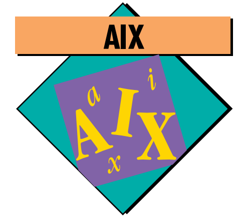
Thread Programming Models: AIX 4.1 vs. OS/2 Warp



By Chary G. Tamirisa

This article presents a comparison of the multithreading models supported in the AIX and OS/2® operating systems. It focuses on the key differences in the two programming models, without describing the details of the Application Programming Interfaces (APIs). The goal is to assist porting of applications between the two platforms. This article assumes the reader is familiar with one of the two models.

The AIX 4.1 and OS/2 (32-bit) operating systems offer powerful programming models for multithreaded programming. The AIX model is based on the IEEE's® POSIX.1c Draft 7 standard for threads whereas the OS/2 thread model is an IBM-specific model. Increasingly, software is being ported from one platform to the other, which creates a need to understand the execution models offered by both systems to simplify application porting. The comparative study of the two threading models that follows does not include details on the actual programming interfaces.

Familiarity with either the AIX 4.1 or the OS/2 model is essential in order to understand this article. The conceptual foundations for the thread services for OS/2 can be found in the OS/2 Warp™ Control Program. The AIX online publications provide extensive documentation of the thread support.

The threads programming model deals with thread creation, mechanisms for synchronization of threads, thread-local memory, thread cancellation, and graceful handling of asynchronous events in a multithreaded process.

Overview

The AIX[1] thread model is based on the POSIX.1c Draft 7 standard for threads. The supported interfaces are a subset of the interfaces defined in the POSIX.1c standard. POSIX.1c specifies the following optional features:

- Stack address for a thread to be specified by the user
- Support for interprocess-shared mutexes and condition variables
- Support for process-scope contention for threads instead of all threads competing at system scope
- Support for protocols that prevent priority inversion in which a low-priority thread runs while a high-priority thread is blocked

AIX does not support these optional features.

The OS/2 thread model shares several common features with AIX. It implements system-scope for threads and allows the thread stack size, but not the stack address, to be specified. It does not provide support for preventing priority inversion. However, OS/2 supports an important feature called *interprocess-shared synchronization*, which allows threads from different processes to synchronize with one another using the mutex and event semaphores.

AIX supports extension of UNIX® signals into a multithreaded process; OS/2 maps signals to exceptions. AIX and OS/2 support process-wide timers. AIX generates timer signals whereas OS/2 uses the semaphore events to inform the process of timer events.

Chary G. Tamirisa

[1] All references to AIX throughout the article refer to AIX 4.1.

The notable differences appear in the synchronization services between AIX and OS/2. The AIX condition variable mechanism differs completely from the OS/2 event semaphores. OS/2 allows mechanisms in which a thread can stop one or all other threads from being scheduled. Since these types of features are very powerful, they should be used very carefully. AIX differs because it has no provision for one thread to directly stop other threads.

The sections that follow provide a detailed comparison of the AIX and OS/2 threading models.

Mapping User and Kernel Threads

AIX and OS/2 map one user thread to one kernel thread called 1:1 mapping. All application threads create corresponding kernel threads. Since each kernel thread uses important kernel resources, the operating system limits the number of threads that can be created in the user process. For example, in AIX each process can have up to 512 threads. In OS/2, each process can have up to a specified (in `CONFIG.SYS`, it is typically 256) number of threads, but subject to a system-wide maximum of 4095 threads.

Threads are the scheduling units of execution in AIX and OS/2.

Threads are the scheduling units of execution in AIX and OS/2. Since all threads are scheduled on a system-wide basis, there is no process-scope scheduling. Due to the system-wide scheduling of threads in both operating systems, applications that create high-priority threads to boost the application performance can adversely affect scheduling of threads from other processes.

