AIX

Introduction to Multithreaded Programming

By Chary G. Tamirisa

AIX 4.1 supports multithreaded programming based on the POSIX 1003.4a Draft 7 specification. The POSIX.4a specification consists of a set of Application Programming Interfaces (APIs) that can be used to parallelize an application. AIX 4.1 provides these APIs in the libpthreads.a (pthreads) library. This article introduces the features of the threads programming model to help developers write applications using this library.

POSIX.4a is an emerging standard to support parallel programming. POSIX.4a is an extension of the base POSIX.1 and POSIX.4 standards. POSIX.4a preserves the programming model of POSIX.1 by making appropriate extensions from the process model to the threads model. It extends the POSIX.4 real-time standard to the threads environment.

The latest draft is POSIX.4a Draft 9, which is expected to soon become a standard with minimal changes.

POSIX.4 and POSIX.4a have been renamed POSIX.1b and POSIX.1c, respectively, which shows that they have a strong affinity to the base POSIX.1 standard. In this article, we use POSIX.4a to refer to POSIX.1c.

Chary G. Tamirisa

POSIX Threads Programming Model

A *thread* is a sequence of instructions that can be scheduled, similar in concept to a process. A thread has an advantage over a process in that it is designed to be very lightweight. Compared to the process, it is inexpensive to create, terminate, schedule a thread, or to synchronize with it.

POSIX.4a modifies the definition of a *process* in POSIX.1 from "an address space with a single thread of control" to "an address space with one or more threads of control." All threads in a process share the following characteristics:

- Address space, shared storage
- The process ID, parent process ID, process group ID
- Session membership
- Real, effective, and saved-set user ID
- Real, effective, and saved-set group ID
- Supplementary group IDs
- Current working directory, root directory
- File-mode creation mask
- File descriptor table
- Signal handlers
- Per-process timers

Each thread has the following thread-specific information:

- Unique thread identifier
- Scheduling policy and priority
- Per-thread `errno`
- Thread-specific key/value bindings
- Resources required to support a flow of control (such as a stack)
- Per-thread cancellation handlers
- Per-thread signal masks

The POSIX.4a model allows for very lightweight threads. It does not require per-thread file tables, timers, or signal handlers—all these are

per process. If a thread changes these per-process entities, all threads will see the changes. A general programming guideline is to create process-wide signal handlers, mutexes, condition variables, context keys, and so on, in the main or initial thread, which is the first thread created when the process begins. The `main()` function of a program is associated with the main thread.

Synchronization

The threads model defines two mechanisms for synchronization: mutexes and condition variables. Mutexes can be used to protect access to shared resources. Together, condition variables and mutexes can be used to synchronize thread execution, such as when long delays are expected in accessing shared resources, or synchronized thread execution is desired.

Thread-Specific Context

The threads model provides a per-thread context for proper creation of thread-specific data. It also specifies a method for proper cleanup of thread-specific data when a thread exits.

Thread Cancellation

The threads model specifies thread-cancel states and types that control thread cancellation. It also defines a mechanism to clean up the thread state after cancellation.

Once-Only Initialization

The mutexes, condition variables, and thread-specific context keys must be initialized before use. Each time they are initialized, a new object results. Consider the synchronization of threads using mutexes. If two threads initialize the same mutex independently, mutual exclusion is not guaranteed because they will be using different mutexes. To do this correctly, the threads model provides a mechanism called *once-only initialization*, which allows an object to be initialized only once for all threads in a process.

Thread Scheduling

Thread scheduling is controlled by specifying contention scopes (process or system-wide) for threads and various scheduling policies on threads.

Relationship Between POSIX.4a and POSIX.1

POSIX.4a preserves and extends the POSIX.1 and POSIX.4 API to the threads environment. Operations such as `read()` and `write()` work on a per-thread basis. The signal model is as follows:

- **Per-thread signal mask**: Each thread has it own signal mask, which is inherited on thread creation.
- **Per-process signal handlers**: All signal handlers are installed on a per-process basis.
- **Single delivery of signals**: A signal is delivered to only one thread and delivered only once.

This model also provides the mechanism to wait for asynchronous signals, such as `SIGINT` and `SIGQUIT`, through the `sigwait()` interface.

Typically, the POSIX.1 functions return the value of -1 and set the global `errno` to indicate the specific error condition. In a multithreaded environment, a per-thread error number is needed. The POSIX.4a interfaces in Draft 7 are defined to return the error number as the function return value instead of setting the global `errno` variable.

