AIX Version 4 Kernel Changes for SMP

By Thomas V. Weaver

The AIX Version 4 kernel contains a number of changes to support Symmetric Multiprocessor (SMP) systems. Two of the most significant are a new locking structure and threads, which provide parallelism and responsiveness to the AIX SMP systems. A kernel extension should be redesigned to use these changes to run efficiently on these systems.

When AIX Version 3 was being developed, system designers wanted to provide as much parallelism as possible on a uniprocessor system—that is, multiple processes sharing the CPU should each proceed with minimal interference. Therefore, they made the AIX Version 3 kernel preemptable so that processes executing a long-running system call can be preempted if a higher priority process becomes eligible to run.

Serialization

The preemptable kernel improves AIX's interactive response—it responds more briskly to short tasks, such as single key strokes. Being preemptable means that a program in the AIX Version 3 kernel must be designed to handle being interrupted at any time by any other program.

Disabling Interrupts

A program in the AIX Version 3 kernel must be designed to handle interrupts by any other program at any time—including another instance of itself.

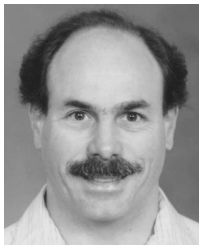
In practice, a kernel program will have some operations that must be *atomic*, meaning they run to completion without interruption. Actions taken by the kernel in response to a system call (a request for service from an application) might be interrupted. If a structure can be referenced both in a system call and while handling an interrupt, it is necessary to ensure that operations performed in the system call are atomic. This is done by disabling interrupts. The `i_disable()` kernel service causes the hardware to hold interrupts for a time, allowing an operation to run to completion. This service must be used carefully because disabling interrupts for long periods of time reduces the parallelism and responsiveness of the system.

Locking

Even if a structure is not referenced during interrupt handling, a process could be interrupted by reaching the end of its time slice, and a higher priority process could be dispatched, which would then reference that same structure. The AIX Version 3 kernel provides locking to allow processes that are not disabled to serialize their access to common structures. Because they are not disabled, the system remains responsive to new requests. If a process requests a lock and the lock is free, it continues without interruption. If the lock is currently held by another process, it will block, making it necessary to wait until the first process releases the lock.

The AIX Version 3 kernel protects many of its data structures with a single lock—the *kernel lock*. Its broad use means that it effectively locks code paths rather than individual data structures. The kernel lock is typically obtained when an operation starts. Even if the owning process is preempted, other processes attempting to get the kernel lock will be blocked. Eventually, the original process will be the highest priority process eligible to run and will complete its work. To improve the responsiveness of the system, the kernel lock is automatically released if a process voluntarily blocks—for example, when waiting for I/O to complete. It is automatically reacquired when the wait is over.

Just as it is possible to build a uniprocessor system in which the kernel processes each



Thomas V. Weaver

request to completion without interruption, it is also possible to build an SMP where the kernel runs on only one processor and processes each request without interruption. The AIX designers chose to provide a more responsive highly parallel system. The designers' attention to parallelism and locking mechanisms in the AIX Version 3 kernel made the work to support an SMP system easier; the individual paths that required protection against preemption were already identified. Much work, however, remained. The kernel lock had to be replaced with a large number of other locks with smaller scope to provide adequate parallelism.

AIX Kernel Support for SMP

The AIX kernel has the responsibility to provide both serialized and fast access to resources.

The individual processors in an SMP system run independently of each other. Any program, including those in the kernel, can run on any or all processors at once. An SMP system provides more throughput than a uniprocessor, but should not require users to do anything more to run existing programs. The AIX kernel has the responsibility to provide both serialized and fast access to resources.

New Locking Functions

A process should be able to access any object, such as a device, without interference by other processes working with other objects. That is, one process attempting to write to a file should not have to wait for another process' write to a terminal to complete before proceeding. Similarly, if two processes are working with the same object, they should expect AIX to make this work as well as it did on a uniprocessor.

This task is complicated because RISC System/6000 hardware disables interrupts only on the processor where the `i_disable()` function was called. Most places where the AIX Version 3 kernel used `i_disable` for serialization require additional locking on an SMP system. A significant part of making AIX Version 4 run on an SMP is providing and using enough locking to serialize access to objects, but not so much that the parallelism and throughput of AIX are impaired.

