Programming Guidelines 7

This chapter gives some pointers on programming with threads. Differences between single-threaded thinking and multithreaded thinking are emphasized.

Historically, most code has been designed for single-threaded programs. This is especially true for most of the library routines called from C programs. The following implicit assumptions were made for single-threaded code:

- When you write into a global variable and then, a moment later, read from it, what you read is exactly what you just wrote.
- This is also true for nonglobal, static storage.
- You do not need synchronization because there is nothing to synchronize with.

The next few examples discuss some of the problems that arise in multithreaded programs because of these assumptions, and how you can deal with them.

7

Rethinking Global Variables

Traditional, single-threaded C and UNIX have a convention for handling errors detected in system calls. System calls can return anything as a functional value (for example, `write()` returns the number of bytes that were transferred). However, the value -1 is reserved to indicate that something went wrong. So, when a system call returns -1, you know that it failed.

Code Example 7-1 Global Variables and `errno`

```
extern int errno;
...
if (write(file_desc, buffer, size) == -1) {
    /* the system call failed */
    fprintf(stderr, "something went wrong, "
        "error code = %d\n", errno);
    exit(1);
}
...
```

Rather than return the actual error code (which could be confused with normal return values), the error code is placed into the global variable `errno`. When the system call fails, you can look in `errno` to find what went wrong.

Now consider what happens in a multithreaded environment when two threads fail at about the same time, but with different errors. Both expect to find their error codes in `errno`, but one copy of `errno` cannot hold both values. This global variable approach simply does not work for multithreaded programs.

The Solaris threads package solves this problem through a conceptually new storage class—thread-specific data. This storage is similar to global storage in that it can be accessed from any procedure in which a thread might be running. However, it is private to the thread—when two threads refer to the thread-specific data location of the same name, they are referring to two different areas of storage.

So, when using threads, each reference to `errno` is thread specific because each thread has a private copy of `errno`.

Providing for Static Local Variables

Code Example 7-2 shows a problem similar to the errno problem, but involving static storage instead of global storage. The function gethostbyname (3N) is called with the computer name as its argument. The return value is a pointer to a structure containing the required information for contacting the computer through network communications.

Code Example 7-2 The gethostbyname() Problem

```
struct hostent *gethostbyname(char *name) {
    static struct hostent result;
        /* Lookup name in hosts database */
        /* Put answer in result */
    return(&result);
}
```

Returning a pointer to an automatic local variable is generally not a good idea, although it works in this case because the variable is static. However, when two threads call this variable at once with different computer names, the use of static storage conflicts.

Thread-specific data could be used on a replacement for static storage, as in the errno problem, but this involves dynamic allocation of storage and adds to the expense of the call.

A better way to handle this kind of problem is to make the caller of gethostbyname() supply the storage for the result of the call. This is done by having the caller supply an additional argument, an output argument, to the routine. This requires a new interface to gethostbyname().

This technique is used in Solaris threads to fix many of these problems. In most cases, the name of the new interface is the old name with "_r" appended, as in gethostbyname_r(3N).

Synchronizing Threads

The threads in an application must cooperate and synchronize when sharing the data and the resources of the process.

A problem arises when multiple threads call something that manipulates an object. In a single-threaded world, synchronizing access to such objects is not a problem, but as Code Example 7-3 illustrates, this is a concern with multithreaded code. (Note that the Solaris `printf`(3S) is safe to call for a multithreaded program; this example illustrates what could happen if `printf`() were not safe.)

Code Example 7-3 The `printf`() Problem

```
/* thread 1: */
    printf("go to statement reached");


/* thread 2: */
    printf("hello world");



printed on display:
    go to hello
```

Single-Threaded Strategy

One strategy is to have a single, application-wide mutex lock that is acquired whenever any thread in the application is running and is released before it must block. Since only one thread can be accessing shared data at any one time, each thread has a consistent view of memory.

Because this is effectively a single-threaded program, very little is gained by this strategy.

Reentrance

A better approach is to take advantage of the principles of modularity and data encapsulation. A reentrant function is one that behaves correctly if it is called simultaneously by several threads. Writing a reentrant function is a matter of understanding just what *behaves correctly* means for this particular function.

Functions that are callable by several threads must be made reentrant. This might require changes to the function interface or to the implementation.

Functions that access global state, like memory or files, have reentrance problems. These functions need to protect their use of global state with the appropriate synchronization mechanisms provided by Solaris threads.

The two basic strategies for making functions in modules reentrant are code locking and data locking.

Code Locking

Code locking is done at the function call level and guarantees that a function executes entirely under the protection of a lock. The assumption is that all access to data is done through functions. Functions that share data should execute under the same lock.

