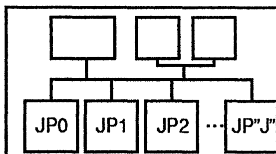
Data General

DG/UX Technical Brief

June 16, 1992

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



In This Issue:

Taking Advantage of Symmetric Multiprocessor Systems

Contents

What's in This Brief........................2
Measuring Performance 3
Reviewing SMP Concepts 4
Design Goals 8
Maximizing Process Parallelism.. 8
Minimizing Context Switching 13
Techniques for IPC and Synchronization 15
FYI—Sample xmem Routines......23
Shared Memory Test Results......28
FYI—Other IPC Techniques.........30

In Technical Brief 012-003886 (*A Second Look at Multiprocessor SMPs*), we talked about how symmetric multiprocessor AViiON® systems provide users with unique and powerful ways of increasing their data processing throughput. In that technical brief, we showed you the kinds of performance advantages that can be obtained when you use multiprocessor AViiON computer systems.

In that earlier technical brief, we said: "you don't need to change your application software to run it on an AViiON multiprocessor system. You can run the exact same applications on single-processor systems and multiprocessor systems."

That statement is true—you don't *need* to change your applications when you move them from a single-processor AViiON system to a multiprocessor system. However, an application that is designed to run on a single-processor system may not take complete advantage of an SMP system's multiprocessors.

This technical brief talks about the kinds of design and implementation techniques that enable you to take complete advantage of multiprocessor systems. Specifically, we'll talk about ways that you may be able to improve application performance by designing applications that provide opportunities for process parallelism. You can use these techniques when you design new applications or when you make changes to existing applications.

AViiON is a registered trademark of Data General Corporation.
DG/UX is a trademark of Data General Corporation.
FrameMaker is a registered trademark of Frame Technology Corporation.
UNIX is a registered trademark of UNIX System Laboratories, Inc.
©1992 Data General Corporation.

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

012-004177-00

What's in This Technical Brief

This technical brief focuses on ways to reduce applications' elapsed time as you scale up the number of Job Processors (JPs) in your system. As an application designer, you have access to a range of techniques that enable you to build applications that take full advantage of multiple-JP SMP systems, and we'll talk about those techniques.

When you are writing applications that will run on SMP systems, your basic goal is to keep all of the Job Processors in the SMP systems productively busy. We say *productively busy,* because a poorly designed application can keep JPs busy doing the "wrong" things, such as switching from one process's context to another, or looping for long periods of time while processes wait on locks.

An application that runs for 6 hours to predict what the weather will be in 2 hours is not particularly useful.

Designing applications that take advantage of an SMP system requires an understanding of two related issues:

- *interprocess communication*—how to select the most efficient ways of enabling the processes in your applications to exchange data.
- *process synchronization*—how to select the most efficient ways of controlling the interaction among the processes in your applications.

In this technical brief, we'll focus on a particular set of interprocess communication and process synchronization techniques; shared memory and semaphores. Before we begin that discussion, we'll review some terminology and some AViiON SMP concepts that form the foundation of the discussions.

A Note About Threads

Currently, the DG/UX operating system supports a single flow of control within a DG/UX process—a single thread. A future implementation of DG/UX will support multiple threads, which will enable different parts of the same process to run simultaneously on multiple JPs. Therefore, threads will provide another important technique for taking advantage of SMPs.

Measuring System Performance

There are two ways of measuring the performance of an SMP system, throughput and elapsed time. *Throughput* is a measure of how many jobs (processes) a system can execute in a given time period—perhaps in a second. As you scale up from a single-JP DG/UX system to a dual-JP SMP system, you automatically see a nearly two-to-one increase in process *throughput*—the number of processes that the machine runs in a second. For example, the throughput of a quad-JP system is nearly twice that of a dual-JP system. The vertical bars in Figure 1 show the relative increases in throughput as you scale up from a single-JP system to a quad-JP system.

Elapsed time is the time that it takes for an application to run. As you scale up from a single-JP system to a multiple-JP system, you may not see an equivalent decrease in an application's elapsed time. That's because an application designed to run on a single-JP system may not provide the process parallelism that can take full advantage of a multiple-JP system.

The lines in Figure 1 (plotted against the right-hand axis) compare the difference in elapsed time between applications that consist of parallel processes and applications that don't contain parallel processes.

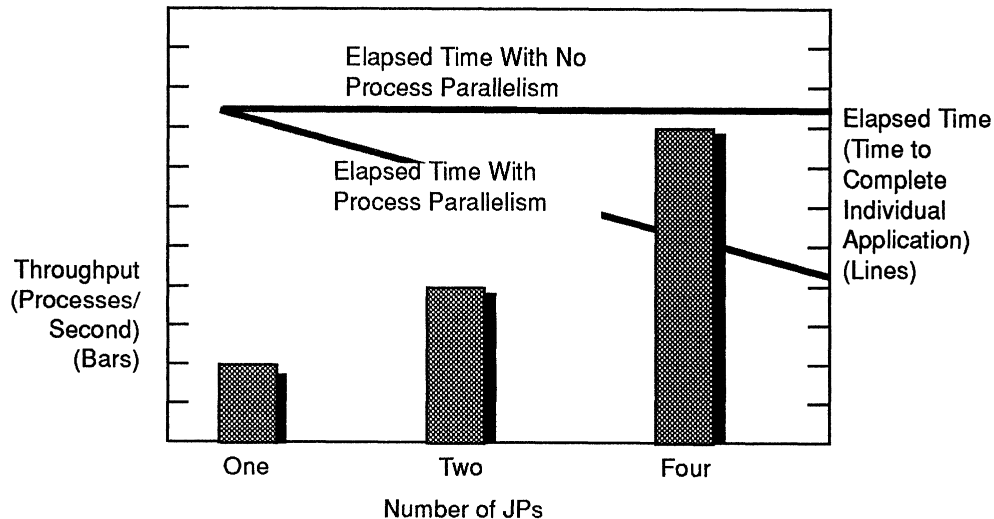


Figure 1 Process Throughput and Elapsed Time

Some AViiON SMP Process Scheduling Concepts

An AViiON multiprocessor system is a *Symmetric Multiprocessing* (SMP) system. The key word here is *symmetric,* which means that the multiple JPs (the CPUs) in an AViiON SMP system are seen by user programs and the DG/UX operating system as equivalent.

Processes, Virtual Processors, and Job Processors

Processes are programs in execution. On AViiON systems, processes run on Virtual Processors (VPs), which are software abstractions of the computer's real, physical JPs (Figure 2). Because they are software abstractions, VPs hide the implementation details of the underlying hardware from the processes.

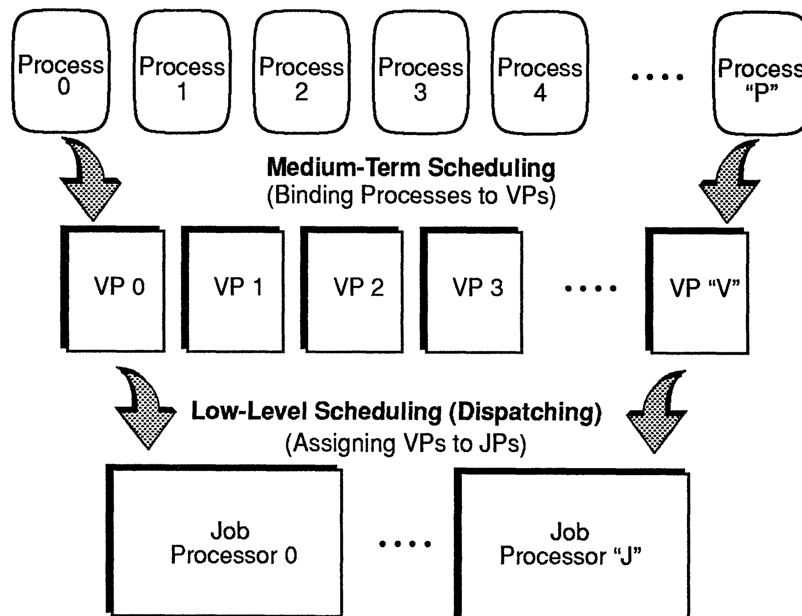


Figure 2 Processes, Virtual Processors, and Job Processors

Process States

On a DG/UX system, a process can be in one of several states. For our purposes, we're interested in the following process states:

- Running
- Runnable
- Sleeping

Running

A running process is executing on a JP. The Medium Term Scheduler has bound the process to a VP and the dispatcher has assigned the VP to a JP. The process can be executing in either user space or in kernel space.