The threads model defines two mechanisms for synchronization: mutexes and condition variables.

Overview of pthread Interfaces

The pthread interfaces, defined in the `<pthread.h>` header file, are discussed in the following sections.

Objects and Attributes

POSIX.4a defines mechanisms for concurrent execution of several threads in a process. Basically, it specifies certain objects and attributes that govern the objects. The objects are threads, mutexes, condition variables, per-thread context keys, and once-only initialization. The attributes are thread, mutex, and condition variable attributes. Several APIs are defined to handle operations on these objects and attributes.

To create a thread with default attributes, use `NULL` as the attribute value. To create a thread with non-default attributes, first initialize an attribute and specify the properties of the thread, then use this attribute to create the thread. Figure 1 lists the functions to create objects and thread attributes.

Thread API

The following operations can be performed on threads:

- Join with a thread (`pthread_join()`)
- Cancel the execution in a controlled manner (`pthread_cancel()`)

Object	Creation Functions
Thread	`pthread_create(pthread_t *thd, const pthread_attr_t *attr, void *(*func)(void*), void *arg)`
Mutex	`pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutex_attr_t *mattr)`
Condition variable	`pthread_cond_init(pthread_cond_t *cond, pthread_cond_attr_t *condattr)`
Context key	`pthread_key_create(pthread_key_t *key, void (*destroy)(void*) )`
Attributes	**Creation Functions**
Thread	`pthread_attr_init(pthread_attr_t *attr)`
Mutex	`pthread_mutexattr_init(pthread_mutexattr_t *mutexattr)`
Condition variable	`pthread_condattr_init(pthread_condattr_t *condattr)`

Figure 1. Functions to create objects and thread attributes

- Compare two thread handles (`pthread_t *`) to see if two threads are the same (`pthread_equal()`)
- Detach a thread after creation (`pthread_detach()`)
- Send a signal to kill a specific thread (`pthread_kill()`)

Figure 2 shows the thread attributes that can be specified with their APIs.

Mutex API

The following operations can be performed on a mutex:

- Create (`pthread_mutex_init()`)
- Destroy (`pthread_mutex_destroy()`)
- Lock (`pthread_mutex_lock()`)
- Unlock (`pthread_mutex_unlock()`)
- Try to lock (`pthread_mutex_trylock()`)

The default mutex attribute—a non-recursive mutex—is currently the only type supported. Locking this type of mutex more than once by the same thread causes the error `EDEADLK` to be returned in the calling thread.

Condition Variable API

The following operations can be performed on a condition variable:

- Create (`pthread_cond_init()`)
- Destroy (`pthread_cond_destroy()`)
- Wait (`pthread_cond_wait()`)
- Wait with timeout (`pthread_cond_timedwait()`)
- Signal (`pthread_cond_signal()`)

Always use the default value of `NULL` as the condition variable value.

Thread-Specific Context

Context keys do not have attributes. The following operations can be performed on a context key:

- Create a context key (`pthread_key_create()`)
- Get the value associated with a key (`pthread_getspecific()`)
- Set the value associated with a key (`pthread_setspecific()`)

Thread Cancellation

Figure 3 shows thread-cancel operations and their APIs.

The thread model allows you to establish thread-cancel cleanup handlers and to remove them using the following APIs:

- Push a cleanup handler (`pthread_cleanup_push()`)
- Pop a cleanup handler (`pthread_cleanup_pop()`)

POSIX.4a API

The following sections provide a detailed discussion of the POSIX.4a interfaces.

Thread Creation

To write a multithreaded application, the programmer must identify the individual flows of control within the application. This should help determine the number of threads to be created. In the POSIX.1 model, a process is initiated through the `fork()` call, as shown in Figure 4.

Thread Attribute	API
Get detach state	pthread_attr_getdetachstate()
Set detach state	pthread_attr_setdetachstate()
Get stack size	pthread_attr_getstacksize()
Set stack size	pthread_attr_setstacksize()
Get stack address	pthread_attr_getstackaddr()
Set stack address	pthread_attr_setstackaddr()
Get scheduling policy	pthread_attr_getschedpolicy()
Set scheduling policy	pthread_attr_setschedpolicy()
Get scheduling parameters	pthread_attr_getschedparam()
Set scheduling parameters	pthread_attr_setschedparam()
Get inheritance scheduling policy	pthread_attr_getinheritsched()
Set inheritance scheduling policy	pthread_attr_setinheritsched()
Get scheduling scope	pthread_attr_getscope()
Set scheduling scope	pthread_attr_setscope()

Figure 2. Thread attributes and their APIs

Thread-Cancel Operation	API
Query the cancel state of a thread	pthread_getcancelstate()
Enable or disable the cancel state of a thread	pthread_setcancelstate()
Query the current cancel type of thread	pthread_getcanceltype()
Set the type of cancel that is allowed	pthread_setcanceltype()
Cancel a thread	pthread_cancel()

Figure 3. Thread-cancel operations and APIs

The example in Figure 4 can be rewritten using the threads API as shown in Figure 5.

