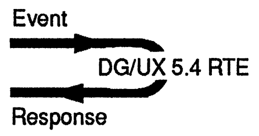
Data General

DG/UX Technical Brief

December 22, 1992

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



In This Issue:

POSIX Realtime Extensions in the DG/UX™ 5.4 R2.01 Operating System

Contents

In response to requests that Data General's UNIX Development Group has received from customers, we are releasing some realtime extensions with the R2.01 version of the DG/UX 5.4 operating system. These realtime extensions add to the operating system several process scheduling enhancements that can be used with many non-critical (soft) realtime applications.

The DG/UX 5.4 operating system already provides support for realtime applications in many areas, including memory locking, memory mapping, and shared memory. However, our customers asked for additional extensions that would provide more support for realtime applications in the areas of process scheduling and synchronization.

Therefore, in addition to the realtime support already available in DG/UX 5.4, the realtime extensions in DG/UX 5.4 R2.01 support new system calls for:

- ❑ process scheduling
- ❑ process synchronization with semaphores
- ❑ process synchronization timers

We'll refer to this set of realtime extensions as *DG/UX 5.4 RTE.*

To maintain Data General's commitment to standardization, we implemented the system calls for these three areas using the functional descriptions of POSIX 1003.4 Draft 12 (P1003.4/D12). P1003.4/D12 describes twelve other sets of realtime options in addition to the three process-scheduling areas that are available in DG/UX 5.4 R2.01.

In addition to the process scheduling and synchronization enhancements, DG/UX 5.4 R2.01 has increased the performance of signal handling routines.

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

012-004244-00

Note that there are no performance penalties if you choose to upgrade to DG/UX 5.4 R2.01 and don't use the extension's realtime enhancements. In fact, the realtime enhancements that are described here will become part of the standard DG/UX operating system in a future release.

What's in This Technical Brief

This technical brief focuses on the new realtime extensions that DG/UX 5.4 R.2.01 provides in the areas of scheduling services, semaphores, and timers. Tables in each section highlight the new system calls associated with these enhancements.

An "FYI" section at the end of the technical brief has a table that summarizes the fifteen P1003.4/D12 options, and highlights the functions and options that DG/UX 5.4 RTE provides.

We'll start by introducing examples of realtime applications, and then review of some of the realtime terminology that's used in this brief.

Realtime Applications and Their Requirements

What is a realtime application?

> A realtime application provides predictable responses to events, within the acceptable response times that are dictated by the application.

Realtime applications have two dimensions; the *speed* of response to an event and the *predictability* of the response, with the focus on predictability. From application to application, the requirement for speed of response is different, but the requirement for predictable response is the same.

Applications that require very fast response to events typically run on special-purpose realtime computers. The combination of DG/UX 5.4 RTE and AViiON computers provides support for applications that are less response-time critical but still require predictable process scheduling.

Realtime applications require fast and predictable process scheduling.

Some situations that realtime applications might be used include:

- ❑ laboratory data collection and analysis
- ❑ industrial process monitor/process control
- ❑ aircraft and industrial simulations
- ❑ interactive multimedia

In realtime applications, processes:

- ❑ *must* respond to events within predetermined time limits
- ❑ *must* start and finish executing within predetermined time limits

Realtime events can originate from outside of the application or from within it. For example, a realtime application might monitor parameters such as temperature and pressure in a piece of production equipment and respond to out-of-limit values by sending commands to open and close valves. Events that originate within an application include a process finishing some calculations or signals generated by a timer in a process.

Realtime applications require a guarantee from the operating system that it will complete the applications' realtime processes within predetermined time limits. The operating system must provide the guaranteed response time, regardless of what other applications the operating system is running.

Realtime Terminology

Here are definitions of some realtime terms that are used in this brief.

Deterministic

A deterministic scheduling policy guarantees that a process will always be scheduled the same way, relative to other processes on the system. Note that a deterministic policy makes no guarantees about the time that it takes to complete a process.

Heuristic

A heuristic scheduling policy makes decisions about which processes to schedule and how long each process's time slice should be. For example, the DG/UX 5.4 Medium Term Scheduler (MTS) uses heuristic algorithms to support fair time-sharing scheduling policies. The MTS assigns and keeps track of processes' Job Processor (JP) usage, and adjusts process-priorities of interactive (I/O intensive) and compute-bound processes. Typically, heuristic scheduling policies are not used to support realtime requirements because the policies are non-deterministic and process scheduling timings can vary.

Hard realtime

A hard realtime requirement is absolute, with no margin for error. The process that controls an event *must* always be scheduled, executed, and completed within a specified time. An example is a valve in a piece of production equipment that must be closed within a few milliseconds after some event occurs, to prevent damage to the equipment. Hard realtime applications are generally run on special-purpose computers.

Soft realtime

A soft realtime requirement allows some (typically small) margin for timing error. In contrast to a hard realtime requirement, no damage will be done if the upper scheduling limit isn't met 100% of the time. In other words, the process scheduler's agreement with an application is that the scheduler "will try to provide a specified reaction time." The realtime extensions of DG/UX 5.4 RTE are designed to be used in soft realtime situations.

DG/UX 5.4 R2.01 Signal Handling Enhancements

Although not specifically part of the POSIX realtime enhancements, DG/UX 5.4 R2.01 includes enhancements to signal handling performance.

In the technical brief *Taking Advantage of Symmetric Multiprocessor Systems* (012-004177), we mentioned that the UNIX signal mechanism is often a poor choice for use in general-purpose process synchronization. Compared to other synchronization methods, signals are expensive users of kernel resources.

However, some application programs use signals to synchronize processes that are sharing data. To increase the performance of these programs, DG/UX 5.4 R2.01 includes enhancements that increase signal handling performance by approximately 30% over signal performance in standard DG/UX 5.4.

Realtime Process Scheduling Enhancements

DG/UX 5.4 RTE provides eight new process scheduling system calls that are described in P1003.4/D12. These system calls enable you to associate processes with three different scheduling policies, instead of the single time-sharing policy that is provided by the standard DG/UX operating system.

To establish a background for how these new realtime system calls work, we talk first about how processes are scheduled in standard DG/UX, then describe the differences of the DG/UX 5.4 RTE implementation.

FYI—Process Scheduling Terminology

Bound

A process that the Medium Term Scheduler (MTS) has associated with a Virtual Processor (VP). A process must be bound to a VP before the MTS can move it to the VP Eligible List.

Runnable

A VP (process/VP pair) that can execute on a (Job Processor (JP). A runnable VP is not waiting for an event to occur (not sleeping). The MTS can place runnable VPs onto the VP Eligible List.

Running

From the perspective of the MTS, a VP that the MTS has moved into the VP Eligible List. A running VP may not actually be executing on a JP, depending on its priority relative to other running VPs and the availability of JP resources.

VP (Virtual Processor)

A software representation of a JP. The MTS binds processes to VPs so that the processes can run. It is the process/VP pair that executes on a JP. Therefore, it is technically correct to say that "processes are bound to VPs, which run on JPs," and we make that distinction in this technical brief. However, most people say simply that "processes run on JPs."

Executing

A process/VP pair that is actually using JP resources—the process's code is being executed by a JP.

Sleeping

A process/VP pair that is waiting for an event to occur—the process is sleeping (blocked) on the event. An example is a process that is waiting for I/O.

Process Scheduling in Standard DG/UX

In the standard version of the DG/UX operating system, processes are scheduled onto Virtual Processors (VPs) and then onto Job Processors (JPs) by the combination of the high-level Medium Term Scheduler (MTS) and the lower-level dispatch scheduler (Figure 1).

The MTS establishes scheduling policies. It binds processes to VPs and uses sophisticated timesharing heuristics to ensure "fair" scheduling among a system's processes. When the MTS binds a process to a VP, it notifies the lower-level dispatch scheduler (*the dispatcher*) that the VP is available. Once the MTS passes information about the VP to the dispatcher, the MTS considers the VP to be "running."

