Porting DCE Threads Programs to AIX 4.1

By Chary G. Tamirisa

The AIX 4.1 POSIX threads (pthreads) package is based on POSIX.4a Draft 7. Existing applications written to AIX 3.2.4 or 3.2.5 Distributed Computing Environment (DCE) pthreads must be rewritten to work on AIX 4.1. This article provides information to help developers migrate existing AIX DCE pthreads applications to AIX 4.1.

This article assumes that the reader is familiar with POSIX.4a Draft 4 and the AIX DCE pthreads package. The article is written as a set of tables showing the differences between the AIX DCE pthreads and the AIX 4.1 pthreads. Since the reentrant C library is part of the POSIX.4a specification, differences in these interfaces are also presented.

AIX DCE 1.2 pthreads

The DCE pthreads package shipped in AIX DCE 1.2 consists of the POSIX.4a Draft 4 Application Programming Interface (API) and several extensions, shown in Figure 1. It also provides an exception support not specified in POSIX.4a.

The AIX 4.1 threads support (the pthreads and reentrant C library functions based on POSIX.4a Draft 7) introduces the following changes:

- Syntax changes to some APIs in Draft 4
- Return of error codes instead of setting the global `errno` for all pthread functions
- New functions introduced in Draft 7 to enhance thread cancellation and thread scheduling
- Differences in signal handling between Draft 4 and Draft 7
- Specification of cancellation points
- Changes to the reentrant C library API in Draft 7

The pthreads Programming Model

The DCE pthreads provides a set of library functions for parallel programming as well as real-time scheduling. In this model, the programmer identifies sequences of program flow that can be parallelized. These parallel sequences of execution can then be hoisted on the threads.

The pthreads programming model preserves the POSIX.1 functionality, especially Input/Output (I/O), on a per-thread basis. For example, a thread issuing a blocking `read()` only blocks itself. The pthreads programming model is written as an extension to POSIX.4, the standard for Real-Time Extensions to POSIX.1. POSIX.4a preserves the POSIX.4 semantics on a per-thread basis. The pthread model adds the following features to enable parallel and real-time programming:

- Thread management functions
- Synchronization functions
- Scheduling functions to meet real-time requirements
- Thread-specific data functions
- Thread cancellation functions

Base Functionality
POSIX.4a Draft 4 API
Draft 4 signal subsystem
Support for POSIX.1 API
Reentrant C library

Extensions
Non-portable extensions to the pthread API (with suffix `_np`)
Exception returning pthreads API (in addition to error returns)
Exception handling (TRY/CATCH)

Figure 1. Contents of AIX DCE 1.2 pthreads package

Chary G. Tamirisa

AIX DCE 1.2 pthreads versus AIX 4.1 pthreads

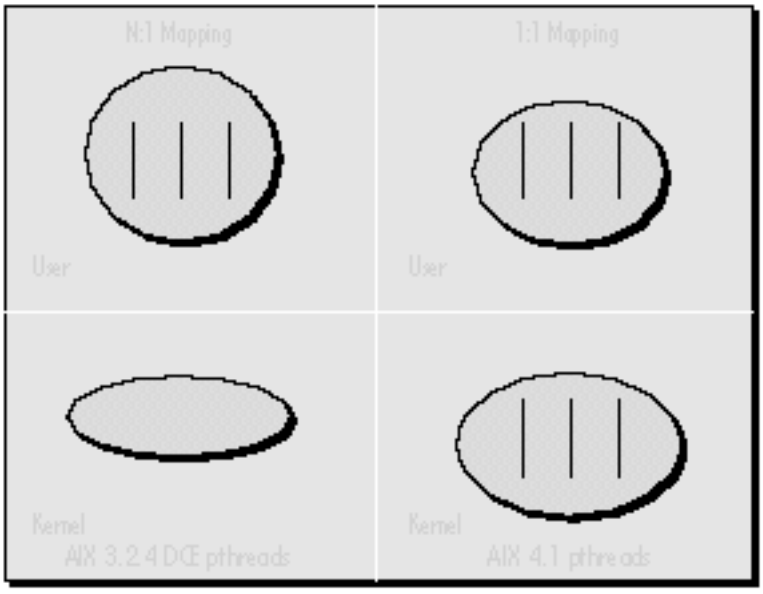

Figure 2. AIX DCE 1.2 pthreads versus AIX 4.1 pthreads

As shown in Figure 2, the AIX DCE pthreads package is purely a user-level threads package; AIX 3.2.4 has no kernel support for threads. AIX 4.1 provides kernel support so that user level threads are supported over these kernel threads. AIX 4.1 provides a 1:1 mapping of user level threads to kernel threads. This 1:1 mapping means that there is a corresponding kernel thread for every user level thread. Currently, AIX 4.1 allows a maximum of 512 simultaneous threads in a user process.