Thread Creation

The thread creation Application Programming Interfaces (APIs) for OS/2 and AIX allow the thread function to be specified and provide for an identifier of the created thread to be returned. However, AIX provides a thread attribute parameter that allows threads to be created with the following properties that can be set:

- Thread detach state
- Thread stack size
- Thread stack address
- Thread scheduling inheritance property
- Thread scheduling policy and scheduling priority
- Thread contention scope

OS/2 has no explicit thread attribute parameter when a thread is created. OS/2 allows a flag to be specified that shows whether the newly created thread is ready to run or in a suspended mode. In addition, the flag can also specify whether the memory for the stack is sparse or committed.

Detach/Join

AIX allows two types of threads:

- **Joinable threads:** A thread that allows other threads to join with the newly created thread. A joinable thread must be either joined or detached explicitly in order to free the state (memory) associated with it. Otherwise, memory will be associated with these threads and not released when they are terminated, which creates memory waste.

  OS/2 threads are always joinable. If a thread joins with an already terminated thread, an error code is returned as the state associated with an already terminated thread is freed. Porting the AIX joinable threads to OS/2 requires explicit synchronization through the OS/2 event semaphore APIs.

- **Detached threads:** A thread in which the state associated with it is freed automatically when the thread exits. AIX allows threads to be created in the detached state (this is also the default state). Such threads cannot be joined by other threads. OS/2 detaches threads automatically on their exit.

Stack Size/Address

AIX and OS/2 enable an application to set the stack size. Although POSIX.1c allows the thread stack address to be specified, AIX does not currently support specification of the stack address. OS/2 does not allow specification of the stack address for a thread. AIX does not allow applications to control the management of the thread's stack. However, OS/2 allows committed stack memory and sparse memory to be created.

Scheduling Inheritance

In AIX, the default thread creation attribute results in the newly created thread inheriting the scheduling policy and scheduling priority from the creator thread. Similarly, an OS/2 thread inherits the scheduling class and level of the creating thread.

Setting the scheduling class and priority level of a thread is possible in both AIX and OS/2. One difference, however, between AIX and OS/2 is that AIX allows the thread scheduling policy and priority to be determined by specifying the thread attribute parameter when a thread is created. This makes it possible to create a thread with a different scheduling policy and priority than the creating (parent) thread. When porting AIX applications to OS/2, if the AIX application creates threads with scheduling attributes that differ from those of the creating thread, then the process is different in OS/2. To create the same functionality in OS/2, use the suspended mode for the new thread, then use the OS/2 priority setting API to modify the scheduling class and level of the suspended thread and then resume it.

Thread Termination

A thread can terminate itself in one of two ways: by returning from the thread function or by calling an explicit API to exit the thread. AIX and OS/2 support both types of thread termination. In AIX, `pthread_exit()` in the main thread does not terminate the process; it only terminates the main thread. In OS/2, exiting the main thread in any way terminates the process.

Exit Status

AIX allows a thread to pass its exit status information to another thread through the `pthread_join()` API. In addition, AIX allows cleanup handlers for thread-local memory to be registered. OS/2 does not provide support for a thread to inform another thread in the same process of its exit status. The `DosExit()` API allows exit status of a child process to be available for the parent process.

Thread Suspension

AIX has no API to suspend another thread; each application must explicitly create synchronization so that threads can be suspended or resumed.

OS/2 allows a thread to suspend another thread through `DosSuspendThread(threadid)`. In addition, the OS/2 API `DosResumeThread()` allows a thread to resume the suspended thread.

Wait for Thread to Terminate

In AIX, the `pthread_join()` API allows a thread to wait for termination of only one other thread; there is no direct API to wait for multiple threads as in OS/2. This call blocks if the target thread did not terminate.

AIX requires the use of explicit synchronization APIs to achieve the OS/2 functionality of waiting for termination of multiple threads. In OS/2, the `DosWaitThread()` API allows a thread to wait for any thread—or a specific thread—in the process to terminate. It also allows a thread to determine if another thread has terminated.