When the dispatcher receives information about a runnable VP, it places the information into the *VP Eligible List*. The dispatcher places the highest priority VP at the head of the list, followed by the next lower priority VP, and so on. When a JP becomes available, the dispatcher places the highest priority VP onto the JP and removes that VP from the list.

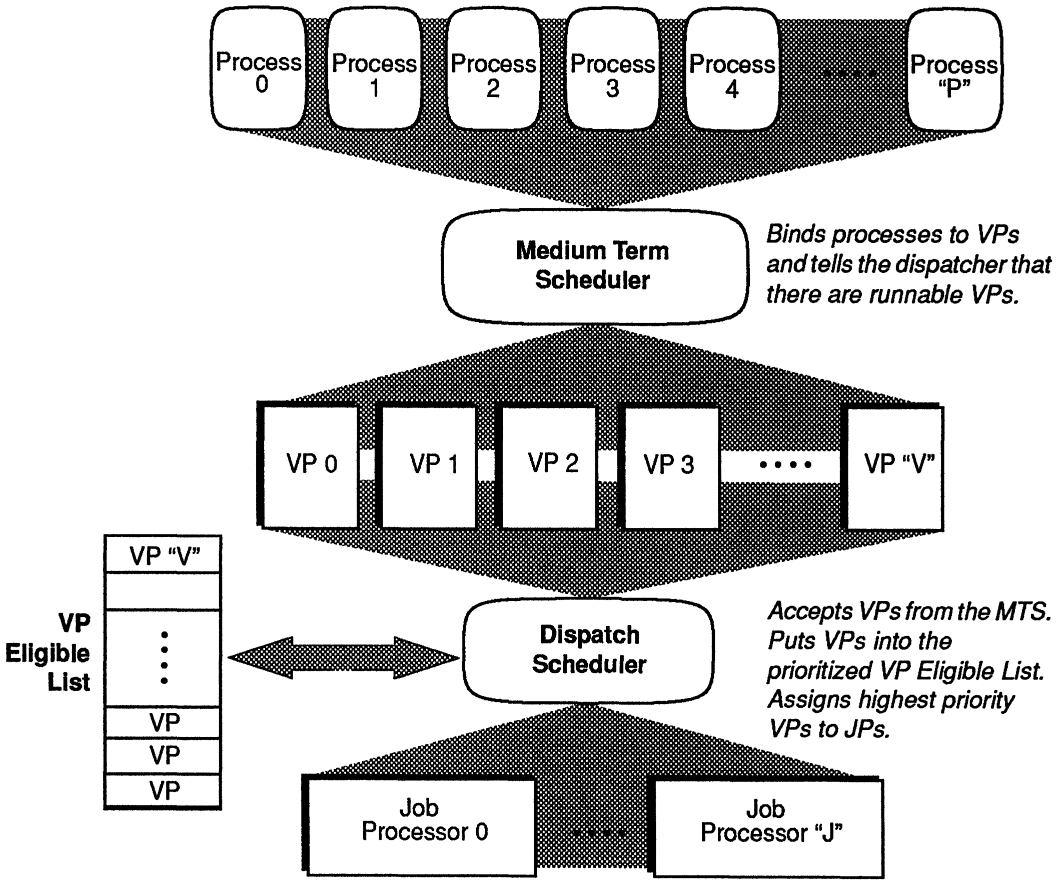

Process 0
Process 1
Process 2
Process 3
Process 4
....
Process "P"
Medium Term Scheduler
Binds processes to VPs and tells the dispatcher that there are runnable VPs.
VP 0
VP 1
VP 2
VP 3
....
VP "V"
VP Eligible List
VP "V"
VP
VP
VP
Dispatch Scheduler
Accepts VPs from the MTS. Puts VPs into the prioritized VP Eligible List. Assigns highest priority VPs to JPs.
Job Processor 0
Job Processor "J"

Figure 1 Medium Term and Dispatch Scheduling

More About the Role of the Dispatcher

The dispatcher's scheduling technique is simple round robin. When a JP is available, the dispatcher places onto that JP the highest priority VP; that is, the VP that is at the head of the VP Eligible List.

To prevent a process from monopolizing a JP, the dispatcher allocates an on-JP time slice to each VP. The dispatcher's time slice is fixed.[1] When a VP uses up its time slice, the dispatcher preempts the VP to enable another process of the same priority to run. The dispatcher places VPs that have used up their time slices behind other VPs of the same priority.

More About the Role of the MTS

The MTS's most important role is to determine which processes it should bind to VPs and send to the dispatcher for execution. To do that, the MTS evaluates the availability of system resources (primarily memory), and the priorities of the processes.

The MTS establishes VP-scheduling policies; the dispatcher carries out the policies.

The MTS also allocates time slices to VPs. An MTS time slice typically consists of several dispatcher time slices. When a process uses up its MTS time slice, the dispatcher takes the process off of the JP and notifies the MTS. The MTS can then change the process's priority and send the process back to the dispatcher.

The MTS can also change a process's priority and high-level time slice, which enables the MTS to dynamically satisfy the needs of interactive (I/O intensive) and compute-bound processes. The MTS gives better priorities and shorter time slices to interactive processes. Interactive processes require quick response time and tend to block themselves waiting for user input. In contrast, the MTS gives lower priorities and longer time slices to compute-bound processes. Compute-bound processes, which do not require quick response times, run without blocking and can monopolize JP resources.

A Scheduling Example

Figure 2 is an example of how this scheduling works. In this scheduling snapshot, process numbers 0 and 4 are running and executing (bound to VPs 3 and 1 respectively). Process 1 is bound to VP 0. It is running, but not executing. Process 2, which is sleeping, is bound to VP 2. Because its process is sleeping, VP 2 does not appear in the VP Eligible List. Unless the MTS reprioritizes the VPs, VP 0 will be next to run.

1. The dispatcher time slice is set in the CFMAXTIME kernel configuration parameter.

The MTS can use its scheduling heuristics to change process priorities (and therefore the scheduling order). For example, the MTS can decide that process 0 on VP 3 is using too much (or too little) JP time relative to other processes, stop the process, and lower or raise its priority. The effect is to move the process's VP toward the tail or the head of the VP Eligible List.

