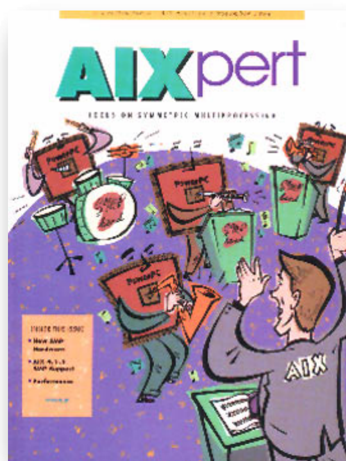
AIXpert DIGITAL VERSION IBM

TABLE OF CONTENTS



Commentary

We Be Jammin'
By George Noren

AIX

SMP Overview
By Debora Blakely-Fogel

Introduction to Multithreaded Programming
By Chary G. Tamirisa

Porting Applications to the AIX 4.1 OS SMP Environment
By Debora Blakely-Fogel

AIX Operating System SMP Performance
By William Alexander, Robert Dimpsey, and Bret R. Olszewski

AIX Version 4 Kernel Changes for SMP
By Thomas V. Weaver

DCE Performance Study
By Bob Russell

Print Spooling Subsystem in AIX 4.1
By Eddie Ho and Ravi Mandava

Object-Oriented Programming

OO Programming Utilized by Rochester Lab
By Bill Berg, Chris Jones, and Mike Tomashek

Q&A

AIX Questions
Compiled by Dave Franks

NOVEMBER 1994

We Be *Jammin'*

During my academic career, I was lucky enough to be a member of a small rock group. We never made the "Top 40", but we had our share of gigs. Although my fingers have long ago forgotten how to trace the riffs and runs on the fretboard, I can still remember the degree of commitment and discipline that was required to coax excellent music from a disparate group of independent-minded musicians. In this issue, we hear from a group with a higher degree of commitment and discipline, those wizards who make the PowerPC™ chip play in concert with other PowerPCs in the new Symmetric Multiprocessor (SMP) products announced in October.

We have two articles to help you understand the issues behind the design of an SMP. "SMP Overview" discusses the types of multiprocessor design and looks at the specific issues of designing to an SMP architecture. "AIX® SMP Performance" takes that discussion to a deeper level to illustrate how AIX wrings the best performance out of an SMP design.

OK, it's hot! So how do you use it? If you're just getting your feet wet in the MP pond, be sure to check out "Introduction to Multithreaded Programming". Skillful management of threads is key to designing an efficient MP-enabled application. If you already have an application and are wondering what it will be like in an MP environment, read the article "Porting to an SMP Environment". And if you are thinking of adding an extension to the new SMP-enabled AIX kernel, don't miss "AIX Version 4 Kernel Changes for SMP".

Moving away from SMP, we have an excellent case study on DCE performance in a heterogeneous environment. If you are running DCE, or plan to in the future, this article provides many valuable insights and conclusions. Administrators will also be interested in the print spooling article that explains the many improvements to the printer subsystem that have been made for AIX 4.1. And, if you are about to begin a new programming project and are considering taking the leap to Object-Oriented Programming (OOP), be sure to read the report from a recent OOP effort at IBM's Rochester laboratory.

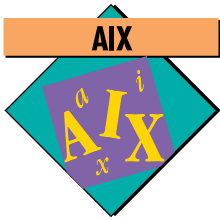
Requalification

Some of you have received your magazine with a request that you update the information that we have on file for you. We must reduce our costs by eliminating addresses that end up with the magazine in the dead letter office. Please take a few minutes to fill out that form and return it. We do not want you to miss a single issue of *AIXpert!*

George Noren

George Noren, IBM Corporation, Internal Zip 2830, 11400 Burnet Road, Austin, TX 78758. Internet: geo@austin.ibm.com. Since joining IBM in September 1979, Mr. Noren has written manuals for System/34, System/36™, and AIX on both the RT® and RISC System/6000® platforms, and was a member of the InfoExplorer™ design team. He has also worked as system administrator for several AIX server machines and their clients, and is currently responsible for the Prototype Evaluation Labs in Austin. Mr. Noren studied engineering at Illinois Institute of Technology, holds a BA in English from the University of Minnesota and an MBA from St. Edwards University in Austin.

George Noren



SMP Overview

By Debora Blakely-Fogel

AIX 4.1.1 operating system, a new release of AIX, supports Symmetric Multiprocessor (SMP) systems based on the PowerPC family of processors. This article defines and provides an overview of SMP systems.

Many types of multiprocessors are being used today. A Symmetric Multiprocessor (SMP) is unique in that the system looks exactly the same to each processor in the system. The following is a brief discussion of three Multiprocessor (MP) technologies supported by the AIX operating system.

Shared Memory MP: Figure 1 shows a symmetric multiprocessor configuration. A symmetric multiprocessor, also known as a *shared memory* multiprocessor, has multiple processors that can each address all memory and all devices. User processes running on any processor see the full machine. If two or more processors access the same word in memory, the hardware keeps the caches consistent—invisible to application processes.

Compared to other types, the advantage of SMPs is their use of the same programming model as uniprocessors. Most existing applications written for uniprocessor POWER platforms will run unchanged on an SMP running AIX 4.1.1.

Shared Nothing MP: Figure 2 shows a *shared nothing* multiprocessor or distributed memory configuration. All processors have their own memory and disks. Uniprocessor programs must be changed to run on this configuration because they must pass messages across an interconnect in order to use the multiple processors. SP2™ is an example of a shared nothing multiprocessor that runs AIX.

Shared nothing MPs generally scale better than SMPs because they have no memory bus contention and no cache coherency problems among the processors. However, the changed programming model often outweighs the advantages of this type of MP.

Shared Disk MP: Figure 3 shows a *shared disk* multiprocessor configuration in which processors share only disks. Unlike the SMP, each processor on a shared disk multiprocessor has its own memory. The Clustered Multiprocessor (CMP) portion of HACMP/6000 software allows users to configure RISC System/6000® machines in a shared disk multiprocessor configuration.

The shared disk MP, like the shared nothing MP, has no memory bus contention or cache coherency problem among the processors. However, a centralized locking scheme is used to control access to the disks. This locking scheme requires change to some applications (such as databases) and generally offsets the performance advantages of no memory bus contention or the cache coherency problem.

Scaling

One of the most important metrics of MP performance is scaling. When a processor is added to the system, how much additional performance is obtained on a given workload? Scaling is workload-dependent; some workloads will scale better than others. In addition, workloads will scale differently on different styles of multiprocessors. For example, a workload that shares a lot of data is likely to scale better on an SMP than on a shared nothing MP. This is because all processes on an SMP have a consistent view of the data, and processes on a shared nothing cluster must do message passing to share data.

To help explain this, Figure 4 shows a hypothetical graph of SMP scaling. In a perfect world,

Debora Blakely-Fogel

one would expect performance to increase linearly as processors are added. For example, if a processor is added to an existing two-processor complex, one might expect a 50% increase in performance. However, this does not happen because of the overhead required to maintain a consistent view of the memory and other shared resources for each of the processors. Generally, each additional processor increases performance by slightly less than the previously added processor. In fact, on all SMPs for all real workloads, adding more processors after some critical number ceases to boost performance, and actually decreases throughput. This happens at the point where the cost to maintain a consistent view of memory and other shared resources becomes greater than the processing power provided by a processor.

The greatest strength of an SMP is that it looks just like a uniprocessor to each process running on each processor. Maintaining this illusion is difficult for both the hardware and the Operating System (OS), and the costs of doing so are the root of all limitations of SMPs. This is because the system must do a significant amount of work to maintain a consistent view of memory for all the processors without having a significant impact on performance. There are usually two critical factors limiting SMP scaling:

- **Contention for the bus or switch that connects all the processors to the shared memory**. This may cause workloads with high cache miss rates to scale poorly.
- **Contention for shared words in memory**. This is viewed as contention for the locks that serialize access to these words and help keep the data coherent. Poor scaling may result for workloads that share data between processes or that spend much time in the kernel.

The SMP Cache Consistency Problem

The most basic problem that all SMPs must deal with is SMP cache consistency. Figure 5 shows an example of the problem. Suppose process p1 is running on processor 1 and process p2 is running on processor 2. Suppose also that processes p1 and p2 are working together on a problem and sharing some memory. Consider the following sequence of events:

1. Process p1 loads address 123, which contains the character “a”.

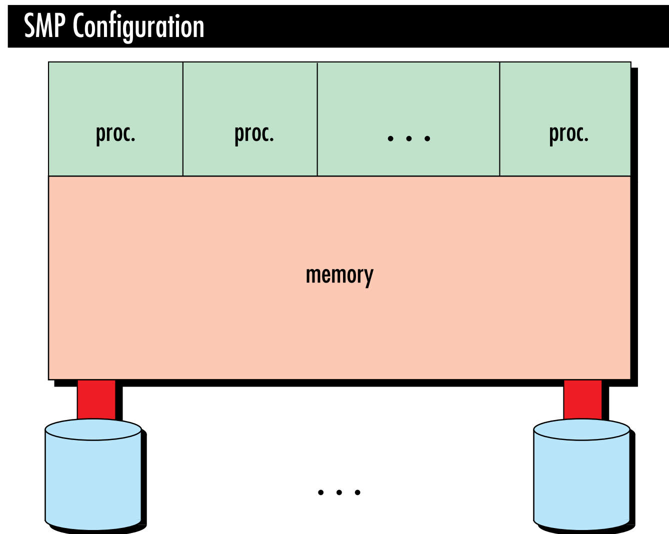


Figure 1. Symmetric multiprocessor configuration

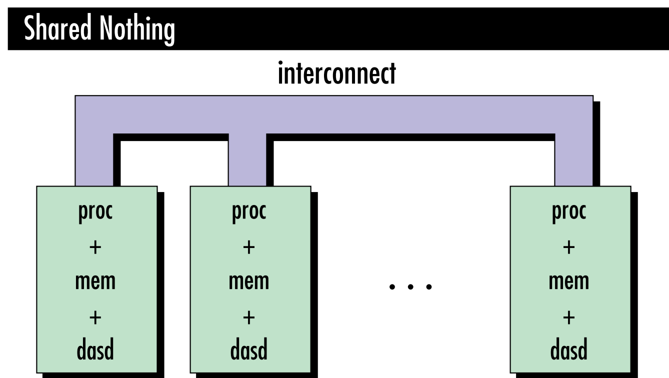


Figure 2. Shared nothing MP cluster

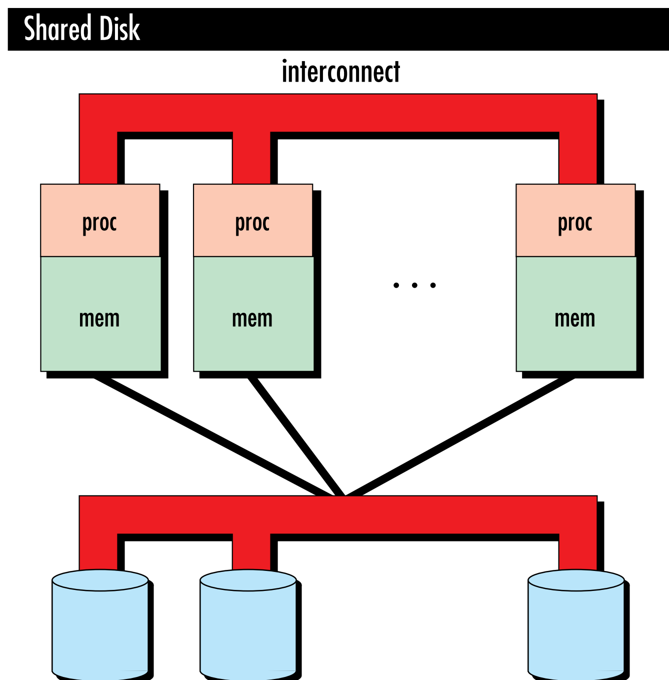


Figure 3. Shared disk MP cluster

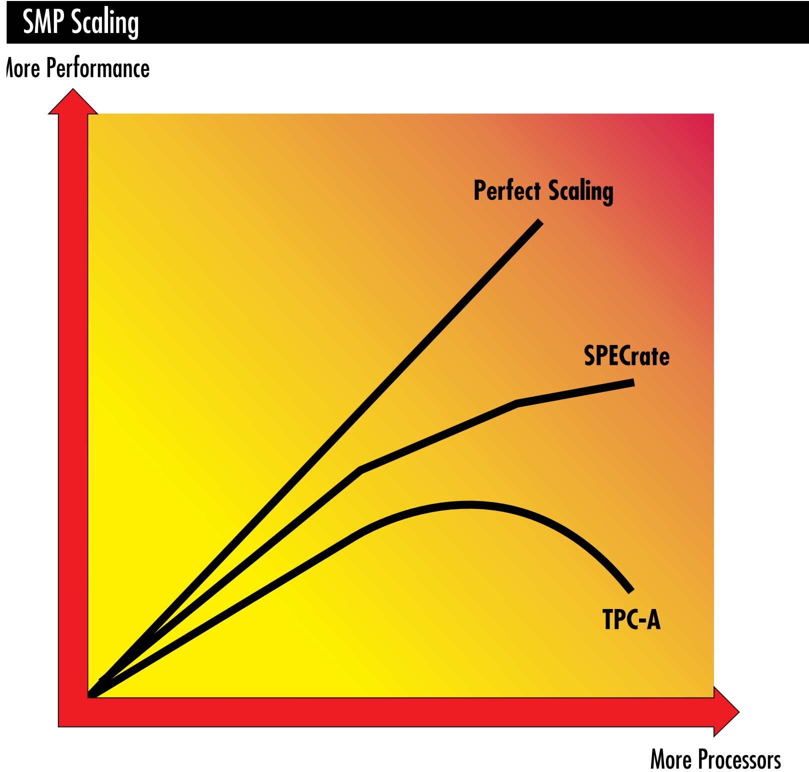


Figure 4. Hypothetical SMP scaling

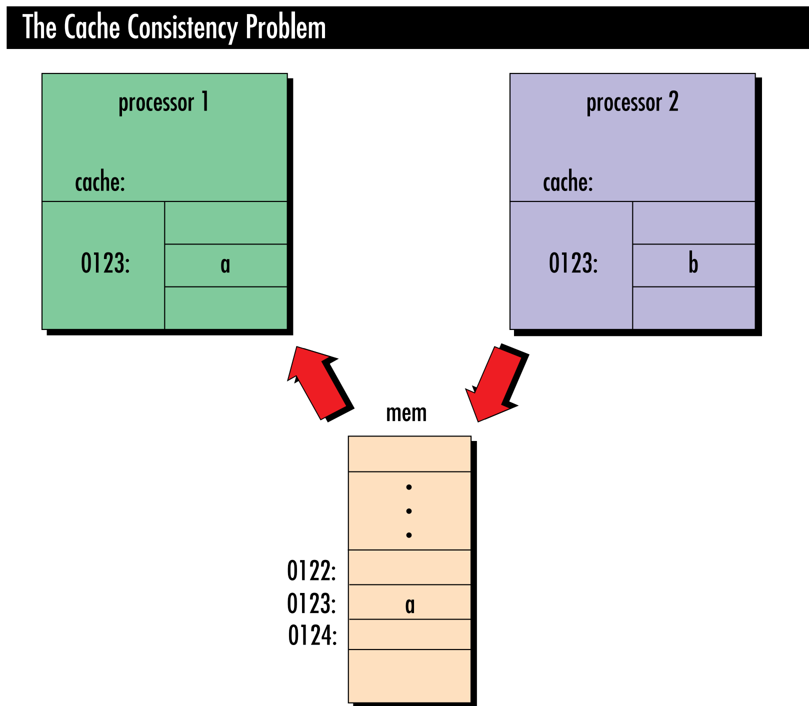


Figure 5. SMP cache consistency problem

2. Process p2 stores the character "b" into address 123.

3. Process p1 loads address 123 again.

The value seen by process p1 at step 3 is very important. With a naive implementation, p1 sees "a" because it has a copy of address 123 in its cache, process p2's load request never goes out to memory, and p1 does not see the new value "b" that process p2 placed there.

As shown in Figure 6, logic at each processor-bus interface broadcasts a message over the bus each time a word in its cache is changed. The logic also snoops on the bus for such messages from other processors. Whenever it detects that another processor changed the value at an address that is copied in its own cache, the snooping logic invalidates that entry in its cache. This *cross invalidate* reminds the processor that the value in that location in the cache is invalid, and it must look somewhere else for the correct value.

In this example, when process p1 reads address 123 the second time, it gets a cache miss and must look elsewhere for the value. The extra snooping logic determines where process p1 should look to get the proper value for address 123. If the new value has been written to memory, process p1 will obtain the value from memory. If the new value has not yet been written to memory, process p1 will get it from processor 2's cache.

Since cross invalidates increase cache misses and the snooping protocol adds to the bus traffic, solving the cache consistency problem reduces the performance and scalability of all SMPs.

False Sharing

The unit of access in a cache is called a *line*. A typical cache line on RISC System/6000 machines is 32 bytes or eight words. It is possible for two processes to reference two different portions of data that fall in the same cache line because they lie close to each other in memory.

In the example in Figure 7, if the process on processor 1 changes the value of d1, the cache consistency logic will invalidate processor 2's cache line causing a cache miss when d2 is accessed, even though the two processes were not sharing any data. This is called *false sharing*.

False sharing increases cache misses and bus traffic, further reducing SMP throughput and scaling.

Locks

One of the well-known problems of SMPs is maintaining data consistency. Consider the classic example of a doubly linked list. Removing an element from the list requires updating two pointers.

- The forward link of the element preceding the element that is being removed
- The backward link of the element following the element that is being removed

When only one of the pointers has been updated, the list is in an inconsistent state. If two processors in a multiprocessor system add or remove elements from the same list simultaneously, the result can be unpredictable.

The Problem: Data Integrity

If multiple processes share a piece of data and if two or more of them try to update it almost simultaneously, the result can be incorrect or incoherent. For example:

- Two processes each add 1 to a shared counter. If the beginning value is x, the correct final value must be x+2. However, if both processes increment the counter simultaneously, the result will be x+1.
- Two processes each try to add an element to a linked list. If both processes try to add an element simultaneously, the result is unpredictable.

A *critical section* is a section of code that modifies shared data, and therefore must not be executed by more than one process at a time. One process must be made to wait until the other process has finished its critical section. The two critical sections must be serialized.

The problem of serializing access to shared data is generic to parallelized code. It occurs both in the kernel and in parallelized user applications. It is especially critical in the kernel because most kernel data must be protected. The kernel performs services on demand, so any processor can be executing any kernel component at any time.

This problem also arises in a milder form in current, uniprocessor, multiprogrammed versions of AIX because of interrupts and explicitly shared data. A kernel component could be in the middle of updating some data when an interrupt occurs. A second instance of the same component might run on behalf of a higher priority thread or

The Cache Consistency Solution

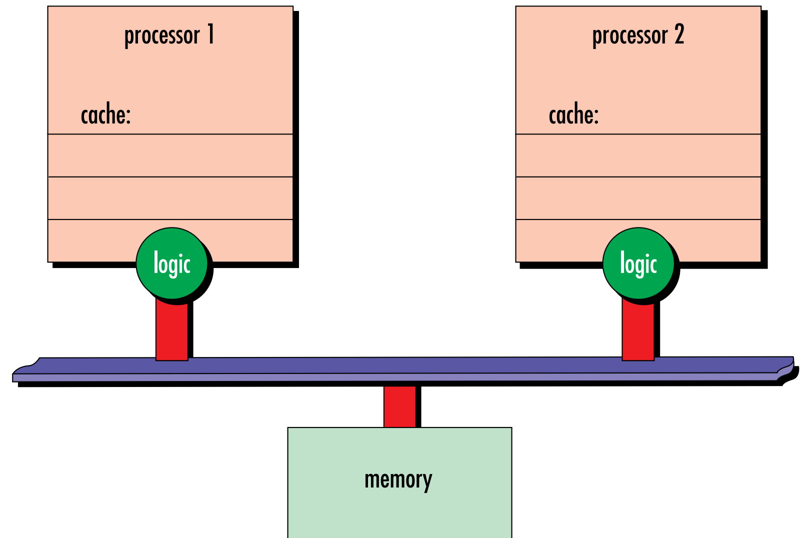


Figure 6. SMP cache consistency solution

False Sharing

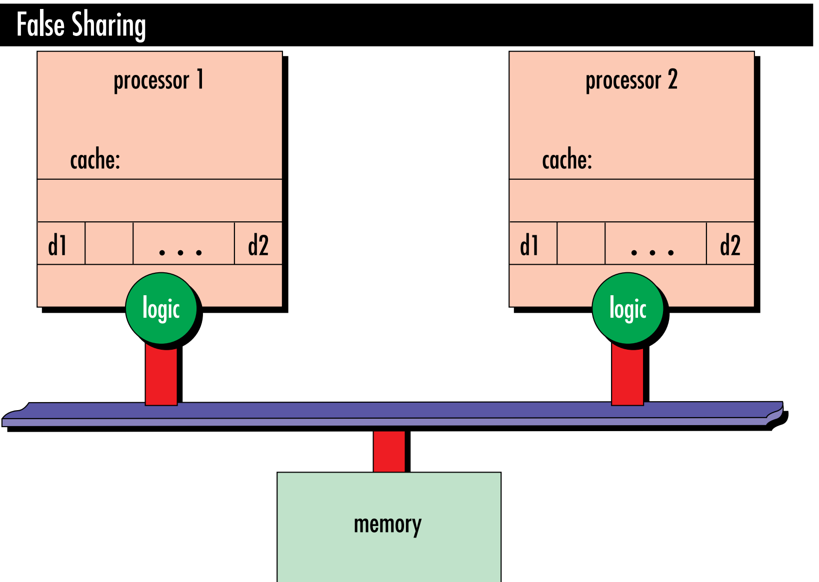


Figure 7. False sharing

process and try to update the same data. On a uniprocessor, this problem can be solved by carefully enabling and disabling interrupts. However, disabling interrupts is not sufficient on an SMP, because interrupts can only be disabled on the processor on which the thread or process is running. If a kernel component is updating some

data with interrupts disabled on its processor, a second instance of the same component can run on another processor and try to update the same data.

The Solution: Locks

Conceptually, a lock is just a bit in memory that processes use to regulate their entry into critical sections. If a process wants to enter a critical section, it examines the corresponding lock. If the bit is off, the process turns it on, then enters the critical section. If the bit is on, the process waits until it is off before entering. Each process must reset the bit when leaving the critical section.

Locks are not quite so simple to implement because two or more processes could test the same lock simultaneously, determine that the lock is available, and enter the critical section. Because taking a lock requires several operations (read, test, and set the lock bit), this operation is itself a critical section. Therefore, multiprocessor hardware must provide a way to perform this test-and-set operation atomically with respect to the other processors. This means that if more than one processor is trying to obtain the same lock simultaneously, exactly one of them will succeed. All MP locking primitives use this kind of atomic operation as their basic building block.

AIX developers can choose between two types of locks: mutually exclusive simple locks and complex read-write locks.

