Threads Programming in AIX Version 4

By Marc Miller

AIX Version 4 provides developers with the power and flexibility of threads programming, an ideal tool for implementing concurrent processing on multiprocessor hardware. This article describes the various threads implementations in AIX, the basic concepts of threads programming, and the differences between developing with threads and with processes. It also describes some basic calls to help developers start creating and managing threads.

Now that multiprocessor hardware is becoming more common and application developers require concurrent processing, many systems require multiple threads of control. To exploit new hardware and software technology while maintaining compatibility with existing systems, the operating system requires new features that help developers split application processing onto multiple processors. Threads is one feature that AIX provides developers to exploit the powers of parallel processing.

Threads

Threads allows developers to partition applications easily. It is an Application Programming Interface (API) defined by POSIX. The threads API in AIX is based on the `pthreads` interface specified in POSIX 1003.4A draft 4 and 7. AIX Version 3 supports threads in the Distributed Computing Environment (DCE) License Program Product (LPP), an add-on product. It is based on the POSIX draft 4, whereas AIX Version 4 supports a draft 7 version of the POSIX specification. Developers can use any of these programming APIs to create and manage multithreaded processes.

Threads has a number of advantages, including the following.

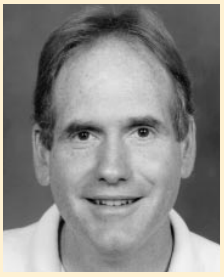
- **Concurrency**: Developers can use threads to implement parallel processing in an application for performance improvements.
- **Overlapping I/O:** Threads provides easier tools for managing asynchronous I/O, which speeds up application performance.
- **Simple Programming Model:** Threads is one of the easiest programming models to learn because of its similarity to C programming.

Single Versus Multithreaded Processes

In a single-thread process, the process is controlled by one stream of instructions at a time. In a multithread process, the process starts with one stream of instructions and creates other instruction streams called *threads* to do tasks. This is similar to one process forking another process. The main difference is that when a process forks off a child process, a hierarchical relationship exists between the parent and child processes. Within a multithreaded process, all threads are peers with no dependency on a parent process. For example, any thread can kill another thread within the same process.

AIX provides many facilities such as thread creation, termination, synchronization, communication, error recovery, and management to help the developer control how threads behave within a process. These facilities help developers write parallel applications.

Just as a simple process can make system calls, so can a thread when it is running in user space. To make system calls, a thread must be able to switch between user and kernel space. In a multithreaded process, a user thread runs in user space, but it is attached to a kernel thread that runs in kernel space. The kernel thread is managed by the scheduler and handles the user thread's kernel requirements. A kernel thread can be attached to a user thread to do user work or it



Marc Miller

can be unattached and running as a kernel thread (similar to a kernel process or kernel extension in AIX Version 3).

The appearance and behavior of kernel threads in the kernel of AIX Version 4 are similar to processes in AIX Version 3. The real differences between threads and processes appear in the user space.

Multithreaded Processes

Figure 1 depicts a typical multithreaded process. Notice that the threads share one version of process-related kernel data, but each thread has its own copy of the registers, some data related to the kernel thread, and a private stack. Therefore, data can be passed via global variables.

The shared process-related kernel data is stored in the proc and user structures. In AIX Version 4 these control blocks have been split into process- and thread-specific structures. The user and proc structures now contain only data that is maintained at the process level. A new thread structure now contains the thread-specific data that used to be in the proc structure. In addition, a uthread structure contains thread-specific data that used to be in the user structure. Figure 2 shows these new control blocks and some sample fields.

In AIX Version 3, the kernel maintained data and scheduled and managed processes. In AIX Version 4, all of these functions must be performed on both processes and threads. Because the thread is now the unit of work, the scheduler needs to manage each kernel thread individually. Some functions, however, still operate on the process level, and these functions affect all threads within a process. For example, setting the `nice` value for a process affects all the threads in the process.

Threads Models

AIX has three basic threads models: M:1 DCE threads, which are implemented in AIX Version 3; 1:1 kernel threads, implemented in AIX 4.1; and M:N threads, which will be available in a future release of AIX.