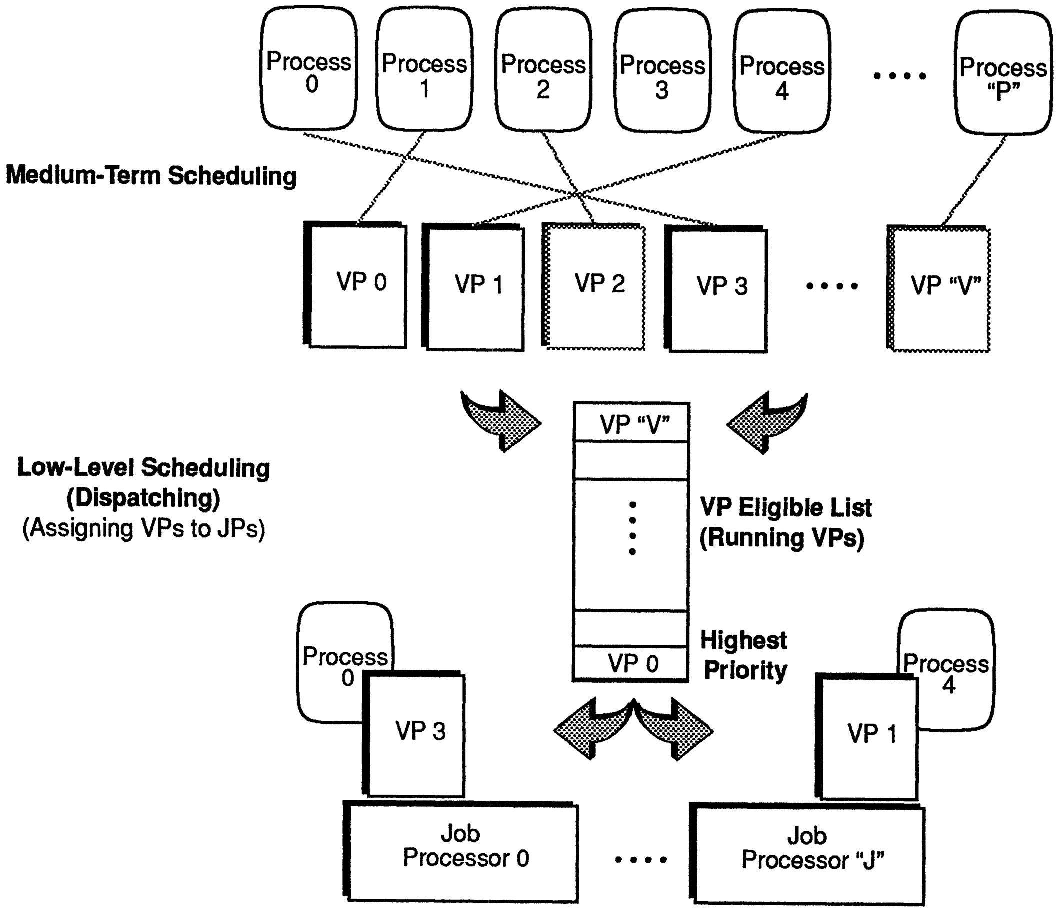


Figure 2 Time-Shared Scheduling of Processes, Virtual Processors, and JPs

Process Scheduling in DG/UX 5.4 RTE

The time sharing algorithms that we described in the previous section are not acceptable for realtime processing—the algorithms are heuristic; not deterministic. Because the MTS and dispatcher have complete control over when (and for how long) processes run, the operating system cannot guarantee that one particular process will always preempt another—a requirement of realtime process scheduling.

To meet the needs of realtime processes, an operating system must have scheduling algorithms that provide deterministic control over when processes will run.

DG/UX 5.4 RTE Scheduling Policies

Instead of the single time-sharing policy of the standard DG/UX operating system, DG/UX 5.4 RTE supports three kinds of scheduling policies, called *scheduling classes*. Using the realtime system calls, you associate processes with both a scheduling class and a priority within the class.

The three scheduling classes, as described by P1004.4/D12, are:

- First in, first out (FIFO), fixed priority scheduling
- Round robin
- Others

The three scheduling classes co-exist on a system, enabling you to have strict control over how an application's processes will be scheduled.

DG/UX 5.4 RTE provides four scheduling classes: FIFO, round robin, standard time-share, and DG/UX-FIFO.

The FIFO class, round-robin class, and one "others" scheduling class are required. The "others" classes are implementation dependent, and an operating system can have one or many "others" classes. DG/UX 5.4 RTE has two scheduling policies in the "others" class: the standard time-share class of DG/UX 5.4[1] and a special-purpose DG/UX-FIFO scheduling class.

For portability, an application's processes should be in the same scheduling class. The interaction among the "others" classes is implementation-defined, so there is no guarantee that the interaction among classes in this category will be the same among different vendors' operating systems.

About Process Priorities

POSIX P1003.4/D12 requires that each scheduling class have a range of at least 32 priorities. However, an implementation can have more priorities within a class. DG/UX 5.4 RTE, for example, provides a 7K range of priorities that is broken into sub-ranges. The priority range of one class can overlap the range of another class. Therefore, scheduling classes are differentiated by their scheduling policies; not necessarily by their priorities.

Figure 3 shows the seven priority-scheduling ranges that are provided by the DG/UX 5.4 RTE operating system. Each range provides 1K priorities.

The horizontal bands in the figure show the current assignments for process priorities in the DG/UX 5.4 RTE operating system. Critical kernel processes are assigned to the highest range of priorities, followed by the range (shaded) that is used by for FIFO and round-robin realtime processes. Processes in this range have higher priorities than time-share

1. We mentioned earlier that standard DG/UX 5.4 provides a type of round robin process scheduling. However, the rules for DG/UX 5.4's standard scheduling are different than the rules for the POSIX round robin scheduling class. Therefore, standard DG/UX 5.4 scheduling falls into an "others" class of P1003.4/D12.

processes, but lower priorities than critical kernel processes. Standard time-share processes and non-critical kernel processes are in the lower priority ranges.

The vertical bars in the figure show the relationship of the FIFO, round robin, time share, and DG/UX-FIFO priority ranges. Processes in the special-purpose DG/UX-FIFO scheduling class can have any priority, which requires that you use this scheduling class with care.