A thread is created using `pthread_create()` as follows:

```
int pthread_create(pthread_t *thd,
pthread_attr_t *attr,
(void *)(*)(void*), void *status )
```

The first argument is a pointer to an opaque handle called `thd`. The second argument is a pointer to the thread attribute. The `NULL` value used in this program fragment specifies the default attribute (this is typical). The third argument is the function that specifies the thread. The programmer must specify what constitutes a thread of activity in a program. The fourth argument is an optional argument to be passed to the called function. The thread is started like a call to a function. When the thread is completed, a return value can be specified to indicate its exit status, which can be obtained by another thread that joins with it by a call to `pthread_join()`.

```
main()
{
     if(fork() == 0){
             /* Child */
     }else{
             /* Parent Process */
     }
```

Figure 4. Using fork() to create a new process

Thread Termination

A thread can call `pthread_exit()` or simply `return()` after completion. An optional argument for `pthread_exit()` or `return()` can be specified to indicate the status of the thread at exit time.

```
#include <pthread.h>
/* Argt_t is an arbitrary structure to be defined by
   the program */
callFunc(Arg_t *arg)
{ /* Thread (Child) code here ..*/
}
main()
{
       pthread_t callThd;
       int ret;
       /* Main thread */
       /* Create another thread */
       ret=pthread_create(&callThd, NULL, callFunc,
           &arg);
       if(ret){
              /* Error in thread creation */
       }
```

Figure 5. Creating a thread

```
#include <pthread.h>
func()
{
       pthread_exit( 1);
}
main()
{
       pthread_t thd;
       int s;
       pthread_attr_t attr;
       pthread_attr_init(&attr);
       pthread_attr_setdetachstate(&attr,
          PTHREAD_CREATE_UNDETACHED);
       pthread_create(&thd, &attr, func, 0);
       pthread_join( thd, &s);
       printf("status=%d\n", s);
}
```

Figure 6. How to obtain thread exit status

Join With a Thread

Sometimes one thread must wait for another thread to complete. For example, the main thread can create several threads, and it may wait for all the threads to complete before it exits. This can be done through a call to `pthread_join ((pthread_t) thread, void **status)`. The first argument is the `pthread_t` handle; the second argument is an optional status that can be obtained from the joined thread.

The example in Figure 6 shows how to return a thread exit status and how to join threads.

AIX 4.1 allows a maximum of 512 simultaneous user-level threads per process. When a thread exits, the application must allow the threads library to release the resources associated with it in a `pthreads` statement. These resources are typically the thread stack and other memory associated with a thread. To accomplish this, the application must do one of the following: call `pthread_join()` on the thread; or create the thread with the default thread attribute of `PTHREAD_CREATE_DETACHED`, which can be done by specifying a `NULL` value in the second argument for the `pthread_create()`. In AIX 4.1, threads created with the `NULL` thread attribute value are detached (or freed) on thread exit or return.

Threads created with the default thread attribute value of `NULL` cannot be joined. To join with a thread `pthread_join()`, create a thread with the attribute value of `PTHREAD_CREATE_UNDETACHED`, as shown in Figure 6. An attempt to join with a thread created with the detached attribute set will return an error of `ESRCH`.

Synchronization

POSIX.4a defines two synchronization mechanisms: mutexes for critical regions and condition variables for synchronizing execution of threads that may involve long delays.

Use of Mutexes

Developers must ensure data consistency when several threads execute concurrently. Concurrent modification of shared data is properly synchronized using the synchronization API provided in the threads library.

The example in Figure 7 shows the mutex lock and unlock operations to protect shared data. A mutex must be initialized before use. In Figure 7, the global integer variable `count` is protected by using a mutex lock.

When the program in Figure 7 is run on AIX 4.1, the output shown in Figure 8 is created.

By properly locking modifications to the global data (`int count`), the count is incremented correctly and the threads do not overwrite each other's modifications.