Note: While it's technically correct to say that "processes are bound to VPs, which run on JPs," most people say that "processes run on JPs." We'll start using that simpler terminology here.

Runnable

A runnable process is waiting for a JP to become available. The process is bound to a VP and is eligible to run when a JP is available.

The dispatcher maintains a VP scheduling queue that lists all of the runnable processes. In the scheduling queue, the runnable processes are arranged by priority—the highest priority runnable process is at the head of the list.

The Medium Term Scheduler (MTS) is responsible for providing priority information for user processes. Kernel processes, such as the MTS itself, are permanently bound to VPs, and usually have higher dispatcher priorities than user processes.

The MTS provides several heuristics that support fair time-sharing scheduling policies. The MTS assigns and keeps track of processes' on-JP time slices, and can adjust time slices for interactive (I/O intensive) and compute-bound processes.

Sleeping

A sleeping process is waiting for an event to occur—the process is sleeping on the event. The distinction between a sleeping process and a runnable process is that a sleeping process is not waiting for a JP to become available.

Processes can go to sleep for many reasons. For example, a sleeping process can be waiting for an I/O operation or waiting for a resource (perhaps a data buffer) to become available.

A sleeping process that becomes runnable may not start executing immediately

In terms of JP resources, sleeping is relatively inexpensive. The kernel does a context switching operation when it takes the process off a JP. After that, a sleeping process doesn't use JP resources, it just uses a slot in a system-wide process table that the kernel maintains. This is in contrast to a process that is doing a busy/wait operation, which uses JP resources while the process runs in a loop.

When an event occurs, the kernel *wakes up all of the processes that are waiting on that particular event.* That's a key point—if more than one process is waiting on the same event, all of the waiting processes are awakened and

placed into the VP scheduling queue as runnable. Therefore, a sleeping process that becomes runnable when an event occurs may not be the next process to start running when a JP becomes available—it depends on the process's priority relative to the other runnable processes.

Figure 3 shows an example of how these different process states can interact. The top of the figure represents a global time line, showing the state of all the processes running on a dual-JP system. Running processes are shown in black. From the timeline, we've isolated part of an application that has four processes.

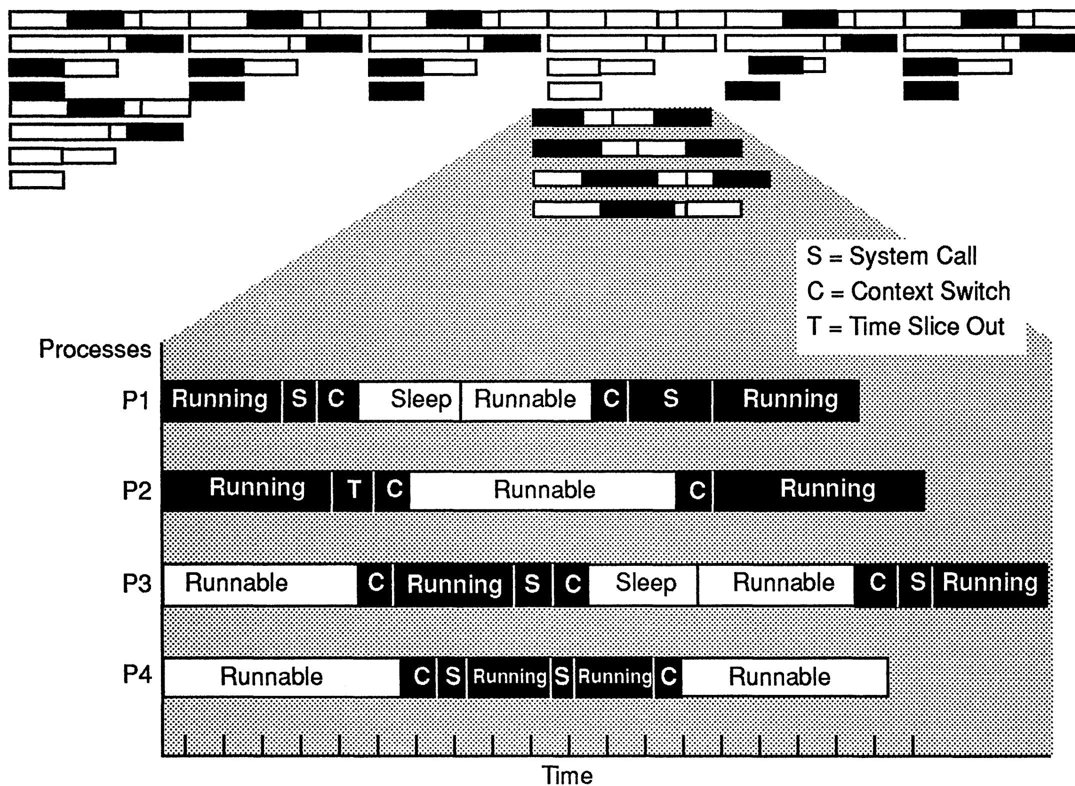


Figure 3 Sleeping, Runnable, and Running Processes on Dual Processors

The figure shows several interesting things about processes and process management. For example:

- Processes are considered to be running while they are executing in user space, making a system call, or performing a context switch.
- When a process's time slice runs out, the process becomes runnable (P2). At that time, other processes at the same priority level can execute (P4 starts running when P2's time slice runs out).
- The time to perform system calls is different from call to call (compare P1 and P3)
- A system call need not result in a context switch (P4).

System Calls, Mode Switching, and Context Switching

A system call causes a process to switch modes—from running in user space (user mode) to running in kernel space (kernel mode), then back to user space (Figure 4).

A mode switch is not the same thing as a context switch—a process can continue running on a JP while it switches modes. By contrast, in a context switch, the kernel takes a process off a JP.

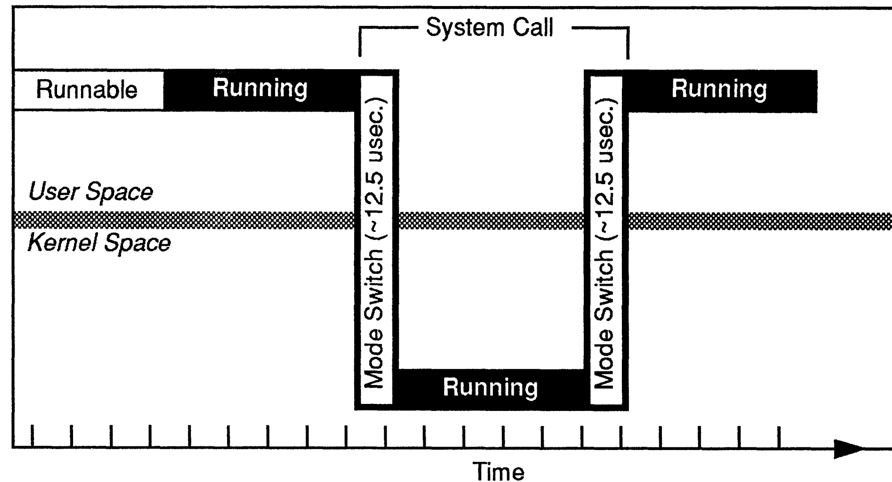


Figure 4 Mode Switching Overhead of a System Call

The kernel (driven by the kernel's Medium Term Scheduler) performs a context switch when it decides that it should take a process off a JP to run another process. A context switch requires that the kernel push a process's state information onto a stack. By popping the process's state information off the stack, the kernel can start running the process at the point it was interrupted.

Context switch operations can be relatively expensive users of JP resources. A typical context switching operation may take from 150-to-200 microseconds.[1] In the time that it takes to perform a context switch, a process could be executing as many as 1500-to-2000 instructions.

Mode switching is far less expensive than context switching. A round-trip mode switching operation requires about 25 microseconds to complete (not counting the time that it takes to execute the system call's code). Therefore, any time that you avoid a system call, a process could run as many as 250 instructions.

1. On a 25Mhz 88100-based AViiON system.

Design Goals for SMP Applications

Based on the information in the previous sections, we can make some observations about how to design applications that will take advantage of SMP systems.

On page 2, we said that the general goal for taking advantage of an SMP system, is to "... keep all of the JPs in the SMP productively busy." The keys to achieving that goal are to:

- maximize opportunities for parallelism— by partitioning applications into cooperating processes;
- minimize process context switching—by providing an optimum ratio of processes to JPs;
- minimize contention for resources—by using appropriate interprocess communication and synchronization techniques.

