

The Philosophy of QNX Neutrino

- [Design goals](#)
- [Why QNX Neutrino for embedded systems?](#)
- [The microkernel](#)
- [Interprocess communication](#)
- [Network distribution of microkernels](#)

Design goals

The primary goal of QNX Neutrino is to deliver the open systems POSIX API in a robust, scalable form suitable for a wide range of systems -- from tiny, resource-constrained embedded systems to high-end distributed computing environments. The OS supports several processor families, including x86, ARM, Intel® XScale™ Microarchitecture, PowerPC, MIPS, and SH-4.

For mission-critical applications, a robust architecture is also fundamental, so the OS makes flexible and complete use of MMU hardware.

Of course, simply setting out these goals doesn't guarantee results. We invite you to read through this *System Architecture* guide to get a feel for our implementation approach and the design tradeoffs chosen to achieve these goals. When you reach the end of this guide, we think you'll agree that QNX Neutrino is the first OS product of its kind to truly deliver open systems standards, wide scalability, and high reliability.

An embeddable POSIX OS?

According to a prevailing myth, if you scratch a POSIX operating system, you'll find UNIX beneath the surface! A POSIX OS is therefore too large and unsuitable for embedded systems.

The fact, however, is that POSIX is *not* UNIX. Although the POSIX standards are rooted in existing UNIX practice, the POSIX working groups explicitly defined the standards in terms of "interface, *not* implementation."

Thanks to the precise specification within the standards, as well as the availability of POSIX test suites, nontraditional OS architectures can provide a POSIX API without adopting the traditional UNIX kernel. Compare any two POSIX systems and they'll *look* very much alike -- they'll have many of the same functions, utilities, etc. But when it comes to performance or reliability, they may be as different as night and day. Architecture makes the difference.

Despite its decidedly non-UNIX architecture, QNX Neutrino implements the standard POSIX API. By adopting a microkernel architecture, the OS delivers this API in a form easily scaled down for realtime embedded systems or incrementally scaled up as required.

Product scaling

Since you can readily scale a microkernel OS simply by including or omitting the particular processes that provide the functionality required, you can use a single microkernel OS for a much wider range of applications than a realtime executive.

Product development often takes the form of creating a "product line," with successive models providing greater functionality. Rather than be forced to change operating systems for each version of the product, developers using a microkernel OS can easily scale the system as needed -- by adding filesystems, networking, graphical user interfaces, and other technologies.

Some of the advantages to this scalable approach include:

- portable application code (between product-line members)
- common tools used to develop the entire product line
- portable skill sets of development staff
- reduced time-to-market.

Why POSIX for embedded systems?

A common problem with realtime application development is that each realtime OS tends to come equipped with its own proprietary API. In the absence of industry standards, this isn't an unusual state for a competitive marketplace to evolve into, since surveys of the realtime marketplace regularly show heavy use of inhouse proprietary operating systems. POSIX represents a chance to unify this marketplace.

Among the many POSIX standards, those of most interest to embedded systems developers are:

- *1003.1* -- defines the API for process management, device I/O, filesystem I/O, and basic IPC. This encompasses what might be described as the base functionality of a UNIX OS, serving as a useful standard for many applications. From a C-language programming perspective, ANSI X3J11 C is assumed as a starting point, and then the various aspects of managing processes, files, and tty devices are detailed beyond what ANSI C specifies.
- *Realtime Extensions* -- defines a set of realtime extensions to the base 1003.1 standard. These extensions consist of semaphores, prioritized process scheduling, realtime extensions to signals, high-resolution timer control, enhanced IPC primitives, synchronous and asynchronous I/O, and a recommendation for realtime contiguous file support.
- *Threads* -- further extends the POSIX environment to include the creation and management of multiple threads of execution within a given address space.
- *Additional Realtime Extensions* -- defines further extensions to the realtime standard. Facilities such as attaching interrupt handlers are described.
- *Application Environment Profiles* -- defines several AEPs (*Realtime AEP*, *Embedded Systems AEP*, etc.) of the POSIX environment to suit different embedded capability sets. These profiles represent embedded OSs with/without filesystems and other capabilities.



For an up-to-date status of the many POSIX drafts/standards documents, see the PASC (Portable Applications Standards Committee of the IEEE Computer Society) report at <http://pasc.opengroup.org/standing/sd11.html>.

Apart from any "bandwagon" motive for adopting industry standards, there are several specific advantages to applying the POSIX standard to the embedded realtime marketplace.

Multiple OS sources

Hardware manufacturers are loath to choose a single-sourced hardware component because of the risks implied if that source discontinues production. For the same reason, manufacturers shouldn't be tied to a single-sourced, proprietary OS simply because their application source code isn't portable to other OSs.

By building applications to the POSIX standards, developers can use OSs from multiple vendors. Application source code can be readily ported from platform to platform and from OS to OS, provided that developers avoid using OS-specific extensions.

Portability of development staff

Using a common API for embedded development, programmers experienced with one realtime OS can directly apply their skill sets to other projects involving other processors and operating systems. In addition,

programmers with UNIX or POSIX experience can easily work on embedded realtime systems, since the nonrealtime portion of the realtime OS's API is already familiar territory.

Development environment: native and cross development

With the addition of interface hardware similar to the target runtime system, a workstation running a POSIX OS can become a functional superset of the embedded system. As a result, the application can be conveniently developed on the self-hosted desktop system.

Even in a cross-hosted development environment, the API remains essentially the same. Regardless of the particular host (QNX Neutrino, Solaris, Windows,...) or the target (x86, ARM, MIPS, PowerPC,...), the programmer doesn't need to worry about platform-specific endian, alignment, or I/O issues.

Why QNX Neutrino for embedded systems?

The main responsibility of an operating system is to manage a computer's resources. All activities in the system -- scheduling application programs, writing files to disk, sending data across a network, and so on -- should function together as seamlessly and transparently as possible.

Some environments call for more rigorous resource management and scheduling than others. Realtime applications, for instance, depend on the OS to handle multiple events and to ensure that the system responds to those events within predictable time limits. The more responsive the OS, the more "time" a realtime application has to meet its deadlines.

QNX Neutrino is ideal for *embedded realtime applications*. It can be scaled to very small sizes and provides multitasking, threads, priority-driven preemptive scheduling, and fast context-switching -- all essential ingredients of an embedded realtime system. Moreover, the OS delivers these capabilities with a POSIX-standard API; there's no need to forgo standards in order to achieve a small system.

QNX Neutrino is also remarkably flexible. Developers can easily customize the OS to meet the needs of their applications. From a "bare-bones" configuration of a microkernel with a few small modules to a full-blown network-wide system equipped to serve hundreds of users, you're free to set up your system to use only those resources you require to tackle the job at hand.

QNX Neutrino achieves its unique degree of efficiency, modularity, and simplicity through two fundamental principles:

- microkernel architecture
- message-based interprocess communication

Microkernel architecture

Buzzwords often fall in and out of fashion. Vendors tend to enthusiastically apply the buzzwords of the day to their products, whether the terms actually fit or not.

The term "microkernel" has become fashionable. Although many new operating systems are said to be "microkernels" (or even "nanokernels"), the term may not mean very much without a clear definition.

Let's try to define the term. A microkernel OS is structured as a tiny kernel that provides the minimal services used by a team of optional cooperating processes, which in turn provide the higher-level OS functionality. The microkernel itself lacks filesystems and many other services normally expected of an OS -- those services are provided by optional processes.

The real goal in designing a microkernel OS is not simply to "make it small." A microkernel OS embodies a fundamental change in the approach to delivering OS functionality. *Modularity is the key, size is but a side effect.* To call any kernel a "microkernel" simply because it happens to be small would miss the point entirely.

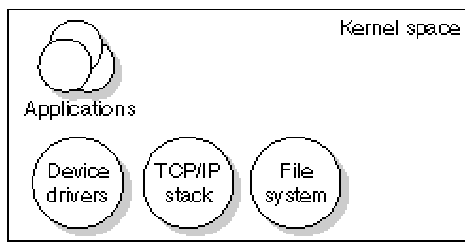
Since the IPC services provided by the microkernel are used to "glue" the OS itself together, the performance and flexibility of those services govern the performance of the resulting OS. With the exception of those IPC

services, a microkernel is roughly comparable to a realtime executive, both in terms of the services provided and in their realtime performance.

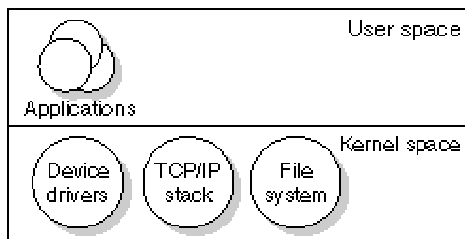
The microkernel differs from an executive in how the IPC services are used to extend the functionality of the kernel with additional, service-providing processes. Since the OS is implemented as a team of cooperating processes managed by the microkernel, user-written processes can serve both as applications and as processes that extend the underlying OS functionality for industry-specific applications. The OS itself becomes "open" and easily extensible. Moreover, user-written extensions to the OS won't affect the fundamental reliability of the core OS.

A difficulty for many realtime executives implementing the POSIX 1003.1 standard is that their runtime environment is typically a single-process, multiple-threaded model, with unprotected memory between threads. Such an environment is only a subset of the multi-process model that POSIX assumes; it cannot support the *fork()* function. In contrast, QNX Neutrino fully utilizes an MMU to deliver the complete POSIX process model in a protected environment.

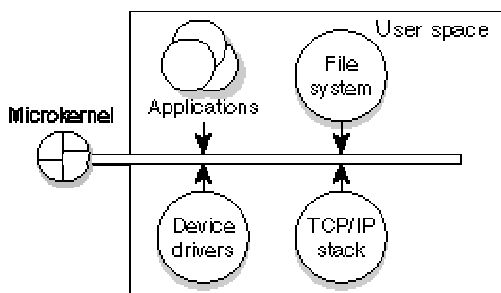
As the following diagrams show, a true microkernel offers *complete memory protection*, not only for user applications, but also for OS components (device drivers, filesystems, etc.):



Conventional executives offer no memory protection.



In a monolithic OS, system processes have no protection.

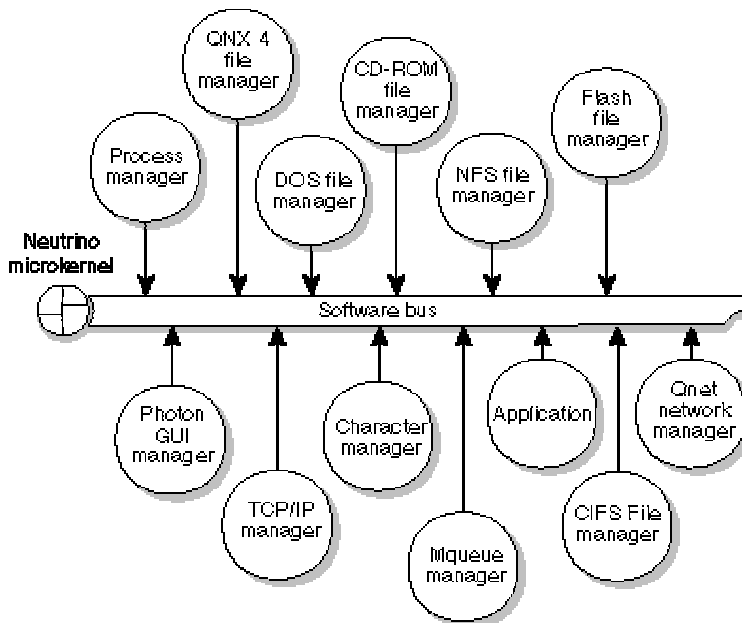


A microkernel provides complete memory protection.

The first version of the QNX OS was shipped in 1981. With each successive product revision, we have applied the experience from previous product generations to the latest incarnation: QNX Neutrino, our most capable, scalable OS to date. We believe that this time-tested experience is what enables the QNX Neutrino OS to deliver the functionality it does using the limited resources it consumes.

The OS as a team of processes

The QNX Neutrino OS consists of a small microkernel managing a group of cooperating processes. As the following illustration shows, the structure looks more like a team than a hierarchy, as several "players" of equal rank interact with each other through the coordinating kernel.



The QNX Neutrino architecture.

QNX Neutrino acts as a kind of "software bus" that lets you dynamically plug in/out OS modules whenever they're needed.

A true kernel

The *kernel* is the heart of any operating system. In some systems, the "kernel" comprises so many functions that for all intents and purposes it *is* the entire operating system!

But our microkernel is truly a kernel. First of all, like the kernel of a realtime executive, it's very small. Secondly, it's dedicated to only a few fundamental services:

- **thread services** via POSIX thread-creation primitives
- **signal services** via POSIX signal primitives
- **message-passing services** -- the microkernel handles the routing of all messages between all threads throughout the entire system.
- **synchronization services** via POSIX thread-synchronization primitives.

- **scheduling services** -- the microkernel schedules threads for execution using the various POSIX realtime scheduling algorithms.
- **timer services** -- the microkernel provides the rich set of POSIX timer services.
- **process management services** -- the microkernel and the process manager together form a unit (called `procnto`). The process manager portion is responsible for managing processes, memory, and the pathname space.

Unlike threads, the microkernel itself is never scheduled for execution. The processor executes code in the microkernel only as the result of an explicit kernel call, an exception, or in response to a hardware interrupt.

System processes

All OS services, except those provided by the mandatory microkernel/process manager module (`procnto`), are handled via *standard processes*. A richly configured system could include the following:

- filesystem managers
- character device managers
- graphical user interface (Photon)
- native network manager
- TCP/IP

System processes vs user-written processes

System processes are essentially indistinguishable from any user-written program -- they use the same public API and kernel services available to any (suitably privileged) user process.

It is this architecture that gives QNX Neutrino unparalleled extensibility. Since most OS services are provided by standard system processes, it's very simple to augment the OS itself: just write new programs to provide new OS services.

In fact, the boundary between the operating system and the application can become very blurred. The only real difference between system services and applications is that OS services manage resources for clients.

Suppose you've written a database server -- how should such a process be classified?

Just as a filesystem accepts requests (via messages) to open files and read or write data, so too would a database server. While the requests to the database server may be more sophisticated, both servers are very much the same in that they provide an API (implemented by messages) that clients use to access a resource. Both are independent processes that can be written by an end-user and started and stopped on an as-needed basis.

A database server might be considered a system process at one installation, and an application at another. *It really doesn't matter!* The important point is that the OS allows such processes to be implemented cleanly, with no need for modifications to the standard components of the OS itself. For developers creating custom embedded systems, this provides the flexibility to extend the OS in directions that are uniquely useful to their applications, without needing access to OS source code.

Device drivers

Device drivers allow the OS and application programs to make use of the underlying hardware in a generic way (e.g. a disk drive, a network interface). While most OSs require device drivers to be tightly bound into the OS itself, device drivers for QNX Neutrino can be started and stopped as standard processes. As a result, adding device drivers doesn't affect any other part of the OS -- drivers can be developed and debugged like any other application.

