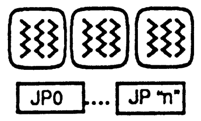
Data General

DG/UX Technical Brief

September 30, 1993

Information About AViiON® Systems from Data General's UNIX® Development Group



In This Issue:

Support for Threads in the DG/UX™ 5.4 R3.00 Operating System

Contents

Data General's UNIX Development Group has released an updated version of the DG/UX 5.4 operating system. One of the new features of this release (DG/UX 5.4 R3.00) is support for POSIX P1003.4a Draft 6 threads.

POSIX threads enable you to create applications that make more efficient use of Symmetric Multiprocessor Systems (SMPs). In applications that have many parallel tasks, using threads instead of processes can provide significant performance advantages.

Compared to processes, threads give application programmers a more precise and efficient way of managing different tasks that make up applications. The thread design fits more closely to the design of SMP hardware. For applications that have parallel tasks, a task-per-thread implementation requires significantly less task-switching time than a task-per-process implementation.

The implementation of threads is analogous to hardware "right sizing." By selecting the right tool for a specific requirement, customers can increase productivity and reduce costs. In general, customers are moving from small numbers of large computers to server/workstation-based systems.

In effect, threads are "right sized" processes. Compared to processes, threads map more closely to the underlying SMP hardware and can be more efficient users of the hardware.

Review Draft
Not for Release

September 30, 1993 11:59 am

Produced on a Data General AViiON AV4000 with FrameMaker® 3.1X.

012-00XXXX-00

What's in This Technical Brief?

This technical brief focuses on the threads extensions that DG/UX 5.4 R3.00 provides. We start by talking about Data General's commitment to standards. Here's a summary of the sections that follow:

- why and when you might want to use threads
- a comparison of the threads and process models
- considerations for designing applications to use threads
- an overview of Data General's thread implementation
- a table of the thread calls that are provided by DG/UX 5.4 R3.00

The Data General Commitment to Standards

As this technical brief was being written, there were no open standards for thread programming. However, POSIX is drafting the P1003.4a open standard for threads, and the DG/UX 5.4 R3.00 thread implementation is based on POSIX P1003.4a Draft 6 (P1003.4a/D6). A subsequent version of the DG/UX 5.4 operating system will adhere to the P1003.4a standard.

Because P1003.4a is still in the draft stages, you should be aware that you may have to make some changes to your code as the standard evolves. However, the DG/UX 5.4 R3.00 thread implementation allows you to get a head start on developing thread code with the P1003.4a/D6 calls. When Draft 7 is released in an update to DG/UX 5.4 R3.00, we will provide backward source-code compatibility to Draft 6 with #define statements, plus backward object and binary compatibility to Draft 6.

FYI—Terminology

Task—A bounded unit of work within the context of an application. Most applications consist of many tasks. We use the word "task" generically in this technical brief, because a task can be mapped to either a process or a thread.

Threads and Pthreads—The shorthand name for POSIX threads is "Pthreads," and the POSIX 1003.4a threads calls use a "pthreads" prefix. Although we use the generic term of "threads" in this brief, we are always talking about the POSIX implementation of threads.

Why Use Threads?

Why use threads? The short answer is to gain the benefits of:

- *performance*
- *programming convenience*
- *portability*

The thread model is a logical evolution of the work that's been done to use SMP-based computers efficiently. Data General's implementation of POSIX threads can help you optimize the performance of your SMP computer system while maintaining portability.

As an extension to the process model, threads give you more precise and convenient ways of managing an application's tasks. Because threads require significantly less overhead than processes, a system can:

- create and exit from 700 threads in the time that it takes to create and exit from a single process.
- switch among threads ten times faster than it can switch among processes.

The thread model, therefore, typically scales much better than the process model as the processing load on a system increases (as more tasks are added). Figure 1 shows the difference in system performance as the number of tasks increases in the process and thread models.

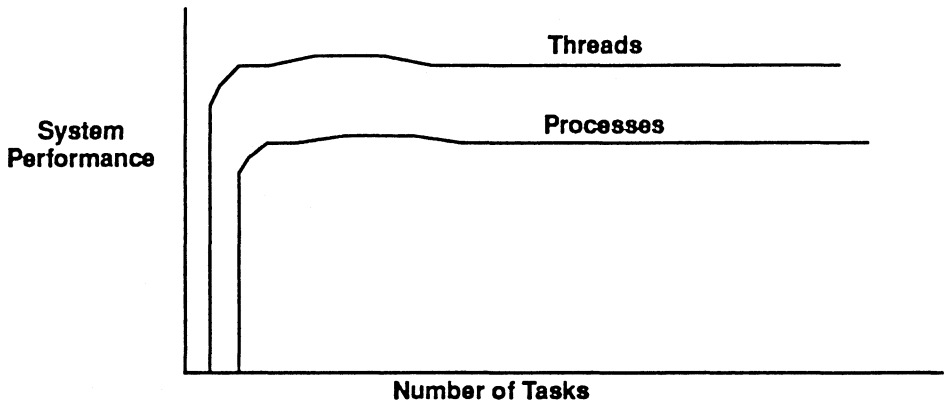


Figure 1 Performance Improvement With the Thread Model

The advantage of using threads in your applications becomes more apparent as you move your applications to AViiON computers that have more Job Processors (JPs).

In previous versions of the DG/UX operating system, a process represented a single flow of control through a program. Scheduling was performed at the granularity of processes; processes were scheduled (dispatched) onto an SMP computer's JPs. The single flow of control through a traditional process is a thread (the lefthand part of Figure 2).

The traditional single-threaded process can use the resources of only one JP at a time. And, the traditional process carries with it a significant amount of overhead just to support its single thread. Because of this overhead, it can take a relatively long time to create and switch among processes. Developers recognized that a process that supported multiple flows of control—multiple threads—could share much of a process's resources among a process's different threads.

The DG/UX 5.4 R3.00 operating system supports multi-threaded processes. As shown in the righthand part of Figure 2, scheduling is performed at the granularity of threads, so that threads from the same process can run concurrently on a computer's JPs. Because of their finer granularity and lower overhead, threads do a better job taking advantage of SMP systems than do processes.

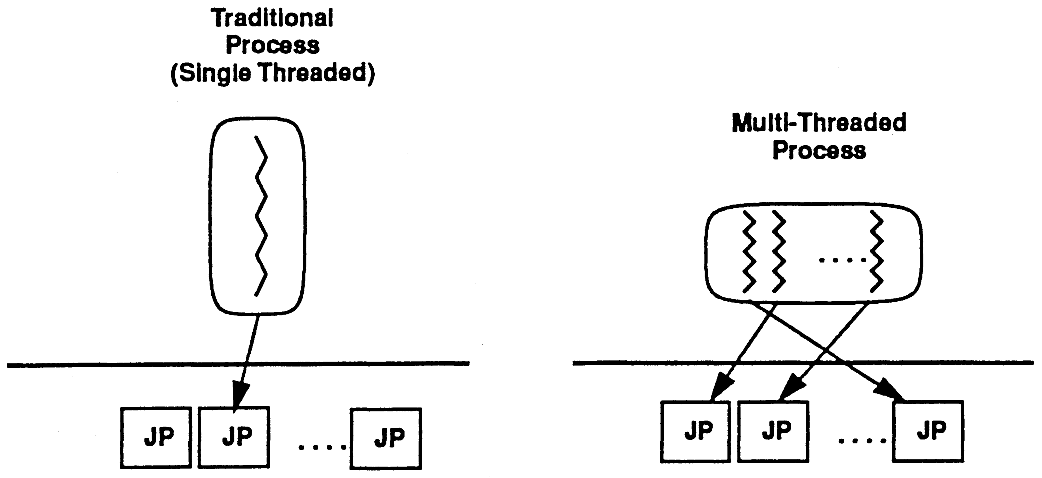


Figure 2 Traditional and Multi-Threaded Processes

Figure 3 is a simple (but practical) example of how a thread implementation can take advantage of multiple SMPs. Suppose that you have a small application that has three separate but cooperating tasks: task A, which reads data from a number of files; task B, which analyzes the data; and task C, which writes the results to different files. If you run the application in the context of a single-threaded process, only one of the tasks can run on a JP at any one time—you can't take advantage of the tasks' parallism. The alternative is to run each task within its own process. This requires three processes, with their relatively high overhead.