Fortunately, application developers do not have to implement their own locks. Locking services are provided by AIX for the following types of synchronization:

- **Kernel thread <—> kernel thread**: AIX provides the `simple_lock()`, `simple_unlock()`, `lock_read()`, `lock_write()`, and `lock_done()` services.
- **User process <—> user process**: AIX provides the `msem_lock()` and `msem_unlock()` services.
- **User thread <—> user thread**: The pthreads library provides the `sem_wait()`, `sem_post()`, `pthread_mutex_lock()`, and `pthread_mutex_unlock()` services.

Different Types of Locks

To execute correctly on an SMP, the system must be Multiprocessor Safe (MP Safe). This means that every subsystem and application that accesses shared or global data must have a locking strategy in place that maintains data consistency.

Mutex Versus Read-Write Locks

The locks that have been discussed so far are mutually exclusive or *mutex locks*. They allow one process or thread at a time into a critical section.

If a piece of shared data is read-mostly, it makes sense to distinguish between the many processes that only want to look at or read the data but not change it, and the few processes that want to change or write the data. A read-write lock allows multiple readers into the critical section at once, but guarantees mutual exclusion for writers.

AIX has both types, and developers must choose the type most appropriate for the situation.

Spin Versus Blocking

When a process wants a lock already owned by another process, the process has to wait. A *spin lock* allows the waiting process to keep its processor, repeatedly checking the lock bit in a tight loop (spin) until the lock becomes available. Spin locks are useful for locks that are held only for very short times. A *blocking lock* suspends the process until the lock is free and then puts it back on the run queue. This is suitable for locks that may be held for longer periods of time.

AIX developers can choose between two types of locks: mutually exclusive simple locks that allow the process to spin while waiting for the lock to become available, and complex read-write locks that block the process while waiting for the lock to become available.

The rules about using locks are strictly conventions. Neither hardware nor software has an enforcement or checking mechanism. Although using locks has made AIX 4.1 MP Safe, it is the developer's responsibility to define and implement an appropriate locking strategy to protect their own global data.

Waiting for Locks

Figure 8 shows an application that causes the system to spend 10% of its time in a certain kernel component. Suppose that because of the complexity of that component, the developer decides to make the whole component one large critical section. That is, there is only one mutex lock for the whole component, and it is requested at all entry points in the component and released at all exit points. On a 4-way SMP, this mutex lock will be busy 40% (4 x 10%) of the time.

According to queuing theory, the busier a resource, the longer the average wait to get it. The relationship is non-linear—if the use of a lock is doubled, the average wait time for that lock more than doubles. In addition, if the use of the same lock were halved, the average wait time for that lock will be reduced by more than half.

Waiting always decreases system performance no matter how it is done. If a spin lock is used, the processor is busy but not doing useful work (not contributing to throughput). If a blocking lock is used, the overhead of context switching and dispatching and the consequent increase in cache misses is incurred.

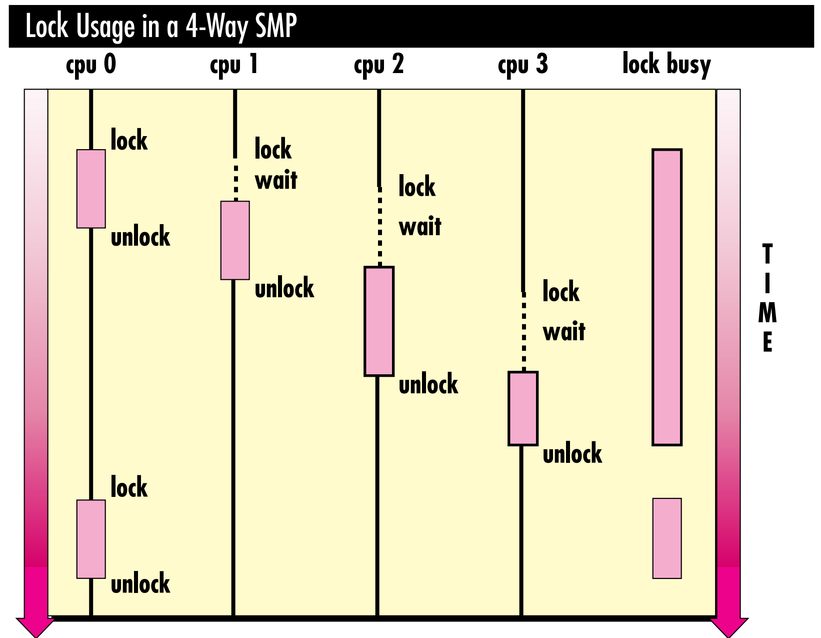


Figure 8. Lock usage in a 4-way SMP

Lock Granularity

The amount of time a given lock is busy is a function of how often it is requested and how long it is held once acquired. One of the most effective ways to reduce lock wait time is to reduce the size of what the lock is protecting: its granularity.

Figure 9 shows that lock holding time can be reduced by ensuring that it encompasses only the code that accesses shared data instead of all the code in a component. Locks should always be associated with specific data items or structures, not with components or routines.

When serializing access to large data structures, consider an array of locks, one for each element of the structure instead of one lock for the whole structure. This will reduce the frequency for which any one of the locks is requested.

Unfortunately, there is a counter-effect to reducing lock granularity. In the example in Figure 9, we suggest requesting and releasing a lock three times instead of once. In the best case, when a simple lock is free and we do not have to spin, each lock-unlock pair costs approximately 20 instructions. Therefore, in the example, we have added a minimum of approximately 40 instructions to our path length. Complex locks are even more expensive. Each uncontested lock-unlock pair costs approximately 125 instructions, and using them in our example would add approximately 250 instructions to our path length.

Figure 10 shows that if lock granularity is too fine, too many instructions are used to request and release the locks. If lock granularity is too coarse, too much time is spent waiting for the locks. The proper balance in each component must be discovered empirically. As an initial guideline for AIX 4.1, try to hold locks for no more than approximately 300 instructions.

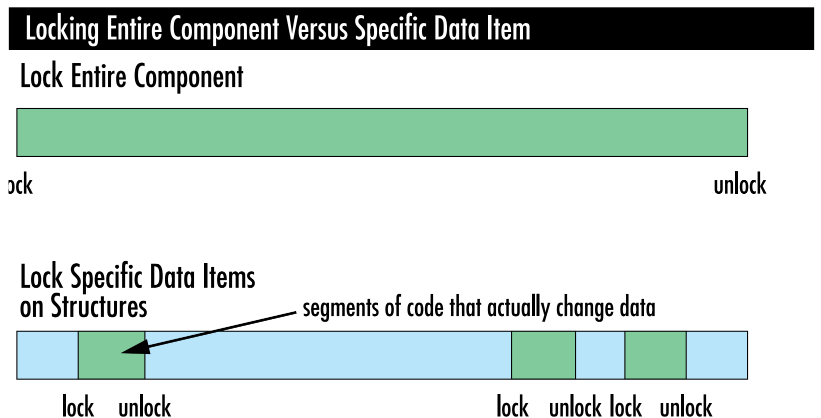


Figure 9. Comparison of locking an entire component versus a specific data item or structure

Performance Tips for Locks

The following are some performance tips for locks:

- Never perform synchronous I/O or any other blocking activity while holding a lock.
- Move all unnecessary instructions (those not directly related to reading or modifying the protected data) outside the critical section.
- If there is more than one access to the same data in a given component, try to move the accesses together so they can be covered by one lock-unlock pair (provided the combined code is under the 300 instruction limit).

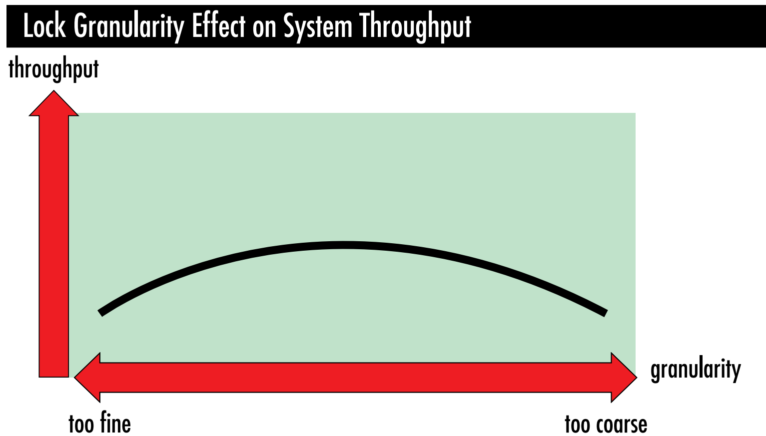


Figure 10. The effect of lock granularity on system throughput

- Do not put two locks in the same cache line. In fact, to reduce false sharing, do not put anything in the same cache line with a lock except data that it is protecting.
- If more than one lock must be held simultaneously, request the busiest one last (if allowed by the hierarchy).
- Avoid double wakeup. If some data must be modified under a lock and another process must be notified when the modification is completed, release the lock and then post the wakeup.
- If the protected data is read-mostly, consider using a complex lock instead of a simple one. However, be sure the extra path length in the normal, uncontested case will be justified.
- Start out with medium- to coarse-grained locks and reduce granularity only if improved performance is needed.

Lock Tuning in an Application

MP performance tuning is very difficult. It should be attempted only if the routine is used frequently and its performance is not satisfactory. Here are the steps to follow when tuning the locks in an application:

1. Use profiling to find where the routine is spending its time.

2. Within these areas, find the locks with the longest wait times. Both kinds of kernel locks are instrumented. When lock instrumentation is turned on, two counters are kept for each lock. The counters indicate how often the lock was used and the lock's wait times.

3. For each of these locks, apply the tuning tips above.

4. Repeat these steps until performance is satisfactory.

Processor Affinity

There is a single, common run queue in AIX 4.1 accessed by all processors. When the thread currently running on a processor blocks or its quantum expires, the highest priority thread that can be run will be dispatched on that processor.

One consequence of this dispatching policy is that over its lifetime a thread bounces around, running first on one processor and then another. Each time it starts executing on a different processor, it suffers cache misses until it has brought its own instructions and data into the cache. If the thread has not been blocked long, some of its instructions and data may still be in the cache of the last processor on which it ran. In this case, it would be more efficient to run the thread there again instead of on a different processor.

The policy of trying to run a thread on the same processor as it ran last is called *processor affinity*. AIX 4.1 has only a very weak form of processor affinity: if two or more threads on the run queue are tied for highest priority, a processor will choose the one that ran on it most recently.

Binding

The strongest form of processor affinity is to “bind” a process or thread to one processor and never run it on any other processor, even if its processor is busy and another is idle. The `bindprocess()` system call allows an application to do this. This might be useful for a process that seldom blocks for long periods and whose response time is important. Binding a process to one processor may help that process, but it can hurt overall system throughput because it sometimes forces a processor to remain idle when it could be doing useful work. Binding is appropriate only in special circumstances, such as on a system that is dedicated to a single application.

Two Ways to Parallelize

MPs are useful only if performance improvements can be realized by processes using an MP instead of a uniprocessor. In an environment where many applications or processors execute independently of each other, performance improvements are realized by allowing the multiple applications to execute simultaneously on separate processors. However, in computing environments dominated by a single application,

parallelizing the application can often provide the improved performance being sought. *Parallelizing* an application means breaking it up into pieces to allow useful work to be done on more than one processor simultaneously.

Processes

The traditional way to parallelize an application has been to break it into multiple processes. These processes communicate using either an Interprocess Communication (IPC) mechanism, such as a pipe, or shared memory. The processes must be able to block waiting for events such as messages from other processes, and they must coordinate access to shared objects with something like locks.

Threads

We can now do much the same thing with threads, which have exactly the same coordination problems and similar mechanisms to deal with them. Using the common POSIX™ interface makes threaded applications portable.

Threads are separate loci of control within one process. A thread is a dispatchable unit with its own program counter, register save area, and stack that shares with its sibling threads most other resources belonging to the process, including memory and open files.

Threads are the only dispatchable unit in AIX 4.1. When a process is created, one thread is automatically created within it. Additional threads are created by explicit calls made by existing threads, such as `pthread_create()` in the pthread library.

A single process can now have any number of its threads running simultaneously on different processors. Coordinating them and serializing access to shared data are the developer's responsibility.

Threads and processes each have their own advantages to be considered when determining which method to use for parallelizing an application. Threads may be faster than processes, and memory sharing is easier with threads than with processes. Because memory sharing must be done explicitly when using processes, the process model provides help with data protection. In addition, a process implementation will distribute more easily to multiple machines (such as client/server) or clusters and will run on AIX 3.2.5.

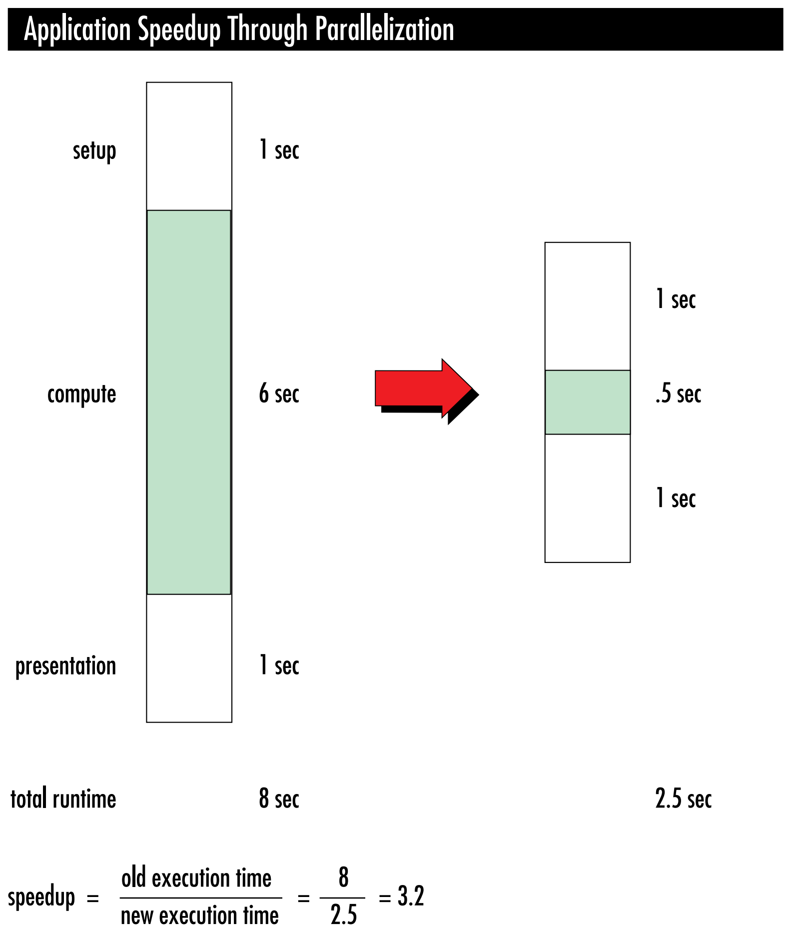


Figure 11. Application speedup through parallelization

Amdahl's Law

Amdahl's Law[1] quantifies the fact that if only part of a program is sped up, the part that was not sped up still runs as slowly as ever. Suppose the program shown in Figure 11 spends one second reading data and doing setup, six seconds computing, and one second producing graphs. Suppose also that the computing part can be parallelized so that it runs 12 times faster.

Making the main part of the program run 12 times faster sounds good, but only makes the program run 3.2 times faster. In fact, if the compute phase of this program could be made to run infinitely fast so that it took zero seconds, the program would still only run 8/2 or 4 times faster. This is an upper bound on the speedup that can be achieved by parallelizing the compute phase of this application.

[1] Schutzer, Daniel. *Parallel Processing and the Future Data Center*. New York: Van Nostrand Reinhold. 1994.

Amdahl's Law

$$\text{Speedup} = \frac{\text{Execution time}_{old}}{\text{Execution time}_{new}}$$

$$\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

Figure 12. Amdahl's law

Application Speedup Prediction

$$\text{Speedup} = \frac{1}{(1 - .75) + \frac{.75}{12}}$$

$$\text{Speedup} = \frac{1}{.25 + .0625}$$

$$\text{Speedup} = 3.2$$

Figure 13. Application speedup prediction

The formula for Amdahl's Law is shown in Figure 12 where $\text{Fraction}_{enhanced}$ is the fraction of the original path length we managed to speed up, and $\text{Speedup}_{enhanced}$ is the amount by which we managed to speed up that fraction.

Amdahl's formula can be used to predict overall speedup before a change is implemented and $\text{Execution time}_{new}$ can be measured. Figure 13 illustrates how overall speedup for the previous example could have been predicted.

Should I Parallelize My Application?

Many developers will ask the question "Should I parallelize my application?" There is no easy answer to this question. Parallelizing a substantial application is not easy and should be undertaken only if the following three criteria are met.

1. The application really needs to be speeded up and it cannot be done another way. Some likely candidates are as follows:

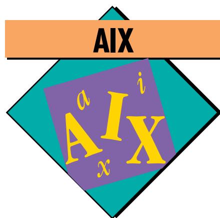
- Current response time is too slow. For example, a simulation or decision-support program takes hours, but the answers are only useful if they can be delivered in less than one hour. On Wall Street, evaluating certain financial instruments is only useful if it can be done in a few seconds.
- Current throughput is inadequate and needs to be increased, such as in an Online Transaction Processing (OLTP) environment
- An application needs to be enhanced, but still run in the same amount of time (such as a more accurate simulation or searching longer lists)

2. There is enough inherent parallelism in the problem. Finding and capitalizing on this parallelism may require redesigning and reimplementing its solution. A significant portion of the problem must be parallelizable. Remember Amdahl's Law!

- Data parallelism (good applications include OLTP, image processing, searching, and so on)
- Algorithm parallelism (good environments include client/server, pipelining, simulating different cases in parallel, and others)

3. You will need to learn parallel programming, invest the necessary resources, and suffer the inevitable frustration. Parallelizing a substantial application is not easy, but it may provide developers the needed performance improvements.

Debora Blakely-Fogel, IBM Corporation, RISC System/6000 Division, 11400 Burnet Road, Austin, TX 78758. Ms. Blakely-Fogel is an advisory programmer responsible for providing technical assistance to software vendors. She has a BS in Mathematics from the University of Massachusetts at Lowell and an MS in Computer Science from New Mexico State University.



Introduction to Multithreaded Programming

By Chary G. Tamirisa

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

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

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

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

Chary G. Tamirisa

POSIX Threads Programming Model

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

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

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

Each thread has the following thread-specific information:

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

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

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

Synchronization

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

Thread-Specific Context

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

Thread Cancellation

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

Once-Only Initialization

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

Thread Scheduling

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

Relationship Between POSIX.4a and POSIX.1

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

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

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

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

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

Overview of pthread Interfaces

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

Objects and Attributes

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

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

Thread API

The following operations can be performed on threads:

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

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

Attributes	Creation Functions
Thread	`pthread_attr_init(pthread_attr_t *attr)`
Mutex	`pthread_mutexattr_init(pthread_mutexattr_t *mutexattr)`
Condition variable	`pthread_condattr_init(pthread_condattr_t *condattr)`

Figure 1. Functions to create objects and thread attributes

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

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

Mutex API

The following operations can be performed on a mutex:

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

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

Condition Variable API

The following operations can be performed on a condition variable:

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

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

Thread-Specific Context

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

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

Thread Cancellation

Figure 3 shows thread-cancel operations and their APIs.

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

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

POSIX.4a API

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

Thread Creation

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

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

Figure 2. Thread attributes and their APIs

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

Figure 3. Thread-cancel operations and APIs

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

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

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

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

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

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

Thread Termination

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

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

Figure 5. Creating a thread

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

Figure 6. How to obtain thread exit status

Join With a Thread

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

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

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

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

Synchronization

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

Use of Mutexes

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

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

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

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

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

Use of Condition Variables

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

```
#include <pthread.h>

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

Figure 7. Use of mutex

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

Figure 8. Resulting output from using mutex

Programming Notes on Mutexes

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

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

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

Figure 9 shows the use of condition variables and mutexes.

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

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

Timed Wait Operation on Condition Variables

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

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

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

Once-Only Initialization

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

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

Thread-Specific Data

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

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

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

Figure 9. Use of condition variables and mutexes

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

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

Figure 10. Use of pthread_cond_timedwait()

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

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

Figure 11. Example of the use of pthread_once()

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

Figure 12. Special APIs to create and destroy context

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

Thread Cancellation

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

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

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

Cancel States

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

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

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

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

Cancel Point

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

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

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

Programming Notes on Condition Variables

1. Define condition variables as global variables since they are usually required to be visible to all the threads that use them.
2. Initialize the condition variables by calling pthread_cond_init() before use.
3. Initialize a condition variable only once.
4. Associate a boolean with a condition variable (as shown in Figure 10).
5. The correct use of condition variable is as follows:
   ```
   pthread_mutex_lock(&m);
         while(!ready)
               pthread_cond_wait(&cv, &m);
   pthread_mutex_unlock(&m);
   ```
   Ready is the boolean.
   Put the condition wait in a while() loop.
6. Whenever pthread_cond_wait() or pthread_cond_timedwait() returns, the mutex specified is locked. Unlock it before exiting the thread (or returning from the thread).
7. When pthread_cond_timedwait() returns on timeout, the mutex is locked. Unlock the mutex if you do not retry.
8. If you allocate memory dynamically for condition variables, always use pthread_cond_destroy() to destroy the condition variable, and then call free() to release the memory.
9. Condition wait and condition timed wait are cancel points.

Programming Notes on Thread-Specific Data

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

```
#include <pthread.h>

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

Figure 13. Thread-cancel API

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

Signals

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

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

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

The syntax for sigwait() is as follows:

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

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

POSIX.1 Functions

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

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

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

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

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

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

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

Programming Notes on Thread Cancellation

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

Programming Notes on Thread-Cancel States

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

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

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

Figure 14. Use of sigwait()

Programming Notes on Cancel Point

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

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

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

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

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

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

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

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

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

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

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

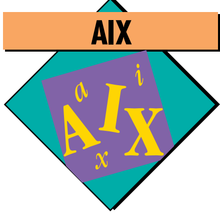
Summary

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

Chary G. Tamirisa, IBM Corporation, LAN Systems Division, 11400 Burnet Road, Austin, TX 78758. Internet: chary@austin.ibm.com. Since 1993, Mr. Tamirisa has been the team lead for the threads package on AIX and OS/2® Distributed Computing Environment (DCE). He has also worked in the fields of communication protocols, system software, and National Language Support. Mr. Tamirisa has an MS in Computer Science from McGill University and a BTech in Electrical Engineering from the Indian Institute of Technology in Madras, India.

Programming Notes on Signals

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



Porting Applications to the AIX 4.1 OS SMP Environment

By Debora Blakely-Fogel

This article highlights the changes needed in AIX 4.1 operating system uniprocessor applications so that they also function correctly on AIX 4.1 SMP systems. It is written for AIX developers who are moving applications from AIX 3.2.5 to AIX 4.1. It addresses only the porting issues directly attributed to MP. The reader should have a thorough understanding of AIX device drivers.

AIX 4.1 supports Symmetric Multiprocessor (SMP) systems based on the PowerPC family of processors. The programming model for AIX 4.1 uniprocessor systems is slightly different from the programming model for AIX 4.1 SMP systems.