Interprocess communication

When several threads run concurrently, as in typical realtime multitasking environments, the OS must provide mechanisms to allow them to communicate with each other.

Interprocess communication (IPC) is the key to designing an application as a set of cooperating processes in which each process handles one well-defined part of the whole.

The OS provides a simple but powerful set of IPC capabilities that greatly simplify the job of developing applications made up of cooperating processes.

QNX Neutrino as a message-passing operating system

QNX was the first commercial operating system of its kind to make use of message passing as the fundamental means of IPC. The OS owes much of its power, simplicity, and elegance to the complete integration of the message-passing method throughout the entire system.

In QNX Neutrino, a message is a parcel of bytes passed from one process to another. The OS attaches no special meaning to the content of a message -- the data in a message has meaning for the sender of the message and for its receiver, but for no one else.

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several processes. As they send, receive, and reply to messages, processes undergo various "changes of state" that affect when, and for how long, they may run. Knowing their states and priorities, the microkernel can schedule all processes as efficiently as possible to make the most of available CPU resources. This single, consistent method -- message-passing -- is thus constantly operative throughout the entire system.

Realtime and other mission-critical applications generally require a dependable form of IPC, because the processes that make up such applications are so strongly interrelated. The discipline imposed by QNX Neutrino's message-passing design helps bring order and greater reliability to applications.

Network distribution of kernels

In its simplest form, local area networking provides a mechanism for sharing files and peripheral devices among several interconnected computers. QNX Neutrino goes far beyond this simple concept and integrates the entire network into a single, homogeneous set of resources.

Any thread on any machine in the network can directly make use of any resource on any other machine. From the application's perspective, there's no difference between a local or remote resource -- no special facilities need to be built into applications to allow them to make use of remote resources.

Users may access files anywhere on the network, take advantage of any peripheral device, and run applications on any machine on the network (provided they have the appropriate authority). Processes can communicate in the same manner anywhere throughout the entire network. Again, the OS's all-pervasive message-passing IPC accounts for such fluid, transparent networking.

Single-computer model

QNX Neutrino is designed from the ground up as a network-wide operating system. In some ways, a native QNX Neutrino network feels more like a mainframe computer than a set of individual micros. Users are simply aware of a large set of resources available for use by any application. But unlike a mainframe, QNX Neutrino provides a highly responsive environment, since the appropriate amount of computing power can be made available at each node to meet the needs of each user.

In a mission-critical environment, for example, applications that control realtime I/O devices may require more performance than other, less critical, applications, such as a web browser. The network is responsive enough to support both types of applications *at the same time* -- the OS lets you focus computing power on the devices in your hard realtime system where and when it's needed, without sacrificing concurrent connectivity to the

desktop. Moreover, critical aspects of realtime computing, such as priority inheritance, function seamlessly across a QNX Neutrino network, regardless of the physical media employed (switch fabric, serial, etc.).

Flexible networking

QNX Neutrino networks can be put together using various hardware and industry-standard protocols. Since these are completely transparent to application programs and users, new network architectures can be introduced at any time without disturbing the OS.

Each node in the network is assigned a unique name that becomes its identifier. This name is the only visible means to determine whether the OS is running as a network or as a standalone operating system.

This degree of transparency is yet another example of the distinctive power of QNX Neutrino's message-passing architecture. In many systems, important functions such as networking, IPC, or even message passing are built on top of the OS, rather than integrated directly into its core. The result is often an awkward, inefficient "double standard" interface, whereby communication between processes is one thing, while penetrating the private interface of a mysterious monolithic kernel is another matter altogether.

In contrast to monolithic systems, QNX Neutrino is grounded on the principle that effective communication is the key to effective operation. Message passing thus forms the cornerstone of our microkernel architecture and enhances the efficiency of *all* transactions among all processes throughout the entire system, whether across a PC backplane or across a mile of twisted pair.

The QNX Neutrino Microkernel

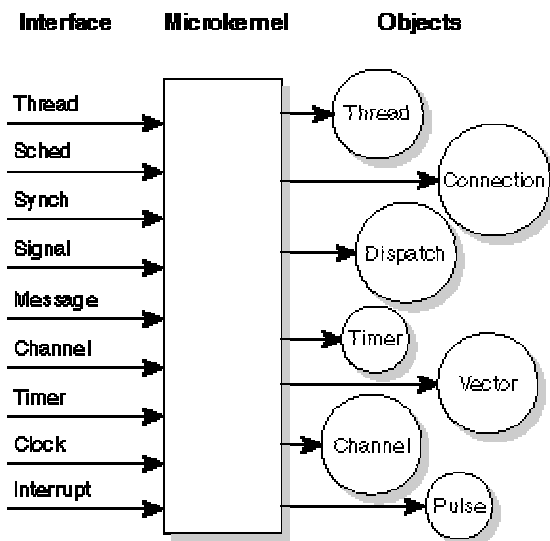
- [Introduction](#)
- [The implementation of QNX Neutrino](#)
- [System services](#)
- [Threads and processes](#)
- [Thread scheduling](#)
- [Synchronization services](#)
- [QNX Neutrino IPC](#)
- [Clock and timer services](#)
- [Interrupt handling](#)

Introduction

The QNX Neutrino microkernel implements the core POSIX features used in embedded realtime systems, along with the fundamental QNX Neutrino message-passing services. The POSIX features that aren't implemented in the microkernel (file and device I/O, for example) are provided by optional processes and shared libraries.

Successive QNX microkernels have seen a reduction in the code required to implement a given kernel call. The object definitions at the lowest layer in the kernel code have become more specific, allowing greater code reuse (such as folding various forms of POSIX signals, realtime signals, and QNX pulses into common data structures and code to manipulate those structures).

At its lowest level, the microkernel contains a few fundamental objects and the highly tuned routines that manipulate them. The OS is built from this foundation.



The QNX Neutrino microkernel.

Some developers have assumed that our microkernel is implemented entirely in assembly code for size or performance reasons. In fact, our implementation is coded primarily in C; size and performance goals are achieved through successively refined algorithms and data structures, rather than via assembly-level peep-hole optimizations.

The implementation of QNX Neutrino

Historically, the "application pressure" on QNX operating systems has been from both ends of the computing spectrum -- from memory-limited embedded systems all the way up to high-end SMP (Symmetrical Multi-Processing) machines with gigabytes of physical memory. Accordingly, the design goals for QNX Neutrino accommodate both seemingly exclusive sets of functionality. Pursuing these goals is intended to extend the reach of systems well beyond what other OS implementations could address.

POSIX realtime and thread extensions

Since QNX Neutrino implements the majority of the realtime and thread services directly in the microkernel, these services are available even without the presence of additional OS modules.

In addition, some of the profiles defined by POSIX suggest that these services be present without necessarily requiring a process model. In order to accommodate this, the OS provides direct support for threads, but relies on its process manager portion to extend this functionality to processes containing multiple threads.

Note that many realtime executives and kernels provide only a nonmemory-protected threaded model, with no process model and/or protected memory model at all. Without a process model, full POSIX compliance cannot be achieved.

System services

The QNX Neutrino microkernel has kernel calls to support the following:

- threads
- message passing
- signals
- clocks
- timers
- interrupt handlers
- semaphores
- mutual exclusion locks (mutexes)
- condition variables (condvars)
- barriers.

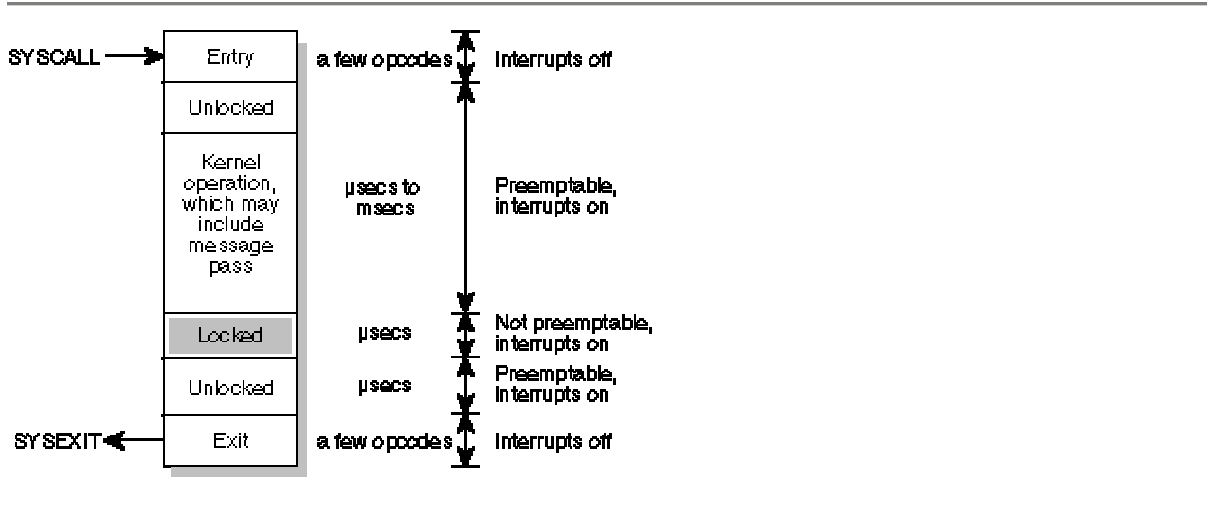
The entire OS is built upon these calls. The OS is fully preemptable, even while passing messages between processes; it resumes the message pass where it left off before preemption.

The minimal complexity of the microkernel helps place an upper bound on the longest nonpreemptable code path through the kernel, while the small code size makes addressing complex multiprocessor issues a tractable problem. Services were chosen for inclusion in the microkernel on the basis of having a short execution path.

Operations requiring significant work (e.g. process loading) were assigned to external processes/threads, where the effort to enter the context of that thread would be insignificant compared to the work done within the thread to service the request.

Rigorous application of this rule to dividing the functionality between the kernel and external processes destroys the myth that a microkernel OS must incur higher runtime overhead than a monolithic kernel OS. Given the work done between context switches (implicit in a message pass), and the very quick context-switch times that result from the simplified kernel, the time spent performing context switches becomes "lost in the noise" of the work done to service the requests communicated by the message passing between the processes that make up the OS.

The following diagram shows the preemption details for the non-SMP kernel (x86 implementation).



QNX Neutrino preemption details.

Interrupts are disabled, or preemption is held off, for only very brief intervals (typically in the order of hundreds of nanoseconds).

Threads and processes

When building an application (realtime, embedded, graphical, or otherwise), the developer may want several algorithms within the application to execute concurrently. This concurrency is achieved by using the POSIX thread model, which defines a process as containing one or more threads of execution.

A thread can be thought of as the minimum "unit of execution," the unit of scheduling and execution in the microkernel. A process, on the other hand, can be thought of as a "container" for threads, defining the "address space" within which threads will execute. A process will always contain at least one thread.

Depending on the nature of the application, threads might execute independently with no need to communicate between the algorithms (unlikely), or they may need to be tightly coupled, with high-bandwidth communications and tight synchronization. To assist in this communication and synchronization, QNX Neutrino provides a rich variety of IPC and synchronization services.

The following *pthread*s (POSIX Threads) library calls don't involve any microkernel thread calls:

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_attr_getinheritsched()`

- *pthread_attr_getschedparam()*
- *pthread_attr_getschedpolicy()*
- *pthread_attr_getscope()*
- *pthread_attr_getstackaddr()*
- *pthread_attr_getstacksize()*
- *pthread_attr_init()*
- *pthread_attr_setdetachstate()*
- *pthread_attr_setinheritsched()*
- *pthread_attr_setschedparam()*
- *pthread_attr_setschedpolicy()*
- *pthread_attr_setscope()*
- *pthread_attr_setstackaddr()*
- *pthread_attr_setstacksize()*
- *pthread_cleanup_pop()*
- *pthread_cleanup_push()*
- *pthread_equal()*
- *pthread_getspecific()*
- *pthread_setspecific()*
- *pthread_testcancel()*
- *pthread_key_create()*
- *pthread_key_delete()*
- *pthread_once()*
- *pthread_self()*
- *pthread_setcancelstate()*
- *pthread_setcanceltype()*

The following table lists the POSIX thread calls that have a corresponding microkernel thread call, allowing you to choose either interface:

POSIX call	Microkernel call	Description
<i>pthread_create()</i>	<i>ThreadCreate()</i>	Create a new thread of execution.
<i>pthread_exit()</i>	<i>ThreadDestroy()</i>	Destroy a thread.
<i>pthread_detach()</i>	<i>ThreadDetach()</i>	Detach a thread so it doesn't need to be joined.
<i>pthread_join()</i>	<i>ThreadJoin()</i>	Join a thread waiting for its exit status.
<i>pthread_cancel()</i>	<i>ThreadCancel()</i>	Cancel a thread at the next cancellation point.
N/A	<i>ThreadCtl()</i>	Change a thread's Neutrino-specific thread characteristics.
<i>pthread_mutex_init()</i>	<i>SyncTypeCreate()</i>	Create a mutex.
<i>pthread_mutex_destroy()</i>	<i>SyncDestroy()</i>	Destroy a mutex.
<i>pthread_mutex_lock()</i>	<i>SyncMutexLock()</i>	Lock a mutex.
<i>pthread_mutex_trylock()</i>	<i>SyncMutexLock()</i>	Conditionally lock a mutex.
<i>pthread_mutex_unlock()</i>	<i>SyncMutexUnlock()</i>	Unlock a mutex.
<i>pthread_cond_init()</i>	<i>SyncTypeCreate()</i>	Create a condition variable.
<i>pthread_cond_destroy()</i>	<i>SyncDestroy()</i>	Destroy a condition variable.
<i>pthread_cond_wait()</i>	<i>SyncCondvarWait()</i>	Wait on a condition variable.
<i>pthread_cond_signal()</i>	<i>SyncCondvarSignal()</i>	Signal a condition variable.
<i>pthread_cond_broadcast()</i>	<i>SyncCondvarSignal()</i>	Broadcast a condition variable.
<i>pthread_getschedparam()</i>	<i>SchedGet()</i>	Get scheduling parameters and policy of thread.
<i>pthread_setschedparam()</i>	<i>SchedSet()</i>	Set scheduling parameters and policy of thread.
<i>pthread_sigmask()</i>	<i>SignalProcMask()</i>	Examine or set a thread's signal mask.
<i>pthread_kill()</i>	<i>SignalKill()</i>	Send a signal to a specific thread.

The OS can be configured to provide a mix of threads and processes (as defined by POSIX). Each process is MMU-protected from each other, and each process may contain one or more threads that share the process's address space.

The environment you choose affects not only the concurrency capabilities of the application, but also the IPC and synchronization services the application might make use of.



Even though the common term "IPC" refers to communicating processes, we use it here to describe the communication between *threads*, whether they're within the same process or separate processes.

