
 Thread Manager 7–101
 application-defined routines for 85–91
 data structures for 44–49
 determining attributes of 19, 44
 functions in 49–85
 relationship to Process Manager 29
 thread options 48
 ThreadOptions data type 48
 thread pools
 creating 18, 20–23, 50–52
 threads
 See also main thread
 allocating 18, 20–27, 56–59
 creating 18, 56–59
 creating a pool of 18, 20–23, 50–52
 defined 9
 disposing of 18, 31, 59–60

 passing data to 33–36
 recycling 59
 returning data from 18, 33–36, 60
 scheduling 11–18
 types of 46
 uses of 8
 Threads Package
 difference from Thread Manager 18
 thread stacks. *See* stacks, for threads
 thread state
 changing 67–68, 71–72
 changing from interrupt-level code 75–77
 defined 45
 obtaining 63
 obtaining from interrupt-level code 74–75
 ThreadState data type 45
 ThreadStyle data type 46
 ThreadTaskRef data type 46
 thread task reference
 defined 46
 obtaining 73–74
 thread type 46

Y, Z

YieldToAnyThread function 32, 64–65
 YieldToThread function 65–66