Thread Information

In AIX, a thread identifier can be obtained for the current thread by a call to `pthread_self()`. It is often convenient to refer to a thread using a small integer value rather than an address. For this purpose, the extension `pthread_get_unique_np()` is available (note POSIX™ allows such non-portable extensions with a suffix of `_np`). This is helpful when indexing an array with a thread identifier. However, since the IDs can be recycled, it is important to clean up the data after a thread exits.

OS/2 allows a thread to obtain thread-specific information by a call to `DosGetInfoBlocks()`. This API provides the current thread ID—a small integer value.

In AIX, the thread stack size and stack address can be obtained by calling the `pthread_attr_getstacksize()` and `pthread_attr_getstackaddr()` APIs respectively. Similarly, the scheduling policy and priority can be obtained dynamically by calling `pthread_getschedparam()`.

In OS/2, a call to `DosGetInfoBlocks()` provides both the thread and the Process Information Blocks. The available information includes the stack pointer, stack limit, thread priority class, and thread priority level.

A thread can terminate itself by returning from the thread function or by calling an explicit API to exit the thread.

Synchronization

AIX supports mutual exclusion and thread synchronization through the condition variables. This support is limited to threads within the same process, and currently does not support mutual exclusion between threads in different processes, or condition variable synchronization between threads in different processes.

OS/2 provides three types of semaphores for thread synchronization within the same process or with threads across processes. The three types of semaphores include Event, Mutex, and Multiple Wait semaphores (Mux semaphore). Multiple Wait semaphores allow a thread to wait for several event or mutex semaphores. AIX does not support the Multiple Wait semaphores; however, they can be supported using the mutexes and condition variable APIs.

```
pthread_mutex_t mutex; /* Lock the associated mutex */
pthread_cond_t condition;/* Condition Variable */
int flag;/* Boolean */

..

pthread_mutex_lock(&mutex);

while(!flag)
pthread_cond_wait(&mutex, &condition);
/* Release the mutex and block atomically */

pthread_mutex_unlock(&mutex);
```

Figure 1. AIX condition variable usage

Mutexes

AIX and OS/2 APIs that allow threads to guarantee mutual exclusion are similar. AIX supports three types of mutexes: recursive, non-recursive, and fast (non-recursive with no error checking). These mutexes can be created by specifying mutex attributes when initializing a mutex. AIX 4.1.4 does not support mutexes between threads in different processes.

The OS/2 mutex semaphores are similar to the AIX recursive mutexes. OS/2 also supports mutex semaphores to allow mutual exclusion between threads in different processes. Therefore, porting applications to AIX that depend on process-shared mutexes requires some porting effort. A hint is to use the AIX atomic instructions (`_check_lock()`) and to use shared memory to obtain process-shared mutex functionality. Another approach may be to write a kernel extension that provides the process-shared mutexes on AIX.

Condition Variables

AIX supports condition variables to allow threads in the same process to synchronize among themselves. Associated with the AIX condition variable are a boolean predicate and a mutex. Together these ensure an atomic operation in which a thread acquires a mutex lock and checks to see if it should block. If it should block, it blocks and releases the lock atomically. The code segment in Figure 1 captures the idiom, illustrating the case of condition variables. The condition variable mechanism differs from the OS/2 event semaphores, which are similar to counting semaphores (a counting semaphore is one that remembers the number of times an event is posted). The significant difference between an OS/2 event semaphore and a counting semaphore is in the wake up when an event is posted. The OS/2 event wakes up all waiting threads whereas a counting semaphore wakes up one of the waiting threads.

The OS/2 event semaphores allow the post count to be set to zero automatically after the query; therefore events that are posted are not lost.

Since semaphore waiting can be indefinite, a timeout parameter is often very useful to keep application threads from blocking forever. Both AIX and OS/2 allow a timeout parameter while waiting for an event.

AIX supports interfaces that wake up either one waiting thread or all the waiting threads. OS/2 event semaphore posting provides a broadcast semantic—all waiting threads for this event are awakened.