Thread attributes

Although threads within a process share everything within the process's address space, each thread still has some "private" data. In some cases, this private data is protected within the kernel (e.g. the *tid* or thread ID), while other private data resides unprotected in the process's address space (e.g. each thread has a stack for its own use). Some of the more noteworthy thread-private resources are:

tid

Each thread is identified by an integer thread ID, starting at 1. The *tid* is unique within the thread's process.

register set

Each thread has its own instruction pointer (IP), stack pointer (SP), and other processor-specific register context.

stack

Each thread executes on its own stack, stored within the address space of its process.

signal mask

Each thread has its own signal mask.

thread local storage

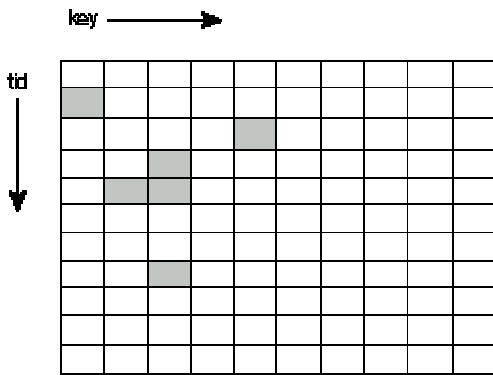
A thread has a system-defined data area called "thread local storage" (TLS). The TLS is used to store "per-thread" information (such as *tid*, *pid*, stack base, *errno*, and thread-specific key/data bindings). The TLS doesn't need to be accessed directly by a user application. A thread can have user-defined data associated with a thread-specific data key.

cancellation handlers

Callback functions that are executed when the thread terminates.

Thread-specific data, implemented in the *pthread* library and stored in the TLS, provides a mechanism for associating a process global integer key with a unique per-thread data value. To use thread-specific data, you first create a new key and then bind a unique data value to the key (per thread). The data value may, for example, be an integer or a pointer to a dynamically allocated data structure. Subsequently, the key can return the bound data value per thread.

A typical application of thread-specific data is for a thread-safe function that needs to maintain a context for each calling thread.



*Sparse matrix (*tid*, *key*) to value mapping.*

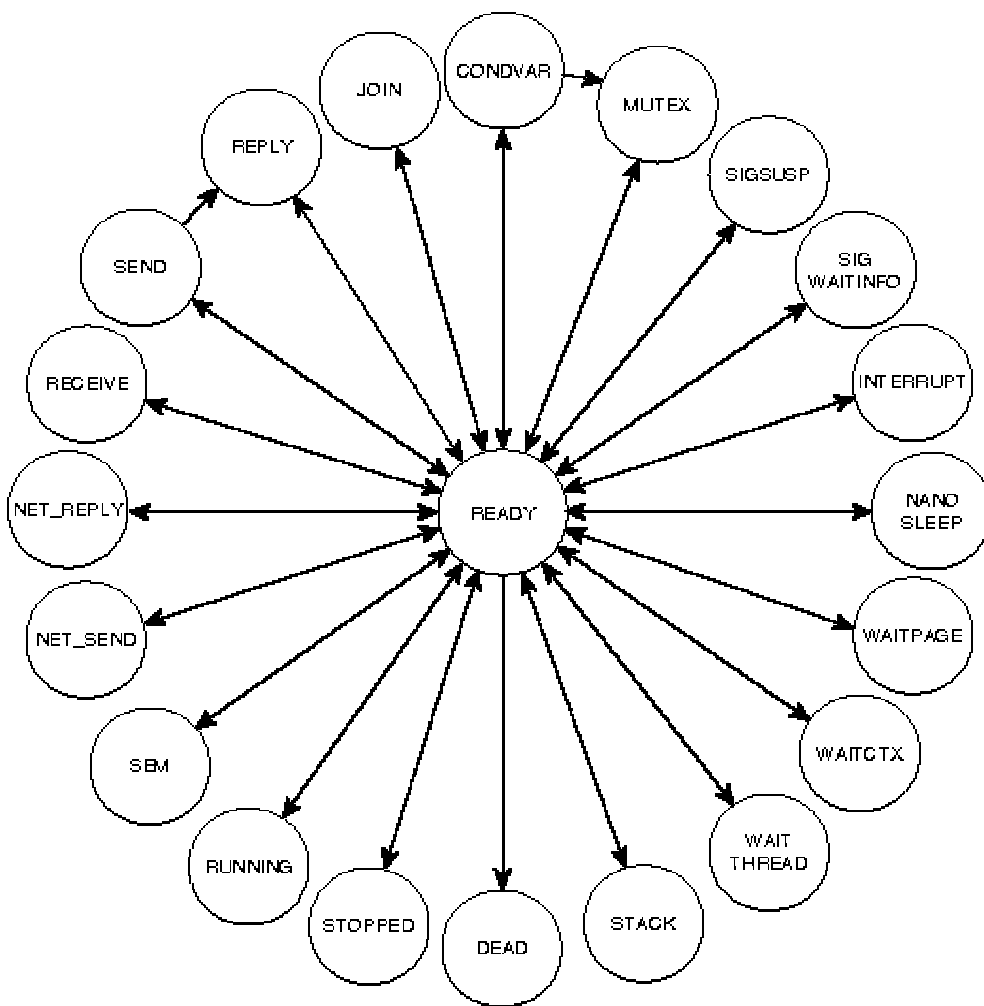
You use the following functions to create and manipulate this data:

Function	Description
<i>pthread_key_create()</i>	Create a data key with destructor function
<i>pthread_key_delete()</i>	Destroy a data key
<i>pthread_setspecific()</i>	Bind a data value to a data key
<i>pthread_getspecific()</i>	Return the data value bound to a data key

Thread life cycle

The number of threads within a process can vary widely, with threads being created and destroyed dynamically. Thread creation (*pthread_create()*) involves allocating and initializing the necessary resources within the process's address space (e.g. thread stack) and starting the execution of the thread at some function in the address space.

Thread termination (*pthread_exit()*, *pthread_cancel()*) involves stopping the thread and reclaiming the thread's resources. As a thread executes, its state can generally be described as either "ready" or "blocked." More specifically, it can be one of the following:



Possible thread states.

CONDVAR

The thread is blocked on a condition variable (e.g. it called *pthread_condvar_wait()*).

DEAD

The thread has terminated and is waiting for a join by another thread.

INTERRUPT

The thread is blocked waiting for an interrupt (i.e. it called *InterruptWait()*).

JOIN

The thread is blocked waiting to join another thread (e.g. it called *pthread_join()*).

MUTEX

The thread is blocked on a mutual exclusion lock (e.g. it called *pthread_mutex_lock()*).

NANOSLEEP

The thread is sleeping for a short time interval (e.g. it called *nanosleep()*).

NET_REPLY

The thread is waiting for a reply to be delivered across the network (i.e. it called *MsgReply*()*).

NET_SEND

The thread is waiting for a pulse or signal to be delivered across the network (i.e. it called *MsgSendPulse()*, *MsgDeliverEvent()*, or *SignalKill()*).

READY

The thread is waiting to be executed while the processor executes another thread of equal or higher priority.

RECEIVE

The thread is blocked on a message receive (e.g. it called *MsgReceive()*).

REPLY

The thread is blocked on a message reply (i.e. it called *MsgSend()*, and the server received the message).

RUNNING

The thread is being executed by a processor.

SEM

The thread is waiting for a semaphore to be posted (i.e. it called *SyncSemWait()*).

SEND

The thread is blocked on a message send (e.g. it called *MsgSend()*, but the server hasn't yet received the message).

SIGSUSPEND

The thread is blocked waiting for a signal (i.e. it called *sigsuspend()*).

SIGWAITINFO

The thread is blocked waiting for a signal (i.e. it called *sigwaitinfo()*).

STACK

The thread is waiting for the virtual address space to be allocated for the thread's stack (parent will have called *ThreadCreate()*).

STOPPED

The thread is blocked waiting for a SIGCONT signal.

WAITCTX

The thread is waiting for a noninteger (e.g. floating point) context to become available for use.

WAITPAGE

The thread is waiting for physical memory to be allocated for a virtual address.

WAITTHREAD

The thread is waiting for a child thread to finish creating itself (i.e. it called *ThreadCreate()*).

Thread scheduling

When scheduling decisions are made

The execution of a running thread is temporarily suspended whenever the microkernel is entered as the result of a kernel call, exception, or hardware interrupt. A scheduling decision is made whenever the execution state of any thread changes -- it doesn't matter which processes the threads might reside within. Threads are scheduled globally across all processes.

Normally, the execution of the suspended thread will resume, but the scheduler will perform a context switch from one thread to another whenever the running thread:

- is blocked
- is preempted
- yields.

When thread is blocked

The running thread blocks when it must wait for some event to occur (response to an IPC request, wait on a mutex, etc.). The blocked thread is removed from the ready queue and the highest-priority ready thread is then run. When the blocked thread is subsequently unblocked, it is placed on the end of the ready queue for that priority level.

When thread is preempted

The running thread is preempted when a higher-priority thread is placed on the ready queue (it becomes READY, as the result of its block condition being resolved). The preempted thread remains at the beginning of the ready queue for that priority and the higher-priority thread runs.

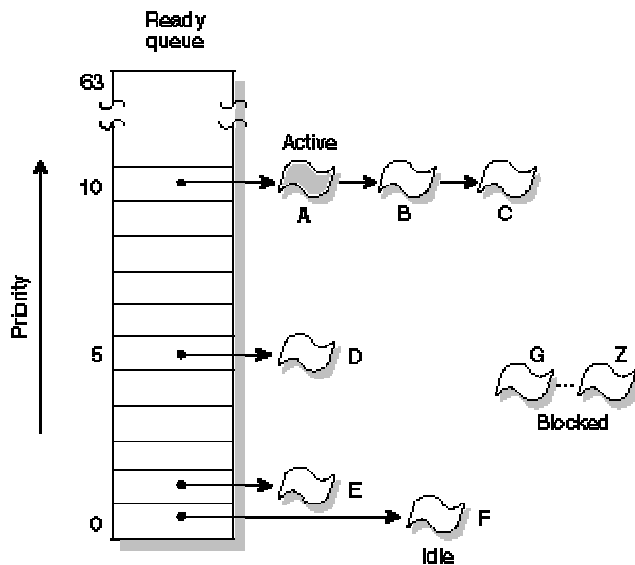
When thread yields

The running thread voluntarily yields the processor (*sched_yield()*) and is placed on the end of the ready queue for that priority. The highest-priority thread then runs (which may still be the thread that just yielded).

Scheduling priority

Every thread is assigned a priority. The scheduler selects the next thread to run by looking at the priority assigned to every thread that is READY (i.e. capable of using the CPU). The thread with the highest priority is selected to run.

The following diagram shows the ready queue for six threads (A-F) that are READY. All other threads (G-Z) are BLOCKED. Thread A is currently running. Thread A, B, and C are at the highest priority, so they'll share the processor based on the running thread's scheduling algorithm.



The ready queue.

Each thread can have a scheduling priority ranging from 1 to 63 (the highest priority), *independent of the scheduling policy*. The special *idle* thread (in the process manager) has priority 0 and is always ready to run. A thread inherits the priority of its parent thread by default.

Note that in order to prevent *priority inversion*, the kernel may temporarily boost a thread's priority.

The threads on the ready queue are ordered by priority. The ready queue is actually implemented as 64 separate queues, one for each priority. Threads are queued in FIFO order in the queue of their priority. The first thread in the highest-priority queue is selected to run.

Scheduling algorithms

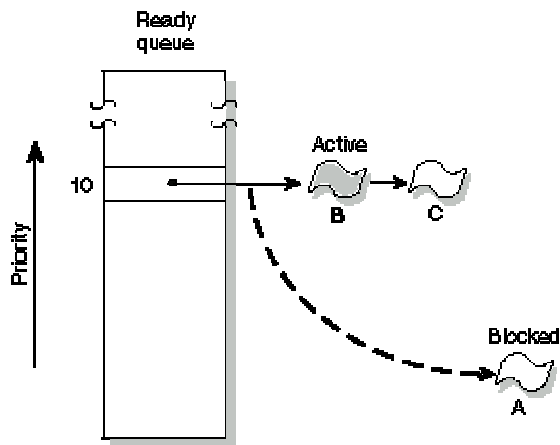
To meet the needs of various applications, QNX Neutrino provides these scheduling algorithms:

- FIFO scheduling
- round-robin scheduling
- sporadic scheduling.

Each thread in the system may run using any method. The methods are effective on a per-thread basis, not on a global basis for all threads and processes on a node.

Remember that the FIFO and round-robin scheduling algorithms apply only when two or more threads that share the *same priority* are READY (i.e. the threads are directly competing with each other). The sporadic method, however, employs a "budget" for a thread's execution. In all cases, if a higher-priority thread becomes READY, it immediately preempts all lower-priority threads.

In the following diagram, three threads of equal priority are READY. If Thread A blocks, Thread B will run.



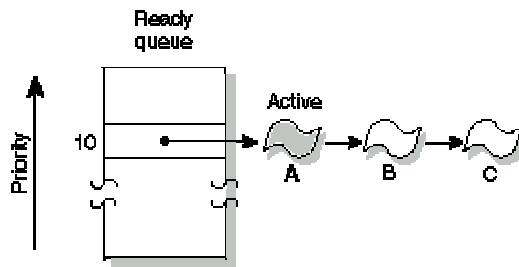
Thread A blocks, Thread B runs.

Although a thread inherits its scheduling algorithm from its parent process, the thread can request to change the algorithm applied by the kernel.

FIFO scheduling

In FIFO scheduling, a thread selected to run continues executing until it:

- voluntarily relinquishes control (e.g. it blocks)
- is preempted by a higher-priority thread.



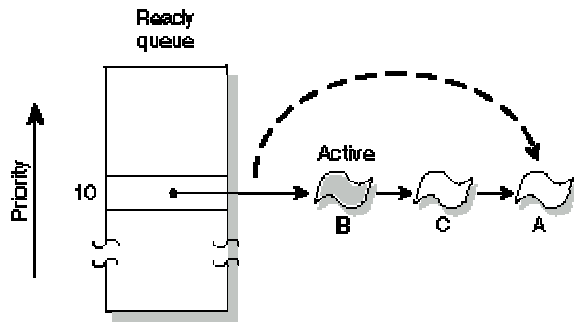
FIFO scheduling.

Round-robin scheduling

In round-robin scheduling, a thread selected to run continues executing until it:


- voluntarily relinquishes control
- is preempted by a higher-priority thread
- consumes its *timeslice*.

As the following diagram shows, Thread A ran until it consumed its timeslice; the next READY thread (Thread B) now runs:



Round-robin scheduling.

A timeslice is the unit of time assigned to every process. Once it consumes its timeslice, a thread is preempted and the next READY thread at the same priority level is given control. A timeslice is $4 * \text{the clock period}$. (For more information, see the entry for [*ClockPeriod\(\)*](#) in the *Library Reference*.)

 Apart from time slicing, round-robin scheduling is identical to FIFO scheduling.

Sporadic scheduling

The sporadic scheduling algorithm is generally used to provide a capped limit on the execution time of a thread *within a given period of time*. This behavior is essential when Rate Monotonic Analysis (RMA) is being performed on a system that services both periodic and aperiodic events. Essentially, this algorithm allows a thread to service aperiodic events without jeopardizing the hard deadlines of other threads or processes in the system.