M:1/DCE Thread Architecture

In AIX Version 3, threads is supported through the DCE `pthreads` LPP, which provides a pure user space implementation of threads. Developers can create and manage multiple threads within a process without any connection to the kernel. The `pthreads` library dispatches and manages all

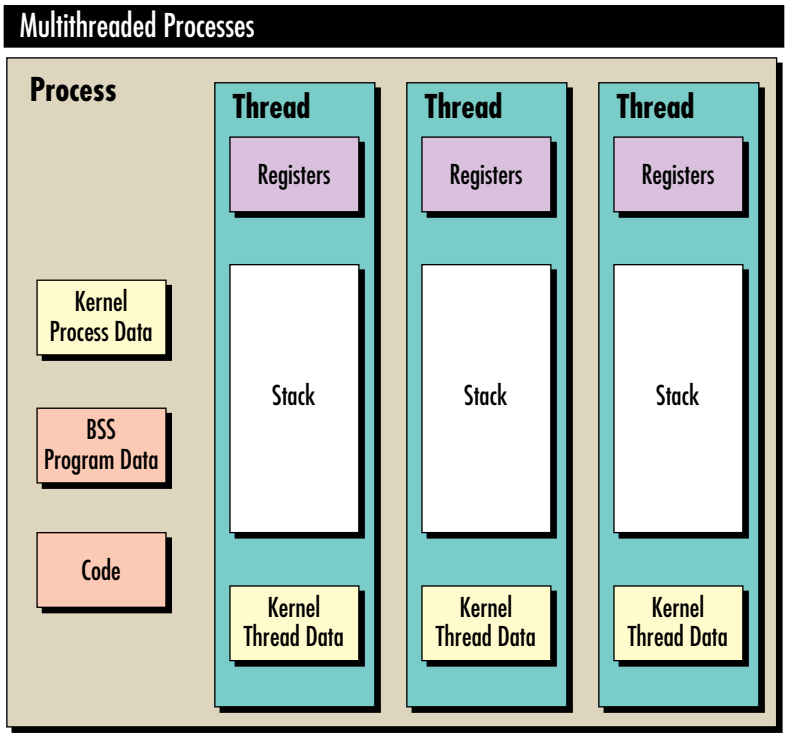


Figure 1. Multithreaded processes

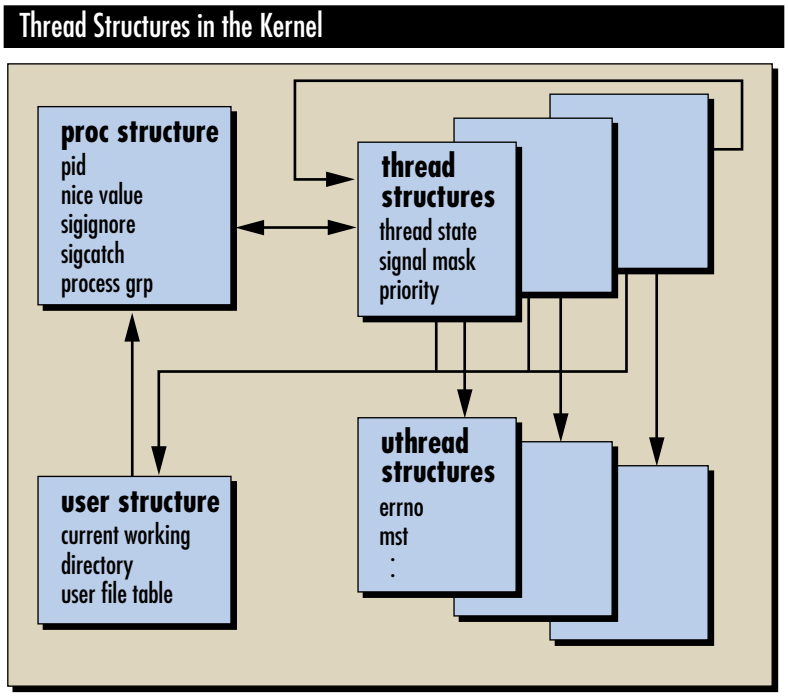


Figure 2. Thread structures in the kernel

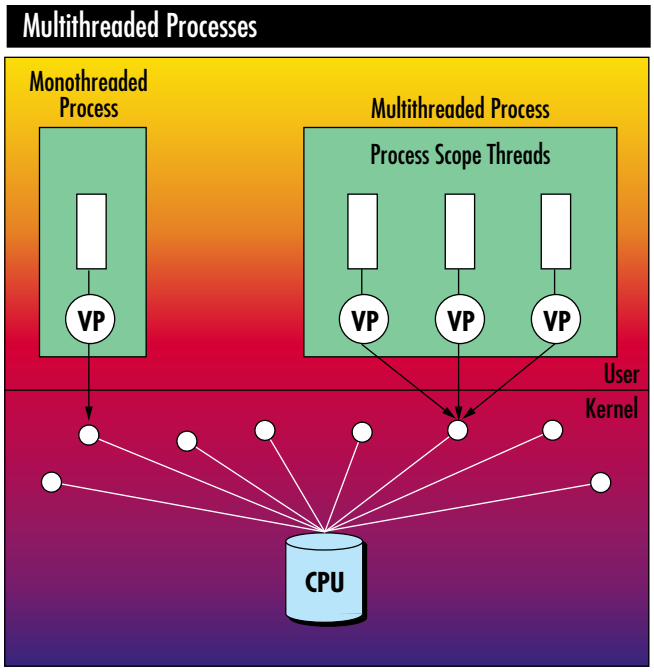


Figure 3. Multithreaded processes

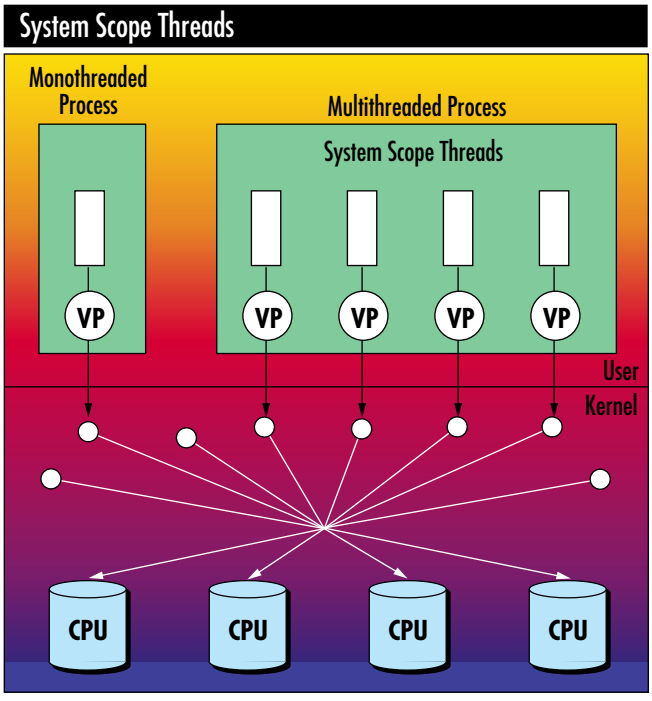


Figure 4. System scope threads

threads using System V signals. These kernel-independent threads are called *process scope* threads (because the scope of the thread does not extend beyond the process).

This model, shown in Figure 3, requires no changes to the kernel for threading to occur—which is a real advantage. Since all of the threads dispatching is handled in the user space, overhead is very low. There is one disadvantage, however. Since the threads dispatching requires the creative use of signals, the time slices are very large—approximately .1 second. These time slices may create problems in time-sensitive applications such as X-Windows applications.

The 1:1 Thread Architecture

AIX 4.1 implements the 1:1 thread architecture. In this model, every user space thread has a corresponding kernel space thread to support it. A virtual processor binds the user space thread to the kernel space thread. The one-to-one correspondence between user and kernel threads enables the kernel to know about each user thread. These permanently bound user threads are called *system scope* threads. All system scope threads contend with each other for resources, just as processes do in AIX Version 3 (see Figure 4).

The advantage of the 1:1 thread architecture is that each thread gets real-time scheduling. And in a symmetric multiprocessing environment, each thread can be run on a different processor to improve performance.

M:N Thread Architecture