Some parallel programming languages provide a construct called a monitor that implicitly does code locking for functions that are defined within the scope of the monitor. A monitor can also be implemented by a mutex lock.

Functions under the protection of the same mutex lock or within the same monitor are guaranteed to execute atomically with respect to each other.

Data Locking

Data locking guarantees that access to a *collection* of data is maintained consistently. For data locking, the concept of locking code is still there, but code locking is around references to shared (global) data, only. For a mutual exclusion locking protocol, only one thread can be in the critical section for each collection of data.

Alternatively, in a multiple readers, single writer protocol, several readers can be allowed for each collection of data or one writer. Multiple threads can execute in a single module when they operate on different data collections and do not conflict on a single collection for the multiple readers, single writer protocol. So, data locking typically allows more concurrency than does code locking.

What strategy should you use when using locks (whether implemented with mutexes, condition variables, or semaphores) in a program? Should you try to achieve maximum parallelism by locking only when necessary and unlocking as soon as possible (*fine-grained locking*)? Or should you hold locks for long periods to minimize the overhead of taking and releasing them (*coarse-grained locking*)?

The granularity of the lock depends on the amount of data it protects. A very coarse-grained lock might be a single lock to protect all data. Dividing how the data is protected by the appropriate number of locks is very important. Too fine a grain of locking can degrade performance. The small cost associated with acquiring and releasing locks can add up when there are too many locks.

The common wisdom is to start with a coarse-grained approach, identify bottlenecks, and add finer-grained locking where necessary to alleviate the bottlenecks. This is reasonably sound advice, but use your own judgment about taking it to the extreme.

Invariants

For both code locking and data locking, *invariants* are important to control locking complexity. An invariant is a condition or relation that is always true.

The definition is modified somewhat for concurrent execution: an invariant is a condition or relation that is true when the associated lock is being set. Once the lock is set, the invariant can be false. However, the code holding the lock must reestablish the invariant before releasing the lock.

An invariant can also be a condition or relation that is true when a lock is being set. Conditional variables can be thought of as having an invariant that is the condition.

Code Example 7-4 Testing the Invariant With `assert`(3X)

```
mutex_lock(&lock);
while((condition)==FALSE)
    cond_wait(&cv,&lock);
assert((condition)==TRUE);
  .
  .
  .
mutex_unlock(&lock);
```

The `assert()` statement is testing the invariant. The `cond_wait()` function does not preserve the invariant, which is why the invariant must be re-evaluated when the thread returns.

Another example is a module that manages a doubly linked list of elements. For each item on the list a good invariant is the forward pointer of the previous item on the list that should also point to the same thing as the backward pointer of the forward item.

Assume this module uses code-based locking and therefore is protected by a single global mutex lock. When an item is deleted or added the mutex lock is acquired, the correct manipulation of the pointers is made, and the mutex lock is released. Obviously, at some point in the manipulation of the pointers the invariant is false, but the invariant is reestablished before the mutex lock is released.

Avoiding Deadlock

Deadlock is a permanent blocking of a set of threads that are competing for a set of resources. Just because some thread can make progress does not mean that there is not a deadlock somewhere else.

The most common error causing deadlock is *self deadlock* or *recursive deadlock*: a thread tries to acquire a lock it is already holding. Recursive deadlock is very easy to program by mistake.

For example, if a code monitor has every module function grabbing the mutex lock for the duration of the call, then any call between the functions within the module protected by the mutex lock immediately deadlocks. If a function calls some code outside the module which, through some circuitous path, calls back into any method protected by the same mutex lock, then it will deadlock too.

The solution for this kind of deadlock is to avoid calling functions outside the module when you don't know whether they will call back into the module without reestablishing invariants and dropping all module locks before making the call. Of course, after the call completes and the locks are reacquired, the state must be verified to be sure the intended operation is still valid.

An example of another kind of deadlock is when two threads, thread 1 and thread 2, each acquires a mutex lock, A and B, respectively. Suppose that thread 1 tries to acquire mutex lock B and thread 2 tries to acquire mutex lock A. Thread 1 cannot proceed and it is blocked waiting for mutex lock B. Thread 2 cannot proceed and it is blocked waiting for mutex lock A. Nothing can change, so this is a permanent blocking of the threads, and a deadlock.

This kind of deadlock is avoided by establishing an order in which locks are acquired (a *lock hierarchy*). When all threads always acquire locks in the specified order, this deadlock is avoided.