The AIX Version 4.1 kernel provides two new types of locks: simple and complex.

Simple Locks: These cause the requester to loop when trying to acquire the lock. These locks are used when it is not possible to wait or during operations that complete quickly, so there would be no advantage in dispatching another thread.

Simple locks must be combined with disabling interrupts if the lock can also be requested from an interrupt handler. If interrupts are not disabled and an interrupt came in on the processor that held the lock, and the interrupt handler attempted to request the lock, it would spin forever: it would never get the lock, but also never give up control to allow the holder of the lock to release it.

The `disable_lock` and `unlock_enable` kernel services combine disablement and locking (in the correct order). These replacements for `i_disable` and `i_enable` are used by components such as device drivers that could rely solely on disablement for serialization on a uniprocessor.

If the lock is not requested from an interrupt handler, disabling interrupts is not required. Then, the requester can wait after a fixed number of tries, or if the current holder of the lock is also waiting.

Complex Locks: These cause the requester to wait if the lock is not available. They also provide additional functions:

- A complex lock can be obtained in read instead of write mode. The difference is that there can be many simultaneous readers, but only a single writer. This is appropriate for data structures that are referenced more often than they are updated. Complex locks in read mode have limited use in the AIX 4.1 kernel, but will be expanded as performance testing reveals places where their use is appropriate.
- A complex lock can be obtained recursively; that is, it can be requested by a thread when the thread already holds the lock. This is a programming convenience for kernel programs or components that could be called either with or without a lock held. The locking function, rather than the program, will keep track of whether the lock needs to be acquired and released.

Complex locks are used by parts of the system, such as the higher levels of the filesystem, that need only serialize with other instances of themselves, and not with interrupt handler code. They are preferred over simple locks when the holder of the lock will generally block—such as waiting for I/O to complete. If the holder of the lock is likely to block, then any thread trying to get the lock might as well block also, if the lock is not available.

In addition to these locking functions, the AIX Version 4 kernel retains the kernel lock from Version 3, although it is rarely used by the kernel itself. It is available for kernel extensions that

may not need the new locking functions (such as those that do not plan to support MP systems) or where it may not be cost-effective to convert to the new locking functions.

The AIX Version 3 kernel used the kernel lock to lock code paths; Version 4 uses a larger number of locks to control access to specific structures. For example, individual device drivers can have locks that control access to their queues and other structures. This allows requests for two different devices to run in parallel on separate processors, which is critical for high performance. The AIX designers had to determine which locks protect which structures and the hierarchy of the locks (the order in which locks must be acquired). Generally, each component has its own specific locks for its private structures.

Atomic Operations

The heart of any SMP locking function must be an atomic, uninterruptible way of checking lock availability and acquiring the lock, if available. This is done on AIX Version 3 (as on most uniprocessor systems) by first disabling interrupts, then checking the contents of a word in memory. This is not sufficient on an SMP RISC System/6000, because the disable operation affects only the processor on which it was issued—the others continue to run. (The RISC System/6000, like many other platforms, provides a way for one processor to stop another. But this procedure is too slow and too disruptive to use for locking a high-performance system.)

The PowerPC Architecture provides special-purpose instructions that allow atomic update of a word in memory:

- **lwarx:** Loads a word from memory and establishes a reservation
- **stwcx:** Stores a word if the reservation is still present

The reservation is automatically removed if another processor updates the word in memory between the `lwarx` and `stwcx` instructions. This allows the locking code to determine if an attempt to acquire a lock failed because another processor got to it first.[1]

The kernel uses the `lwarx` and `stwcx` instructions to provide the basis of locking and for other operations. There are kernel services that provide `fetch_and_or`, `fetch_and_and`, and `compare_and_swap` functions. The first two allow atomic "and" or "or" operations to a word in memory. The last stores a new value to a word in memory, if a known value is still there.

The advantage of these atomic operations is their speed. A bit in a flags field can be turned on or off, or a structure can be added to a linked list with no software locking. Their disadvantage is they are operating only on a single word of memory. Coordinated updates to larger structures, such as the pointers in a doubly linked list, still require locking. In many instances in the kernel, structures have some sections that are updated only under a specific lock and others that are updated only through atomic operations. The actual choice depends on where a particular component can use atomic operations to avoid getting a lock.