The event semaphores in OS/2 allow an interrupt (exception) handler to post on an event semaphore. The AIX condition variables mechanism with its associated mutex is not generally safe to be invoked from signal handlers because self-deadlock may occur.

Thread Scheduling

AIX provides two types of scheduling priorities: fixed and variable. The real-time scheduling policies—First-In-First-Out (FIFO) and Round-Robin (RR)—are in the fixed priority class where the operating system will not change the priorities selected by the application.

AIX supports a UNIX type of variable priority scheduling policy called `SCHED_OTHER` under the second category.

In OS/2, thread scheduling behavior is determined by the priority class and the priority level to which it belongs. The priority class and level are similar to the AIX thread scheduling policy and scheduling priority respectively.

OS/2 provides four scheduling classes (policies) that allow applications to set threads with scheduling behavior that is appropriate to the job they do:

- **Time-Critical.** Class of threads with the highest priority; since they run with fixed priorities, the OS will not adjust the priorities of these threads
- **Fixed-High.** Scheduling class (policy) that allows the OS to adjust its priorities under the `DYNAMIC` option; threads have higher scheduling priority than those with the Regular class
- **Regular.** Scheduling class (policy) that allows the OS to adjust its priorities under the `DYNAMIC` option

- **Idle-Time.** The last class in scheduling class priority; has the lowest scheduling priority; when no other work is to be done, threads with this priority class get to run

OS/2 provides for round-robin scheduling of threads with equal priority within the same class. The OS/2 `TIME-CRITICAL` statement corresponds to the round-robin scheduling policy in AIX. Under the `DYNAMIC` option, the Fixed-High and Regular scheduling classes are similar to the `SCHED_OTHER`.

AIX offers 127 priorities (1–127); all scheduling policies share this range. This means there can be multiple threads within a process that have one priority value, but have different scheduling policies. Although threads are scheduled based on their priorities, their scheduling policies determine what happens when a thread gets preempted. OS/2 scheduling is based on a similar model. OS/2 offers 32 priority-level values (0–31) within each class. These scheduling classes govern thread scheduling.

In AIX, an application needs the effective user ID of the root to create threads with fixed scheduling policies. This is not necessary in OS/2.

It is not possible in AIX to specify scheduling behavior for all threads on a system-wide basis. In OS/2, two types of scheduling behaviors can be chosen system-wide. If the `PRIORITY` statement in `CONFIG.SYS` is set to `ABSOLUTE`, the thread priorities are fixed by the applications. In this case, the operating system will not change priorities of threads. However, if the `PRIORITY` is set to `DYNAMIC`, the priorities of threads are adjusted based on system load and process activity, and on whether the process is doing interactive I/O.

There are two exceptions to the dynamic adjustment that occurs in the Time-Critical and Idle-Time thread classes. The default value for `PRIORITY` is `DYNAMIC` and this allows the operating system to achieve optimal performance. If this is not sufficient, specifying `ABSOLUTE` option for `PRIORITY` enables applications to have better control of the scheduling behavior. AIX has no such system-wide modification of thread scheduling policy or priority. The system implements certain documented scheduling policies and an application can create a thread based on these policies. In AIX, threads created with the default thread attributes share the same scheduling policy and priority.

Scheduling Time Slice

In AIX, an application cannot choose the thread scheduling time slice. In OS/2, the minimum and maximum value that controls the time allocated for threads can be specified by using the `TIMESLICE` statement (in `CONFIG.SYS`). The min value must be an integer greater than or equal to 32 milliseconds. The min value specifies the minimum time that a thread is processed before yielding CPU to another thread of the same priority. The max value must be an integer within the interval of 32 to 65536 milliseconds. The max value specifies the maximum amount of time a thread can be processed before yielding CPU.