DCE pthreads Package

As shown in Figure 3, DCE pthreads provides the reentrant C library (`libc_r.a`), the pthreads API based on POSIX.4a Draft 4, the signal handling, and the exception handling support. Applications written to DCE pthreads link with `libc_r.a` and `libpthreads.a`. A stanza is provided in `/etc/xlc.cfg` to specify the method of invocation of `cc_r` and `xlc_r`. Multithreaded applications should use these stanzas for compiling threaded programs written to AIX DCE pthreads. Note that special modifications are made to `crt0` for supporting multithreaded processes. These modifications are also specified in the stanzas for `cc_r`.

Porting Issues

This section details the differences between AIX DCE pthreads and the AIX 4.1 pthreads library.

Changes in Error Returns

In AIX DCE, the pthreads functions on error, return a -1, and set the global `errno` variable to the appropriate value. All AIX 4.1 pthreads functions return error number as return values. Applications written to AIX DCE pthreads must be changed to handle AIX 4.1 error returns properly. Figure 4 shows an example.

Data Types

AIX DCE pthreads introduced additional data types, and these data types should be replaced with those specified in POSIX.4a Draft 7, as shown in Figure 5.

Exception Handling

AIX DCE pthreads supports exceptions to handle error conditions and thread cancellations. Exceptions occur in the following situations:

- **When the current thread is canceled**. If a TRY/CATCH block exists around the cancel point (such as a `pthread_join()` call), the appropriate CATCH block is entered. If there is no TRY/CATCH block, the cancelled thread terminates execution.

  POSIX.4a specifies the method of cleanup upon cancellation through two functions: `pthread_cleanup_push()` and `pthread_cleanup_pop()`. The AIX DCE pthreads integrated these two mechanisms so that if a TRY/CATCH and a cleanup handler exist, both are invoked in the Last-In-First-Out (LIFO) order.

- **When the signals `SIGPIPE` or `SIGSYS` are received in the current thread** (unless the signal is set to be ignored). If a TRY/CATCH block exists for handling these exceptions, then the proper block is entered.

- **When the exception raising pthread APIs are used**. These APIs are invoked when a program includes `<pthread_exc.h>` instead of the `<pthread.h>` header file. In `<pthread_exc.h>`, the `pthread_*()` functions are mapped to a corresponding exception-raising API.

AIX 4.1 pthreads does not support exception handling because POSIX.4a does not specify exception handling. Therefore, programs written to handle exceptions must be rewritten to work on AIX 4.1. For cancellation cleanup handling, replace TRY/CATCH with `pthread_cleanup_push()` and `pthread_cleanup_pop()`. Replace all other excep-

tion handling with code that handles error returns from the `pthread_*()` calls.

POSIX.1 Support

All POSIX.1 and C standard functions that suspend the calling process are redefined in POSIX.4a to suspend the calling thread. In AIX DCE this is called *non-blocking I/O support.*

The following functions suspend the entire process: `waitpid()`, `wait()`, `sigsuspend()`, `pause()`, and `tcdrain()`. The DCE pthreads package does not support timed input operations using `VMIN` and `VTIME` (through `ioctl()`).

The AIX DCE pthreads provides support for POSIX.1 I/O functions through a library emulation of non-blocking I/O (there is no kernel support for this in AIX 3.2.4); however, not all functions are supported. For example, no non-blocking support exists for the following: shared memory, System V semaphores, message queue operations, or `poll()`.

This problem is bypassed in AIX 4.1 because all I/O calls are supported to block only the calling thread, not the entire process. Thus, AIX 4.1 now supports all of these functions.

Thread Functions

There are several thread functions described below.