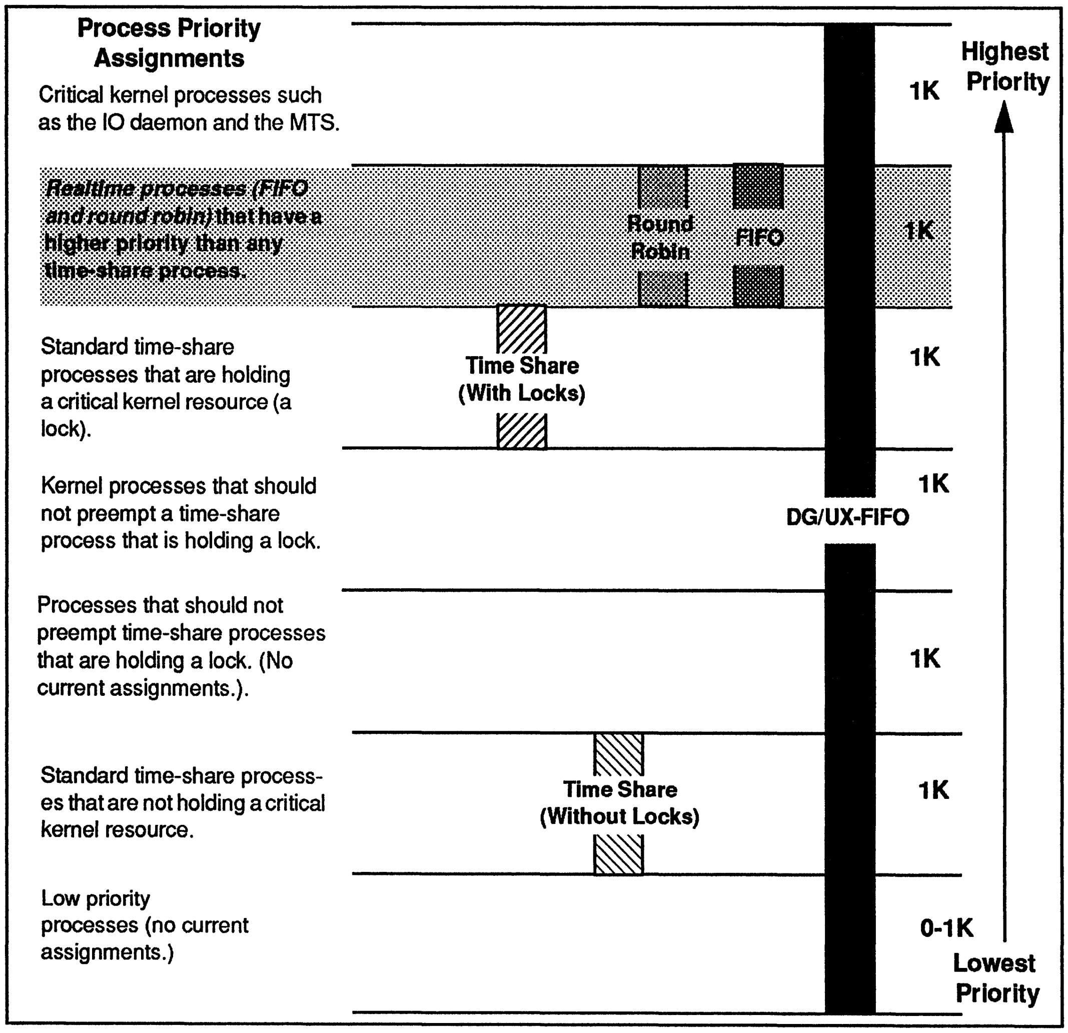

Process Priority Assignments
Critical kernel processes such as the IO daemon and the MTS.
Realtime processes (FIFO and round robin) that have a higher priority than any time-share process.
Standard time-share processes that are holding a critical kernel resource (a lock).
Kernel processes that should not preempt a time-share process that is holding a lock.
Processes that should not preempt time-share processes that are holding a lock. (No current assignments.).
Standard time-share processes that are not holding a critical kernel resource.
Low priority processes (no current assignments.)
Round Robin
FIFO
Time Share (With Locks)
Time Share (Without Locks)
DG/UX-FIFO
1K
1K
1K
1K
1K
1K
0-1K
Highest Priority
Lowest Priority

Figure 3 Scheduling Priority Ranges and Assignments for DG/UX 5.4 RTE

FYI—The Scheduling-Priority Numbers Game

In the scheduling classes of POSIX P1003.4/D12 and DG/UX 5.4 RTE, higher numbers are associated with higher priority processes. For example, a running VP of priority 64 will preempt an executing VP of priority 61. This numbering scheme is different from most UNIX systems (including standard DG/UX 5.4), which assign lower priority numbers to higher priority processes.

The FIFO Scheduling Class

The FIFO scheduling class provides fixed-priority process scheduling for realtime support.

FIFO processes have an infinitely long time slice. Once a FIFO-class process is executing on a JP, the process stays on a JP until the process:

- completes
- is preempted by a higher priority process
- is blocked by an action that it takes
- voluntarily gives up the JP

When the FIFO process completes, is blocked, or gives up a JP, the dispatcher runs the next highest priority process.

You can use the sched_yield() realtime system call to allow a process to voluntarily give up a JP, enabling you to prevent a compute-bound process from monopolizing JP resources. The "yielding" process moves to the end of its priority queue, allowing any processes of the same priority to obtain a JP.

Within the FIFO classes' scheduling list, processes are arranged by priority sublists (Figure 4). Higher numbers represent higher priorities. We've arbitrarily used a priority range of 65-96 in the figure. (See the "FYI" section on page 9 for a note about priority numbering.)

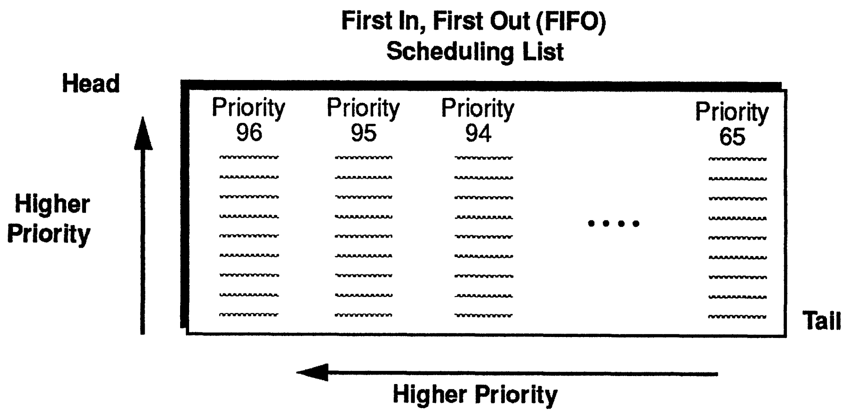


Figure 4 FIFO Scheduling List

Here's an overview of the scheduling policies for processes in the realtime FIFO class:

- When an executing process is preempted, it remains in its place in the priority list.
- When an executing process issues a sched_yield() system call to give up a JP, the process is placed at the tail of its priority list.

- When a blocked process becomes runnable, it is placed at the tail of the list that contains processes of the same priority.
- If a process's priority changes (if the process issues the sched_setparam() system call) the process is placed at the tail of the new priority list. Note that the "new" priority could be the same as the old priority. Regardless, the process goes to the tail of the list.

The Round-Robin Scheduling Class

The round-robin class is designed primarily to support fixed-priority time-sharing processes instead of realtime processes. With one exception, the rules for the round-robin class are the same as the rules for the FIFO class. The exception is that the round-robin class includes the concept of a time slice, which promotes fair scheduling by helping prevent processes *of the same priority* from monopolizing a JP.

When a round-robin process exceeds its time slice, the dispatcher takes the process off the JP to allow other processes of the same priority to run. As the name "round robin" implies, the suspended process is placed at the tail of its priority list (Figure 5). The effect of a time-slice running out is the same as if a FIFO process volunteered to give up a JP with the sched_yield() system call.

In Figure 5, we've arbitrarily shown a priority range of 33 to 64. However, remember that the priorities of a round-robin class process can be higher than those of the FIFO class, or can overlap the FIFO class or an "others" class.

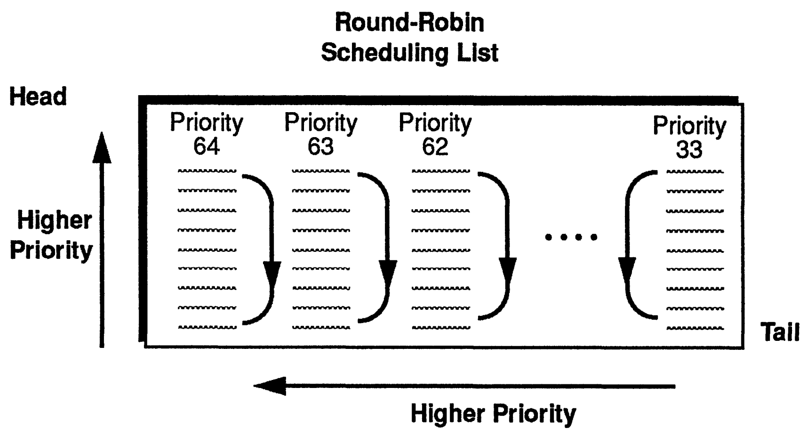


Figure 5 Round-Robin Scheduling List

The "Others" Scheduling Classes

For flexibility, P1003.4/D12 provides for implementation-defined scheduling classes, called "others." The draft does not specify what scheduling policies are implemented in this class; only that the class be defined. In DG/UX 5.4 RTE, the time-sharing algorithms of the standard DG/UX operating system and the special-purpose DG/UX scheduling policy are in the "others" class.

Time Sharing Class

Although the standard DG/UX 5.4 time sharing algorithms are round robin, there are some subtle differences between the DG/UX time-share class and the round-robin class described in the previous section. For example, the round-robin class processes have a fixed priority—the MTS does not change their priorities as it can for the DG/UX time-share class. And, round-robin processes have no MTS time slice; DG/UX time-share processes do.

DG/UX-FIFO Scheduling Class

The special-purpose DG/UX-FIFO scheduling class is provided for developers or users who require absolute control over the way processes are scheduled. Processes within this class can be assigned any priority. It's not likely that you'll need to use this class. For example, you could use the class for a process that periodically profiles all of the other processes in a system. Or, you could use the class for a process that must service a time-critical external device.

Important – The DG/UX-FIFO scheduling class provides unlimited flexibility in how you can assign priorities to processes. Because a DG/UX-FIFO scheduling-class process can be set to any priority, you should take care not to inadvertently set the priority of a process higher than that of critical kernel processes.