The next three sections talk about techniques that you can use to achieve these three goals. Bear in mind that these three objectives are very much inter-related, and an application won't perform as well as it could unless all three objectives are met. For example, it's not enough to simply partition an application into some arbitrary number of processes. For best performance, your application must strike a balance between the number of processes and JPs. If there are many more processes than JPs, an application can spend too much time doing context switch operations. If there are too few processes, all of the JPs won't be productively busy.

Maximizing Opportunities for Process Parallelism

An application that takes advantage of an SMP system is one that provides cooperating processes that an SMP system can run in parallel.

One key to providing parallel processes is to find the natural boundaries in an application, based on what kind of work needs to be done. Then, you can determine which parts of an application should have their own processes and which parts can be grouped together in a process.

The boundaries of an existing application may not provide the best way of partitioning the application.

When you partition an application into processes, you should recognize that the logical or functional boundaries of an application may not provide the most efficient way of creating processes. This is especially true if you're working with an existing application that was designed to run on a single processor system. For example, some database management systems provide one process for each client; the more clients, the more processes, with the potential for more context switches. In cases like this, you should consider partitioning the application horizontally (across similar functions) rather than vertically.

The amount that you can speed up an application is limited by the application's longest sequential process.

Another key to achieving the goal of parallelism is to design your applications so that they minimize the time that the processes in the application must run sequentially. By sequentially, we mean that the process must run to completion before a follow-on process can start. The amount that you can speed up an application is limited by the time that it takes to run the longest process in the application, no matter how many processors are available to run the application.

Figure 5 shows an example of how minimizing the sequential layout of processes pays off when you run the processes on a machine with multiple JPs.

Assume that we have an application with four cooperating processes: P1, P2, P3, and P4. (By cooperating, we mean that process P2 uses data from process P1, P3 uses data from P2, and so on.) This is typical of processes that support the different phases of a compilation, or processes that analyze the same stream of data in many different ways.

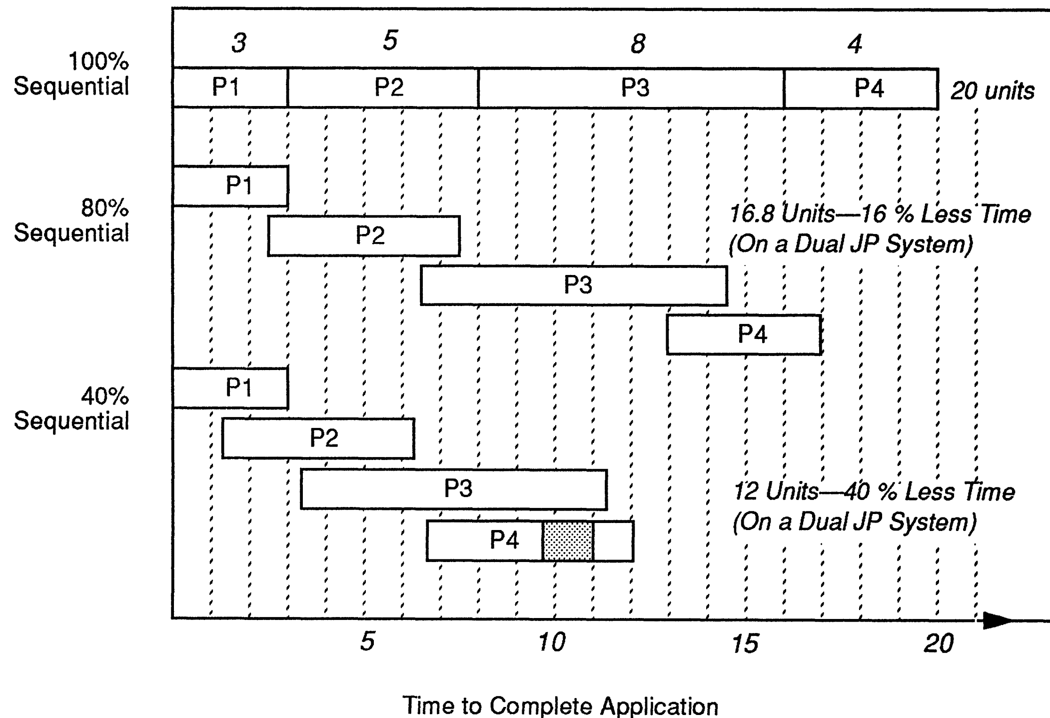


Figure 5 Sequential and Parallel Processes

In Figure 5, we've assigned arbitrary execution times to each process; 3, 5, 8, and 4 units respectively. Assume that this is pure execution time, without considering time waiting for I/O operations. Assume also that a time unit is much greater than a process's context switching time.

Therefore, the total time required to run the application is 20 time units, assuming that each process runs 100% sequentially—that each process in the application runs to completion before the next process can get its data and start running. The other cases show what happens when a cooperating process can start when 80% and 40% of its predecessor process is complete.

It's easy to see how reducing the time that processes must run sequentially increases opportunities for parallelism. For example, if each process in the application runs sequentially for 80% of its execution time before the following process can start, there's 20% of each process's execution time available for performance increases from parallelism.

While the actual on-JP execution times of the four processes is the same for all three cases, the wall-clock time (elapsed time) for downstream processes can be longer. The shading in process P4 of the 40% sequential case indicates time during which the process has to wait for data from process P3, and does not fully utilize the services of a JP. During this time, the process could be swapped off a JP to enable a higher priority process to run.

Figure 6 shows the opportunities that the example application has to take advantage of two JPs. The 80% version has three dual-JP time slots (marked with background shading). The 40% version has four opportunities to use two JPs.

FYI—Producer/Consumer Processes

It can be helpful to think of cooperating processes as producers/consumers, where a process produces something (data) that another process consumes. You can then look at whether a consumer can start work before a producer finishes its work.

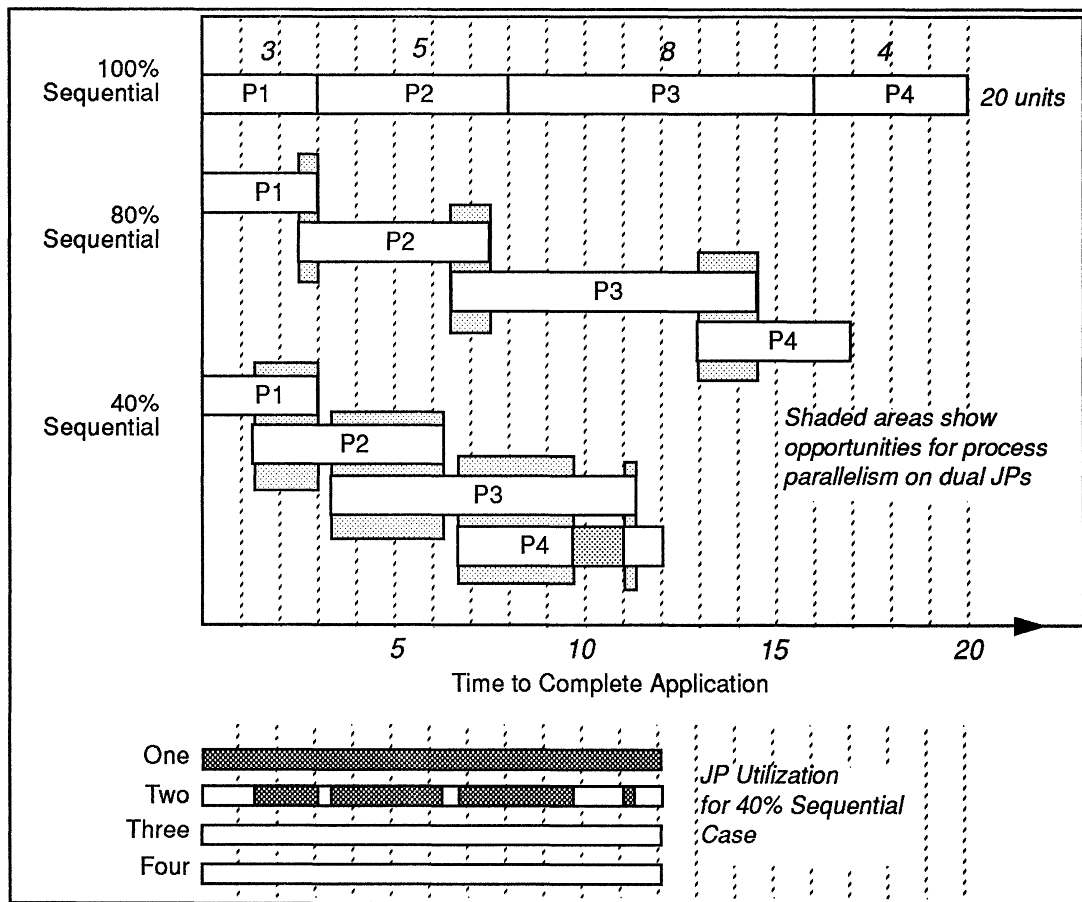


Figure 6 Opportunities to use Dual JPs

The bar chart at the bottom of the figure shows JP utilization for the 40% sequential design. The 40% sequential design takes good advantage of dual JPs. However, neither the 80% version or 40% versions provide opportunities to use more than two JPs, because only two processes are able to run in parallel (unless there are multiple instances of the application running).

