Signals in Multithreaded Programs

By Chary G. Tamirisa

POSIX.1c, the standard for parallel programming, is part of the POSIX series of standards. The behavior of POSIX.1 signals in a multithreaded program has undergone dramatic changes during the standardization process of POSIX.1c. This article compares the traditional POSIX signals model with the new model in a multithreaded environment. The article describes the signal model in AIX 4.1 and provides some guidelines for signal handling in applications. It also describes the DCE signals model with tips on porting DCE pthreads-based applications to AIX 4.1.

AIX 4.1 provides POSIX™ threads support based on draft 7 of the POSIX.1c standard for multithreaded programming. The `pthreads` standard defines several Application Programming Interfaces (APIs) that are provided in the `pthreads` (`libpthreads.a`) library.

The POSIX.1c draft 10 has become the final standard. The basic signal model has not changed from draft 7. This article describes the changes to the traditional signal model in POSIX.1 and provides some important guidelines on how to handle signals in the presence of multiple threads in a process.

The article also addresses the porting issues of applications written to the `pthreads` library supported on AIX Version 3. Although AIX Version 3 did not support kernel threads, the `pthreads` library from DCE provides threads support. Since the DCE `pthreads` library was based on the earlier draft 4 of POSIX.1c, the article explains how to modify existing programs when porting from AIX Version 3 to Version 4. This article has several parts. A discussion of the traditional UNIX (POSIX.1) signal model presents an overview and provides the context necessary to explain the changes made to it in the multithreaded programming model in POSIX.1c. The new signal model is then described with an example. Programming guidelines provide help in writing well-behaved multithreaded programs with respect to signals. Finally, we discuss migration issues related to signals for DCE-based multithreaded programs.

See "Introduction to Multithreaded Programming" (*AIXpert*, November 1994) and "Porting DCE Threads Programs to AIX 4.1" (*AIXpert*, August 1995) for details of the POSIX.1c interfaces.

Traditional (POSIX.1/UNIX) Signal Model

A traditional POSIX.1 process has one signal thread of control. POSIX.1 specifies the signal model.

Signal Handlers

A process can register a signal handler through `sigaction()` to capture signals. There are two types of signals:

- **Synchronous signals** result when an instruction executes in the process. The synchronous signals are `SIGILL`, `SIGTRAP`, `SIGIOT`, `SIGEMT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, and `SIGPIPE`.
- **Asynchronous signals** are delivered to the process irrespective of the current instruction it is executing. Examples include `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGCHLD`, and `SIGUSR1`.

Signal handlers are installed using `sigaction()`, defined as follows in POSIX.1.

Chary G. Tamirisa

```
void catcher(int sig)
 {
        foo();
 }
 foo()
 {
        /* Add an element to a linked list */
  }
 main()
 {
        sigset_t set;
        struct sigaction action;

        action.sa_handler=(void)catcher;
        action.sa_flags = 0;
        /*
        ** You can specify a set of signals
           that need to
        ** be masked while in the catcher
           routine.
        ** Note that the signal that is
           delivered is
        ** already blocked while executing
           the catcher()
         ** routine.
        */
        sigemptyset( &action.sa_mask);
        sigaction(SIGINT, &action, NULL);

        sigemptyset(&set);
        sigaddset(SIGINT, &set, NULL);

        sigprocmask(SIG_BLOCK, &set, NULL);
        /* Signal Safe code */
        foo();          /* Add an element
          to a linked
            list */
        sigprocmask(SIG_UNBLOCK, &set, NULL);
 }
```

Figure 1. Code sequence for a single-threaded process

```
#include <signal.h>
int sigaction(int signal,
   struct sigaction *newaction,
   struct sigaction
*oldaction);
```

Signal Masks

A signal can be masked by specifying a block signal mask through the `sigprocmask()` interface. Typically, a process in a critical section that must not be interrupted by a signal handler could block the required signals until the critical section is done. Signal handlers are registered via `sigaction()` by specifying a mask that must be installed when the signal handler executes. Since a signal handler can prevent signals from interrupting, it can guarantee signal safety.

