T E C H N O T E :
Working With Apple's Multiprocessing API

By Chris Cooksey

ccooksey@daystar.com
DayStar Digital

This Technote describes the basic steps required to use the Apple Multiprocessing API and attempts to clarify the things that can and cannot be done from tasks created using that API.

This Technote is directed primarily at developers working with, or preparing to work with the Multiprocessing API. Although the examples given are aimed at application writers it contains information useful for system level engineers also.

About the Apple Multiprocessing API

The Apple Multiprocessing API provides a set of calls that allow an application to create separate threads of execution called **tasks**. Tasks are preemptively scheduled on the available processors in the system, even if there is only one. Tasks have the same view of system memory as the application.

An application creates tasks that are used to perform work on the application's behalf. Since the work can be performed simultaneously by the tasks in a multiprocessor system, the work throughput of the application can be increased.

For example, an image-processing application can create tasks that transform arbitrary blocks of image data. When the user chooses to transform an image, the application splits the data into smaller pieces and then asks each of the tasks to transform one of the pieces. This process is repeated until all of the pieces have been transformed. Since the pieces are processed simultaneously, the time required to transform the entire image is greatly reduced.

Another example is development environments. An environment could use tasks to compile different files. In this way multiple files could be compiled simultaneously, reducing the time it takes to build a project.

Using Multiprocessing

Using The Library

To use multiprocessing, the Multiprocessing API Library should be added and weak-linked into your project. To weak link to the library in Metrowerks, use the pull down menu next to the library name in the project window and select 'weak link'. This will ensure that if the library is not present your application will still be launched.

Near the beginning of the application you should test for the presence of the Multiprocessing API Library. In the file in which you want to do this include the file MP.h. To test for the presence of the library, call `MPLibraryIsLoaded`. `MPLibraryIsLoaded` is a macro that tests for the presence of one of the library entry points. If `MPLibraryIsLoaded` returns true, then the library is present and its services are available. Otherwise, your application should run without invoking any multiprocessing services.

Deciding How Many Tasks To Create

Use `MPProcessors` to count the number of processors in the system. If there is only one, you may wish to proceed as though the multiprocessing services are not available. However, you can still create preemptive tasks in a single processor environment if you want to.

The count returned by `MPProcessors` is usually used as an indication of how many tasks to create. There are a number of factors that should go into this decision.

First and foremost, your application should strive to keep all the processors busy. The simplest way to do this is to create at least as many tasks as there are processors. The application then splits the work to be done into that many pieces and asks each task to work on a piece. The application waits on the tasks to finish

before proceeding. If your tasks take more than one tenth of a second to finish their pieces, then the application should test for events while it is waiting for the tasks to complete.

An alternative and frequently-adopted technique is to create one less task than there are processors. In this case, the application still breaks the work into a number of pieces equivalent to the number of processors. Each task is asked to work on a piece and then the main application works directly on the remaining piece. When the application is finished, it then waits on the tasks to finish. If the work was split reasonably evenly, then the tasks should be finished or just about finished when the application starts to wait. This approach is very popular because it allows for easy generalization of the single processor case: the number of tasks created is zero and the application naturally ends up doing all the work. It is also a convenient path to follow when the Multiprocessing API Library is not available for some reason. Note that since the application is now involved in performing work, it is important to limit the time required to perform the work to less than one tenth of a second, so that events can be checked frequently enough and the application's responsiveness can be preserved.

The above cases assume that the tasks are all doing pretty much the same thing. If your application creates tasks that do very different types of work, then how many tasks you create will more likely depend on what different kinds of work need to get done. The important thing to bear in mind is that the processors should be kept as busy as possible. To do this, make sure that at least as many tasks as there are processors will have something to do during the times when you wish to maximize throughput. Don't forget that four processor systems are not at all uncommon and more powerful systems should be expected in the future.

Communicating With Tasks

Communication between the application and tasks occurs in two basic ways:shared memory, and synchronization methods. Since all memory is shared, anything the application writes into memory is available to a task and vice-versa. However, before a task tries to access memory that has been prepared by the application, the task MUST synchronize with the application using one of the three methods available in the Multiprocessing API Library. This is extremely important. The PowerPC architecture allows for writes to memory to be deferred. This is a resource management feature that helps the PowerPC achieve its tremendous speed. In order for another processor to see the correct values in memory, certain hardware dependent instructions need to be executed. When a task uses a synchronization method these instructions are executed, thus ensuring that the processors involved have a consistent view of memory from that point on. It is also important to use synchronization methods so that when one of the communicants is not yet ready to synchronize for some reason, the other one can yield the processor it is on. This makes the processor immediately available to some other task that may be able to make more productive use of it.