As in FIFO scheduling, a thread using sporadic scheduling continues executing until it blocks or is preempted by a higher-priority thread. And as in adaptive scheduling, a thread using sporadic scheduling will drop in priority, but with sporadic scheduling you have much more precise control over the thread's behavior.

Under sporadic scheduling, a thread's priority can oscillate dynamically between a *foreground* or normal priority and a *background* or low priority. Using the following parameters, you can control the conditions of this sporadic shift:

Initial budget (C)

The amount of time a thread is allowed to execute at its normal priority (N) before being dropped to its low priority (L).

Low priority (L)


The priority level to which the thread will drop. The thread executes at this lower priority (L) while in the background, and runs at normal priority (N) while in the foreground.

Replenishment period (T)

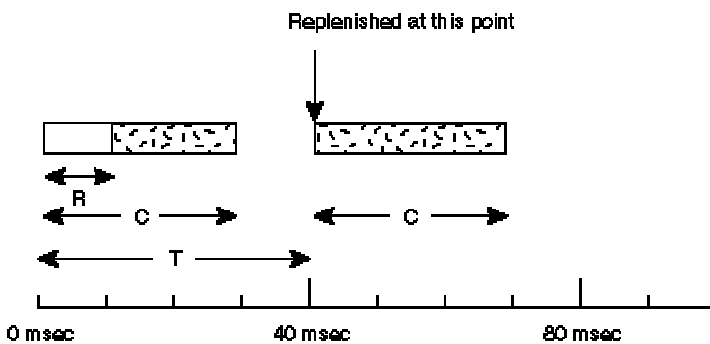
The period of time during which a thread is allowed to consume its execution budget. To schedule replenishment operations, the POSIX implementation also uses this value as the offset from the time the thread becomes READY.

Max number of pending replenishments

This value limits the number of replenishment operations that can take place, thereby bounding the amount of system overhead consumed by the sporadic scheduling policy.

 In a mis-configured system, a thread's execution budget may become eroded because of too much blocking -- i.e. it won't receive enough replenishments.

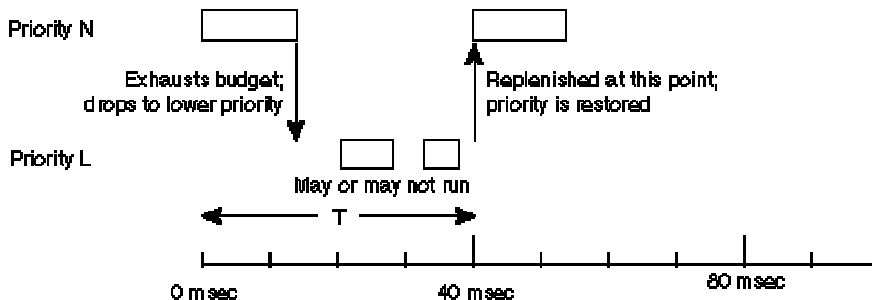
As the following diagram shows, the sporadic scheduling policy establishes a thread's initial execution budget (C), which is consumed by the thread as it runs and is replenished periodically (for the amount T). When a thread blocks, the amount of the execution budget that's been consumed (R) is arranged to be replenished at some later time (e.g. at 40 msec) after the thread first became ready to run.



A thread's budget is replenished periodically.

At its normal priority N, a thread will execute for the amount of time defined by its initial execution budget C. As soon as this time is exhausted, the priority of the thread will drop to its low priority L until the replenishment operation occurs.

Assume, for example, a system where the thread never blocks or is never preempted:

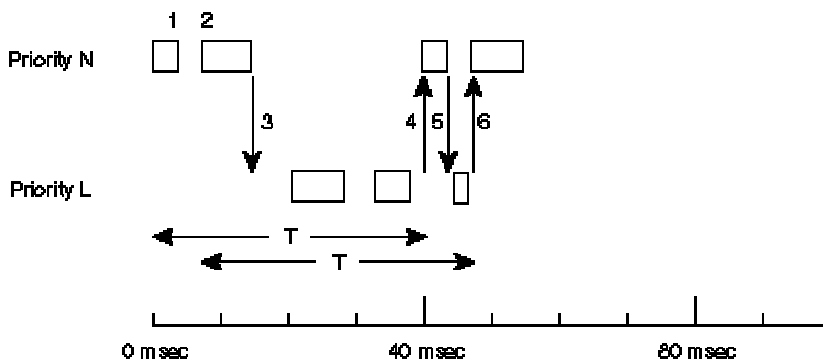


Thread drops in priority until its budget is replenished.

Here the thread will drop to its low-priority (background) level, where it may or may not get a chance to run depending on the priority of other threads in the system.

Once the replenishment occurs, the thread's priority is raised to its original level. This guarantees that within a properly configured system, the thread will be given the opportunity *every period T* to run for a maximum execution time C. This ensures that a thread running at priority N will consume only C/T percent of the system's resources.

When a thread blocks multiple times, then several replenishment operations may be started and occur at different times. This could mean that the thread's execution budget will total C within a period T; however, the execution budget may not be contiguous during that period.



Thread oscillates between high and low priority.

In the diagram above, the thread has a budget (C) of 10 msec to be consumed within each 40-msec replenishment period (T).

1. The initial run of the thread is blocked after 3 msec, so a replenishment operation of 3 msec is scheduled to begin at the 40-msec mark, i.e. when its first replenishment period has elapsed.
2. The thread gets an opportunity to run again at 6 msec, which marks the start of another replenishment period (T). The thread still has 7 msec remaining in its budget.
3. The thread runs without blocking for 7 msec, thereby exhausting its budget, and then drops to low priority L, where it may or may not be able to execute. A replenishment of 7 msec is scheduled to occur at 46 msec (40 + 6), i.e. when T has elapsed.
4. The thread has 3 msec of its budget replenished at 40 msec (see Step 1) and is therefore boosted back to its normal priority.
5. The thread consumes the 3 msec of its budget and then is dropped back to the low priority.
6. The thread has 7 msec of its budget replenished at 46 msec (see Step 3) and is boosted back to its normal priority.

And so on. The thread will continue to oscillate between its two priority levels, servicing aperiodic events in your system in a controlled, predictable manner.

Manipulating priority and scheduling algorithms

A thread's priority can vary during its execution, either from direct manipulation by the thread itself or from the kernel adjusting the thread's priority as it receives a message from a higher-priority thread.

In addition to priority, you can also select the scheduling algorithm that the kernel will use for the thread. Here are the POSIX calls for performing these manipulations, along with the microkernel calls used by these library routines:

POSIX call	Microkernel call	Description
<code>sched_getparam()</code>	<code>SchedGet()</code>	Get scheduling priority
<code>sched_setparam()</code>	<code>SchedSet()</code>	Set scheduling priority
<code>sched_getscheduler()</code>	<code>SchedGet()</code>	Get scheduling policy
<code>sched_setscheduler()</code>	<code>SchedSet()</code>	Set scheduling policy

IPC issues

Since all the threads in a process have unhindered access to the shared data space, wouldn't this execution model "trivially" solve all of our IPC problems? Can't we just communicate the data through shared memory and dispense with any other execution models and IPC mechanisms?

If only it were that simple!

One issue is that the access of individual threads to common data must be *synchronized*. Having one thread read inconsistent data because another thread is part way through modifying it is a recipe for disaster. For example, if one thread is updating a linked list, no other threads can be allowed to traverse or modify the list until the first thread has finished. A code passage that must execute "serially" (i.e. by only one thread at a time) in this manner is termed a "critical section." The program would fail (intermittently, depending on how frequently a "collision" occurred) with irreparably damaged links unless some synchronization mechanism ensured serial access.

Mutexes, semaphores, and condvars are examples of synchronization tools that can be used to address this problem. These tools are described later in this section.

Although synchronization services can be used to allow threads to cooperate, shared memory per se can't address a number of IPC issues. For example, although threads can communicate through the common data space, this works only if all the threads communicating are within a single process. What if our application needs to communicate a query to a database server? We need to pass the details of our query to the database server, but the thread we need to communicate with lies *within* a database server process and the address space of that server isn't addressable to us.

The OS takes care of the network-distributed IPC issue because the one interface -- message passing -- operates in both the local and network-remote cases, and can be used to access all OS services. Since messages can be exactly sized, and since most messages tend to be quite tiny (e.g. the error status on a write request, or a tiny read request), the data moved around the network can be far less with message passing than with network-distributed shared memory, which would tend to copy 4K pages around.

Thread complexity issues

Although threads are very appropriate for some system designs, it's important to respect the Pandora's box of complexities their use unleashes. In some ways, it's ironic that while MMU-protected multitasking has become common, computing fashion has made popular the use of multiple threads in an unprotected address space. This not only makes debugging difficult, but also hampers the generation of reliable code.

Threads were initially introduced to UNIX systems as a "light-weight" concurrency mechanism to address the problem of slow context switches between "heavy weight" processes. Although this is a worthwhile goal, an obvious question arises: Why are process-to-process context switches slow in the first place?

Architecturally, the OS addresses the context-switch performance issue first. In fact, threads and processes provide nearly identical context-switch performance numbers. QNX Neutrino's process-switch times are faster than UNIX thread-switch times. As a result, QNX Neutrino threads don't need to be used to solve the IPC performance problem; instead, they're a tool for achieving greater concurrency within application and server processes.

Without resorting to threads, fast process-to-process context switching makes it reasonable to structure an application as a team of cooperating processes sharing an explicitly allocated shared-memory region. An application thus exposes itself to bugs in the cooperating processes only so far as the effects of those bugs on the contents of the shared-memory region. The private memory of the process is still protected from the other processes. In the purely threaded model, the private data of all threads (including their stacks) is openly accessible, vulnerable to stray pointer errors in any thread in the process.

Nevertheless, threads can also provide concurrency advantages that a pure process model cannot address. For example, a filesystem server process that executes requests on behalf of many clients (where each request takes significant time to complete), definitely benefits from having multiple threads of execution. If one client process requests a block from disk, while another client requests a block already in cache, the filesystem process can utilize a pool of threads to concurrently service client requests, rather than remain "busy" until the disk block is read for the first request.

As requests arrive, each thread is able to respond directly from the buffer cache or to block and wait for disk I/O without increasing the response latency seen by other client processes. The filesystem server can "precreate" a team of threads, ready to respond in turn to client requests as they arrive. Although this complicates the architecture of the filesystem manager, the gains in concurrency are significant.

Synchronization services

QNX Neutrino provides the POSIX-standard thread-level synchronization primitives, some of which are useful even between threads in different processes. The synchronization services include at least the following:

Synchronization service	Supported between processes	Supported across a QNX LAN
Mutexes	Yes	No
Condvars	Yes	No
Barriers	No	No
Sleepon locks	No	No
Reader/writer locks	Yes	No
Semaphores	Yes	Yes (named only)
FIFO scheduling	Yes	No
Send/Receive/Reply	Yes	Yes
Atomic operations	Yes	No

The above synchronization primitives are implemented directly by the kernel, except for:



- barriers, sleep-on locks, and reader/writer locks (which are built from mutexes and condvars)
- atomic operations (which are either implemented directly by the processor or emulated in the kernel).

Mutual exclusion locks

Mutual exclusion locks, or mutexes, are the simplest of the synchronization services. A mutex is used to ensure exclusive access to data shared between threads. It is typically acquired (`pthread_mutex_lock()`) and released (`pthread_mutex_unlock()`) around the code that accesses the shared data (usually a critical section).

Only one thread may have the mutex locked at any given time. Threads attempting to lock an already locked mutex will block until the thread is later unlocked. When the thread unlocks the mutex, the highest-priority thread waiting to lock the mutex will unblock and become the new owner of the mutex. In this way, threads will sequence through a critical region in priority-order.

On most processors, acquisition of a mutex doesn't require entry to the kernel for a free mutex. What allows this is the use of the compare-and-swap opcode on x86 processors and the load/store conditional opcodes on most RISC processors.

Entry to the kernel is done at acquisition time only if the mutex is already held so that the thread can go on a blocked list; kernel entry is done on exit if other threads are waiting to be unblocked on that mutex. This allows acquisition and release of an uncontested critical section or resource to be very quick, incurring work by the OS only to resolve contention.

A nonblocking lock function (`pthread_mutex_trylock()`) can be used to test whether the mutex is currently locked or not. For best performance, the execution time of the critical section should be small and of bounded duration. A condvar should be used if the thread may block within the critical section.

Priority inheritance

If a thread with a higher priority than the mutex owner attempts to lock a mutex, then the effective priority of the current owner will be increased to that of the higher-priority blocked thread waiting for the mutex. The owner will return to its real priority when it unlocks the mutex. This scheme not only ensures that the higher-priority thread will be blocked waiting for the mutex for the shortest possible time, but also solves the classic priority-inversion problem.

The attributes of the mutex can also be modified (using `pthread_mutex_setrecursive()`) to allow a mutex to be recursively locked by the same thread. This can be useful to allow a thread to call a routine that might attempt to lock a mutex that the thread already happens to have locked.



Recursive mutexes are *non-POSIX* services -- they don't work with condvars.

Condition variables

A condition variable, or condvar, is used to block a thread within a critical section until some condition is satisfied. The condition can be arbitrarily complex and is independent of the condvar. However, the condvar must always be used with a mutex lock in order to implement a monitor.

A condvar supports three operations:

- wait (*pthread_cond_wait()*)
- signal (*pthread_cond_signal()*)
- broadcast (*pthread_cond_broadcast()*).



Note that there's no connection between a condvar signal and a POSIX signal.

Here's a typical example of how a condvar can be used:

```
pthread_mutex_lock( &m );  
  
. . .  
while (!arbitrary_condition) {  
    pthread_cond_wait( &cv, &m );  
}  
  
. . .  
pthread_mutex_unlock( &m );
```

In this code sample, the mutex is acquired before the condition is tested. This ensures that only this thread has access to the arbitrary condition being examined. While the condition is true, the code sample will block on the wait call until some other thread performs a signal or broadcast on the condvar.

The while loop is required for two reasons. First of all, POSIX cannot guarantee that false wakeups will not occur (e.g. multi-processor systems). Second, when another thread has made a modification to the condition, we need to retest to ensure that the modification matches our criteria. The associated mutex is unlocked atomically by *pthread_cond_wait()* when the waiting thread is blocked to allow another thread to enter the critical section.

A thread that performs a signal will unblock the highest-priority thread queued on the condvar, while a broadcast will unblock all threads queued on the condvar. The associated mutex is locked atomically by the highest-priority unblocked thread; the thread must then unlock the mutex after proceeding through the critical section.

A version of the condvar wait operation allows a timeout to be specified (*pthread_cond_timedwait()*). The waiting thread can then be unblocked when the timeout expires.

Barriers

A barrier is a synchronization mechanism that lets you "corral" several cooperating threads (e.g. in a matrix computation), forcing them to wait at a specific point until all have finished before any one thread can continue.

Unlike the *pthread_join()* function, where you'd wait for the threads to terminate, in the case of a barrier you're waiting for the threads to *rendezvous* at a certain point. When the specified number of threads arrive at the barrier, we unblock *all of them* so they can continue to run.