The heart of any SMP locking function must be an atomic, uninterruptible way of checking lock availability and acquiring the lock.

Performance Considerations for Locks

Good performance is difficult to maintain when an operating system is changed from a uniprocessor platform to support many processors. For example, the additional locking required on a multiprocessor system adds unnecessary path length on a uniprocessor system. To deal with this, the AIX kernel is built in two versions: a uniprocessor and a multiprocessor version. The appropriate version is automatically chosen at install time. Preprocessor directives are used to avoid compiling spin locks into the uniprocessor version.

Experience shows that good performance on a multiprocessor requires that individual locks be held no longer than necessary, and that contention for these locks be kept to a minimum. There are, however, difficult trade-offs: the finer the granularity of the locking, the smaller the set of system structures that each lock controls. The less each lock is held, the more locks must be held and the more time spent acquiring and releasing locks to perform each function.

AIX Version 4 has built-in tools to measure lock usage, hold time, and contention. These tools can help identify the cause of performance problems. If the kernel is created with the `-L` option of the `bosboot` command, lock instrumentation is built into the system. Information on locks can then be recovered with the `lockstat` command. Since collecting information on locks

[1]See *PowerPC Architecture* (SR28-5124) for a complete discussion of these instructions, storage coherence, and other related issues. While these are important issues, they are confined to small portions of the kernel that deal with hardware dependencies.

Address Space Layout

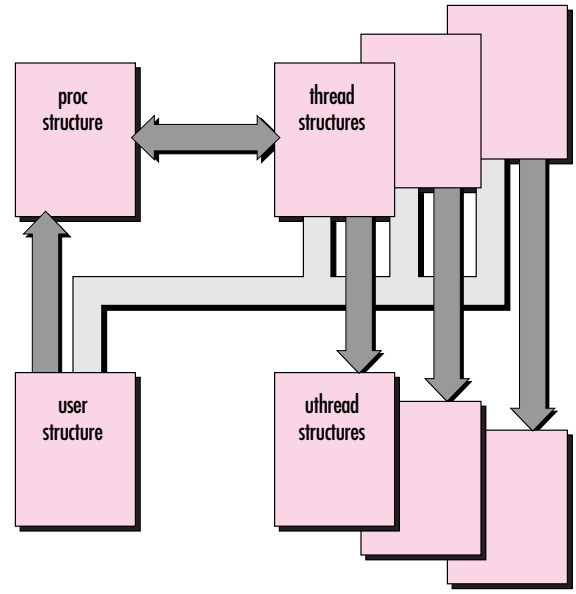


Figure 1. Address space layout

adds extra path length and some performance costs, this option must be explicitly selected.

Kernel Support of Threads

Most application programs are unaffected by the SMP support in AIX Version 4. For example, a program such as `vi` makes no special use of multiple processors, and end users expect it to function without change on SMP systems. However, there are programs that could make good use of the intrinsic parallelism provided by an SMP system. A good example of this is a transaction processor, which allows multiple access to a common database. The AIX Version 4 kernel provides threads to allow applications access to multiple processors.

In a traditional UNIX system, the *process* is the dispatchable unit. It contains a single flow of instructions—executing one program at a time. The process also serves as a *resource collection point*. For example, a process opens files, owns file descriptors, and automatically closes files when it terminates.

AIX Version 4 separates the concepts of "dispatchable unit" and "resource collection point" of the traditional UNIX process. The process remains a resource collection point; however, the *thread* becomes the dispatchable unit. A traditional UNIX process is a single dispatchable unit. A process in AIX Version 4 has a single thread; additional threads must be requested.

In a multithreaded process, all threads share the process' resources. For example, they all exist in the same address space and have access to all the files opened by the process. Threads allow an application access to the parallelism of the underlying SMP system; that is, they allow an application to overlap different sections of processing.

AIX Version 4 was designed to provide support for the POSIX 1003.4a Draft 7 proposed standard for threads, commonly referred to as *pthreads* (see "Introduction to Multithreaded Programming" in this issue for information about how applications use threads). Most functions in the standard are provided by the pthreads library; the kernel supports basic threads functions on which `libpthreads` is built.