Background of the PowerPC Architecture

To make an existing AIX application run correctly on an AIX Version 4 SMP system, it helps to first understand two key features of the PowerPC Architecture™: its weakly ordered memory and its out-of-order instruction execution.

Weakly Ordered Memory

The PowerPC Architecture specifies a memory model with relaxed instruction ordering requirements. Execution of a load or store instruction is considered complete when the associated address translation completes. At this point, the instruction has completed to the point that it will not generate an internal exception, although a subsequent read or write operation can still generate an external exception.

Load and store instructions are always issued and translated in program order with respect to other load and store instructions. However, a load or store operation that hits in the cache can complete ahead of those that miss in the cache. In addition, loads and stores that miss in the cache can be reordered as they arbitrate for the system bus.

Debora Blakely-Fogel

The PowerPC 601™ processor ensures memory consistency by comparing target addresses and prohibiting instructions from completing out of order if an address matches. Load and store operations can be forced to execute in strict program order.

Out-Of-Order Instruction Execution

In the PowerPC family of processors, instructions are not necessarily executed in sequential order. An instruction in the instruction cache can be sent to an idle execution unit for execution, even though it is not the next sequential instruction. For example, if an integer instruction is executing and the floating-point unit is not busy, a floating-point instruction can be sent to a floating-point unit to execute out of order while the integer unit is executing the integer instruction. Instructions that depend on the result of previous instructions will not execute out of order.

SMP Implications of Binary Compatibility

In general, applications that execute correctly on a uniprocessor RISC System/6000 running AIX 4.1 will also run correctly on an SMP RISC System/6000 running AIX 4.1. The exceptions are applications that serialize shared memory access among multiple running processes or threads and those that share memory with I/O. Typically, these applications use semaphores such as "compare and swap" to synchronize access to a particular segment of memory, and use a simple assignment to release the lock. While the following process works in AIX 3.2, it is important to

note that this will also work on an AIX 4.1 uniprocessor system. It is not sufficient for a PowerPC-based SMP system.

- Obtain the lock for the data, which can be done by calling `cs()` on AIX 3.2.
- Use the data secured by the lock.
- Release the lock with a simple assignment (such as `lock_word = 0;`).

Because of its weakly ordered memory model, the PowerPC can allow access to the application's critical data before obtaining the lock and also release the lock before the changed data is visible to the other processors. The following sequence of events will prevent these problems:

- Obtain the lock and synchronize the instruction caches.
- Use the data secured by the lock.
- Synchronize the buffers and release the lock.

Locking and unlocking for serializing access to shared memory does not require synchronization on a uniprocessor system, but it does on an MP system. AIX 4.1 operating system services can perform the extra required processing, making it easier for application developers. With AIX 4.1, locking is provided by the `_check_lock()` function and unlocking is provided by the `_clear_lock()` function.

The _check_lock() Primitive

Import and export fences are special processor-dependent synchronization instructions. They temporarily block other reads and writes to these locations. They protect against concurrent access by several processors, and against the read and write reordering performed by the PowerPC family of processors.

The `_check_lock()` function is an AIX 4.1 service similar to the compare and swap `cs()` service in AIX 3.2, with the addition of an appropriate import fence when the swap is successful. It differs from the `cs()` service in that it is restricted to word-aligned accesses only, as shown in Figure 1.

```
#include <sys/atomic_op.h>
boolean_t _check_lock(atomic_pword, int old_value, int
   new_value)
```

Figure 1. The _check_lock() service

The _clear_lock() Primitive

This service is an atomic store with the appropriate export fence, as shown in Figure 2.

```
#include <sys/atomic_op.h>
void _clear_lock(atomic_p word, int value)
```

Figure 2. The _clear_lock service

For uniprocessor systems, this routine is a simple store. For PowerPC SMP systems, it is preceded by the appropriate export fence. This routine does not need to execute disabled. The export fence guarantees that the data being protected is visible to all other processors prior to the lock word being released.

In addition to the synchronization changes required to make user applications run correctly on SMP systems, there are changes necessary for AIX device drivers to run correctly on SMP systems.

Applications that execute correctly on a uniprocessor RS/6000 running AIX 4.1 will also run correctly on an SMP RS/6000 running AIX 4.1.

Device Drivers

The AIX device driver models differ for multiprocessors and uniprocessors. Device drivers, as kernel extensions, have the same multiprocessor requirements as other kernel extensions. New features in AIX 4.1 that provide compatibility with AIX Version 3 uniprocessor device drivers also allow the programmer to tell the system the type of device driver when the device driver is registered. AIX 4.1 SMP systems support three types of device drivers:

- **Funneled**: Funneled device drivers run only on the master processor; therefore, the current uniprocessor serialization is sufficient. These device drivers are intended to support low-throughput devices. The base kernel provides binary compatibility for these device drivers.
- **MP Safe**: These device drivers run on any processor and contain modified code for adding a code lock to serialize the device driver's execution. MP Safe device drivers are intended for medium-throughput devices.
- **MP Efficient**: MP Efficient device drivers run on any processor. Their code has been modified to add data locks to serialize the device

driver's accesses to devices and data. MP Efficient device drivers, intended for high-throughput devices, are MP Safe device drivers that have been tuned for performance and efficiency.

The `devswadd()` kernel service defines a device driver to the AIX kernel. A field added to the device switch table indicates whether a device driver is to be funneled, MP Safe, or MP Efficient. To the base kernel, MP Safe and MP Efficient are the same. The base kernel assumes that the device driver is a funneled device driver unless the `devsw` structure's new `d_opts` field is modified to specify the `DEV_MPSAFE` flag before registering the device driver with the call to `devswadd()`. The device driver will be treated as an MP Safe device driver if `DEV_MPSAFE` is specified.

The AIX device driver models differ for multiprocessors and uniprocessors.

Top Half Considerations

A device driver's *top half* is the part that executes at the process level. It is not part of an interrupt-thread critical section, which serializes the execution of code that executes either on an interrupt level or at the process level with code that executes on an interrupt level. An interrupt-interrupt critical section is a special case of this type of critical section. The device driver's top half can be part of a thread-thread critical section, one that serializes the execution of code executing at the process level with other code executing at the process level.

The problem of concurrent access to global data is not specific to MP systems nor is it new to AIX 4.1. AIX Version 3 is a preemptable kernel, requiring that device drivers synchronize access to critical sections of code in their top halves. AIX Version 3 provided the `lockl()` and `unlockl()` kernel services for this serialization, which are also available with AIX Version 4.

In addition, two new types of locks can be used for thread-thread serialization: simple locks and complex locks. There are at least two reasons to consider replacing existing `lockl()` and `unlockl()` calls with these new services. The services for these new locks have instrumentation built into them that can be used to detect bottlenecks and to help in debugging deadlocks. During the development of AIX Version 4, these new lock services were tuned to provide efficient use of the system's resources and will often provide better performance than the `lockl()` and `unlockl()` locks.

Converting `lockl` locks to either simple or complex locks is not always straightforward. The simple locks, the preferred lock type, are functionally closer to `lockl` locks; however, `lockl` locks, unlike simple locks, allow nesting. Simple locks are recommended when `lockl` locks are replaced except when the work to remove nesting is too expensive or another feature of complex locks would help simplify the code. For example, complex locks can synchronize access to code that is rarely modified but frequently read.

The logical filesystem provides several forms of serialization for device driver top half routines:

- Ensures that the device driver's close routine is not called while it is performing a `read`, `write`, or `ioctl`.
- Serializes calls to the device driver's configuration routine (`sysconfig SYS_CFGDD`) with calls to the open and close routines. This serialization is only a code lock to ensure that the two sets of routines are not called simultaneously.

The device driver ensures that the call is valid for the current state. For example, the device driver must return an error when a request is made to unconfigure an open device.

Bottom Half Considerations

A device driver's *bottom half* is the part that executes in an interrupt-thread critical section at the process level, and all code that executes at the interrupt level.

There are many more considerations for making a device driver's bottom half MP Safe or MP Efficient than there are for the top half. The primary considerations for making the device driver MP Safe relate to serialization. In addition, there are some required interface changes for MP Safe device drivers.

Serialization Considerations

AIX Version 3 provides the `i_disable()` and `i_enable()` kernel services to mask interrupts or disable the processor. Although this disabling of interrupts provides adequate serialization on a uniprocessor system, it is not sufficient to serialize an MP Safe device driver's bottom half. Since the I/O is symmetric in an SMP and interrupts can be routed to any of the processors in the complex, masking interrupts will not prevent the interrupt routine from executing simultaneously on another processor. Serialization in an SMP requires the use of MP locks.

Not all device driver code disables interrupts to protect global data. In some cases, careful

ordering is used instead of disabling interrupts. This coding style will not work on a multiprocessor. Consider an example in which entries are added to and removed from a singly linked list only in a process environment. Updating the list can be done carefully to keep the list consistent. For example, an element might be added to the list with a simple assignment instruction only after all of its data fields are updated. Or the fields of an element might be modified only after it is removed from the list with a simple assignment instruction.

On a uniprocessor, this allows an interrupt handler to safely scan the list. This depends on process-level code never running simultaneously with the interrupt-level code. Although this is a correct assumption for a uniprocessor, it is incorrect for a multiprocessor. MP locks must be used to provide proper serialization.

Changing a device driver's bottom half serialization to make it MP Safe involves carefully replacing the uniprocessor interrupt locking with multiprocessor spin locking. This work consists of the following:

- Replacing `i_disable()` and `i_enable()` calls with calls to `disable_lock()` and `unlock_enable()`
- Adding serialization when disabled
- Removing nested locking
- Ensuring that interrupt-thread locks are never held across a sleep

AIX 4.1 provides the `disable_lock()` and `unlock_enable()` MP lock services for serializing interrupt-thread critical sections of a device driver's code. The `disable_lock()` service provides a spin lock that disables interrupts on the processor from which it is called. It also blocks the execution of code that tries to attain the same spin lock on another processor. The purpose of the `unlock_enable()` service is to release this spin lock and reenable the interrupts on the processor from which it is called.

Device Drivers Disabled When Called

Several device driver bottom half routines are disabled by the base kernel before they are called. These routines might not have serialization logic to prevent concurrent access to global data. To make these routines MP Safe, MP locks (`disable_lock()` and `unlock_enable()`) must be added to provide proper serialization. Figure 3 shows the device driver routines that are called disabled by the base kernel.

Routine	Location	Called at Interrupt Priority
EPOW handler	intr.handler	INTEPOW
Interrupt handler	intr.handler	INTCLASS0 - INTCLASS3 (intr.priority)
Timeout routine	trb.func	INTCLASS0 - INTCLASS3 (trb.ipri)
Watchdog	watchdog.func	INTTIMER
Off-level	intr.handler	INTOFFL0 - INTOFFL3 (intr.priority)
Iodone routine	buf.b_iodone	INTIODONE

Figure 3. Device driver routines disabled by base kernel

Uniprocessor Device Driver

```
oldpri = i_disable(my_intpri);
while ( io_not_done )
    e_sleep(&my_event);
i_enable(oldpri);
```

SMP Device Driver

```
oldpri = disable_lock(my_intpri, &my_lock);
while ( io_not_done )
    e_sleep_thread(&my_event, &my_lock,
    LOCK_HANDLER);
unlock_enable(oldpri, &my_lock);
```

Figure 4. Replacing MP Safe and MP Efficient device driver calls

Interrupt-Thread Locks Across Sleeps

Interrupts are automatically enabled when a process is put to sleep, then automatically disabled when the process is dispatched after wakeup. This is not true for multiprocessor spin locks; therefore, both MP Safe and MP Efficient device drivers must be changed to ensure that they never hold an interrupt-thread lock across a sleep. In general, this consists of replacing the device driver's calls to `e_sleep()` with calls to `e_sleep_thread()`. Figure 4 shows a typical example in which uniprocessor code would be changed to multiprocessor code.

The `LOCK_HANDLER` flag tells `e_sleep_thread()` that this simple lock is an interrupt-thread spin lock and not a thread-thread wait lock. This flag must always be specified for an interrupt-thread spin lock.

In addition, MP Safe and MP Efficient device drivers must be careful to release their multiprocessor spin locks when they call outside routines that might sleep. Although this is not a

```
if (disable_count == 0)
    oldpri = i_disable(my_intpri);
disable_count++;
    do_critical_section;
disable_count—;
if (disable_count == 0)
    i_enable(oldpri);
```

Figure 5. Device driver uses a variable to save path length

concern for the bottom half routines that run only on an interrupt level, it is a concern for the routines that can run at the process level. It is not necessary for locks to be released and reacquired when calling a base kernel service that can be called from an interrupt environment.

This discussion of explicit sleep calls also applies to implicit sleeps, such as page faults. An MP Safe or MP Efficient device driver cannot touch data that is not pinned while holding an interrupt-thread lock.

The order in which locks are acquired and the type of lock used are very important.

Removing Lock Nesting

The `i_disable()` service fully supports nesting, and many device drivers have nested calls to `i_disable()`. Since the `disable_lock()` service does not support nesting, all nested locking must be removed from a device driver to make it either MP Safe or MP Efficient. The best way to do this is to restructure the device driver so that nesting is not needed.

One successful technique to remove nesting is to have two names for each routine that can be called from either outside or inside a critical section. The first name is a small routine that is called from outside a critical section. This routine acquires the lock, calls the other routine to do the work, then releases the lock. The other routine assumes that the lock is already acquired and performs the function for which it has been called.

As shown in Figure 5, some device drivers may keep a variable to prevent nested calls to `i_disable` to save path length.

This technique cannot be used on a multiprocessor, even after the `i_disable()` and `i_enable()` calls have been changed to `disable_lock()` and `unlock_enable()`. That is because the initial test of `disable_count` is a critical section and could run concurrently on more than one processor. Since the AIX 4.1 base kernel does not provide an equivalent test that works for a multiprocessor, you must prevent nesting in a different way.

Preventing Deadlocks

The order in which locks are acquired and the type of lock used are very important. System deadlocks can occur when locks are not acquired and released in the correct order, or when an inappropriate type of lock is used.

One type of lock deadlock can occur when an interrupt-thread simple lock is acquired in a process environment without interrupts disabled. The system will deadlock if that thread is interrupted for any reason, and another instance of that device driver is called. The device driver will attempt to get its spin lock and will spin waiting on the preempted process to release the simple lock.

This can happen only when an interrupt-thread simple lock calls `simple_lock()` when interrupts are not disabled. It cannot happen when `disable_lock()` is called—yet another reason why `simple_lock()` should never be used directly for interrupt-thread critical section locks. The uniprocessor base kernel saves path length by not acquiring or releasing the simple lock when `disable_lock()` and `unlock_enable()` are called.

A more complex lock problem can occur when more than one lock must be acquired to perform an operation. A strict lock order must be defined to prevent deadlocks.

Consider the example illustrated in Figure 6 in which the different processes, pl and p2, need to acquire two different locks, lock1 and lock2. Assume the code being executed by process pl acquires lock1 before acquiring lock2, and the code being executed by process p2 acquires the locks in the opposite order. Assume also that between the time that p1 successfully acquires lock1 and the time that p1 tries to acquire lock2, process p2 runs and successfully acquires lock2. This results in a deadlock because process p1 will never release lock1 for use by process p2 until it has acquired lock2, and process p2 will never release lock2 for process p1 to use until it has acquired lock1. The locks must always be acquired in the same order and released in the inverse order.

Interface Changes for MP Safe and MP Efficient Device Drivers

Lock ordering problems are the reason for most changes in kernel interfaces for MP Safe and MP

Efficient device drivers. The base kernel either has to release its interrupt-thread locks before calling a device driver routine, which opens a serialization window, or it has to hold its interrupt-thread locks while calling a device driver routine, resulting in a possible deadlock. The changes in the base kernel interfaces are designed to prevent the problems that result from either of these situations.

MP Safe Registration

Since the kernel is designed to support binary compatibility for funneled device drivers, MP Safe and MP Efficient device drivers must inform the kernel that the extra funneling serialization does not apply to them. This is generally done when the `devswadd()` call registers the device driver; however, the kernel cannot relate certain device driver routines to the device switch entry of the device driver.

A device driver's strategy routine is defined to the base kernel via the `devswadd()` kernel service along with the device driver's top half routines. The `devstrat()` service will assume that the device driver is funneled unless the `d_opts` field of the `devsw` structure is modified to specify the `DEV_MPSAFE` flag prior to calling `devswadd()`. The device driver will be treated as an MP Safe or MP Efficient device driver if `DEV_MPSAFE` is specified. Strategy routine calls for funneled device drivers will be queued to be run on the master processor if the strategy routine is for a funneled device driver. Otherwise, the strategy routine will just be called. It may or may not have been called when `devstrat()` returns.

The `buf` structure's `b_iodone` field defines a device driver's `iodone` routine to the base kernel. The `iodone()` service assumes that the device driver is funneled unless the `dev` structure's `b_flags` field specifies `B_MPSAFE`. An MP Safe or MP Efficient device driver must set this flag in the `buf` structure before calling `devstrat()`. Otherwise, `iodone()` will run the device driver's `b_iodone` routine on the master processor.

The `i_init()` kernel service defines an interrupt handler to the base kernel. This service assumes a funneled device driver unless the `intr` structure's `flags` field is modified to specify the `INTR_MPSAFE` flag prior to calling `i_init()`. The device driver is treated as an MP Safe or MP Efficient device driver if `INTR_MPSAFE` is specified. The `INTR_MPSAFE` flag should also be specified for EPOW interrupt handlers and off-level

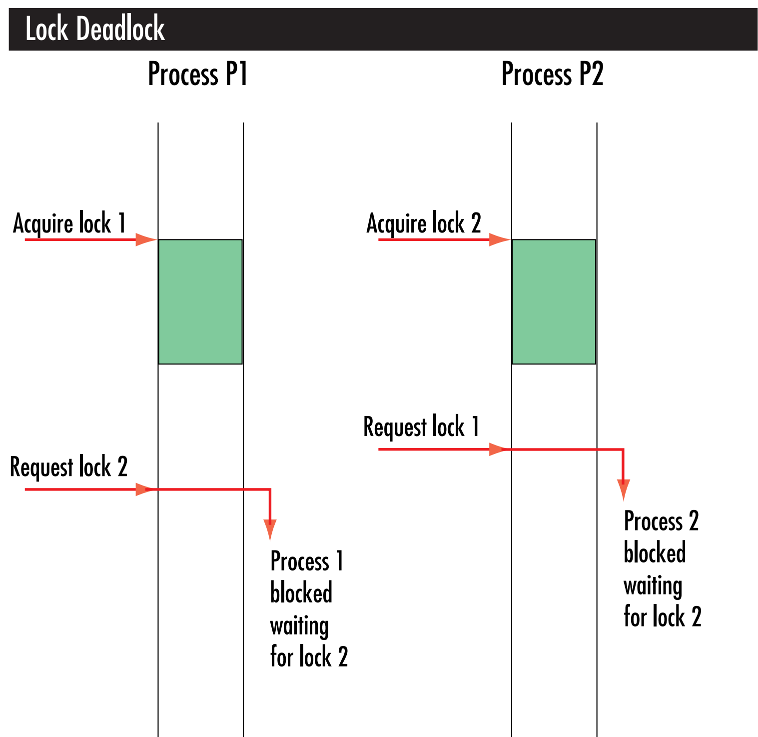


Figure 6. Lock deadlock

interrupt handlers. Although the initial multiprocessor base kernel does not require these flags to be set properly, setting them allows for future base kernel design changes. Similarly, the `T_MPSAFE` flag should be set in the flags field of the `trb` structure for MP Safe and MP Efficient device drivers.

Since there is no flags field in the watchdog handler's `watchdog` structure, watchdog handlers are architected to run only on the master processor or only on the processor from which `w_init()` is called.

Interrupt Handlers

The `i_init()` and `i_clear()` kernel services are called to add or remove an interrupt handler from the list of interrupt handlers known to the base kernel. The base kernel can always succeed at adding or removing the specified interrupt handler on a uniprocessor. It may not be successful on a multiprocessor if interrupts are being processed on another processor. Therefore, neither `i_init()` nor `i_clear()` can be called while holding an interrupt-thread lock. Calling either of these services while holding an interrupt-thread lock can result in a deadlock.

Timeout Request Handlers

The tstop() kernel service is called to remove a timeout request from the list of timeout requests known to the base kernel. Although the base kernel can always succeed at removing the specified timeout request on a uniprocessor, it may not be successful on a multiprocessor if timeout requests are being processed on another processor. A return value of 0 has been added to tstop() if the timeout request is successfully removed; otherwise it is -1. The tstop() calls must be changed in an MP Safe or MP Efficient device driver.

If an interrupt-thread lock was held when tstop() was called, the lock must be released and reacquired before retrying the call to tstop(). This allows the requests on the other processors to complete. Some delay may be required between releasing the interrupt-thread lock and reacquiring it. The tstop() service may fail even if an interrupt-thread lock is not held when it is called. In this case, the caller does not have to release and then reacquire the lock, but only retry the tstop() call.

Watchdog Timers

The w_init() and w_clear() kernel services add or remove a watchdog timer handler from the list of watchdog timer handlers known to the base kernel. The base kernel can always succeed at adding or removing the specified watchdog timer handler on a uniprocessor. It may not be successful on a multiprocessor if watchdog timers are being processed on another processor; therefore, a return value of 0 has been added to w_init() and w_clear() if the watchdog timer handler is successfully removed; otherwise it is -1. The w_init() and w_clear() calls in MP Safe or MP Efficient device drivers must be changed in a similar way to the calls to tstop().

If an interrupt-thread lock was held when w_init() or w_clear() was called, the lock must be released and reacquired before retrying the call to w_init() or w_clear(). This allows the requests on the other processors to complete. Some delay may be required between releasing the interrupt-thread lock and reacquiring it. The w_init() or w_clear() service can fail even if an interrupt-thread lock is not held when it is called. In this case, the caller does not have to release and then reacquire the lock, but just retry w_init() or w_clear().

The device driver's watchdog timer routine might be called even after the device driver calls w_stop() to disable the watchdog timer. This occurs only in a very small window when the kernel is in the process of calling the watchdog timer and the device driver calls w_stop() because of an interrupt that completes the I/O operation. This window exists in the current uniprocessor versions of AIX and will continue to occur in future versions of AIX. The device driver may need to use some form of state information between its interrupt handler and its watchdog timer routine to prevent accidental timeouts because of this.

I/O Processing

The base kernel currently sets B_DONE before calling the buffer's b_iodone routine. It will continue to do this for funneled device drivers, but it will not set B_DONE for MP Safe or MP Efficient device drivers. The MP Safe or MP Efficient device driver's b_iodone routine must set B_DONE within its critical section while holding its interrupt-thread lock. Otherwise, the setting of B_DONE and the wakeup of any process waiting on this I/O are not serialized correctly. This also implies that MP Safe and MP Efficient device drivers cannot call iowait(). Instead, they must provide a routine that sleeps on the buf's b_event field when B_DONE is not set. In addition, after setting B_DONE, the device driver must awaken processes that are waiting for the I/O to complete.