The ability to use a multi-threaded process for this example offers the advantage of being able to run the three tasks within a single process. The tasks can run concurrently on different JPs, without the overhead of creating two additional processes.

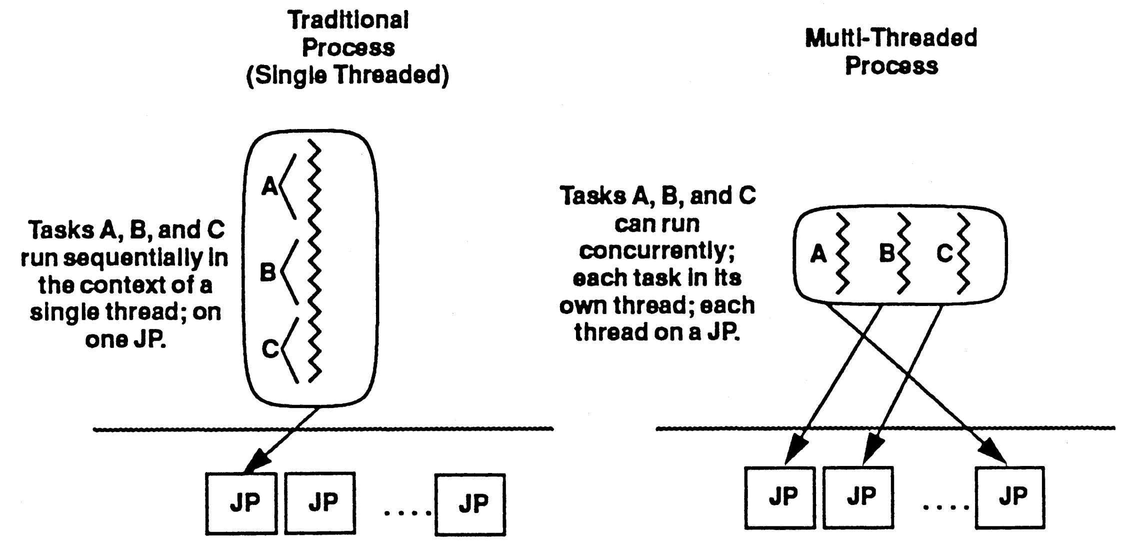

Traditional Process (Single Threaded)
Multi-Threaded Process
Tasks A, B, and C run sequentially in the context of a single thread; on one JP.
A
B
C
Tasks A, B, and C can run concurrently; each task in its own thread; each thread on a JP.
JP
JP
....
JP

Figure 3 Executing Parallel Tasks

What's common to applications that take advantage of threads is their ability to have many tasks that can run concurrently. Applications that can take advantage of threads include:

- database servers
- applications that take advantage of code produced by parallelizing compilers
- commercial distributed and realtime computing
- X servers

Database applications, for example, present many opportunities to take advantage of the thread model's ability to support *fanning out*. Fan out is the ability to split a large task into multiple parallel tasks to take advantage of multiple JPs.

In the database arena, an example of fanning out is to support the SQL "order-by" clause. In a large database, the order-by clause can result in sorting operations on hundreds or thousands of rows in a table. Rather than perform the sorting operation in the context of one process, a database designer can split the sorting operation into parallel tasks by fanning out the operation among multiple threads. When the parallel operations are complete, the results can be fanned back in (Figure 4).

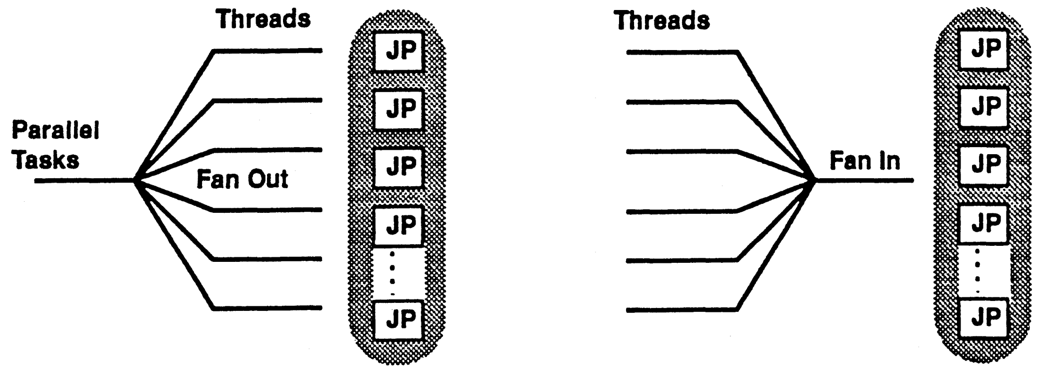


Figure 4 Fan-Out and Fan-In

Another example where threads can provide performance benefits is in an X server application. The X protocol requires that an X server run client requests to completion. Concurrent requests are handled by a round-robin scheduler. This does not cause a problem for most requests, which are typically very short. However, if a client request requires a lot of CPU time, the response to interactive requests can suffer while the X server executes the first request. An example is a request that renders a complex image.

One (relatively expensive) solution to this problem is to use hardware-based accelerators to render images. Another solution is to use a multi-threaded X server. Assigning each client request to its own thread enables the kernel to perform preemptive scheduling of the threads so that no one request can take control of the CPU. The multi-threaded X server also automatically takes advantage of SMPs so that client requests can run concurrently on different JPs.

Comparing the Process and Thread Models

A good way to learn more about threads is to compare the thread model to the traditional process model. At a high level, a thread is a stripped-down process. A thread runs in the context of a process, and all of the threads in a process share the process's address space and most of the process's state information. Note that the thread model does not eliminate the need for processes. Instead, processes act as "wrappers" for threads (Figure 5 on page 7).

Only the information that concerns flow-of-control is unique to a thread. For example, a thread has its own ID number, its own scheduling priority and class, and its own stack, which the thread uses to store local variables. And, threads work independently with separate error numbers and signal masks.

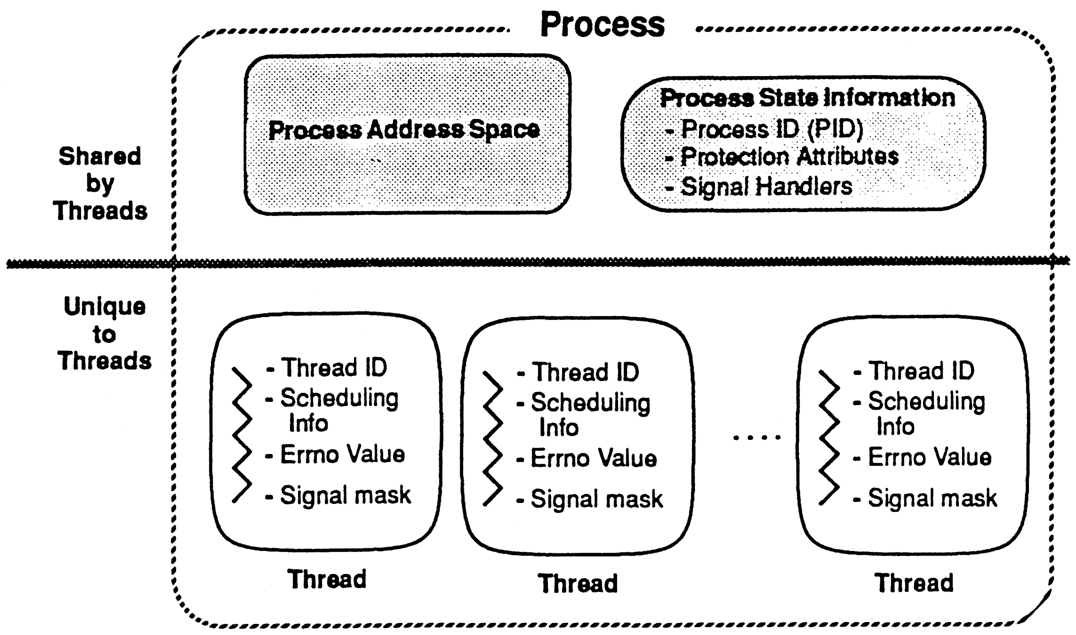


Figure 5 Threads in the Context of a Process

Unlike processes, which are organized hierarchically, threads are equal siblings that share the resources of a process. This sharing reduces significantly the thread's overhead and simplifies inter-thread communication.