If you have a situation like this (a multiple process application with poor parallelism) you may be able to achieve more opportunities for parallelism by reordering the application's processes.

Figure 8 shows what happens when we change the order of the first three processes in the example. Ordering the processes this way starts to take advantage of three and four JPs. (The bar chart at the bottom of the figure shows JP utilization for the 40% sequential design.)

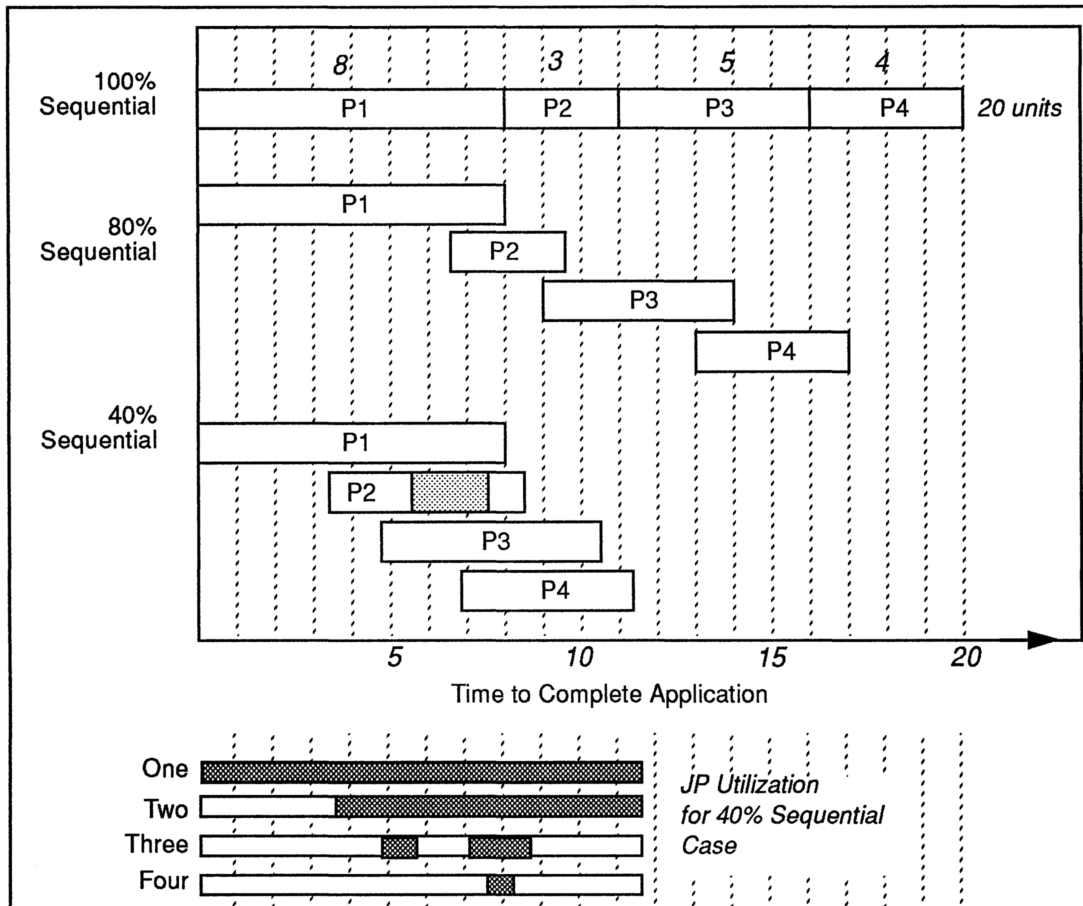


Figure 7 Effect of Reordering Processes

When you design some applications, you may find that it's possible to use one process to divide the application's data into separate pieces. You can then pass the data off to other processes, which work on the data independently. The first (main) process splits up the data, then collects and combines the results from each data-processing process.

Figure 8 shows an example of this technique. Here, process P1 starts three other processes, which then go to sleep. As soon as process P1 starts generating data that the other processes can use, process P1 wakes them up and they start working on the data.

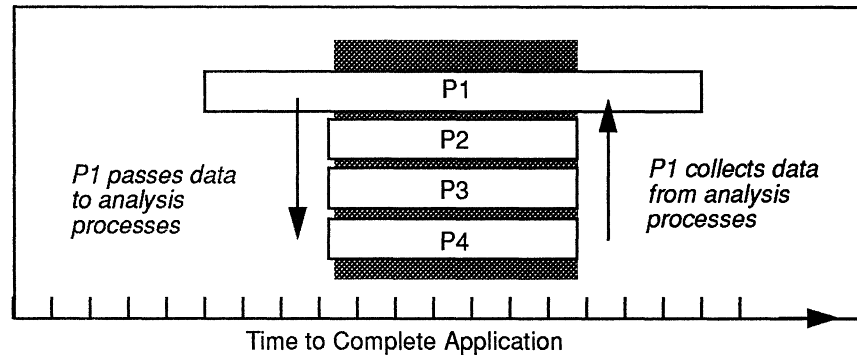


Figure 8 Creating Parallel Processes for a Quad JP SMP

This technique of splitting data is used often in applications that process graphical data. For example, you can split the RGB bands of a graphical data file, process each band's data separately, then collect and post-process the data.

Minimizing Process Context Switching

Because DG/UX is a multi-programming system, you can't avoid context switches completely. Ideally though, you'd like each process in your application to run on a JP until the process's time slice expires. When a time slice expires, the Medium Term Scheduler looks at the run queue to see if there are other processes that need to run.

To achieve this run-to-completion goal with a real world application is probably not possible (unless the application is compute-bound). However, a focus on the goal of keeping processes running on JPs for their entire time slice can help you minimize the number of context switches in an application.

The general way to minimize context switching operations is to partition an application into the number of processes that balance JP utilization and context switches. Some more specific ways of achieving a balance between JP utilization and context switches are listed below.

- Try to balance the workload among the application's processes—don't let one process do all of the work.
- Try to design processes that can run for relatively long periods of time. Too many short duration processes can cause excessive context switches. For example, rather than writing small amounts of data to a file, you might be able to batch the data and write it all at once.
- When you have a very short duration process that has to wait for some I/O, consider keeping the process running by using a busy/wait instead of allowing a context switch. (By short duration, we mean a process that runs for a period of time less than it takes to do a context switch.)

Ideally, you'd like each process in your application to run until the process's time slice expires

- Avoid unnecessary system calls. By taking system calls out of loops or by using library functions, you can enable a process to spend more time running in user space and save the time that it takes a process to switch modes.
- Try to increase the probability that a resource will be available when a process tries to access the resource. Instead of having a process check to see if a resource is available, use one of the interprocess communication techniques (such as semaphores or message queues) that puts a process to sleep if a resource is unavailable, and wakes it up when the resource becomes available.
- Recognize that it's useful to design single-purpose processes if the work that the process has to do is relatively infrequent. An example is a process that listens for mouse input.
- Try to minimize disk I/O operations. Relative to JP time, disk I/O is very time consuming. For example, some applications use files as resource-locking mechanisms. You can minimize disk I/O by replacing file locks with semaphores (described later). Or, you could use memory mapping (**mmap**) operations and shared memory to avoid file-based system calls.

Techniques for Interprocess Communication and Synchronization

The third goal for taking advantage of SMP systems is to minimize contention for resources, by using appropriate interprocess communication and synchronization techniques.

The DG/UX operating system provides several interprocess communication (IPC) facilities (Table 1) that you can use to exchange and share data among processes, and to synchronize processes.

Table 1 IPC Techniques

To Share/Exchange Data	To Synchronize Processes
❑ Shared memory ❑ Message queues ❑ Pipes and sockets ❑ Shared files	❑ Semaphores and **xmem** instructions ❑ Signals ❑ Files used as locks

Table 2 on the next page summarizes the pros and cons of these techniques.

