A Posix Thread Implementation for Mac OS

Nigel Perry and Robert Pointon

Institute of Information Sciences and Technology
Massey University
Palmerston North, New Zealand.
{N.Perry,R.Pointon}@massey.ac.nz

Abstract

Our research work involving the design and implementation of concurrent programming languages suffered a blow when the projects Unix multiprocessor died. The cost to repair this machine was not only rather expensive but, given the speed of advance in computer hardware and the age of the machine (4 years), ill-advised. We therefore sought a suitable low-end (up to about NZ dollar 10,000) replacement. The surprising answer was a Mac OS machine. Mac OS is not the obvious first choice for a multiprocessor, there are few models and the software is not well developed. The hardware has future promise in supporting concurrent programming with upcoming operating systems: Mac OS X (maybe), BeOS, and Linux; but the time scale for any of these would have been too long. However, if we could use the current Mac OS it offered a good price/performance, future promise, and would support other non-project work well.

In the Unix world a standard has emerged for thread-based programming, Posix Threads (pthreads), and our existing software was based on this. Under Mac OS there are currently two distinct choices; the thread manager, and the multiprocessing API. The thread manager is a standard part of the system, but unfortunately does not use additional processors and hence provide true concurrency. Contrawise the multiprocessing API is not so well developed, has major limitations, but does use multiple processors. The limitations are quite severe with tasks unable to perform most OS calls, such as memory management and I/O, making them primarily suitable for compute-only use.

This paper describes a C-based pthread compatible system that has been designed to run over the multiprocessing API. It describes the problems that needed to be tackled and what it was not possible to emulate from the Posix standard. Also covered is how the related problems of thread based I/O and memory management are tackled.

The pthread compatible system has enabled us to port Unix-based code with minimal changes, though this result will vary with application. Such code should in future be easily ported to MkLinux, and also Mac OS X if it retains the Mach microkernel base of Rhapsody.

1 INTRODUCTION

The motivation for implementing posix threads under Mac OS, was the desire to port a multi-threaded C program from a UNIX machine. In choosing the new platform the following points needed to be considered:

- The price
- Multiprocessing and thread support

  The following platforms and operating systems were evaluated:

- Unix (Sun/Dec/HP) - The traditional Unix multiprocessor machines were beyond our budget.
- PC - Multiprocessing PCs are available but fairly expensive. However: converting Unix/pthread code to Win 95/NT, finding a suitable development environment, and other problems such as the lack of word alignment for functions; combine to make switching difficult. Using Linux might be an alternative and solve most of these problems.
- Mac - Multiprocessing machines are available, but not many. We considered three OS options:
  - BeOS - The BeOS thread stack size was limited to about 256K and we found the system very unstable.
  - MkLinux - Multiprocessing isn't yet properly supported under Mac versions of Linux.
  - Mac OS - Multiprocessing supported using the MP API, but not very well developed and designed primarily for compute-bound processing. Psuedo-concurrency is well supported through the thread library, but this does not use multiple processors.

As can be seen, the choice of a low-cost system to support multiprocessing is not straightforward. The Mac offered the best priced/performance platform but the choice of OS was more difficult. The obvious choices of BeOS or MkLinux where either not stable or developed enough to meet our timeframe. However an evaluation suggested that a Posix-style interface to the Mac OS MP API could be developed, so we ended up with what, to us at least, was a surprising choice: Mac OS for multiprocessing development.

2 IMPLEMENTING THE POSIX THREAD MODEL UNDER MAC OS

Under Mac OS, multiprocessing of tasks is provided by the MP API [MPAPI 95, 96]. The differences between the Mac OS MP task and Posix thread models are significant [1]. To provide a transparent emulation of the Posix thread environment, two issues need to be addressed:

- Basic Thread Environment

  The creation and control of Posix threads, provision of Posix mutexes, etc. need to be emulated over the Mac OS MP API MP Safe Runtime Library.

  The C runtime library must be adapted to operate correctly for concurrently executing threads. This includes facilities such as I/O and dynamic memory management.

- SMP vs ASMP

  Pthreads usually run on a UNIX machine using symmetrical multiprocessing (SMP) where each CPU has full capabilities, including I/O. Threads each have the same capabilities and access operating system services.