You first create a barrier with *pthread_barrier_init()*:

```
#include <pthread.h>  
  
int  
pthread_barrier_init (pthread_barrier_t *barrier,  
                     const pthread_barrierattr_t *attr,
```

```
    unsigned int count);
```

This creates a barrier object at the passed address (a pointer to the barrier object is in *barrier*), with the attributes as specified by *attr*. The *count* member holds the number of threads that must call *pthread_barrier_wait()*.

Once the barrier is created, each thread will call *pthread_barrier_wait()* to indicate that it has completed:

```
#include <pthread.h>
```

```
int pthread_barrier_wait (pthread_barrier_t *barrier);
```

When a thread calls *pthread_barrier_wait()*, it blocks until the number of threads specified initially in the *pthread_barrier_init()* function have called *pthread_barrier_wait()* (and blocked also). When the correct number of threads have called *pthread_barrier_wait()*, all those threads will unblock *at the same time*.

Here's an example:

```
/*
 * barrier1.c
 */

#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <sys/neutrino.h>

pthread_barrier_t  barrier; // barrier synchronization object

main () // ignore arguments
{
    time_t  now;

    // create a barrier object with a count of 3
    pthread_barrier_init (&barrier, NULL, 3);

    // start up two threads, thread1 and thread2
    pthread_create (NULL, NULL, thread1, NULL);
    pthread_create (NULL, NULL, thread2, NULL);

    // at this point, thread1 and thread2 are running

    // now wait for completion
    time (&now);
    printf ("main() waiting for barrier at %s", ctime (&now));
    pthread_barrier_wait (&barrier);

    // after this point, all three threads have completed.
    time (&now);
```

```

    printf ("barrier in main() done at %s", ctime (&now));
}

void *
thread1 (void *not_used)
{
    time_t  now;

    time (&now);
    printf ("thread1 starting at %s", ctime (&now));

    // do the computation
    // let's just do a sleep here...
    sleep (20);
    pthread_barrier_wait (&barrier);
    // after this point, all three threads have completed.
    time (&now);
    printf ("barrier in thread1() done at %s", ctime (&now));
}

void *
thread2 (void *not_used)
{
    time_t  now;

    time (&now);
    printf ("thread2 starting at %s", ctime (&now));

    // do the computation
    // let's just do a sleep here...
    sleep (40);
    pthread_barrier_wait (&barrier);
    // after this point, all three threads have completed.
    time (&now);
    printf ("barrier in thread2() done at %s", ctime (&now));
}

```

The main thread created the barrier object and initialized it with a count of the total number of threads that must be synchronized to the barrier before the threads may carry on. In the example above, we used a count of 3: one for the *main()* thread, one for *thread1()*, and one for *thread2()*.

Then we start *thread1()* and *thread2()*. To simplify this example, we have the threads sleep to cause a delay, as if computations were occurring. To synchronize, the main thread simply blocks itself on the barrier, knowing that the barrier will unblock only after the two worker threads have joined it as well.

In this release, the following barrier functions are included:

Function	Description
----------	-------------

<u><i>pthread_barrierattr_getpshared()</i></u>	Get the value of a barrier's process-shared attribute
<u><i>pthread_barrierattr_destroy()</i></u>	Destroy a barrier's attributes object
<u><i>pthread_barrierattr_init()</i></u>	Initialize a barrier's attributes object
<u><i>pthread_barrierattr_setpshared()</i></u>	Set the value of a barrier's process-shared attribute
<u><i>pthread_barrier_destroy()</i></u>	Destroy a barrier
<u><i>pthread_barrier_init()</i></u>	Initialize a barrier
<u><i>pthread_barrier_wait()</i></u>	Synchronize participating threads at the barrier

Sleepon locks

Sleepon locks are very similar to condvars, with a few subtle differences. Like condvars, sleepon locks (*pthread_sleepon_lock()*) can be used to block until a condition becomes true (like a memory location changing value). But unlike condvars, which must be allocated for each condition to be checked, sleepon locks multiplex their functionality over a single mutex and dynamically allocated condvar, regardless of the number of conditions being checked. The maximum number of condvars ends up being equal to the maximum number of blocked threads. These locks are patterned after the sleepon locks commonly used within the UNIX kernel.

Reader/writer locks

More formally known as "Multiple readers, single writer locks," these locks are used when the access pattern for a data structure consists of many threads reading the data, and (at most) one thread writing the data. These locks are more expensive than mutexes, but can be useful for this data access pattern.

This lock works by allowing all the threads that request a read-access lock (*pthread_rwlock_rdlock()*) to succeed in their request. But when a thread wishing to write asks for the lock (*pthread_rwlock_wrlock()*), the request is denied until all the current reading threads release their reading locks (*pthread_rwlock_unlock()*).

Multiple writing threads can queue (in priority order) waiting for their chance to write the protected data structure, and all the blocked writer-threads will get to run before reading threads are allowed access again. The priorities of the reading threads are not considered.

There are also calls (*pthread_rwlock_tryrdlock()* and *pthread_rwlock_trywrlock()*) to allow a thread to test the attempt to achieve the requested lock, without blocking. These calls return with a successful lock or a status indicating that the lock couldn't be granted immediately.

Reader/writer locks aren't implemented directly within the kernel, but are instead built from the mutex and condvar services provided by the kernel.

Semaphores

Semaphores are another common form of synchronization that allows threads to "post" (*sem_post()*) and "wait" (*sem_wait()*) on a semaphore to control when threads wake or sleep. The post operation increments the semaphore; the wait operation decrements it.

If you wait on a semaphore that is positive, you will not block. Waiting on a nonpositive semaphore will block until some other thread executes a post. It is valid to post one or more times before a wait. This use will allow one or more threads to execute the wait without blocking.

A significant difference between semaphores and other synchronization primitives is that semaphores are "async safe" and can be manipulated by signal handlers. If the desired effect is to have a signal handler wake a thread, semaphores are the right choice.



Note that in general, mutexes are much faster than semaphores, which always require a kernel entry.

Another useful property of semaphores is that they were defined to operate between processes. Although our mutexes work between processes, the POSIX thread standard considers this an optional capability and as such may not be portable across systems. For synchronization between threads in a single process, mutexes will be more efficient than semaphores.

As a useful variation, a *named* semaphore service is also available. It uses a resource manager and as such allows semaphores to be used between processes on different machines connected by a network.



Note that named semaphores are *slower* than the unnamed variety.

Since semaphores, like condition variables, can legally return a nonzero value because of a false wakeup, correct usage requires a loop:

```
while (sem_wait(&s) && errno == EINTR) { do_nothing(); }  
do_critical_region(); /* Semaphore was decremented */
```

Synchronization via scheduling algorithm

By selecting the POSIX FIFO scheduling algorithm, we can guarantee that no two threads of the same priority execute the critical section concurrently on a non-SMP system. The FIFO scheduling algorithm dictates that all FIFO-scheduled threads in the system at the same priority will run, when scheduled, until they voluntarily release the processor to another thread.

This "release" can also occur when the thread blocks as part of requesting the service of another process, or when a signal occurs. *The critical region must therefore be carefully coded and documented so that later maintenance of the code doesn't violate this condition.*

In addition, higher-priority threads in that (or any other) process could still preempt these FIFO-scheduled threads. So, all the threads that could "collide" within the critical section must be FIFO-scheduled at the *same* priority. Having enforced this condition, the threads can then casually access this shared memory without having to first make explicit synchronization calls.



This exclusive-access relationship doesn't apply in multi-processor systems, since each CPU could run a thread simultaneously through the region that would otherwise be serially scheduled on a single-processor machine.

Synchronization via message passing

Our Send/Receive/Reply message-passing IPC services (described later) implement an implicit synchronization by their blocking nature. These IPC services can, in many instances, render other synchronization services unnecessary. They are also the only synchronization and IPC primitives (other than named semaphores, which are built on top of messaging) that can be used across the network.

Synchronization via atomic operations

In some cases, you may want to perform a short operation (such as incrementing a variable) with the guarantee that the operation will perform *atomically* -- i.e. the operation won't be preempted by another thread or ISR (Interrupt Service Routine).

Under QNX Neutrino, we provide atomic operations for:

- adding a value
- subtracting a value
- clearing bits
- setting bits
- toggling (complementing) bits.

These atomic operations are available by including the C header file `<atomic.h>`.

Although you can use these atomic operations just about anywhere, you'll find them particularly useful in these two cases:

- between an ISR and a thread
- between two threads (SMP or single-processor).

Since an ISR can preempt a thread at any given point, the only way that the thread would be able to protect itself would be to *disable interrupts*. Since you should avoid disabling interrupts in a realtime system, we recommend that you use the atomic operations provided with QNX Neutrino.

On an SMP system, multiple threads *can* and *do* run concurrently. Again, we run into the same situation as with interrupts above -- you should use the atomic operations where applicable to eliminate the need to disable and reenable interrupts.

Synchronization services implementation

The following table lists the various microkernel calls and the higher-level POSIX calls constructed from them:

Microkernel call	POSIX call	Description
<i>SyncTypeCreate()</i>	<i>pthread_mutex_init()</i> , <i>pthread_cond_init()</i> , <i>sem_init()</i>	Create object for mutex, condvars, and semaphore.
<i>SyncDestroy()</i>	<i>pthread_mutex_destroy()</i> , <i>pthread_cond_destroy()</i> , <i>sem_destroy()</i>	Destroy synchronization object.
<i>SyncCondvarWait()</i>	<i>pthread_cond_wait()</i> , <i>pthread_cond_timedwait()</i>	Block on a condvar.
<i>SyncCondvarSignal()</i>	<i>pthread_cond_broadcast()</i> , <i>pthread_cond_signal()</i>	Wake up condvar-blocked threads.
<i>SyncMutexLock()</i>	<i>pthread_mutex_lock()</i> , <i>pthread_mutex_trylock()</i>	Lock a mutex.
<i>SyncMutexUnlock()</i>	<i>pthread_mutex_unlock()</i>	Unlock a mutex.
<i>SyncSemPost()</i>	<i>sem_post()</i>	Post a semaphore.
<i>SyncSemWait()</i>	<i>sem_wait()</i> , <i>sem_trywait()</i>	Wait on a semaphore.

QNX Neutrino IPC

IPC plays a fundamental role in the transformation of QNX Neutrino from an embedded realtime kernel into a full-scale POSIX operating system. As various service-providing processes are added to the microkernel, IPC is the "glue" that connects those components into a cohesive whole.

Although message passing is the primary form of IPC in QNX Neutrino, several other forms are available as well. Unless otherwise noted, those other forms of IPC are built over our native message passing. The strategy is to create a simple, robust IPC service that can be tuned for performance through a simplified code path in the microkernel; more "feature cluttered" IPC services can then be implemented from these.

Benchmarks comparing higher-level IPC services (like pipes and FIFOs implemented over our messaging) with their monolithic kernel counterparts show comparable performance.

QNX Neutrino offers at least the following forms of IPC:

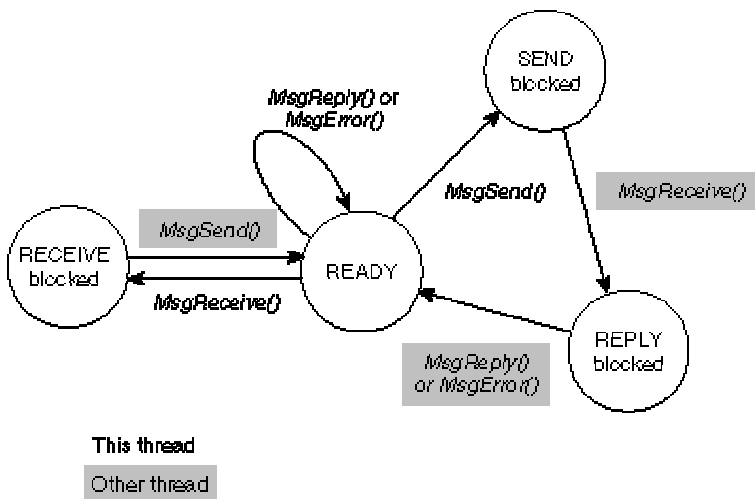
Service:	Implemented in:
Message-passing	kernel
Signals	kernel
POSIX message queues	external process
Shared memory	process manager
Pipes	external process
FIFOs	external process

These services can be selected by the designer on the basis of bandwidth requirements, the need for queuing, network transparency, etc. The tradeoff can be complex, but the flexibility is useful.

As part of the engineering effort that went into defining the QNX Neutrino microkernel, the focus on message passing as the fundamental IPC primitive was deliberate. As a form of IPC, message passing (as implemented in *MsgSend()*, *MsgReceive()*, and *MsgReply()*), is synchronous and copies data. Let's explore these two attributes in more detail.

Synchronous message passing

A thread that does a *MsgSend()* to another thread (which could be within another process) will be blocked until the target thread does a *MsgReceive()*, processes the message, and executes a *MsgReply()*. If a thread executes a *MsgReceive()* without a previously sent message pending, it will block until another thread executes a *MsgSend()*.




State changes in a send-receive-reply transaction.

This inherent blocking synchronizes the execution of the sending thread, since the act of requesting that the data be sent also causes the sending thread to be blocked and the receiving thread to be scheduled for execution. This happens without requiring explicit work by the kernel to determine which thread to run next (as would be the case with most other forms of IPC). Execution and data move directly from one context to another.

Data-queuing capabilities are omitted from these messaging primitives because queuing could be implemented when needed within the receiving thread. The sending thread is often prepared to wait for a response; queuing is unnecessary overhead and complexity (i.e. it slows down the nonqueued case). As a result, the sending thread doesn't need to make a separate, explicit blocking call to wait for a response (as it would if some other IPC form had been used).

While the send and receive operations are blocking and synchronous, *MsgReply()* (or *MsgError()*) doesn't block. Since the client thread is already blocked waiting for the reply, no additional synchronization is required, so a blocking *MsgReply()* isn't needed. This allows a server to reply to a client and continue processing while the kernel and/or networking code asynchronously passes the reply data to the sending thread and marks it ready for execution. Since most servers will tend to do some processing to prepare to receive the next request (at which point they block again), this works out well.

 Note that in a network, a reply may not complete as "immediately" as in a local message pass. For more information on network message passing, see the chapter on [Qnet networking](#) in this book.

MsgReply()* vs *MsgError()

The *MsgReply()* function is used to return a status and zero or more bytes to the client. *MsgError()*, on the other hand, is used to return *only* a status to the client. Both functions will unblock the client from its *MsgSend()*.

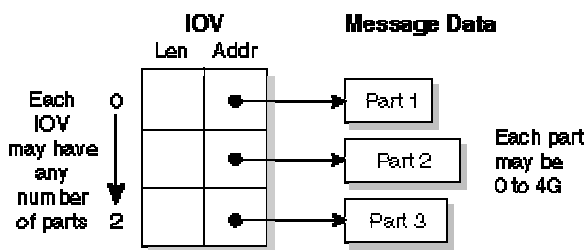
Message copying

Since our messaging services copy a message directly from the address space of one thread to another without intermediate buffering, the message-delivery performance approaches the memory bandwidth of the underlying hardware. The kernel attaches no special meaning to the content of a message -- the data in a message has

meaning only as mutually defined by sender and receiver. However, "well-defined" message types are also provided so that user-written processes or threads can augment or substitute for system-supplied services.

