ST17 - Late Breaking News

Disclaimer

SunSoft, Inc.
2550 Garcia Avenue
Mountain View, California 94043

Practice Safe Threads

There are four “safety” levels to be observed when writing threaded code.

Signal Safe

The function can be safely (i.e. without potential of deadlock) called from within a signal handling function. POSIX.1c provides `pthread_sigmask` to access the per thread signal mask. POSIX.1c also provides a function to synchronously wait for asynchronous signals, `sigwait`.

MT Safe

Function can be safely called from within a threaded program without explicit locking. POSIX.1c provides mutual exclusion locks to guarantee invariants.

Fork Safe

Function or library protects itself against `fork` operations. POSIX.1c provides `pthread_atfork` function to register functions to be called around fork time.

Cancel Safe

Function protects itself against cancellation. POSIX.1c provides the `pthread_setcancelstate` function to disable cancellation during “critical sections.”

Scheduling

The Solaris implementation of POSIX.1c currently only supports the **SCHED_OTHER** domain. For threads created with “system-scope”, **SCHED_OTHER** is equivalent to the Solaris Time-Share class. Threads created with “process-scope” and domain **SCHED_OTHER** use the Solaris thread library scheduling scheme.

The other two POSIX.1c domains (**SCHED_FIFO** and **SCHED_RR**) are not accessible using the POSIX.1c API. However, by using “system-scope” threads, the Solaris Real-Time scheduling class and priocntl(2) functionality approximately equivalent to these two domains can be achieved.

Availability

POSIX.1c/D8 Release 3 (SPARC and x86) are available in Early Access (EA) form for Solaris 2.4 from the Solaris Developer Support Centre (threads@sun.com). POSIX.1c/D10 is part of Solaris 2.5 EA.

Timers and Alarms

In Solaris 2.5, we are announcing the eventual End of Life for two Solaris features. The features are per-LWP POSIX timers and per-thread alarms. Both features are being supplemented in 2.5 with per-process variants.

per-LWP POSIX timers:

Under 2.3 and 2.4, the `timer_create(3r)` function returns a timer object whose timer ID is meaningful only within the calling LWP and whose expiration signals are delivered to that LWP. Such timers are automatically deleted upon termination of the creating LWP. Hence, for multi-threaded programs, only bound threads can use the POSIX timer facility. Even with this restricted use, under 2.3 and 2.4, the use of POSIX timers in multi-threaded applications is unreliable with respect to masking the resulting signals and the delivery of the associated value from the `sigevent` structure.

With 2.5, an application compiled defining the macro **_POSIX_PER_PROCESS_TIMERS** (or with the symbol **_POSIX_C_SOURCE** having a value greater than 199500L) will create per-process timers. The timer IDs of per-process timers are usable from any LWP and the expiration signals are generated for the process rather than directed to a specific LWP. Further, per-process timers are deleted only by `timer_delete(3r)` or at process termination.

Applications compiled prior to 2.5, or without the feature test macros described above, will continue to create per-LWP POSIX timers. In some future release, calls to create per-LWP timers will return per-process timers. The end of life for POSIX per-LWP timers is announced on the `timer_create(3r)` man page.

per-thread alarms:

Under 2.3 and 2.4, a call to `alarm(2)` and `setitimer(2)` is meaningful only within the calling LWP. Such timers are automatically deleted upon termination of the creating LWP. Hence for multi-threaded programs, only bound threads can use this facility. Even with this restricted use, under 2.3/2.4, the use of `alarm(2)` and `setitimer(2)` timers in multi-threaded applications is unreliable with respect to masking the resulting signals from the bound thread which issues these calls. If such masking is not required, then these two system calls work reliably from bound threads.

With 2.5, an application linking with -lpthread (POSIX threads), will get per-process delivery of **SIGALRM** when calling `alarm(2)`, but the effect due to `setitimer(2)` will continue to be per-LWP. The **SIGALRM** generated by `alarm(2)` is generated for the process rather than directed to a specific LWP. Further, the alarm is reset at process termination.

Applications compiled prior to 2.5 or not linked with -lpthread will continue to see a per-LWP delivery of signals generated by `alarm(2)` and/or `setitimer(2)`.

In some future release, calls to `alarm(2)` and/or to `setitimer(2)` with the **ITIMER_REAL** will cause the resulting **SIGALRM** to be sent to the process. The end of life for per-LWP **SIGALRM** generation is announced on the `alarm(2)` and `setitimer(2)` man pages.

Flags other than the **ITIMER_REAL** flag to `setitimer(2)` will continue to result in the generated signal being delivered to the LWP that issued the call, and hence will be usable only from bound threads.