In the *process* scheduling model, an application's tasks are mapped to single-threaded processes. The resulting processes, along with the other processes on a system, compete for the services of a (usually) smaller number of JPs. (The FYI section on the bottom of the next two pages summarizes how JPs are arranged in AViiON computers.)

In the *thread* scheduling model, an application's tasks are mapped to threads, which compete with all of the other threads for the system's JPs. Note that the thread model does not preclude you from creating single-threaded processes. That enables you to migrate your software to the multi-threaded environment as time permits.

Because threads have less overhead than processes, threads can be created much faster than processes. And, an operating system can switch among threads faster than it can switch among processes. Therefore, the thread model generally scales better than the process model. It's not inconceivable that you could have systems supporting thousands of threads.

Therefore, compared to the process model, the thread model ensures that tasks *scale* well among SMP systems with multiple JPs because threads:

- ❑ minimize the time that it takes to create tasks
- ❑ minimize the resource overhead required to support a task
- ❑ minimize the time that it takes to switch tasks on and off JPs
- ❑ enable related tasks to automatically *share* an address space

FYI—Caches in AViiON Configurations

Currently, an AViiON SMP computer can have two, four, eight, or sixteen JPs. In these computers, each JP has its own instruction and data caches. These per-JP caches, called primary caches, are at the first level of the memory hierarchy.

Figure 6 shows a quad-processor AViiON SMP configuration with just primary caches. The JPs and main memory connect via the main system bus, and peripheral devices connect via the I/O bus.

In addition to main (physical) memory and disk storage, the AViiON memory hierarchy can take advantage of more levels of cache memory. 8- and 16-JP AViiON computers have second-level caches as well as primary caches. The 8-JP AV6280 computer is an example (Figure 7 at the bottom of page 9). In the AV6280, each of the 4 second-level caches service a group of 2 JPs.

The parts of the memory hierarchy are defined by speed and size.

As a JP executes, its caches accumulate more and more of the task's instructions and data. Once the caches are full, the JP can run at maximum efficiency—the JP is getting most of its instructions and data from the caches, and doesn't have to wait for relatively slow memory references.

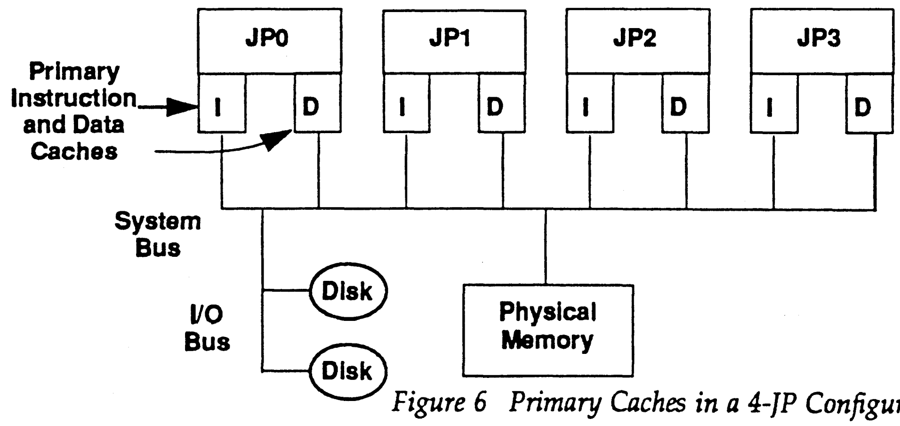


Figure 6 Primary Caches in a 4-JP Configuration

Creating Tasks

The time that it takes to create (fork) a process then exit from it is quite high; on the order of 12,000 usec. In contrast, the DG/UX 5.4 R3.00 thread scheduler can create and exit from a thread in 16 usec. This means that a thread-based scheduler can create and exit from more than 700 threads in the time that it takes to create and exit from one process.

FYI—AViiON Configurations (continued)

When a task voluntarily suspends itself or is interrupted, the operating system's task scheduler performs a context switch operation. A context switching operation carries two kinds of overhead:

- the time that it takes to save the task's state, take the task off the JP, and load a new task onto the JP
- the time that the JPs need to load their caches with the new task's instructions and data.

As a JP executes a new task, the new task's data and instructions write over the previous task's instructions and data that had been stored in the JP's caches. If the previous task starts running again on the same JP, it's likely that the JP will find none of the task's instructions or data in its cache. Until the caches are reloaded, the JP cannot run a task at peak performance.

The worst case situation is when, for whatever reason, a task is taken off a JP as soon as the JP's caches are loaded with the task's instructions and data. This means that the task is effectively running on a JP that has no caches. This can present a significant performance penalty, in addition to the cumulative overhead of task switching times.

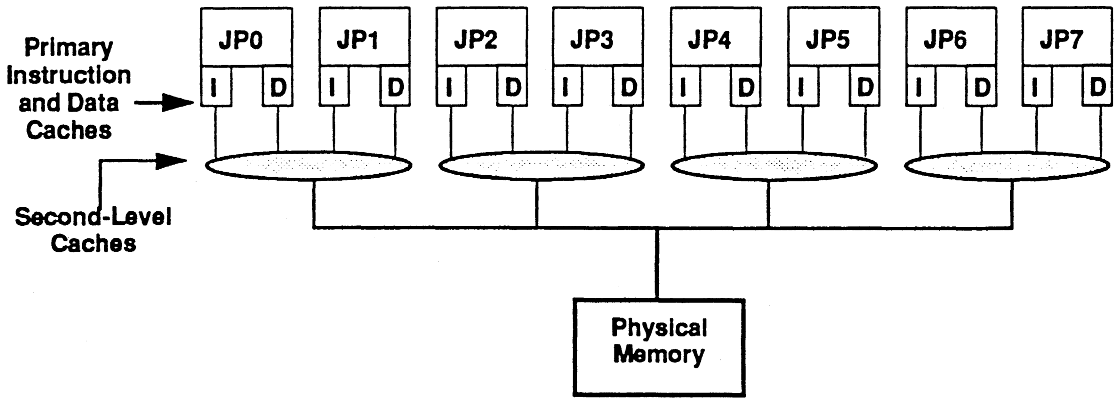


Figure 7 Second-Level Caches in an 8-JP AViiON Computer

Figure 8 compares the time that it takes to create and exit from a process and a thread. Because of the large difference in the times, the create/exit time for a thread is shown with its own time scale.

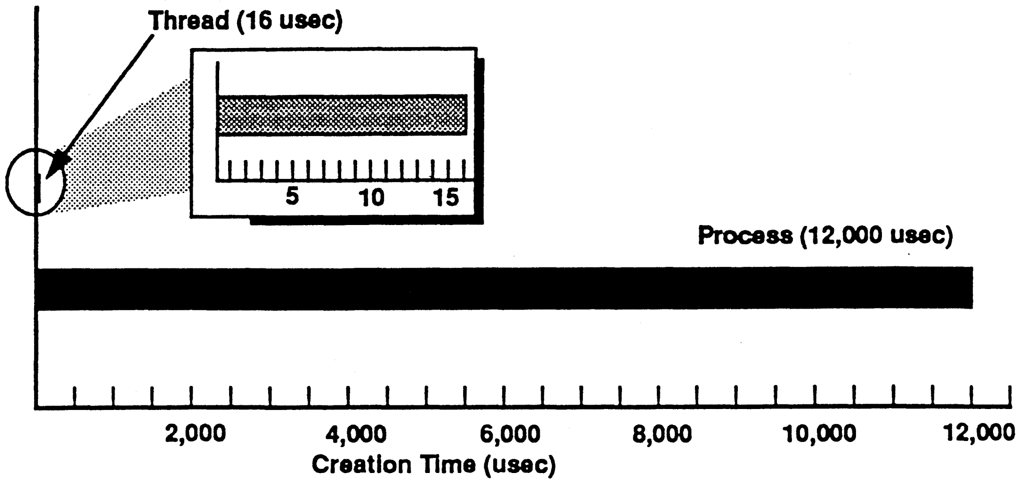


Figure 8 Comparing Create/Exit Times of Threads and Processes

To see how these timing differences can accumulate in a real-world scenario, assume that you have an application with three tasks, which you want to run separately so that the tasks have the opportunity to run in parallel. Using a task-per-process model, you create a "parent" process and then create two other processes. With the task-per-thread model, you create the parent process, which has in it one thread, and then create two more threads within that process.