New Roles of the MTS and Dispatcher

Conceptually, the prioritized process lists for the three classes provide a second level of process scheduling, which feeds into the dispatcher's VP Eligible List (Figure 6 on page 13).

Depending on the rules of a scheduling class, the MTS may or may not be involved with changing a process's priority once the process is in the VP Eligible List. For example, the MTS will not change the priority of a FIFO or round-robin class process.

The dispatcher doesn't know about scheduling classes; it only needs to know a VP's priority and whether it should associate a time slice with the VP. When the MTS sends a runnable VP to the dispatcher, it also passes information about the VP, including whether the VP should have an on-JP time slice.

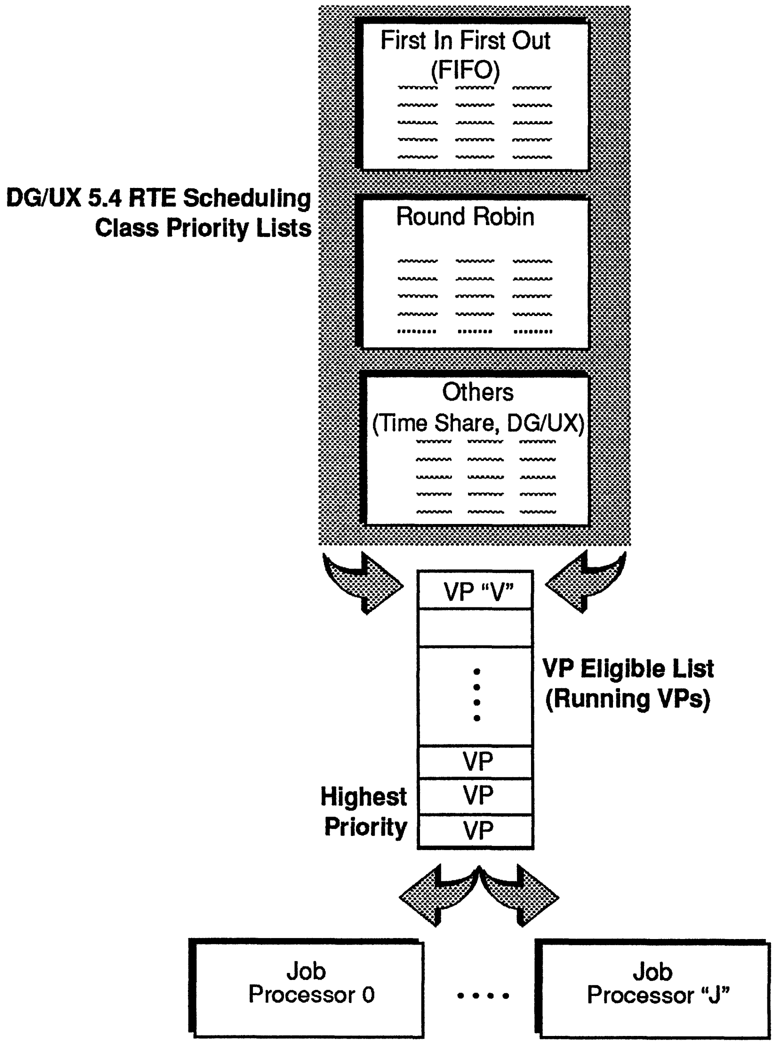

First In First Out (FIFO)
DG/UX 5.4 RTE Scheduling Class Priority Lists
Round Robin
Others (Time Share, DG/UX)
VP "V"
VP Eligible List (Running VPs)
VP
Highest Priority
VP
VP
Job Processor 0
Job Processor "J"

Figure 6 Scheduling DG/UX 5.4 RTE Classes

Bringing the Scheduling Classes Together

Figure 7 shows conceptually how the DG/UX 5.4 RTE operating system brings together processes from different scheduling classes. In this example, we've arbitrarily assigned the highest priorities to the FIFO class, the next highest to the round-robin class, and the lowest to the standard time-share class. Remember that this is only an example of how priorities can be assigned.

In this example, VP #21, with a priority of 92, is in the FIFO scheduling class. When VP #21 is loaded onto a JP, it will run to completion (unless it is blocked or volunteers to give up a JP). As soon as a JP is available, VP #84 will start executing. On a dual-JP system, both VP #21 and VP #84 would run at the same time. On a quad-JP system, all four VPs in the eligible list could run at the same time.

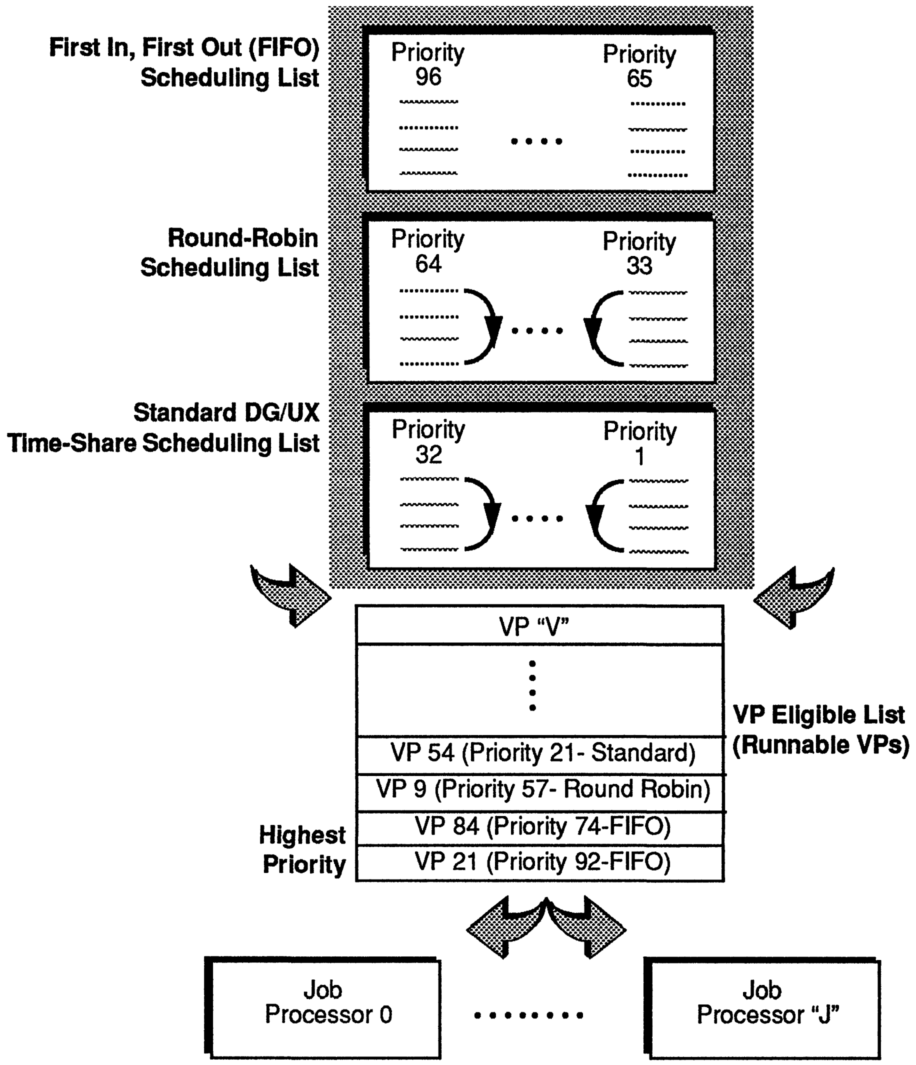

First In, First Out (FIFO) Scheduling List
Priority 96
Priority 65
Round-Robin Scheduling List
Priority 64
Priority 33
Standard DG/UX Time-Share Scheduling List
Priority 32
Priority 1
VP "V"
VP 54 (Priority 21- Standard)
VP 9 (Priority 57- Round Robin)
VP 84 (Priority 74-FIFO)
VP 21 (Priority 92-FIFO)
Highest Priority
VP Eligible List (Runnable VPs)
Job Processor 0
Job Processor "J"

Figure 7 Bringing the Scheduling Classes Together

Realtime System Calls

Table 1 summarizes the DG/UX 5.4 RTE realtime system calls for process scheduling. You can refer to P1003.4/D12 for more information about these functions.