With these IPC techniques, you can implement the same interprocess communication and synchronization models. The trick is to select the most cost-effective techniques for your particular application or mix of applications—cost-effective in the context of a technique's use of system time and resources.

In this technical brief, we're going to focus on the shared memory technique. Unlike the other data sharing/exchanging facilities, shared memory does not provide automatic process synchronization. Therefore, we'll talk about semaphores and **xmem** instructions, which you can use to synchronize processes that are sharing memory.

Table 2 Comparing IPC Techniques

Technique	What Does It Do?	Pros	Cons
Shared memory	Enables processes to map areas of virtual memory into their address spaces. Transfers arbitrary amounts of unformatted data.	▮ Best performance, doesn't use kernel resources or system calls (after memory segments are set up) ▮ Good for sharing large amounts of data ▮ Supports sharing among multiple processes	▮ Requires code to synchronize processes and regulate data flow ▮ Cannot communicate across network
Semaphores	Supports the synchronization of processes. Often used in conjunction with shared memory.	▮ Very flexible synchronization features ▮ Very inexpensive when combined with **xmem** routines	▮ Cannot communicate across a network
Messages	Enables processes to send and receive messages among arbitrary processes via queues. Transfers data bi-directionally in datagrams.	▮ Convenient—easy to code ▮ Automatic synchronization and self regulated data flow ▮ Security—supports private message queues	▮ Requires two data-copy operations ▮ Limits on amount of data that can be transferred ▮ Cannot communicate across a network.
Pipes	Provides explicit communication path between two cooperating processes. Transfers data bi-directionally in streams.	▮ Convenient—easy to code. ▮ Automatic synchronization and self regulated data flow ▮ Signalled when cooperating process has failed.	▮ Requires two data-copy operations ▮ Cannot communicate across a network. ▮ Works only between a parent and its descendant.
Sockets	Provides communication path between (potentially) unrelated processes, across a network. Transfers data bi-directionally in streams or datagrams.	▮ Can communicate among processes on a network ▮ Automatic synchronization and self regulated data flow ▮ Signalled when cooperating process has failed.	▮ Requires two data-copy operations.
Signals	Way of synchronizing processes. Informs a process that an exceptional event has occurred.	▮ Provided automatically by kernel	▮ Expensive—uses significant kernel resources ▮ Difficult to code—maintaining process synchronization is error prone
Files	Send and receive data via shared files.	▮ Very portable ▮ Easy to use ▮ NFS support over network	▮ No automatic synchronization ▮ Slow ▮ Contention for locked files

Shared Memory

In terms of performance, especially if several processes are sharing or exchanging large amounts of data, shared memory is almost always your best choice. The shared memory technique is the fastest and the most general of the IPC techniques. Shared memory enables processes to map the same memory pages into their virtual address spaces.

Once you've set up the shared memory segments, processes can read and write data from and to the segments, without using system calls.

Sharing memory requires only one copy of shared data—using shared memory doesn't require the extra data-copy operation that is needed by other IPC techniques. To pass data through a message queue, for example, requires that data be copied from a process's address space to the kernel (into a message queue) and then to the other process's address space. With shared memory, processes simply read and write data from the shared segment of memory. Therefore, the performance advantage of shared memory becomes more evident as the amount of shared data increases.

Because shared memory provides a very general interface, it requires that you establish the rules of how processes are going to use a shared memory segment. Also, shared memory provides no process-synchronization primitives—you must establish the rules and write the code that supports synchronization. This isn't as bad as it sounds, because semaphores (outlined in the next section) are a natural complement to shared memory.

The system calls that support shared memory are listed below.

- **shmget**—creates a new shared memory segment and sets up the segment's attributes, or gets the identifier of a shared memory segment
- **shmctl**—sets or gets shared memory-segment attributes or destroy a shared memory segment
- **shmat**—attaches a shared memory segment to the virtual address space of a process
- **shmdt**—detaches a shared memory segment from a process's address space

Semaphores and xmem Routines

Semaphores are data structures that are used primarily to synchronize processes that are sharing a resource or service, such as shared memory. By examining the value of a semaphore (or set of semaphores), a consumer process can tell whether it can obtain the resource or service.

Semaphore system calls, by themselves, provide a robust way of handling interprocess synchronization. However, for situations that require maximum performance, you can combine the use of semaphores with assembly language routines that use the AViiON 88K processor's **xmem** instruction. By writing small **xmem** assembly language routines, you can avoid much of the overhead of making semaphore system calls to access shared memory.

Semaphores

The point of using semaphores is to avoid wasting JP resources by having processes continually checking to see if a resource is available. The semaphore facility wakes up processes that are waiting for a semaphore value to change.

The value of a semaphore represents the number of a particular resource that is available. The simplest (binary) semaphores have values of one or zero. If the semaphore's value is zero, the resource isn't available. The resource is locked and a process trying to access the resource will go to sleep (or do something else). When the resource becomes available, the semaphore's value goes to one, and processes waiting for the resource will wake up.

FYI—Origin of Semaphores

The general concept of semaphores was proposed originally by Dijkstra (as the Dekker algorithm). Later, AT&T provided in UNIX System V an enhanced implementation of the semaphore abstraction, and the DG/UX operating system provides that implementation.

A UNIX semaphore is an integer value. A process can perform several operations on a semaphore, including P, V, and wait for zero. A semaphore P operation decrements the semaphore's value; the V operation increments it.

Prior to the implementation of semaphores, UNIX programs often used files to control access to a resource or service. However, file locking is not as robust or efficient as semaphores. File locking presents several problems, such as processes not knowing when to try again to access a resource or the need to clean up locking files if a process exited without removing the file.

You can initialize semaphores to selected values. For example, if you have four instances of a resource, you can initialize a semaphore to four. The first four processes that access the resource decrement the semaphore's value. When a fifth process tries to access the resource, the decrement operation causes the semaphore value to go negative, and the process goes to sleep—waiting for one of the first four processes to finish its work with the resource.

The semaphore facility keeps track of how many processes are waiting for different semaphore events to occur. The facility also keeps track of how many processes are waiting for a semaphore to go to zero, or go to a positive number from zero. Semaphores also provide an option that enables the operating system to undo semaphore operations if a process terminates.

The system calls that support semaphore operations are listed below.

- **semget**—creates a new semaphore (or set of semaphores) and sets up their attributes, or gets the identifier of an existing semaphore.
- **semctl**—performs one of several semaphore commands, such as returning the semaphore's value, returning the PID of the last process that operated on the semaphore, and returning the number of processes that are sleeping on the semaphore.
- **semop**—performs P (decrement), V (increment), or wait-for-zero operations on a semaphore or set of semaphores.

Using xmem Routines and Semaphores to Control Access to a Resource

If your goal is to maximize the performance of shared memory operations, you can consider using the AViiON 88K processor's assembly-language **xmem** instructions in conjunction with semaphores.

The advantage of using **xmem** routines in conjunction with semaphores is that you can avoid the overhead of making semaphore system calls if the shared memory segment (or other resource) that you want to access is available.

The disadvantage of using **xmem** routines is that they perform machine-dependent test and set operations, and programs that use them are not directly portable to non-88K platforms. However, other platforms have similar test and set instructions. Furthermore, assembly language routines that use the **xmem** instructions to speed access to shared memory are typically very small, simple, and easy to isolate.

Obtaining a Resource

Figure 9 shows how to use a **xmem** routines and semaphore system calls to obtain a resource. Because the goal is to obtain and release a resource without using system calls (when there is no contention for the resource), the logic is a bit trickier than you might expect.

The technique uses two locations in shared memory; one location as a resource lock, the other location as a contention flag.

Samples of xmem routines are provided in the "FYI" section on page 23.

The binary resource lock is set to one when a process has obtained the resource. The lock is set to zero when the resource is available. The contention flag does two things: it enables the process to avoid a system call when the process releases the resource, and it enables the "release resource" routines to know when another process is waiting for the resource. (We talk about the contention flag in the next section.)

The first step in obtaining a resource is to use an **xmem** instruction to perform a test and set operation on the lock location's value. If the test is successful, the process gets the lock (sets the lock to one), and can safely access memory without using system calls (the shaded path in the figure).

If the test operation on the lock fails, the process knows that the resource is being used. Because the **xmem** instruction has no facility to put a process to sleep if the test operation fails, the process makes a **semop** system call to set a semaphore, then sets the contention flag.

> **Important Note**
> The resource locking technique described here is biased toward the case where there is little contention for a resource. Performance will actually get worse if there is too much contention for the resource.