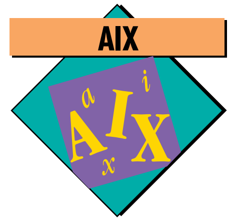
Dump Routines

The dump routine for an MP Safe or MP Efficient device driver should be coded to assume the worst case. The dump routine should assume that one critical section of the device driver was active on another processor when the dump was initiated, and the device driver's hardware may be in an indeterminate state. The dump routine should first establish a well-defined state at the start of the first dump write.

The dump routine may call parts of the device driver or kernel that have interrupt-thread lock calls. The kernel will ensure that no deadlock occurs due to these calls.

Debora Blakely-Fogel, IBM Corporation, RISC System/6000 Division, 11400 Burnet Road, Austin, TX 78758. Ms. Blakely-Fogel is an advisory programmer responsible for providing technical assistance to software vendors. She has a BS in Mathematics from the University of Massachusetts at Lowell and an MS in Computer Science from New Mexico State University.

AIX Operating System SMP Performance



By William Alexander, Robert Dimpsey, and Bret R. Olszewski

This article provides an introduction to Symmetric Multiprocessor (SMP) performance concepts. It also discusses using measurements on uniprocessors to successfully tune the SMP version of AIX. The result is an exceptionally well-performing SMP operating system.

Performance was a major emphasis in AIX Version 4.1. Analysts and developers worked together to improve Multiprocessor (MP) performance using measurements and analysis of AIX 4.1 on uniprocessors before the MP hardware was available. This strategy allowed work to begin on the MP hardware with an operating system that already performed reasonably well, and to concentrate on the more complex performance issues that arise from dynamic MP system behavior. The result of this effort is that AIX 4.1 performs exceptionally well for a first-release SMP operating system. IBM has tools and techniques in place to further improve performance as we gain experience from customer workloads.

SMP Performance and Scalability

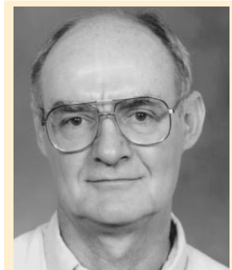
In an SMP system, multiple user tasks can run concurrently, and multiple copies of operating system routines can run concurrently on their behalf. However, there is only one copy of most system data structures: one run queue, one process table, one file table, and so on. If two system routines try to access the same system data simultaneously, the operating system must ensure that they access the data one at a time (serially). A lock is the primary way to serialize access to shared data.

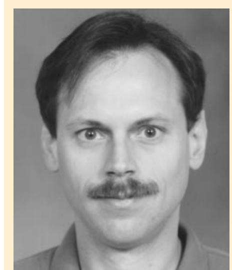
Locks

A *lock* is a bit associated with each piece of shared data. Before a routine accesses shared data, it must check the associated lock. If the bit is off, no other routine is currently accessing the data. The routine then sets the bit and accesses the data (testing the bit and setting it must be atomic). If a routine wanting to access the data finds the bit on, it must wait until the bit is turned off by the routine currently accessing the data.

Even with locks, it is difficult to ensure correct operation in an SMP. Hewlett-Packard® has stated that the most difficult technical problem moving to an MP system was adding protection to the data structures to allow multiple processes to execute[1]. This process is so complex that it is often started by using a single lock for the entire kernel. This implementation was used in SunOS Versions 4.1.2 and 4.1.3; however, this solution can result in unsatisfactory performance. It requires some kernel routines to wait unnecessarily when two routines access entirely disjoint sets of data, and could therefore safely run concurrently.

Although locks are necessary, they cause many performance problems in SMP systems, either directly or indirectly. First, acquiring and releasing locks adds path length to nearly all system routines. To minimize the negative effect of this extra lock-unlock path length on the performance of uniprocessor systems, IBM provides two versions of the AIX 4.1 kernel—one for uniprocessors and the other for SMPs. The uniprocessor version of the kernel contains fewer locks than the SMP version (some locks are still required in the uniprocessor kernel because AIX is preemptable).


William Alexander


Bret R. Olszewski

[1]Larson, Douglas V. and Polychronis, Kyle A. “A Multiprocessor HP/UX Operating System for HP 9000 Computers.” *Hewlett-Packard Journal.* December 1992.

SMP Scaling

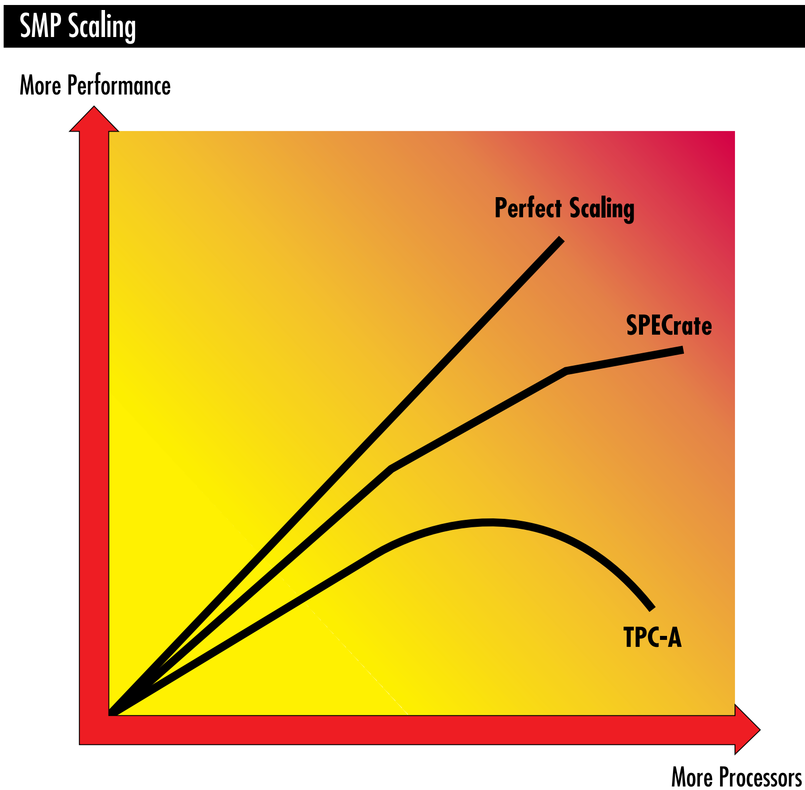


Figure 1. SMP scaling

Performance problems can occur with locks that protect heavily used data structures. The memory locations containing such locks can become very "hot," causing undesirable cache effects. This and other hardware considerations are discussed briefly in the next section.

The most serious performance problem caused by locks is the processor power wasted by routines waiting for locks held by another thread. If the lock will be released soon, the most efficient action is to *spin*, repeatedly testing the lock bit until it becomes free. If the lock will not be released soon, the waiting thread should be suspended and put in a queue waiting for that lock. Another thread must be dispatched, and the waiting thread must later be awakened and redispatched. Either way, these cycles are wasted from the user's perspective. Because waiting time grows non-linearly as the number of competing tasks increases, waiting has a very adverse effect on scalability.

Scalability

Scalability is the effective speedup of the system as processors are added. Typically, we calculate scalability as the effective throughput of a given workload running on multiple processors divided by the throughput of that same workload running on a single processor (the base). Scalability is often expressed as a fraction; for example, 2.5/4 means that a particular four-processor SMP achieves two and a half times the performance of a single processor in the same system on a given workload.

Measuring scalability can be straightforward because most SMPs allow the dynamic disabling of processors from software. We can turn off three processors on a 4-way SMP and measure throughput with a workload running on one processor, then measure it again with the workload running on all four processors.

Customers who upgrade to an SMP generally compare the performance of a 4-way SMP running AIX 4.1 with that of a uniprocessor running AIX 3.2 rather than AIX 4.1. The performance of the SMP running AIX 4.1 will be less than the uniprocessor running 3.2 because of the extra lock-unlock path length. Much of this extra path length penalty is masked because there are many performance enhancements in AIX 4.1 unrelated to MP.

For real-life workloads on SMPs, each additional processor adds less throughput than the previous one until a point is reached where adding processors decreases throughput. As shown in Figure 1, scalability is highly workload-dependent, so the point of diminishing returns will be different for different workloads and benchmarks. For example, a benchmark such as SPECrate™—which suffers few cache misses, makes few operating system calls, and has little shared user data—will scale linearly on most SMPs. At the same time, a system-intensive workload may gain little benefit from more than a certain number of processors.

Although many impediments to perfect scaling in SMP systems exist, the following four are the most important:

- Contention for memory subsystem hardware, especially the bus and data switch
- Increased overhead of cache misses because of coherency protocol
- Communication and contention for shared data within applications
- Contention for shared data within the operating system

Each processor in IBM's SMPs has its own Level 1 (L1) and Level 2 (L2) caches. There may

be copies of a given memory location in the caches of any of the processors. If there are copies in more than one cache, the copies must be kept consistent. If one processor executes a load instruction for a particular location, and another processor has a copy of that location in its cache, the hardware ensures that the load is satisfied from the other processor's cache rather than from memory. This guarantees that the load instruction has the freshest value.

A *lateral read* is reading from another processor's cache, not from memory. If two processors have a copy of the same location in their caches and one processor modifies the value, the hardware invalidates the copy in the second processor's cache; this is called a *cache cross-invalidate*. Although lateral reads and cache cross-invalidates are necessary to keep the caches consistent, this cache-to-cache traffic increases the load on the system bus as shown in Figure 2.

With multiple processors, contention for hardware elements of the memory subsystem is a scaling impediment inherent in SMPs. Caches successfully reduce memory accesses, but the cache coherency protocol adds bus traffic of its own. Together, contention for memory access and cache coherency protocol overhead cause the average Cycles Per Instruction (CPI) for a given workload to grow as more processors are added to a system. Naturally, if it takes more cycles to execute the same number of instructions, performance suffers.

These hardware effects are not directly visible to software; a program cannot directly detect when a load from memory had to wait for access to the bus, or when a store caused a cache cross-invalidate. But this does not mean that software developers should ignore these hardware-scaling impediments. Programs can reduce cross-invalidates by ensuring that two "hot" data items or locks are not in the same cache line.

Because a cache miss is more costly on a multiprocessor than on a uniprocessor, the principles of distributed memory apply at the cache level for shared memory machines: keep the data close to the processor. The workload scalability is closely related to the percentage of memory references that can be satisfied in the nearest memory (L1 and L2 caches for an SMP). Hewlett-Packard has stated that the bulk of their tuning work for the 4-way HP™ 9000 827 was spent reducing cache effects[2].

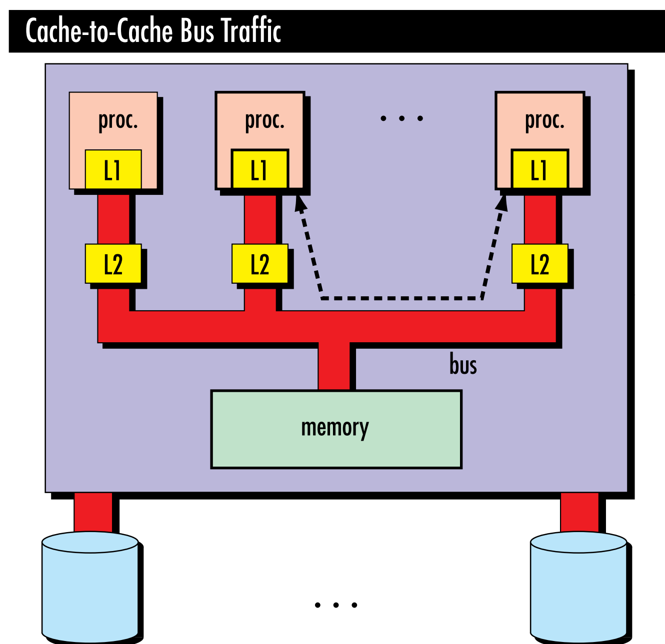


Figure 2. Cache-to-cache bus traffic

If a single application consists of multiple tasks, then scalability can be reduced by communication and synchronization among tasks. Contention for shared application data can also limit scalability. Both considerations apply in some database implementations.

The remainder of this article will focus on our efforts to confront the fourth impediment to scaling: contention for system data structures.

System Performance Goals

During the early development of AIX 4.1, we established two types of performance goals. The first applies to base performance on a single processor—a path length goal for system components. The second goal is related to scalability.

The base goal, which applies to the MP version of the kernel, states specifically that the performance of our benchmarks would not degrade more than 15% when run on the MP-enabled kernel compared to the AIX 4.1 Uniprocessor (UP) kernel. Enabling AIX to run on SMPs adds path length to many kernel components because of its support for threads and the extra lock-unlock code. Our approach was to apply standard path length analysis techniques to the new code while it ran on a uniprocessor. In addition, we made performance improvements to AIX that were unrelated to MP, which partially offset the path

[2] Ibid.

Component	TPC-A C/S	TPC-C C/S	LADDIS	SDET	Kenbus
Virtual Memory Manager (VMM)	High	High	High	High	High
Process Management (PROC)	High	High	Low	High	High
Journaled File System (JFS)	None	None	Low	High	Low
Logical File System (LFS)	Low	Low	Low	High	High
Network File System (NFS)	None	None	High	None	None
TCP/IP	Low	Low	High	None	None
Network Device Drivers	Low	Low	High	None	None
Loader (LDR)	None	None	None	Low	High
Disk Device Drivers	Low	Low	Low	Low	Low

Figure 3. System component usage

length increases that could not be avoided. Most of these improvements applied to both kernels. Our measurements show that the AIX 4.1 UP kernel performs better than AIX 3.2.5 on most internal performance regression tests, and the MP kernel running on a uniprocessor meets the 15% goal on all industry-standard benchmarks.

AIX 4.1 UP kernel performs better than AIX 3.2.5 on most internal performance regression tests.

Since the second goal concerned operating system scalability, we focused on locks. Lock contention is the principal software culprit limiting scalability. As more processors are added, more system routines contend for the same locks, the locks are held longer, and waiting time rises.

Experience with other SMPs and queueing theory suggests that waiting time for a given lock becomes unacceptable if the lock is held more than 20% of the time. We therefore adopted the goal that no AIX lock be held more than 20% of the time under our test benchmarks. This implies that in a four-processor system, one processor cannot hold a particular lock more than 5% (20%/4) of the time. With a design goal to provide scalability to eight processors, the target hold time per processor was set to 2.5%. As AIX 4.1 was being developed, we measured the fraction of time each lock was held as we ran our benchmarks on a uniprocessor, and flagged any lock that was held more than 2.5% of the time. This enabled us to make substantial progress on scalability, far ahead of the availability of MP hardware. In fact, we could accurately predict the locking rates that we later measured on the SMP.

Since we began early and AIX developers were involved in the performance effort from the beginning, the AIX 4.1 SMP kernel meets the 5% goal for all locks on all of our benchmarks. In addition, most locks on benchmarks also clear the 2.5% mark. These facts, plus early scalability measurements, give us confidence that most workloads will scale well on our 4-way SMP. We will continue to achieve 8-way scalability.

Workload-Based Design

Because IBM's SMP is a general-purpose server, it must perform well in complex multi-user situations. A performance measurement process was designed around a few workloads that were used for performance evaluation of the AIX 4.1 MP software. The selected workloads—industry-standard benchmarks representing a cross section of the critical markets defined for the product—included the following:

TPC-A: At the time of initial SMP planning, this was the most frequently used database benchmark. All implementations studied were client/server.

TPC-C: This is a significant database benchmark. All implementations studied were client/server.

SPEC SFS (LADDIS): This benchmark tests NFS® server performance. Typically, it is run with multiple Ethernet™ or Fiber-optic Data Distribution Interface (FDDI) networks sending read, write, and lookup requests to the server.

SPEC SDM SDET: This workload tests multi-user performance by simulating several users executing typical UNIX® commands.

SPEC SDM Kenbus: This workload tests multi-user software development performance. It is similar to SDET, but simulates thousands of users and must run on large memory systems.

All these benchmarks involve large amounts of disk I/O. A typical four-processor MP system requires 20 to 40 disks to sustain peak throughput on these workloads. The client/server TPC and LADDIS benchmarks also require network interaction with the server systems.

In addition to these real-world workloads, we have developed stress tests that focus on the disk, communication, and Streams framework subsystems.

Each workload was analyzed on AIX 3.2, categorizing path length use in components and subsystems. This analysis was used to plan the software development effort needed for parallelization. Figure 3 shows a summary of the findings.

The operating system components exercised during the analysis vary with the workload. No one workload can characterize the real world well enough to guarantee good general-purpose performance.

Lower level analysis showed that use within components varies considerably. For example, although the Virtual Memory Manager (VMM) is used extensively in all workloads, different VMM functions are exercised by various workloads. For the TPC workloads, VMM is used almost exclusively for copy operations between user space and the kernel for system calls. For LADDIS, VMM exercises page operations related to I/O. For Kenbus and SDET, VMM exercises operations associated with process creation and deletion.

By understanding which components were not critical, we could avoid investing too much in making them parallel. We targeted the components that were critical to achieving excellent system-level performance.

Lock Implementation

Locks are requested frequently in commercial workloads, so the implementation lock and unlock code is crucial for good performance. Early measurements of locking rates are shown in Figure 4.

Because locks are so central to SMP performance, considerable effort was invested in their implementation. Principal goals in the lock primitives design included different locking options, low path length, and sophisticated instrumentation. Two basic lock types, both based on the OSF/1® model, are used: simple locks and complex locks.

Simple locks are mutually exclusive locks; only one entity may be holding the lock at any time. Simple locks in AIX 4.1 can behave as spin locks. When a process cannot obtain a lock because it is held by another process, the lock code spins in a tight loop, continuously testing the lock bit until it can gain the lock or until a threshold of spinning is reached (`MAX_SPIN`). Threads that reach the threshold of spinning are put to sleep and awakened when the lock is free. Interrupt handlers, which are not allowed to sleep, must spin until the lock becomes free. Another useful feature of simple locks is that if the thread holding the lock is not currently dispatched, the requester is put to sleep immediately. That is because it is unlikely that the required unlock will occur soon.

Complex locks are mutually exclusive when used for operations that modify data (WRITE access), but allow multiple entities to hold the lock simultaneously for viewing data (READ access). Otherwise their implementation is similar to simple locks.

	Lock Rate/sec Quad
SDET	102,000/sec for 4 processors at 75 MHz
TPC-C	57,000/sec for 4 processors at 75 MHz
LADDIS	56,000/sec for 4 processors at 75 MHz

Figure 4. Locking rates for workloads

Lock instrumentation is critical for understanding the dynamic behavior of the system. There are two sets of instrumentation: counters and trace hooks. Counters allow gross level analysis of lock and lock miss rates in the system and can be viewed through the `lockstat` command. Counters indicate which locks are hottest. Trace hook instrumentation allows sophisticated analysis of system behavior, such as identifying lock hierarchy problems.

Lock instrumentation is crucial for SMP tuning, but it has a noticeable (2% to 3%) impact on performance, even when the tracing facility is not activated. For this reason, AIX can be run with lock instrumentation enabled or not; this decision is made when the kernel is booted.

Principal goals in the lock primitives design included different locking options, low path length, and sophisticated instrumentation.

Tuning AIX During Development

This section describes performance improvements that were made running AIX 4.1 on UP platforms, before MP hardware was available.

To help compensate for the unavoidable path length expansion required to support threads and implement locking, a significant tuning effort was undertaken. This resulted in the performance of the AIX 4.1 UP kernel being better than the AIX 3.2.5 kernel, in spite of supporting threads and some restructuring related to MP enablement. The improvements to the UP kernel also typically improved MP performance.

One large class of problems is associated with copying data between user and kernel space. The primary difficulty is that thread support requires locking the process address space, which is not necessary in previous versions of AIX. For single-threaded processes, significant design changes mean that copies run much faster in AIX 4.1 than in any previous version of AIX.

One pleasant surprise was the performance benefit of compiling the operating system using the data-in-toc compiler option. This option allows the compiler to move some constants into the Table of Contents (TOC) area—removing one instruction of overhead for each access. An

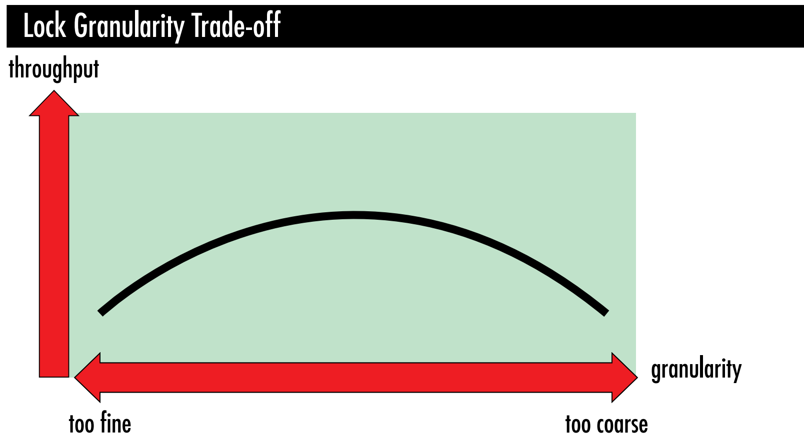


Figure 5. Lock granularity trade-off

average of 3% improvement in system performance was observed merely by rebuilding the operating system with the data-in-toc compiler option.

The path length for the lock and unlock operations is one key to good performance. The path length associated with locking was initially higher than expected. Three optimization passes were performed on the locking code to reach the current implementation. The lock and unlock primitives are processor type-specific, so performance work centered on the implementation for the PowerPC. PowerPC systems have four special registers available to software. The pointers to the current thread structure, the current save area (`csa`), and the per-processor data area (`ppda`) are kept in these special registers. This allows these data structures to be accessed in one instruction. Since the thread identifier is used in all lock operations, a readily available pointer to the current thread improves lock performance on the PowerPC.

When measurements show that a lock is held in excess of the 2.5% target, one common technique is to reduce its granularity. For example, a lock protecting a table might be replaced with an array of locks, each protecting one entry in the table. This technique was used in the `pmap` layer of the VMM.

Sometimes during AIX 4.1 development, measurements showed that path length was unacceptably degraded because of too many locks. The solution was to coalesce them. For example, the initial granularity of the communication subsystem's memory allocator turned out to be too fine, and locks were coalesced to protect larger structures. This is a classic SMP tuning trade-off: decreasing the number of locks decreases path length in both UP and MP versions of the kernel, but may increase lock holding and waiting times in MP systems. Finding the right balance is more art than science. Figure 5 illustrates the trade-off between granularity and throughput.

Another example of increased UP performance obtained by removing a lock occurred in the logical filesystem. Here, the user file descriptor (`ufd`) per-process structure must be modified when a new file is opened. Initially, a lock was taken to support threads, since two or more threads could try to open a file simultaneously. An easy optimization was to check if there was only one thread in the process, in which case locking was not needed.

Another common optimization was converting code that modified single data fields under lock to use atomic operations such as `fetch_and_add`. Significant processor cycle savings can occur using atomic operations.

Some non-critical statistics counters were converted to being "fuzzy"; that is, counters are not 100% accurate, but memory inconsistency may cause a count to be lost from time to time. Some statistics, such as the number of free pages available to the VMM, cannot be fuzzy; but many can be with no noticeable effect to users. Several counters in TCP/IP were made fuzzy.