The synchronization methods available in the Multiprocessing API Library are queues, semaphores and critical regions.

Queues are first-in-first-out queues of 96-bit messages. Inserting and extracting elements is an atomic operation -many tasks can try to extract the next message from a given queue but only one will successfully obtain it.

Semaphores represent a single 32-bit value that can be atomically incremented up to a predetermined maximum and atomically decremented to a minimum of zero.

Critical regions prevent sections of code that they encompass from being executed by more than one task or the application at once.

Before creating tasks it is usually a good idea to create the means by which to synchronize with them. Queues and semaphores are the two most common methods used. People starting out with multiprocessing generally use queues since they provide the most flexibility and are relatively easy to understand. Semaphores are quicker and less memory intensive but do not offer the same degree of flexibility. Queues and semaphores are usually created in pairs:one by which to signal a request, the other by which to signal results. A classic mistake often made by beginners (the author is speaking from experience here) is to create only one synchronization object and to try to use it for both purposes. This does not work. After a request is posted, the application will at some point start waiting for results. If it waits at the same place the request was posted, the request itself may appear to be the result. Since the application clears the request in the mistaken belief that it was a result, no work at ALL gets done. This is why it is important to use two distinct entities for two-way communication.

Creating Tasks

Creating a task is done by calling `MPCreateTask`. The first parameter to this call is a pointer to a function that will become the running task. The task function must have the following prototype:

```
OSStatus fTask( void *parameter );
```

The task receives a 32-bit parameter when it starts up and should return a 32-bit result when it finishes. The parameter received at startup is specified as the second parameter to the `MPCreateTask` call. It is through this parameter that all the initial information that the task will need is communicated to it. It can be anything at all, a message queue id, a pointer to a C++ object, etc. It is not uncommon for it to be a pointer to a task specific block of memory through which the application will communicate various information throughout the task's lifetime.

Everything up to this point, from calling `MPLibraryIsLoaded` to calling `MPCreateTask` can be done when your application starts up. Leaving tasks running for the lifetime of an application is usually a more efficient strategy than creating and destroying them as needed. If you find yourself creating a lot of different tasks that do different types of things you should consider creating a task that can call many different types of functions through a selector based scheme, or through a variable function pointer.

The following code illustrates how an application could establish its multiprocessing capabilities soon after it begins running. It uses the technique of creating one less task than there are processors.

```
typedef struct {
    long firstThing;
    long totalThings;
    }sWorkParams, *sWorkParamsPtr;

typedef struct {
    MPTaskID taskID;
    MPQueueID requestQueue;
    MPQueueID resultQueue;
    sWorkParams params;
    }sTaskData, *sTaskDataPtr;

long gNumProcessors;
sTaskDataPtr gTaskData;
MPQueueID gNotificationQueue;

void fStartMP( void ) {

    OSErr theErr;
    long i;

    theErr = noErr;

    /* Assume single processor mode */
    gNumProcessors = 1;

    /* Initialize remaining globals */
    gTaskData = NULL;
    gNotificationQueue = NULL;

    /* If the library is present create the tasks (no tasks on a      */
    /* single CPU system)                                             */
    if( MPLibraryIsLoaded() ) {
        gNumProcessors = MPProcessors();
        gTaskData = (sTaskDataPtr)NewPtrClear( (gNumProcessors - 1) *
            sizeof( sTaskData ) );
        theErr = MemError();
        if( theErr == noErr  )
            theErr = MPCreateQueue( &gNotificationQueue );
        for( i = 0; i < gNumProcessors - 1 && theErr == noErr; i++ ) {
            if( theErr == noErr )
                theErr = MPCreateQueue( &gTaskData[i].requestQueue );
            if( theErr == noErr )
                theErr = MPCreateQueue( &gTaskData[i].resultQueue );
            if( theErr == noErr )
                theErr = MPCreateTask( fTask, &gTaskData[i],
                    kMPUseDefaultStackSize, gNotificationQueue,
                    NULL, NULL, kMPNormalTaskOptions,
                    &gTaskData[i].taskID );
            }
        }

    /* If something went wrong, just go back to single processor      */
    /* mode                                                           */
    if( theErr != noErr ) {
        fStopMP();
        gNumProcessors = 1;
        }
    }
```

The structure sWorkParams defines the parameters that will be passed into the function called by the task. The content of this block is specific to the type of work being performed. Among other things, the parameters should define the specific data that the function is to process.