Default attributes for thread: Figure 6 shows a comparison of how a thread is created with default attributes in AIX DCE and AIX 4.1. The figure shows two methods for creating a thread with default attributes in AIX 4.1.

Changes in thread detach: In AIX DCE pthreads, the detach state can be specified by calling the function `pthread_detach()` with the thread ID as the argument. In AIX 4.1, you can specify the thread detach state either at the time of thread creation (by specifying the thread attribute), or after thread creation by using `pthread_detach()`. Programs can use thread attributes to specify the detach attribute or use `pthread_detach()` as appropriate.

There is a difference in the default behavior of detach between AIX DCE pthreads and AIX 4.1 pthreads. In AIX DCE pthreads, a thread can be joined by a call to `pthread_join()` by default; however, in AIX 4.1 pthreads, a thread is created by default in a detached state and cannot be joined as a default. In AIX 4.1, you must set the detach state to `PTHREAD_CREATE_UNDETACHED` by a call to `pthread_attr_setdetachstate()` to

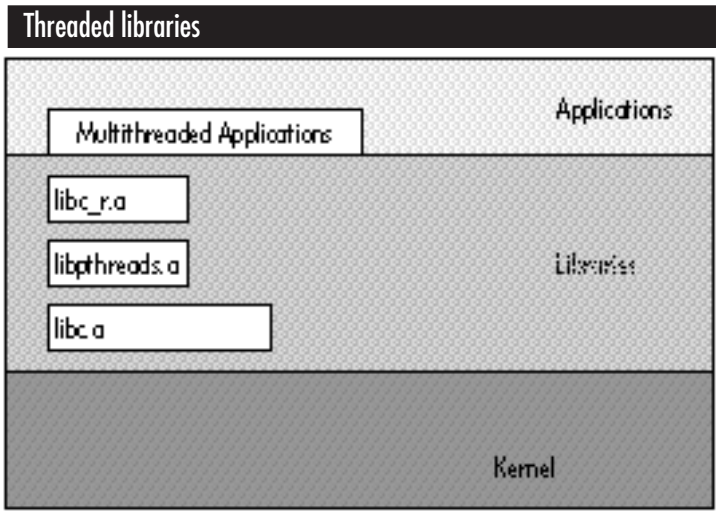

Figure 3. The relationship between threaded libraries

AIX DCE pthreads

```
{int ret;
int ptr;
ret = pthread_join(thread, &ptr);
if (ret== -1) printf("error =%d\n", errno);
}
```

AIX 4.1 pthreads

```
{ int ret;
int ptr;
ret = pthread_join(thread, &ptr);
if(ret) printf("error =%d\n", ret);
}
```

Figure 4. Comparison of error returns in AIX DCE pthreads and AIX 4.1 pthreads

enable you to execute `pthread_join()` on a thread. Figure 7 shows an example.

Dynamic package initialization: The `pthread_once()` initialization symbolic constant is changed from `pthread_once_init` in AIX DCE to `PTHREAD_ONCE_INIT` (lower case to upper case).

Changes in scheduling: Significant changes were made in thread scheduling because of integration of scheduling with the POSIX.4 real-time standard. Specifically, the `sched_param` structure is used in AIX 4.1 to specify scheduling policy and priority. Figure 8 shows the AIX 4.1 structure.

You can use the thread attribute for scheduling to specify scheduling policy and priority. Set `sched_policy` and `sched_priority` in the thread attribute object by specifying a `sched_param` structure with the desired values for policy and priority, and invoke `pthread_setschedparam()`.

AIX DCE pthreads

```
pthread_startroutine_t start_routine
pthread_addr_t arg
pthread_destructor_t destructor
pthread_initroutine_t init_routine
```

AIX 4.1 pthreads

```
void *(*start_routine) (void *)
void *arg
void (*destructor(void *))
void (*init_routine)(void)
```

Figure 5. Mapping AIX DCE introduced types to AIX 4.1

AIX DCE pthreads

```
{
pthread_t thd;
extern func(int arg);
int arg;
pthread_create( &thd, pthread_attr_default, func, arg);
}
```

AIX 4.1 pthreads Alternative 1 (Recommended)

```
{
pthread_t thd;
extern func(int arg);
int arg;
pthread_create( &thd, NULL, func, arg); /* Note NULL implies default thread attributes */
}
```