Some large system effects were observed, even when running on UP systems. For example, during development a convoy problem was noted with the Kenbus workload. Frequently, a process holding a lock was preempted at the end of its time slice. The next process to attempt to gain the lock would miss it. The lock miss code identifies the process holding the lock and tries to ensure that it will run soon because it is causing lock contention. A priority boosting mechanism now bumps the priority of the process holding the lock so that it will run again soon.

The large system effect was related to the many processes in Kenbus; the bumped process queued at a more favorable priority, but the list of ready processes on the priority queue was so long that subsequent lock misses occurred on the same lock. The solution implemented in AIX 4.1 was to bump the process and queue it at the front of the priority queue. This improved performance, and perhaps as importantly, reduced variance in system response time.

MP System Dynamics During Development

The most serious limitation of tuning SMP software on a UP is that the dynamic behavior of any code will be different on an SMP than it is on a UP. Therefore, it is difficult to predict how any tuning change will affect SMP behavior. This section provides a few examples of the kind of tuning work that could be accomplished on SMP hardware.

After a "hot" (frequently accessed) lock is broken into multiple locks to reduce waiting time, affected kernel routines run faster. The result is that the rate at which the kernel requests certain other locks is now increased. This increase in access rate will not be uniform across all locks, and it is not always possible to determine which locks will now be hottest.

The two major "hot" locks identified through UP analysis were the Journaled File System (JFS) lock and the VMM lock. Both subsystems were initially constructed with just one lock protecting all of their respective data structures. This MP Safe version of the operating system was coded so runnable software would be available when functioning hardware was first ready. Developing and implementing a more granular set of locks to protect these data structures was done in parallel with designing the first software and hardware.

Our initial MP measurements highlighted the VMM and JFS locks, as expected from the UP measurements and the lock protecting the process table. Installing the more efficient VMM and JFS locking design dramatically improved performance. For example, SDET performance improved by over 70%.

Fixing the first tier of hot locks also revealed the next tier, which we had not been able to predict from our UP measurements. This set included two new locks introduced with the more efficient versions of JFS and VMM: the `inode` cache lock and the `pmap` lock.

The JFS `inode` cache lock protects the `inode` pool. Contention for this lock was reduced by improving the hashing algorithm used to locate the `inode` in the pool. This reduced the path length under lock; no additional locks were introduced. The `pmap` lock was split into multiple locks.

One interesting problem that UP-based analysis could not reveal concerned an interaction between sequential disk I/O and interrupts. AIX uses a write-behind algorithm to greatly increase sequential write throughput. Because SMP interrupts can generally be fielded by any processor, the system can sometimes send smaller than optimal write requests to the disks during sequential I/O. We developed a coalescing algorithm to correct this problem.

Another non-intuitive observation came with analysis of the optimum value for `MAX_SPIN` (number of times to spin on a lock before sleeping). We expected this value to be important to system performance, but experiments showed that most workloads were not sensitive to it. This is because our design reduces the number of lock misses. Of the misses that do occur, some fraction are in interrupt handlers that are not allowed to sleep, and thus do not use `MAX_SPIN`.

Two internal trace postprocessing tools were indispensable in the tuning effort. One displays events from the AIX trace log in a graphical manner, making abnormal or undesirable system behavior easier to spot. The other tool summarizes system call times, lock holding times, and lock miss rates; this tool highlighted the remaining hottest locks after each successive wave of tuning.

The dynamic behavior of any code will be different on an SMP than it is on a uniprocessor.

Processor Affinity Scheduling

Our processor affinity scheduling policy could not be addressed solely from UP data. On SMP systems, several dispatching disciplines can be implemented. The most obvious allows any thread to be dispatched on any processor, enabling the system to keep all processors busy. This policy usually results in the best system throughput.

A policy that results in fewer cache misses binds each thread to a particular processor and allows it to execute only on that processor. For example, if a system has a large second-level cache for each processor, dispatching a thread on the processor on which it last ran benefits from reusing the lines of the cache still containing context for that thread. This policy, called *strict affinity*, can result in better response times for high-priority tasks, but generally causes lower system throughput because one or more processors may become idle while there are threads ready to dispatch.

The AIX dispatching strategy implements *opportunistic affinity.* When dispatching, the list of threads available at the highest priority is searched. If one of these threads was the last to run on the processor doing the dispatch, it is selected; otherwise, the first thread available is

dispatched. In this way, most benefits of affinity are preserved, but system throughput is also maintained.

Applications may use the `bindprocessor()` system call to explicitly bind threads to processors, if desired.

References

- Campbell, Mark, et al. "The Parallelization of UNIX System V Release 4.0." Presented at USENIX, Winter 1991. Dallas, Texas.
- Campbell, Mark; Holt, Russ; and Slice, John. "Lock Granularity Tuning Mechanisms in SVR4/MP." *Distributed & Multiprocessor Systems* (SEDMS II), USENIX Association.
- Kleinman, Steven, et al. "Symmetric Multiprocessing in Solaris 2.0. USENIX, Summer 1992. San Antonio, Texas.
- Eykholt, J.R., et al. "Beyond Multiprocessing: Multithreading the SunOS Kernel." Presented at USENIX, Summer 1992. San Antonio, Texas.

William Alexander, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Dr. Alexander is a senior programmer. Since joining IBM in 1991, he has worked on AIX and MP performance. He has a BA in Philosophy from Rice University and a PhD in Computer Science from the University of Texas at Austin.

Robert Dimpsey, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: dimpsey@ austin.ibm.com. Mr. Dimpsey is an advisory programmer. He has worked in the AIX Performance group since joining IBM in 1992.

Bret R. Olszewski, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Olszewski is an advisory programmer working on MP performance. He joined IBM in 1989 and has worked on various aspects of AIX performance. He has a BS in Computer Science from the University of Minnesota.

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.

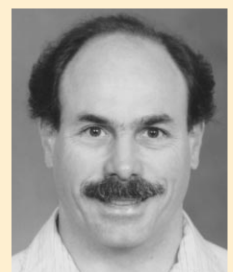
In practice, a kernel program will have some operations that must be *atomic*, meaning they run to completion without interruption. Actions taken by the kernel in response to a system call (a request for service from an application) might be interrupted. If a structure can be referenced both in a system 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

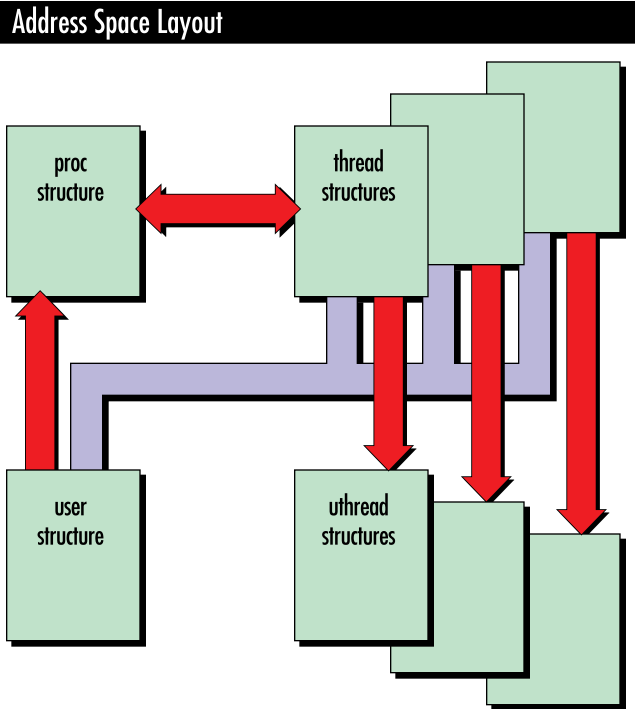


Figure 1. Address space layout

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

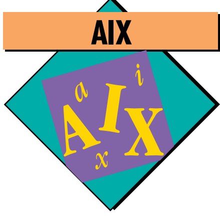
The AIX Version 4 kernel continues to support device drivers written by customers and other software vendors. Since those written for AIX Version 3 are unlikely to be quickly redesigned for an SMP system, the kernel supports uniprocessor-oriented device drivers through the following procedure called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



DCE Performance Study

By Bob Russell

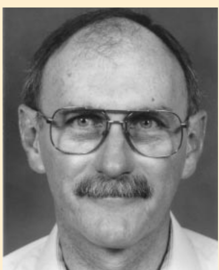
This article presents DCE performance characteristics observed in a complex heterogeneous enterprise environment. It addresses some of the frequently asked performance questions, focusing on DCE topology performance, while noting the behavior of the DCE administrator, OS/2 Database Manager, and Network File System (NFS) services contending for the system resources.

Distributed Computing Environment (DCE) application behaviors were studied on a variety of OS/2-, AIX-, UNIX-, and Windows™-based systems. Over 1,500 clients were supported by 12 to 16 DCE and application servers. We used off-the-shelf hardware and software including IBM and non-IBM solutions for the Distributed Computing Environment (DCE) performance study. The IBM Point-of-Sale (POS) DCE application benchmark was used to drive the workload for this evaluation.

Study Requirements

The needs of IBM customers who are implementing applications on DCE set the direction for this performance study. During the first half of 1994, we surveyed and met with many of these customers. Figure 1 shows the results that led to the following requirements for the study:

- **More than 1,500 clients**. The surveys indicate the highest interest is between 100 and 500 clients. We studied workloads up to 2,000 clients.
- **Heterogeneous client and server environment**. The following clients and application servers were studied:
  - **Clients**: Windows, OS/2, OS/2 WARP, AIX/6000™, Sun® and Hewlett-Packard (HP) workstations
  - **DCE and application servers**: Sun, HP, and PS/2® and RISC System/6000 (RS/6000™) systems
- **Performance comparison between a large single cell and two smaller cells**.
- **Performance comparison between single and multiple cells connected by a Wide Area Network (WAN)**. The survey feedback indicates the greatest interest in system geographies beyond a single contiguous location.



Bob Russell

Methodology

The philosophy for these tests was to view the system behavior from a high level. We did not use high-resolution timers to obtain microsecond-level measurements. Generally, we monitored end-user response time and system throughput, and used the following guidelines for interpreting the results:

- Response times less than one second are considered instantaneous.
- Response times greater than one second may be perceived by the user as a long time.
- The *maximum throughput* is the highest arrival rate applied to the system without causing an objectionable degradation in end-user response time.

In many of the test cases, higher throughput could have been achieved. However, if we had pushed the arrival rate any higher, the response times would have begun to grow above our one-second guideline.

Standard Configuration

Figure 2 shows the standard system configuration for these studies. The performance characteristics of the standard system were studied before introducing variations. The standard configuration

Platform	Windows	OS/2	AIX	Midrange	Mainframe
Client Workstation	5		10		6
Application Server		10	9	4	3
DCE Server		9	10	3	2
Number of Clients	**1-99**	**100-499**	**500-999**	**1000-4999**	**5000+**
	2	10	7	5	1
System Geography	**Building**	**City**	**State**	**Country**	**Global**
	8	10	1	6	2
Type of Application	**Database**	**Data Transfer**	**Graphical**		
	9	10	3		
Data Transfer Size	**1 KB-4 KB**	**32 KB-64 KB**	**512 KB-1 MB**		
	10	6	4		

Relative importance (normalized): 1 = least important, 10 = most important

Figure 1. Customer survey results

consists of three separate networks connected by WANs.

Headquarters is the primary network and provides most of the system support. In the standard configuration, all DCE servers and POS application servers are in the headquarters network. The baseline throughput for the standard system at the Headquarters level (shown in Figure 3) is 1,640 Customer Sales Per Minute (CSPM), and the 16 Mbps Token Ring is 40% to 45% utilized. The POS application servers in the standard configuration can support about 1,640 POS clients. An OS/2 Database Manager Online Transaction Processing (OLTP) benchmark, also running at 15 Transactions Per Second (TPS) on a separate server, is not affected by the POS traffic on the network.

The **Region Office** is a smaller network connected to Headquarters by a high-speed WAN. We simulated the high-speed WAN using two IBM Token-Ring Network Bridges linked together by a 4 Mbps Token-Ring network. This is faster than the 1 Mbps links generally used, but the network analyzer indicated that traffic on the 4 Mbps link remained below 0.8 Mbps during these tests. The maximum workload exerted by the clients in the Region Office is 190 CSPM.

The **Branch Office** has a small 16 Mbps Token-Ring network with two PS/2 clients: one running DCE for OS/2 and one running DCE for Windows. The Branch Office is linked to Region Office using IBM LAN Distance™. The link used in the study between the Branch and Region was a null modem set at 38,400 bps. The maximum throughput of the Branch Office was limited to 7.5 CSPM by the line speed of the null modem.

Applications

The following benchmark applications provided the system load:

- The Point-of-Sale DCE application benchmark is patterned after a retail order-entry system. Since our objective was to look at DCE performance, the POS workload is the predominate application in the study.
- A suite of `rgy_edit`, `acl_edit`, `rpccp`, and `cdscp` updates is run from one workstation. This simulates the ongoing activity of a DCE system administrator.
- An OS/2 Extended Services 1.0 Database Manager OLTP benchmark using the NetBIOS transport for the Database Application Remote

Enterprise Server Topology

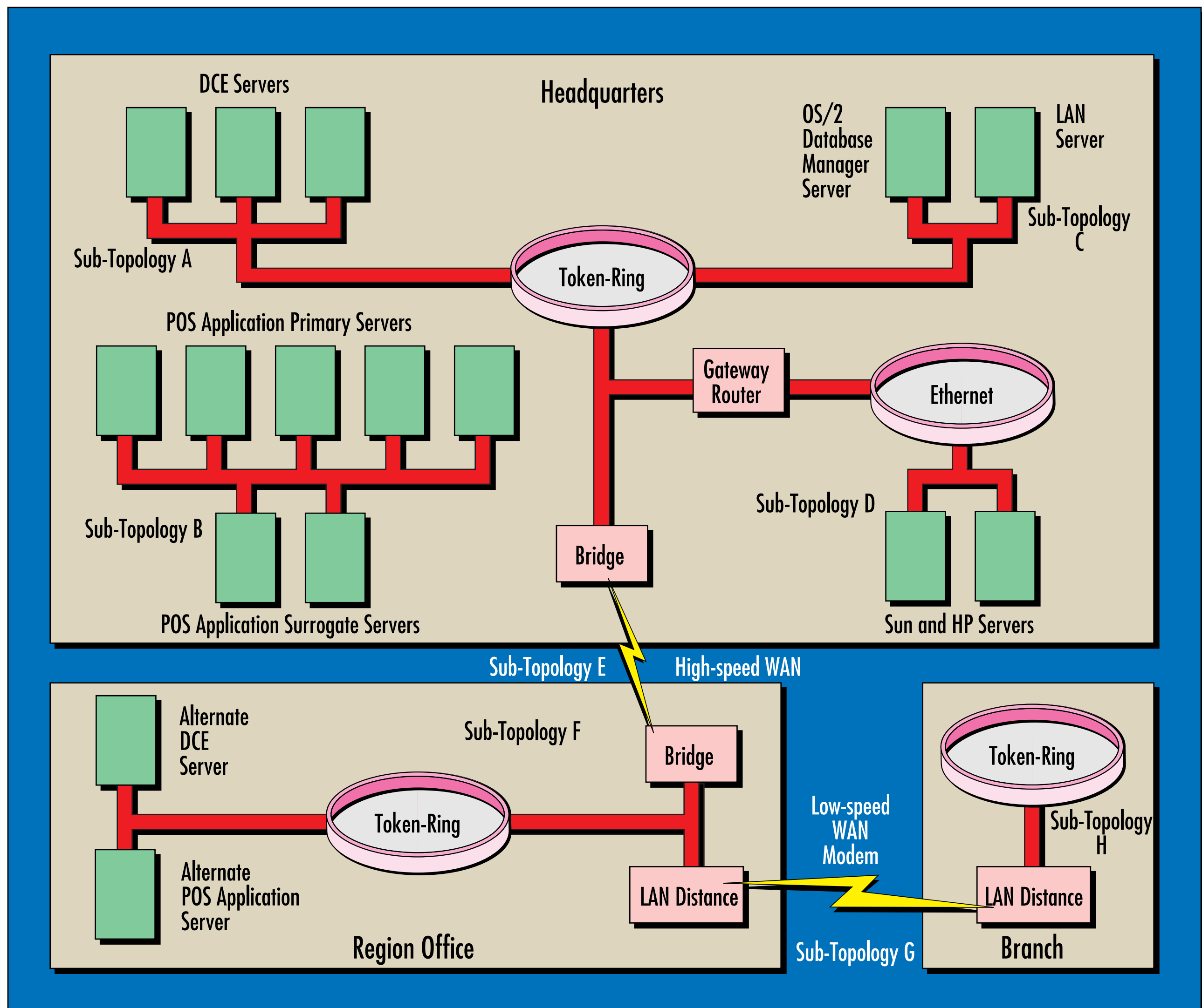


Figure 2. Server topology for the enterprise

Interface (DARI) stored procedures is run on a small number of PS/2 clients on the Headquarters' Token-Ring network.

- IBM LAN Server/Requester 3.0 provided file sharing for the benchmark applications and tools. The LAN Server performance was not measured in the study.

Results

Due to the complexity of the system topology shown in Figure 2, this article discusses the sub-topologies (A-H) and the performance of each sub-topology individually.

The DCE Servers

In the standard configuration, the DCE servers (Figure 2, Sub-topology A) are all PS/2s:

- The Primary DCE Cell Directory Service (CDS) server was a PS/2 9585-50 MHz. The CPU utilization remained under 40% during all tests. We found that the type of DCE server has a minimal impact on the overall cell throughput; the total difference between a 33 MHz and 50

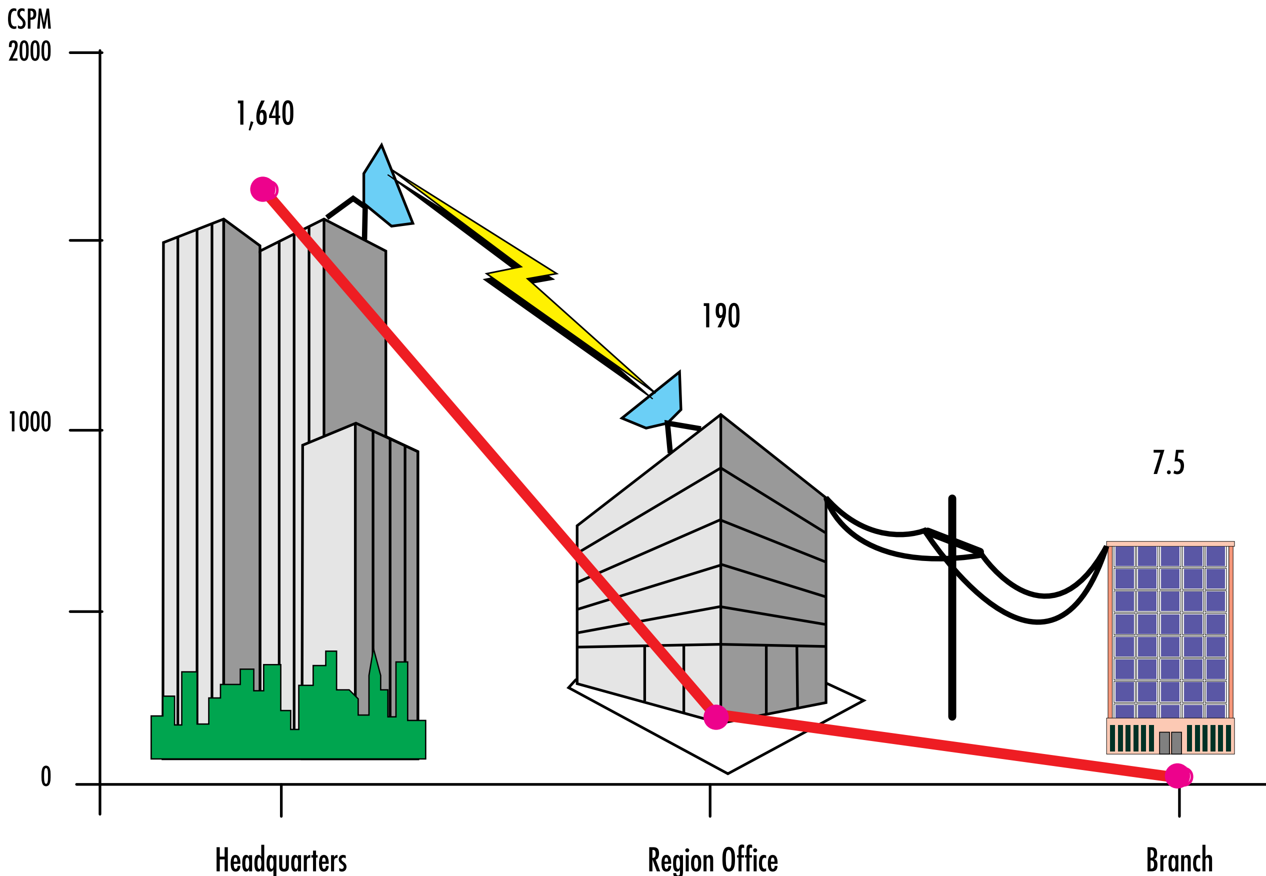


Figure 3. POS throughput

MHz PS/2, RS/6000 (Models 220, 520, and 580), and the HP 9000 Model 827S was less than 5%.

- The Secondary CDS server, a 50/25 MHz PS/2, also remained less than 40% utilized.
- The DCE Security and Distributed Local Time servers ran on a 33 MHz PS/2. The CPU of the DCE Security Server remained under 10% utilized except when there was a flurry of DCELOGIN activity. There was no measurable impact when a faster RS/6000 or HP 9000 security server was used.

Security Registry

The following effects of increasing the size of the Security Registry were noted.

- The disk space required for the Security Registry grows by nearly 1 KB per principal and account, which is about 1 MB per 1,000 principals and accounts.
- The memory working set for the DCE Security server grows by about 1 KB for each unique principal currently logged into DCE—about 1 MB per 1,000 unique principals.
- The cell throughput and response time of DCELOGIN are not measurably affected after adding the 1,000 principals and accounts.
- The only negative effects were that both the startup time of the Security Server Daemon (SECD) when the DCE Security Server is restarted, and the initial startup time of each client's Security Client Daemon (SCLIENTD) are about two times longer.

CDS Namespace

The memory and disk requirements for CDS objects are somewhat higher than for Security Registry entities. The effects of increasing the size of the CDS Namespace follow.