These mutex locks should be used to obtain mutual exclusion when it is known that the locks will be available in a short time, and that these locks are not held for long durations.

Use of Condition Variables

Sometimes it is necessary to wait for a condition to occur before a thread can proceed. In such cases, a process-wide boolean is used to indicate

```
#include <pthread.h>

int count = 0;    /* Global count */
int thdcount[2];  /* Count of how many times the loop is done in each thread*/
pthread_mutex_t m;
int func1(int i)
{
    int n;
    pthread_mutex_lock(&m);
    while(count <2) {
        n = count;
        sleep(1);
        thdcount[i]++;
        printf("thread id %d: replacing count: count old value=%d", i,n);
        n = n+1;
        count = n;
        printf(" and new value=%d\n", count );
    }
    pthread_mutex_unlock(&m);
    printf("Thread Function %d : count = %d \n",i, count);
    return(1);
}
main()
{
    int i;
    pthread_t t[2];
    int status;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_UNDETACHED);
    pthread_mutex_init(&m, NULL);
    for(i=0;i<2;i++)
            thdcount[i]=0;
    for(i=0;i<2;i++)
            pthread_create(&t[i], &attr, func1, i);
    for(i=0;i<2;i++){
            pthread_join(t[i], &status);
    }
    for(i=0;i<2;i++){
            printf("Thread %d thdcount=%d\n", i, thdcount[i]);
    }
}
```

Figure 7. Use of mutex

```
thread id 0: replacing count: count old value=0 and new value=1
thread id 0: replacing count: count old value=1 and new value=2
Thread Function 0 : count = 2
Thread Function 1 : count = 2
Thread 0 thdcount=2
Thread 1 thdcount=0
```

Figure 8. Resulting output from using mutex

Programming Notes on Mutexes

1. Define mutexes as global variables since they are generally required to be visible to all the threads that contend.
2. Initialize the mutex by calling `pthread_mutex_init()`.
3. Initialize a mutex only one time.
4. After deciding that a mutex is no longer used or needed, use `pthread_mutex_destroy()` to release the resources associated with it.
5. For dynamic memory allocation of mutexes, always use `pthread_mutex_destroy()` to release the resources associated with the mutex, then call `free()` to free up the memory.
6. Note that the second argument to `pthread_mutex_init()` specifies the attributes of the mutex; a `NULL` value indicates the default mutex attribute. In AIX 4.1, the default mutex is a non-recursive mutex, meaning that it can be locked only one time. If the owner of the mutex (the thread that locked it successfully) tries to lock it again, the error `EDEADLK` is returned.

if the condition is satisfied so that a thread can proceed to execute. To protect modifications to the boolean, locks must be used. However, if the boolean condition indicates that the thread must wait, it must release the lock so that another thread can modify the boolean. To do this properly, condition variable APIs are provided to perform the wait operations—`pthread_cond_wait()` and `pthread_cond_timedwait()`—and the signal (or wakeup) operations—`pthread_cond_signal()` and `pthread_cond_broadcast()`.

Typically, the sequence of operations is as follows: the thread locks the mutex and tests if the flag indicates that the resource is available. If the resource is not available, it invokes `pthread_cond_wait()` or `pthread_cond_timedwait()` with the mutex and condition variable as arguments. The `pthread_cond_wait()` or `pthread_cond_timedwait()` atomically releases the mutex and blocks the calling thread. When the thread is awakened by another thread through a call to `pthread_cond_signal()` or `pthread_cond_broadcast()`, the call to `pthread_cond_wait()` or `pthread_cond_timedwait()` returns with the mutex locked. These functions may return spuriously, so re-evaluate the boolean when these functions return.

Figure 9 shows the use of condition variables and mutexes.

The results of the program in Figure 9 are as follows:

```
Thread Function 0 : count = 2
Thread Function 1 : count **2 = 4
Thread 0 thdcount=2
Thread 1 thdcount=0
```

Timed Wait Operation on Condition Variables

To recover from a potentially long wait on a condition variable in `pthread_cond_wait()`, another API is provided to perform timed waits:

```
int pthread_cond_timedwait
      (pthread_cond_t *cv,
      pthread_mutex_t *mutex,
      const struct timespec *absolutetime)
```

The code fragment in Figure 10 shows how to use `pthread_cond_timedwait()`.

Once-Only Initialization

To ensure that the mutexes and condition variables are initialized only once, call the code that initializes them only once. In the examples in Figure 11, the main thread initializes the mutexes—this is guaranteed to be done once. However, if the other threads perform the initialization, use the pthreads API `pthread_once(pthread_once_t *once, void((init_routine)(void))` to guarantee the once-only semantics.