Adhering to a strict order of lock acquisition is not always optimal. When thread 2 has many assumptions about the state of the module while holding mutex lock B, giving up mutex lock B to acquire mutex lock A and then reacquiring mutex lock B in order would cause it to discard its assumptions and reevaluate the state of the module.

The blocking synchronization primitives usually have variants that attempt to get a lock and fail if they cannot, such as `mutex_trylock`(). This allows threads to violate the lock hierarchy when there is no contention. When there is contention, the held locks must usually be discarded and the locks reacquired in order.

Scheduling Deadlocks

Because there is no guaranteed order in which locks are acquired, a common problem in threaded programs is that a particular thread never acquires a lock (usually a condition variable), even though it seems that it should.

This usually happens when the thread that holds the lock releases it, lets a small amount of time pass, and then reacquires it. Because the lock was released, it might seem that the other thread should acquire the lock. But, because nothing blocks the thread holding the lock, it continues to run from the time it releases the lock until it reacquires the lock, and so no other thread is run.

You can usually solve this type of problem by calling `thr_yield`(3T) just before the call to reacquire the lock. This allows other threads to run and to acquire the lock.

Because the time-slice requirements of applications are so variable, the threads library does not impose any. Use calls to `thr_yield`() to make threads share time as you require.

Locking Guidelines

Here are some simple guidelines for locking.

- Try not to hold locks across long operations like I/O where performance can be adversely affected.
- Don't hold locks when calling a function that is outside the module and that might reenter the module.
- Don't try for excessive processor concurrency. Without intervening system calls or I/O operation, locks are usually held for short amounts of time and contention is rare. Fix only those locks that have measured contention.
- When using multiple locks, avoid deadlocks by making sure that all threads acquire the locks in the same order.

Following Some Basic Guidelines

- Know what you are importing and whether it is safe.

  A threaded program cannot arbitrarily enter nonthreaded code.
- Threaded code can safely refer to unsafe code only from the initial thread.

  This ensures that the static storage associated with the initial thread is used only by that thread.
- Sun-supplied libraries are defined to be *safe* unless explicitly documented as unsafe.

  If a reference manual entry does not say whether a function is MT-Safe, it is safe. All MT-unsafe functions are identified explicitly in the manual page.
- Use compilation flags to manage binary incompatible source changes.

  Either specify `-D_REENTRANT` when compiling or be sure that `_REENTRANT` is defined before any header file is included.
- When making a library safe for multithreaded use, do not thread global process operations.

  Do not change global operations (or actions with global side effects) to behave in a threaded manner. For example, if file I/O is changed to per-thread operation, threads cannot cooperate in accessing files.

  For thread-specific behavior, or *thread cognizant* behavior, use thread facilities. For example, when the termination of `main()` should terminate only the thread that is exiting `main()`, the end of `main()` should be:

  ```
  thr_exit();
  /*NOTREACHED*/
  ```

Creating Threads

The Solaris threads package caches the threads data structure, stacks, and LWPs so that the repetitive creation of unbound threads can be inexpensive.

Unbound thread creation is very inexpensive when compared to process creation or even to bound thread creation. In fact, the cost is similar to unbound thread synchronization when you include the context switches to stop one thread and start another.

So, creating and destroying threads as they are required is usually better than attempting to manage a pool of threads that wait for independent work.

A good example of this is an RPC server that creates a thread for each request and destroys it when the reply is delivered, instead of trying to maintain a pool of threads to service requests.

While thread creation is relatively inexpensive when compared to process creation, it is not inexpensive when compared to the cost of a few instructions. Create threads for processing that lasts at least a couple of thousand machine instructions.

Thread Concurrency

By default, Solaris threads attempts to adjust the system execution resources (LWPs) used to run unbound threads to match the real number of active threads. While the Solaris threads package cannot make perfect decisions, it at least ensures that the process continues to make progress.

When you have some idea of the number of unbound threads that should be simultaneously active (executing code or system calls), tell the library through `thr_setconcurrency`(3T).

For example:

- A database server that has a thread for each user should tell Solaris threads the expected number of simultaneously active users.
- A window server that has one thread for each client should tell Solaris threads the expected number of simultaneously active clients.
- A file copy program that has one reader thread and one writer thread should tell Solaris threads that the desired concurrency level is two.

Alternatively, the concurrency level can be incremented by one through the `THR_NEW_LWP` flag as each thread is created.

Include unbound threads blocked on interprocess (`USYNC_PROCESS`) synchronization variables as active when you compute thread concurrency. Exclude bound threads—they do not require concurrency support from Solaris threads because they are equivalent to LWPs.