Table 1 shows the difference in the startup times for these two approaches.

Table 1 Example Create/Exit Times

Using Processes	**Time (usec)**	**Using Threads**	**Time (usec)**
Create initial process (task #1)	12,000	Create initial process (task #1, thread #1)	12,000
Create child process (task #2)	12,000	Create thread (task #2)	16
Create child process (task #3)	12,000	Create thread (task #3)	16
Total Time	36,000		12,032

There's a 3X difference in times for this example. This difference can pay big dividends in long-running applications that go through the sequence many times. An example is a transaction processing application that executes this sequence many times a minute.

Task Overhead

We said earlier that a thread runs in the context of a process, and all of the threads in a process share the process's address space and most of the process's state information. Therefore, two tasks that are mapped to threads require significantly less overhead than the same two tasks that are mapped to two processes.

Because threads have low overhead, you can typically map tasks to threads without worrying as much about the cumulative effects of system overhead. For example, each client in a client/server application can be mapped to a thread. In the process model, a client-per-process mapping can "run out of gas" quickly as more clients are added, and the system starts to use more and more resources to handle the overhead of the extra processes.

Switching Among Tasks

In a multiprogrammed operating system like DG/UX, a task is often switched off a JP before the task finishes its job. A task is switched off a JP when:

- ❑ the task gives up the JP due to priority pre-emption
- ❑ the task's on-JP timeslice runs out
- ❑ the task is suspended, such as when it has to wait for an I/O operation
- ❑ the task must wait for a resource that's not available

The time that it takes to switch among tasks becomes critical to system performance. Figure 9 compares average thread-to-thread and process-to-process switching times. Compared to processes switching times, thread switching times are 10x faster.

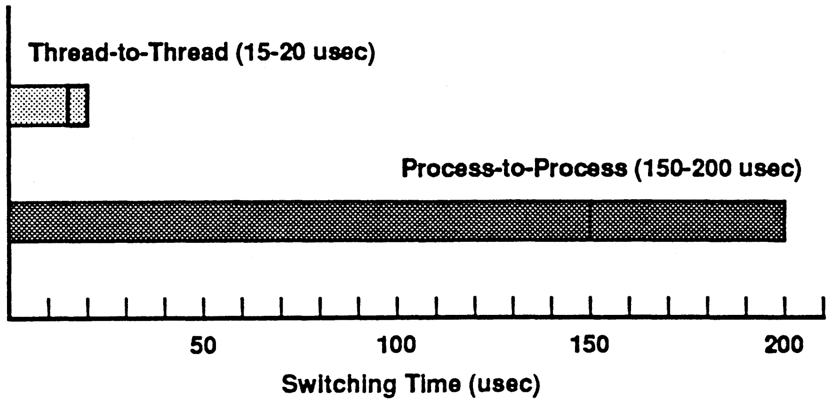


Figure 9 Comparing Switching Times of Threads and Processes

Because process context switches are relatively expensive, the process model is not well suited for handling applications that have many short-duration tasks. Overall system performance starts to degrade at the point that the JPs are spending a disproportionate amount of time handling context switching operations.

Also notice that thread-to-thread switching times (15-20 usec) are comparable to or longer than the time that it takes to create and exit from a thread (16 usec). This is another advantage of the thread model, which gives you the option of creating, using, and exiting from threads "on demand." We talk more about this technique in the section *Look for Opportunities to Reuse Threads* on page 19.

Figure 10 summarizes the performance advantage that can be gained when running the same short-duration tasks with two threads instead of two processes. The figure shows how a pair of processes and a pair of threads might be scheduled onto a single JP.

For comparison, we've assumed that each system call results in a context switch, and that the run time between system calls is small. Over a span of just three switching operations, the thread implementation completes in 40% less time than the process implementation.

To be honest, this scenario is biased toward the use of the thread model, and the performance advantage becomes less significant as the run times between system calls becomes longer. However, the scenario points out the kind of tasks that can take advantage of threads.

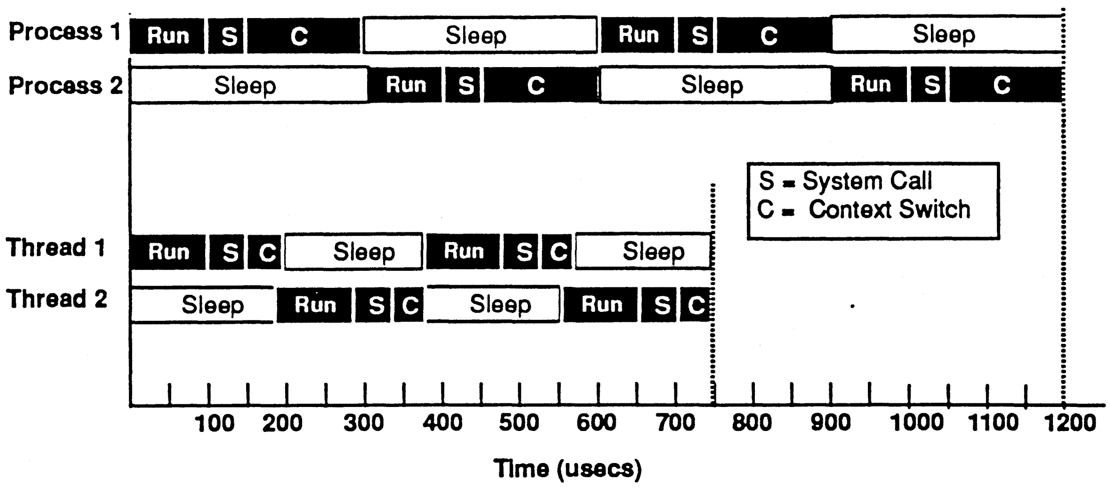


Figure 10 Comparing Process and Thread Switching Times

Sharing Address Spaces Among Tasks

In the process model, a task (process) has its own address space. In many cases, this address-space partitioning is exactly what you want because it requires that a task take explicit action to share an address space.

In the thread model, a task's ability to share a process's address space with other tasks in the process is implicit—there's no set up involved and there's no need to use the **mmap** system call.

The difference between the two models is more distinct when you look at the facilities they provide to synchronize access to shared resources. The process model's synchronization facilities are based on semaphores, which go through the kernel and require a context switching operation. You can decrease the time that it takes to access a shared resource by using a proprietary combination of sequenced locks[1] and semaphores. When there is no contention for a lock, a sequenced lock uses an XMEM instruction to set the lock. However, if a resource is locked, you must still manipulate a semaphore.

In addition to semaphores, the thread model provides a pair of simple, fast, and portable synchronization primitives: Mutual Exclusion locks (mutexes) and condition variables. In the thread model, semaphores are used to synchronize processes; mutexes and condition variables are used to synchronize threads.

Mutexes

A mutex is a low-overhead binary lock that provides mutually-exclusive access to a shared resource, such as a segment of shared memory. Like sequenced locks, uncontended references to a mutex-protected resource use an XMEM instruction. However, contended references to a mutex-protected resource are 10 times faster than a reference to a resource that is protected by a semaphore.

Figure 11 shows an example of how a mutex works. In the figure, thread T3 "owns" the mutex that protects the shared data segment, so it is the only thread that can read and write the segment. Other threads that try to obtain the mutex while thread T3 holds it are put to sleep on the data segment's mutex sleep queue. The sleep queue is ordered by priority. When thread T3 is done accessing the shared data segment, it wakes up the highest priority sleeping thread; in this case, thread T2.

1. Sequenced locks are described in the *Taking Advantage of Symmetric Multiprocessor Systems* technical brief (012-004301).