Distributed POS Schematic

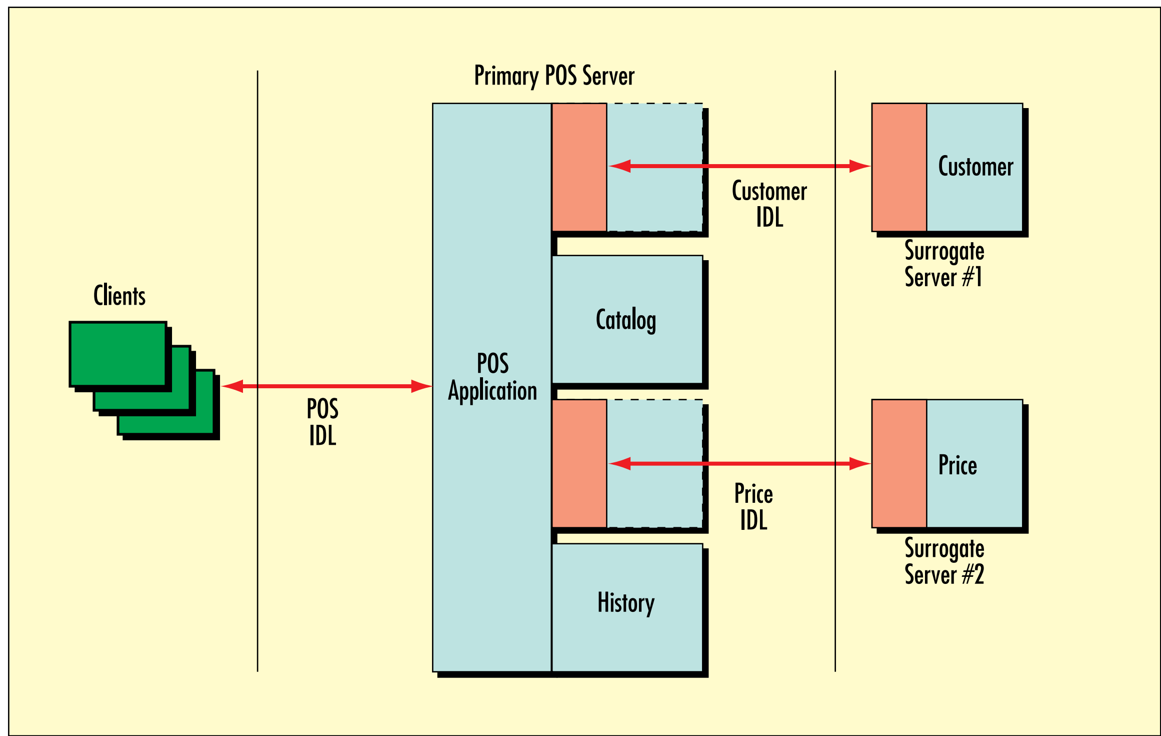


Figure 4. Distributed POS schematic

- Since CDS directory objects require 14.2 KB of CDS server memory and disk space for each object, the 1,000 directory objects in our tests required 14.2 MB. The base OSF® code allocates a maximum size of 14.2 KB per object, regardless of the actual space needed. Generally, only about 11% of the allocated object space is actually used. There is currently no committed plan by the OSF technology provider to address this memory/disk concern.
- CDS program objects each required about 1 KB of CDS server memory and disk space, or 160 KB for the 160 program objects in the CDS Namespace of our test cell.
- There was no measurable effect on cell throughput or client response time.
- The only negative effect was that the CDS Server Daemon (`CDSD`) startup time increased up to 10 times.

POS Application Servers

The POS Application Servers are shown in Figure 2, Sub-topology B. The POS application ran in a distributed configuration with five primary and two surrogate POS servers. Figure 4 illustrates the distribution of the four POS server Remote Procedure Call (RPC) routines. Each client imports a new RPC binding from the CDS server for one of the five primary POS servers before each customer sale. The `rpc_ns_binding_import_next` Application Programming Interface (API) randomly selects one of the five available POS primary servers, providing a random and somewhat even distribution of the workload across the five primary POS servers.

The two surrogate POS servers were shared by the five primary POS servers, allowing all clients to access a single copy of the POS Price and POS Customer databases. The five primary POS servers call the two surrogate servers through nested RPC calls within the POS primary server application.

In the standard configuration, the following hardware supports the POS application:

- The POS Price surrogate server is on an RS/6000 Model 580. The POS Price RPC routine was called most often: 2.5 times for every customer sale. The CPU and disk utilization of this POS Price server was about 15% when the system throughput was 1,640 CPSM.

- The POS Customer surrogate server was on an RS/6000 Model 220. The CPU and disk utilization was about 65% at 1,640 CSPM. When a Model 520 was substituted, the CPU was about 40% utilized.

- All five of the primary POS servers are 33 MHz PS/2 Model 90s or 95s. At 1,640 CSPM, the CPU utilization on all five primary servers was near 100%—definitely the bottleneck in the standard configuration. To achieve higher throughput, the primary POS servers must be upgraded to faster hardware. A single RS/6000 Model 580 primary POS server was fully utilized at about 1,100 CSPM, suggesting five 580s could support about 5,000 POS clients.

  It might seem reasonable to substitute a faster machine for one or two of the primary POS servers. However, this will not help since the `rpc_ns_binding_import_next` DCE API call randomly distributes binding handles for multiple instances of the same application. The CPU capacity of the slower machines would continue to gate the total throughput.

  A general rule-of-thumb is that the total throughput is equal to *n* times the throughput of the slowest machine where *n* is the number of instances of the application server. In effect, the faster machine would respond faster, but would be idle much of the time while most of the requests wait in the queues of the slower machines. Therefore, to improve the total throughput, all instances of an application server must be upgraded to faster machines. This characteristic of DCE currently has no workaround.

Normally, the POS clients import a new binding from CDS for every customer sale. If the clients remain bound to the application server for longer periods, throughput will be much higher. In the standard configuration, the throughput is 1,640 CSPM with rebinding for every customer sale. If rebinding is not done, the throughput increases to 2,080 CSPM. If there is no specific need to rebind frequently, the POS application realizes a 22% improvement in total throughput.

Elimination of frequent rebinding drops utilization of the CDS server CPU from about 40% to near 0%. The CPU utilization of the application server also drops as a result of eliminating handshakes between the client and application server to negotiate the RPC authentication and reestablish the binding. The client's aggregate response time for a complete customer sale is reduced by about 0.5 seconds when rebinding is eliminated. Higher throughput can be achieved while maintaining acceptable end-user response time.

The LAN and OS/2 Database Manager Servers

The LAN and OS/2 Database Manager servers, shown in Figure 2 (Sub-topology C), include the following:

- **Two IBM LAN Server 3.0 domain controllers**: An Advanced LAN Server and an Entry LAN Server support benchmark program sharing, performance data collection, and various network install services. All OS/2 systems are also LAN Requester 3.0 clients.

- **OS/2 Extended Services 1.0 Database Manager Server**: This server supports the database OLTP benchmark clients and runs continuously to provide network loading.

- **NFS for OS/2 Servers**: The two LAN servers, also NFS servers, are used to compare the NFS I/O performance among OS/2, AIX/6000, SunOS/Solaris®, and HP/UX servers.

The presence of the LAN and database load have no measurable effect on our system performance other than increased traffic on the Headquarters Token-Ring LAN.

The memory and disk requirements for CDS objects are somewhat higher than for Security Registry Entities.

The Token-Ring/Ethernet Gateway

The Headquarters' network includes an Ethernet LAN segment to support the HP and Sun

machines (Figure 2, Sub-topology D). The 16 Mbps Token Ring and 10 Mbps Ethernet are linked together by an RS/6000 Model 220 acting as a gateway router.

The Ethernet begins as a thick-wire connection to the RS/6000 gateway, then to thin-wire, and finally to 10BaseT terminating at the Sun SPARCserver 1000. The HP 9000 827S server and 730 client are attached to the thin wire; the Sun SPARCstation® LX is attached to the thick wire.

Two simple tests were run to evaluate the efficiency of this particular Token-Ring-to-Ethernet LAN gateway. We did not intend to compare the performance of specific hardware and operating system platforms, so only general observations about this LAN segment will be discussed.

Sixty percent of the customers surveyed indicate that their systems extend beyond one contiguous location.

- The first Token-Ring-to-Ethernet gateway test measured a 4,096 byte ping from a single OS/2 client in each LAN segment. (The LAN segments are Headquarters' Token-Ring segment, Headquarters' Ethernet segment, Region Token-Ring segment, and the Branch Token-Ring segment.) The OS/2 TCP/IP ping function was used for measurement since it reports its response time with a timer resolution of 31 milliseconds. Each of the selected OS/2 workstations ping each of the 15 PS/2, RS/6000, HP, and Sun servers.

  Within each subnetwork (Headquarters, Region, and Branch), the 4 KB ping responded in 31-34 milliseconds (one timer tick). A ping between the Region and Branch (null modem) took about 2,580 milliseconds, between the Region and Headquarters (4 Mbps link) about 62-64 milliseconds, and crossing the Token-Ring/Ethernet gateway added an additional 10-30 milliseconds.
- The second gateway test was to indirectly access the four POS data repositories using the NFS `mount` command. For this test, a single instance of the POS primary server was running.

  Although applications usually employ a Relational Database Management System (RDBMS) to manage their data, the POS data was stored in binary files and accessed using portable ANSI®/POSIX C language calls. This was a result of our objective of portability and comparability across hardware and software platforms. Using a specific RDBMS would have precluded this.

The results of these NFS-redirected I/O tests are normalized. The results of independent external performance tests are used to establish the basis for normalization of these servers. Accessing the POS data via NFS `mount` on an RS/6000 on the Headquarters' 16 Mbps Token Ring was assigned an efficiency factor of 1.0 (100%). When accessing data on the HP and Sun servers on the Ethernet, two levels of degradation are encountered:

- When the NFS mount was from a Token-Ring-attached POS Primary server through the gateway to the Ethernet-attached servers, the efficiency was 0.75—a 25% loss in maximum throughput.
- When the POS Primary server is moved to the gateway server and the NFS `mount` is performed on this gateway server, communication is directly to the Ethernet adapter. The efficiency improved to 0.88—only a 12% loss in maximum throughput.

The following additional observations were made during the gateway tests:

- Although the line speed of Ethernet is 10 Mbps, the Sniffer Network Analyzer indicated that the Ethernet was less than 20% utilized. The CPU and disk of the HP and Sun NFS servers were less than 30% utilized. This suggests that the bottleneck is the communication adapter, not the capacity of the HP or Sun NFS servers.
- When POS Primary servers are switched among the RS/6000 Models 580, 520, and 220, the efficiency ratio is consistent. This suggests that server congestion and arrival rate are not factors in this test.
- When the results of the independent performance tests on the RS/6000, Sun, and HP servers are compared to our NFS throughput results, the inconsistencies further suggest a network hardware bottleneck.

Remote Offices and WANs

Sixty percent of the customers surveyed indicate that their systems extend beyond one contiguous location. It is therefore interesting to look at options for linking remote locations.

Region Office (Figure 2, Sub-topology F): Two servers in the Region Office that are used only for the "one versus two cell" testing are discussed later in this article. These servers are not used in the standard configuration.

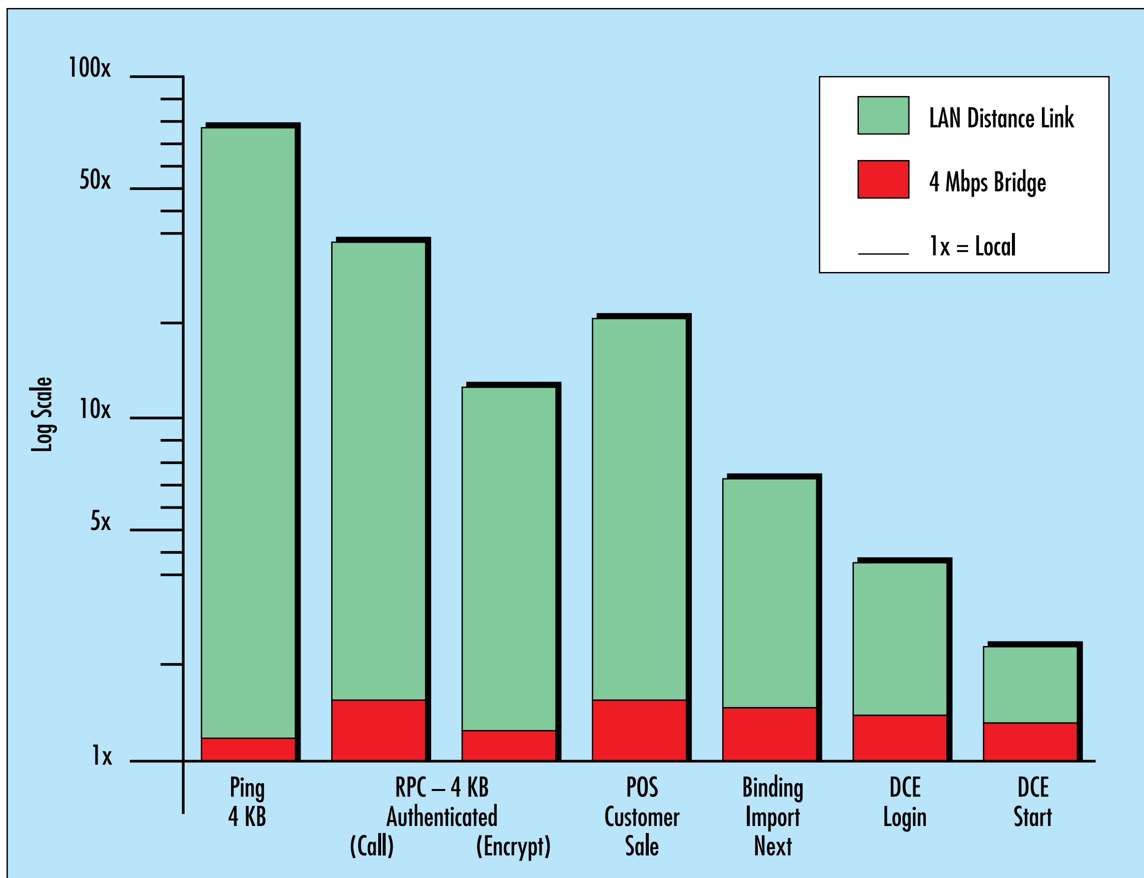


Figure 5. WAN response time cost

The DCE clients in the Region Office are OS/2, OS/2 WARP, and Windows. In the standard configuration, these clients can exert the workload of 190 clients, or 190 CSPM. The 190 CSPM includes the 7.5 CSPM workload of the Branch Office.

Branch Office (Figure 2, Sub-topology H): The Branch Office represents a remote location or work-at-home user connected to the Region Office using IBM LAN Distance and a modem. For our tests, we installed two PS/2 clients on the Branch Token Ring: one OS/2 client and one Windows client. The maximum throughput of the Branch Office clients is gated by the null modem line speed.

High-Speed Link (Figure 2, Sub-topology E): A typical high-speed link is 1 Mbps. We did not have access to a link of this speed, so we implemented a 4 Mbps link. Two PS/2s with two Token-Ring adapters were installed in the local office (16 Mbps) rings of both the Headquarters and the Region. These two bridges were connected by an isolated 4 Mbps Token Ring attached to their second adapter. The IBM Token-Ring Network Bridge Program ran on the bridges.

When running the entire system at 1,640 CSPM, the network analyzer shows that less than 0.8 Mbps flows across the 4 Mbps link. The throughput exerted by the Region Office is 190 CSPM. Therefore, if we assume a 1 Mbps link, the 0.8 Mbps traffic is nearly full utilization of the link. Thus, 190 clients would be about the maximum that a single 1 Mbps link could support.

Low-Speed Link: The Branch Office is connected to the Region Office using an IBM LAN Distance low-speed link (Figure 2, Sub-topology G). The physical link is a null modem set at 38,400 bps.

We did encounter a problem configuring the Windows client in the Branch Office. The increased response time across the null modem caused an unidentified time-out situation, which terminated the Windows 3.1 desktop. When the Novell® LAN Workplace 4.1 was replaced with the IBM TCP transport for DOS, the problem was

eliminated. The IBM TCP transport for DOS is slower than the Novell transport or the other two non-IBM transports that we tested, and it would not normally have been our first choice.

Several tests were performed to evaluate the performance impact of traversing the links between Headquarters and the Region Office, and between the Region and Branch Offices. Figure 5 shows the portion of response times attributable to each link for the following tests. A normalized value of 1.0 is assigned to each test when performed locally on the Headquarters' Token-Ring network. These measurements are from a single OS/2 client in an otherwise idle system.

- **TCP/IP 4 KB ping**: Probably the closest application to the TCP/IP transport level is ping. Since it is also the most subject to the line speed of each LAN segment, it should be expected to have the greatest degradation across the WANs.

  The Region-to-Headquarters ping is two times slower than the local ping. This is good considering the 4:1 difference between the 16 and 4 Mbps line speeds.

  Between the Branch and Region (38,400 bps), the ratio is 75:1, whereas the line-speed ratio is 426:1.

  The ratio is 77:1 between the Branch and Headquarters (across both WANs).

- **Authenticated RPC 4 KB Data Transfer**: This test used an application program to transfer 4 KB of data from a client to a server in an authenticated RPC call. This test should perform better than the ping test, but since the application is still data transfer-intensive, the degradation remains high because of the effect of the line speeds.

  Using DCE Security call-level authentication, the ratio was 36:1 across the 38,400 bps link and 1.4:1 across the 4 Mbps link.

  Using DCE Security packet-privacy (encryption) authentication level, the ratio was 8:1 across the 38,400 bps link and 1.1:1 across the 4 Mbps link. The ratio improved because the larger portion of the total response time was expended by the server and client encrypting and decoding the data packets.

- **POS Customer Sale**: In real life, the time spent within the client and server application is much longer than the time spent actually moving data across the network. Therefore, we should expect the data transfer ratio to improve to a more acceptable level since the RPC portion of the POS application is less than 5% of the aggregate response time when run

Major AIX Events in Europe

AIX Systems Europe announces two major events in Europe to share IBM's strategy for AIX and the RISC System/6000 with Customers, Business Partners, and other professionals interested in working with AIX and the RISC System/6000.

Each event will consist of main tent and technical electives. The main tent sessions will be presentations by key IBM executives discussing IBM's strategies. The technical electives will be presented by IBM's technical experts and will cover a broad range of topics.

AIX in Focus

Date: February 6-10, 1995
Audience: Technical AIX Professionals, Software Developers, Business Partners
Topics: IBM's Technology Principles
Middleware (Databases, Networking, and so on)
"From Laptop to Supercomputer"
Object Technologies
Competition

The Customer Colloquium

Date: February 15-17, 1995
Audience: Customers
Topics: Re-engineering Your Business
The IT Imperatives
Solutions – Not Just Technology

For more information on either event, contact either of the following:

John Childs
(UK): 44-256 344875

Richard Sandbaek
(UK): 44-256 344661

locally in Headquarters (the average data size was 3.2 KB, with a range of 1 to 16,384 bytes).

The POS ratio between the Branch and Region on the 38,400 bps link was 9.2:1, and it was 1.5:1 between the Region and Headquarters.

- **DCE Calls**: Although many DCE functions and API calls make some RPC calls under the covers, most response time is spent in the client DCE programs.
  – **DCELOGIN**: The ratio was 1.8:1 across the 38,400 bps link and 1.4:1 across the 4 Mbps link.
  – **rpc_ns_binding_import_next**: The ratio was 2.3:1 across the 38,400 bps link and 1.5:1 across the 4 Mbps link.
  – **DCESTART**: The ratio was 1.4:1 across the 38,400 bps link and 1.3:1 across the 4 Mbps link.

End-user response time across the WAN is closely tied to the frequency and data size of data transfer operations. The POS application makes an average of 5.5 RPC calls and 7 DCE API calls for each customer sale. The overall response time ratios for a complete POS customer sale (null modem 9.2:1 and 4 Mbps link 1.5:1) are probably more representative of many interactive end-user applications across these specific WANs.

The two clients in the Branch Office can saturate the 38,400 bps link at 7.5 CSPM. So under the best conditions, this LAN Distance link can handle up to 7.5 clients. In reality, LAN Distance is intended for a single client, and extrapolating from our results, a minimum 5,120 bps link would be sufficient for one client.

When we initially installed the LAN Distance link, some LAN Server 3.0 Client time-out problems were encountered due to the longer response times across the null modem. The following changes were recommend by our LAN Distance performance analyst:

- In `IBMLAN.INI`
  – `wrkheuristics` bits 0, 6, 8, 9, and 11 were changed to 0
  – `sizeworkbuf` was changed from 512 to 4096
- In `PROTOCOL.INI` (`NETBEUI.NIF` section)
  – `dlcretries` was changed to 20
  – TI was changed from 30000 to 60000
  – T1 was changed from 500 to 10000
  – T2 was changed from 200 to 2000

One Versus Two DCE Cells

We tried adding a second DCE cell server and a POS application server in the Region Office to handle both the Region and Branch clients' DCE and POS activity, while the Headquarters' cell handled only the clients on the Headquarters' network. LAN Server activity was the only traffic across the 4 Mbps link between Headquarters and the Region Office. There was no DCE Global Directory Server (GDS) for this test. Results were as follows:

- The throughput in Headquarters drops from 1,640 CSPM to 1,570 CSPM because of the reduced client horsepower. When we started more clients to replace those in the other cell, the throughput returned to 1,640 CSPM; therefore, the capacity of the Headquarters' DCE cell was not changed by splitting the Region and Branch into a new cell.
- The throughput in the Region Office remained at 190 CSPM with a PS/2 DCE server and an RS/6000 Model 520 POS server in the Region DCE cell. Since the RS/6000 Model 520 can support about 360 POS clients in this configuration, the 190 CSPM is clearly a limitation of the client's ability to exert workload rather than a limitation of either the WAN or POS servers when configured in a single cell.

Implementing two DCE cells to eliminate traffic across the 4 Mbps WAN improved the aggregate throughput. However, a similar improvement could be achieved by adding the additional POS application server in the original single-cell configuration. The only performance gain from this two-cell experiment was a small improvement in the response time in the Region and Branch by eliminating the hop across the 4 Mbps WAN.

This two-cell configuration could result in a performance improvement if there were a resource bottleneck in the original single DCE cell. In our test configuration, we could expect a resource bottleneck if the arrival rate was increased to 4,000 or 5,000 CSPM. Both the 16 Mbps Token Ring and the PS/2 CDS server could become saturated if we had sufficient client hardware to drive the POS workload.

Summary

The following summarizes the findings from the DCE performance study:

- IBM DCE 1.2 for OS/2 and AIX provide good performance and interoperability in our

testing. IBM DCE for AIX 1.3 (OSF DCE 1.0.3) is now available. When we ran the new version on our test system, we found the new version to be compatible with OS/2 DCE 1.2 clients, IBM DCE for Windows 1.0 clients, and our current HP DCE/9000 1.2 installation. There were no significant differences in the system performance.

- Performance problems have been encountered while integrating the HP 9000-827S system into our test environment stemming from our specific Token-Ring/Ethernet gateway and network hardware configuration.
- Using WANs provided satisfactory performance, given the line speeds. No DCE interoperability problems were encountered using IBM LAN Distance or IBM Token-Ring Bridge software.
- We encountered no interoperability or performance concerns while integrating OS/2, Windows, and AIX DCE clients.
- We encountered no network contention problems with DCE (TCP/IP), LAN Server (NetBIOS), and OS/2 Database Manager running concurrently.
- The overall performance of our test system is well-behaved, and we did not encounter any bottlenecks that would preclude expanding to much higher workloads.
  – The OS/2-based DCE configuration could be expanded to 4,000 or 5,000 clients in a single DCE cell. The limiting factor would be the single 16 Mbps Token Ring.
  – Using an AIX-based DCE configuration, the estimated capacity of our single DCE cell would be even higher. The capacity of the single 16 Mbps network would probably be the limiting factor.

