Compiling and Debugging 7

This chapter describes how to compile and debug multithreaded programs.

Compiling a Multithreaded Application	*page 155*
Debugging Multithreaded Programs	*page 159*

Compiling a Multithreaded Application

There are many options to consider for header files, define flags, and linking.

Preparing for Compilation

The following items are required to compile and link a multithreaded program. Except for the C compiler, all should come with your Solaris 2.x system.

- A standard C compiler
- Include files:
  - <thread.h> and <pthread.h>
  - `<errno.h>`, `<limits.h>`, `<signal.h>`, `<unistd.h>`
- The regular Solaris linker, `ln(1)`
- The Solaris threads library (`libthread`), the POSIX threads library (`libpthread`), and possibly the POSIX realtime library (`libposix4`) for semaphores
- MT-safe libraries (`libc`, `libm`, `libw`, `libintl`, `libnsl`, `libsocket`, `libmalloc`, `libmapmalloc`, and so on)

7

Choosing Solaris or POSIX Semantics

Certain functions, including the ones listed below, have different semantics in the POSIX 1003.1c standard than in the Solaris 2.4 release, which was based on an earlier POSIX draft. Function definitions are chosen at compile time. See the *man Pages(3): Library Routines* for a description of the differences in expected parameters and return values.

Table 7-1 Functions with POSIX/Solaris Semantic Differences

`sigwait`(2)	
`ctime_r`(3C)	`asctime_r`(3C)
`ftrylockfile`(3S) – new	`getlogin_r`(3C)
`getgrnam_r`(3C)	`getgrgid_r`(3C)
`getpwnam_r`(3C)	`getpwuid_r`(3C)
`readdir_r`(3C)	`ttyname_r`(3C)

The Solaris `fork(2)` function duplicates all threads (*fork-all* behavior), while the POSIX `fork(2)` function duplicates only the calling thread (*fork-one* behavior), as does the Solaris `fork1()` function.

The handling of an `alarm(2)` is also different: a Solaris alarm goes to the thread's LWP, while a POSIX alarm goes to the whole process (see page 126).

Including `<thread.h>` *or* `<pthread.h>`

The include file `<thread.h>`, used with the `-lthread` library, compiles code that is upward compatible with earlier releases of the Solaris system. This library contains both interfaces—those with Solaris semantics and those with POSIX semantics. To call `thr_setconcurrency(3T)` with POSIX threads, your program needs to include `<thread.h>`.

The include file `<pthread.h>`, used with the `-lpthread` library, compiles code that is conformant with the multithreading interfaces defined by the POSIX 1003.1c standard. For complete POSIX compliance, the define flag `_POSIX_C_SOURCE` should be set to a (long) value $\geq$ 199506:

`cc` [*flags*] *file*... **`-D_POSIX_C_SOURCE=`*`N`*** (*where* ***`N`*** **`199506L`**)

You can mix Solaris threads and POSIX threads in the same application, by including both `<thread.h>` and `<pthread.h>`, and linking with either the `-lthread` or `-lpthread` library.

In mixed use, Solaris semantics prevail when compiling with `-D_REENTRANT` and linking with `-lthread`, whereas POSIX semantics prevail when compiling with `-D_POSIX_C_SOURCE` and linking with `-lpthread`.

Defining `_REENTRANT` *or* `_POSIX_C_SOURCE`

For POSIX behavior, compile applications with the `-D_POSIX_C_SOURCE` flag set ≥ 199506L. For Solaris behavior, compile multithreaded programs with the `-D_REENTRANT` flag. This applies to every module of an application.

For mixed applications (for example, Solaris threads with POSIX semantics), compile with the `-D_REENTRANT` and `-D_POSIX_PTHREAD_SEMANTICS` flags.

To compile a single-threaded application, define neither the `_REENTRANT` nor the `-D_POSIX_C_SOURCE` flag. When these flags are not present, all the old definitions for `errno`, `stdio`, and so on, remain in effect.

To summarize, POSIX applications that define `-D_POSIX_C_SOURCE` get the POSIX 1003.1c semantics for the routines listed in Table 7-1. Applications that define only `-D_REENTRANT` get the Solaris semantics for these routines. Solaris applications that define `-D_POSIX_PTHREAD_SEMANTICS` get the POSIX semantics for these routines, but can still use the Solaris threads interface.

Linking With `libthread` *or* `libpthread`

For POSIX threads behavior, load the `-lpthread` library. For Solaris threads behavior, load the `-lthread` library. Some POSIX programmers might want to link with `-lthread` to preserve the Solaris distinction between `fork()` and `fork1()`. All that `-lpthread` really does is to make `fork()` behave the same way as the Solaris `fork1()` call, and change the behavior of `alarm(2)`.

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

To use `libpthread`, specify `-lpthread` before `-lc` on the `ld` command line, or last on the `cc` command line.