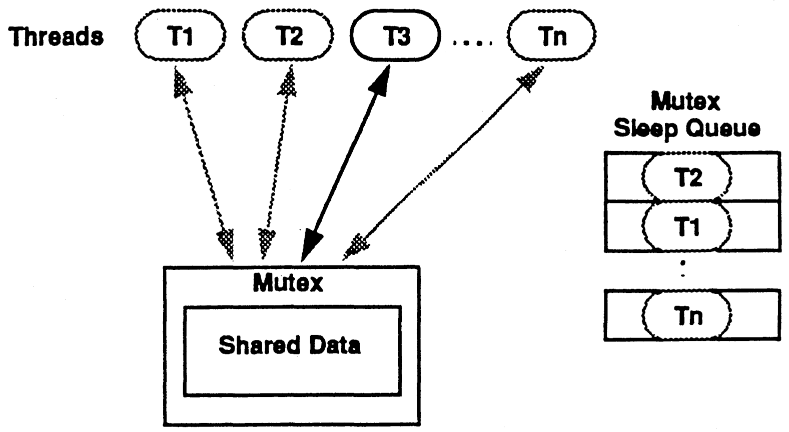


Figure 11 Resource Protected by a Mutex

Condition Variables

Condition variables enable a thread to wait for a particular event that will be signaled by another thread. They perform a similar function as semaphores in the process model, but they are typically 10x faster than semaphores. Used in conjunction with a mutex, a condition variable provides an efficient sleep queue that you can use to handle the case where a thread has to sleep while waiting for a resource to become available, for data to arrive, or for an operation to complete. By using a condition variable, you avoid tying up a resource for long periods of time.

Thread-based condition variables are typically 10x faster than semaphores.

Figure 12 is an example of how a mutex and a condition variable might work together. In the figure, thread T3 "owns" the mutex that protects the shared data segment, so it is the only thread that can read and write the segment. Other threads that tried to obtain the mutex while thread T3 held it were put to sleep on the data segment's mutex sleep queue. The sleep queue is ordered by priority. When thread T3 is done accessing the shared data segment, it wakes up the highest priority sleeping thread; in this case, thread T2.

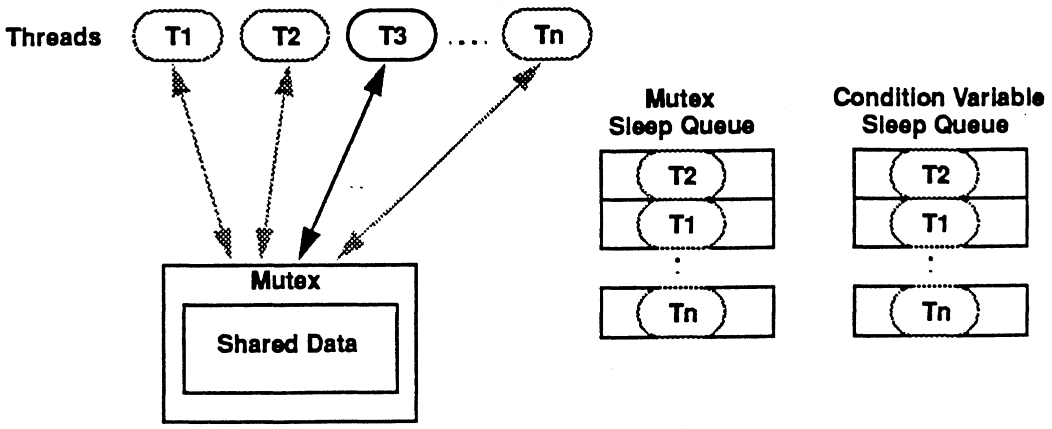


Figure 12 Mutex and a Condition Variable

Comparing Processes and Threads—A Summary

Table 2 summarizes the differences between the process and thread models.

Table 2 Comparing the Process and Thread Models

	Processes	Threads
Scaling		
❑ Task creation time	12,000 usec.	Nearly 1000x faster: 16 usec.
❑ Memory overhead	For each process, on the order of 8-to-10Kbytes non-pageable memory, plus a minimum of 64Kbytes pagable memory.	After the first thread (process-thread pair) is created, on the order of 100 bytes of non-pageable memory plus 4Kbytes of pageable memory.
❑ Task switching time	150-200 usec. process-to-process.	10x faster: 15-20 usec. thread-to-thread.
Shared address space	No—each process has its own address space.	Threads in a process share the process's address space.
Inter-Process Communication and synchronization	Sharing memory requires system-call setup. Synchronization with semaphores and sequenced locks.	Shared memory needs no setup. Synchronization with low-overhead mutexes and condition variables.

Design Considerations for Using Threads

Should you implement some, all, or none of an application's tasks with the process model or with the thread model? If you choose to use threads, what design issues should you be aware of?

Here are two very general guidelines for choosing between the two models.

- ❑ The process model is an appropriate choice for applications that require long-term serial computations or that require the security of separate address spaces. The process model is also a good choice if programming convenience outweighs performance.
- ❑ Applications that fit the thread model best are those that have many short-lived tasks that can run concurrently or have tasks that can share (or need to share) an address space.The thread model is probably the right choice if performance outweighs programming convenience.

Remember that the process model is a subset of the thread model. If you do nothing to an existing non-threaded application, it will run according to the process model.

Guidelines for Thread Design

When you choose the thread model, follow these design guidelines:

- ❑ Look for parallelism in your applications
- ❑ Establish good data partitioning and locking hierarchies
- ❑ Look for opportunities to reuse threads

Look for Parallelism

Because you may be used to thinking in terms of single-JP systems, you may not see the parallelism that's inherent in most applications. However, it's often easy to find where you can establish parallel tasks, particularly in similar computations and in I/O operations.

In general, you should look for places in applications that you can use sequences of fan out and fan in operations. The time that it takes to run a task can often be reduced significantly if you can identify opportunities to string together a series of fan out and fan in operations (Figure 13).

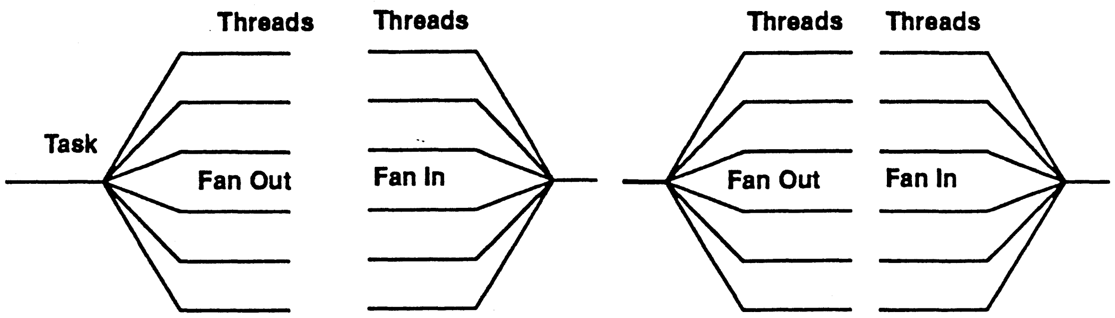


Figure 13 Series of Fan Out and Fan In Operations

On page 6 was an example of using threads to break a database sorting operation into multiple threads. This same fan-out/fan-in technique applies to other kinds of operations—computations involving arrays, for example.

Lets say that you have an array with 100 rows and 1,000 columns and you want to find the average for the values in each row. If you think in terms of a single-process you would set up a do loop to perform the computation one row at a time. If you think in terms of a multi-threaded process, you could fan-out the computation by setting up a do loop that created 100 threads—one thread to perform the computations on each row.
This approach would enable the threads to run concurrently on multiple JPs and you would have the results significantly faster than for the single threaded case.

Establish Good Data Partitioning.

You can reduce contention for data by segregating large datasets into smaller datasets and controlling access to each group with a lock. This data partitioning enables multiple JPs to work in parallel on different parts of a dataset (Figure 14).