Signal masks are installed by using `sigprocmask()`, defined as follows in POSIX.1:

```
#include <signal.h>
int sigprocmask( int how,
   sigset_t *newset, sigset_t *oldset);
```

Signal Safety in a POSIX.1 Process

For a program to provide signal safety in a single-threaded POSIX.1 process, it could invoke `sigprocmask()` to block the signal when a chosen code sequence is executed, then later unblock the signal. For example, the code sequence in Figure 1 will work in a single-threaded process.

When a signal handler such as `catcher()` is invoked, the operating system installs a signal mask that blocks the received signal from further occurrence. The correct mask is restored when the signal handler returns.

To protect a signal handler from signals other than the currently handled signal, a program can specify additional signals to be masked (`sa_mask`) when the signal handler (`catcher`) executes. This enables a signal handler to safeguard itself against further signals.

POSIX.1c Signal Model

Since a multithreaded program can have more than one thread, several questions arise:

- What happens if more than one thread calls `sigaction()` for the same signal?
- What is the behavior when a thread blocks a signal? Does it block the signal for the whole process, or does it block the signal for the current thread?
- How do we ensure signal safety?

Understanding the new programming model can help answer these questions.

New Programming Model for Signals

The POSIX.1 signal is modified as follows:

- All signal handlers are per-process. Use `sigaction()` to install a process-wide signal handler for all signals.
- All signal masks are per-thread. Use the `sigthreadmask()` function to install a per-thread signal mask. The `sigthreadmask()` is identical to `sigprocmask()` except it now works on a per-thread basis. Use of `sigprocmask()` is not defined in multithreaded programs.

Signal Handlers

Because the signals handlers are installed on a per-process basis, there is one signal handler for a given signal. If more than one thread installs a signal handler for the same signal, the last one installed is used. This means if predictable behavior is needed in a program, signal handlers must be used carefully.

It is good practice not to install signal handlers in libraries; however, if signal handlers are needed, the library must restore the previous handler before returning to the application code. When multiple libraries are invoked from multiple threads, there is always a possibility for one library to overwrite the handler installed by another library, leading to a non-deterministic behavior if the signal handler is supposed to perform different actions.

Signal Masks

Signal masks are per-thread. In fact, POSIX.1c draft 7 defines a new function— `sigthreadmask()`—to install the signal mask on a per-thread basis. This means that a thread cannot modify the signal mask of another existing thread. If a thread installs a signal handler and masks it later when it is executing a function (such as `foo()` in Figure 1), it is possible for another thread, which has not masked it, to receive the signal.

The signal handler can be invoked even though the current thread has masked the signal. Therefore, it breaks the assumption under which the code ran successfully in a single-threaded process. To fix this problem, the signal must be blocked in all the threads. The only way to do this in AIX 4.1 is to rely on the inheritance property for signal masks when threads are created. If the signal of interest is blocked in the main or the initial thread, any thread created thereafter will inherit the signal mask. This is the only way to install a signal mask in all threads in POSIX.1c.

Waiting for Signals (sigwait())

The new API `sigwait()` allows the calling thread to be the focal point for receiving asynchronous signals. This API enables the action on asynchronous signals to be deferred, allowing other threads to continue their work in the presence of asynchronous signals.

If a process wants to receive asynchronous signals, it should invoke `sigwait()`, specifying a signal mask that indicates which signals to wait for. Before invoking `sigwait()`, the thread must block the signals in the mask. This dedicated thread waits for the signals specified. When any signals are received, the `sigwait()` call returns with the signal number. The following sections provide more details.

The New Signal Model

The new POSIX.1c signal model has several important features that allow proper signal handling in the multithreaded environment. The signal handler and signal mask functions are easy to understand because they are extensions of an older POSIX.1 model into the multithreaded environment. However, `sigwait()` is a new API to most programmers. The following sections cover the proper use of the signal action and the signal mask functions.

The signal handler and signal mask functions are extensions of an older POSIX.1 model into the multithreaded environment.

Sigwait()

The `sigwait()` API is specified as follows in POSIX.1c draft 7:

```
#include <signal.h>
    int sigwait(sigset_t *set, int
    *signal);
```

The first argument set specifies the signals to be waited—an input argument to `sigwait()`. The second argument is the signal that is received by the `sigwait()` call—an output argument. The `sigwait()` returns zero on successful return, or returns an `errno` value on error.