It's possible that the process holding the resource lock released the lock during the semaphore system call. Therefore, we test the lock again. If the lock is available, we get it and access the shared resource.

If the resource is still locked, we use a **semop** system call to put the process to sleep and wait for the semaphore to go zero. The routine that releases the resource is responsible for resetting the semaphore.

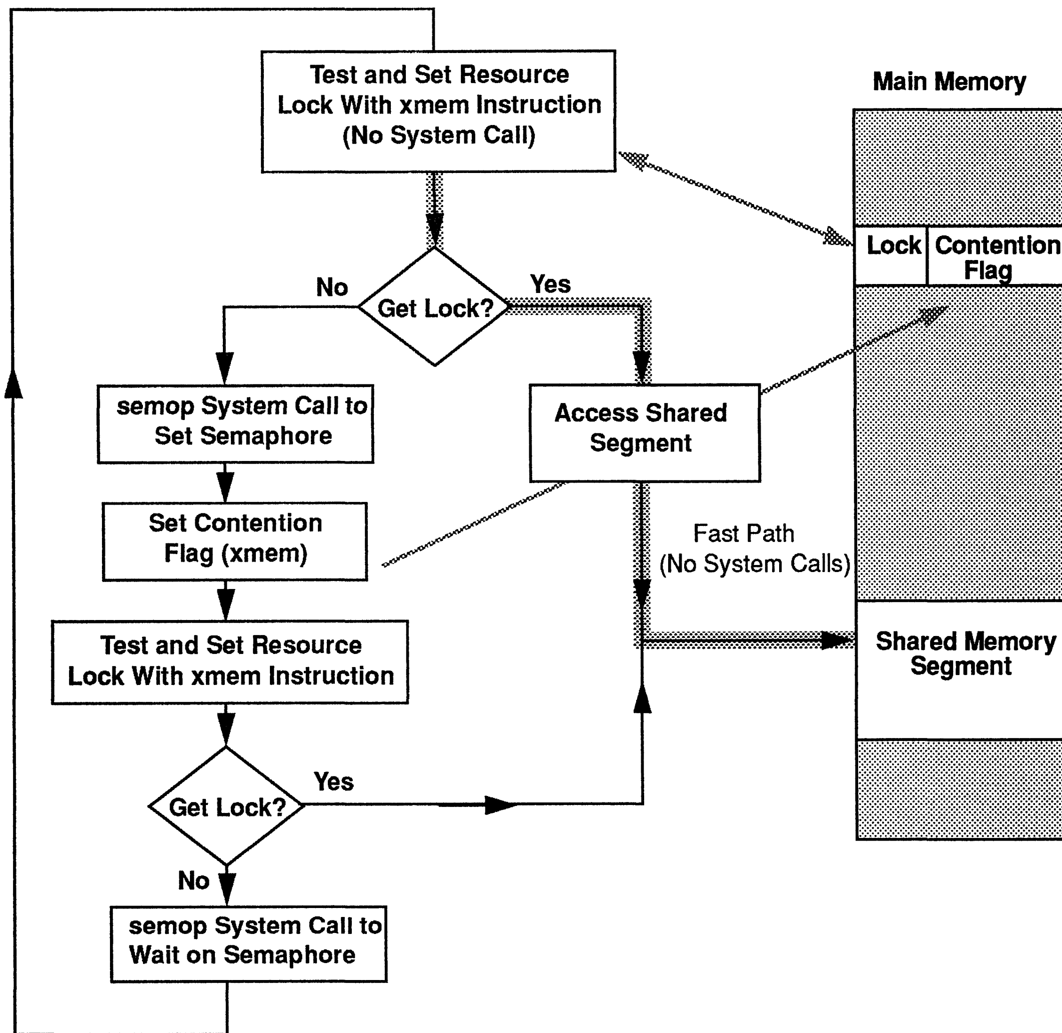


Figure 9 Obtaining a Resource

Releasing a Resource

In the last section, we mentioned the contention flag, but didn't say much about it. The key to releasing an uncontended resource, without making a system call, is to use a contention flag in conjunction with the resource lock.

A process sets the contention flag if it isn't able to access a resource—if the resource's lock is set. When a process finishes accessing a resource, it resets the resource lock and tests the contention flag (Figure 10). If the contention flag is set, the process does a **semop** system call to wake up processes sleeping on the resource's semaphore. If the contention flag isn't set, there are no processes waiting for the resource, and there's no need to do the semaphore operation.

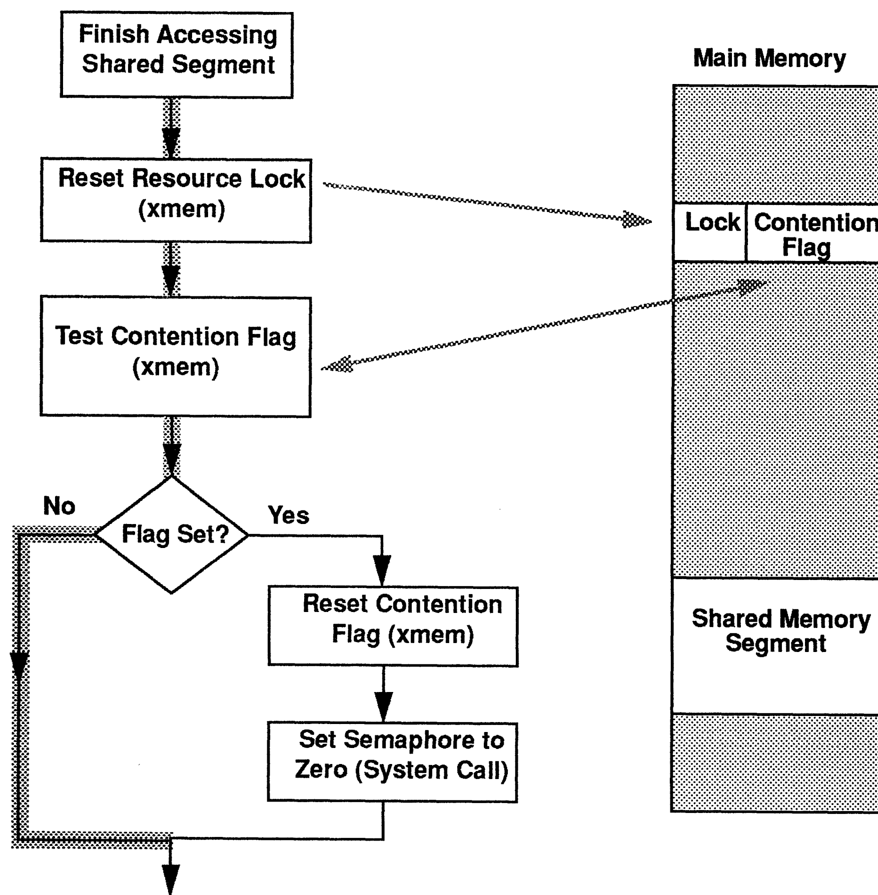


Figure 10 Releasing a Resource

Note – A process that uses **xmem** routines with semaphores should include signal-catching routines that advance semaphores and release any locks if the process terminates unexpectedly.

FYI—Sample xmem Routines

The following set of sample routines were developed at DG RTP for use with semaphores and shared memory. These routines support unsequenced locks. Support for sequenced locks requires somewhat more complex code.

Unsequenced locks are pending locks. If a process tries to obtain a lock that is held by another process, the requesting process will be pended until the lock is released. If several processes are waiting for the same lock, all processes that are waiting on the lock will be awakened when the lock is released, and the operating system's scheduler will determine which process to run. Therefore, this lock does not guarantee fairness, and a low priority process can starve if a lock has heavy contention. This implementation is biased toward the case where there is little or no contention for the lock. In this case, no system calls are needed, and processes can perform a lock or unlock operation in about 10 instructions.

The routines, which are shown in the following sections, are:

- **lock_def.h**—header definitions
- **lock_util.c**—C routines
- **lock_mgr.s**—assembly language routines

lock_def.h

The **lock_def.h** file defines an unsequenced lock type, which contains the variables **held**, **contended**, and **semid**.

A value of zero for **held** indicates that the lock is not being held, and a process can obtain it immediately. A value of one indicates that another process holds the lock.

The **contended** value is relevant only when **held** has a value of one. If the value of **contended** is zero, no other process are waiting for the lock. A value of one means that other processes are waiting for the lock.