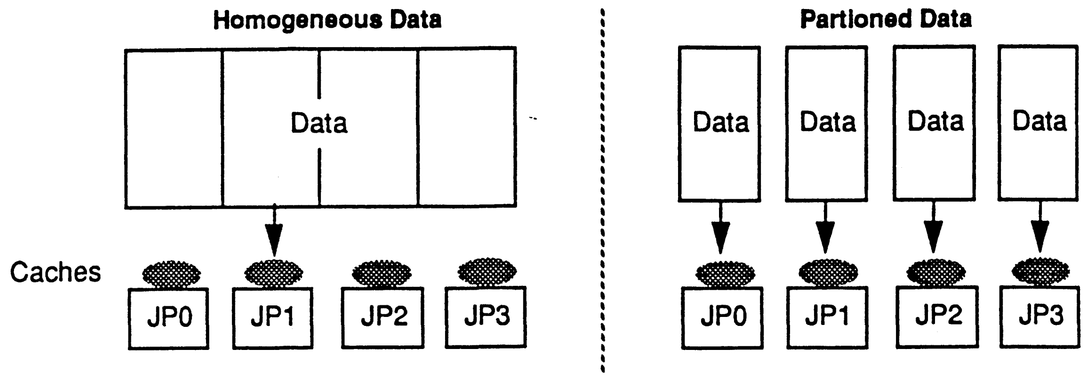


Figure 14 Data Partitioning

Data partitioning is often inherent in an application. Figure 15 shows part of a relational database, which partitions data into tables and further, into rows. By assigning mutexes to each of the data elements, you can "lock down" through the data hierarchy.

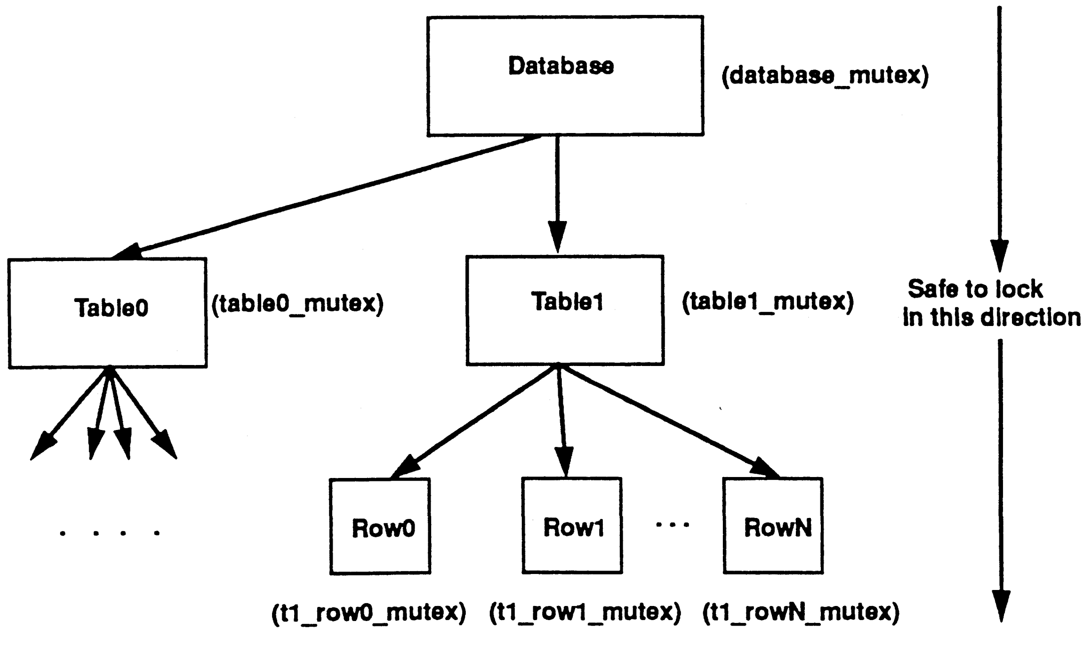

Database
(database_mutex)
Table0
(table0_mutex)
Table1
(table1_mutex)
Safe to lock in this direction
Row0
Row1
RowN
(t1_row0_mutex) (t1_row1_mutex) (t1_rowN_mutex)

Figure 15 Example Database Locking Hierarchy

Look for Opportunities to Reuse Threads

Earlier, we made the point that you can often create and exit from a thread more quickly than you can switch among threads. This enables you to avoid tying a thread to a task for too long by reusing threads.

Figure 16 shows two options for assigning threads to a task. The task performs some computation, followed by a wait period, then some more computation. You can choose to assign one thread to follow the task from beginning to completion (T1). Or, you can switch from one thread to another during the wait period (T1 to T2). Both options have their place in thread design.

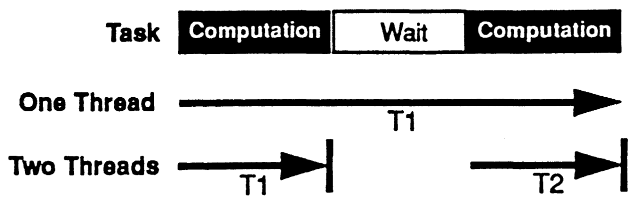


Figure 16 Assigning Threads

When choosing to tie one thread to a task, consider whether the thread is being wasted during wait time. Or, does the thread carry with it so much context (cache information) that you want to keep it attached to the task. You also consider the length of the wait period, since the information stored in the cache doesn't stay there indefinitely if it's not being used by the processor.

You may decide to switch from one thread to another. Instead of tying up a thread during a wait period, you create a new thread to finish the task. When a thread's job is done, the thread's cache information doesn't immediately disappear. Good thread design can take advantage of the already-cached data by picking up an available thread with its cache information still intact, especially if a new thread is in the same group as the old one.

Data General's Thread Implementation

Compared to some other thread implementations, Data General's implementation of the thread model is unique in two ways:

- ❑ threads are implemented mostly in kernel space—threads are created, deleted, and scheduled within the kernel
- ❑ there is a one-to-one mapping of threads to small lightweight processes (LWPs)

These two design features deliver some significant performance advantages over other thread implementations.

Kernel Space Implementation

In Data General's kernel-space thread implementation, only a small part of a thread appears in user space. Each thread's corresponding LWP is in kernel space. These highly-optimized LWPs have very low memory overhead (128 bytes).

The kernel-space implementation delivers several advantages over implementations that support threads in user space. For example:

- ❑ both local and global thread operations are fast because they can use kernel function calls (fast kernel traps)
- ❑ reliability is increased because the thread' data structures are in the kernel and cannot be corrupted by a user's application

One-to-One Threads Mapping

A one-to-one mapping of threads to LWPs eliminates the thread-to-LWP scheduling layer that is required by implementations that multiplex threads onto LWPs. The top part of Figure 17 shows the one-to-one mapping; the bottom part of the figure shows a multiplexed implementation, where there are more threads than LWPs. The one-to-one mapping of threads to LWPs results in minimal dispatch latency when a thread/LWP pair is removed from a JP to allow another thread to run.