The structure sTaskData defines the block of data that the application will use to communicate a variety of different information to a task. The two main things being communicated are the queue IDs and work function parameters.

The global gNumProcessors stores a count of the number of processors found in the system. This variable is set to one if the Multiprocessing API Library is not loaded or if task or queue creation fails for any reason. The rest of the application code is fashioned in such a way that if gNumProcessors is one then the application will do all the work itself and never make any Multiprocessing API calls.

The global gTaskData points to a dynamically allocated array of sTaskData blocks. There is one entry for each task to be created.

The global gNotificationQueue is used to receive notification messages from terminating tasks. All the tasks share one notification queue in this example.

A number of tasks equal to the number of processors minus one are created. Each task has its own pair of message queues by which the application can communicate with it. The IDs of the queues are stored in the gTaskData entry for the task. The task is then created using `MPCreateTask`. The first parameter is a pointer to the function that will become the running task. In this example all the tasks share the same function:`fTask`. If a function can be correctly executed by multiple processors at once it is called 'reentrant'. Note that 'interrupt-safe' does not necessarily imply 'reentrant'. Interrupts generally are not interruptable in the Mac environment and engineers sometimes take advantage of this. However, in an MP environment you must anticipate that there could be tasks simultaneously executing at any point at any time within your task code.

The second parameter is a pointer to the task's gTaskData entry. The task will be able to extract the IDs of the request and result queues it should use from this block. Note that two queues per task is often unnecessary. In many cases it is possible to use two queues total. All requests are posted to one queue and all results are returned on another queue. This works when it is irrelevant which task processes which request, as is often the case. Note, however, that the parameters for each task must be either completely contained within the message, or preestablished for every task prior to submitting the first request.

The third parameter is the desired stack size for the task. Each task has its own stack. If you are going to be creating more than a handful of tasks, you should consider limiting the size of the stack each one will receive. The default size is 64K, which can seriously impact the amount of memory available to the Multiprocessing API Library if large numbers of tasks are going to be created. If you do specify the stack size, be sure to allocate at least as much space as your task's deepest call chain will require.

The fourth parameter is for an optional notification queue. This queue is very important during task termination sequences. In fact, it really isn't optional unless you tightly coordinate task termination with the task itself. If you terminate a task without warning, you will definitely need a notification queue. The reason for this will be given later.

The fifth and sixth parameters are returned on the notification queue when the task is terminated.

The seventh parameter is for modifying the nature of task creation. There are no options available at this time.

The eighth parameter is filled in by `MPCreateTask`. It will be the ID of the newly created task. For convenience it is stored in the task's gTaskData entry. Note that tasks rarely have a need to know what ID they are.

If anything goes wrong during task creation, `fStopMP` is called. It will delete and terminate everything that has already been created. The variable gNumProcessors is then reset to 1, which will cause the application to proceed as though there were only one processor available. The function `fStopMP` is described later.

The following is an example task. The first thing it does is establish a pointer to its gTaskData entry which was specified in `MPCreateTask`. It obtains the request and result queue from this block. The task then waits for a request on the request queue. It uses the message it receives to select a function to call. The parameters for the function are extracted from the task's gTaskData entry which will have been set up by the application prior to posting the request message. The application must be very careful to preserve the validity of all the parameters passed to the task until the task sends its result message. It would, for example, be catastrophic for an application to move or delete memory being written to by a currently running task.

```
#define kMyRequestOne         1
#define kMyRequestTwo         2

#define kMyResultException    -1

OSStatus fTask( void *parameter ) {

    OSErr theErr;
    sTaskDataPtr p;
    Boolean finished;
    long message;

    theErr = noErr;

    /* Get a pointer to this task's unique data */
    p = (sTaskDataPtr)parameter ;

    /* Process each request handed to the task and return a result */
    finished = false;
    while( !finished ) {
        theErr = MPWaitOnQueue( p->requestQueue, (void **)&message,
            NULL, NULL, kDurationForever );
        if( theErr == noErr ) {
            /* Pick a function to call and pass in the parameters.    */
```

```
            /* The parameters should be set up prior to sending the   */
            /* message just received. Note that we could also just    */
            /* pass in a pointer to the desired function instead of   */
            /* using a selector.                                      */
            switch( message ) {
                case kMyRequestOne:
                    theErr = fMyTaskFunctionOne( &p->params );
                    break;
                case kMyRequestTwo:
                    theErr = fMyTaskFunctionTwo( &p->params );
                    break;
                default:
                    finished = true;
                    theErr = kMyResultException;
                }
            MPNotifyQueue( p->resultQueue, (void *)theErr, NULL, NULL );
            }
        else
            finished = true;
        }

    /* Task is finished now */
    return( theErr );
    }
```