Table 1 RTE Functions for Process Scheduling

Function	Description
sched_setparam()	Sets the priority and time-slice scheduling parameters of a process (within a class).
sched_getparam()	Gets the scheduling parameters of a process.
sched_setscheduler()	Sets the scheduling policy and parameters of a process.
sched_getscheduler()	Gets the scheduling policy of a process.
sched_yield()	Enables an existing process to give up a JP to an equal or higher priority VP.
sched_get_priority_max()	Gets the maximum setting of a scheduling policy's priority range. This is the highest (best) priority for that class.
sched_get_priority_min()	Gets the minimum setting (the lowest priority number) of a scheduling policy's priority range.
sched_rr_get_interval()	Gets the on-JP time limit (the time slice) of a round-robin process.

DG/UX 5.4 RTE Semaphores

The DG/UX operating system has always provided general-purpose counting semaphores as a way of controlling access to shared resources.[1] In addition to these general-purpose semaphores, DG/UX 5.4 RTE provides the counting semaphores that are specified in POSIX 1003.4/D12.

Semaphores are used to control access to a shared resource. An example of a shared resource is a data buffer into which a "producer" process writes data and from which a "consumer" process reads data.

If a realtime semaphore's value is positive, there are no processes blocked on the resource (or resources) that the semaphore protects. If a semaphore has a negative value, there are processes that are waiting for the resource (blocked on the resource's semaphore).

Semaphore System Calls

Table 2 summarizes the system calls that DG/UX 5.4 RTE provides for use with realtime semaphores.

Table 2 RTE Functions for Semaphores

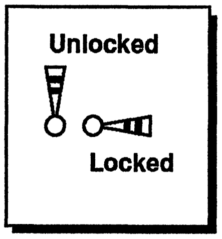
Function	Description
sem_init()	Initializes (opens) a semaphore and assigns to it a descriptor. Used in conjunction with sem_destroy().
sem_destroy()	Deallocates (closes) a semaphore. Used in conjunction with sem_init().
sem_unlink()	Unlinks (deletes) a semaphore when all of its initializations have been destroyed.
sem_lock()	Locks a semaphore. If the semaphore is already locked, the process waits on the semaphore.
sem_trylock()	Locks a semaphore only if it is not already locked (the process does not wait on the semaphore).
sem_unlock()	Unlocks a semaphore.

1. The DG/UX operating system's general purpose counting semaphores are based on the AT&T UNIX System V semaphore implementation. The use of general-purpose semaphores to control access to a shared resource is discussed in the Data General Technical Brief *Taking Advantage of Symmetric Multiprocessor Systems* (012-004177).

An Example Using Semaphores

Figure 8 on page 18 shows an example of how semaphores can be used to control access to shared data buffers. The example is of a closed-loop application that reads temperature data from a piece of production equipment, analyzes the temperature data, and uses the results to send commands to a valve that controls the temperature. The application has three processes, two shared data buffers, and a semaphore to protect each buffer.

Sensor Data Buffer and Semaphore



The sensor data buffer stores raw temperature data from a sensor in a piece of production equipment. The data collection process and the data analysis process share the sensor data buffer; the data collection process writes data to the buffer and the data analysis process reads data from the buffer.

When either the data collection or data analysis process wants to access the buffer, the process issues a sem_lock() call (or sem_trylock() call) to try to lock the buffer's semaphore.

Let's say that the data analysis process wants to read data from the sensor data buffer. The data analysis process issues a sem_lock() call. If the semaphore that protects the buffer is *not* locked, the data analysis process locks the semaphore and accesses the buffer. When the process is finished accessing the buffer, it unlocks the semaphore with the sem_unlock() call.

If the semaphore is already locked (because the data collection process is accessing the buffer), the data analysis process goes to sleep (the process waits on the semaphore) until the data collection process unlocks the semaphore.

Valve Data Buffer and Semaphore

To close the control loop in Figure 8, the data analysis process reads the raw temperature data from the sensor data buffer and manipulates the data. The process might, for example, determine how quickly the temperature is changing. The data analysis process writes this rate-of-change information to the valve data buffer.

The valve control process reads the control data from the buffer and uses the data to assemble control instructions for the valve. The semaphore that controls the valve data buffer works the same way as the one that controls the data buffer.

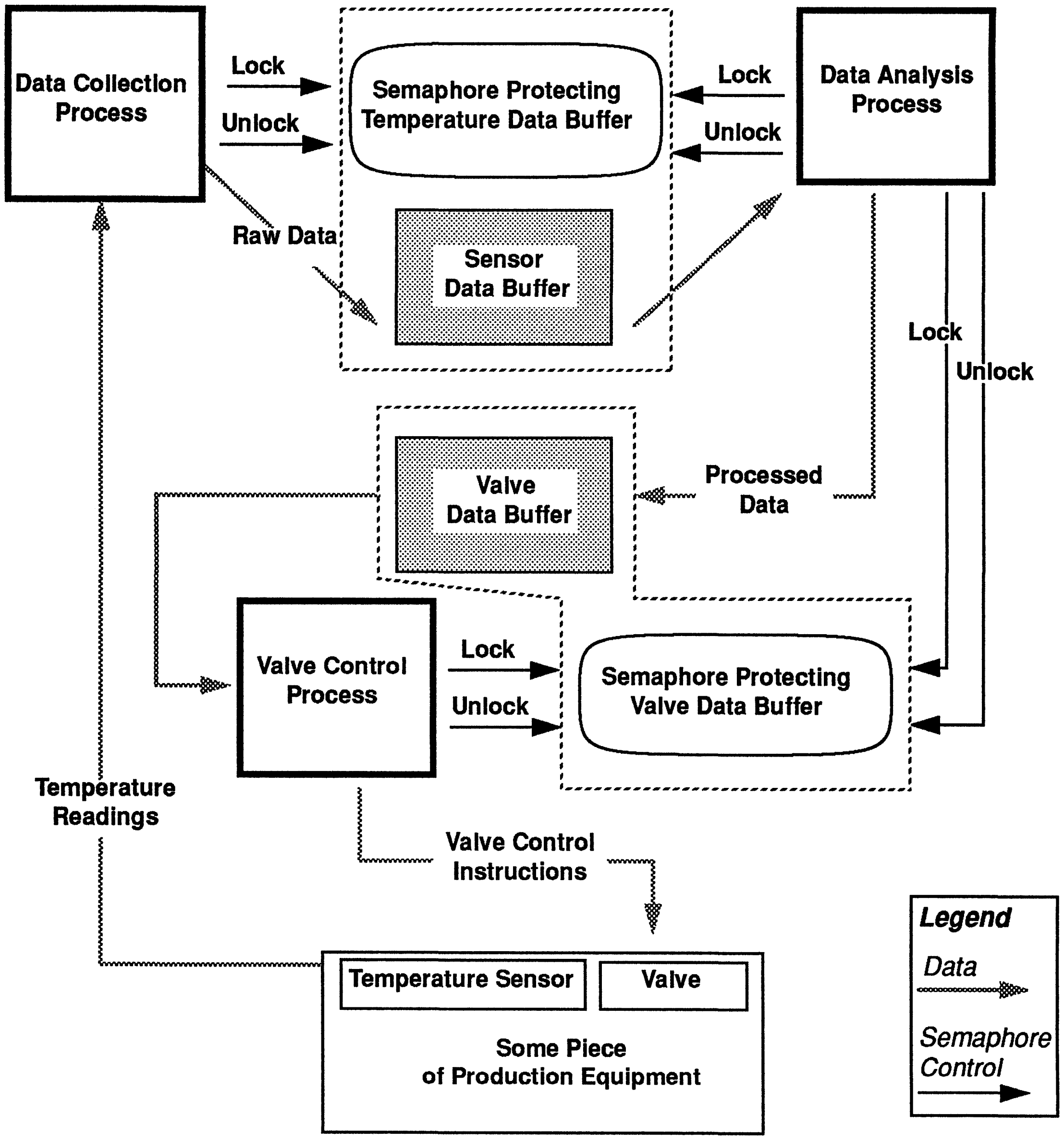


Figure 8 Using Semaphores to Synchronize Realtime Processes

DG/UX 5.4 RTE Timer Extensions

Timers are useful tools for synchronizing processes in realtime applications. The getitimer() and setitimer() system calls in the standard DG/UX operating system are often not sufficient for use in realtime applications because they support only one timer per process and may not provide enough time resolution.