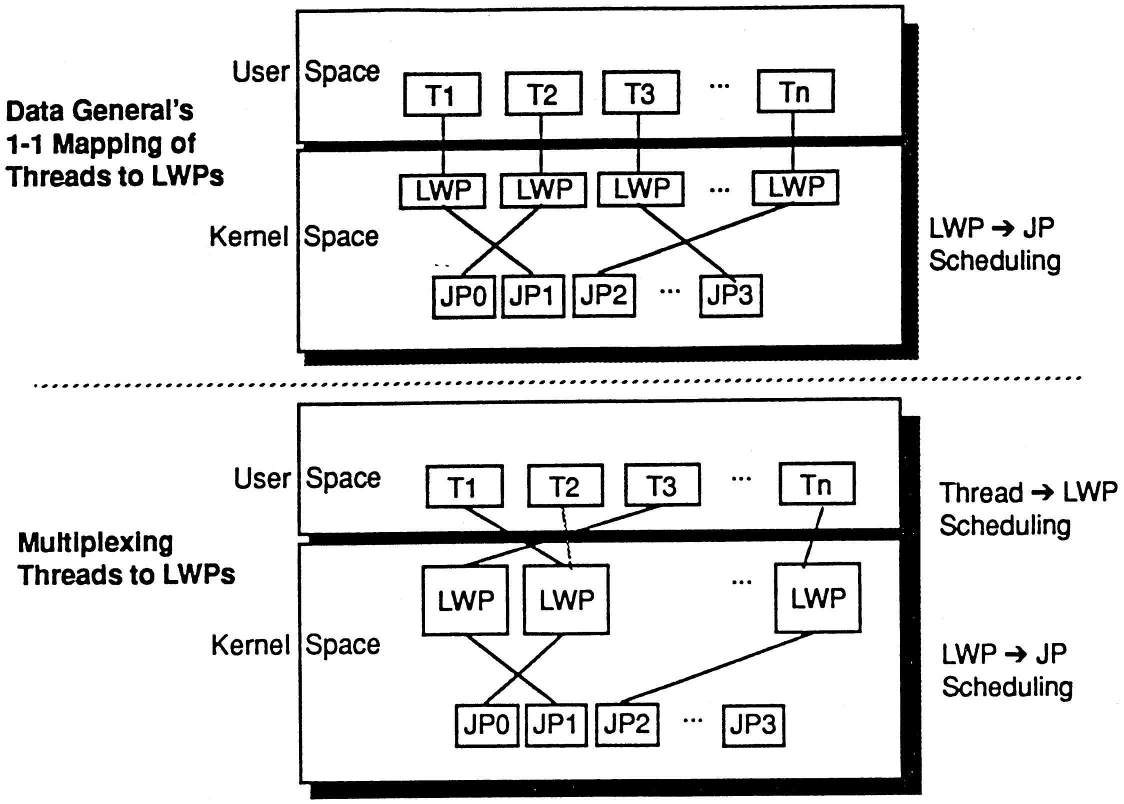


Figure 17 One-to-One and Multiplexed Thread Implementations

Advanced Features

The Data General thread implementation also supports more advanced performance features, such as the ability to group threads and the ability to associate groups of threads with groups of JPs. Although these advanced features are proprietary, their system calls are similar to the POSIX thread calls and are transparent when used on systems that don't support thread grouping and affinity.

Grouping Threads

Assigning just a single thread to a task that has groups of related computations may not always provide optimal performance. In a database server application, for example, it's likely that related computations will work with the same data files or tables in a database. Using thread groups enables you to take advantage of the cache locality of this data.

Figure 18 compares two options for grouping threads. If you take no explicit programming action, there's a N-to-one mapping of threads to groups. For example, thread group 0 in Figure 18 has a single thread, which implies that the thread does not have a close relationship with other threads.

If you have threads that are cooperatively working on the same computational problem, you can set them up in a group. Thread group 1 in Figure 18 has three cooperating threads.

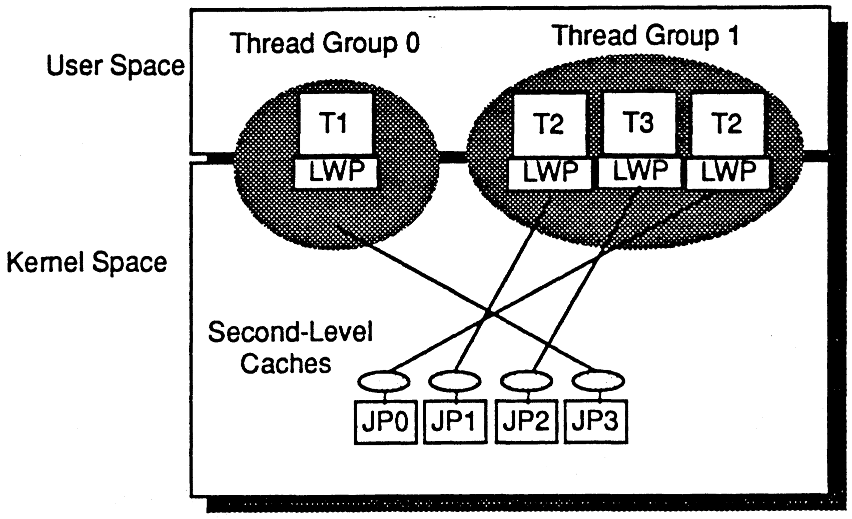


Figure 18 Thread Groups

A thread group has its own global priority and scheduling policy. And, all the threads in a thread group share the same on-JP time slice—the threads are scheduled onto multiple JPs as a group. Because the threads in a group share scheduling resources, thread-to-thread operations within a group can be as much 2X faster than thread-to-thread operations that cross group boundaries.

The other benefit is that the threads in the group can take better advantage of data that's stored in a system's caches. The cache hit ratio will be high if a group's threads are working on the same data (a database table, for example).

Creating Thread → JP Affinity Relationships

If you're using an AViiON system that shares caches among JPs, you can take the benefits of thread groups a step further and establish an affinity relationship between a thread group and a set of JPs. In an 8-JP AV6280 system, for example, the four groups of two JPs each have their own 1 Mbyte second-level caches (Figure 19). Each group of four JPs is aJP set.

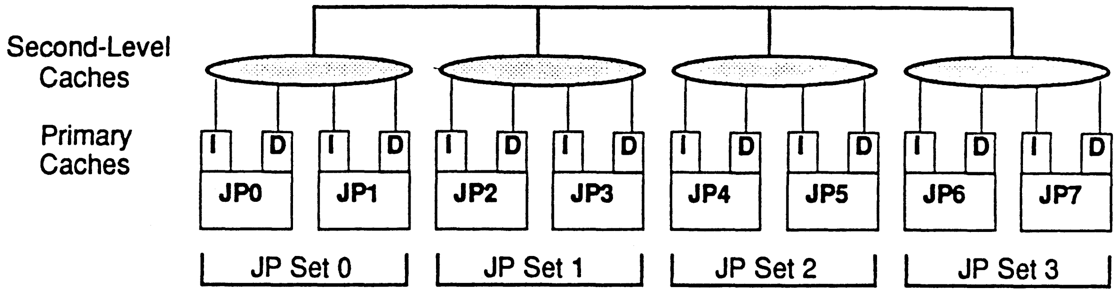


Figure 19 JP Sets

By establishing an affinity relationship between a thread group and a JP set, you can ensure that any JP in the JP set can run any thread in the thread group. This enables you to take longer term advantage of the cache locality among the threads in a thread group (Figure 20).

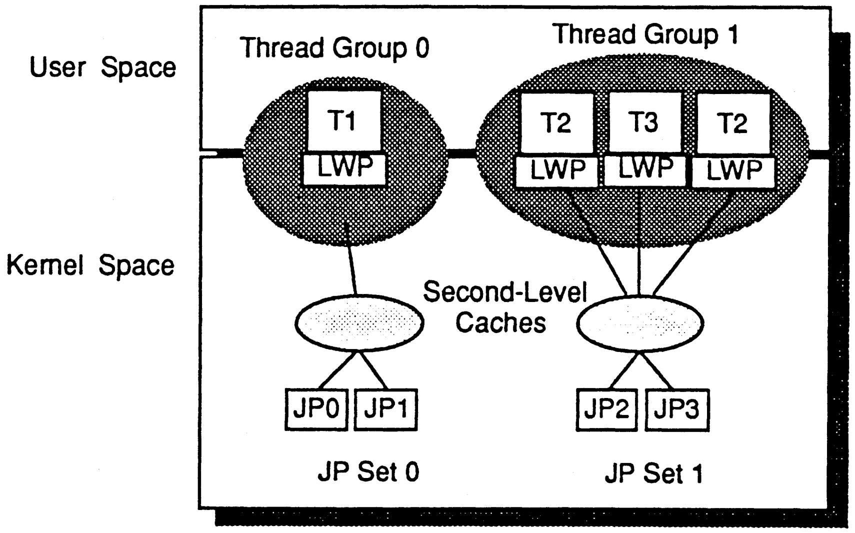


Figure 20 Threads-Set to JP-Set Affinity Relationship

FYI—Thread System Calls

The tables in this section list the thread calls that are defined in POSIX P1003.4a/D6 and provided in DG/UX 5.4 R3.00. Also included (in the last two tables) are the DG/UX 5.4 R3.00 operating system's proprietary system calls. The calls are separated into the following categories:

- Thread management—Table 3 on page 24
- Synchronization primitives—Table 4 on page 25
- Thread-specific data—Table 5 on page 26
- Thread priority scheduling—Table 6 on page 26
- Process scheduling—Table 7 on page 26
- Process control—Table 8 on page 27
- Signals—Table 9 on page 27
- Thread cancellation—Table 10 on page 28
- Thread extensions (DG/UX 5.4 R3.00 proprietary)—Table 11 on page 28
- Thread grouping (DG/UX 5.4 R3.00 proprietary)—Table 12 on page 28
- JP affinity (DG/UX 5.4 R3.00 proprietary)—Table 13 on page 29

