Compiling and Debugging 6

This chapter describes how to compile and debug your multithreaded programs.

Compiling a Multithreaded Application	*page 109*
Debugging Multithreaded Programs	*page 111*

Compiling a Multithreaded Application

Using the C Compiler

Make sure the following software is available so you can successfully compile and link a multithreaded program.

- Include files:
  - `thread.h`
  - `errno.h`
- The standard C compiler
- The standard Solaris linker
- The threads library (`libthread`)
- MT-safe libraries (`libc`, `libm`, `libw`, `libintl`, `libmalloc`, `libmapmalloc`, `libnsl`, and so on)

Compiling With the _REENTRANT Flag

Compile multithread programs with the -D _REENTRANT flag.

This applies to every module of a new application. When the -D_REENTRANT flag is not present, the old definitions for errno, stdio, and so on, are used. To compile a single-threaded application, make sure that the _REENTRANT flag is undefined.

Link Old With New Carefully

Table 6-1 shows that multithreaded object modules should be linked with old object modules only with great caution.

Table 6-1 Compiling With and Without the _REENTRANT Flag

The File Type	Compiled	Reference	And Return
Old object files (non-threaded) and new object files	*Without* the _REENTRANT flag	Static storage	The traditional errno
New object files	*With* the _REENTRANT flag	__errno, the new binary entry point	The address of the thread's definition of errno
Programs using TLI in libnsl[1]	*With* the _REENTRANT flag (required)	__t_errno, a new entry point	The address of the thread's definition of t_errno.

1. Include tiuser.h to get the TLI global error variable.

Using libthread

To use libthread, specify -lthread before -lc on the ld command line, or last on the cc command line.

All calls to libthread are no-ops if the application does not link libthread.

libc has defined libthread stubs that are null procedures. True procedures are interposed by libthread when the application links both libc and libthread.

The behavior of the C library is undefined if a program is constructed with an `ld` command line that includes the fragment:

```
.o's ... -lc -lthread ...
```

Do not link a single-threaded program with `-lthread`. Doing so establishes multithreading mechanisms at link time that are initiated at run time. These waste resources and produce misleading results when you debug your code.

Using Non-C Compilers

The threads library uses the following items from `libc`.

- System call wrappers
- Something (usually `printf`()) to display error messages
- Runtime linking support to resolve symbols (because the library is dynamically linked)

You can eliminate these dependencies by writing both your own system call wrappers and your own `printf`() function, and by having the linker resolve all `libthread` symbols at link time rather than at runtime.

The threads library does not use dynamically allocated memory when the threads are created with application-supplied stacks. The `thr_create`(3T) routine lets the application specify its own stacks.

Debugging Multithreaded Programs

Common Oversights

The following list points out some of the more frequent oversights that can cause bugs in multithreaded programming.

- Using a local or global variable for passing an argument to a new thread
- Accessing global memory (shared changeable state) without the protection of a synchronization mechanism
- Creating deadlocks caused by two threads trying to acquire rights to the same pair of global resources in alternate order (so that one thread controls the first resource and the other controls the second resource and neither can proceed until the other gives up)

- Creating a hidden gap in synchronization protection. This is caused when a code segment protected by a synchronization mechanism contains a call to a function that frees and then reacquires the synchronization mechanism before it returns to the caller. The result is that it appears to the caller that the global data has been protected when it actually has not.
- Making deeply nested, recursive calls and using large automatic arrays can cause problems because multithreaded programs have a more limited stack size than single-threaded programs.
- Specifying an inadequate stack size
- Providing stack other than through the thread library calls

And, note that multithreaded programs (especially buggy ones) often behave differently in two successive runs given identical inputs because of differences in the thread scheduling order.

In general, multithreading bugs are statistical instead of deterministic in character. Tracing is usually more effective in finding problems in the order of execution than is breakpoint-based debugging.

Using `adb`

When you bind all threads in a multithreaded program, a thread and an LWP are synonymous. Then you can access each thread with the following `adb` commands that support multithreaded programming.

Table 6-2 MT `adb` commands

pid:A	Attaches to process # *pid*. This stops the process and all its LWPs.
:R	Detaches from process. This resumes the process and all its LWPs.
$L	Lists all active LWPs in the (stopped) process.
n:l	Switches focus to LWP # *n*
$l	Shows the LWP currently focused
num:i	Ignores signal number *num*

Using dbx

With the dbx utility you can debug and execute source programs written in C++, ANSI C, FORTRAN, and Pascal. dbx accepts the same commands as the SPARCworks™ Debugger but uses a standard terminal (tty) interface. Both dbx and the SPARCworks Debugger now support debugging multithreaded programs.

For a full overview of dbx and Debugger features see the SunPro dbx(1) man page and the *Debugging a Program* user's guide.

The following dbx options support multithreaded programs.

Table 6-3 dbx Options for MT Programs

cont at *line* [*sig signo id*]	Continues execution at line *line* with signal *signo*. See continue for dbx command language loop control. The *id*, if present, specifies which thread or LWP to continue. Default value is *all*.
lwp	Displays current LWP. Switches to given LWP [lwpid].
lwps	Lists all LWPs in the current process.
next ... tid	Steps the given thread. When a function call is skipped, all LWPs are implicitly resumed for the duration of that function call. Nonactive threads cannot be stepped.
next ... lid	Steps the given LWP. Does not implicitly resume all LWPs when skipping a function. The LWP on which the given thread is active. Does not implicitly resume all LWP when skipping a function.
step... tid	Steps the given thread. When a function call is skipped, all LWPs are implicitly resumed for the duration of that function call. Nonactive threads cannot be stepped.
step... lid	Steps the given LWP. Does not implicitly resume all LWPs when skipping a function.
stepi... lid	The given LWP.
stepi... tid	The LWP on which the given thread is active.
thread	Displays current thread. Switches to thread *tid*. In all the following variations, an optional *tid* implies the current thread.
thread -info [tid]	Prints everything known about the given thread.
thread -locks [tid]	Prints all locks held by the given thread.
thread -suspend [tid]	Puts the given thread into suspended state.

Table 6-3 `dbx` Options for MT Programs

`thread -continue [ tid ]`	Unsuspends the given thread.
`thread -hide [ tid ]`	*Hid*es the given (or current) thread. It will not show up in the generic `threads` listing.
`thread -unhide [ tid ]`	*Unhide*s the given (or current) thread.
`allthread-unhide`	*Unhide*s all threads.
`threads`	Prints the list of all known threads.
`threads-all`	Prints threads that are not usually printed (zombies).
`all|filterthreads-mode`	Controls whether `threads` prints all threads or filters them by default.
`auto|manualthreads-mode`	Enables automatic updating of the thread listing in the Thread Inspector of the GUI interface (SPARCworks Debugger).
`threads-mode`	Echoes the current modes. Any of the previous forms can be followed by a thread or LWP ID to get the traceback for the specified entity.