Bringing the C Libraries With Us into a Multi-Threaded Future

Michael B. Jones – Carnegie Mellon University

ABSTRACT

An enormous amount of UNIX (and UNIX-like) code has been written (by a likewise enormous amount of programmers) that uses the standard C libraries. Use is made throughout much of this code of the knowledge that traditional UNIX programs have exactly one thread of control. However, increasing numbers of UNIX-like systems are beginning to provide support for programs with multiple threads of control. To the extent possible, it is highly desirable to preserve the existing C library interfaces for multi-threaded programs; this will aid both in code and programmer portability between traditional UNIX environments and new multi-threaded ones.

A number of issues must be confronted in order to produce versions of the C libraries which can be used in multi-threaded programming environments. Among these are: functions with non-reentrant interfaces, functions which maintain state between invocations, use of macros in the library interfaces, interactions with signals, compatibility with single-threaded library data structures, performance issues, and of course, errno. Despite these and other problems, experience has shown that reasonable solutions are available. This paper presents both a detailed explanation of the problems inherent in producing multi-thread-safe C libraries and the different solutions which are available. Finally, the solutions to these problems adopted by a number of research and industry groups are presented.

Introduction

Existing C Libraries

An enormous amount of UNIX (and UNIX-like) code has been written that uses the standard C libraries. Such libraries include libc, libm, libtermcap, libcurses, etc. Likewise, there is a wide base of programmer experience with these existing interfaces.

Use is made throughout much of this code of the knowledge that traditional UNIX programs have exactly one thread of control. However, increasing numbers of UNIX-like systems are beginning to provide support for programs with multiple threads of control.

Multi-Threaded UNIX-like Systems Appearing

Systems by both research and industry groups such as Apollo, Mach, Encore, Convex, Chorus, OSF/1, future System V releases, and future SunOS releases are all supporting multi-threaded address spaces. The P1003.4a IEEE POSIX standards group has begun balloting a proposed threads interface standard.

Several advantages of multiple threads over the traditional single-threaded programming model have driven the industry towards supporting multi-threaded programs. Chief among them are:

- Threads provide a natural paradigm for expressing inherent program parallelism; they provide a synchronous alternative to asynchronous interruption or polling.
- Threads allow for the efficient utilization of multiple processors, when available.

Clear Need for C Libraries Usable in Multi-Threaded Programs

To the extent possible, it is highly desirable to preserve the existing C library interfaces for multi-threaded programs. This will maximize code portability and reusability between traditional UNIX environments and new multi-threaded ones. Possibly even more importantly, this will also maximize programmer portability between the old and new environments by allowing programmers to continue to use already familiar programming facilities.

General Requirements on Multi-Threaded Libraries

Operating in a multi-threaded programming environment imposes additional requirements upon libraries which are not present for single-threaded environments. The main new requirement is that concurrent accesses to data structures shared between threads must be synchronized in such a way that the data structure invariants are maintained. Also, some processor and bus cache architectures require explicit cache synchronization operations to be performed in order to provide a consistent view of shared data structures.

A wide range of different synchronization mechanisms may be used for protecting shared data structure accesses. Among them are are mutual exclusion locks (mutexes) and condition variables, spin locks, binary semaphores, counted semaphores, reader-writer locks, and Ada-like rendezvous mechanisms. While the different mechanisms provide somewhat different semantics, each can actually be implemented in terms of the others, giving them equivalent semantic power. However, efficiency considerations make some choices better than others.

Probably the most important criteria for choosing among synchronization mechanisms is that uncontested accesses to shared data should introduce almost no additional overhead. Any of mutexes and condition variables, spin locks, semaphores, and reader-writer locks can meet this requirement; each might be appropriate in different application contexts. Each rendezvous, however, typically requires at least one thread context switch, introducing extra overhead even in the best case. The mutexes and condition variables shall be used in illustrations for the remainder of this paper, recognizing that other synchronization primitives could also have been used.

Locking Approaches

Two general approaches can be taken towards performing synchronization (which henceforward shall also be called simply ''locking'') for functions which require restrictions upon concurrent execution for correct operation.

- **Internal Locking**: Concurrent execution restrictions are enforced by placing appropriate lock calls within the function itself. No additional preconditions are placed upon callers.
- **External Locking**: Concurrent execution restrictions are enforced by requiring that callers ensure improper concurrent calls are not made. Typically, this means that callers must perform explicit synchronization operations around calls.

A simple example serves to illustrate the difference between the two approaches. The function *put*():

```
void put(char val)
{
    *put_ptr++ = val;
}
```

which would be called as:

```
put(val1);
```

cannot be safely concurrently executed since the pointer fetch, value store, pointer increment, and pointer store will not be atomically executed.

With internal locking, the function would be modified to execute correctly in the presence of concurrent calls as follows:

```
void put(char val)
{
    mutex_lock(&put_mutex);
    *put_ptr++ = val;
    mutex_unlock(&put_mutex);
}
```

Since the function itself is ensuring mutual exclusion, the critical section is never executed concurrently.