AIX 4.1 pthreads Alternative 2

```
{
pthread_t thd;
extern func(int arg);
int arg;
pthread_create( &thd, &pthread_attr_default, func, arg);
}
```

Figure 6. Comparison of thread creation with default attributes

AIX DCE pthreads

```
func(int i)
{
   /* The thread function here.*/
}
main()
{
   int status;
   pthread_t thd;
   pthread_create(&thd, pthread_attr_default, func, 0);
   pthread_join(thd, &status);
}
```

AIX 4.1 pthreads

```
main()
{
   pthread_t thd,
   pthread_attr_t attr;
   /* Initialize the thread attribute and specify the detach state */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_UNDETACHED);
   /* Create a thread that is joinable */
   pthread_create(&thd, &attr, func, 0);
   pthread_join(thd, &status);
}
```

Figure 7. Creation of a joinable thread

In addition, applications that use `pthread_attr_setprio()` or `pthread_attr_getprio()` can be rewritten to use the `sched_param` structure to specify the priority.

In AIX DCE, the function `pthread_setprio()` sets the priority and returns the old priority. A similar functionality can be obtained in AIX 4.1, as shown in Figure 9. Note also that AIX DCE pthreads scheduling symbolic constants are different in AIX 4.1 (see Figure 10).

Differences in Mutex

In AIX DCE, the default value of the mutex attribute is `pthread_mutexattr_default`. In AIX 4.1, you specify this value with `NULL`. AIX 4.1 supports static initialization of mutex variables and has changed the arguments to mutex variable initialization. Figure 11 shows examples.

AIX 4.1 allows static initialization of mutexes, as shown in the following example:

```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
```

Figure 12 shows changes in attribute creation and destruction functions.

Figure 13 shows changes in `pthread_mutex_trylock()` return values.

Mutex Types

AIX DCE provides three types of mutexes: fast, non-recursive, and recursive. It also provides `-np` functionality (see Figure 33) to obtain these types of mutexes. The AIX 4.1 `libpthreads` implementation supports only the non-recursive mutex type. Application programs needing other mutex types must provide it themselves.

Differences in Condition Variables

In AIX DCE, the default value is `pthread_condattr_default`; in AIX 4.1, this value is specified with a `NULL` value. AIX 4.1 supports static initialization of condition variables and has changed the arguments to condition variable initialization. Figure 14 shows examples.

Since AIX 4.1 does not support the `{_POSIX_THREAD_PROCESS_SHARED}` options, it also does not support sharing of interprocess condition variables.

Static initialization of condition variables is permitted in AIX 4.1, as shown in the following example:

```
struct sched_param {
     int sched_policy;
     int sched_priority;
}
```

Figure 8. New sched_param structure

```
{
struct sched_param param;
int policy;
pthread_getschedparam(thd, &policy, &param);
/* policy and param structure is filled */
/* Set the new priority */
param.sched_priority = new; /* priority */
pthread_setschedparam(thd, &policy, &param);
}
```

Figure 9. Setting scheduling parameters

AIX DCE pthreads	AIX 4.1 pthreads
SCHED_FG_NP	Use SCHED_OTHER
SCHED_BG_NP	Not Supported
PRI_FIFO_MIN	PTHREAD_PRIO_MIN
PRI_FIFO_MAX	PTHREAD_PRIO_MAX
PRI_RR_MIN	PTHREAD_PRIO_MIN
PRI_RR_MAX	PTHREAD_PRIO_MAX
PRI_FG_MIN_NP	Use PRI_OTHER_MIN
PRI_FG_MAX_NP	Use PRI_OTHER_MAX
PRI_BG_MIN_NP	Not Supported
PRI_BG_MAX_NP	Not Supported
PRI_OTHER_MIN	DEFAULT_PRIO
PRI_OTHER_MAX	DEFAULT_PRIO
PTHREAD_DEFAULT_SCHED	PTHREAD_EXPLICIT_SCHED
PTHREAD_INHERIT_SCHED	PTHREAD_INHERIT_SCHED

Figure 10. Changes to the scheduling-related symbolic constants

AIX DCE pthreads

```
{
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, pthread_mutexattr_default);
}
```

AIX 4.1

```
{
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
}
```

Figure 11. Comparison of specifying default mutex attributes

AIX DCE pthreads	AIX 4.1 pthreads
pthread_mutexattr_create() pthread_mutexattr_delete()	pthread_mutexattr_init() pthread_mutexattr_destroy()

Figure 12. Mapping mutex attribute functions

AIX DCE pthreads	AIX 4.1 pthreads
Returns 0 if another thread owns the lock. Returns 1 if the current thread acquires the lock. Returns -1 and sets errno to indicate EINVAL.	Returns EBUSY if another thread owns the lock. Returns 0 if the current thread acquires the lock. Returns EINVAL in case of error in args.