References

Corn, Cindy; Li, Tim; Pekowski, Ray; and Santeford, Bob. “Performance of Key Functions in DCE for OS/2.” *IBM Personal Systems Technical Solutions.* January/February 1994.

Perry, Benetta and Russell, Bob. “IBM DCE for OS/2 Multi-user Application Performance.” *IBM Personal Systems Technical Solutions.* January/February 1994.

Perry, Benetta and Russell, Bob. “Distributed Performance Characteristics of IBM DCE for OS/2.” *IBM Personal Systems Technical Solutions.* July/August 1994.

Pekowski, Ray. “Performance of Key Services in IBM DCE Client for Windows.” IBM White Paper. Available from MKTTOOLS in the DCE Performance package from your IBM representative.

Bob Russell, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Russell is team leader for the LAN Systems Performance Workgroup Environment team. His most recent positions include performance analysis for OS/2 Database Manager and DCE for OS/2.

RISC System/6000 Developer’s Discount Program

As a developer, you need the latest equipment for development and testing. Your ability to create quality applications and solutions depends on having the latest technology available. And at prices you can afford.

IBM offers special purchase programs for qualified hardware and software developers. These programs include the following:

- Hardware
- Software
- Peripherals

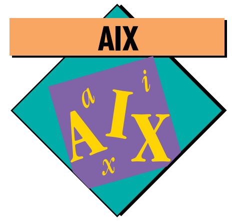
The program includes systems ranging from the P40 through the Symmetric Multiprocessors (SMP). With discounts as great as 50% and up to 10 machines in one year, you can’t afford not to evaluate the program.

Programs may vary outside the United States. Contact your local representative for more information.

For information, call **1-800-222-2363** and ask for extension 63. ■

IBM reserves the right to change terms and conditions of this program without notice.

Print Spooling Subsystem in AIX 4.1



By Eddie Ho and Ravi Mandava

This article describes the Print Spooling Subsystem in AIX 4.1, which provides many improvements over AIX 3.2. It simplifies the overall task of system management—particularly queue management—by providing better job control and improved usability.

In a typical client/server environment, servers are dedicated to a special purpose, such as databases, applications, communications, or printing. Each server must be scalable and flexible in order to meet constantly changing user requirements. The AIX 4.1 Print Spooling Subsystem has been reengineered to meet this need. Enhancements include the following:

- Overall user interface improvements
  – Restructuring all System Management Interface Tool (SMIT) panels and the navigation flow
  – Integrating with Visual System Management (VSM) for the desktop user
- Better job control capability
  – Holding and releasing a job
  – Moving a job to a different queue
  – Validating job options before spooling
- Standardized support for OEM printer attachment
- Printer colon file enhancements
- Streams-based serial-printer-discipline support

Details of these enhancements are described in *AIX Version 4.1 System Management Guide: Operating System and Devices* (SC23-2525) and *AIX Version 4.1 General Programming Concepts: Writing Programs* (SC23-2533).

Printing Overview

The AIX 4.1Print Spooling Subsystem is a robust platform to meet the needs of client/server applications in a distributed commercial environment. The subsystem is flexible enough to scale from desktop ASCII terminals with a printer to high-end systems with massive printing needs. The subsystem, which can be used for both printing and plotting, supports the following types of devices:

- ASCII terminal with an attached printer (transparent printing)
- IBM Xstation with a printer attached to its serial or parallel port
- Printer attached to the 7318 (P10) Network Communication Server's parallel port
- Printers attached to a RISC System/6000 using the built-in serial ports or the multiport adapter with 8, 16, or 128 ports
- LAN-based direct printers, such as the HP JetDirect® or IBM 4033 LAN Connection
- Remote printers accessed by routing a job from a local to a remote queue, with the remote system being AIX, System V Interface Definition (SVID), Berkeley Software Distribution (BSD), or any host that supports the Line Printer Daemon (LPD) protocol

Figure 1 summarizes the attachment types in a workgroup environment.

The printing subsystem can support printers with multiple data streams and printers with a country/language locale such as single byte, Extended UNIX Code (EUC), or Unicode. It can also support a range of speeds from personal printing to high-speed printing in a warehouse environment. Supported printer data streams include the following:

- PostScript®
- HP Printer Control Language (PCL)
- Graphics Language (GL) for plotters

Print Server

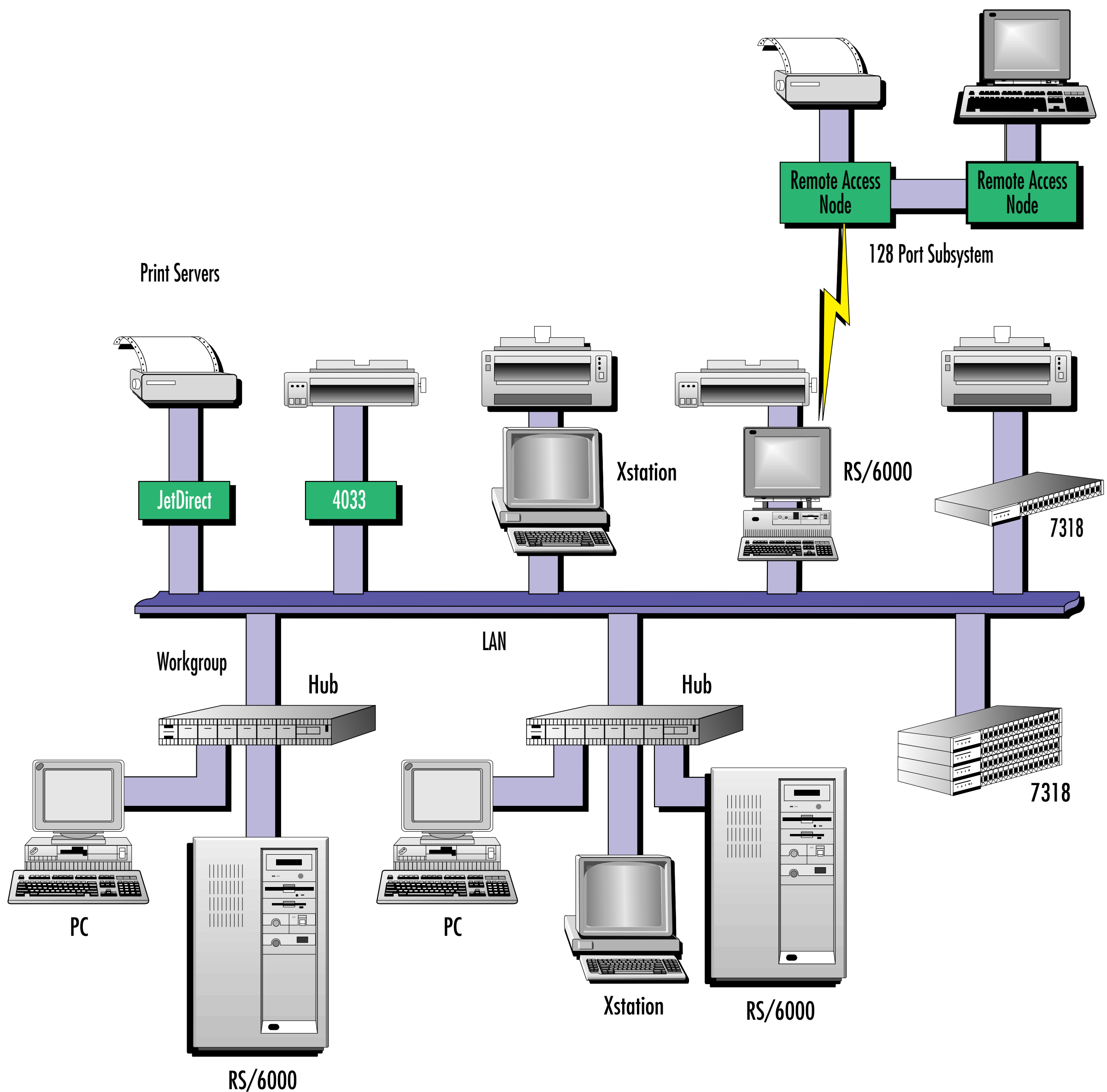


Figure 1. Print server in a workgroup environment

- ASCII

The many print options provide the flexibility to meet most business printing requirements. Options include the following:

- Paper-size and page-layout selection
- Print quality, typestyle, and pitch selection
- Page orientation
- Simplex/duplex printing
- Paper-tray selection
- Header/trailer page
- User-configurable filter for output-data customization
- Code-page selection
- Double-wide and double-high printing

Spooling Subsystem Structure

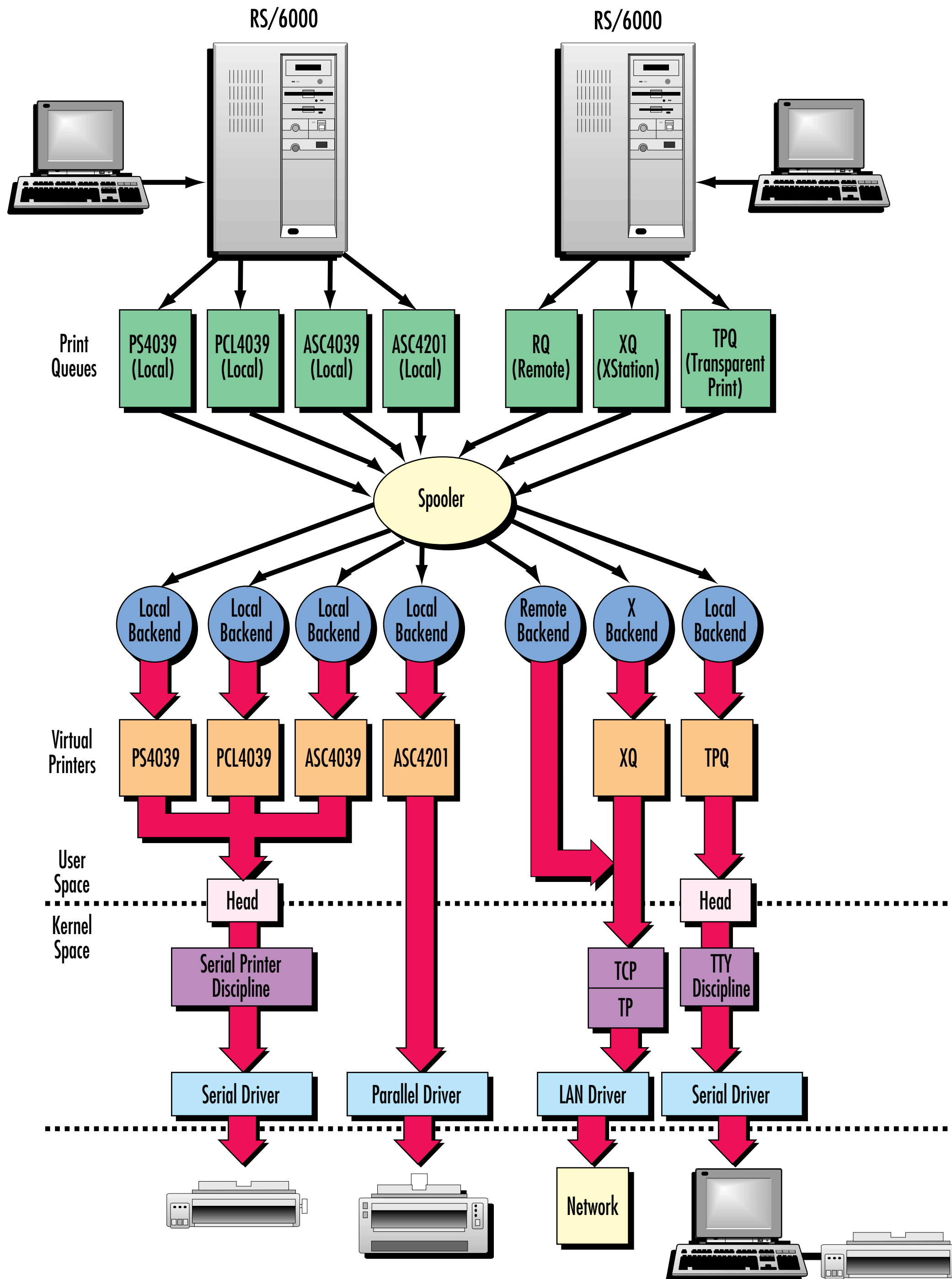


Figure 2. Spooling subsystem structure

Subsystem Infrastructure

AIX 4.1 Print Spooling Subsystem provides the flexibility required in today's printing environment, in which printers often have multiple personalities that allow them to emulate a wide range of other printers. There are two important views of the subsystem: logical view of how printers, virtual printers, and queues interact with each other, and data flow of the file being processed by the subsystem.

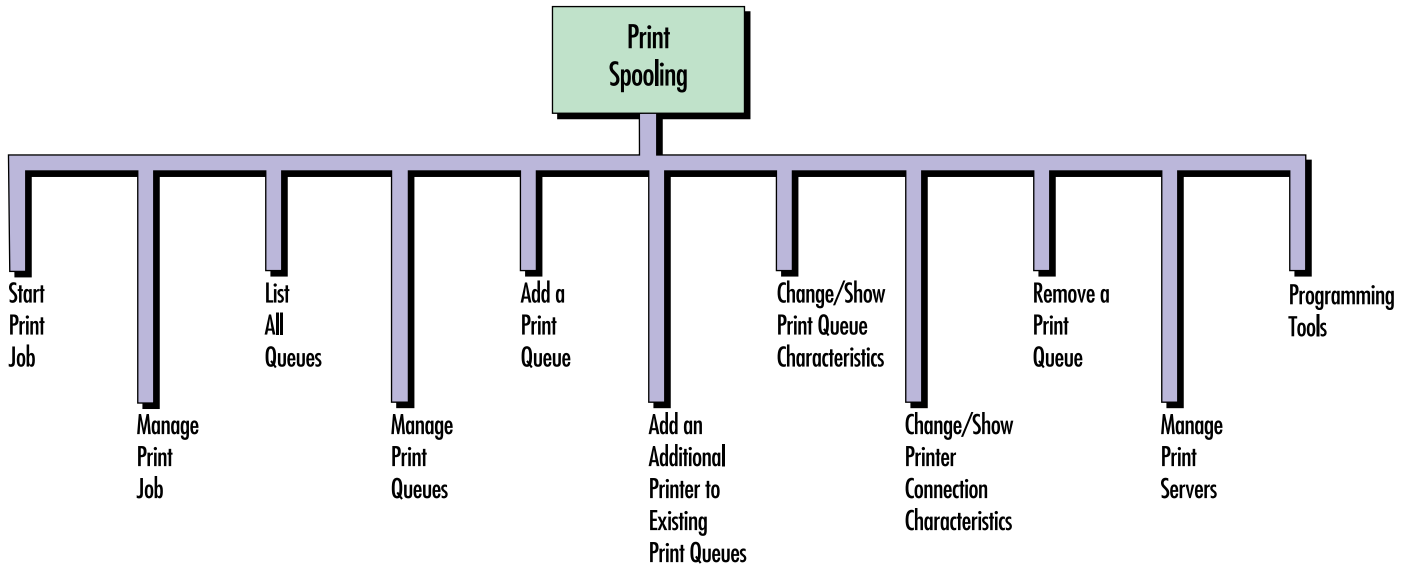


Figure 3. Print spooling SMIT panels

For the administrator, there are three major components in this subsystem:

- **Printers**: Actual devices on which output is printed
- **Virtual printers**: Components that translate the job data into specified data streams that can be supported by the printer
- **Print queues**: Queues that handle print requests

Figure 2 shows the various queues and the data path when users submit jobs to a print queue.

User-Interface Improvements

Many improvements have been made based on experience and customer feedback from AIX 3.2. With AIX 3.2, users had difficulties in configuring and changing printers and the associated queues. This difficulty has been alleviated in AIX 4.1 with the following improvements:

- Configuration is provided as part of the graphical VSM. The VSM tool allows users to create print queues and devices by using the familiar drag-and-drop interface.
- The print-queue and device configuration procedure is simplified. The entire internal infrastructure of a queue and a device are created in a single path based on the type of attachment. This can eliminate the need for the system administrator to remember the sequence of operations.
- The interface for job submission is improved. The SMIT interface shows all feature options that are unique to a model. In AIX 3.2, only generic flags are provided in SMIT, and print-formatting options that are unique to a printer model can be specified only from the command line. This requires you to deal with two user interfaces and to have a full understanding of the printer capabilities.
- The concept of virtual printer and queue device is not externalized to the user, who only has to create and administer print queues. The underlying objects are managed automatically.
- A consistent SMIT interface is provided throughout all panels, eliminating use of the curses interface in some panels.
- For remote queue characteristics, users have options provided for viewing and changing queue characteristics during job submission.

Figure 3 shows the hierarchy of the main panel.

Job Control Enhancements

The following improvements give the administrator better control over each job in a loaded environment to alleviate downtime during printer problems and to give better feedback to the job submitter.

Hold and Release

Job holding and releasing is useful for managing printing throughput in a congested environment. For example, a large print job can be held and printed during off hours. In AIX 4.1, each job can be held (by the job submitter or the system administrator) when submitted or queued, but cannot be held while being printed.

Move a Job to a Different Queue

During printer outage, inactive jobs can be moved to a different print queue using a different printer, which can improve availability. This can be done by the job submitter or the system administrator.

Job Option Validation

Job option validation is provided during submission. This is useful in detecting user errors, such as invalid flags and values, and allows you to troubleshoot the problems quickly. In AIX 3.2, job submission errors are not detected until the job is being sent to a printer, which causes delays.

Non-IBM Printer Support

An ease-of-use interface supports the attachment of non-IBM printers, such as the HP JetDirect. In AIX 3.2, the cumbersome `mkvirprt` command incorporates non-IBM configurations. In AIX 4.1, the SMIT interface is recommended and incorporated through *attachment files*. Attachment files contain links to the manufacturer-provided SMIT screens from the print spooling panels.

Printer Colon File Enhancements

The *printer colon file* is a device-specific file that contains the unique printer-control sequences for a particular model. There is one colon file for each type of data stream that a printer supports. Enhancements such as the following are added to improve usability.

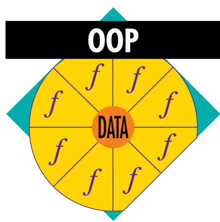
- Support `ksh` instead of `bsh` while executing pipelines, filters, and commands. This allows AIX to conform with the default shell standard.
- Accept hexadecimal notation in addition to the current octal notation. The representation is -xXX, where XX is the hexadecimal representation of the character value.
- Generate selection lists in SMIT panels and flag validation. A limits field has been added to the colon files to provide this.
- Execute AIX shell commands with output directed to the specified attributes. The operator syntax is `%`"<command>..."`. This is helpful if the user is not familiar with all the values of an attribute when submitting a print job.
- Provide capability to extract data matching a specific regular expression. This is useful when generating a font selection list while submitting a job. The operator syntax is `%#xx"...@..."` where `xx` is an attribute that contains the source data.

Streams-based Discipline Module

For local printing, there are two key attachments: serial and parallel. The printing behavior is controlled by the discipline module feeding data to the device drivers. This module is used to format print data and manage current printer status. Due to the migration to the Symmetric Multiprocessor (SMP) environment and the new Streams-based TTY subsystem, the serial discipline module is redesigned to use Streams. This allows the flexibility of real-time control and customization of the printing discipline module.

Eddie Ho, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Ho is a senior programmer in the AIX Executive Briefing Center. He has a BS in Computer Science from the University of Wisconsin and an MS in Computer Science from North Dakota State University.

Ravi Mandava, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Mandava is a staff programmer in AIX Base Commands and Libraries. He has a BS in Mechanical Engineering and an MS in Computer Science from Osmania University, and an MSIM from the Indian Institute of Technology in India.



OO Programming Utilized by Rochester Lab

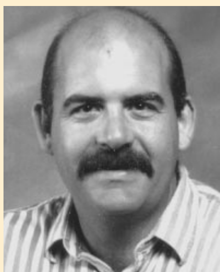
By Bill Berg, Chris Jones, and Mike Tomashek

Object-Oriented (OO) programming and C++ were used extensively in the effort to produce a level of System Licensed Internal Code (SLIC) to run on the RISC-based AS/400®. This article[1] discusses the justification, experience, and early results of that project.

The Need to Shift SLIC to OO

As the IBM Rochester Lab approached moving the AS/400 operating system from the IMPI processor architecture used since its introduction to a new PowerPC-based RISC architecture, many changes were required in the lower layers. A comprehensive assessment of our strategy yielded these observations:

- A major design impact was evident for low-level software components that are hardware-dependent by nature.
- Storage Management, the Bus Manager, Machine Indices, and common functions like exception management, tasking, and queuing were impacted to some degree.
- Traditional processor microcode (Horizontal Microcode, or HMC on the AS/400) would not exist on a RISC system, so this had to be implemented as part of the new operating system.
- The existing Licensed Internal Code (LIC) was de-evolving due to frequent change and upgrade. Programmer productivity and quality measurement numbers would be lower without some technology change.


Bill Berg

Mike Tomashek

Rewrite or Port

There were two ways to attack this problem:

- Redesign/rewrite the low-level components from scratch, expecting a more extensible design and implementation. (Rewriting in a new language while keeping the old structure was not considered acceptable.)
- Attempt to port these components, with minimal changes to accommodate the new hardware, to bring up the system quickly. This, therefore, meant retaining existing data structures and control flow.

In either case, components not affected by the processor change would be migrated with as little modification as possible; for example, database, communications, and so on.

There were questions as to the feasibility of any redesign at all. The AS/400 was strongly based on design elements brought forward from the S/38, and essentially had 15 major functional releases that had not only added functions, but also had been performance-tuned. To many, it was inconceivable that major portions of this software could be rewritten in just two years.

Training: 'In-Demand' Skills Enhancement

There are ample warnings in print about the fallacies of trying to do your first OO project on a short schedule with inexperienced developers.

[1]Reprinted with permission from the Spring 1994 issue of *ASsociation/400*, the newsletter for the Association of AS/400 Business Partners.

We hoped our case would be different for the following reasons:

- The technology had matured since the initial horror stories. There was a better set of information available. Grady Booch, Bjarne Stroustrup, James O. Coplien, Scott Meyers, and Stanley B. Lippman all had excellent books out, some in second editions.
- The language was "ready." Templates and exceptions were approved and would be available in our Toronto-sourced compiler in time to meet our needs.
- It was possible to hire quality consulting skills. And, we did just that! We hired a solid consultant who put our people through six weeks of intensive training. When not teaching, he sat through many whiteboard talks while those of us with some self-taught OO/C++ skills bounced fledgling designs off him. These sessions yielded diagrams and class descriptions for our recent class graduates. The synergy of domain experts working with an OO design and a development expert is a common experience of the SLIC project. We started with a consultant, but now the most sought after internal skill is the ability to leverage domain expertise to OO design through the "knowledge map" of a more experienced OO person. Building these OO design skills is an important output of the SLIC project.