Using Tasks To Perform Work

When your application needs to perform some work, it should make sure everything the tasks are going to need is in memory. For each task, the application will establish the parameters of the work that it wants the task to perform and then it will signal the task through either a queue or a semaphore to begin performing that work. The specific work that the task should perform can be completely defined within a message, or possibly in a block of memory reserved for that task as described above. Both methods are in common use. Some applications also pass in a pointer to the function that the task should call to perform the work. That way one task can perform many different types of chores.

Once the task has been signaled, the application can help out with the work, or it could return to its event loop and just check in on the tasks from time to time using kDurationImmediate waits.

When kDurationImmediate is specified to either `MPWaitOnQueue`, `MPWaitOnSemaphore` or `MPEnterCriticalRegion` the function always returns immediately. If the return value is kMPTimeoutErr then whatever was being waited on could not be obtained. That is, no message was available, the semaphore was zero, or the critical region was being executed by another processor.

Therefore, if the application is checking for task results in its event loop, use kDurationImmediate waits and check the return value. If it is noErr, a result was present and obtained by the call. If it is kMPTimeoutErr, then the tasks have generated no new results since the last time the application checked. Don't forget that other kinds of errors could be returned also.

As described above, when a task finishes handling the request, it should post a result to let the application know that the work has been performed.

An example of an application using tasks to perform work follows. In this case the application is going to perform part of the work also. Note that events are not being handled, so it is assumed that `fMyTaskFunctionOne` will not take more than a tenth of a second to perform the work.

```
OSErr fDoMP( long realFirstThing, long realTotalThings ) {

    long i;
    OSErr theErr;
    long thingsPerTask;
    long message;
    sWorkParams appData;

    theErr = noErr;

    thingsPerTask = realTotalThings / gNumProcessors;

    /* Start each task working on a unique piece of the total data */
    for( i = 0; i < gNumProcessors - 1; i++ ) {
        gTaskData[i].params.firstThing = realFirstThing + thingsPerTask * i;
        gTaskData[i].params.totalThings = thingsPerTask;
        message = kMyRequestOne;
        MPNotifyQueue( gTaskData[i].requestQueue, (void *)message,
            NULL, NULL );
        }

    /* Let the application do whatever is left over. Note that if     */
    /* gNumProcessors is one, then the application will do everything */
    /* and the Multiprocessing API Library will not be called.        */
    appData.firstThing = realFirstThing + thingsPerTask * i;
    appData.totalThings = realTotalThings - thingsPerTask * i;
    fMyTaskFunctionOne( &appData );

    /* Now wait for the tasks to finish */
    for( i = 0; i < gNumProcessors - 1; i++ )
        MPWaitOnQueue( gTaskData[i].resultQueue, (void **)&message,
            NULL, NULL, kDurationForever );

    return( theErr );
    }
```

This particular approach is used in a lot of real world applications. It is best suited to applications that transform large blocks of data. Data is split into even pieces for the tasks, they are started, and the remaining potentially uneven piece is processed by the application. Once the application has processed its piece, it then waits for the tasks to finish. A common mistake, even for experienced engineers, is to assume that the data will be perfectly divisible by the number of processors.

Applications that work on large uneven pieces, such as a development environment trying to compile multiple files simultaneously, need to approach the problem differently. The application should sit in its event loop and as each task finishes the work it was assigned, new work, if any, should be assigned to the task.

Terminating Tasks

When your application finishes it should call `MPTerminateTask`. Note that `MPTerminateTask` does not kill a running task immediately. It only flags it to be

killed at some convenient time in the future. Therefore it is very important not to delete any of the resources the task was using until the task is truly terminated. To be sure that this is the case you should wait on a notification queue that was provided to the `MPCreateTask` call. Every time `MPTerminateTask` is called you should immediately wait on the notification queue for a message. Once you receive one you can be sure that the task is no longer running and that it is safe to delete any shared resources.