With external locking, the function would be unmodified, but callers would typically have to place locks around calls, as in:

```
mutex_lock(&put_mutex);
put(val1);
mutex_unlock(&put_mutex);
```

Since the callers are ensuring mutual exclusion, the critical section is never executed concurrently.

Discussion of Internal Locking

Some of the advantages of internal locking are:

- **Simplicity of Use**: Such functions' invariants are maintained by the functions' implementations, rather than by their callers, placing no extra semantic burden the callers.
- **Modularity of Use**: Calls to functions utilizing internally locking can be written independently since independent callers need not coordinate with one other to maintain internal function invariants.
- **Existing Function Interfaces**: Existing function interfaces can often be used without modification. Since internal locking makes synchronization issues the responsibility of the implementation, callers need not be changed to address them.
- **Minimal Code Size**: Locking code for each function will typically exist only within the function, and not be potentially replicated at each call site.
- **Program Robustness**: A function's data structures cannot be corrupted by concurrent calls (provided, of course, that the internal locking needed by the function was implemented correctly). The burden of maintaining the function's invariants is on the function's implementer, rather than the function's callers.

The main disadvantage of internal locking is:

- **Extra Locking**: Internal locking potentially performs more locking than is actually needed. By definition, locking is being performed at a granularity necessary to independently protect each such function's invariants across every call. In some circumstances, however, significantly less locking could be performed by performing explicit locking around a group of otherwise unprotected calls, rather than implicitly performing locking internal to each call.

Discussion of External Locking

Some of the advantages of external locking are:

- **Flexible Locking Granularity**: Locking granularity decisions are left up to the application. Thus, less total locking may potentially be performed than that which would have been by internal locking. For instance, applications might place several calls to functions requiring external locking within a critical section protected by a single lock.
- **Existing Function Implementations**: Existing function implementations can be used without modification. Since external locking makes synchronization issues the responsibility of the callers, implementations need not be changed to address them. Of course, to the extent that one library function is a client of another which requires locking, the client function's implementation still must be changed. Thus in practice, many implementations will still have to be changed even if external locking is used.

Some of the disadvantages of external locking are:

- **Increased Code Complexity**: Clients of functions requiring external locking are required to be aware of and maintain concurrency invariants via explicit programming discipline. Explicit locking mechanisms must be introduced in order to maintain these invariants. Furthermore, all modules using such functions must use the same locking mechanisms for the locking to be effective. The result of this is that the external locking mechanism to be used for each such function must also be specified with the function in order for such functions to be usable in a modular fashion.
- **Increased Code Size**: There will typically be a set of locking operations needed around each call to a function needing locking. This makes the amount of locking code needed proportional to the number of calls made to such functions (rather than to the number of such functions called).
- **Undetected Race Conditions**: External locking potentially increases the possibility of undetected race conditions in programs which appear to (sometimes) work. If a programmer forgets to lock around calls to functions requiring external locking, the program will still sometimes work since concurrent calls may not have actually been been executed. Sometimes, however, concurrent calls will have been executed, violating the functions' preconditions, resulting in arbitrarily bad results (such as corrupted data structures or illegal memory references). Functions requiring external locking do not fail gracefully when called in subtly incorrect ways, often making debugging such code a nightmare.

Approaches Not Mutually Exclusive

While it might at first appear that internal and external locking are mutually exclusive choices, this is not actually the case. There are contexts where both might appropriately be used together, gaining the advantages (and disadvantages) of both. For instance, both approaches are used together in many stdio implementations.

Cache Synchronization Techniques

Some processor and bus cache architectures require explicit cache synchronization operations to be performed in order to provide a consistent view of shared data structures. Two different cache synchronization approaches can be taken:

1. **Explicit Flushing**: This approach requires applications to perform explicit cache synchronization operations whenever shared memory needs to be made consistent. The advantage of this approach is that the application may be able to perform the least possible number of such synchronization operations. The disadvantages are that such operations are inherently non-portable and error-prone.
2. **Flushing by Locking Primitives**: This approach requires the locking primitives to perform any cache synchronization operations necessary in to provide a consistent view of the data structures protected by the locks. This is a logical approach since acquiring a lock expresses intent to modify a shared data structure, and releasing it expresses that the modification has been completed. These are the two points where cache synchronization operations would likely be required anyway. While in theory extra synchronization might be performed, in practice it would have been needed in any event. Also, hiding the cache synchronization operations removes a source of non-portability.

Problems with Existing C Libraries

Many of the functions provided by the standard C libraries are already reentrant and thread-safe. Nonetheless, a number of problems must be overcome to make versions of the standard C libraries which are usable in multi-threaded programming environments. The following sections outline some of the problems and solutions.