The **semid** variable is an integer that contains the identification number of the semaphore on which processes will wait if they cannot obtain the lock immediately.

```
typedef struct
        {
        unsigned long                   held;
        unsigned long                   contended;
        int                             semid;


        }  unsequenced_lock_type   ;


typedef unsequenced_lock_type *      unsequenced_lock_ptr_type ;
```

lock_util.c

The module **lock_util.c** contains C language routines for implementing unsequenced locks.

All code that uses a lock should be within the pair of functions **initialize_unsequenced_lock** and **deinitialize_unsequenced_lock**.

The functions **wait_for_unsequenced_lock** and **wakeup_unsequenced_lock** are internal to the implementation of the locks and should not be called directly by users.

The obtain (lock) and release (unlock) functions for unsequenced locks are in the module **lock_mgr.s** (page 26).

```
/*<-------------------------------------------------*/
/* lock_util.c */
/*>-------------------------------------------------*/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

#include "lock_def.h"

/*.function */

void initialize_unsequenced_lock (lock_ptr)

unsequenced_lock_ptr_type     lock_ptr;      /* READ WRITE */

/*
/*      This function initializes an unsequenced lock. This function
/*      must be called before any lock operations (obtain, release)
/*      are applied against the lock.
/*
/*      The caller is responsible for allocating the storage for the
/*      lock instance. This function simply initializes that
/*      storage.
/*
{

lock_ptr->held = 0;
lock_ptr->contended = 0;
lock_ptr->semid = semget(IPC_PRIVATE, 1, 0777);
if (lock_ptr->semid == -1)
        {
        printf("Semget failed in lock initialize with errno %d.\n", errno);
        }
}
/*.End_Function initialize_unsequenced_lock */


/*.function */

int deinitialize_unsequenced_lock (lock_ptr)

unsequenced_lock_ptr_type     lock_ptr;      /* READ WRITE */
```

```
/*
/*      This function performs the inverse of the
/*      initialize_unsequenced_lock routine. Any resources allocated
/*      for use in implementing the lock functions are freed.
/*      Note that this function does NOT reclaim the memory
/*      occupied by the lock. Management of the memory is the
/*      responsibility of the caller.
/*

{

if (semctl(lock_ptr->semid, 0, IPC_RMID, 0))
        {
        printf("Semctl failed in lock deinitialize with errno %d.\n", errno);
        }
}
/*.End_Function deinitialize_unsequenced_lock */


/*.function */

void wait_for_unsequenced_lock (lock_ptr)

unsequenced_lock_ptr_type     lock_ptr;      /* READ ONLY */

/*
/*      This function waits for an unsequenced lock to be released.
/*      When the lock is released, this function tries again to
/*      obtain the lock, and returns only when the lock has been
/*      successfully obtained.
/*
/*      Note that this function may pend waiting for the lock to
/*      become available.
/*

{

struct sembuf          sembuf[1];
union semun            semun;

while (!obtain_unsequenced_lock_no_wait(lock_ptr))
        {
        semun.val = 1;
        if (semctl(lock_ptr->semid, 0, SETVAL, semun) == -1)
                {
                printf("Lock wait semctl failed with errno %d.\n", errno);
                }
        lock_ptr->contended = 1;
        if (obtain_unsequenced_lock_no_wait(lock_ptr))
                {
                break;
                }
        sembuf[0].sem_num = 0;
        sembuf[0].sem_op = 0;
        sembuf[0].sem_flg = 0;

        if (semop(lock_ptr->semid, sembuf, 1) == -1)
                {
                printf("Lock wait semop failed with errno %d.\n", errno);
                }
        }

}
```

```
/*.End_Function wait_for_unsequenced_lock */


/*.function */

void wakeup_unsequenced_lock (lock_ptr)

unsequenced_lock_ptr_type     lock_ptr;      /* READ ONLY */

/*
/*     This function is part of the internal implementation of
/*     unsequenced locks. It is called when a lock is released
/*     and there are processes waiting for the lock. This function
/*     awakens those waiting processes.
/*

{

union semun    semun;

lock_ptr->contended = 0;
semun.val = 0;
if(semctl(lock_ptr->semid, 0, SETVAL, semun) == -1)
       {
       printf("Lock wakeup semctl failed with errno %d.\n", errno);
       }
}

/*.End_Function wakeup_unsequenced_lock */

/*.End_Module lock_util.c */
```

lock_mgr.s

The module **lock_mgr.s** contains the assembly language functions **obtain_unsequenced_lock**, **release_unsequenced_lock**, and **obtain_unsequenced_lock_no_wait.**

```
;*.function *;

; void _obtain_unsequenced_lock (lock_ptr)

;unsequenced_lock_ptr_type    lock_ptr;      /* READ WRITE */

;*
;*     This function obtains an unsequenced lock. If the lock is
;*     not immediately available, the calling process is pended
;*     until the lock is available.
;*

_obtain_unsequenced_lock:
       or      r3,r0,1                       ; Get a constant 1
       xmem    r3,r2,r0                      ; Try to obtain the lock
       bcnd    ne0,r3,wait_for_lock          ; Did we obtain the lock?
       jmp     r1                            ; Yes. Return to the caller.

wait_for_lock:
       br      _wait_for_unsequenced_lock    ; go wait for the lock

;*.End_Function _obtain_unsequenced_lock *;
```

```
;*.function *;

; void _release_unsequenced_lock (lock_ptr)

;unsequenced_lock_ptr_type    lock_ptr;      /* READ WRITE */

;*
;*      This function releases an unsequenced lock that was previously
;*      obtained. If any processes are waiting on the lock, they are
;*      awakened.
;*

_release_unsequenced_lock:
        st      r0,r2,0                     ; Clear the 'held' flag
        ld      r3,r2,4                     ; Get the contended flag
        bcnd    ne0,r3,wakeup_waiters       ; Is anybody waiting?
        jmp     r1                          ; No. Return to the caller.

wakeup_waiters:
        br      _wakeup_unsequenced_lock    ; Go wakeup waiters

;*.End_Function _release_unsequenced_lock *;


;*.function *;

; int _obtain_unsequenced_lock_no_wait (lock_ptr)

;unsequenced_lock_ptr_type    lock_ptr;      /* READ WRITE */

;*
;*      This function obtains an unsequenced lock if it can, but does
;*      not pend if some other process holds the lock.
;*
;*      Return_Value:
;*
;*      %rval("1") The lock was successfully obtained. The calling
;*      process holds the lock.
;*
;*      %rval("0") The lock was not obtained because some other
;*      process holds the lock.
;*

_obtain_unsequenced_lock_no_wait:
        or      r3,r0,1             ; Get a constant 1
        xmem    r3,r2,r0            ; Try to obtain the lock
        cmp     r3,r3,r0            ; Compare with 0
        jmp.n   r1                  ; Return to the caller
        extu    r2,r3,1<eq>         ; Return equal bit

;*.End_Function _obtain_unsequenced_lock_no_wait *;
;*.End_Module lock_mgr.s *;
```

Semaphores and xmem Routines—Test Results

We developed a program to test the hypothesis that the use of **xmem** routines and semaphores is an efficient way of synchronizing processes that are exchanging data.

The test program timed how long it took for one and two pairs of processes to exchange "N" bytes of data 1,000 times, using shared memory. The test was run on a 25 Mhz quad-processor AViiON AV5240 with 400 Mbytes of memory, running at init level 1.

The amount of data exchanged ("N") was 1K, 2K, 4K, and 16K. For each combination of process-pairs and amount of data, we made one run using semaphores to synchronize the processes and one run using semaphores plus **xmem** routines to synchronize the processes.

Table 3 shows the time (in seconds) to complete 1,000 one-way transfers at each of four transfer sizes. The times are wall clock times, measured with the **time** command. In all cases, the use of semaphores plus **xmem** routines offers a performance advantage over the use of just semaphores. We've put in parenthesis the time (in seconds) that semaphore plus **xmem** was faster than using semaphores alone.

Table 3 Data From Semaphore Tests

	Number of Process Pairs	
	1	2
	1 Kbyte Data Transfers	
Semaphores	1.6	3.0
Semaphores + **xmem**	1.5 (0.1)	2.7 (0.3)
	2 Kbyte Data Transfers	
Semaphore	3.2	5.7
Semaphores + **xmem**	2.5 (0.7)	4.9 (0.8)
	4 Kbyte Data Transfers	
Semaphore	7.5	11.4
Semaphores + **xmem**	4.7 (2.8)	6.3 (5.1)
	16 Kbyte Data Transfers	
Semaphore	28.1	43.5
Semaphores + **xmem**	19.2 (8.9)	33.9 (9.6)

Because the times in the table are wall clock times, they include the time that it takes to start the processes, the time to transfer the data, and the time to synchronize the processes. The times in parentheses isolate the difference in synchronization times between the two techniques.