  Mac OS supports asymmetrical multiprocessing (ASMP). Only the main processor runs Mac OS while other processors act as slaves and run a very simple kernel with limited capabilities. In particular I/O and dynamic memory operations are limited or unavailable on the slaves. Tasks created using the MP API may run on any processor. However tasks executing on slave processors are intended for computational jobs only and are very limited in what toolbox services they can access.

  To use ASMP as SMP the illusion of calling toolbox services must be provided. This can be achieved by transparently suspending the task on the slave processor, arranging for a task on the primary processor to call the Mac OS service required, and then resuming the task on the slave.

3 PROVIDING A TRANSPARENT POSIX THREAD EMULATION

To provide a transparent emulation of Posix threads each Posix feature has to be mapped onto one or more underlying Mac OS MP task facility.

- Thread Specific Data

  Each pthread has its own stack as well as some global storage which is local to that thread only, this is known as thread specific data. This mechanism is widely used by pthread programs.

  Mac OS tasks have no notion of task specific data at this time. However a task can determine its own task ID.

  To address this issue our system maintains thread specific data in an internal structure accessed by task ID. Pthread routines use the current tasks ID to

[1]The Mac OS MP API refers to a process executing as part of another process, sometimes called a lightweight process, as a task. Posix refers to such a process as a thread. We will use the term task when referring to a Mac OS MP process, and thread or pthread when referring to our emulated Posix thread. There is a one-to-one correspondence between a task and a thread, the different terms simply described a different view of the process.

access the thread specific data. The overhead in maintaining the thread specific data in this way is not large.

- Mutexes

  Posix mutexes provide a mechanism for exclusive access to a portion of code.

  Under Mac OS MP critical regions are provided which perform the same function, the only difference is that critical regions allow recursive locking while mutexes generally don't.

  The implementation of mutexes is therefore straightforward. We have chosen to omit the extra check needed to prevent recursive locking, as it makes no difference to a correctly written (in the sense of Posix semantics) program and would impose an extra cost of using mutexes. In addition some extensions of Posix support recursive locking.

- Conditions

  Posix conditions combined with mutexes allow threads to signal each other, and wait for some signal. This forms the basis for implementing higher-level communication mechanisms, for example asynchronous queues.

  Mac OS MP also provides some higher-level communication mechanisms such as notification queues, along with semaphores.

  There is no direct mapping between these concepts so Posix conditions need to be emulated by a combination of Mac OS MP mechanisms. First an MP semaphore is associated with every task which tasks can wait on, thus providing a mechanism to selectively suspend and resume tasks. Conditions use this semaphore to control the tasks and a queue of waiting tasks is maintained per condition. As concurrent tasks can access a condition at the same time the queue is protected with an MP critical region to ensure its consistency.

- Thread Attributes

  Thread attributes can be set at the creation of a pthread and control, for example, the threads scheduling policy or maximum stack size.

  The Mac OS MP provides little control of task attributes, with only the size of the stack size being definable.

  Under the current Mac OS MP there is nothing that can be done to provide the flexibility of Posix in this area. Fortunately the lack of these features does not effect the correctness of a Posix program, only its performance, and the Posix standard does not require them all to be provided.

3.1 Overall Finer Details - Keeping The Emulation Transparent

Under Posix there is no semantic or operational difference between the initial thread of a process and any threads subsequently created. To address this and to also support the illusion of SMP our system creates an MP task to evaluate the main() routine of a C program. The initial (non-MP task) process is then used as a service thread to handle remote procedure calls from tasks requesting toolbox services, and to handle program termination and task cleanup.

Under Mac OS MP a message queue is usually associated with each task so that notification of termination and other messages can be sent back to the parent. Our system only creates one queue shared by all tasks. This queue is monitored by the initial process which then cleans up resources for tasks as they terminate.

Posix identifies threads using a value of an implementation specific type `pthread_t`. Our emulated pthread consists of an MP task, a semaphore to control task suspension, and some other internal information. This information is kept in a single thread description object, the address of which is defined to be `pthread_t`, thus the information required by our emulation is transparently handled by Posix thread programs and they do not need to know they are using an emulation. The tasks are also maintained in a linked list to support task cleanup at program termination.

