Safe and Unsafe Interfaces 6

This chapter defines MT-safety levels for functions and libraries.

Thread Safety	*page 149*
MT Interface Safety Levels	*page 151*
Async-Signal-Safe Functions	*page 153*
MT Safety Levels for Libraries	*page 153*

Thread Safety

Thread safety is the avoidance of *data races*—situations in which data are set to either correct or incorrect values depending upon the order in which multiple threads access and modify the data.

When no sharing is intended, give each thread a private copy of the data. When sharing is important, provide explicit synchronization to make certain that the program behaves deterministically.

A procedure is *thread safe* when it is logically correct when executed simultaneously by several threads. At a practical level, it is convenient to recognize three levels of safety.

- Unsafe
- Thread safe—Serializable
- Thread safe—MT-safe

An unsafe procedure can be made thread safe and serializable by surrounding it with statements to lock and unlock a mutex. Code Example 6-1 shows a simplified implementation of `fputs()`, initially thread unsafe.

Next is a serializable version of this routine with a single mutex protecting the procedure from concurrent execution problems. Actually, this is stronger synchronization than is usually necessary. When two threads are sending output to different files using `fputs`(), one need not wait for the other—the threads need synchronization only when they are sharing an output file.

The last version is MT-safe. It uses one lock for each file, allowing two threads to print to different files at the same time. So, a routine is MT-safe when it is thread safe and its execution does not negatively affect performance.

Code Example 6-1 Degrees of Thread Safety

```
/* not thread-safe */
fputs(const char *s, FILE *stream) {
    char *p;
    for (p=s; *p; p++)
        putc((int)*p, stream);
}

/* serializable */
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&m);
    for (p=s; *p; p++)
        putc((int)*p, stream);

    mutex_unlock(&m);
}

/* MT-Safe */
mutex_t m[NFILE];
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&m[fileno(stream)]);
    for (p=s; *p; p++)
        putc((int)*p, stream);
    mutex_unlock(&m[fileno(stream)]0;
}
```

MT Interface Safety Levels

The *man Pages(3): Library Routines* use the following categories to describe how well an interface supports threads (these categories are explained more fully in the `Intro(3)` man page).

Safe	This code can be called from a multithreaded application.
Safe with exceptions	See the NOTES sections of these pages for a description of the exceptions.
Unsafe	This interface is not safe to use with multithreaded applications unless the application arranges for only one thread at a time to execute within the library.
MT-Safe	This interface is fully prepared for multithreaded access in that it is both *safe* and it supports some concurrency.
MT-Safe with exceptions	See the NOTES sections of these pages in the *man Pages(3): Library Routines* for a list of the exceptions.
Async-Signal-Safe	This routine can safely be called from a signal handler. A thread that is executing an Async-Signal-Safe routine does not deadlock with itself when it is interrupted by a signal.
Fork1-Safe	This interface releases locks it has held whenever the Solaris `fork1(2)` or the POSIX `fork(2)` is called.

See the table in Appendix B, "MT Safety Levels: Library Interfaces," for the safety levels of interfaces from the *man Pages(3): Library Routines*. Check the man page to be sure of the level.

Some functions have purposely not been made safe for the following reasons.

- Making the interface MT-Safe would have negatively affected the performance of single-threaded applications.
- The interface has an Unsafe interface. For example, a function might return a pointer to a buffer in the stack. You can use reentrant counterparts for some of these functions. The reentrant function name is the original function name with "`_r`" appended.

Caution – There is no way to be certain that a function whose name does not end in "_r" is MT-Safe other than by checking its reference manual page. Use of a function identified as not MT-Safe must be protected by a synchronizing device or by restriction to the initial thread.

Reentrant Functions for Unsafe Interfaces

For most functions with Unsafe interfaces, an MT-Safe version of the routine exists. The name of the new MT-Safe routine is always the name of the old Unsafe routine with "_r" appended. The following "_r" routines are supplied in the Solaris system:

Table 6-1 Reentrant Functions

in alphabetical order	gethostbyaddr_r(3n)	getrpcent_r(3n)
asctime_r(3c)	gethostbyname_r(3n)	getservbyname_r(3n)
ctermid_r(3s)	gethostent_r(3n)	getservbyport_r(3n)
ctime_r(3c)	getlogin_r(3c)	getservent_r(3n)
fgetgrent_r(3c)	getnetbyaddr_r(3n)	getspent_r(3c)
fgetpwent_r(3c)	getnetbyname_r(3n)	getspnam_r(3c)
fgetspent_r(3c)	getnetent_r(3n)	gmtime_r(3c)
gamma_r(3m)	getnetgrent_r(3n)	lgamma_r(3m)
getauclassent_r(3)	getprotobyname_r(3n)	localtime_r(3c)
getauclassnam_r(3)	getprotobynumber_r(3n)	nis_sperror_r(3n)
getauevent_r(3)	getprotoent_r(3n)	rand_r(3c)
getauevnam_r(3)	getpwent_r(3c)	readdir_r(3c)
getauevnum_r(3)	getpwnam_r(3c)	strtok_r(3c)
getgrent_r(3c)	getpwuid_r(3c)	tmpnam_r(3s)
getgrgid_r(3c)	getrpcbyname_r(3n)	ttyname_r(3c)
getgrnam_r(3c)	getrpcbynumber_r(3n)	

Async-Signal-Safe Functions

Functions that can safely be called from signal handlers are *Async-Signal-Safe*. The POSIX standard defines and lists Async-Signal-Safe functions (IEEE Std 1003.1-1990, 3.3.1.3 (3)(f), page 55). In addition to the POSIX Async-Signal-Safe functions, these three functions from the Solaris threads library are also Async-Signal-Safe.

- `sema_post(3T)`
- `thr_sigsetmask(3T)`, similar to `pthread_sigmask(3T)`
- `thr_kill(3T)`, similar to `pthread_kill(3T)`

MT Safety Levels for Libraries

All routines that can potentially be called by a thread from a multithreaded program should be MT-Safe.

This means that two or more activations of a routine must be able to *correctly* execute concurrently. So, every library interface that a multithreaded program uses must be MT-Safe.

Not all libraries are now MT-Safe. The commonly used libraries that are MT-Safe are listed in Table 6-2. Additional libraries will eventually be modified to be MT-Safe.

Table 6-2 Some MT-Safe Libraries

Library	Comments
`lib/libc`	Interfaces that are not safe have thread-safe interfaces of the form `*_r` (often with different semantics)
`lib/libdl_stubs`	To support static switch compiling
`lib/libintl`	Internationalization library
`lib/libm`	MT-Safe only when compiled for the shared library, but not MT-Safe when linked with the archived library
`lib/libmalloc`	Space-efficient memory allocation library; see `malloc(3X)`
`lib/libmapmalloc`	Alternative `mmap(2)`-based memory allocation library; see `mapmalloc(3X)`

Table 6-2 Some MT-Safe Libraries

Library	Comments
`lib/libnsl`	The TLI interface, XDR, RPC clients and servers, `netdir`, `netselect` and `get`*XX*`by`*YY* interfaces are not safe, but have thread-safe interfaces of the form `get`*XX*`by`*YY*`_r`
`lib/libresolv`	Thread-specific `errno` support
`lib/libsocket`	Socket library for making network connections
`lib/libw`	Wide character and wide string functions for supporting multibyte locales
`lib/straddr`	Network name-to-address translation library
`lib/libX11`	X11 Windows library routines
`lib/libC`	C++ runtime shared objects

Unsafe Libraries

Routines in libraries that are not guaranteed to be MT-Safe can safely be called by multithreaded programs only when such calls are single-threaded.