The kernel and libraries in future releases of AIX Version 4 will support an M:N implementation, which combines the advantages of the M:1/DCE and 1:1 threading models. In this implementation, developers can use both system scope threads as well as user threads that are not permanently attached to kernel threads, as shown in Figure 5. Multiple user threads can be scheduled and multiplexed onto a smaller number of kernel threads. Since thread management is handled in the user space, overhead is reduced.

The advantage of this model is that developers can balance and control the application environment by using the type of thread ideally suited for the task. For example, developers can use system scope threads for foreground events that require real-time scheduling on the processor and process scope threads for background tasks that do not require immediate system resources.

Programming with Threads

Threads programming is similar to other types of parallel programming in that developers must create and manage the various tasks. However,

threads gives developers a higher level of control over the programming environment than processes do. Specifically, developers can control how threads pass information to each other, specify how a task gets scheduled, and assign a priority level to the task.

The following sections describe a few basic calls that developers can use to begin creating and managing threads. See `libpthreads.a` for more information about these library calls and others.

Creating and Managing Threads

Creating a thread is easier and more straightforward than forking a process. Only one single call is required to create a thread, whereas forking a process requires two calls with several possible variations. Use the `pthread_create()` call to create a thread, as shown in Figure 6.

When a thread is created, a thread ID (TID) is assigned to that thread. TIDs are similar, if not identical, to process IDs. The `pthread_create()` call returns the TID value at thread creation time. The developer can assign characteristics such as stack size, priority, and scheduling algorithm to a thread when it is created by creating and passing a thread attribute block to `pthread_create()`. If the developer specifies NULL, then the thread will be assigned the default behavior. The developer can also pass a `start_routine`, which is simply a pointer to the subroutine the thread should execute, and a pointer to the arguments to be passed to that routine.