Inheritance of Priority

In AIX, when default thread creation attributes are used, an application can create a thread with a different scheduling policy/priority than its own. In OS/2, the priority class and level of the newly created thread are set to the same as the thread that created it.

AIX does not allow a thread to change the priority of another running thread or to change the priority of threads in another process. OS/2 allows the thread to change the priority of threads within the process or the priority of those threads in processes that are its children.

AIX does not allow a thread to change the priority of another running thread or to change the priority of threads in another process.

Once-only Semantics and Critical Sections

For applications and libraries to ensure that shared resources such as mutexes and condition variables are initialized once-only in a multi-threaded environment, AIX provides the API `pthread_once()`. This API guarantees that only one thread invokes the specified once-only initialization function.

OS/2 allows a thread to stop any other thread in the same process from being scheduled when an application invokes the `DosEnterCriticalSection()` API. This can lead to dead-locked threads that may hold resources needed by the thread that invoked `DosEnterCriticalSection`. To avoid this, applications must ensure that this API is invoked when no other thread can hold resources needed by the calling thread. This API is typically invoked, for example, to ensure that initialization of semaphores is done only once in a Dynamic Link Library (DLL).

Thread-Specific Data

AIX provides the interfaces to create, destroy, get, and set thread-specific data. OS/2 provides

interfaces to allocate thread-specific data and to free the data. In AIX, approximately 1,000 thread-specific keys can be allocated, whereas OS/2 allows a maximum of 128 bytes to a thread for associating thread-specific data.

In AIX, the thread-specific data can be destroyed automatically by associating a destructor upon the data creation. Since no destructors are associated with thread-specific data, the application must explicitly call any destructors needed to free up resources when the thread is terminated. If one thread cancels another thread, then it is not possible to automatically free up memory allocated by the canceled thread. The applications must consider this case carefully and free up memory to avoid memory leaks, perhaps by registering exception handlers. The kernel state of any semaphores held by a terminated thread is reset.

Thread Cancellation Services

AIX provides the ability to cancel (that is, to terminate) threads while they are executing in a controlled fashion. For this purpose, AIX defines cancel points, typically provided when a function may block indefinitely. For instance, a `read()` system call may block forever; therefore, it is a cancel point.

AIX allows resources to be released when a thread is terminated.

AIX specifies a list of functions that are cancel points in its pthreads library and the standard C (thread-safe) library. A thread can set two states of cancel: cancel enable or cancel disable. A thread can also set two cancel types: deferred or asynchronous cancellation.

The cancel state can allow or disallow cancellation of a thread independent of its cancel type setting. If the cancel state is set to enable, the cancel type setting determines what happens when a cancel is posted on the thread. If the cancel state is enabled and cancel type is set to asynchronous, the thread is canceled immediately when a cancel is delivered to it. If the cancel state is enabled and cancel type is set to deferred, the thread is canceled only if it is waiting at a cancel point.

AIX also defines APIs to establish cancel cleanup handlers so that when a thread is canceled, it can clean up its state. Whenever a thread terminates, the registered cancel cleanup handlers are invoked.

OS/2 provides a simpler cancel interface in `DosKillThread()`, which terminates any thread other than the current thread. Any thread that is terminated can install an exception handler to capture the termination of the thread and perform any needed cleanup.

Thread-Cleanup

AIX allows resources to be released when a thread is terminated. Thread-specific data can be cleaned up (released) by registering thread-specific data destructor routines when creating a thread-specific data key. Thread cancel cleanup handlers allow a thread to release mutexes and other resources held while executing when threads are canceled. AIX also allows a process to register exit time cleanup handlers.

In OS/2, a process can be terminated through a call to `DosKillProcess()`. In addition, a thread can terminate another thread in the process through `DosKillThread()`. A thread can terminate itself through a call to `DosExit()`. In all these cases, resources held by the application may need to be cleaned up. To allow this, OS/2 raises appropriate exceptions. To clean up semaphores in OS/2, each thread must have an exception handler to clean up during process termination (`XCPT_PROCESS_TERMINATE` or `XCPT_ASYNC_PROCESS_TERMINATE`). Alternately, `DosExitList()` can be used to add a cleanup handler at exit time for the process. If a process owing a mutex semaphore terminates without releasing a mutex, any other process that tries to request the mutex will get the error `ERROR_SEM_OWNER_DIED`.