DG/UX 5.4 RTE enables you to set as many as four timers[1] per process. And, you can specify the resolution of the timers in nanoseconds, rather than in microseconds as with the standard timers. However, the granularity of a system's hardware clocks are system-dependent, and some systems may not support nanosecond resolution. You can use the new clock_getres() system call to determine the clock resolution of a system.

There are two kinds of timers: one-shot and periodic (Figure 9). A one-shot timer runs to completion and generates a signal. At that point, a process can perform some operation, then reset the timer or delete it.

A periodic timer runs continuously (for the life of its process or until it is disarmed), and generates a signal at the end of each timing interval.

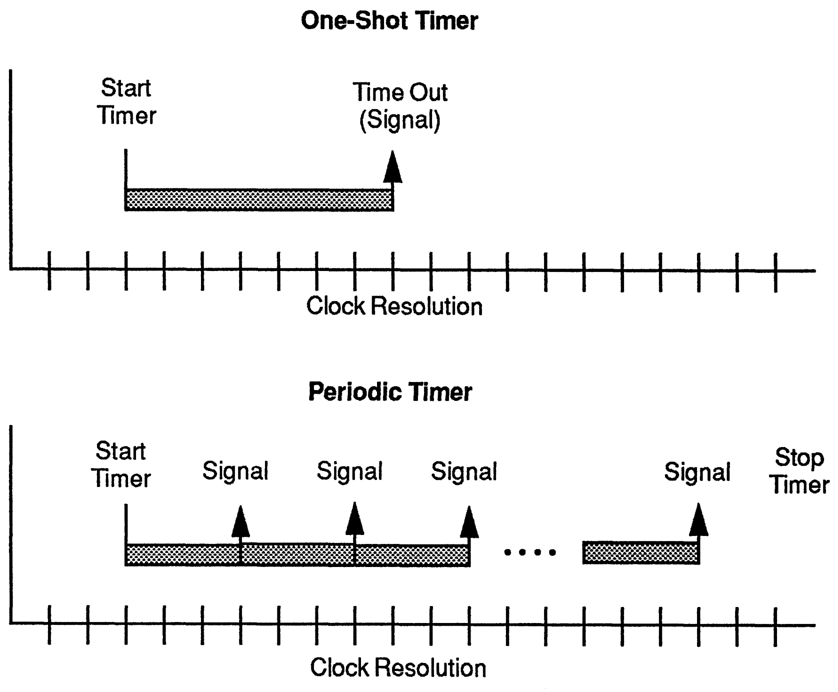


Figure 9   One-Shot and Periodic Timers

1. POSIX 1003.4/D12 specifies at least 32 timers per process. However, our current implementation provides four timers per process.

Timer System Calls

Table 3 summarizes the system calls that DG/UX 5.4 RTE provides for use with timers.

Table 3 RTE Functions for Timers

Function	Description
timer_create()	Creates a per-process timer using a specified clock and specifies a signal number to deliver.
timer_delete()	Deletes a timer.
timer_settime()	Sets a timer to run for a specified time; either one-shot or periodic.
timer_gettime()	Gets the time left before a timer expires.
timer_getoverrun()	Returns a count of the timer intervals that occurred between the time that the timer-expired signal was generated and the time that the signal was delivered.
clock_getres()	Gets the resolution of a specified clock.

An Example Using Timers

Figure 10 on page 21 shows how multiple periodic timers might be used in the sample application from the semaphore section.

In this example, the data collection process has two periodic timers: "slow," and "fast." In normal operation, the data collection process samples data from the sensor at the "slow" rate. If the data analysis process determines that the temperature is changing too quickly, it can tell the data collection process to start collecting temperature data at the "fast" rate of the other periodic timer.

The same approach could be used to create log files of the closed loop process. For example, the data analysis process could have multiple timers that could trigger a data logging process at different intervals. The intervals could become shorter as the temperature's rate-of-change increased.