Figure 13. Changes in the return values of pthread_mutex_trylock()

AIX DCE pthreads	AIX 4.1 pthreads

{
pthread_cond_t cond;
pthread_cond_init(&cond, pthread_condattr_default);
}
	{
	pthread_cond_t cond;
	pthread_cond_init(&cond, NULL);
	}

Figure 14. Specification of default attribute of condition variables

AIX DCE pthreads	AIX 4.1 pthreads
pthread_condattr_create() pthread_condattr_delete()	pthread_condattr_init() pthread_condattr_destroy()

Figure 15. Mapping condition variable attribute functions

AIX DCE pthreads	AIX 4.1 pthreads
Returns 0 on success. Returns -1 and sets errno to EAGAIN on timeout. Returns -1 and sets errno to EINVAL on invalid args.	Returns 0 on success. Returns ETIMEDOUT on timeout. Returns EINVAL on invalid args.

Figure 16. Changes in pthread_cond_timedwait() return values

```
pthread_cond_t cond =
     PTHREAD_COND_INITIALIZER;
```

Figure 15 shows the changes in attribute creation and destruction functions.

Changes in `pthread_cond_timedwait()` return values are shown in Figure 16.

Thread Cancellation

A thread has a cancel state and cancel type associated with it. State of cancelability determines whether or not a thread can be cancelled. If the state has been set to enable cancellation, then the type of cancelability will determine when the thread is actually cancelled.

In the AIX DCE pthread function, `pthread_setcancel (int` state) is used to set general cancelability. If a thread's general cancelability is `ON`, then its cancelability is controlled by the state of the async cancelability. If its async cancelability is `ON`, it can be cancelled at any point. In all other cases, the thread can be cancelled only at specific cancellation points, such as condition waits, thread joins, or calls to `pthread_testcancel()`. The default states of cancelability in DCE are general cancelability `ON` and async cancelability `OFF`. This state can be achieved in AIX 4.1, as shown in Figure 17.

AIX DCE-based applications should be rewritten as described in Figure 18 to work on AIX 4.1.

```
pthread_setcancelstate( PTHREAD_CANCEL_ENABLE, &oldstate);
pthread_setcanceltype( PTHREAD_CANCEL_DEFERRED &oldtype)
```

Figure 17. Setting default cancel state and cancel type in AIX 4.1

AIX DCE pthreads	AIX 4.1 pthreads
`pthread_setcancel(CANCEL_ON)`	`pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate)`
`pthread_setcancel(CANCEL_OFF)`	`pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate)`
`pthread_setasynccancel(CANCEL_ON)`	`pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype)`
`pthread_setasynccancel(CANCEL_OFF)`	`pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype)`

Figure 18. Mapping thread cancel functions to set cancel state and cancel type

AIX DCE pthreads	AIX 4.1 pthreads
`void atfork( void *user_state, void (*pre_fork)(), void (*parent_fork)(), void (*child_fork)())`	`int pthread_atfork(void (*prepare)(void), void (*parent)(void), void (*child)(void)`

Figure 19. Atfork handler

Process Primitives

The function that establishes pre- and post-fork handling has been changed in AIX 4.1, as shown in Figure 19. The order of invocation of the pre-fork handlers is now LIFO, while the order of post-fork handlers is First-In-First-Out (FIFO).

In addition, the AIX DCE `atfork()` function raises an exception on error, whereas `pthread_atfork()` returns error numbers.

Signal Subsystem Differences

The following changes were introduced in AIX 4.1:

- **Signal handlers**: AIX DCE pthreads allows two types of signal handlers: a per-thread signal handler for synchronous signals (`SIGILL`, `SIGTRAP`, `SIGIOT`, `SIGEMT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, and `SIGPIPE`) and per-process signal handlers for all other signals except for `SIGKILL`, `SIGSTOP`, and `SIGVIRT`. These handlers can be installed using the `sigaction()` or `signal()` API.