Table 3 Thread Management

Call	Description
pthread_create()	Create a thread
pthread_join()	Wait for thread termination
pthread_exit()	Terminate a thread normally
pthread_detach()	Detach a thread
pthread_self()	Find caller's thread ID
pthread_equal()	Compare thread IDs
pthread_once()	Dynamic package initialization
pthread_attr_init()	Initialize a thread attributes object
pthread_attr_destroy()	Delete a thread attributes object
pthread_attr_setstacksize()	Set stack size thread attribute
pthread_attr_getstacksize()	Retrieve stack size thread attribute
pthread_attr_setdetachstate()	Create a new thread in the detached state
pthread_attr_getdetachstate()	Determine if a thread was created as detached

Table 4 Synchronization Primitives

Call	Description
pthread_mutex_init()	Initialize a mutex
pthread_mutex_destroy()	Destroy a mutex
pthread_mutex_lock()	Acquire a mutex lock
pthread_mutex_unlock()	Release a mutex lock
pthread_mutex_trylock()	Acquire a mutex lock conditionally
pthread_mutexattr_init()	Initialize default attributes object for a Mutex
pthread_mutexattr_destroy()	Destroy attributes object for a Mutex
pthread_cond_init()	Initialize a condition
pthread_cond_destroy()	Destroy a condition
pthread_cond_wait()	Wait for a condition
pthread_cond_timedwait()	Timed wait for a condition
pthread_cond_signal()	Signal a condition
pthread_cond_broadcast()	Broadcast a condition
pthread_mutexattr_getpshared()	Get process-shared attribute for a Mutex
pthread_mutexattr_setpshared()	Set process-shared attribute for a Mutex
pthread_condattr_init()	Initialize default attributes object for a Condition
pthread_condattr_destroy()	Destroy attributes object for a Condition
pthread_condattr_getpshared()	Get process-shared attribute for a Condition
pthread_condattr_setpshared()	Set process-shared attribute for a Condition

Table 5 Thread-Specific Data

Call	Description
pthread_key_create()	Create a thread-specific data key
pthread_setspecific()	Associate a value with a thread-specific key
pthread_getspecific()	Retrieve the value associated with a thread-specific key

Table 6 Thread Priority Scheduling

Call	Description
pthread_attr_setprio()	Set scheduling priority thread attribute
pthread_attr_getprio()	Retrieve scheduling priority thread attribute
pthread_attr_setscope()	Set contention scope attribute
pthread_attr_getscope()	Get contention scope attribute
pthread_attr_setinheritsched()	Set scheduling inheritance thread attribute
pthread_attr_getinheritsched()	Retrieve scheduling inheritance thread attribute
pthread_attr_setsched()	Set scheduling policy thread attribute
pthread_attr_getsched()	Retrieve scheduling policy thread attribute
pthread_setschedattr()	Set thread scheduling attributes
pthread_getschedattr()	Retrieve thread scheduling attributes
pthread_yield()	Yield to other threads

Table 7 Process Scheduling

Call	Description
setprio()	Set scheduling priority
getprio()	Get scheduling priority
setscheduler()	Set scheduling policy
getscheduler()	Get scheduling policy
yield()	Yield to another process

Table 8 Process Control

Call	Description
fork()	Create a new process
exec	Execute a file
_exit()	Terminate a process
wait()	Wait for process termination
waitpid()	Wait for process termination

Table 9 Signals

Call	Description
sigwait()	Synchronously wait for an asynchronously generated signal
pthread_kill()	Send a signal to a thread
sigaction()	Examine and change process signal action
sigprocmask()	Examine and change thread blocked signals
sigsuspend()	Wait for a signal
sigpending()	Examine pending signals
pause()	Wait for signal delivery
longjmp()	Non-local jump
siglongjmp()	Non-local jump
setjmp()	Non-local jump handler
sigsetjmp()	Non-local jump handler
alarm()	Schedule alarm
sleep()	Delay process execution
raise()	Send a signal to a process

Table 10 Thread Cancellation

Call	Description
pthread_cancel()	Cancel execution of a thread
pthread_setintr()	Enable or disable interruptability of a thread
pthread_setintrtype()	Set interruptability type of a thread
pthread_testintr()	Establish an interruption point for a thread
pthread_cleanup_push()	Register a per-thread cleanup handler
pthread_cleanup_pop()	Unregister a per-thread cleanup handler

Table 11 DG/UX Thread Extensions

Call	Description
dg_pthread_get_lwpid()	Gets a thread's LWP identifier
dg_pthread_is_preempted()	Returns 1 if a thread has been timeslice pre-empted; returns 0 otherwise
dg_pthread_sleep()	Puts a thread to sleep for a specified amount of time

Table 12 DG/UX Thread Grouping

Call	Description
dg_pthread_groupattr_init()	Initialize a thread group's attributes
dg_pthread_groupattr_destroy()	Destroy a thread group's attributes
dg_pthread_groupattr_setinheritsched()	Set a thread group's scheduling inheritence attribute
dg_pthread_groupattr_getinheritsched()	Gets thread group's scheduling inheritence attribute
dg_pthread_groupattr_setsched()	Set a thread group's scheduling policy attribute
dg_pthread_groupattr_getsched()	Get a thread group's scheduling policy attribute

Table 12 DG/UX Thread Grouping (Continued)

Call	Description
dg_pthread_groupattr_setprio()	Set a thread group's scheduling priority attribute
dg_pthread_groupattr_getprio()	Get a thread group's scheduling priority attribute
dg_pthread_group_create()	Create an empty thread group
dg_pthread_group_destroy()	Destroy a thread group
dg_pthread_group_self()	Return a thread group's identifier
dg_pthread_group_equal()	Returns 1 if two thread groups are the same; returns 0 otherwise
dg_pthread_group_get_lwp_group_id()	Returns a thread group's underlying LWP group identifier
dg_pthread_group_setschedattr()	Sets the scheduling attributes of a thread group
dg_pthread_group_getschedattr()	Gets the scheduling attributes of a thread group
dg_pthread_group_get_times()	Returns the amount of user and system time used by a thread group
dg_pthread_attr_setgroup()	Sets a group identifier thread attribute
dg_pthread_attr_getgroup()	Gets a group identifier thread attribute

Table 13 DG/UX JP Affinity

Call	Description
dg_cpu_id_set_init()	Initializes a CPU id set
dg_cpu_id_set_destroy()	Destroys a CPUid set
dg_cpu_id_set_add_id()	Adds a CPU identifier to a CPU id set
dg_cpu_id_set_remove_id()	Removes a CPU identifier from a CPU id set
dg_cpu_id_set_is_member()	Returns 1 if an identifier is in a CPU id set; returns 0 otherwise
dg_cpu_id_set_assign_set()	Assigns the contents of a CPU id set to another CPU set
dg_cpu_id_set_add_set()	Adds the contents of a CPU id set to another CPU set

Table 13 DG/UX JP Affinity (Continued)

Call	Description
dg_cpu_id_set_remove_set()	Removes CPU identifiers from a CPU id set
dg_cpu_id_set_has_members()	Returns 1 if a CPU id set contains the specified list of identifiers; returns 0 otherwise
dg_cpu_info_init()	Initializes the dg_cpu_info structure
dg_cpu_info_destroy()	Destroys the dg_cpu_info structure
dg_cpu_info()	Returns information about a CPU/cache/memory hierarchy
dg_cpu_affinity_attr_init()	Initializes the affinity attributes object
dg_cpu_affinity_attr_destroy()	Destroys the affinity attributes object
dg_cpu_affinity_attr_set_cpu_id_set()	Sets the CPU id set on which the affined LWP groups can run
dg_cpu_affinity_attr_get_cpu_id_set()	Gets the CPU id set on which the affined LWP groups can run
dg_cpu_affinity_attr_set_minimum_level()	Sets the minimum allowed CPU hierarchy-level for the affined LWP group
dg_cpu_affinity_attr_get_minimum_level()	Gets the minimum allowed CPU hierarchy-level for one LWP group
dg_cpu_set_affinity()	Sets the affinity attributes of one or more LWP groups
dg_cpu_get_affinity()	Gets the affinity attributes of one LWP group