The example in Figure 11 shows how to use `pthread_once()`.

Thread-Specific Data

If a thread wants to create data that is not globally shared across all other threads, it must create the data on its stack or in the local variables. Keeping the data on the stack is not useful if the data must be available across procedure calls, such as with library functions. To solve this problem, special APIs, shown in Figure 12, are provided to create and destroy context on a per-thread basis.

Several non-reentrant functions that keep static data in the standard C library can become reentrant by providing thread-specific data, therefore eliminating the static data. One example is the `strtok()` function that keeps the search pointer in a static location. By making this a thread-specific value, `strtok()` can be made reentrant.

```
#include <pthread.h>
int count = 0;    /* Global count */
int thdcount[2];  /* Count of how many times the loop is done in each thread*/
pthread_mutex_t m;
pthread_mutex_t cv;
int ready=0;
int func1(int i)
{
       int n;
       pthread_mutex_lock(&m);
       ready=0;
       while(count <2) {
              n = count;
              sleep(1);
              thdcount[i]++;
              n = n+1;
              count = n;
       }
       /* Wake up the waiting thread */
       ready=1;
       pthread_cond_signal(&cv);
       pthread_mutex_unlock(&m);
       printf("Thread Function %d : count = %d \n",i, count);
       return(1);
}
int func2(int i)
{
       int n;
       pthread_mutex_lock(&m);
       while(!ready)
              pthread_cond_wait(&cv, &m);
       n = count * count;
       pthread_mutex_unlock(&m);
       printf("Thread Function %d : Count **2  = %d \n",i, n);
       return(1);
}
main()
{
       int i;
       pthread_t t[2];
       int status;
       pthread_attr_t attr;
       pthread_attr_init(&attr);
       pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_UNDETACHED);
       pthread_mutex_init(&m, NULL);
       pthread_cond_init(&cv, NULL);
       for(i=0;i<2;i++)
              thdcount[i]=0;
       i=0;
       pthread_create(&t[i], &attr, func1, i);
       i=1;
       pthread_create(&t[i], &attr, func2, i);
       for(i=0;i<2;i++){
              pthread_join(t[i], &status);
       }
       for(i=0;i<2;i++){
              printf("Thread %d thdcount=%d\n", i, thdcount[i]);
       }
}
```

Figure 9. Use of condition variables and mutexes

```
/* func() is a thread (main not shown ) */
/* On timeout, it returns ETIMEDOUT */
/* On success, it returns zero */
#include <pthread.h>
int func(struct timespec *delta)
{
struct timespec abstime;
getclock(TIMEOFDAY, &abstime);

/* Calculate the absolute time for timeout */
abstime.tv_sec += delta->tv_sec;
abstime.tv_nsec += delta->tv_nsec;
pthread_mutex_lock(&mutex);
while(!ready){
        ret  = pthread_cond_timedwait(&cv, &mutex, &abstime);
        if(ret == ETIMEDOUT){
                pthread_mutex_unlock(&mutex);
                return(ETIMEDOUT);
        }
/*Do other error handling*/\
}
/* Do the processing here */
pthread_mutex_unlock(&mutex);
return(0);
}
```

Figure 10. Use of pthread_cond_timedwait()

```
#include <pthread.h>
pthread_once_t mutex_once = PTHREAD_ONCE_INIT;
pthread_mutex_t m;
void init_routine(void){
        pthread_mutex_init(&m, NULL);
}

/* The thread function */
func()
{
        pthread_once( &mutex_once, init_routine );
        ...
}
main()
{
        pthread_t thd;
        pthread_create(&thd, NULL, func, NULL);
        ...
}
```

Figure 11. Example of the use of pthread_once()

```
int pthread_key_create(pthread_key_t *key, void (*destructor(void *)))
void *pthread_getspecific(pthread_key_t key )
int pthread_setspecific(pthread_key_t key, const void *value)
```

Figure 12. Special APIs to create and destroy context

The C library, however, provides a separate reentrant version `strtok_r()`.

Thread Cancellation

It is sometimes necessary to cancel a thread because its operation is no longer needed, or it is in a wait (such as blocked system calls) that may never happen. When a thread is terminated gracefully by another thread, ensure that the cancelled thread releases all the resources it holds (specifically mutexes). The pthreads library defines the following specific APIs to cancel a thread and cleanup on thread cancellation:

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
void pthread_cleanup_push( (void
   (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

Figure 13 shows how to use the thread-cancel API.