  All signal handlers in AIX 4.1 are installed on a per-process basis; therefore, programs installing signal handlers must be aware that they may replace existing handlers installed by other threads. You cannot install signal handlers for `SIGKILL`, `SIGSTOP`, `SIGALRM1`, and `SIGWAITING`.

- **Waiting for signals**: In AIX DCE pthreads, `sigwait()` can be executed on all asynchronous signals, excluding the synchronous signals listed above and the signals `SIGKILL`, `SIGSTOP`, and `SIGVIRT`. The signal `SIGVIRT` is used for thread time slicing (scheduling), so it is not available for any applications.

  In AIX 4.1, `sigwait()` can be called for all asynchronous signals except the synchronous signals listed above and the signals `SIGKILL`, `SIGSTOP`, `SIGALRM1`, and `SIGWAITING`.

- **Per-process signal mask**: AIX DCE pthreads allows `sigprocmask()` to be used to block signals for the entire process rather than just the calling thread. In AIX 4.1, the `sigthreadmask()` is used to specify the block mask on a per-thread basis. Use of `sigprocmask()` is unspecified in a multithreaded environment, as per POSIX.4a Draft 7. In AIX 4.1, `sigprocmask()` is mapped to `sigthreadmask()`.

Detailed Differences

Figures 20 through 31 show differences in threads API between AIX DCE and AIX 4.1. Figure 32 shows the remaining pthreads functions in AIX DCE and AIX 4.1 in which the only change is the error-return mechanism. This information can be used when porting existing DCE threads-based applications to AIX 4.1 threads.

AIX DCE provides a set of nonportable extensions to the pthreads functions. These are summarized in Figure 33.

AIX DCE pthreads	AIX 4.1 pthreads
`int pthread_attr_create(pthread_attr_t *attr)`	`int pthread_attr_init( pthread_attr_t *attr)`
`int pthread_attr_delete(pthread_attr_t *attr)`	`int pthread_attr_destroy(pthread_attr_t *attr)`

Figure 20. Mapping pthread attribute functions

AIX DCE pthreads	AIX 4.1 pthreads
None	`int pthread_attr_setdetachstate( pthread_attr_t *attr, int detachstate )`
None	`int pthread_attr_getdetachstate( const pthread_attr_t *attr, int *detachstate )`

Figure 21. New detach state thread attributes in AIX 4.1

AIX DCE pthreads	AIX 4.1 pthreads
`int pthread_attr_setstacksize(pthread_attr_t *attr, long stacksize)`	`int pthread_attr_setstacksize ( pthread_attr_t *attr, size_t stacksize)`
`unsigned long pthread_attr_getstacksize (pthread_attr_t attr)`	`int pthread_attr_getstacksize( const pthread_attr_t *attr, size_t *stacksize)`

Figure 22. Mapping thread stack attribute functions

AIX DCE pthreads	AIX 4.1 pthreads
`int pthread_attr_setprio ( pthread_attr_t *attr, int priority)`	`int pthread_attr_setschedparam( pthread_attr_t *attr, const struct sched_param *param)`
`int pthread_attr_getprio ( pthread_attr_t attr)`	`int pthread_attr_getschedparam ( const pthread_attr_t *attr, struct sched_param *param )`
`int pthread_attr_setsched ( pthread_attr_t *attr, int scheduler)`	`int pthread_attr_setschedpolicy ( pthread_attr_t *attr, int policy)`
`int pthread_attr_getsched ( pthread_attr_t attr)`	`int pthread_attr_getschedpolicy ( const pthread_attr_t *attr, int *policy)`
`int pthread_attr_setinheritsched ( pthread_attr_t *attr, int inherit )`	`int pthread_attr_setinheritsched ( pthread_attr_t *attr, int inherit)`
`int pthread_attr_getinheritsched ( pthread_attr_t attr)`	`int pthread_attr_getinheritsched ( const pthread_attr_t *attr, int *inheritsched)`