Efficiency

A new thread is created with `thr_create`(3T) in less time than an existing thread can be restarted. This means that it is more efficient to create a new thread when one is needed and have it call `thr_exit`(3T) when it has completed its task than it would be to stockpile an idle thread and restart it.

Bound Threads

Bound threads are more expensive than unbound threads. Because bound threads can change the attributes of the underlying LWP, the LWPs are not cached when the bound threads exit. Instead, the operating system provides a new LWP when a bound thread is created and destroys it when the bound thread exits.

Use bound threads only when a thread needs resources that are available only through the underlying LWP, such as a virtual time interval timer or an alternate stack, or when the thread must be visible to the kernel to be scheduled with respect to all other active threads in the system, as in realtime scheduling.

Use unbound threads even when you expect all threads to be active simultaneously. This allows Solaris threads to efficiently cache LWP and thread resources so that thread creation and destruction are fast.

Thread Creation Guidelines

Here are some simple guidelines for using threads.

- Use threads for independent activities that must do a meaningful amount of work.
- Use threads to take advantage of CPU concurrency.

- Use bound threads only when absolutely necessary, that is, when some facility of the underlying LWP is required.

Use `thr_setconcurrency`(3T) to tell Solaris threads how many threads you expect to be simultaneously active.

Working With Multiprocessors

The Solaris threads package lets you take advantage of multiprocessors. In many cases, programmers must be concerned with whether the multithreaded application runs on a uniprocessor or a multiprocessor.

One such case involves the memory model of the multiprocessor. You cannot always assume that changes made to memory by one processor are immediately reflected in the other processors' views of that memory.

Another multiprocessor issue is efficient synchronization when threads must wait until all have reached a common point in their execution.

Note – The issues discussed here are not important when the threads synchronization primitives are *always* used to access shared memory locations.

The Underlying Architecture

When threads synchronize access to shared storage locations using the Solaris threads synchronization routines, the effect of running a program on a shared-memory multiprocessor is identical to the effect of running the program on a uniprocessor.

However, in many situations a programmer might be tempted to take advantage of the multiprocessor and use "tricks" to avoid the synchronization routines. As Code Example 7-5 and Code Example 7-6 show, such tricks can be dangerous.

Understanding the memory models supported by common multiprocessor architectures helps to understand the dangers.

The major multiprocessor components are:

- The *processors* themselves
- *Store buffers*, which connect the processors to their caches

- *Caches*, which hold the contents of recently accessed or modified storage locations
- *memory*, which is the primary storage (and is shared by all processors).

In the simple traditional model, the multiprocessor behaves as if the processors are connected directly to memory: when one processor stores into a location and another immediately loads from the same location, the second processor loads what was stored by the first. Caches can be used to speed the average memory access, and the desired semantics can be achieved when the caches are kept consistent with one another.

A problem with this simple approach is that the processor must often be delayed to make certain that the desired semantics are achieved. Many modern multiprocessors use various techniques to prevent such delays, which, unfortunately, change the semantics of the memory model. Two of these techniques and their effects are explained in the next two examples.

“Shared-Memory” Multiprocessors

Consider the purported solution to the producer/consumer problem shown in Code Example 7-5. Although this program works on current SPARC-based multiprocessors, it assumes that all multiprocessors have strongly ordered memory. This program is therefore not portable.

Code Example 7-5 The Producer/Consumer Problem—Shared Memory Multiprocessors

```
                    char buffer[BSIZE];
                    unsigned int in = 0;
                    unsigned int out = 0;

void                                 char
producer(char item) {                  consumer(void) {
                                        char item;
    do
        ;/* nothing */                    do
    while                                     ;/* nothing */
        (in - out == BSIZE);               while
                                             (in - out == 0);
    buffer[in%BSIZE] = item;               item = buffer[out%BSIZE];
    in++;                                 out++;
}                                    }
```

When this program has exactly one producer and exactly one consumer and is run on a shared-memory multiprocessor, it appears to be correct. The difference between `in` and `out` is the number of items in the buffer. The producer waits (by repeatedly computing this difference) until there is room for a new item, and the consumer waits until there is an item in the buffer.

For memory that is *strongly ordered* (for instance, a modification to memory on one processor is immediately available to the other processors), this solution is correct (it is correct even taking into account that `in` and `out` will eventually overflow, as long as `BSIZE` is less than the largest integer that can be represented in a word).

Shared-memory multiprocessors do not necessarily have strongly ordered memory. A change to memory by one processor is not necessarily available immediately to the other processors. When two changes to different memory locations are made by one processor, the other processors do not necessarily see the changes in the order in which they were made because changes to memory don't happen immediately.