The emulation of Posix conditions requires a queue of threads. As a thread can only be on one condition queue at a time, a field is included in the thread description object to maintain the queue links. This reduces memory management overhead for conditions and allows threads to be quickly added to or removed from queues. The emulation also requires a single critical region per condition. This and a pointer to the first thread in the queue are stored in a condition description object and the address of this used as the Posix `pthread_cont_t` type, thus maintaining the transparency of the emulation.

3.2 Unimplemented Posix Thread Routines

Currently several issues remain unaddressed because either:

The features were not required for our porting; or They would either overly complicate the existing code; or They were simply too tricky!

The main Posix routines not supported are:

- `pthread_cancel()`
- `pthread_kill()`
- `pthread_cleanup_push()`
- `pthread_cleanup_pop()`

Cleanup handlers and issues such as asynchronous cancellation and signals have been tactfully avoided.

- `pthread_cond_timedwait()`

Unfortunately the MP API doesn't support any times other than forever and immediate.

4 AN MP-SAFE C LIBRARY

No MP-Safe C library is provided as part of Mac OS MP. Furthermore an MP task cannot in general perform I/O or access most other toolbox services, unless it happens to be executing on the primary processor.

However, the MP API provides a general remote procedure call operation so a task running on a slave may execute code on the primary processor. This can be used to create the illusion of symmetrical multiprocessing. There is a potential cost in this as the toolbox calls are sequentialised to run on the primary processor.

The impact of this will vary depending on the application; if threads rarely perform toolbox calls at the same time the impact is minimal, but two I/O intensive threads would operate sequentially. However, as Mac OS evolves we expect there will be less need for these calls, and current evaluation shows an acceptable cost.

4.1 Metrowerks CodeWarrior

We choose Metrowerks CodeWarrior as our development environment for a number of key reasons:

- MP Debugging Support

  Metrowerks provides a special debugger nub which allows source-level debugging of MP tasks. This facility is a huge benefit as not even Macsbug calls can be made by MP tasks.

- Library Code Written For Portability

  The CodeWarrior system is available for many different operating systems, including Mac OS, Win32, BeOS and Solaris. To make their own task easier Metrowerks have written a library with all platform-specific code separated from general platform-independant code. This enables the library to be retargeted for a different platform with the minimum of effort.

- Multiprocessing Ready Library Code

  To allow safe execution in a concurrent environment all critical parts of the library code are protected by multiprocessing synchronisation macros. For the normal Mac OS environment these macros do nothing. However by suitable definition they can use whatever multiprocessing system is available.

4.2 Converting The C Library

The Metrowerks library defines four routines; __init_critical_regions(), __kill_critical_regions(), __begin_critical_region() and __end_critical_region(); to guard access to MP-unsafe sections of the platform-independant code. These four routines are trivially implemented using Mac OS MP critical region routines.

The platform-specific parts of the library are isolated into about a dozen code modules labelled with the platform name; for example for Mac OS these files are called ¡module¿.mac.c. These platform-specific modules generally access operating system services.

Under Mac OS MP a task cannot use the operating system services directly, to do so it must use an MP remote procedure call to execute code to access the service on the primary processor. To provide a transparent MP-Safe C library all that is needed is to replace every platform-specific routine with an intermediary routine which uses a remote procedure call to execute the original library code.

This process is complicated by the restriction that any remote procedure can only take one argument, so any multi-argument routine must be converted to a routine which accepts a single record argument made up of the original individual arguments. This is potentially a lot of changes to the platform-specific library code, and every time the library is revised by Metrowerks would require significant reworking.

The solution to this problem was found in C++ and its recently introduced namespaces. In C++ a namespace is similar to a module in other languages, that is a named collection of definitions which can be accessed by qualifying them by the namespace name. Multiple namespaces can be declared in the same program file.

To construct and MP-safe version of a platform-specific module we can hide the original platform-specific code within a namespace, this changing each global name to be qualified by the namespace name, and then provide a global interface routine under the original name to access the routine through the remote procedure call mechanism. A code fragment showing this scheme is included in the addendum.