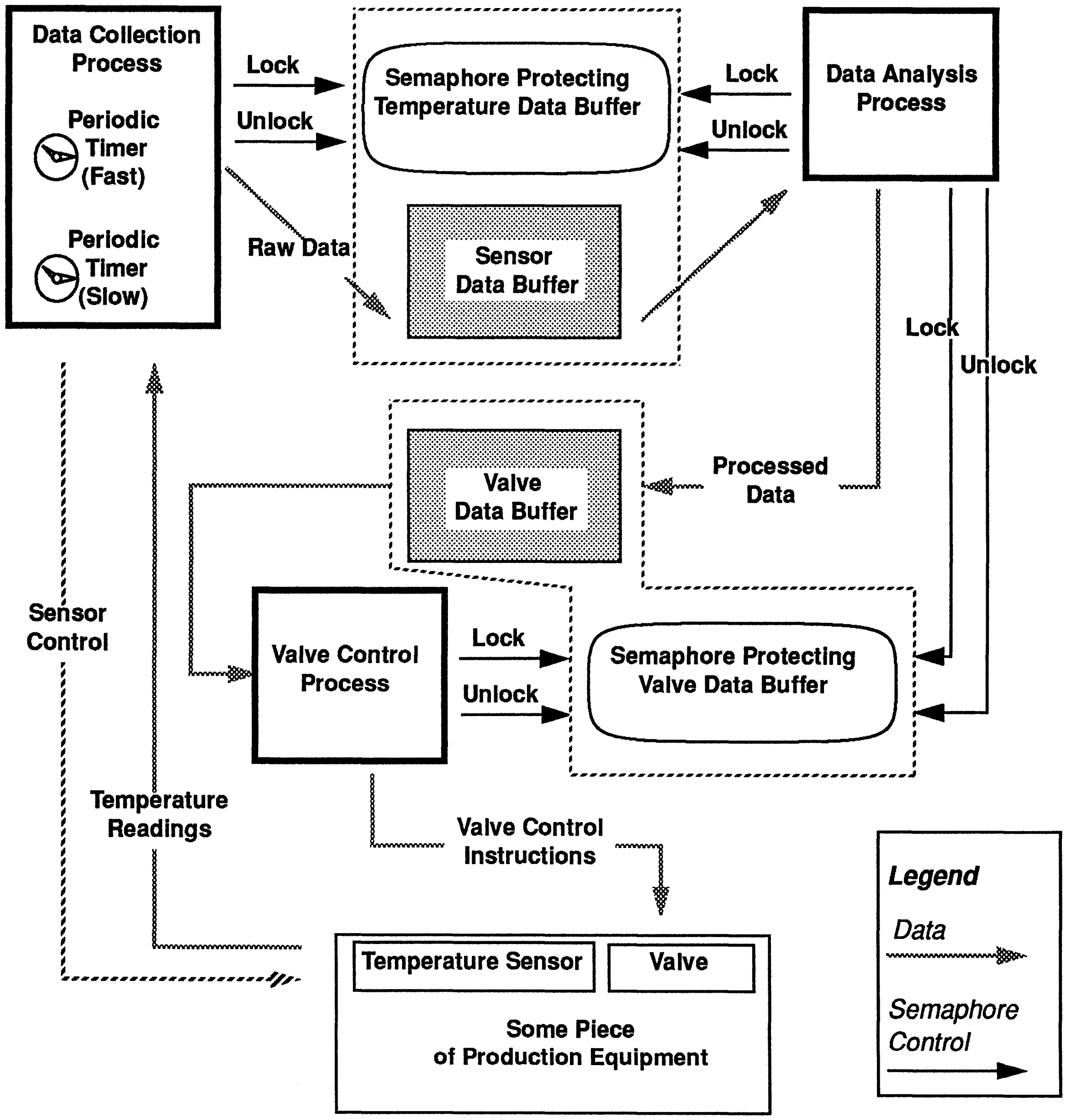


Figure 10 Using Periodic Timers

FYI—Comments About DG/UX 5.4 RTE and POSIX P1003.4 /D12

Data General is actively involved with formulating the POSIX P1003.4 standard and other POSIX standards, and we are committed to providing products that meet these standards. Therefore, the standard version of the DG/UX 5.4 operating system already provides many of the options that are described in POSIX P1003.4/D12. For example, standard DG/UX 5.4 already provides support for memory locking, memory mapping, and shared memory, which are part of POSIX P1003.4/D12.

POSIX P1003.4/D12 describes fifteen functional options and a performance measurement option. The functional options specify new system calls or specify changes to existing system calls. In general, the options are independent and can be implemented incrementally, as we have done with the process scheduling, process synchronization, and timer options in DG/UX 5.4 RTE.

Table 4 on page 23 summarizes the options that are described in POSIX 1003.4/D12 and shows to what level the options are implemented in the DG/UX 5.4 RTE operating system. The following list describes the icons that are used to describe the implementation levels.

■ *Full*	A complete implementation of the functions of an option that is described in POSIX 1003.4/D12. **Note**: Full implementation does not necessarily imply conformance to a draft option, because of differences in header files and types.
◪ *Partial*	Some of the system calls within an option are implemented as specified in the POSIX draft. An example is the Clock and Timer option, for which DG/UX 5.4 RTE implements the timer system calls, but not all of the clock system calls. In this category, some of the system calls that are not fully implemented may have an equivalent (next category).
▨ *Equivalent*	Some of the names or syntaxes of system calls within an option are different, but the result of using the call is the same for DG/UX 5.4 RTE and POSIX 1003.4/D12.
□ *None*	The option is not implemented in DG/UX 5.4 RTE and there are no equivalent functions.

Table 4 POSIX 1003.4D/12 and the DG/UX 5.4 RTE Implementations

POSIX 1003.4/D12 Options	P1003.4/D12 System Calls	Support in DG/UX 5.4 RTE	Comments
Counting semaphores	sem_init sem_destroy sem_unlink sem_lock sem_trylock sem_unlock	■ *Full*	Support for counting semaphores. Full POSIX draft implementation added to DG/UX 5.4 RTE. This is in addition to the traditional counting semaphores of standard DG/UX 5.4.
Process memory locking	mlockall munlockall	▨ *Equivalent*	Allows locking of all of the address space of a process. Equivalent to System V.4 functions of the same name.
Page memory mapping	mlock munlock	▨ *Equivalent*	Allows page-by-page locking of the address space of a process. Equivalent to System V.4 functions of the same name.
Memory mapped files	mmap munmap	▨ *Equivalent*	Adds file-system name space to provide names for shared memory segments, semaphores, and message queues. Equivalent to System V.4 functions of the same name.
Memory protection	mprotect msync	■ *Full*	Sets protection of memory mapping and synchronization of memory with physical storage. Fully implemented (already supported by standard DG/UX 5.4). Equivalent to System V.4 functions of the same name.

Table 4 POSIX 1003.4D/12 and the DG/UX 5.4 RTE Implementations (Continued)

POSIX 1003.4/D12 Options	P1003.4/D12 System Calls	Support in DG/UX 5.4 RTE	Comments
Shared memory objects	shm_open shm_unlink	▨ *Equivalent*	Allows adding shared memory segments. Similar to System V.4 except that the file system name space provides names for the shared segments. Standard DG/UX 5.4 and DG/UX 5.4 RTE provide equivalent System V.4 functionality (without file-system based names).
Process priority scheduling	sched_setparam sched_getparam sched_setscheduler sched_getscheduler sched_yield sched_get_priority_max sched_get_priority_min sched_get_rr_interval	■ *Full*	Support for additional realtime scheduling policies (FIFO and round robin). Full implementation added to DG/UX 5.4 RTE, in addition to standard time-share scheduling.
Realtime signals extension	sigwaitrt sigtimedwait sigqueue	□ *None*	Allows a process to define and wait for arbitrary, asynchronous events, and poll those events.
Clocks and Timers	clock_settime clock_gettime clock_getres timer_create timer_delete timer_settime timer_gettime timer_getoverrun nanosleep	◪ *Partial*	The five timer_ system calls and the clock_getres system call are included in DG/UX 5.4 RTE. The timer system calls support only four timers per process. The remaining clock_ and nanosleep system calls are not implemented.
Interprocess message passing	mq_open mq_close mq_send mq_receive mq_getattr mq_setattr mq_destroy mq_notify	▨ *Equivalent*	Defines and manipulates messages and message queues. DG/UX 5.4 and DG/UX 5.4 RTE support System V.3/V.4 message queues, which are similar to those of POSIX, but do not use the file system as the basis for their name space.

Table 4 POSIX 1003.4D/12 and the DG/UX 5.4 RTE Implementations (Continued)

POSIX 1003.4/D12 Options	P1003.4/D12 System Calls	Support in DG/UX 5.4 RTE	Comments
Synchronized files	fsync	▨ *Equivalent*	Provides a mechanism for enabling applications to ensure the integrity of file data and attributes, by specifying when the data and attributes are written to disk.
Synchronized I/O	fdatasync	▨ *Equivalent*	Similar to the standard DG/UX 5.4 fsync system call, but requires that file data (without attributes) be written to disk.
Prioritized I/O	N/A	□ *None*	Changes the semantics of several I/O system calls, such as aio_read and aio_write, to allow specification of the priority that is used to queue I/O requests.
Asynchronous I/O	aio_read aio_write lio_listio aio_error aio_return aio_cancel aio_suspend aio_fsync	▨ *Equivalent*	Permits starting, cancelling, and waiting for asynchronous I/O requests. Equivalent implementation in DG/UX 5.4 and DG/UX 5.4 RTE.
Realtime files	rf_create rf_getattr rf_setattr rf_getalloccap rf_getcachecap rf_getbiocap rf_getaiocap rf_getdiocap rf_getincr rf_getbuf rf_freebuf	▨ *Equivalent*	Allows the creation and manipulation of realtime files, or files with realtime attributes. Equivalent implementation in DG/UX 5.4 and DG/UX 5.4 RTE.
Performance metrics	N/A	□ *None*	Not implemented in DG/UX 5.4 RTE.

For More Information

The following articles and documents discuss realtime operating systems and process scheduling in more detail.

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

DG/UX™ Technical Brief: *Taking Advantage of Symmetric Multiprocessor Systems* (012-004177), June 17, 1992, Data General Corporation

IEEE Standards Project. *P1003.4 Draft 12. Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 1: Realtime Extension [C Language]*. February 1992

IEEE Standards Office
PO Box 1331
445 Hoes Lane
Piscataway, N.J. 08855-1331
(909) 562-3811