The Thread-Monitor Library: A System for Monitoring Solaris-Threads Programs

Thomas W. Doeppner Jr.

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-53
November 1993

The Thread-Monitor Library

A System for Monitoring Solaris-Threads Programs*

Thomas W. Doeppner Jr.
Department of Computer Science
Brown University
Providence, RI 02912-1910

September 17, 1993

1 Introduction

Debugging a multi-threaded program involves all the difficulties of debugging a single-threaded program with the additional concern of the interaction between threads. Multi-threaded programs can be nondeterministic—the outcome of the program depends upon the ordering of the executions of the various threads. When testing a program, one would like to exert some control over this ordering, to force certain execution sequences to occur so that they can be tested.

The Thread-Monitor Library is an experimental system that allows one to monitor and control the execution of a multithreaded program (written using the Solaris Threads package). It creates separate windows for controlling each thread and separate windows for monitoring each synchronization object (currently, only mutexes and condition variables are supported). Whenever a thread enters a synchronization construct, its execution is immediately suspended and the programmer is prompted for permission to resume execution. The operations applied to each of the synchronization objects are listed in the synchronization object windows, allowing one to keep track of their state.

All of this is accomplished through a collection of "wrappers" for the Solaris Threads API. Rather than call the standard Solaris routine, one instead calls the appropriate wrapper which calls upon the monitor code as well as completing the call to the Solaris routine.

2 The API

Many of the Solaris Threads API routines have been augmented with a *message* parameter, allowing the caller to supply an informative message, which will be displayed as part of the action of the call. These messages are mandatory in some cases and are optional, but highly recommended in others. The modified version of each Solaris Threads routine is named by appending "`mon_`" to the front of its standard name. The message argument is the first argument, the remaining arguments are as they are in the Solaris API.

For example, rather than call `thr_create` with six arguments, one calls `mon_thr_create` with seven arguments—a message followed by the normal six arguments:

*. The research reported here was supported in part by ARPA order 8225, ONR grant N00014-91-J-4052; and by a grant from Sun Microsystems, Incorporated.

```
mon_thr_create("WRITER1", 0, 0, writer_code, 0, 0, 0);
```

For the following routines one *must* use the "mon_" version, since the message parameter is mandatory:

- thr_create
- mutex_init
- cond_init

For the following routines, the use of the "mon_" version is optional. If you use the Solaris version, an empty message is implicitly supplied.;

- mutex_lock
- mutex_unlock
- mutex_trylock
- cond_wait
- cond_timedwait
- cond_signal
- cond_broadcast

2.1 Creating and Terminating a Thread

The creation of a thread works exactly as it does in Solaris Threads, except that one must call mon_thr_create with an additional "message" argument, as already discussed. As part of thread creation, a new window is created, which is labelled with the message supplied in the call—the idea is that this message is the name of the thread. A creation message is written in the new window. Both the parent thread (the one calling mon_thr_create) and the child thread are suspended; neither will continue execution until you type a carriage return in the window. Every time the child thread calls a synchronization routine a message will be printed in the thread's window. In some cases (described below) the thread will be suspended and you will have to type a carriage return to resume it.

The main thread, though it is not created by explicitly calling mon_thr_create, also gets a window. This window is created the first time you call mon_thr_create, and is labelled "MAIN." As with the other thread windows, the MAIN thread's execution is suspended until you type a carriage return in the window.

One terminates a thread no differently than with Solaris Threads—either call thr_exit or return from the outermost procedure. At the moment, *detached* threads are not supported—if such a thread terminates, its window will not go away. When a *nondetached* thread terminates, its window will continue to exist until the thread is "joined," i.e., some other thread calls thr_join and the terminated thread's ID is returned. When this happens, a prompt is given in the terminating thread's window, and the thread which called thr_join is suspended until you type carriage return. Once you do this, the terminated thread's window will disappear in about two seconds.

When the MAIN thread terminates, a prompt will appear in its window. Once you type carriage return, this window disappears, as well as any other remaining windows. *In the cur-*

rent implementation, it is assumed that the MAIN thread is the last thread to terminate. It is unlikely that the library will work correctly if this isn't the case!

2.2 Mutexes and Condition Variables

Mutexes and condition variables are created by calling mon_mutex_init and mon_cond_init. Even if you are relying on zeroed storage, you must place these calls! The message parameter is used to name the mutex or condition variable and a window is created whose title is the message parameter. In this window will be displayed the operations performed on the object and which thread is performing them. These windows are there strictly to display the state of the mutex or condition variable—there is no need for you to type into them. The windows go away when you call mon_mutex_destroy and mon_cond_destroy, or when the program terminates.

2.3 Informational Calls

Two routines are supplied so that a thread can put a message in its window, helping give the programmer a clue about what the thread is doing:

- mon_send_info_wait
- mon_send_info_nowait

Both take a single parameter—a message to be printed in the calling thread's window. A thread making the first call will be suspended until you respond to the prompt in the window. A thread making the second call is not suspended—it simply writes the message and continues on.

2.4 Turning on Monitoring

To take full advantage of the Thread-Monitor library, you must turn it on. The following routines are used:

- mon_set_comm
- mon_set_debug

Both take a single integer as an argument—1 means *on* and 0 means *off*. You want to turn on *comm*; you probably should do this before you make any thread-related calls. You probably don't want to turn on *debug*; it causes a lot of diagnostic output to be written to the window in which you are running your program.

3 Using the Thread-Monitor Library

The library resides in /course/cs169/lib/monlib.a. The header file you will need to use it are in /course/cs169/include/thread_mon.h. Thus to compile a program that uses the library, you would use a command line resembling:

```
cc -o prog -I/course/cs169/include/thread_mon.h prog.c \
    /course/cs169/lib -lthread
```

3.1 Bugs, Etc.

If you wish to terminate your program while it is running, type <ctrl>C. This should have the usual effect of killing your program and it should cause all the windows set up by the library to go away. However, if any of the windows are prompting for a carriage return, you must type the carriage returns (in the appropriate windows). Once you do this, the windows should go away. If not, then you'll have to get rid of them manually: First type "tty" in the window in which you ran your program. It will respond with something like, "/dev/pty/5". Then type "ps -t pty/5". This will give you a list of all processes associated with that window. Among these will be some that are running *xterm*. These are the ones you want to get rid of. Each of these processes will have a number in the PID column (i.e., its process ID). Type the command "kill -9", followed by the list of the xterm process IDs. If this doesn't work, it's time to go home.

3.2 Features (and Lack Thereof)

The Thread-Monitor Library has been in existence for about one week. If you have any ideas on how it can be improved, we would be very interested in hearing about them. You might post your ideas to the cs169 news group, but also send them directly to *twd*.