Functions with Non-Reentrant Interfaces

A number of functions (e.g., *localtime*(), *opendir*(), *getpwent*(), etc.) have interfaces which are non-reentrant because they return results in statically allocated buffers or structures. Such functions cannot be safely used in multi-threaded programs in an unrestricted fashion since a call by one thread may overwrite the results of a call by another thread before they are used. A number of different solutions are possible:

1. **External Locking**: This approach requires clients to hold locks when calling such functions and using the results. For instance, a mutex could be introduced for each such static result area which clients would be required to hold. A call to *localtime*() might then look like:

```
struct tm *tp;

mutex_lock(&localtime_mutex);
tp = localtime(&clock);
/* Use results pointed to by tp */
mutex_unlock(&localtime_mutex);
/* Results at *tp potentially
   corrupted */
```

2. **Thread-Specific Data**: This approach has such functions return pointers to thread-specific data areas instead of statically allocated data areas. This would require such functions to be modified to allocate thread-specific storage. This approach has the advantage that calls remain the same and no extra client locking is needed. One disadvantage is the potential performance penalty of thread-specific data manipulation. Another disadvantage is the extra per-thread storage space required for each such result area; unless garbage collection is being used, thread-specific storage will not be freed until at least thread exit time, and possibly not until the program exits.
3. **Alternate Reentrant Versions**: This approach provides alternate reentrant versions of such functions. These versions would be passed pointers to result areas allocated by the caller. For instance, a reentrant version of *localtime*() might look like:

```
int localtime_r(struct tm *result,
                time_t *clock);
```

One unfortunate point about some C implementations should probably be noted here. Some C compilers (for instance, those which are derived from *pcc*) return structure values from functions in statically allocated storage. Thus, functions which may appear to be reentrant are actually not when compiled with some compilers. Fortunately, such functions are typically not used by the traditional C libraries.

Functions which Maintain State Between Invocations

A number of functions (e.g., *rand*(), *malloc*(), *getpwent*(), the stdio functions, etc.) maintain state between invocations. One problem which must be resolved in a multi-threaded environment is whether such state should be maintained on a per-thread basis, a per-process basis, or whether allocation of the state should be placed under application control. A number of different solutions are possible:

1 **External Locking**: This approach requires clients to hold locks when calling such functions. The locks guard manipulations of the persistent state. This presumes that the persistent state should be maintained on a per-process basis. In the case of *rand*(), for instance, this means that a single random number stream would be shared by all threads.

2 **Internal Locking**: This approach performs any locking necessary for manipulation of persistent state within the function manipulating it. This approach also presumes that the persistent state should be maintained on a per-process basis.

3 **Thread-Specific Data**: This approach maintains the persistent state in thread-specific storage areas. In the case of *rand*(), for instance, this means that a different random number stream would be used by each thread. Issues of thread-specific storage allocation, initialization, and deallocation must be addressed with this approach.

4 **Alternate Reentrant Versions**: This approach places the allocation of persistent state under program control. This is the most general solution for such functions. This can be done by providing alternate reentrant versions of such functions with explicit parameters for all state. For instance, new versions of *rand*() and *srand*() might look like:

```
#include <stdlib.h>
/* Defines rand_state_t */

int rand_r(rand_state_t *s);
int srand_r(unsigned int seed,
            rand_state_t *s);
```

This is sufficiently general for implementing per-thread, per-process, and other sharing schemes, depending upon the allocation and use of the persistent state parameter.

The stdio Functions

Locking for the stdio Functions

The I/O buffers manipulated by the stdio functions are a special case of state which persists across function invocations. In theory, any of the approaches outlined above could be taken for these functions. In practice, thread-specific buffer allocation can probably be immediately ruled out, as the consequence would be that stdio calls (e.g., *getc*(), *printf*()) from different threads would operate on different I/O streams even though they used the same ''FILE *'' parameter; *stdin*, *stdout*, and *stderr* would mean different things from different threads! Per-process buffer allocation is almost certainly sufficient.

Next, note that no new reentrant versions of the stdio functions need be introduced since they can already be made reentrant. The ''FILE *'' parameters already serve as a handle for identifying each set of persistent state, with *fopen*(), *fclose*(), etc., serving as general purpose allocators and deallocators. Also note that while the I/O buffers are allocated on a per-process basis, more flexible usage schemes are certainly possible, depending upon how the ''FILE *'' pointers are shared between threads.

This leaves either external locking or internal locking. Internal locking has the advantages that typical client code need not perform any extra locking and that the stdio data structures would not be corrupted by erroneous unlocked calls. External locking has the advantage that applications can explicitly gain and relinquish control of I/O streams, preventing unwanted interleaved reads or writes. A combination of these approaches with the advantages of both has actually been used by several implementations.

The combination locking approach taken by some implementations for the stdio functions internally locks the stdio streams while manipulating them, protecting the internal data structure integrity and allowing clients to call the stdio functions (e.g., *printf*()) without holding explicit locks. It also allows clients to explicitly lock stdio streams across stdio calls, preventing unwanted interleaved use by multiple threads. To accomplish this, the implementation uses and exports a recursive (or counted) mutex which allows a thread to recursively reacquire and release the same lock without deadlocking, only actually releasing the lock when the lock count goes to zero.