Cancel States

The key concept to thread cancellation is to control when a thread can be cancelled. There are two cancel states—`PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`—and two cancel types—`PTHREAD_CANCEL_DEFERRED` and `PTHREAD_CANCEL_ASYNCHRONOUS`.

These control what happens when a cancel is generated for a thread. If the state is disabled, it is left pending and is delivered when the state is enabled.

If the state is enabled and the cancel type is set to `PTHREAD_CANCEL_ASYNCHRONOUS`, the cancel is delivered immediately to the thread.

If the state is enabled and the cancel type is set to `PTHREAD_CANCEL_DEFERRED`, the cancel is delivered if the thread is at a cancel point; otherwise, it is left pending until it reaches a cancel point.

Cancel Point

The following discussion applies to the thread if the cancel type is set to the default `PTHREAD_CANCEL_DEFERRED`.

A thread can be cancelled while it is executing the following functions defined in POSIX.4a:

`pthread_cond_wait()` and
`pthread_cond_timedwait()`
`pthread_join()`
`pthread_testcancel()`
`sigwait()`

Programming Notes on Condition Variables

1. Define condition variables as global variables since they are usually required to be visible to all the threads that use them.
2. Initialize the condition variables by calling `pthread_cond_init()` before use.
3. Initialize a condition variable only once.
4. Associate a boolean with a condition variable (as shown in Figure 10).
5. The correct use of condition variable is as follows:

```
pthread_mutex_lock(&m);
      while(!ready)
           pthread_cond_wait(&cv, &m);
pthread_mutex_unlock(&m);
```

   `Ready` is the boolean.
   Put the condition wait in a `while()` loop.
6. Whenever `pthread_cond_wait()` or `pthread_cond_timedwait()` returns, the mutex specified is locked. Unlock it before exiting the thread (or returning from the thread).
7. When `pthread_cond_timedwait()` returns on timeout, the mutex is locked. Unlock the mutex if you do not retry.
8. If you allocate memory dynamically for condition variables, always use `pthread_cond_destroy()` to destroy the condition variable, and then call `free()` to release the memory.
9. Condition wait and condition timed wait are cancel points.

Programming Notes on Thread-Specific Data

1. Create a key only once by using `pthread_once()`.
2. Specify a destructor function to free any memory that has been associated with the key by a thread.
3. When a thread first calls `pthread_key_create()`, the value of `NULL` will be associated with the key in all the threads.
4. Whenever a new thread is created afterwards, the value of `NULL` will be associated with all defined keys in the new thread.

```
#include <pthread.h>

pthread_mutex_t m;
pthread_cond_t cv;
int ready=0;
cleanup_handler(pthread_mutex_t *m)
{
       pthread_mutex_unlock(m);
}
func()
{
       pthread_mutex_lock(&m);
       pthread_cleanup_push(cleanup_handler, &m);
       while( !ready)
               pthread_cond_wait(&cv, &m);
       pthread_cleanup_pop(1);
}
main()
{
       pthread_attr_t attr;
       int status;
       pthread_attr_init(&attr);
       pthread_attr_setdetachstate(&attr, PTHREAD_CREATED_UNDETACHED);
       pthread_mutex_init(&m, NULL);
       pthread_cond_init(&cv, NULL);
       pthread_create(&thd, &attr, func, 0);
       pthread_cancel(thd);
       pthread_mutex_lock(&m);
       printf("ready= %d\n", ready);
       ready=1;
       pthread_mutex_unlock(&m);
       pthread_join(thd, &status);
       /* Cancel status is specified by symbolic constant: PTHREAD_CANCELED */
       printf("status = %d\n", status);
       /* status must be equal to PTHREAD_CANCELED */
}
```

Figure 13. Thread-cancel API

Additional POSIX.1 functions such as `open()`, `read()`, and `write()` also have cancel points. The POSIX.4a Draft 7 or AIX 4.1 documentation contains a complete list of cancel points.

Signals

POSIX.4a Draft 7 defines the following model for signal handling in a threaded program.