Figure 11 shows graphically the percentage or time that the use of semaphores and **xmem** routines was better than using semaphores alone. On the average, the time to exchange the data was 24% better for semaphores plus **xmem** routines.

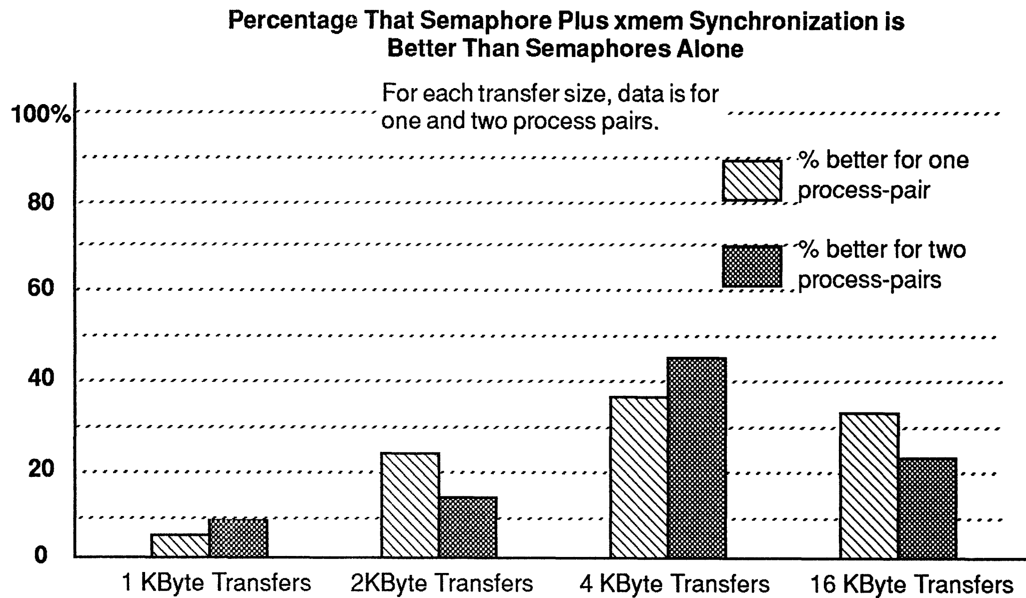


Figure 11 Comparison of Semaphores and Semaphores Plus xmem Routine

Remember that the performance increases that are highlighted by this test are based on the assumption that there is little contention for the shared memory resource. The assumption is that you will generally get the lock (and access the resource). In cases where there is potentially heavy contention for a resource, the use of semaphores by themselves may offer performance advantages.

FYI—Other Interprocess Communication Techniques

For comparison, this section talks about some of the other interprocess communication techniques that were mentioned earlier in this technical brief. These techniques include:

- signals
- message queues
- pipes and sockets

Signals

Signals are asynchronous software interrupts that tell a process that an error or exceptional event has occurred. When a process receives a signal, the process has the option of taking some action other than its normal execution path.

Although you can use signals to synchronize data-sharing processes, they were not designed for use in a synchronization role. Compared to semaphores, signals are expensive users of kernel resources.

Message Queues

Message queues are a bi-directional facility that enables processes to exchange packets of data (messages) by sending and receiving messages to and from a kernel buffer (a message queue). To borrow from network terminology, message queues support datagram connections.

The performance of message queues isn't as good as that of shared memory because message queues require two data-copy operations—one to send a message to the queue, the other to send the message from the queue to the receiving process's address space. However, compared to shared memory, message queues can be a good choice if programming convenience outweighs performance, and you are infrequently passing relatively small amounts of data. Regardless, message queues are a significantly more efficient way of exchanging data than using files in the file system.

Message queues are not designed to work across different machines. To pass data across networks requires that you use sockets, which are described in the next section.

Pipes and Sockets

Pipes and sockets are designed to transfer arbitrary amounts of data among processes. Unlike message queues, which transfer data in packets, pipes and sockets work with streams of data. Borrowing again from network terminology, pipes support streams connections, while sockets can support both streams and datagram connections.

Pipes support transfers of data among process on a single machine. Sockets support transfers of data across a network.

The performance of pipes and sockets isn't as good as that of shared memory because pipes and sockets require two data-copy operations: from the address space of the sending process to a kernel buffer, then from the kernel into the receiving process's address space. However, pipes and sockets can be a good choice if you are passing relatively small amounts of unformatted data and programming convenience outweighs performance.

Pipes

There are two kinds of pipes: unnamed and named. Unnamed pipes provide a communication path between related processes—the process that created the pipe and its descendants. Named pipes support communication among unrelated processes. Unlike unnamed pipes, named pipes have a pathname; a name in a file system's name space.

In the DG/UX 5.4 operating system, pipes are implemented with the Streams facility instead of using the file system as in earlier versions of the operating system. That means that pipes are now bi-directional and that each end of the pipe is a Streams interface.

Pipes automatically synchronize processes. A process can sleep while waiting for data to come into the pipe. Likewise, a process can sleep while waiting for a pipe to empty.

Another advantage of pipes over message queues is that processes using the pipe are signalled if the process at the other end of the pipe terminates. The kernel detects that a process in a pipe-pair has exited and sends to the other a SIGPIPE signal.

Sockets

Sockets, originally a BSD extension to UNIX, are a standard part of the DG/UX operating system. Sockets provide processes with a general way of using network protocols to communicate across networks. They can be used to pass streams of data or to pass datagrams (packets of data).

Sockets allow you to write client and server programs that are independent of the type of network that the programs use to communicate. Sockets support network independence by enabling you to specify the type of

network protocol (the socket's domain) that you want a process to use. As examples, sockets in the Unix domain enable processes to communicate locally within a machine. Sockets in the TCP/IP domain enable the same processes to communicate across a network. This allows a process such as an X windows client to run on the same machine as the server process, or use a network domain, such as TCP/IP, to access a server on a different machine.

Sockets are implemented in layers (Figure 12). The socket layer accepts system calls from a user process and sends them to the protocol layer. The protocol layer contains code that implements the functions of a particular network. The protocol layer typically supports sub-layers. For example, a TCP/IP protocol layer implements TCP's virtual circuits at one level and the IP's handling of datagrams at a lower level. The device driver layer is machine dependent, and provides support for physical protocols, such as Ethernet.

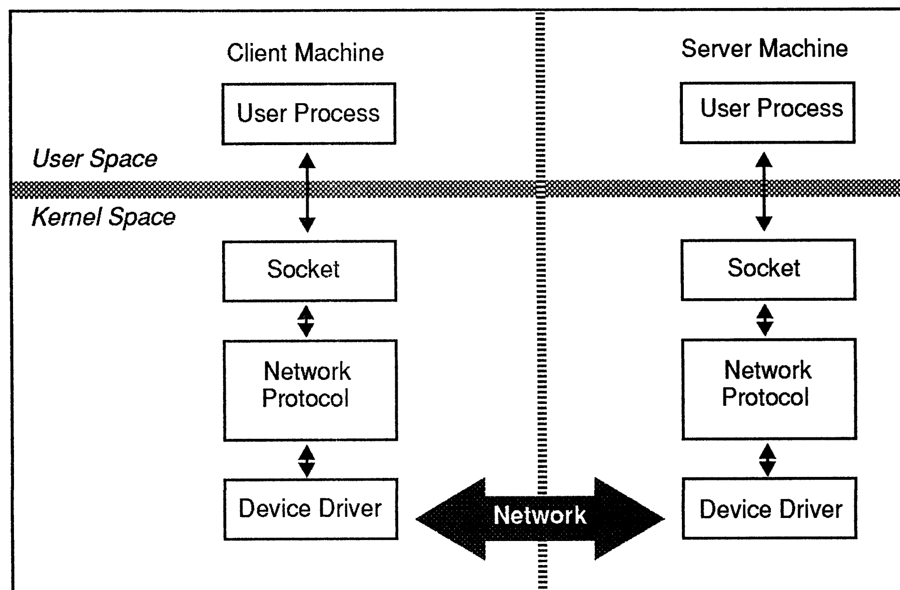


Figure 12 Layered Implementation of Sockets

For More Information

Among the articles that discuss multiprocessor systems in more detail are:

DG/UX™ *Technical Brief: A Second Look at Multiprocessor SMPs (012-003886-02), July 31, 1991, Data General Corporation*