The function `fStopMP` below is an example of what could be done when the application is about to terminate. The one important thing to note about `fStopMP` is that as soon as a task has been terminated using `MPTerminateTask` the function halts until a message on the notification queue arrives.

```
void fStopMP( void ) {

    long i;

    if( gTaskData != NULL ) {
        for( i = 0; i < gNumProcessors - 1; i++ ) {
            if( gTaskData[i].taskID != NULL ) {
                MPTerminateTask( gTaskData[i].taskID, noErr );
                MPWaitOnQueue( gNotificationQueue, NULL,
                    NULL, NULL, kDurationForever  );
                }
            if( gTaskData[i].requestQueue != NULL )
                MPDeleteQueue( gTaskData[i].requestQueue );
            if( gTaskData[i].resultQueue != NULL )
                MPDeleteQueue( gTaskData[i].resultQueue );
            }

        if( gNotificationQueue != NULL ) {
            MPDeleteQueue( gNotificationQueue );
            gNotificationQueue = NULL;
            }

        DisposePtr( (Ptr)gTaskData  );
        gTaskData = NULL;
        }
    }
```

Multiprocessing Do's and Don'ts

Do

If you get a message at startup telling you that the MPLibrary could not load because it was out of memory, then you should open your copy of Metronub, which resides in the extension folder, using ResEdit or Resourcer. Change the `'sysz'` resource to read 2500000. If you still get the problem, keep increasing the `'sysz'` value by 1 MB at a time until you don't.

Tasks should call functions that perform faceless processing. Calculation intensive code is really the only type of code that should be considered for MP tasks. This rule will be substantially relaxed under Mac OS 8, but even so, throughput will really only be improved by using MP for calculation intensive code. Under Mac OS 8 responsiveness will be the main thing improved by multitasking other types

of code.

The work performed by a task between request and result signals should be 'substantial'. It can take several hundred machine cycles to send or receive a signal via one of the synchronization methods. If your task only takes a few cycles to complete the work that is requested, your application's performance is going to be dramatically worse with multiprocessing. Tasks should try to consume at least a million cycles per request. That's 5 milliseconds on a 200MHz processor and 20 times faster than necessary to maintain the tenth of a second response time quoted throughout this document.

If your task needs to allocate memory you will have to either allocate the memory prior to signaling the task, or use the function `MPAllocate`. The function `MPAllocate` will return a block of memory allocated from the application's heap. Unfortunately `MPAllocate` is very slow. It suspends the task, asks the application to fulfill the request, and then resumes the task. If it is used a lot it will significantly reduce a task's throughput. The best solution to obtaining memory is to preallocate all of it. If you cannot, then use `MPAllocate` to allocate one large page of memory at a time and draw smaller blocks from the page as needed.

Don't

Don't attempt to call 68K code. There is no emulator on the secondary processors and they will fault if they attempt to perform a mixed mode switch.

Do not call the Toolbox. The Toolbox still contains large amounts of 68K code but even worse it is largely non-reentrant. For example if one task is calling `NewPtr` and another task also decides to call `NewPtr`, both will be manipulating the same global heap structure at the exact same time and they will almost certainly corrupt it. In Mac OS 8, a number of Toolbox routines will be callable from an MP task. These routines will be conditionalized by the flag FOR_SYSTEM8_PREEMPTIVE in the MacOS 8 interface files.

Do not call into unknown code. If you provide a means by which a third party can specify a callback then do not attempt to call that function from a task. There is no telling what the callback is going to do. This rule will never be relaxed. Unless you specifically require that the callback be reentrant, then there is always going to be the possibility that it is not.

Avoid globals. The main cause of non-reentrancy is the manipulation of globals. Tasks that manipulate globals, global state, or buffers pointed to by globals must use synchronization techniques to prevent other tasks from attempting to do so at the same time. Globals that are read only are fine.

Do not call any MP API routines at interrupt time. The Multiprocessing API Library is not, strictly speaking, reentrant. While you can call any Multiprocessing API routine from a multiprocessing task any time, you may not call them from a deferred task, a time manager task or any other system interrupt handler. Workarounds exist but they are inefficient and generally discouraged. Contact

Apple DTS or DayStar for more information.

Summary

After reading this Technote, you should be comfortable with the basic steps involved in producing a multiprocessing-aware application. In short, you need to make sure the Multiprocessing API Library is available, you need to count the number of processors, you need to create the means by which to synchronize with tasks, and you need to create a sufficient number of tasks that will keep all the processors busy. Unique information can be communicated to a task when it is created that will allow a task to coordinate with the application when work needs to be performed. When your application quits, it should delete the synchronization objects and terminate the tasks.

You should be familiar with the types of things a task can do and you should know what a task cannot do.

Further References

Multiprocessing SDK.

Acknowledgments

Thanks to Phil Koch, Irvan Krantzler, David Sowell, Jason Wallace, and Dan Wilk.

Change History

Originally written in July, 1996 by Chris Cooksey