- **Signal handlers are per process**: When a thread installs a signal handler, this handler is invoked when the signal occurs in any thread.
- **Signal masks are per thread**: A thread can block a signal from delivery, but this will not prevent other threads from receiving the `signal.int sigthreadmask(int which, sigset_t *set, sigset_t *oset)`. Similar to the `sigthreadmask()` function, it specifies the same set of arguments.
- **Single delivery of signals**: A signal is delivered to one thread; if more than one thread is interested in the same signal and this signal occurs, then one thread (in an unspecified order) will receive the signal.
- **Wait for asynchronous signals**: POSIX 4a defines a new interface called `sigwait()` to wait for asynchronous signals. Use `sigwait()` to wait for asychronous signals in a dedicated thread. Do not use `sigwait()` for synchronous signals such as `SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, SIGSYS,` and `SIGPIPE`. The remaining signals are asynchronous. In addition, the signals `SIGKILL` and

SIGSTOP cannot be caught by sigwait(); if attempted, sigwait() will return EINVAL.

The syntax for sigwait() is as follows:

```
int sigwait(const sigset_t *set,
   int *signal)
```

The set specifies a set of signals that can be waited. The signal actually caught is returned in the signal argument. It is important to block specified signals before calling sigwait. Figure 14 shows how to use sigwait() to wait for asynchronous signals.

POSIX.1 Functions

This section discusses the behavior and changes necessary to use fork() in a multithread environment. It also addresses pthreadatfork() and non-local jumps.

fork( ): If a thread invokes fork(), only the calling thread is re-created in the child process. The entire data is copied to the new process. The mutex states in the parent process are duplicated into the child process in addition to any other state. AIX 4.1 does not support the forkall() function in which all the threads in the parent are duplicated in the child.

To protect the state of mutexes and other states in the presence of fork(), the pthread_atfork() is provided in POSIX.4a. A discussion of this is followed by the impact of threads on standard C functions.

pthread_atfork( ): Deadlocks can occur if a multithreaded program invokes fork(). For example, before fork() is called, if a thread locks a mutex, that thread may not exist in the child process to release the mutex. This is because the entire data space is copied in the child process without re-creating all the threads (except the calling thread). Any further attempt to lock this mutex will result in a deadlock. Therefore, it is important to bring the mutexes to a known state before fork() is called and to restore the state after the fork() by using pthread_atfork(). The syntax is as follows:

```
int pthread_atfork(void (*prepare)(),
  void (*parent)(), void (*child)())
```

The prepare handler is invoked before fork() is called. The parent handler is invoked in the parent process after fork(). The child handler is invoked in the child process after fork().

Several calls can be made to pthread_atfork() before invoking fork().

Programming Notes on Thread Cancellation

1. You can join a thread that is created with attributes set to PTHREAD_CREATED_UNDETACHED after it is cancelled. A cancelled thread returns the status of PTHREAD_CANCELED (which is -1 in AIX 4.1).
2. Thread cancellation occurs only at cancel points.
3. If a thread locks a mutex and the thread is cancelled, be sure to push a cleanup handler to release the mutex. Use pthread_cleanup_push() and pthread_cleanup_pop() to do this.
4. Note that the cancellation cleanup handlers work on a per-thread basis and with the block.
5. When a thread is cancelled, the thread cleanup handlers are invoked, followed by the thread-specific data destructors. The thread is then terminated.

Programming Notes on Thread-Cancel States

1. The cancel state and cancel type work on a per-thread basis. Changing the state and type of a given thread does not modify the corresponding values of other threads in the process.
2. A cancel issued on a thread will be left pending until the thread enables the state or enters a cancel point.
3. A cancelled thread should release any mutexes held and should destroy any thread-specific data held.
4. If a thread created with PTHREAD_CREATE_UNDETACHED is cancelled, it can be joined. When it is joined, it returns the status of PTHREAD_CANCELED (which is typically the value of -1).
5. Just as POSIX.4a functions cannot be invoked from signal handlers, there are also restrictions on what can be done when asynchronous cancelability is turned on and the pthreads APIs are invoked. The only async-cancel safe functions are pthread_cancel(), pthread_setcancelstate(), and pthread_setcanceltype(). Because most pthreads APIs are not async-cancel safe, be careful when turning on the async cancelability. Do this only for invoking non-threaded libraries that need to be cancelled and whose state can be cleaned up, or for libraries whose state does not impact the process adversely.

```
#include <pthread.h>
#include <signal.h>
waiter()
{
     struct sigset_t set, oset;
     int signal;
     int ret;
     sigemptyset(&set);
     sigaddset(&set, SIGUSR1);
     sigaddset(&set, SIGQUIT);
     sigaddset(&set, SIGALRM);
     /* Block the signals. */
     sigthreadmask(SIG_BLOCK, &set, &oset);

     /* Wait for the asynchronous signals */
     ret = sigwait(&set, &signal);
     if(ret) {
             /* Handle the error case. ret contains the error number */
     }
     /* signal contains the signal caught */
     printf("Signal caught=%d\n", signal);
}
main()
{
     pthread_t thd;
     pthread_create(&thd, NULL, waiter, NULL);
     /* Wait for signal */
     for(;;);
}
```