Killing Threads

Since all threads are peer processes, any thread can kill any other thread. The `pthread_kill()` call, which is used to kill a thread, behaves very much like the `kill` system call.

```
int pthread_kill (thread, signal)
pthread_t thread;
int signal;
```

To make a thread terminate itself, use `pthread_exit()` with the status code. Remember that `pthread_exit()` terminates only the current thread, whereas `exit()` terminates the entire process, not just the currently executing thread.

```
void pthread_exit (status)
void *status;
```

Any other thread in the process can retrieve the status code from a terminated thread if it knows the terminated thread's TID. Use the `pthread_join()` routine to retrieve the terminated thread's status code:

```
int pthread_join (thread, status)
pthread_t thread;
void **status;.
```

If `pthread_join()` is called with a TID for a thread that is still running, `pthread_join()` blocks until the thread has terminated. The `pthread_join` call provides nearly the same function for threads that `wait()` provides for processes. The system does not free a thread's storage until `pthread_join` is called for that thread. Another difference between processes and threads is that when a process terminates, the parent process must call `wait()` to free the resources; but with threads, any sibling thread can perform this task.

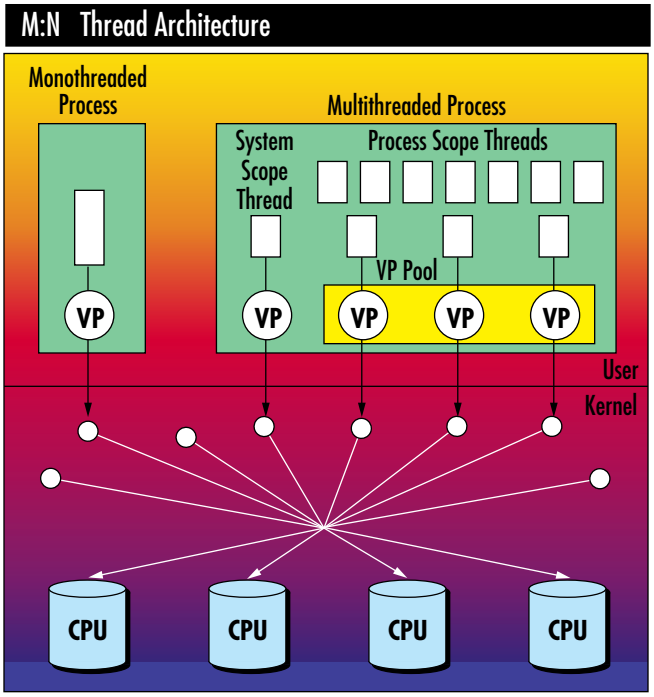


Figure 5. M:N thread architecture

```
int pthread_create (thread, attr,
start_routine, arg)
pthread_t *thread;
const pthread_attr_t *attr;
void *(*start_routine) (void *);
void *arg;
```

Figure 6. Using pthread_create() to create a thread

Threads Scheduling

In AIX Version 4, the thread is now the unit of work that the scheduler dispatches, so each thread can have a different scheduling priority. Threads also allows developers to specify the scheduling policy for each thread, providing a higher level of control over scheduling than processes do. There are three scheduling policies:

- **SCHED_FIFO**: Denotes fixed-priority first-in first-out (non-preemptive) scheduling
- **SCHED_RR:** Denotes fixed-priority round-robin (preemptive) scheduling
- **SCHED_OTHER**: Denotes the default AIX scheduling policy; this is the default value

Note that `SCHED_RR` is nearly identical to fixed-priority scheduling for processes on AIX Version 3. However, `SCHED_FIFO` now gives developers the choice of using a non-preemptive scheduling algorithm. This can be a valuable but dangerous tool. Both methods require root authority to run and should not be mixed on the same priority level.

Threads Communications and Synchronization

Two different mechanisms can be used to pass communication between threads: mutexes and condition variables.

A *mutex* is a mutual exclusion lock that protects data or other resources from concurrent access. When one thread holds the lock, no other threads can access the locked resource. When a thread tries to acquire a lock already held by another lock, it has two choices: to wait for the lock to be unlocked, or to return with a return code indicating that the lock is held by another thread. Use the `thread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` routines to handle these basic locking tasks.

Condition variables are powerful but complex tools. Condition variables allow threads to wait until some event or condition has occurred. A condition consists of three parts: a variable, usually boolean, that indicates a condition that can be tested; a mutex, which is used to serialize access to the variable; and a condition variable that causes a thread to wait until the condition is met. Although condition variables require more effort for the developer, they can be quite powerful.

To learn more about condition variables, read the section in InfoExplorer™ on "Using Condition Variables."

Conclusion

Threads programming in the UNIX environment is still relatively new, so the design, development, and debugging tools are still evolving. Nevertheless, the power and flexibility of threads demand a developer's consideration.

Marc Miller, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Miller works in the Austin AIX Executive Briefing Center, where he delivers presentations on future client/server directions, AIX, and DCE. He has a BS in Computer Science from Northwestern University.