Figure 23. Mapping thread scheduling attribute functions

AIX DCE pthreads	AIX 4.1 pthreads
`int pthread_setprio( pthread_t thread, int priority)`	`int pthread_setschedparam ( pthread_t thread, int policy, const struct sched_param *param)`
`int pthread_getprio( pthread_t thread)`	`int pthread_getschedparam ( pthread_t pthread, int *policy, struct sched_param *param)`
`int pthread_setscheduler ( pthread_t thread, int scheduler, int priority)`	`int pthread_setschedparam ( pthread_t thread, int policy, const struct sched_param *param)`
`int pthread_getscheduler ( pthread_t thread)`	`int pthread_getschedparam ( pthread_t pthread, int *policy, struct sched_param *param)`

Figure 24. Mapping thread scheduling functions

AIX DCE pthreads	AIX 4.1 pthreads
int pthread_create (pthread_t *thread, pthread_attr_t attr, pthread_startroutine_t func, pthread_addr_t arg)	int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*)(void *), void *arg)
int pthread_once (pthread_once_t *once_block, pthread_initroutine_t init)	int pthread_once (pthread_once_t *once_block, void(*)(void)) _
int pthread_detach(pthread_t *thread)	int pthread_detach(pthread_t thread)
int pthread_delay_np(struct timespec *interval)	None
int pthread_getunique_np(pthread_t *thread)	None

Figure 25. Mapping thread functions

AIX DCE pthreads	AIX 4.1 pthreads
int pthread_keycreate(pthread_key_t *key, pthread_destructor_t destructor)	int pthread_key_create (pthread_key_t *key, void (*destructor)(void *))
int pthread_getspecific(pthread_key_t key, pthread_addr_t *value)	void *pthread_getspecific(pthread_key_t key)[1]
int pthread_setspecific(pthread_key_t key, pthread_addr_t value)	int pthread_setspecific(pthread_key_t key, const void *value)

[1]Note that pthread_getspecific() does not return error in AIX 4.1; it just returns NULL in case of error. To differentiate between a NULL thread-specific value from an error situation, call pthread_setspecific() with the same key and a NULL value. Then, check the error return, if any, and pthread_setspecific() will return EINVAL if the key is found to be invalid.

Figure 26. Mapping thread-specific data

AIX DCE pthreads	AIX 4.1 pthreads
int pthread_mutexattr_create (pthread_mutexattr_t *attr)	int pthread_mutexattr_init(pthread_mutexattr_t *attr)
int pthread_mutexattr_delete (thread_mutexattr_t *attr)	int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)
int pthread_mutexattr_setkind_np (pthread_mutexattr_t *attr, int kind)	None
int pthread_mutexattr_getkind_np (pthread_mutexattr_t attr)	None

Figure 27. Mapping mutex attribute functions

AIX DCE pthreads	AIX 4.1 pthreads
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t attr)	int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *attr)
void pthread_lock_global_np()	None
void pthread_unlock_global_np()	None

Figure 28. Mapping mutex functions

AIX DCE pthreads	AIX 4.1 pthreads

`int pthread_condattr_create ( pthread_condattr_t *attr)`
`int pthread_condattr_delete ( pthread_condattr_t *attr)`
	`int pthread_condattr_init( pthread_condattr_t *attr)`
	
	`int pthread_condattr_destroy( pthread_condattr_t *attr)`

Figure 29. Mapping condition variable attribute functions

AIX DCE pthreads	AIX 4.1 pthreads
`int pthread_cond_init( pthread_cond_t *cond, pthread_condattr_t attr)` `int pthread_get_expiration_np( struct timespec *delta, struct timespec *abstime)`	`int pthread_cond_init( pthread_cond_t *cond, pthread_condattr_t *attr)` `None`

Figure 30. Mapping condition variable functions

AIX DCE pthreads	AIX 4.1 pthreads
`int pthread_setcancel( int state)` `int pthread_setasynccancel( int state)`	`int pthread_setcancelstate( int state, int *oldstate)` `int pthread_setcanceltype( int type, int *oldtype)`