The new stdio lock functions are of the form:

```
void flockfile(FILE *stream);
void funlockfile(FILE *stream);
```

and allow usage of the form:

```
flockfile(f);
fprintf(f, "in");
fprintf(f, "divisible");
funlockfile(f);
```

sending the characters ''indivisible'' to stream *f* without any intervening characters possibly being inserted.

The putc(), getc(), putchar(), and getchar() Macros

The *putc*(), *getc*(), *putchar*(), and *getchar*() macros present special problems. If internal locking is to be used, then the macro definitions themselves must be changed to add it or call locking versions. Unlike most other changes, this requires recompilation instead of just relinking.

Also, the reason that these functions are implemented as macros rather than subroutines in the first place is performance. While locking overhead may be small relative to the work done by most functions, it will be somewhat larger relative to the tiny amount of work done by these macros. For some programs, this extra overhead may be unacceptable.

Four alternatives are possible:

1 **Fast but Unsafe**: Don't introduce locks into *putc*(), etc. All calls would then have to occur inside explicit *flockfile*() and *funlockfile*() scopes so that necessary locking would be done; otherwise the stdio data structures could be corrupted. Existing code would have to have the lock calls added to be converted.

2 **Slow but Safe**: Introduce locks into *putc*(), etc. Explicit locking would be unnecessary, but each call would incur lock overhead. Existing code would not have to be changed.

3 **Safe allowing Fast**: Introduce locks into *putc*(), etc. as in the ''slow but safe'' choice, but also provide versions of them without locks as in the ''fast but unsafe'' choice using different names. For instance, some implementations use the names *putc_unlocked*(), *getc_unlocked*(), etc. for versions without locking, and the normal names for the locking versions. Existing code can be recompiled and will be correct. The unlocked versions can be used inside explicitly locked scopes where performance is critical. An example using the unlocked versions is:

```
flockfile(stdout);
putchar_unlocked('O');
putchar_unlocked('K');
putchar_unlocked('\n');
funlockfile(stdout);
```

From a software engineering point of view, this approach is the clear winner. Existing code recompiles and works; no unexpected race conditions are introduced. Making the macros thread-safe is consistent with the treatment of the other stdio functions. Plus, the fast versions are available for contexts where the extra performance is critical.

4 **Fast allowing Safe**: This is similar to the ''safe allowing fast'' choice, except that the default is the unsafe versions, with the safe versions provided under new names. It has all the drawbacks of ''fast but unsafe'' and adds very little since the safe versions could already be constructed by using explicit lock calls around the unsafe versions. It is included in this list largely for the sake of completeness.

The errno Variable

The *errno* variable presents a problem which is a special case of functions with non-reentrant interfaces; it is a result returned in statically allocated storage from a large number of functions. The same solutions apply, namely:

1 **External Locking**: This approach requires an ''errno lock'' to be held when calling functions using errno and using the results. This would present an immense, tedious burden to programmers, serialize nearly all multiprocessor system calls, and require changing all code making system calls when converting it to be multi-threaded.

2 **Thread-Specific Data**: This approach allocates a thread-specific *errno* value for each thread. This requires modification of the *errno* variable declaration to contain an address calculation using the current thread as an implicit parameter. One such declaration might be:

```
extern int *_errno(void);
#define errno (*_errno())
```

where the *_errno*() function returns the thread-specific *errno* address for the calling thread. Fortunately, ANSI C specified that the *errno* declaration is provided by the header file ''<errno.h>'' in anticipation of such modifications. Calls to functions returning values in *errno* would not have to be changed; likewise, code fetching the value of *errno* would remain unchanged.

3 **Alternate Reentrant Versions**: This approach provides alternate reentrant versions of functions using *errno* to return results. These versions would be passed a pointer to an int for storing the *errno* value. For instance, a reentrant version of *kill*() might look like:

```
int kill_r(int pid, int sig,
           int *error);
```

This would require changing all calls to functions returning values in *errno* when converting them to be multi-threaded.

Functions with Non-Reentrant Implementations

Some existing libraries contain non-reentrant implementations of functions even though they have reentrant interfaces. One common problem area is mathematical function implementations; such functions often store temporary results in statically allocated variables, particularly when written in assembly language. Software floating point implementations are often culprits. Such implementations need to be fixed when creating multi-threaded libraries.

As pointed out earlier, some compilers generate needlessly non-reentrant code as well (e.g., returning structure values in statics). The only solutions possible in this case are either to fix the compiler or avoid using the broken compiler features entirely.

Signals in the Multi-Threaded Environment

Signals cause a whole range of potential complications for multi-threaded programs. These problems stem from several causes:

- Signals have been heavily overloaded to perform several unrelated functions (e.g., synchronous exception handling, low-bandwidth asynchronous IPC, process control).
- Signal semantics vary significantly between different implementations and even between different signals in the same implementation.
- Signals can potentially cause asynchronous interruption of a program's critical sections.