It may be surprising to note that `sigwait()` must be called with the signals that it is waiting for blocked! How will the signals be delivered if the signals are masked from the beginning? It is up to `sigwait()` to do whatever is necessary to receive signals delivered to the process or to the `sigwaiter` thread. Some implementations install a signal handler when `sigwait()` is called and unblock signals that are specified. When one of the signals it is waiting for is delivered, it reblocks the signals and returns with the signal. It is important to remember that there is an invariant that is maintained: the signal mask of the calling thread is the same before and after `sigwait()` returns.

How to Use sigwait()

To use `sigwait()`, create a dedicated thread in which `sigwait()` is invoked to capture signals of interest. The main issue with `sigwait()` is that signals specified in the set argument must be blocked—at least in the calling thread. Two interesting scenarios describe the problems related to blocking.

Scenario 1: Using `sigwait()` with multiple threads that can receive signals. It is possible to have multiple threads that are eligible to receive a `sigwaited` signal if they do not block the signals being waited for in `sigwait()`. The AIX 4 library ensures that the signals received are directed to the `sigwaiter` thread, but there is a side-effect of signal delivery. A thread in a system call that is interrupted will return with errno global variable set to EINTR. The application must handle this return properly. If the application does not want the system calls to be interrupted this way, it must block the signals in the threads that issue system calls.

Scenario 2: `Sigwait()` in a loop. Imagine that you want to receive asynchronous signals in a thread, process them, then return to `sigwait()` in a loop. During the time between the return of `sigwait()` and when it is invoked again, the dedicated thread is not really in `sigwait()`. If a signal of interest occurs during this time, the kernel will deliver the signal to any other thread that has the signal unmasked (assuming the `sigwaiter` thread has blocked the `sigwaited` signals). If there are threads that have not masked the signal, the signal delivery can cause default action to take place, which might include termination of the process.

It is up to sigwait() to do whatever is necessary to receive signals delivered to the process or to the sigwaiter thread.

To safeguard against the two scenarios, block the `sigwaited` signals in all the threads.

POSIX.1c does not provide a global signal mask, yet a program needs to block signals in all the threads. To accomplish a global mask, the first action is to block the signals in the main or initial thread before creating any threads. Using the property of inheritance, the block signal mask will be propagated to all the threads in the process. If no thread unblocks the signals in the mask, a process has been created in which the signal mask is set up in all of the threads. A `sigwaiter` thread can then receive signals and process them as needed.

Guidelines for Using sigwait()

There are certain rules to follow for obtaining predictable behavior from `sigwait()`.

- For handling asynchronous signals in a multithreaded program, the recommended method is to convert asynchronous signal actions to synchronous ones by dedicating a thread to do `sigwait()` for them. In the `sigwaiter` thread, once the signal is received, it can be processed.
- The signals of interest must be blocked in the main or initial thread of the process.
- If a new process is created via `fork()`, ensure that the required signals are blocked in the initial thread of the child process if asynchronous signal handling is required. Use `sigwait()` in a dedicated thread in the child process.
- If signal handler-to-thread synchronization is required, synchronize in the `sigwaiter` thread after `sigwait()` returns. Signal-to-thread synchronization is mapped to thread-to-`sigwaiter` thread synchronization.
- Do not use any of the `pthreads` APIs from signal handlers installed through `sigaction()` or `signal()`. It is not safe to do locking from signal handlers installed this way. This alone is a good reason why multithreaded programs must not use signal handlers to modify or set global data that threads also modify.
- If your program needs to ensure atomicity with signals, use `sigwait()`. Capture asynchronous signals, then try to acquire locks as needed within a `sigwaiter` thread. It is not possible to wait for synchronous signals, and blocking (or masking) synchronous signals is probably unwise.
- In general, it is not a good idea to have a `sigwaiter` thread for a signal and to install a signal handler through `sigaction()` or `signal()` for the same signal. POSIX.1c leaves the behavior undefined in this case, and programs depending on a certain behavior in this case are not portable.
- If you replace an asynchronous handler, ensure that there is no conflict with any handler installed previously for the same signal.