Cleanup can be done by calling `DosQueryMutexSem()` on it to determine which process died, and then reinitializing appropriately. When a thread terminates itself through `DosExit()`, the `XCPT_PROCESS_TERMINATE` exception is generated. When a thread terminates another thread through `DosKillThread()`, the `XCPT_ASYNC_PROCESS_TERMINATE` exception is generated in the terminated thread. These exceptions provide opportunities for cleaning up resources held by the affected threads.

Signal/Exceptions

AIX multithreading offers the `sigwait()` API that allows a dedicated thread to be set up to wait for asynchronous signals. This prevents undue interruption of application threads. A `sigwait` thread can receive the signal, then arrange for further synchronization with other threads through the mutex/condition variable APIs.

OS/2 does not have native support for `sigwait()`. Instead, it allows asynchronous signals (control C or control Break) to be

delivered to the main or initial thread (thread ID 1). Exception handlers can be set up in the main thread to act on these signal exceptions (signals converted to exceptions).

OS/2 supports the concept of deferring delivery asynchronous signals to the current thread through the concept of "must complete sections." An application can then protect itself from asynchronous exceptions during critical operations that need to mask asynchronous exceptions. Note, however, that synchronous exceptions are not deferred by this mechanism; the application should arrange to handle these synchronous exceptions.

Timers

Both AIX and OS/2 support process-wide timers. When a thread sets up a timer, the expiration of the timer will be reported to the process as a whole, not to the individual thread that set up the timer.

However, AIX and OS/2 differ in the way they communicate the timer expiration. AIX sends a signal (`SIGALRM` or `SIGVIRT`) , whereas OS/2 posts an event semaphore associated with the timer.

Appendix

Figure 2 shows the direct mapping of AIX and OS/2 thread APIs. Not all the AIX thread

```
int pthread_attr_setstacksize(pthread_attr_t *, size_t)
DosCreateThread()

int pthread_attr_getstacksize(const pthread_attr_t *, size_t *)
DosGetInfoBlock()

int pthread_attr_setschedpolicy(pthread_attr_t *, int)
DosSetPriority(ULONG scope, ULONG ulClass, LONG  delta, ULONG PorTid)

int pthread_attr_getschedpolicy(const pthread_attr_t *, int *)
DosGetInfoBlock()

int pthread_attr_setschedparam(pthread_attr_t *,const struct sched_param *)
DosSetPriority(ULONG scope, ULONG ulClass, LONG  delta, ULONG PorTid)

int pthread_attr_getschedparam( const pthread_attr_t *, struct sched_param *)
DosGetInfoBlock()

pthread_t pthread_self(void)
DosGetInfoBlocks()

int pthread_create(pthread_t *, const pthread_attr_t *, void *(*)(void *), void *)
DosCreateThread(PTID ptid, PFNTHREAD pfn, ULONG param, ULONG flag, ULONG cbStack)

int pthread_join(pthread_t, void **)
DosWaitThread(PTID ptid, ULONG option)

void pthread_exit(void *)
DosExit(EXIT_THREAD, ULONG result))

void pthread_yield(void)
DosSleep(ULONG msec)
```

(continued on page 22)

Figure 2. AIX and OS/2 APIs that correspond to each other (OS/2 APIs are set in bold type).