A number of different approaches to signal handling are possible in multi-threaded programs. Most make a distinction between two types of signals:

- **Synchronously Generated Signals**: These signals are attributable to a specific thread. For example, executing an illegal instruction or touching invalid memory causes a synchronously generated signal.
- **Asynchronously Generated Signals**: These signals are not attributable to a specific thread. For example, signals sent via *kill*() or from the keyboard are asynchronously generated.

Most approaches deliver synchronously generated signals to the thread which generated them. For asynchronously generated signals an enormous space of behaviors is possible.

An outline of the major decisions which can be made concerning signals is:

- Are synchronous facilities provided for handling asynchronously generated signals?
- Are asynchronous signal handlers supported in multi-threaded programs? If so,
- Are (a) handler vectors, (b) signal masks, and (c) pending bits, maintained on a per-thread or per-process basis?
- Are asynchronously generated signals delivered to (a) a single distinguished thread, (b) a single arbitrarily chosen thread, (c) a single ''interested'' thread, (d) all ''interested'' threads, (e) all threads, (f) some combination of the above, or (g) none of the above?

To further muddy the waters, note that a SIGALRM sent as a result of an *alarm*() call might logically be considered to be attributable to the thread which called *alarm*(), making it very much like a synchronously generated signal, even though it is delivered asynchronously! And remember, of course, that the traditional implementation of *sleep*() (which should cause the calling thread to sleep) uses SIGALRM (which may or may not be delivered on a per-thread basis).

Suffice it to say that signals and multi-threaded programs taken together are complicated. Those desiring a more detailed treatment of the subject should consult the POSIX threads (pthreads) draft.

Areas Possibly Requiring New Functionality

Some problems come up in multi-threaded programs which wouldn't otherwise. This section examines two such areas and presents possible solutions.

Thread-Specific Data

Many types of computations require maintaining some amount of state on a per-thread basis. Such data is logically thread-specific, and can be thought of as additional context with which the thread executes. Single-threaded programs have no need for a thread-specific data mechanism since global or static storage can be considered to be ''specific'' to the single ''thread''.

Typically programs will require a number of different thread-specific data areas of different sizes and data types. Independently written modules must be able to allocate their own thread-specific data variables if the mechanism is to be usable in a modular fashion.

Several different thread-specific data mechanisms are possible:

1 **Keyed on Thread-ID**: The most primitive type of thread-specific data support requires only that each thread have some form of identifier which is available at runtime which can be used as a hash or index value. Hash tables keyed on the thread-id can then be constructed to contain thread-specific data values.

2 **Fixed Set of Locations**: Some thread packages provide a small fixed number of thread-specific data locations to the application. These can be used to construct a more general thread-specific data mechanism by using one such location to hold a pointer to larger, possibly extensible data structures.

3 **General Key-Value Mechanism**: Some thread packages provide facilities for allocating new thread-specific variable ''keys'' which are used to refer to thread-specific data values. Logically, each (key, thread-id) pair serves as an index into the set of values. Some mechanisms of this type also provide facilities for calling destructor functions on thread exit to perform any necessary thread-specific data cleanup. Alternatively, garbage collection could also be used to effect cleanup.

4 **Compiler and/or Architectural Support**: Some systems provide compiler and/or architectural support for thread-specific data. Some compilers, for instance, have introduced a ''thread'' storage class which is analogous to ''static'' and ''auto''. Some systems provide for thread-specific virtual memory mappings, allowing some virtual addresses to refer to thread-specific data. While this can be efficiently implemented on some hardware when threads are mapped directly onto processors, it means that each thread requires a full-weight kernel context, precluding lighter-weight coroutine-like thread contexts. Other systems support thread-specific data through use of a dedicated register.

Orderly Cancellation Mechanism

One style of multi-threaded programming allows a thread to cancel the computation being performed by another. The prototypical example application is parallel chess search: One thread has found a ''best'' move and thus can cancel the searches for moves which other threads are performing. Such facilities are unnecessary for single-threaded programs since there are no other threads to cancel.

One possible kind of cancellation is to simply stop a thread wherever it is and return its stack, etc. to the free pool. Unfortunately, this is inadequate and unsafe; if a thread is holding resources (.e.g., locks, file descriptors, dynamic memory, etc.) when canceled then they will never be released. This can cause deadlocks, resource leakage, or both.

At the opposite extreme, threads could be required to poll for cancellation. Unfortunately, this places both undue complexity and performance burdens on code which needs to be cancelable.

With orderly cancellation is should be possible to:

- establish handlers in program scopes which need to perform cleanup upon cancellation; this makes orderly cancellation possible.
- efficiently control the points at which cancellation may occur; this keeps the cleanup complexity manageable.

Not too surprisingly, this begins to look at lot like a general exception handling facility coupled with the ability to raise a cross-thread exception.

Two main approaches can be taken for supporting cancellation:

1 **Build from Existing Facilities**: It is possible to construct an orderly cancellation facility given a mechanism to asynchronously change the execution state (i.e., program counter) of another thread. For instance, certain kinds of per-thread signal support are sufficient. Cleanup handlers can be implemented as procedures which are logically pushed on scope entry (for scopes needing cleanup) and popped on scope exit.