Signal Safety in Multithreaded Processes

In multithreaded processes, the mask specified in `sa_mask` for the `sigaction()` call will block the signal only for the current thread. That is because the mask itself can be installed on a per-thread basis. If another signal arrives, the signal handler can be invoked in the context of another thread that has not masked it. For this reason, signal handler-to-signal handler safety cannot be guaranteed by just masking one or more signals. Because of this characteristic, it is difficult to protect signal handlers from other signals.

Thread-to-signal handler safety also cannot be guaranteed. Even if the current thread masks a

signal while entering a critical section that a signal handler also tries to use, it is possible for another thread that does not block the signal to receive the signal and run the signal handler in its context. You might think that this critical section can be enforced by trying to acquire a lock from the signal handler. However, this is not supported in a threaded environment. Signal handlers cannot invoke any POSIX threads locking functions, such as `pthread_mutex_lock()` and `pthread_mutex_trylock()`, because these functions are not required to be signal safe. In fact, they are usually not signal safe in most implementations.

The pthread Locks and Signal Handlers

The `pthread_mutex_lock()` is not signal safe, because when a call to this function results in blocking, the calling thread is the one that gets blocked. Imagine a thread that has locked a mutex and has a signal unmasked. Let the signal be delivered while the thread holds the lock. If the signal handler also tries to acquire the same lock, the call has to be blocked. Since the only way to get blocked is to block the calling thread, the current thread gets into a blocked state waiting for the mutex to be released. However, because the mutex is already locked by the current thread, it can never be released, resulting in a deadlock. This type of deadlock is known as self-deadlock.

It is clear that it is not possible to safeguard a critical section from asynchronous signals. Masking signals in the current thread alone is not sufficient in a multithreaded program. You will have to mask signals in all the threads, then ensure that a signal is selectively unmasked in one specific thread. Thus, a signal can invoke critical sections. In most applications, this might be difficult (if not impossible) to implement, since a thread does not control the signal mask of another thread. Solving this problem requires a method of directing asynchronous signals to a dedicated thread that waits for them. When a signal is received, it can try to acquire a lock and then invoke the critical section. POSIX.1c provides the `sigwait()` API for this purpose.

DCE Signals

The services provided in DCE, such as the DCE Remote Procedure Calls (RPCs), use POSIX thread interfaces extensively. Because DCE is based on draft 4 of POSIX.1c, a compatibility library provides the necessary function (wrappers to map the draft 4 APIs to the draft 7 APIs, except support) for DCE and its applications. The compatibility library does not support the draft 4 signal behavior in AIX 4.1.

DCE's signal model, based on POSIX draft 4, specified the following signal behavior for the AIX DCE 1.1:

- Per-process signal handlers for asynchronous signals
- Per-thread signal handlers for synchronous signals
- Per-process signal mask (`sigprocmask()`)
- Wait for asynchronous signals (`int sigwait(sigset_t *set)`—based on draft 4)

It is possible to implement signal handlers, but the per-process signal mask is difficult to implement without kernel support. Even if draft 4 semantics can be implemented in the compatibility library, conflicts in the signal model will occur when libraries written to draft 7 and draft 4 are used in the same process. To eliminate these problems, we have decided to base DCE signal handling on the draft 7 signal model: per-process signal handlers and per-thread signal masks.

All DCE application developers should evaluate the signal needs and make suitable modifications to applications based on the above guidelines for POSIX.1c draft 7. This generally involves blocking the `sigwaited` signals in the main or initial thread, and not much more. If applications used the global mask to protect critical sections, the same functionality can be achieved with a dedicated `sigwaiter` thread and by acquiring a mutex after the call to `sigwait()` returns with the signal.

The services provided in DCE , such as Remote Procedure Calls, use POSIX thread interfaces extensively.

Chary Tamirisa, IBM Corporation, LAN Systems Division, 11400 Burnet Road, Austin, TX 78758. Internet: chary@austin.ibm.com. Since 1993, Mr. Tamirisa has been the team leader for the threads package on AIX and OS/2 DCE. He has also worked in the fields of communication protocols, system software, and National Language Support. Mr. Tamirisa has an MS in Computer Science from McGill University and a BTech in Electrical Engineering from the Indian Institute of Technology in Madras, India.