Interfaces and Threads: A Taxonomy and Guidelines

Revision 0.9

Introduction

This document presents a taxonomy of interface behavior with respect to threads and some simple guidelines to defining multithreaded interfaces and implementations. The intended consumers of this document are the Architecture Review Committees and their clients. The goal of the taxonomy is to define labels for behavior that are to be used in interface documentation so that programmers can quickly determine an interface's behavior when used in a multithreaded program. The criteria for determining the labeling of an interface need not be absolute; the labels are for human programmers and are not necessarily strictly verifiable by machine. Therefore aspects of the taxonomy may require some "taste" to apply.

Definitions

Object

For the purposes of this discussion, the term "object" refers to a data structure whose contents is completely maintained by an interface or set of interfaces, whose contents is opaque to the users of the object and which is only passed in interfaces by indirect reference (pointer, index, ID ...).

Data-race

The possibility that more than one thread will simultaneously access (read or write) a location, with at least one thread modifying the location, is a *data-race*. The possibility that more than one thread reads a location simultaneously is *not* a data-race. The functions that a program may use to prevent data-races are:

```
cond_timedwait() mutex_trylock()  rw_unlock()     thr_create()
cond_wait()      mutex_unlock()   rw_wrlock()     thr_join()
fork()           rw_rdlock()      sema_post()     wait()
fork1()          rw_tryrdlock()   sema_trywait()  waitid()
mutex_lock()     rw_trywrlock()   sema_wait()     waitpid()
```

These functions are defined to ensure that the order of memory operations is preserved with respect to the execution of the routine[1]. Using the blocking, thread creation, or thread destruction characteristics of these functions, the programmer must ensure that it is not possible for a thread to write to a particular location at the same time that another thread is either reading or writing it[2].

The exception to this is locations that are accessed as primitive types and are guaranteed by the machine architecture *and* the compiler to be accessed atomically with respect to multiple processors and signals. Such locations must be accessed using the same data type (that is marked "volatile" in C and C++) when it is possible for more than one thread to be accessing and modifying it. Any sequence of such access as seen by other processors, may be reordered by the hardware. So this mode can only provide access to one independent datum.

For Sun (SPARC, X86, PowerPC), system software may assume that 32 bit, 16 bit and 8 bit primitive types guarantee atomicity. Software that is intended for portability to more diverse environments should be evaluated on a case by case basis to see if it must be more restrictive.

1. This is usually done by executing the appropriate special instructions and/or having the compiler recognize calls to these routines and avoid optimizing memory operations so that the order with respect to calls to these routines is changed.

2. The compilation system must also recognize that the ordering of access to global memory with respect to these functions must not be changed by optimization.

MT-safe

An interface can be labelled MT-safe if it satisfies the following criteria:

1. The interface can be simultaneously invoked by more than one thread of control and will function correctly according to the semantics of the function.

2. The caller is not required to do any explicit synchronization before calling the function to protect objects manipulated by the interface. However references to memory not under the control of the interface or set of interfaces must be locked appropriately by the caller. For the purposes of locking, the caller must assume that the referenced memory can be modified, unless modification is explicitly disallowed in the documentation.

3. The implementation of the interfaces must be data-race free.

4. The implementation of the interface must contain enough concurrency to be usefully used by more than one thread.

Discussion

The correct behavior of an interface when invoked by more than one thread need not be the behavior of some serial execution of the interface. In other words, the behavior need not be serializable. For example, consider an interface that adds a series of items to a list. The interface can be considered MT-safe if it correctly adds the items one at a time instead of adding them all atomically. Therefore concurrent executions of this interface may interleave the items in a way not reproducible by a serial execution of the identical calls. On the other hand, if the behavior of the interface is serializable, then the interface is MT-safe (unless it fails under the fourth "performance" criterion)—serializability is a sufficient but not necessary condition.

The second criterion allows interfaces such as `memcpy()` or `strlen()` to made MT-safe, even though some of the memory they manipulate is not directly under interface control. For example, two threads calling `memcpy()` on the same memory is a data-race.

The last criterion, that the interfaces must be usefully usable by more than one thread, is subjective. The purpose of this criterion is to label interfaces that will function correctly, but have such severe concurrency restrictions that there are very few multithreaded programs that will be able to use them as MT-safe. An example of this is the RPC library. Suppose the implementation of the RPC library contained a single lock that is held for the duration of any RPC call. This allows only one RPC to be active at any one time. There would be almost no useful advantage in a multithreaded program calling it from more than one thread. Furthermore, an application programmer would probably be surprised (and probably annoyed) by this behavior if this interface were labeled as safe to use by more than one thread.