2 **Compiler Support**: Compiler support for

exception handling and stack unwinding makes the bulk of cancellation trivial. Canceling a thread then becomes simply raising an exception in the thread to be canceled.

Other Multi-Threaded Library Issues

The Compilation Environment

A number of approaches are possible for providing a compilation environment for building multi-threaded programs. These approaches vary depending upon whether or not they continue to also support building traditional single-threaded programs and if so, how they do so. Some approaches are outlined below.

- **Single compilation environment**: This approach provides a single compilation environment for both single-threaded and multi-threaded programs which combines the facilities used by both kinds of programs. Simplicity is a major advantage of this approach. The main drawback is that single-threaded programs will use thread-safe versions of many library functions, incurring some unnecessary lock overhead. Another disadvantage is that multi-threaded programs can mistakenly call functions which are only usable in single-threaded programs.
- **Alternate compilation environment**: This approach provides a separate compilation environment for multi-threaded programs. It provides alternate versions of some include files, libraries, and possibly programs (e.g., ‘‘<stdio.h>’’, ‘‘<stdlib.h>’’, ‘‘libc.a’’, *cc*, etc.). The alternate versions could be selected at build time either by using search paths (with separate search paths for finding include files, libraries, and programs) or via explicit ‘‘-I’’ and ‘‘-L’’ switches to *cc*.

  This approach allows both single and multi-threaded programs to be built with appropriate headers and libraries: single-threaded programs pay no unnecessary lock costs; multi-threaded programs can only call multi-thread-safe library functions. Attempts to use inappropriate functions can be detected either at compile time or at link time. The main disadvantage of this approach is the complexity of establishing the correct search paths. Space is also required for two versions of many libraries.
- **Conditional compilation environment**: This approach selects additional or changed features needed by multi-threaded programs via compile-time switches and separate library names. For instance, some implementations require the symbol ‘‘_REENTRANT’’ to be defined when compiling multi-threaded programs, and use the suffix ‘‘_r’’ (e.g., ‘‘libc_r.a’’) for identifying reentrant versions of libraries. This allows a single set of include files to be used for both types of programs.

  Like the ‘‘alternate compilation environment’’ approach, both single and multi-threaded programs may be built with appropriate declarations and libraries. Inappropriate calls may be detected at compile or link time. Having a single set of include files increases simplicity. Also, a compiler option could be implemented which automatically defined the appropriate compiler symbols (e.g., ‘‘_REENTRANT’’) and rewrote library names to select reentrant versions (e.g., ‘‘-lc’’ to ‘‘-lc_r’’), making the environment selection process nearly foolproof. The one drawback of this approach is that space is required for two versions of many libraries.

A strong argument can be made against functions which are only usable in single-threaded programs being available when building multi-threaded programs; if the functions can’t be safely used they can only cause trouble and shouldn’t be available. Attempts to call them will likely have resulted from incompletely converted single-threaded code. Debugging will be far easier if such errors are caught at compile or link time than if they result in run time errors.

Strong arguments can also be made, however, for any new functions which are introduced specifically for multi-threaded programs also being available when building single-threaded programs; any functions which are useful in multi-threaded programs will also be useful in single-threaded programs since single-threaded programs are merely a special case of multi-threaded programs. Moreover, having the new functions available for both types of programs further enhances the potential compatibilities between them.

Compilation Support for Fast Locking

The time needed to successfully acquire and release an uncontested lock defines a lower bound on the time needed for a thread to access any shared data structure in a multi-threaded program. Since this lower bound will be a limiting factor for many programs, it is important that it be as small as possible.

If procedures are called to acquire and release locks, then the lock acquire/release bound will be at least two procedure call times. On many machines, this may be unacceptably high. Several methods may be used to eliminate these procedure calls:

- **Use of locking macros**: While lock acquisition typically requires a special instruction to be executed, lock release often only requires a simple store. On such machines, the lock release procedure call can be eliminated through the use of a macro.

- **Preprocessing assembler input**: Most C compilers will generate assembler input which can be transformed by other tools before assembly. Lock calls can be eliminated from the generated assembler input by substituting equivalent inline expansions. (While less common for applications code, this technique has a long history of use in building kernels.)
- **Compiler locking support**: Some compilers have support for directly generating machine-specific locking instructions. When such support is available it can be used to optimize the generated locking code.

Note that any machine-specific code can be embedded in macros or tools provided through machine-independent interfaces. Thus, while all these techniques rely on machine-specific features, each can still be used by portable programs.

Compatibility Between Single and Multiple Threaded Code

Source Compatibility

A large degree of source compatibility is possible between code using single-threaded and multi-threaded versions of C libraries. Many library functions are already reentrant and need no special treatment. Functions maintaining persistent state can use internal locking, allowing for source compatibility between single and multiple threaded calls. If *errno* is implemented as a thread-specific variable, then source compatibility can be maintained for functions which use it as well. In general, source compatibility is only inhibited when functions are used which have non-reentrant interfaces; on systems which allocate former statics as thread-specific data, nearly perfect source compatibility is achievable.