The messaging primitives support multipart transfers, so that a message delivered from the address space of one thread to another needn't pre-exist in a single, contiguous buffer. Instead, both the sending and receiving threads can specify a vector table that indicates where the sending and receiving message fragments reside in memory. Note that the size of the various parts can be different for the sender and receiver.

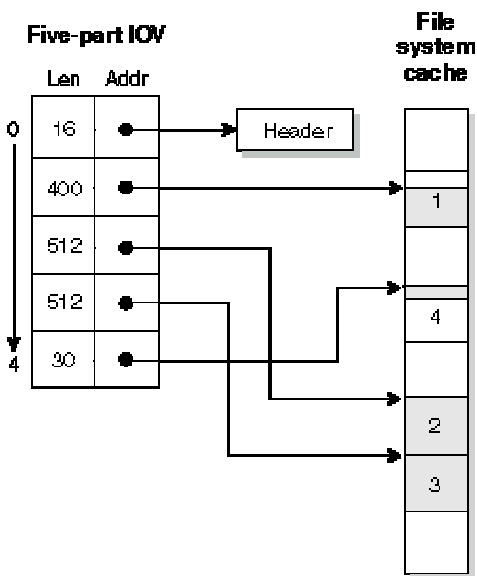
Multipart transfers allow messages that have a header block separate from the data block to be sent without performance-consuming copying of the data to create a contiguous message. In addition, if the underlying data structure is a ring buffer, specifying a three-part message will allow a header and two disjoint ranges within the ring buffer to be sent as a single atomic message. A hardware equivalent of this concept would be that of a scatter/gather DMA facility.



A multipart transfer.

The multipart transfers are also used extensively by filesystems. On a read, the data is copied directly from the filesystem cache into the application using a message with n parts for the data. Each part points into the cache and compensates for the fact that cache blocks aren't contiguous in memory with a read starting or ending within a block.

For example, with a cache block size of 512 bytes, a read of 1454 bytes can be satisfied with a 5-part message:



Scatter/gather of a read of 1454 bytes.

Since message data is explicitly copied between address spaces (rather than by doing page table manipulations), messages can be easily allocated on the stack instead of from a special pool of page-aligned memory for MMU "page flipping." As a result, many of the library routines that implement the API between client and server processes can be trivially expressed, without elaborate IPC-specific memory allocation calls.

For example, the code used by a client thread to request that the filesystem manager execute `lseek` on its behalf is implemented as follows:

```
#include <unistd.h>
#include <errno.h>
#include <sys/iomsg.h>

off64_t lseek64(int fd, off64_t offset, int whence) {
    io_lseek_t      msg;
    off64_t          off;

    msg.i.type = _IO_LSEEK;
    msg.i.combine_len = sizeof msg.i;
    msg.i.offset = offset;
    msg.i.whence = whence;
    msg.i.zero = 0;
    if(MsgSend(fd, &msg.i, sizeof msg.i, &off, sizeof off) == -1) {
        return -1;
    }
    return off;
}

off64_t tell64(int fd) {
    return lseek64(fd, 0, SEEK_CUR);
}

off_t lseek(int fd, off_t offset, int whence) {
    return lseek64(fd, offset, whence);
}

off_t tell(int fd) {
    return lseek64(fd, 0, SEEK_CUR);
}
```

This code essentially builds a message structure on the stack, populates it with various constants and passed parameters from the calling thread, and sends it to the filesystem manager associated with *fd*. The reply indicates the success or failure of the operation.



This implementation doesn't prevent the kernel from detecting large message transfers and choosing to implement "page flipping" for those cases. Since most messages passed are quite tiny, copying messages is often faster than manipulating MMU page tables. For bulk data transfer, shared memory between processes (with message-passing or the other synchronization primitives for notification) is also a viable option.

Simple messages

For simple single-part messages, the OS provides functions that take a pointer directly to a buffer without the need for an IOV (input/output vector). In this case, the number of parts is replaced by the size of the message directly pointed to. In the case of the *message send* primitive -- which takes a send and a reply buffer -- this introduces four variations:

Function	Send message	Reply message
<i>MsgSend()</i>	simple	simple
<i>MsgSendsv()</i>	simple	IOV
<i>MsgSendvs()</i>	IOV	simple
<i>MsgSendv()</i>	IOV	IOV

The other messaging primitives that take a direct message simply drop the trailing "v" in their names:

IOV	Simple direct
<i>MsgReceivev()</i>	<i>MsgReceive()</i>
<i>MsgReceivePulsev()</i>	<i>MsgReceivePulse()</i>
<i>MsgReplyv()</i>	<i>MsgReply()</i>
<i>MsgReadv()</i>	<i>MsgRead()</i>
<i>MsgWritev()</i>	<i>MsgWrite()</i>

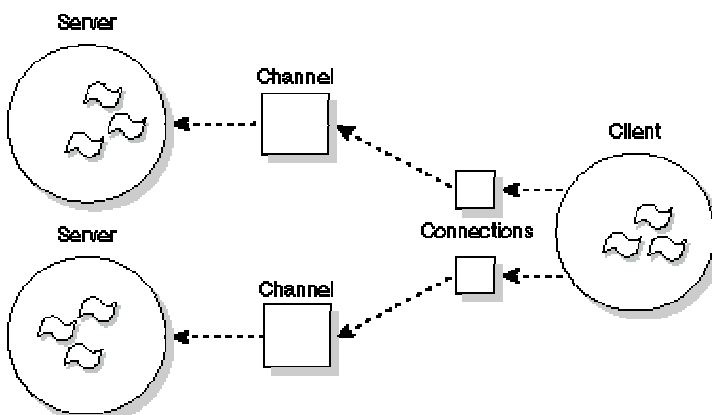
Channels and connections

In QNX Neutrino, message passing is directed towards channels and connections, rather than targeted directly from thread to thread. A thread that wishes to receive messages first creates a channel; another thread that wishes to send a message to that thread must first make a connection by "attaching" to that channel.

Channels are required by the message kernel calls and are used by servers to *MsgReceive()* messages on. Connections are created by client threads to "connect" to the channels made available by servers. Once connections are established, clients can *MsgSend()* messages over them. If a number of threads in a process all attach to the same channel, then the connections all map to the same kernel object for efficiency. Channels and connections are named within a process by a small integer identifier. Client connections map directly into file descriptors.

Architecturally, this is a key point. By having client connections map directly into FDs, we have eliminated yet another layer of translation. We don't need to "figure out" where to send a message based on the file descriptor (e.g. via a *read(fd)* call). Instead, we can simply send a message directly to the "file descriptor" (i.e. connection ID).

Function	Description
<i>ChannelCreate()</i>	Create a channel to receive messages on.
<i>ChannelDestroy()</i>	Destroy a channel.
<i>ConnectAttach()</i>	Create a connection to send messages on.
<i>ConnectDetach()</i>	Detach a connection.



Connections map elegantly into file descriptors.

A process acting as a server would implement an event loop to receive and process messages as follows:

```

chid = ChannelCreate(flags);
SETIOV(&iiov, &msg, sizeof(msg));
for(;;) {
    rcv_id = MsgReceivev( chid, &iiov, parts, &info );

    switch( msg.type ) {
        /* Perform message processing here */
    }

    MsgReplyv( rcv_id, &iiov, rparts );
}

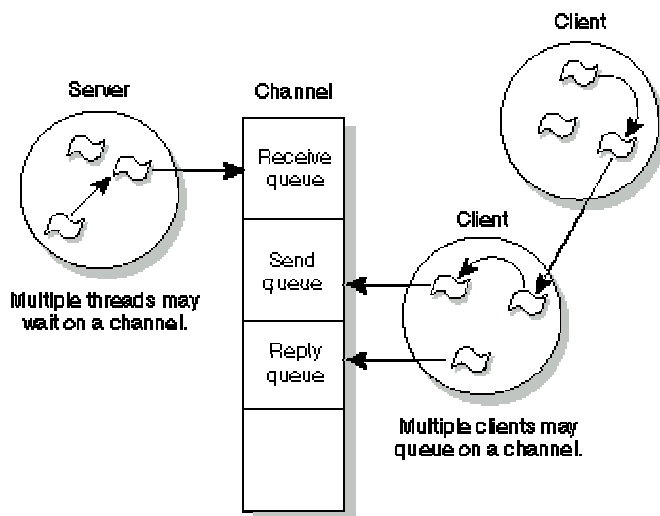
```

This loop allows the thread to receive messages from any thread that had a connection to the channel.

The channel has three queues associated with it:

- one queue for threads waiting for messages
- one queue for threads that have sent a message that hasn't yet been received
- one queue for threads that have sent a message that has been received, but not yet replied to.

While in any of these queues, the waiting thread is blocked (i.e. RECEIVE-, SEND-, or REPLY-blocked).

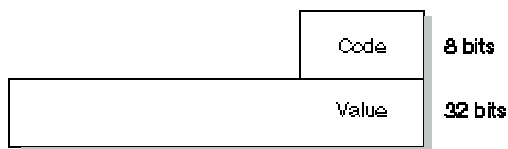


Threads blocked while in a channel queue.

Pulses

In addition to the synchronous Send/Receive/Reply services, the OS also supports fixed-size, nonblocking messages. These are referred to as *pulses* and carry a small payload (four bytes of data plus a single byte code).

Pulses pack a relatively small payload -- eight bits of code and 32 bits of data. Pulses are often used as a notification mechanism within interrupt handlers. They also allow servers to signal clients without blocking on them.



Pulses pack a small payload.

Priority inheritance

A server process receives messages in priority order. As the threads within the server receive requests, they then inherit the priority of the sending thread (but not the scheduling algorithm). As a result, the relative priorities of the threads requesting work of the server are preserved, and the server work will be executed at the appropriate priority. This message-driven priority inheritance avoids priority-inversion problems.

Message-passing API

The message-passing API consists of the following functions:

Function	Description
<i>MsgSend()</i>	Send a message and block until reply.

<code>MsgReceive()</code>	Wait for a message.
<code>MsgReceivePulse()</code>	Wait for a tiny, nonblocking message (pulse).
<code>MsgReply()</code>	Reply to a message.
<code>MsgError()</code>	Reply only with an error status. No message bytes are transferred.
<code>MsgRead()</code>	Read additional data from a received message.
<code>MsgWrite()</code>	Write additional data to a reply message.
<code>MsgInfo()</code>	Obtain info on a received message.
<code>MsgSendPulse()</code>	Send a tiny, nonblocking message (pulse).
<code>MsgDeliverEvent()</code>	Deliver an event to a client.
<code>MsgKeyData()</code>	Key a message to allow security checks.

Robust implementations with Send/Receive/Reply

Architecting a QNX Neutrino application as a team of cooperating threads and processes via Send/Receive/Reply results in a system that uses *synchronous* notification. IPC thus occurs at specified transitions within the system, rather than asynchronously.

A significant problem with asynchronous systems is that event notification requires signal handlers to be run. Asynchronous IPC can make it difficult to thoroughly test the operation of the system and make sure that no matter when the signal handler runs, that processing will continue as intended. Applications often try to avoid this scenario by relying on a "window" explicitly opened and shut, during which signals will be tolerated.

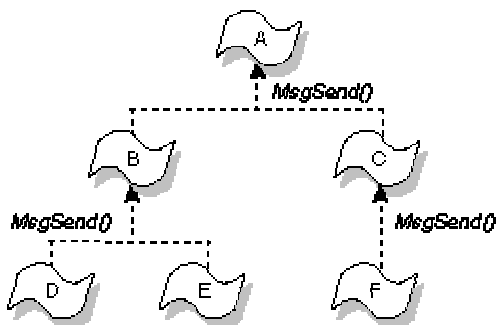
With a synchronous, nonqueued system architecture built around Send/Receive/Reply, robust application architectures can be very readily implemented and delivered.

Avoiding deadlock situations is another difficult problem when constructing applications from various combinations of queued IPC, shared memory, and miscellaneous synchronization primitives. For example, suppose thread A doesn't release mutex 1 until thread B releases mutex 2. Unfortunately, if thread B is in the state of not releasing mutex 2 until thread A releases mutex 1, a standoff results. Simulation tools are often invoked in order to ensure that deadlock won't occur as the system runs.

The Send/Receive/Reply IPC primitives allow the construction of deadlock-free systems with the observation of only these simple rules:

1. Never have two threads send to each other.
2. Always arrange your threads in a hierarchy, with sends going up the tree.

The first rule is an obvious avoidance of the standoff situation, but the second rule requires further explanation. The team of cooperating threads and processes is arranged as follows:



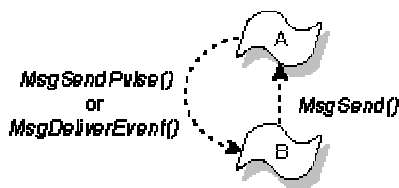
Threads should always send up to higher-level threads.

Here the threads at any given level in the hierarchy never send to each other, but send only upwards instead.

One example of this might be a client application that sends to a database server process, which in turn sends to a filesystem process. Since the sending threads block and wait for the target thread to reply, and since the target thread isn't send-blocked on the sending thread, deadlock can't happen.

But how does a higher-level thread notify a lower-level thread that it has the results of a previously requested operation? (Assume the lower-level thread didn't want to wait for the replied results when it last sent.)

QNX Neutrino provides a very flexible architecture with the *MsgDeliverEvent()* kernel call to deliver nonblocking events. All of the common asynchronous services can be implemented with this. For example, the server-side of the *select()* call is an API that an application can use to allow a thread to wait for an I/O event to complete on a set of file descriptors. In addition to an asynchronous notification mechanism being needed as a "back channel" for notifications from higher-level threads to lower-level threads, we can also build a reliable notification system for timers, hardware interrupts, and other event sources around this.



A higher-level thread can "send" a pulse event.

A related issue is the problem of how a higher-level thread can request work of a lower-level thread without sending to it, risking deadlock. The lower-level thread is present only to serve as a "worker thread" for the higher-level thread, doing work on request. The lower-level thread would send in order to "report for work," but the higher-level thread wouldn't reply then. It would defer the reply until the higher-level thread had work to be done, and it would reply (which is a nonblocking operation) with the data describing the work. In effect, the reply is being used to initiate work, not the send, which neatly side-steps rule #1.

Events

A significant advance in the kernel design for QNX Neutrino is the event-handling subsystem. POSIX and its realtime extensions define a number of asynchronous notification methods (e.g. UNIX signals that don't queue or pass data, POSIX realtime signals that may queue and pass data, etc.)

The kernel also defines additional, QNX-specific notification techniques such as pulses. Implementing all of these event mechanisms could have consumed significant code space, so our implementation strategy was to build all of these notification methods over a single, rich, event subsystem.