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

This diagram summarizes the compile options:

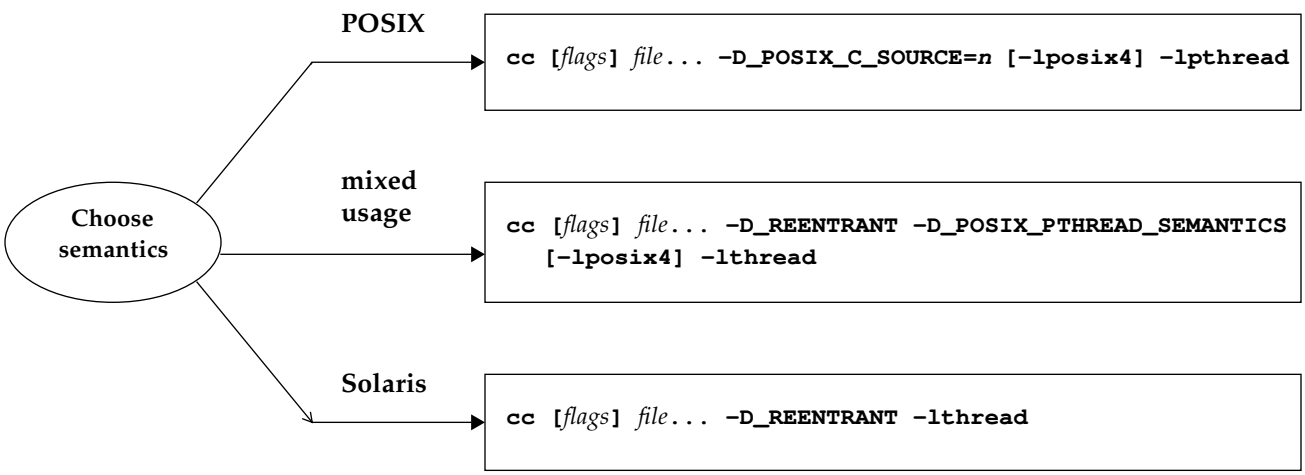

Figure 7-1 Compilation Flowchart

In mixed usage, you need to include both `<thread.h>` and `<pthread.h>`.

All calls to `libthread` and `libpthread` are no-ops if the application does not link `-lthread` or `-lpthread`. The runtime library `libc` has many predefined `libthread` and `libpthread` stubs that are null procedures. True procedures are interposed by `libthread` or `libpthread` when the application links both `libc` and the thread library.

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

```
    .o's ... -lc -lthread ... (this is incorrect)
or
    .o's ... -lc -lpthread ... (this is incorrect)
```

Linking with `-lposix4` *for POSIX Semaphores*

The Solaris semaphore routines, `sema_*(3T)`, are contained in the `-lthread` library. By contrast, you link with the `-lposix4` library to get the standard `sem_*(3R)` POSIX 1003.1c semaphore routines described in the section "Semaphores" on page 106.

Link Old With New Carefully

Table 7-2 shows that multithreaded object modules should be linked with old object modules only with great caution.

Table 7-2 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` or `_POSIX_C_SOURCE` flag	Static storage	The traditional `errno`
New object files	*With* the `_REENTRANT` or `_POSIX_C_SOURCE` 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` or `_POSIX_C_SOURCE` 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.

Debugging Multithreaded Programs

Common Oversights

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

- Passing a pointer to the caller's stack as 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)
- Trying to reacquire a lock already held (recursive deadlock)

- 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.
- Mixing UNIX signals with threads—it is better to use the `sigwait`(2) model for handling asynchronous signals
- Using `setjmp(3B)` and `longjmp(3B)`, and then long-jumping away without releasing the mutex locks
- Failing to reevaluate the conditions after returning from a call to `*_cond_wait(3T)` or `*_cond_timedwait(3T)`
- Forgetting that default threads are created `PTHREAD_CREATE_JOINABLE` and must be reclaimed with `pthread_join(3T)`; note, `pthread_exit(3T)` does not free up their storage space
- 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, or using non-default stacks

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.

Tracing and Debugging With the TNF Utilities

Use the TNF utilities (included as part of the Solaris system) to trace, debug, and gather performance analysis information from your applications and libraries. The TNF utilities integrate trace information from the kernel and from multiple user processes and threads, and so are especially useful for multithreaded code.

With the TNF utilities, you can easily trace and debug multithreaded programs. See the TNF utilities chapter in the *Programming Utilities Guide* for detailed information on using `prex(1)`, `tnfdump(1)`, and other TNF utilities.

Using `truss(1)`

See `truss(1)` in the *man Pages(1): User Commands* for information on tracing system calls and signals.

Using `adb(1)`

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 7-3 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*

These commands to set conditional breakpoints are often useful.

Table 7-4 Setting `adb` Breakpoints

[*label***],[***count***]:b [***expression***]**	Breakpoint is hit when *expression* evaluates to zero
foo`,ffff:b <g7-0xabcdef`	Stop at *foo* when g7 = the hex value 0xABCDEF

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 Debugger now support debugging multithreaded programs. For a full overview of `dbx` and Debugger features see the SunSoft Developer Products (formerly SunPro) `dbx(1)` man page and the *Debugging a Program* user's guide.

All the `dbx` options listed below can support multithreaded applications.

Table 7-5 `dbx` Options for MT Programs

`cont at` *line* [*sig signo id*]	Continues execution at *line* with signal *signo*. The *id*, if present, specifies which thread or LWP to continue. The 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.
`thread -continue [ tid ]`	Unsuspends the given thread.
`thread -hide [ tid ]`	*Hide*s the given (or current) thread. It will not appear 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 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.