Object Compatibility

A certain degree of object compatibility is possible between code using single-threaded and multi-threaded versions of C libraries. Any previously reentrant functions will not have changed, preserving object compatibility. Likewise, if internal locking is done by multi-threaded library functions, then old single-threaded main programs may sometimes be successfully linked against them. For instance, an old single-threaded program might be linked with a new multi-threaded quicksort routine, accomplishing a substantial speedup for very little effort. Most exported library data structures will not need to have locks added and so can remain unchanged.

Some library data structures do need to have locks added, but these can often be added in upward-compatible ways. In particular, while a recursive (counted) mutex lock must logically be added to every stdio ''FILE'' buffer, the locks can actually be allocated in a parallel data structure. This preserves the layout of the original ''FILE'' structure, allowing single-threaded objects using stdio functions to call new multi-threaded objects also using them, and allowing some single-threaded objects which use stdio functions to be called by multi-threaded programs (provided that the multi-threaded code performs explicit *flockfile*() and *funlockfile*() calls around uses of the old objects). Of course, a performance price will be paid for this type of object compatibility; finding locks in parallel data structures will almost certainly be slower than accessing them directly in expanded ''FILE'' structures.

Object compatibility for functions with non-reentrant interfaces is achievable when a common compilation/execution environment is used for both single and multi-threaded programs. Such objects can be compatible when both the single and multi-threaded programs actually allocate logically static return values in thread-specific storage.

One other trick (a.k.a. ''hack'') may be performed to enhance object compatibility. Assuming that *errno* is allocated in thread-specific storage and is accessed via a clever macro or function, a global int named *errno* can still be present as well. Multi-threaded versions of library functions can then both set the thread-specific and global *errno* values whenever an *errno* value is returned. This allows the multi-threaded version of such a function to be called by single-threaded code which expects the *errno* value in the global.

Approaches Being Taken by the UNIX Community

A number of different implementations of multi-threaded libraries have been built by research and industry groups. Several more are proposed or under construction. Some of those which already exist have been built by Mach, Encore, Apollo, Chorus, Convex, and the Open Software Foundation. The OSF has implemented the draft POSIX Threads proposal (P1003.4a a.k.a. ''pthreads'') which includes reentrant libraries. Implementations of a UNIX International threads requirements document which specifies reentrant libraries are are under way by UNIX System Laboratories (USL, formerly the UNIX Software Operation of AT&T), Sun and other member companies. Substantial numbers of multi-threaded library implementations are appearing. Happily to a large extent, many of these implementations agree on the approaches taken.

A more detailed presentation of some of the particular choices made is presented below. While not intended to be comprehensive, the choices presented are intended to be indicative of some of the directions that the industry has taken.

- **Cache Synchronization Techniques**: All known implementations of multi-threaded C libraries perform any necessary cache synchronization operations automatically.

- **errno**: Unanimous agreement seems evident within the UNIX community that *errno* should be maintained on a per-thread basis. Several implementations such as those by Encore, Chorus, and USL also store into a global *errno* value.
- **The stdio Functions**: All multi-threaded implementations use internal locking for the stdio functions. Most, including Mach, pthreads/OSF, UI/USL, Sun, Convex, and Chorus have taken the ''safe allowing fast'' approach towards the stdio macros (e.g., *putc*()). Most also provide a *flockfile*()-like primitive permitting explicit locking control.
- **Functions with Non-Reentrant Interfaces**: Two different approaches have been adopted for functions with non-reentrant interfaces. Chorus and Convex have chosen to maintain such state in thread-specific storage. (Chorus does this in software; Convex does this with virtual memory.) Others such as pthreads/OSF and UI/USL have chosen to provide alternate reentrant versions of such functions.
- **Functions which Maintain State Across Invocations**: All multi-threaded implementations of *malloc*(), *free*(), etc. use internal locking. Beyond this agreement, the same two approaches have been taken for functions which maintain state across invocations as were taken for functions with non-reentrant interfaces.
- **Signals**: Signal treatment is an area of divergence. Mach and OSF/1 have per-process signals. Chorus has broadcast to ''interested'' threads signals. Encore has several implementations. Pthreads is going to ballot with per-thread singly delivered signals.
- **Thread-Specific Data**: The Mach C-Threads package provides a single thread-specific data variable. Chorus uses a key-value scheme. Convex uses architectural support for thread-private memory. Sun provides compiler support for a small set of thread-private data locations. Pthreads provides a general key-value scheme with destructors.
- **Orderly Cancellation Mechanism**: The pthreads proposal contains cancellation interfaces which can be layered onto existing facilities. (This was modeled after the Alert facility used in Modula-2+.)
- **Compilation Environment**: The two main approaches taken towards compilation environments have been to provide a single compilation environment, and to provide a conditional compilation environment. Apollo and Chorus have both adopted single unified compilation environments. Mach, OSF, and USL have all adopted conditional compilation environments.
- **Compilation Locking Support**: Mach uses both locking macros and (optional) preprocessing of assembler input to eliminate procedure calls for locking. Convex has compiler support for locking and parallel code generation. UI/USL plans to support inline lock expansion on some architectures.
- **Compatibility Between Single and Multiple Threaded Code**: Convex and Chorus achieve nearly perfect compatibility between single and multi-threaded code through the use of thread-specific data. Pthreads/OSF and UI/USL, on the other hand, both took the position that using thread-specific data to approximate perfect source compatibility represented unnecessary and non-obvious mechanism, and that multi-threaded programs would be better served by using new reentrant functions instead.