Figure 14. Use of sigwait()

Programming Notes on Cancel Point

1. Cancel points are usually provided whenever a particular function call (system call or library) may take a long time to complete. For example, a blocking system call (such as `read()`) is a cancel point. Also, cancel points occur whenever the four pthread APIs are invoked.
2. A cancel point also occurs whenever the state changes from `PTHREAD_CANCEL_DISABLE` to `PTHREAD_CANCEL_ENABLE`.
3. Whenever a thread is about to enter a cancel point, there must be cancel cleanup handlers to cleanup the state if the thread is cancelled. Mutexes held must be released after the thread is cancelled.

These registered handlers will be invoked in a specific order when `fork()` is called:

- Prepare handlers: Last-In-First-Out (LIFO)
- Parent handlers: First-In-First-Out (FIFO)
- Child handlers: First-In-First-Out (FIFO)

It is important to register the handlers in the correct order in the application. Each library is responsible for registering its `atfork` handlers.

Standard C Library Functions: A function is *thread safe* if it can be called concurrently by multiple threads. Most standard C library functions are already thread safe, but some have interfaces that require keeping state within the called function, which is usually done in static memory. Such functions cannot, however, be called safely from multiple threads concurrently without modifying each other's data stored in the static memory. Such functions are redefined to pass the state as an argument, and these functions are renamed with an `_r` suffix.

Stdio Functions: The functions in C Standard Input/Output (`stdio`) such as `putc()`, `getc()`, `getchar()`, and `putchar()` are thread safe. The application can do its own locking by using the following unlocked functions:

```
#include <stdio.h>
int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

An application can perform its own locking before it invokes the above functions using the following locking functions:

```
#include <stdio.h>
void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

This is consistent with the locking used in the thread-safe I/O functions: `getc()`, `getchar()`, `putc()`, and `putchar()`.

Non-Local Jumps: A thread can save context in the jump buffer (`jmp_buf`) and use `longjmp()` or `siglongjmp()` to restore the saved context within the context of the same thread. It is incorrect to call `longjmp()` or `siglongjmp()` to transfer control to a thread using the jump buffer of some other thread.

Summary

To write well-behaved multithreaded programs, it is necessary to understand the threads concepts formalized in POSIX.4a. It is equally important to understand how the POSIX.1 model is extended to the multithreaded programming environment. This article has discussed the key aspects of the POSIX.4a threads model and the relationship with POSIX.1. Once these are well understood, multithreaded programming becomes easier.

Chary G. 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 lead for the threads package on AIX and OS/2® Distributed Computing Environment (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.*

Programming Notes on Signals

1. None of the POSIX.4a APIs are signal safe, so they should not be invoked from signal handlers. Do not create threads, lock or unlock mutexes, signal or wait for condition variables, manipulate the cancel state or type of any thread, create or destroy thread-specific context, or change or set the scheduling policy or priority of any thread. The only safe functions to call from a signal handler are those specified in POSIX.4.
2. When using the `sigwait()` call, the signals to be waited on must be set to be blocked by a call to `sigthreadmask()`. To get portable and deterministic behavior, the `sigwaited` signals need to be blocked in all the threads in the process.
3. The `sigprocmask()` call is not defined in a multithreaded environment, although AIX 4.1 aliases it to `sigthreadmask()`.
4. Do not install a signal handler for a signal and also wait for it through a call to `sigwait()`, because results are not guaranteed.
5. The `pthread_kill()` operation can send a signal to a specific thread. The result is that the signal handler, if any, is invoked in the context of the target thread. If there is no signal handler for the signal, the default action occurs. Thus, a terminating or a stopping signal will stop the process, not just the target thread.
6. A call to `kill()` can send a signal to the process.
7. Be careful in making calls to functions that modify global data in a multithreaded environment. If a function is to be called from a signal handler, make sure that it is reentrant with respect to signals. A function is reentrant with respect to signals if it can be called from a thread and a signal handler concurrently, with deterministic results.
8. If a function is to be called from a signal handler as well as a thread, make it reentrant by ensuring that changes to global data occur atomically by using `_check_lock()` or similar atomic instructions.