The half-life of new OO/C++ knowledge appears to be about three weeks. New converts must immediately code and design. The switch to "thinking objects" is hard. We still don't have a magic bullet for getting there. We observed that if ten people hammered out a project, two or three would become OO designers—maybe!

This education also contributed to morale. Our 150 people benefited from cutting-edge design/coding knowledge—highly marketable skills. More importantly, these skills are critical to our continued development efforts and our support of independent developers in their OO efforts.

Background

Both the AS/400 and the new RISC machine divide the operating system in two pieces: OS/400® and SLIC. (See Figure 1.) SLIC supports an architected interface for OS/400 called the Machine Interface (MI), which is effectively a high-level instruction set. This MI instruction set is targeted by the compilers rather than the actual hardware instructions. (SLIC includes a translator which converts MI programs into a combination of hardware instructions and SLIC calls.)

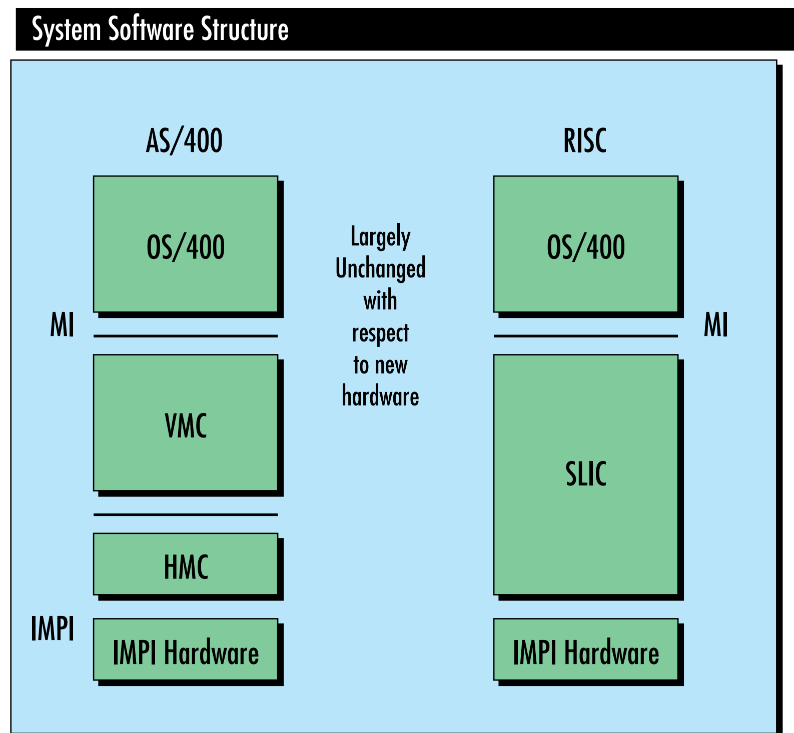


Figure 1. System software structure

The AS/400 is 'object-based': its objects are Abstract Data Types (ADTs) from an MI perspective. The MI provides Application Programming Interfaces (APIs) to represent a wide range of facilities and functions. For example, queues, programs, user profiles, and all I/O facilities are represented by these ADTs in the existing AS/400 MI. To the MI, these are real ADTs consisting of data and code. But, below the MI, they are implemented as global data structures that SLIC components are free to read and modify.

The experience of dealing with this encapsulation of APIs at the MI and maintaining compatibility for code at the MI was useful. Existing MI objects provided no experience with inheritance or polymorphism, however, so this still needed to be learned. Since we added the C++ code into SLIC mostly from the bottom up, we gained experience in interfacing existing functional decomposition-style code with new OO/C++ code. In general, this technique was a positive experience. Typically, the old code is improved since a macro or call is introduced to interface to an abstract C++ facility, and often a large chunk

of existing code that exhibited inappropriate coupling with another component is removed. We have less experience with wrapping existing function with objects, but this was done in some places successfully. The existence of a firm base and firm target definitely assisted this project. At the simplest, the mission could be described as "run the MI on the RISC processor." Much more was actually involved, but that core requirement provided a clear anchor against "creeping elegance." There was always existing reality (existing code interfaces/MI) against which we could test new abstractions.

The Process

First, we benefited from the following:

The ability to iterate in design is a cornerstone of OO.

- Teams naturally formed that paired a domain (AS/400) knowledgeable veteran with a UNIX- and C-literate new hire.
- Domain experts with OO and C++ knowledge were given fewer assignments to free them to consult with those less skilled in these categories.
- Everyone participated in coding. A representative from each major component formed a "petal" team that met weekly to make broad policy and direction decisions. These meetings allowed the team to raise their heads above water to understand what was really happening at a project level.

The petal name comes from a chart that showed the key domain experts and OO skills assigned to various major componentry of SLIC (database, communications, storage management, I/O, and so on). The chart looked like a flower with the chief designer being in the center and the other component experts around the outside as "petals."

The ability to iterate in design is a cornerstone of OO. As we looked at the work, it was also clear that some clear deliverables needed to be created that allowed us to honestly gauge progress. The solution was "Bring Up Binds" (BUBs) which gave rise to the classic phrase, "This BUB's for you." Each BUB had a clear set of functions that let us iterate at the interface level by putting together a large set of objects and verifying/learning at early stages. The process included the following.

- **Early design**: This involved a short text design description, Booch diagrams, basic header files, some main classes, and key methods.
- **Interface availability**: Completed header files provided the complete interface required by a set of clients.
- **Function availability**: The developer unit tested the classes provided, and the client could reasonably expect the function would operate.

We started in June 1992, iterated within components through the Fall, and pulled together the first BUB in early December. It took nearly two full months to get this bind to the point that a screen was displayed by the infant operating system. Much time was taken scaffolding and creating functions we discovered we needed, not due to bugs. The type checking provided by C++ is a major benefit in getting initial versions of code running. C++ does not intrinsically assist with issues of concurrency, assumptions about side effects made in old code, or in failing to set a pointer correctly. The code must be structured so that type checking is utilized as much as possible, and to leverage reviews and testing, to ensure other problems are caught.

The following points are important to the construction as well as the future of OO systems:

- Functionality must be staged through the delivery of objects to at least minimally meet the contract provided by their interface with a simple implementation.
- At the same time, the implementation of these newly delivered objects must be able to be upgraded at a later date without breaking clients.

The Results

The SLIC project has over 750,000 lines of code and 6,000 classes in C++. Most of this code is very low level and performance sensitive. The following key elements were converted to C++:

- **Storage Management**: This code provides the abstraction of Single Level Storage (a unique feature of the AS/400 architecture).
- **I/O Support**: The base support to communicate the buses and manage hardware resources was converted to C++. The SLIC support for DASD, tape, and optical devices were also converted.

- **HMC Replacement**: With the advent of RISC processors, many higher level operations provided by the HMC (queuing, tasking, hold record support) needed to be coded in SLIC.
- **Service Code**: Support for screens and tools, provided with SLIC to service the machine.

Execution Experience

Not surprisingly, there were performance challenges. (This is common and predictable.) In general, the C++ code has been very "tune-able." The ability to change implementations without changing the interface is extremely powerful. C++ provides solid mechanisms like in-lining methods and object containment that are good for performance tuning, and the ability to override new and optimize storage allocation is always a good tuning knob. It is far easier to tune a well-understood design.

OO—A "Do as you learn" Technology

OO is one of the first effective ways developers have had to create and communicate abstractions. As the project progressed, we had to stop thinking about how we did our job and start thinking about what we did. It was common to spend much time trying to "OO-ize" a data structure that had been in the solution space for years, only to discover that the data structure may be unimportant, or even inappropriate.

Through the project, the scope of tools has increased. First, we only had a few re-usable classes of the stack, string, list, or map ilk. Then, we acquired idioms like when-done which allows a possibly asynchronous server to call back to the desired method after a completed operation. This idiom was extensively used. Others include:

- **Constructor branch table**: Allows the construction of a subclass of a known base without changing to existing code. Support for the new subclass just needs linking into the system.
- **Proxy objects** (also ribbon cables): Allows an object to stand in for another used in lieu of smart pointers when the real object may be destroyed. It can be aware of its proxies and inform them. Clients deal only with proxies. As the project progressed, we heard of frameworks and began to see them in the design. There are two views of a framework.

"The pundits have said you must sneak up on OO, but our experience says the opposite. We had tried 'stick your toe in' approaches before, with non-conclusive results. If the project succeeded, the naysayers claimed, 'It was an isolated, non-real-world set of code.' But, if there were problems, the OO champions claimed, 'There were too many interactions with old code.' This project is large, performance-sensitive, and interacts heavily with existing code. We've come far enough to comfortably say this project will be a big score for OO."

—Mike Tomashek

"A good bridge designer respects the properties of his materials and uses them to enhance the design. Similarly, a good software designer builds on the strengths of her implementation language and—as far as possible—avoids using it in ways that cause problems for implementers."

—Bjarne Stroustrup, designer of C++ and author of *The C++ Programming Language*

"In our experience, the design of classes and objects is an incremental, iterative process. Except for the most trivial abstractions, we've never defined a class exactly right the first time."

—Grady Booch, author of *Object-Oriented Analysis & Design with Applications*

"The knowledge acquired on this project is invaluable; there is no substitute for putting something into production that enhances our AS/400 OO development offering. This also puts us in a better position for joint projects with outside organizations using RISC and C++."

—Bill Berg

– **The client view:** Interfaces the client sees. Normally very stable abstract base classes.

– **The framework extender view:** When extending the framework, developers will subclass from the appropriate classes to create a new function. This is where the "don't call us, we'll call you" framework designation comes from. As a new subclass is produced, the code will simply run at the right time when the framework is used.

In the I/O part of the project, a framework evolved to add new I/O hardware to the system. Clients work with a set of abstract base classes, and when a new piece of I/O gear is added, we

Useful Vocabulary

Abstraction—Process of making complex ideas and structures more understandable by removing the detail and generalizing their behavior. An abstraction is, in many ways, the antithesis of hard-coding.

Class—Templates for defining kinds of objects; usually defined as special cases of each other, organizing information about objects in a natural, intuitive manner.

Framework—A set of classes that provide a set of services for a particular domain.

Idiom—A standard way to solve a class of problems in a given programming language.

Inheritance—When classes are defined in a hierarchy, special cases share all the characteristics of their more general cases.

Iterate—The ability to successively refine a design by improvements.

Polymorphism—Allows methods to be written that generically tell other objects to do something without requiring the sending object to have any knowledge about the way the receiving object will understand the message.

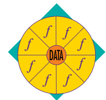
provide a set of subclasses to allow the new gear to operate. Once added, these classes are used automatically by the I/O framework when an operation is requested at the new device.

OO is moving forward as higher level concepts are built on top of the paradigm and languages. Idioms form the first step. An evolving second step is of design patterns. The current top of the abstraction heap is the framework, offering exciting promises.

Conclusion

We are very pleased with our first large scale experience with OO and C++. After some initial performance and object code size scares, SLIC is stabilizing. We are working to create more code commonality. Here are some observations:

- An OO project can be started from scratch assuming solid technology transfer. OO technology is mature enough to use consultants and printed information to bootstrap a project.
- Strong education pays royally, but the new skill must be used right away, and training must be ongoing.
- Start coding quickly—just do it! And iterate, iterate, iterate!
- Explicit, frequent checkpoints showing success/failure are important.
- C++ performs in low-level OS situations.
- Idioms are powerful as both design and coding constructs.



Bill Berg, IBM Corporation, Rochester, Minnesota. Mr. Berg is a programmer in the advanced technologies area of the Rochester lab, exploring the usage of Taligent® and microkernel technologies. He helped design the OO framework used to attach I/O gear to the AS/400 RISC machine. His interests include C++ and the application of OO technology to operating systems.

Chris Jones, IBM Corporation, Rochester, Minnesota. Mr. Jones works as a development manager in the AS/400 Division.

Mike Tomashek, IBM Corporation, Rochester, Minnesota. Mr. Tomashek is a product manager in the AS/400 Division. He has managed SLIC from its inception. For the past ten years he has designed and managed embedded controllers and operating systems.

Upcoming Object-Oriented Programming Classes

Date	Course	Code	Location
12/05-12/09	Object-Oriented Programming and Design with C++	(Q1073)	Research Triangle Park, NC
12/12-12/16	Object-Oriented Programming and Design with C++	(Q1073)	Charlotte, NC
12/12-12/16	Object-Oriented Software Engineering	(N1498)	Austin, TX
12/13-12/15	SOMobjects Toolkit Developer's Workshop	(N1602)	San Jose, CA
01/16-01/20	Object-Oriented Software Engineering	(N1498)	San Jose, CA

For additional information call, 1-800-IBM-TEACh

AIX Questions

Compiled by Dave Franks

The AIX Solution Provider Technical Support group in Austin, Texas, supports software vendors who are porting applications to AIX. This article is a compilation of questions that are frequently asked by vendors. The name of the responding Technical Support Group staff member appears after each response.

I have a C application and would like to know how to set up my signal handlers to create a full core dump.

To enable a full core dump, set the `SA_FULLDUMP` flag in the `sigaction` subroutine for the signal that is to generate a full core dump. If this flag is set when the core is dumped, the data section is included in the core dump. Depending on the signal handler being used, there are many ways to implement this procedure. Figure 1 shows an example.

By default, the user data is not included in a core dump. However, a partial core dump includes the current process' stack, the user structure, and the state of the registers at the time of the fault, which will contain sufficient information for a stack traceback. If an error occurs, `abort()` will produce a core file, and `SA_FULLDUMP` will ensure a full core dump.

—Jeff Simon

```
struct sigaction siga;
siga.sa_handler = SIG_DFL;
siga.sa_flags   = SA_OLDSTYLE;
sigaction(SA_FULLDUMP,&siga,NULL);
abort();
```

Figure 1. Setting the SA_FULLDUMP flag to get a full core dump

What is the danger of turning off quorum checking in two-way mirroring?

The only danger in turning off quorum checking is that if one of the disks becomes inaccessible, the volume group will still vary on, and the user may be unaware that a bad disk needs to be repaired.

The user can automate the process of error notification by adding a stanza to the `/etc/objrepos/errnotify` file via the `odmadd` command. The stanza indicates to the error notification daemon which errors are of interest and what command to run when such an error is encountered by the error logging daemon. The command that is run can be a user-written shell script or application that takes the appropriate action.

More information on this subject is available in InfoExplorer in the section "Error Notification Object Class."

—Priyamvada

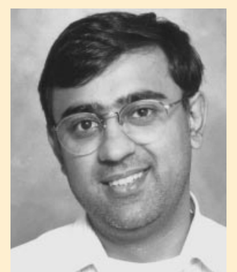
Why does my National Language Support- (NLS-) enabled application fail to find the catalogue file?

There are two issues.

1. If an application does not call `setlocale()` before any of the other NLS functions, such as `catopen()`, the application runs in the default C locale. This locale states that the default messages supplied to `catgets()` should be used and no message catalogue should be opened.
2. Be sure that PTF U425984 is applied. Previously, if the catalogue could not be found, `catopen()` would return an error only if `LANG` was unset or set to an invalid locale. If `LANG` was valid, `catopen()` would return a file descriptor, and calls to `catgets()` failed.

—Ismat Dhanjibhai

Priyamvada



Ismet Dhanjibhai

```
/usr/ucb/nm $1  |egrep ‘ [BAD] ’|cut -c12- | sed -e
    ‘s/^#/ #/’| \
        fgrep -v ___start | sort | uniq >$1.exp
```

Note: There are spaces around `[BAD]` and before the `#` in the substitution string of the `sed` command.

Figure 2. Creating an export list

What is the total number of files that AIX can open at one time?

AIX has an `NFILE` parameter, which refers to the maximum number of open files that AIX 3.2.5 will support for the entire system at one time. This number is hard-coded to 200,000 and cannot be configured.

AIX also has a `NOFILE` parameter that refers to the maximum number of open files allowed per process. This number is hard-coded to 2,000 and cannot be configured.

—Jeff Simon

How can I keep I/O signals from getting detected in `dbx/xde`?

Type `ignore io` before running the application in `dbx`.

—Sai P. Ramanath

Jeff Simon

What is the `kproc` with PID 514 on a `ps aux` and why is it taking so much CPU time?

Process 514 is the “idle process” that runs when there is no other work for the system to do. It runs at the lowest possible CPU priority and does not slow down your system.

—Rick Malone

Sai P. Ramanath

The `-n` option on the `echo` command does not work in AIX 4.1. For example, echo -n “Press return to continue” prints out the `-n` along with the text. How can this be suppressed?

AIX 4.1 no longer supports the `-n` option for the `echo` command in `ksh` and `bsh`. This conforms to the XPG4 requirements. The same result can be obtained in `ksh` with a `\c` within double quotes.

Therefore, your script can be changed to the following:

```
echo “Press return to continue\c”
```

This will work on AIX 3.2.5 and 4.1. The `-n` option is valid only in `csh`.

—Priyamvada

Why are my NFS filesystems not being mounted at system reboot?

The filesystems are mounted from nodes whose IP addresses can only be resolved via Network Information Services (NIS). In AIX, NIS starts after all filesystems with the stanza `mount = true` within `/etc/filesystems`. Because NIS is not active at that time, the mounts fail. The following workarounds may help:

1. Add the hosts to `/etc/hosts`.
2. Make the stanza `mount = false`, and mount these manually.
3. Make the stanza `mount = false`, and make all those filesystems part of a group. (In `/etc/rc.nfs`, after NIS has been started, mount that group.)

—Rick Malone

Can I create an export list from an executable that I have already built?

The script in Figure 2 will create an export list from an executable. From the command line, enter `<script> <executable>`.

—Jeff Simon

I am trying to find the Simple Network Management Protocol (SNMP) protocol in an `IPREPORT` output. The `IPREPORT` raw file was generated with the `IPTRACE` command that used the `-P` flag. Figure 3 defines the `-P` flag for the `IPTRACE` daemon in InfoExplorer 3.2.5.

Why does the SNMP protocol not show up when the `<Protocol>` value to the `-P` is `tcp`?

The SNMP protocol uses User Datagram Protocol (UDP) instead of TCP packets. You were looking for the SNMP/TCP protocol instead of the

SNMP/UDP protocol on the IPTRACE output. We recommend using the following flags for the IPREPORT command used on the IPTRACE file captured:

```
ipreport -rns <IPREPORT raw output file>
```

where:

- -r Decodes Remote Procedure Call (RPC) packets
- -n Includes a packet number to facilitate easy comparison of different output formats
- -s Prepends the protocol specification to every line in a packet

—Rick Malone

How do I modify the "From:" line for mail destined for the outside world?

On all mail clients identified as mail originators, use the following steps to change /etc/sendmail.cf:

1. Edit the file /etc/sendmail.cf using an editor, for example, vi /etc/sendmail.cf.
2. Search for the following text under the TCP Mailer Specification title, rule set S14:

   ```
   #
   # Append full local domain where none exists at all.
   #
   ```

   The vendor found the following in the /etc/sendmail.cf file:

   ```
   #
   # Append full local domain where none exists at all.
   #
   R$+                   $@$1<@LOCAL.D>
   ```
3. Figure 4 shows the suggested changes.
4. Enter the following commands:

   ```
   sendmail -bz
   refresh -s sendmail
   ```
5. Then try it out by sending mail.

—Rick Malone

```
-P <Protocol>
Records packets that use the protocol specified by the
<Protocol> variable.

The <Protocol> variable can be a decimal number or name
from the following file:

/etc/protocols
```

Figure 3. The –P flag for the IPTRACE daemon

```
#
#  Leave all other typical domain addresses alone.
#
R$+<$+>                  $@$1<$2>
#
#  Append full local domain where none exists at all.
#
#R$+                     $@$1<@LOCAL.D>
# If it is a local name, append local host. Else, append
   domain.
R$=N                     $@$1<@LOCAL.D>
R$+                      $@$1<@D>
```

Figure 4. Suggested changes to the /etc/sendmail.cf file

What are if-req, coreq, and prereq in relation to installation and Table of Contents (TOC) format?

An if-req is a relationship between Program Temporary Fixes (PTFs). For example, you may want PTF "A." This PTF specifies that if product "B" is on the system, then PTF "C" must be installed for product "B" before PTF "A" can be applied. In short, product "B" must be upgraded before that PTF can be applied.

A coreq is a PTF required by the PTF you are applying. Both PTFs are for the same product and must be applied before the installation processing completes.

A prereq is a PTF that must be applied before the PTF you have selected. Both PTFs are for the same product.

—Sue Lowe

Sue Lowe

What are the numbers preceding the { symbol within the TOC file, such as in 1:88 3 R G?

The first field in the first number is the volume number (such as the tape volume to which this product or fix relates). In the example, the first tape is where this fix is found.

The second field in the first number is the image number (the location of this image on the volume). In the example, the image would be the 88th image on volume 1.

The second character is the AIX revision number::

1 = AIX 3.1
2 = AIX 3.1.1
3 = AIX 3.2

The third character indicates the platform (R represents the hardware platform—RISC System/6000).

The last character, G, is the type. When you view the Table of Contents on a tape using System Management Interface Tool (SMIT), the information under the I/U column is coded as follows:

G	GOLD or automatic fix
I	Install image
S	Subsystem
M	Maintenance

—Sue Lowe

What level of AIX is required to support X11R6?

IBM does not ship X11R6. Other companies have ported X11R6 onto AIX 3.2.5.

What can be expected with X11R6 to handle large protocol requests to a server?

The standard X protocol only allows requests up to 2^{18} bytes long. A new protocol extension, `BIG-REQUESTS`, has been added that enables a client to extend the length field in protocol requests to be a 32-bit value. This is useful for PEX and other extensions that transmit complex information to the server.

—Sai Ramanath

Fred Arnold

How can I save space on the hard drive?

There are three steps to do this:

1. Commit all software.
2. Run the `ptfdir_clean` script in `/usr/sbin` and note the parameters in the script. Do not do this step if you have a server with diskless clients.
3. Remove all files in `/usr/lpp/<LPPNAME>/inst_<PTFNUMBER>`.

—Fred Arnold

How much space needs to be available in `/tmp` when doing an install?

You need at least 8 MB available.

—Fred Arnold

How can I make Xstation X server recycle back to a login window instead of getting hung when exiting from the X server?

At this time there are only two workarounds to resolve this situation: use XDM or kill the `aixterms` before exiting the X server. It is not necessary to kill other clients.

—Sai P. Ramanath

We are having problems with ftp. When one particular user ftps to the system, she gets an error message: 530 (username) access denied.

You are using `tcsh`. The problem can occur because the user's login shell is not in the `/etc/security/login.cfg` file. Use the shell called `/usr/local/bin/tcsh`. Add this shell to the shell's stanza in `/etc/security/login.cfg`.

—Priyamvada

Dave Franks, IBM Corporation, RISC System/6000 Division, 11400 Burnet Road, Austin, TX 78758. Internet: franks@austin.ibm.com. Mr. Franks is a member of the PowerTeam group providing customer assistance in writing and debugging device drivers. He has a BS in Computer Science and Engineering from the University of Toledo.