Conclusions

UNIX-like systems supporting multi-threaded programs are becoming increasingly common. Both parallel programming requirements and the advantages of a synchronous programming paradigm are driving the industry in this direction.

As systems supporting multi-threaded programs become more common, increasing numbers of applications are being written as multi-threaded programs. To the extent that programmers can capitalize on existing C library interfaces and implementations when writing multi-threaded code, such code will be both easier to write and easier to understand.

While migrating the C libraries to be usable in multi-threaded programs allows for a wide range of possible design choices, some alternatives have been shown to have clear advantages. These advantages have been recognized and utilized by many of the multi-threaded C library implementations which have appeared to date.

While clearly many details differ between the existing implementations, many of the core elements, such as using internal locking whenever possible, do agree. Measured in terms of the similarity of commonly used routines, they typically have far more in common than they have differences. There has been substantial industry convergence around the important issue of multi-threaded C libraries.

Nonetheless, a number of details are different between the many implementations. In the long run, the industry will not be well served by numerous interfaces which are similar in spirit but incompatible in practice. Collaboration by a wide range of the industry on the POSIX pthreads effort provides one possible sign that convergence will continue.

Finally, of course, multi-threaded libraries are not an end in and of themselves. Nonetheless, they will provide (and are already providing) an important basic tool for utilizing parallel programming to accomplish real-world tasks.

Acknowledgments

A number of individuals deserve special mention for having contributed to my understanding of the issues presented in this paper. In particular, Eric Cooper (of CMU), Bill Cox (of USL), Rick Greer (of Interactive), Bob Knighten (of Encore), Simon Patience (of OSF), and Garret Swart (of DEC SRC) have all participated in long (and sometimes actually enjoyable!) sessions where we tried to look at all the alternatives. Trish Jones (my wife!) also deserves special mention both for her technical writing help and her patience.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. Summer 1986 USENIX Technical Conference and Exhibition*. June, 1986.

[2] American National Standards Institute. *American National Standard for Information Systems, Programming Language, C.* American National Standards Institute, Inc., 1990. ANSI X3.159-1989.

[3] Francois Armand, Frederic Herrmann, Jim Lipkis, and Marc Rozier. Multi-threaded Processes in Chorus/MIX. In *Proc. of EUUG Spring'90 Conference*, pages 1-13. Munich, Germany, April, 1990. Chorus systemes Technical Report CS/TR-89-37.3.

[4] *Unix System V Release 4.0 Programmer's Reference Manual* AT&T, 1989.

[5] Eric C. Cooper, Richard. P. Draves. *C Threads.* Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, June, 1988.

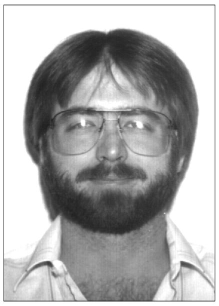
[6] Julie Kucera, CONVEX Computer Corporation. Making libc Suitable for Use by Parallel Programs. In *First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*. October, 1989.

[7] Paul R. McJones, Garret F. Swart. *Evolving the UNIX System Interface to Support Multithreaded Programs.* Technical Report Research Report 21, Digital Systems Research Center, September, 1987.

[8] Open Software Foundation. *Application Environment Specification (AES), Operating System Programming Interfaces Volume.* Prentice Hall, Inc., 1990. ISBN 0-13-043522-8.

[9] *Threads Extension for Portable Operating Systems.* Technical Committee on Operating Systems of the IEEE Computer Society for Work Item: JTC1.22.21.2, 1990. Draft P1003.4a/D5.

[10] UNIX International. *Multiprocessing Work Group, Final Report.* UNIX International, Inc., 1989. Draft of 24 October 1989.



Michael Jones is currently pursuing his Ph.D. in Computer Science at Carnegie Mellon University. His interests include operating systems, parallel programming and his fellow human beings. He implemented the multi-thread safe versions of the stdio functions for the Mach project upon which the OSF and Encore libraries are based. He has also worked on the Mach C-Threads implementation and the Mach Kernelization project. He is an active participant in the POSIX pthreads effort and in the UNIX International Multiprocessing Working Group. His U.S. Mail address is Michael B. Jones; Carnegie Mellon University; Computer Science Department; Pittsburgh, PA 15213. His email address is Michael.Jones@cs.cmu.edu .