Figure 31. Mapping thread cancellation

```
int pthread_cancel( pthread_t thread)
void pthread_cleanup_push( void (*routine)(void*), void *arg)
pthread_cleanup_pop( int execute)
int pthread_join( pthread_t thread, void **status)
int pthread_mutex_lock( pthread_mutex_t *mutex)
int pthread_mutex_trylock( pthread_mutex_t *mutex)
int pthread_mutex_unlock( pthread_mutex_t *mutex)
int pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_timedwait( pthread_cond_t *cond, pthread_mutex_t *mutex, struct timespec *abstime)
int pthread_cond_broadcast( pthread_cond_t *cond)
int pthread_cond_signal( pthread_cond_t *cond)
int pthread_mutex_destroy( pthread_mutex_t *mutex)
```

Figure 32. Remaining pthreads functions in AIX DCE and AIX 4.1 in which the only change is the error-return mechanism

```
int pthread_signal_to_cancel_np( sigset_t *sigset, pthread_t *thread)
int pthread_delay_np(struct timespec *interval)
int pthread_get_expiration_np(struct timespec *delay, struct timespec *abstime)
void pthread_lock_global_np(void)
void pthread_unlock_global_np(void)
int pthread_mutexattr_setkind_np(pthread_mutexattr_t *attr, int kind)
int pthread_mutexattr_getkind_np(pthread_attr_t attr)
int pthread_cond_signal_int_np(pthread_cond_t *cond)
unsigned int pthread_getunique_np(pthread_t *thread)
int pthread_equal_np(pthread_t thread1, pthread_t thread2)
```

Figure 33. AIX DCE non-portable ("_np") functions

AIX DCE libc_r

```
int localtime_r ( struct tm *result,
   time_t *clock)
int gmtime_r( struct tm *result,
   time_t *clock)
int asctime_r( struct tm *tm, char
   *buffer, int buflen)
char *ctime_r( time_t *clock, char
   *buffer, int buflen)
int rand_r(unsigned int *seed, int *randval)
void utmpname_r(char *newfile, struct utmp_data
   *utmp_data)
int readdir_r( DIR *dirp, struct dirent *result )
```

AIX 4.1 libc_r

```
struct tm* localtime_r (const time_t
   *clock, struct tm *result)
struct tm *gmtime_r( const time_t *clock,
   struct tm *result)
char *asctime_r( const struct tm *tm, char *buf)

char *ctime_r( const time_t *clock, char *buf )

int rand_r(unsigned int *seed)
int utmpname_r(char *newfile, struct
   utmp_data*utmp_data)
int readdir_r( DIR *dirp, struct dirent *entry, struct
   dirent **result)
```

Figure 34. Mapping reentrant C library functions (libc_r)

AIX DCE pthreads

```
int sigwait(sigset_t *set)
int sigprocmask( int how, const
   sigset_t *set, sigset_t *oset)
None
```

AIX 4.1 pthreads

```
int sigwait( const sigset_t *set, int *sig)
int sigthreadmask( int sig, const sigset_t *set, sigset_t *oset)

int pthread_kill( pthread_t thread, int sig)
```

Figure 35. Mapping signal functions

Reentrant C Library Function

Figures 34 and 35 show changes introduced in AIX 4.1. Figure 34 shows the changes made to reentrant C library functions, while figure 35 summarizes the changes made to the pthreads signal-handling subsystem.

Chary G. Tamirisa, *IBM Corporation, LAN Systems Division, 11400 Burnet Road, Austin, TX, 78758. Internet: chary@austin.ibm.com. Since January, 1993, Mr. Tamirisa has been the team lead 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 holds an MS in Computer Science from McGill University and a BTech in Electrical Engineering from the Indian Institute of Technology in Madras, India.*

TPC Ratings for New RISC System/6000s Sizzle

New RS/6000s announced in May of this year have produced industry-leading performance numbers. Results are as follows:

◆ The RISC System/6000 POWERserver™ R24 produced a TPC-A rating of 357.24 tpsA using Oracle® Version 7 (client/server configuration). This is the fastest TPC-A ever reported for a single processor server.

◆ Demonstrating the scalability of RISC System/6000 clusters, a cluster of four POWERserver R24 systems using HACMP/6000 and Oracle Version 7 (client/server configuration) produced a TPC-A rating of 894.08 tpsA.

◆ In the more complex TPC-C benchmark, IBM achieved the fastest Sybase® TPC-C result ever produced. An RS/6000 POWERserver Model 59H using Sybase Version 10.0.1 (client/server configuration) produced a rating of 1122.30 tpmC. n