A benefit of this approach is that capabilities exclusive to one notification technique can become available to others. For example, an application can apply the same queueing services of POSIX realtime signals to UNIX signals. This can simplify the robust implementation of signal handlers within applications.

The events encountered by an executing thread can come from any of three sources:

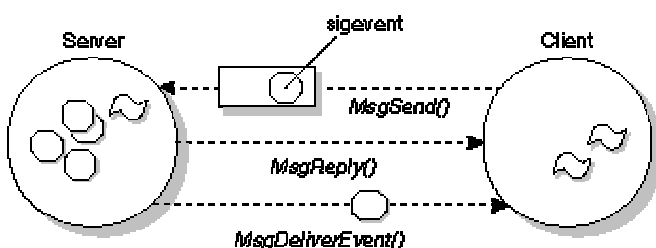
- a *MsgDeliverEvent()* kernel call invoked by a thread
- an interrupt handler

- the expiry of a timer.

The event itself can be any of a number of different types: QNX Neutrino pulses, interrupts, various forms of signals, and forced "unblock" events. "Unblock" is a means by which a thread can be released from a deliberately blocked state without any explicit event actually being delivered.

Given this multiplicity of event types, and applications needing the ability to request whichever asynchronous notification technique best suits their needs, it would be awkward to require that server processes (the higher-level threads from the previous section) carry code to support all these options.

Instead, the client thread can give a data structure, or "cookie," to the server to hang on to until later. When the server needs to notify the client thread, it will invoke *MsgDeliverEvent()* and the microkernel will set the event type encoded within the cookie upon the client thread.



The client sends a sigevent to the server.

I/O notification

The *ionotify()* function is a means by which a client thread can request asynchronous event delivery. Many of the POSIX asynchronous services (e.g. *mq_notify()* and the client-side of the *select()*) are built on top of it. When performing I/O on a file descriptor (*fd*), the thread may choose to wait for an I/O event to complete (for the *write()* case), or for data to arrive (for the *read()* case). Rather than have the thread block on the resource manager process that's servicing the read/write request, *ionotify()* can allow the client thread to post an event to the resource manager that the client thread would like to receive when the indicated I/O condition occurs. Waiting in this manner allows the thread to continue executing and responding to event sources other than just the single I/O request.

The *select()* call is implemented using I/O notification and allows a thread to block and wait for a mix of I/O events on multiple *fd*'s while continuing to respond to other forms of IPC.

Here are the conditions upon which the requested event can be delivered:

- `_NOTIFY_COND_OUTPUT` -- there's room in the output buffer for more data.
- `_NOTIFY_COND_INPUT` -- resource-manager-defined amount of data is available to read.
- `_NOTIFY_OUT_OF_BAND` -- resource-manager-defined "out of band" data is available.

Signals

The OS supports the 32 standard POSIX signals (as in UNIX) as well as the POSIX realtime signals, both numbered from a kernel-implemented set of 64 signals with uniform functionality. While the POSIX standard defines realtime signals as differing from UNIX-style signals (in that they may contain four bytes of data and a

byte code and may be queued for delivery), this functionality can be explicitly selected or deselected on a per-signal basis, allowing this converged implementation to still comply with the standard.

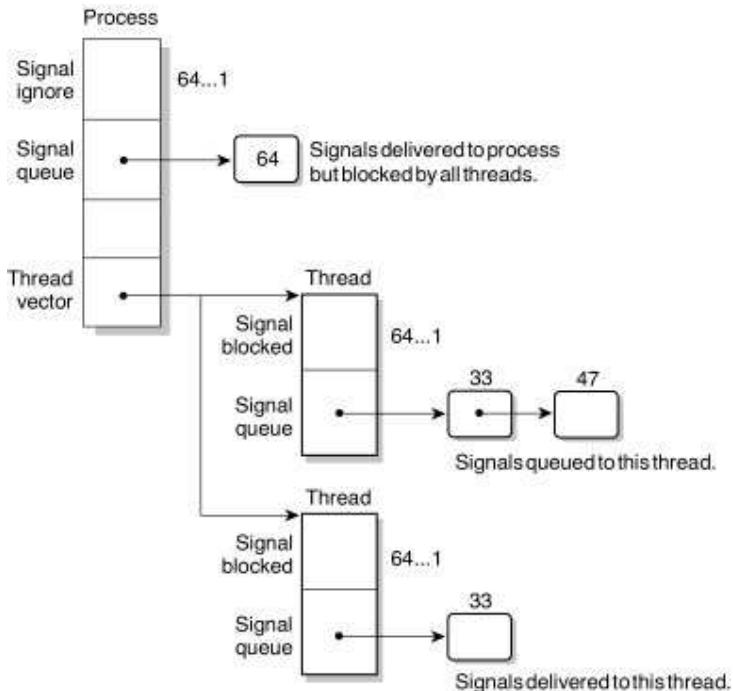
Incidentally, the UNIX-style signals can select POSIX realtime signal queuing, if the application wants it. QNX Neutrino also extends the signal-delivery mechanisms of POSIX by allowing signals to be targeted at specific threads, rather than simply at the process containing the threads. Since signals are an asynchronous event, they're also implemented with the event-delivery mechanisms.

Microkernel call	POSIX call	Description
<i>SignalKill()</i>	<i>kill()</i> , <i>pthread_kill()</i> , <i>raise()</i> , <i>sigqueue()</i>	Set a signal on a process group, process, or thread.
<i>SignalAction()</i>	<i>sigaction()</i>	Define action to take on receipt of a signal.
<i>SignalProcmask()</i>	<i>sigprocmask()</i> , <i>pthread_sigmask()</i>	Change signal blocked mask of a thread.
<i>SignalSuspend()</i>	<i>sigsuspend()</i> , <i>pause()</i>	Block until a signal invokes a signal handler.
<i>SignalWaitinfo()</i>	<i>sigwaitinfo()</i>	Wait for signal and return info on it.

The original POSIX specification defined signal operation on processes only. In a multi-threaded process, the following rules are followed:

- The signal actions are maintained at the process level. If a thread ignores or catches a signal, it affects *all* threads within the process.
- The signal mask is maintained at the thread level. If a thread blocks a signal, it affects only that thread.
- An un-ignored signal targeted at a thread will be delivered to that thread alone.
- An un-ignored signal targeted at a process is delivered to the first thread that doesn't have the signal blocked. If all threads have the signal blocked, the signal will be queued on the process until any thread ignores or unblocks the signal. If ignored, the signal on the process will be removed. If unblocked, the signal will be moved from the process to the thread that unblocked it.

When a signal is targeted at a process with a large number of threads, the thread table must be scanned, looking for a thread with the signal unblocked. Standard practice for most multi-threaded processes is to mask the signal in all threads but one, which is dedicated to handling them. To increase the efficiency of process-signal delivery, the kernel will cache the last thread that accepted a signal and will always attempt to deliver the signal to it first.



Signal delivery.

The POSIX standard includes the concept of queued realtime signals. QNX Neutrino supports optional queuing of any signal, not just realtime signals. The queuing can be specified on a signal-by-signal basis within a process. Each signal can have an associated 8-bit code and a 32-bit value.

This is very similar to message pulses described earlier. The kernel takes advantage of this similarity and uses common code for managing both signals and pulses. The signal number is mapped to a pulse priority using `_SIGMAX -- signo`. As a result, signals are delivered in priority order with *lower* signal numbers having *higher* priority. This conforms with the POSIX standard, which states that existing signals have priority over the new realtime signals.

Special signals

As mentioned earlier, the OS defines a total of 64 signals. Their range is as follows:

Signal range	Description
1 ... 57	57 POSIX signals (including traditional UNIX signals)
41 ... 56	16 POSIX realtime signals (SIGRTMIN to SIGRTMAX)
57 ... 64	Eight special-purpose QNX Neutrino signals

The eight special signals cannot be ignored or caught. An attempt to call the [*signal\(\)*](#) or [*sigaction\(\)*](#) functions or the [*SignalAction\(\)*](#) kernel call to change them will fail with an error of `EINVAL`.

In addition, these signals are always blocked and have signal queuing enabled. An attempt to unblock these signals via the [*sigprocmask\(\)*](#) function or [*SignalProcmask\(\)*](#) kernel call will be quietly ignored.

A regular signal can be programmed to this behavior using the following standard signal calls. The special signals save the programmer from writing this code and protect the signal from accidental changes to this behavior.

```
sigset_t *set;
struct sigaction action;

sigemptyset(&set);
sigaddset(&set, signo);
sigprocmask(SIG_BLOCK, &set, NULL);
```

```
action.sa_handler = SIG_DFL;
action.sa_flags = SA_SIGINFO;
sigaction(signo, &action, NULL);
```

This configuration makes these signals suitable for synchronous notification using the *sigwaitinfo()* function or *SignalWaitinfo()* kernel call. The following code will block until the eighth special signal is received:

```
sigset_t *set;
siginfo_t info;

sigemptyset(&set);
sigaddset(&set, SIGRTMAX + 8);
sigwaitinfo(&set, &info);
printf("Received signal %d with code %d and value %d\n",
       info.si_signo,
       info.si_code,
       info.si_value.sival_int);
```

Since the signals are always blocked, the program cannot be interrupted or killed if the special signal is delivered outside of the *sigwaitinfo()* function. Since signal queuing is always enabled, signals won't be lost -- they'll be queued for the next *sigwaitinfo()* call.

These signals were designed to solve a common IPC requirement where a server wishes to notify a client that it has information available for the client. The server will use the *MsgDeliverEvent()* call to notify the client. There are two reasonable choices for the event within the notification: pulses or signals.

A pulse is the preferred method for a client that may also be a server to other clients. In this case, the client will have created a channel for receiving messages and can also receive the pulse.

This won't be true for most simple clients. In order to receive a pulse, a simple client would be forced to create a channel for this express purpose. A signal can be used in place of a pulse if the signal is configured to be synchronous (i.e. the signal is blocked) and queued -- this is exactly how the special signals are configured. The client would replace the *MsgReceive()* call used to wait for a pulse on a channel with a simple *sigwaitinfo()* call to wait for the signal.

This signal mechanism is used by Photon to wait for events and by the *select()* function to wait for I/O from multiple servers. Of the eight special signals, the first two have been given special names for this use.

```
#define SIGSELECT      (SIGRTMAX + 1)
#define SIGPHOTON      (SIGRTMAX + 2)
```

Summary of signals

Signal	Description
SIGABRT	Abnormal termination signal such as issued by the <i>abort()</i> function.
SIGALRM	Timeout signal such as issued by the <i>alarm()</i> function.
SIGBUS	Indicates a memory parity error (QNX-specific interpretation). Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGCHLD	Child process terminated. The default action is to ignore the signal.
SIGCONT	Continue if HELD. The default action is to ignore the signal if the process isn't HELD.
SIGDEADLK	Mutex deadlock occurred. If you haven't called <i>SyncMutexEvent()</i> , and if the conditions that would cause the kernel to deliver the event occur, then the kernel delivers a SIGDEADLK instead.
SIGEMT	EMT instruction (emulator trap).
SIGFPE	Erroneous arithmetic operation (integer or floating point), such as division by zero or an operation resulting in overflow. Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGHUP	Death of session leader, or hangup detected on controlling terminal.
SIGILL	Detection of an invalid hardware instruction. Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGINT	Interactive attention signal (Break).
SIGIOT	IOT instruction (not generated on x86 hardware).
SIGKILL	Termination signal -- should be used only for emergency situations. <i>This signal cannot be caught or ignored.</i>
SIGPIPE	Attempt to write on a pipe with no readers.
SIGPOLL	Pollable event occurred.
SIGQUIT	Interactive termination signal.
SIGSEGV	Detection of an invalid memory reference. Note that if a second fault occurs while your process is in a signal handler for this fault, the process will be terminated.
SIGSTOP	Stop process (the default). <i>This signal cannot be caught or ignored.</i>
SIGSYS	Bad argument to system call.
SIGTERM	Termination signal.
SIGTRAP	Unsupported software interrupt.
SIGTSTP	Stop signal generated from keyboard.
SIGTTIN	Background read attempted from control terminal.
SIGTTOU	Background write attempted to control terminal.
SIGURG	Urgent condition present on socket.

SIGUSR1	Reserved as application-defined signal 1.
SIGUSR2	Reserved as application-defined signal 2.
SIGWINCH	Window size changed.

POSIX message queues

POSIX defines a set of nonblocking message-passing facilities known as message queues. Like pipes, message queues are named objects that operate with "readers" and "writers." As a priority queue of discrete messages, a message queue has more structure than a pipe and offers applications more control over communications.



To use POSIX message queues in QNX Neutrino, the message queue resource manager (`mqueue`) must be running.

POSIX message queues are implemented in QNX Neutrino via an *optional* resource manager called `mqueue` (see the [Resource Managers](#) chapter in this book).

Unlike our inherent message-passing primitives, the POSIX message queues reside *outside* the kernel.

Why use POSIX message queues?

POSIX message queues provide a familiar interface for many realtime programmers. They are similar to the "mailboxes" found in many realtime executives.

There's a fundamental difference between our messages and POSIX message queues. Our messages block -- they copy their data directly between the address spaces of the processes sending the messages. POSIX messages queues, on the other hand, implement a store-and-forward design in which the sender need not block and may have many outstanding messages queued. POSIX message queues exist independently of the processes that use them. You would likely use message queues in a design where a number of named queues will be operated on by a variety of processes over time.

For raw performance, POSIX message queues will be *slower* than QNX Neutrino native messages for transferring data. However, the flexibility of queues may make this small performance penalty worth the cost.

File-like interface

Message queues resemble files, at least as far as their interface is concerned. You open a message queue with `mq_open()`, close it with `mq_close()`, and destroy it with `mq_unlink()`. And to put data into ("write") and take it out of ("read") a message queue, you use `mq_send()` and `mq_receive()`.

For strict POSIX conformance, you should create message queues that start with a single slash (/) and contain no other slashes. But note that we extend the POSIX standard by supporting pathnames that may contain multiple slashes. This allows, for example, a company to place all its message queues under its company name and distribute a product with increased confidence that a queue name will *not* conflict with that of another company.

In QNX Neutrino, all message queues created will appear in the filename space under the directory `/dev/mqueue`.

<i>mq_open()</i> name:	Pathname of message queue:
/data	/dev/mqueue/data

The Photon microGUI

This chapter covers the following topics:

- [A graphical microkernel](#)
- [The Photon event space](#)
- [Graphics drivers](#)
- [Font support](#)
- [Unicode multilingual support](#)
- [Animation support](#)
- [Multimedia support](#)
- [Printing support](#)
- [The Photon Window Manager](#)
- [Widget library](#)
- [Driver development kits](#)
- [Summary](#)

A graphical microkernel



This chapter provides an overview of the Photon microGUI, the graphical environment for QNX Neutrino. For more information, see the [Photon documentation set](#).

Many embedded systems require a UI so that users can access and control the embedded application. For complex applications, or for maximum ease of use, a graphical windowing system is a natural choice. However, the windowing systems on desktop PCs simply require too much in the way of system resources to be practical in an embedded system where memory and cost are limiting factors.