In AIX Version 3, the information kept by the kernel about a process is split between a process structure or `proc` block and a user structure or `u` block. The `proc` block is never paged out, so it is used to hold information that must be referenced without a page fault. The `u` block holds information that is not needed when a process is swapped out. In AIX Version 4, much of this information is kept on a per-thread rather than per-process basis. There is a `thread` structure that is never paged and a `uthread` structure that is pageable. The `thread` structure holds state information about the thread that might be accessed by other threads, such as its priority and whether it was waiting for an event. The `uthread` structure contains information that is not referenced unless that thread is running, as shown in Figure 1.

A single-threaded process in AIX Version 4 behaves like a traditional UNIX process, and the distinction between thread and process is small. When multithreaded processes are considered, several important differences must be handled by kernel programs:

- Because the thread—not the process—is the dispatchable unit, all synchronization primitives such as sleep, wakeup, wait, and post are thread-based rather than process-based.
- Signals must be dealt with on both the thread and process levels. Signal handlers are defined at the process level; for example, a given process has, at one time, only one handler for `SIGSEGV`. Synchronous signals are delivered to the thread that created the condition; for example, if a thread uses an invalid pointer, the `SIGSEGV` is delivered to that thread. A signal sent to a process, for example, by the `kill`

command, is delivered to the first available thread. It is also possible to send a signal to a specific thread.

In both cases, any other threads in the process continue unimpeded. This means that a kernel program can no longer expect these functions to completely control a process. For example, in AIX Version 3, a kernel extension could reasonably expect that a data area associated with a process would be untouched by the process while it waited for I/O to complete. In AIX Version 4, the same kernel program must employ another mechanism—typically locking—to obtain the same results.

New Threads Services

A series of new kernel services is associated with threads. These fall in three categories:

- **Synchronization primitives** that allow a thread to sleep or wait for an event. There are also services that wake up a sleeping thread or notify it that an event has occurred.
- **Services to create, terminate, and send signals to threads**. These are analogous to the equivalent process-level services.
- **Services to set the state (instruction counter, registers) and scheduling policy of a thread**. These have no equivalent in AIX Version 3. They were added to allow the `libpthreads` functions more direct and efficient control over threads. They allow for the possibility of an application-level scheduler multiplexing many user-level threads onto a smaller number of kernel threads.

AIX Version 4 adds two new scheduling policies that may be selected by authorized threads:

- **Round robin**: Threads remain at a fixed priority level and run until they either voluntarily release the CPU or reach the end of their time slice.
- **First In First Out (FIFO)**: A thread runs until it voluntarily releases the CPU; it is never time-sliced out.

These facilities were added to support the POSIX threads draft standard. They must be used judiciously to avoid degrading overall system performance.

Kernel Extensions

The AIX Version 4 kernel continues to support device drivers written by customers and other software vendors. Since those written for AIX Version 3 are unlikely to be quickly redesigned for an SMP system, the kernel supports uniprocessor-oriented device drivers through the following procedure called *funneling*:

- During device driver configuration, a device driver must indicate that it is safe to run on an MP system. The default is that the device driver must be funneled.
- There is a kernel service that allows a thread to be bound to a specific processor—the thread will not be dispatched on any other processor. When an outbound request is issued to a funneled device driver, the current thread is temporarily bound to processor 0. At the completion of the outbound request, the thread in unbound and becomes available to run on any processor.
- When an inbound interrupt arrives from this device on a processor other than processor 0, the lowest level interrupt handler uses a hardware facility to send an interrupt to processor 0. Completion handling for the original interrupt then continues on processor 0.

Funneling allows a uniprocessor device driver to function on an AIX Version 4 system without redesign. Since the device driver runs only on processor 0, the device driver can continue to use `i_disable/i_enable` for serialization. This mechanism is also used by AIX Version 4 for device drivers, such as diskette and CD-ROM, where the expected low usage and low speed of the device does not warrant redesigning the device driver to use locking.

The AIX Version 4 kernel continues to support device drivers written by customers and other software vendors.

Thomas V. Weaver, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: tweaver@austin.ibm.com. Mr. Weaver, a senior programmer with the AIX kernel development group, has been working with AIX since 1988. He has worked in kernel architecture and design, and clustering and system RAS. He has an MS in Mathematics from the California Institute of Technology in Pasadena.*