(continued from page 21)

```
int pthread_delay_np(struct timespec *)
DosSleep(ULONG msec)

int pthread_equal(pthread_t, pthread_t)
No Direct Support, but the thread ids can be compared since they are long unsigned
integers.

int pthread_getunique_np(pthread_t *, int *)
DosGetInfoBlocks() gives a structure that contains the thread id.

int pthread_setschedparam(DosSetPriority(ULONG scope, pthread_t,int,const struct
sched_param *)
DosSetPriority(ULONG scope, ULONG ulClass, LONG  delta, ULONG PorTid)

int pthread_getschedparam( pthread_t, int *, struct sched_param *)
DosGetInfoBlocks()  gives a structure that contains the priority class and level that
correspond with the scheduling class and priority respectively.

void pthread_cleanup_push( void (*)(void *), void *)
DosSetExceptionHandler( PEXCEPTIONREGISTRATIONRECORD pERegRec)

void pthread_cleanup_pop(int)
DosUnsetExceptionHandler( PEXCEPTIONREGISTRATIONRECORD pERegRec)

DosUnwindException( PEXCEPTIONREGISTRATIONRECORD phandler, PVOID pTargetIP,
PEXCEPTIONREPORTRECORD pERepRec)

int pthread_cancel(pthread_t)
DosKillThread(TID tid)

int pthread_mutexattr_setpshared( const pthread_mutexattr_t *, int)
(AIX4 does not support)
DosCreateMutexSem ( PSZ pszName, PHMTX phmtx, ULONG flAttr, BOOL32 fState)

int pthread_mutex_init( pthread_mutex_t *, pthread_mutexattr_t *)
DosCreateMutexSem ( PSZ pszName, PHMTX phmtx, ULONG flAttr, BOOL32 fState)

int pthread_mutex_destroy( pthread_mutex_t *)
DosCloseMutexSem (HMTX hmtx)

int pthread_mutex_lock(pthread_mutex_t *)
DosRequestMutexSem (HMTX hmtx, ULONG ulTimeout)

int pthread_mutex_trylock(pthread_mutex_t *)
DosRequestMutexSem (HMTX hmtx, 0)

int pthread_mutex_unlock(pthread_mutex_t *)
DosReleaseMutexSem (HMTX hmtx)
```

(continued on page 23)

Figure 2 (continued). AIX and OS/2 APIs that correspond to each other (OS/2 APIs are set in bold type)

(continued from page 22)

```
int pthread_mutex_setprioceiling(pthread_mutex_t *, int, int *)
(AIX4 does not support)
None
```

```
int pthread_mutex_getprioceiling(pthread_mutex_t *, int *)
(AIX4 does not support)
None
```

```
pthread_mutex_getowner_np(pthread_mutex_t *mutex)
DosQueryMutexSem (HMTX hmtx, PID *ppid, TID *ptid, PULONG pulCount)
```

```
int pthread_key_create( pthread_key_t *, void (*)(void *))
DosAllocThreadLocalMemory( ULONG cb, PULONG *p)
```

```
int pthread_once( pthread_once_t *, void (*)(void))
DosExitCritSec(VOID), DosEnterCritSec(VOID)
```

```
int sigthreadmask(int how, const sigset_t *set, sigset_t *oset)
DosEnterMustComplete( PULONG pulNesting)
      DosExitMustComplete( PULONG pulNesting)
```

```
int pthread_kill(pthread_t, int)
DosRaiseException( PEXCEPTIONREPORTRECORD pexcept)
DosSendSignalException(PID pid, ULONG exception)
```

Figure 2 (continued). AIX and OS/2 APIs that correspond to each other (OS/2 APIs are set in bold type)

interfaces have corresponding OS/2 APIs, and not all the OS/2 APIs have matching AIX interfaces. Figure 2 shows APIs that correspond to each other. Those APIs not listed here do not have direct mapping. Since AIX 4 does not support interprocess shared mutexes and condition variables, porting applications in OS/2 that depend on these facilities is a challenge. It is possible to use other interprocess communications facilities in AIX 4 to obtain an equivalent functionality.

Chary G. Tamirisa, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: chary@austin.ibm.com. Mr. Tamirisa was 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.