First the changes are stored in *store buffers* that are not visible to the cache. The processor looks at these store buffers to ensure that a program has a consistent view, but because store buffers are not visible to other processors, a write by one processor doesn't become visible until it is written to cache.

The Solaris synchronization primitives (see Chapter 3, "Programming With Synchronization Objects") use special instructions that flush the store buffers to cache. So, using locks around your shared data ensures memory consistency.

When memory ordering is very relaxed, Code Example 7-5 has a problem because the consumer might see that `in` has been incremented by the producer before it sees the change to the corresponding buffer slot. This is called *weak ordering* because stores made by one processor can appear to happen out of order by another processor (memory, however, is always consistent from the same processor). To fix this, the code should use mutexes to flush the cache.

The trend is toward relaxing memory order. Because of this, programmers are becoming increasingly careful to use locks around all global or shared data. As demonstrated by Code Example 7-5 and Code Example 7-6, locking is essential.

Peterson's Algorithm

The code in Code Example 7-6 is an implementation of Peterson's Algorithm, which handles mutual exclusion between two threads. This code tries to guarantee that there is never more than one thread in the critical section and that, when a thread calls `mut_excl()`, it enters the critical section sometime "soon."

An assumption here is that a thread exits fairly quickly after entering the critical section.

Code Example 7-6 Mutual Exclusion for Two Threads?

```
void mut_excl(int me /* 0 or 1 */) {
    static int loser;
    static int interested[2] = {0, 0};
    int other; /* local variable */

    other = 1 - me;
    interested[me] = 1;
    loser = me;
    while (loser == me && interested[other])
        ;

    /* critical section */
    interested[me] = 0;
}
```

This algorithm works some of the time when it is assumed that the multiprocessor has strongly ordered memory.

Some multiprocessors, including some SPARC-based multiprocessors, have store buffers. When a thread issues a store instruction, the data is put into a store buffer. The buffer contents are eventually sent to the cache, but not necessarily right away. (Note that the caches on each of the processors maintain a consistent view of memory, but modified data does not reach the cache right away.)

When multiple memory locations are stored into, the changes reach the cache (and memory) in the correct order, but possibly after a delay. SPARCbased multiprocessors with this property are said to have *total store order* (TSO).

When one processor stores into location *A* and then loads from location *B*, and another processor stores into location *B* and loads from location *A*, the expectation is that either the first processor fetches the newly modified value in location *B* or the second processor fetches the newly modified value in location *A*, or both, but that the case in which both processors load the old values simply cannot happen.

However, with the delays caused by load and store buffers, the “impossible case” can happen.

What could happen with Peterson’s algorithm is that two threads running on separate processors each stores into its own slot of the interested array and then loads from the other slot. They both see the old values (0), assume that the other party is not present, and both enter the critical section. (Note that this is the sort of problem that might not show up when you test a program, but only much later.)

This problem is avoided when you use the threads synchronization primitives, whose implementations issue special instructions to force the writing of the store buffers to the cache.

Parallelizing a Loop on a Shared-Memory Parallel Computer

In many applications, and especially numerical applications, while part of the algorithm can be parallelized, other parts are inherently sequential (as shown in Code Example 7-7).

Code Example 7-7 Multithreaded Cooperation (Barrier Synchronization)

```
while(a_great_many_iterations) {

    sequential_computation

    parallel_computation
}
```

For example, you might produce a set of matrices with a strictly linear computation, then perform operations on the matrices using a parallel algorithm, then use the results of these operations to produce another set of matrices, then operate on them in parallel, and so on.

The nature of the parallel algorithms for such a computation is that little synchronization is required during the computation, but synchronization of all the threads employed is required at the end to ensure that all have finished.

When the time spent executing the parallel algorithm is large compared to the time required to create and synchronize the threads, the cost of thread creation and synchronization is no problem. But if the time required for the computation is not so large, then the thread-creation and synchronization times become very important.

7

Summary

This guide has covered basic threads programming issues. Look in Appendix A, “Sample Application Code” for program examples that use many of the features and styles that have been discussed.

Further Reading

For more information related to the subjects in this guide, see the following books:

- *Algorithms for Mutual Exclusion* by Michel Raynal (MIT Press, 1986)
- *Concurrent Programming* by Alan Burns & Geoff Davies (Addison-Wesley, 1993)
- *Distributed Algorithms and Protocols* by Michel Raynal (Wiley, 1988)
- *Operating System Concepts* by Silberschatz, Peterson, & Galvin (Addison-Wesley, 1991)
- *Principles of Concurrent Programming* by M. Ben-Ari (Prentice-Hall, 1982)