4.2.1 Handling exit()

Mac OS MP requires that all tasks are terminated before a process exits, failure to do this can result in a system crash. Unfortunately a C program can at any time call exit() or abort() to terminate the process immediately. To make the C library MP-safe these calls must be intercepted and all MP tasks terminate properly. Our system handles this issue as follows:

- The service thread executing on the primary processor, in addition to handling remote procedure service requests, monitors a global flag set by a replacement exit() and abort() routines. If this flag is set all MP tasks are forcefully terminated and then the process exits back to Mac OS.
- If the exit() and abort() routines are called by an MP task they set a global flag and then just terminate the current MP task.
- If the exit() and abort() routines are called from a remote procedure call then the problem is more involved. Without going into the finer details our system first terminates the remote procedure call, using the C longjmp mechanism, and then sets the global flag and terminates the current task. This implementation also requires the replacement of the MP API remote procedure call by a custom routine.

As is apparent, handling exit() and abort() correctly is an involved process. A reasonable question is why bother to handle the final case above. It is rare for a remote procedure call to require the use of exit() and abort(), so why not just forbid this and not support it? However doing this is either as involved as supporting the routines, if an error message is to be produced and the program terminated, or if the prohibition is not enforced results in a system crash if violated - with the consequential problems and inconvenience that causes. We therefore decide to support these routines, and though complicated, there is little runtime cost.

4.2.2 Program initialisation

Program initialisation is handled by supplying a custom replacement for the Metrowerks startup code and custom MP Stationary to base projects upon. This enables a pthread program to be written in a completely transparent manner.

5 MAC HARDWARE ISSUES

The PPC processor [Motorola] uses write deferment where the CPU automatically schedules memory write operations - possibly changing the order and/or delaying them for a long time. The synchronisation routines in the MP API (semaphores, critical regions, etc.) call specific instructions which complete any deferred writes for the given processor.

Thus if a writing thread does not use a synchronisation method, and instead signals completion via a global, it is possible that the global write would complete prior to previous memory writes. A reading thread on a different processor examining memory based on the value of the global can potentially see the original values.

The problem only shows up if tasks (running on different processors) are sharing memory, and relying on methods of synchronisation other than that provided by the MP API. Provided pthread programs use Posix synchronisation routines this is not a problem. Presumably any system running on PPC processors would face a similar problem (though the BeOS documentation does not refer to it) and it is outside the control of our emulation.

6 CONCLUSION

We have described a Posix Thread emulation package with allows a large proportion of pthread programs to be ported without change from Unix to Mac OS MP.

There are some potential performance issues due to the serialisation of I/O and other operating system services, but these will only be significant if there are multiple I/O bound threads within a single program.

We have ported most of a large compiler system and associated runtime for a concurrent language to Mac OS from Unix without change using this system. We do not expect problems with pthreads to be an issue in completing this project.

REFERENCES

- [Motorola] PowerPC 604 RISC Microprocessor User's Manual, Motorola Semiconductors.
- [MPAPI 95] Multiprocessor API Specification, Apple Computer and DayStar Digital, Inc. August 1995.
- [MPAPI 96] Tech note 1071 - Working With Apple's Multiprocessing API. Chris Cooksey, Apple Computer and DayStar Digital, Inc. 10/4/96.
- [Posix] P1003.4A Posix Threads.

ADDENDUM

The following code fragment, taken from the implementation of file_io, shows how C++ namespaces are used to avoid editing original source files:

```
// file_io.mac.c

namespace _unsafe_file_io
{
#include "file_io.mac.c" // hide the MP-unsafe code
}

namespace _private_file_io // add some inter-
face routines
{ struct open_args

  { const char * name;
    __file_modes mode;
    __file_handle * handle;
  };

  void* __open_file(open_args *args)
  { return (void*)_unsafe_file_io::__open_file(
      args->name,
      args->mode,
      args->handle);
  }

}
```

```
int __open_file(
  const char * name,
  __file_modes mode,
  __file_handle * handle)
{ _private_file_io::open_args args;

  // package arguments for RPC
  args.name = name;
  args.mode = mode;
  args.handle = handle;

  return int(
    MPRPC(_private_file_io::__open_file,&args) );
}
```