Implementations that only allow one thread at a time to execute the code can still be labelled MT-safe if this implementation is useful to many MT programs. For example, if the implementation does not do a large amount of computation or potentially long I/O before returning to the caller than an implementation using only one lock can still be MT-safe. In the RPC case, an implementation that allows concurrent RPC to different hosts but only allowed one concurrent RPC per client handle, may be acceptable.

Explicitly Locked

Explicitly locked interfaces impose the restriction that calls to such interfaces by a thread must be surrounded by calls to associated interface routines that perform some locking protocol. Explicitly locked interface will not behave correctly when called by more than one thread of control otherwise.

Explicitly locked interfaces are usually appropriate only in interfaces that are critical to performance. Each use should be carefully justified. Otherwise the interface should be made MT-safe, which is far easier to use.

Even when an explicitly locked interface is provided, it is common practice to also provide an MT-safe version of the interface with the explicitly locked interface carefully named to ensure that the programmer understands that this interface must be used with a special protocol.

An example of an explicitly locked interface is `putc_unlocked()` which must be surrounded by calls to `flockfile()` and `funlockfile()`.

Object-unsafe

Object-unsafe interfaces impose the restriction that concurrent uses of this interface or set of interfaces must operate on distinct objects. Invocations that operate on the same object simultaneously will not function correctly. The caller must ensure that only one thread at a time operates on an object by using the memory synchronization functions (See “Data-race” on page 2) before all calls to an interface or set of interfaces. The synchronization cannot be done using “volatile” global variables. The documentation must be clear what the set of interfaces that operate on the object are.

There are few guidelines on the use of object-unsafe interfaces either internal or external to Sun. Externally this category does not seem to be recognized widely and such interfaces are generally categorized as unsafe. Therefore, the definition of object-unsafe interfaces is highly discouraged. In addition, determining that threads only call an interface with unique objects can be error-prone and once the error is made it is extremely hard to debug (as are most data-races).

Package-unsafe

Package-unsafe interfaces impose the restriction that only one thread at a time may call any interface in a set of interfaces. There may also be sequences of operations that carry state between invocations that must be locked around so that they complete atomically. The key point is that it is possible to do useful calls into the set of interfaces from more than one thread by the caller providing its own locking.

In many cases the cause of an implementation not being package-unsafe is the binary using the global `errno`. This causes calls to this interface to affect and be affected by calls to unrelated interfaces. Simply ensuring that the source includes `<errno.h>` and recompiling the library with `-D_REENTRANT` will fix this problem. However, access to other global variables shared with other packages require other fixes.

Unsafe

Unsafe interface meet none of the above criterion. They are unsafe to call from any thread but the main thread.

For the sake of simplicity, packages that may be called from more than one thread but are harmful to concurrency (i.e. they don’t meet the last criterion for being labeled MT-safe) also fall into this category (rather than inventing another label).

Other Definitions

These definitions are informal in nature and are mainly for internal consumption. They should not be used in external documentation, but may be used in design documents.

MT-hot

An interface implementation is MT-hot if it increases the degree of concurrency beyond the minimum for being MT-safe. An MT-hot implementation may, in fact, be slower than an a non-MT-hot implementation when run on a uniprocessor. An MT-hot implementation may not actually run concurrently on more than one CPU; it may instead take advantage of extra I/O parallelism. For example, an implementation of RPC that allows concurrent RPCs to different hosts, but serializes access to one host may be considered to be adequately concurrent to be MT-safe. However an implementation that allows concurrent RPCs to same or different hosts may be MT-hot even though the internal implementation only allows the RPC code to be executed by only one thread at a time (the lock are released while the RPC is in progress).

Thread Interface Guidelines

Per-thread State

Older interfaces sometimes were designed assuming a single thread of control. For example interface that return a pointer to an internal static buffer (e.g. `getpwent()`). Another example are interfaces that build rely on an external, global context that is an implicit argument to a set of interfaces (e.g. `setlocale()`). One solution to this problem is to define per-thread buffers or per-thread contexts using thread-specific data (TSD). This solution should be avoided, if possible, since TSD can be expensive to maintain. More seriously, keeping per-thread state can prevent more than one thread from cooperating to use an interface. However, sometime a legacy interface is so pervasive that this cannot be avoided (e.g. `errno`).

Typically this is done to pass implicit arguments (or return values) to routines. Consider an implementation of a transaction system that causes a series of invocations on a set of interfaces to be handled atomically. There is a temptation to not augment the interfaces to include a transaction context argument and instead maintain the transaction context per-thread. First, implicit arguments are usually bad programming practice as they can make a program much harder to understand. Secondly this can prevent more than one thread from cooperating to accomplish a particular transaction.