Drawing upon the successful approach of the QNX Neutrino microkernel architecture to achieve a POSIX OS environment for embedded systems, we have followed a similar course in creating the Photon microGUI windowing system.

To implement an effective microkernel OS, we first had to tune the microkernel so that the kernel calls for IPC were as lean and efficient as possible (since the performance of the whole OS rests on this message-based IPC). Using this low-overhead IPC, we were able to structure a GUI as a graphical "microkernel" process with a team of cooperating processes around it, communicating via that IPC.

While at first glance this might seem similar to building a graphical system around the classic client/server paradigm already used by the X Window System, the Photon architecture differentiates itself by restricting the functionality implemented within the graphical microkernel (or server) itself and distributing the bulk of the GUI functionality among the cooperating processes.

The Photon microkernel runs as a tiny process, implementing only a few fundamental primitives that external, optional processes use to construct the higher-level functionality of a windowing system. Ironically, for the Photon microkernel itself, "windows" do not exist. Nor can the Photon microkernel "draw" anything, or manage a pen, mouse, or keyboard.

To manage a GUI environment, Photon creates a 3-dimensional *event space* and confines itself only to handling *regions* and processing the clipping and steering of various events as they flow through the regions in this event space.

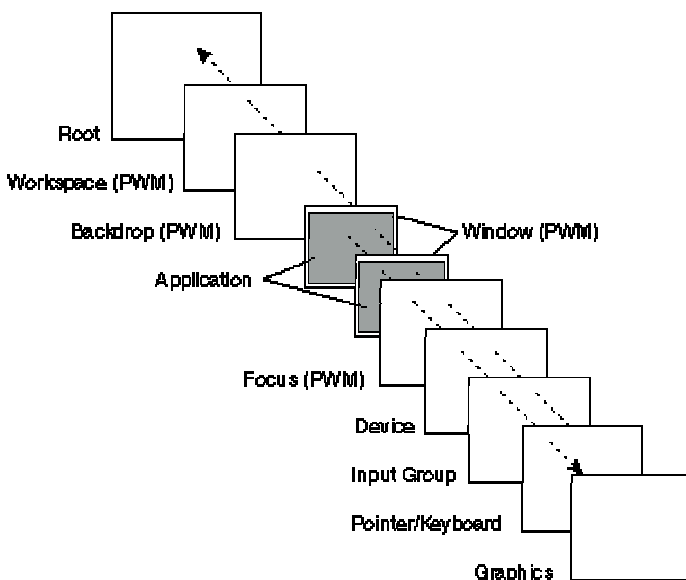
This abstraction is roughly parallel to the concept of a microkernel OS being incapable of filesystem or device I/O, but relying instead on external processes to provide these high-level services. Just as this allows a microkernel OS to scale up or down in size and functionality, so also a microkernel GUI.

The core microkernel "abstraction" implemented by the Photon microkernel is that of a graphical event space that other processes can populate with regions. Using native IPC to communicate with the Photon microkernel, these other processes manipulate their regions to provide higher-level graphical services or to act as user applications. By *removing* service-providing processes, Photon can be scaled down for limited-resource systems; by *adding* service-providing processes, Photon can be scaled up to full desktop functionality.

The Photon event space

The "event space" managed by the Photon microkernel can be visualized as an empty void with a "root region" at the back. The end-user can be imagined to be "looking into" this event space from the front. Applications place "regions" into the 3-dimensional space between the root region and the end-user; they use those regions to generate and accept various types of "events" within this space.

Processes that provide device driver services place regions at the front of the event space. In addition to managing the event space and root region, the Photon microkernel projects *draw events*.



Photon regions.

We can think of these events that travel through this space as "photons" (from which this windowing system gets its name). Events themselves consist of a list of rectangles with some attached data. As these events flow through the event space, their rectangle lists intersect the "regions" placed there by various processes (applications).

Events traveling away from the root region of the event space are said to be traveling outwards (or towards the user), while events from the user are said to be traveling inwards towards the root region at the back of the event space.

The interaction between events and regions is the basis for the input and output facilities in Photon. Pen, mouse, and keyboard events travel away from the user towards the root plane. Draw events originate from regions and travel towards the device plane and the user.

Regions

Regions are managed in a hierarchy associated as a family of rectangles that define their location in the 3-dimensional event space. A region also has attributes that define how it interacts with various classes of events as they intersect the region. The interactions a region can have with events are defined by two bitmasks:

- *sensitivity* bitmask
- *opaque* bitmask.

The sensitivity bitmask uses specific event types to define which intersections the process owning the region wishes to be informed of. A bit in the sensitivity bitmask defines whether or not the region is sensitive to each event type. When an event intersects a region for which the bit is set, a copy of that event is enqueued to the application process that owns the region, notifying the application of events traveling through the region. This notification doesn't modify the event in any way.

The opaque bitmask is used to define which events the region can or can't pass through. For each event type, a bit in the opaque mask defines whether or not the region is opaque or transparent to that event. The optical property of "opaqueness" is implemented by modifying the event itself as it passes through the intersection.

These two bitmasks can be combined to accomplish a variety of effects in the event space. The four possible combinations are:

Bitmask combination:	Description:
Not sensitive, transparent	The event passes through the region, unmodified, without the region owner being notified. The process owning the region simply isn't interested in the event.
Not sensitive, opaque	The event is clipped by the region as it passes through; the region owner isn't notified. For example, most applications would use this attribute combination for draw event clipping, so that an application's window wouldn't be overwritten by draw events coming from underlying windows.
Sensitive, transparent	A copy of the event is sent to the region owner; the event then continues, unmodified, through the event space. A process wishing to log the flow of events through the event space could use this combination.
Sensitive, opaque	A copy of the event is sent to the region owner; the event is also clipped by the region as it passes through. By setting this bitmask combination, an application can act as an event filter or translator. For every event received, the application can process and regenerate it, arbitrarily transformed in some manner, possibly traveling in a new direction, and perhaps sourced from a new coordinate in the event

	space. For example, a region could absorb pen events, perform handwriting recognition on those events, and then generate the equivalent keystroke events.
--	---

Events

Like regions, events also come in various classes and have various attributes. An event is defined by:

- an originating region
- a type
- a direction
- an attached list of rectangles
- some event-specific data (optional).

Unlike most windowing systems, Photon classifies *both* input (pen, mouse, keyboard, etc.) and output (drawing requests) as events. Events can be generated either from the regions that processes have placed in the event space or by the Photon microkernel itself. Event types are defined for:

- keystrokes
- pen and mouse button actions
- pen and mouse motion
- region boundary crossings
- expose and covered events
- draw events
- drag events
- drag-and-drop events.

Application processes can either poll for these events, block and wait for them to occur, or be asynchronously notified of a pending event.

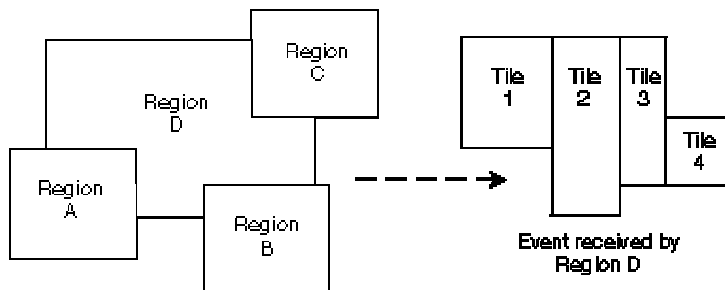
The rectangle list attached to the event can describe one or more rectangular regions, or it can be a "point-source" -- a single rectangle where the upper-left corner is the same as the lower-right corner.

When an event intersects a region that is opaque to it, that region's rectangle is "clipped out" of the event's list of rectangles such that the list describes only the portion of the event that would ultimately be visible.

The best way to illustrate how this clipping is performed is to examine the changes in the rectangle list of a draw event as it passes through various intersecting regions. When the draw event is first generated, the rectangle list consists of only a single, simple rectangle describing the region that the event originated from.

If the event goes through a region that clips the upper-left corner out of the draw event, the rectangle list is modified to contain only the two rectangles that would define the area remaining to be drawn. These resulting rectangles are called "tiles."

Likewise, every time the draw event intersects a region opaque to draw events, the rectangle list will be modified to represent what will remain visible of the draw event after the opaque region has been "clipped out." Ultimately, when the draw event arrives at a graphics driver ready to be drawn, the rectangle list will precisely define only the portion of the draw event that is to be visible.



Opaque regions are clipped out.

If the event is entirely clipped by the intersection of a region, the draw event will cease to exist. This mechanism of "opaque" windows modifying the rectangle list of a draw event is how draw events from an underlying region (and its attached process) are properly clipped for display as they travel towards the user.

Graphics drivers

Graphics drivers are implemented as processes that place a region at the front of the event space. Rather than inject pen, mouse, or keyboard events (as would an input driver), a graphics driver's region is *sensitive* to draw events coming out of the event space. As draw events intersect the region, those events are received by the graphics driver process. In effect, the region can be imagined to be coated in "phosphor," which is illuminated by the impact of "photons."

Since the Photon drawing API accumulates draw requests into batches emitted as single draw events, each draw event received by the driver contains a list of individual graphical primitives to be rendered. By the time the draw event intersects the graphics driver region, its rectangle list will also contain a "clip list" describing exactly which portions of the draw list are to be rendered to the display. The driver's job is to transform this clipped draw list into a visual representation on whatever graphics hardware the driver is controlling.

One advantage of delivering a "clip list" within the event passed to the driver is that each draw request then represents a significant "batch" of work. As graphics hardware advances, more and more of this "batch" of work can be pushed directly into the graphics hardware. Many display controller chips already handle a single clip rectangle; some handle multiple clip rectangles.

Although using the OS IPC services to pass draw requests from the application to the graphics driver may appear to be an unacceptable overhead, our performance measurements demonstrate that this implementation performs as well as having the application make direct calls into a graphics driver. One reason for such performance is that multiple draw calls are batched with the event mechanism, allowing the already minimal overhead of our lightweight IPC to be amortized over many draw calls.

Multiple graphics drivers

Since graphics drivers simply put a region into the Photon event space, and since that region describes a rectangle to be intersected by draw events, it naturally follows that multiple graphics drivers can be started for multiple graphics controller cards, all with their draw-event-sensitive regions present in the same event space.

These multiple regions could be placed adjacent to each other, describing an array of "drawable" tiles, or overlapped in various ways. Since Photon inherits the OS's network transparency, Photon applications or drivers can run on any network node, allowing additional graphics drivers to extend the graphical space of Photon to

the physical displays of many networked computers. By having the graphics driver regions overlap, the draw events can be replicated onto multiple display screens.

Many interesting applications become possible with this capability. For example, a factory operator with a wireless-LAN handheld computer could walk up to a workstation and drag a window from a plant control screen onto the handheld, and then walk out onto the plant floor and interact with the control system.

In other environments, an embedded system without a UI could project a display onto any network-connected computer. This connectivity also enables useful collaborative modes of work for people using computers -- a group of people could simultaneously see the same application screen and cooperatively operate the application.

From the application's perspective, this looks like a single unified graphical space. From the user's perspective, this looks like a seamlessly connected set of computers, where windows can be dragged from physical screen to physical screen across network links.

Color model

Colors processed by the graphics drivers are defined by a 24-bit RGB quantity (8 bits for each of red, green, and blue), providing a total range of 16,777,216 colors. Depending on the actual display hardware, the driver will either invoke the 24-bit color directly from the underlying hardware or map it into the color space supported by the less-capable hardware.

Since the graphics drivers use a hardware-independent color representation, applications can be displayed without modifications on hardware that has varied color models. This allows applications to be "dragged" from screen to screen, without concern for what the underlying display hardware's color model might be.

Font support

Photon uses Bitstream's Font Fusion object-oriented font engine, which provides developers with full font fidelity and high-quality typographic output at any resolution on any device, while maintaining the integrity of the original character shapes.

Photon is shipped with a limited number of TrueType fonts. These industry-standard fonts are readily available from various sources.

Stroke-based fonts

To support Asian languages (e.g. Chinese, Japanese, and Korean), Photon relies on Bitstream's stroke-based fonts. These high-speed fonts are ideal for memory-constrained environments. For example, a complete traditional Chinese font with over 13,000 characters can occupy as much as 8M in a conventional desktop system -- a stroke-based version of the same font occupies less than 0.5M!

Apart from their compact size and fast rasterization, these fonts are also fully *scalable*, which makes them perfect for various nondesktop displays such as LCDs, TVs, PDAs, and so on.

Unicode multilingual support

Photon is designed to handle international characters. Following the Unicode Standard (ISO/IEC 10646), Photon provides developers with the ability to create applications that can easily support the world's major languages and scripts.



Scripts that read from right to left, such as Arabic, aren't supported at this time.

Unicode is modeled on the ASCII character set, but uses a 16-bit (or 32-bit) encoding to support full multilingual text. There's no need for escape sequences or control codes when specifying any character in any

language. Note that Unicode encoding conveniently treats all characters -- whether alphabetic, ideographs, or symbols -- in exactly the same way.

UTF-8 encoding

Formerly known as UTF-2, the UTF-8 (for "8-bit form") transformation format is designed to address the use of Unicode character data in 8-bit UNIX environments.

Here are some of the main features of UTF-8:

- Unicode characters from U+0000 to U+007E (ASCII set) map to UTF-8 bytes 00 to 7E (ASCII values).
- ASCII values don't otherwise occur in a UTF-8 transformation, giving complete compatibility with historical filesystems that parse for ASCII bytes.
- The first byte indicates the number of bytes to follow in a multibyte sequence, allowing for efficient forward parsing.
- Finding the start of a character from an arbitrary location in a byte stream is efficient, because you need to search at most four bytes backwards to find an easily recognizable initial byte. For example: `isInitialByte = ((byte & 0xC0) != 0x80);`
- UTF-8 is reasonably compact in terms of the number of bytes used for encoding.
- UTF-8 strings are terminated with a single NULL byte, like traditional ASCII C strings.

Animation support

Photon provides flicker-free animations by employing off-screen video memory where possible. For instance, a special container widget (`PtOSContainer`) creates a dedicated off-screen memory context for drawing images. The `PtOSContainer` widget uses a block of video memory large enough to hold an image the size of its canvas. (For more information about widgets, see the section "[Widget library](#)" in this chapter.)

Photon's graphics drivers also maximize the use of off-screen memory to enhance the perceptual performance of animated images. The graphics drivers support other advanced techniques, such as direct graphics mode, alpha-blending, chroma-key substitution, and more.

Video overlay

Besides the ability to superimpose a semi-transparent image on top of a background (alpha-blending) or to place a color-masked foreground image on top of a separate background (chroma-key), Photon also supports *video overlay* -- a full-motion video image is rendered within a window on the display.

Layers

Some display controllers let you transparently overlay multiple "screens" on a single display. Each overlay is called a *layer*.

You can use layers to combine independent display elements. Since the graphics hardware performs the overlaying, this can be more efficient than rendering all of the display elements onto a single layer. For example, a fast navigational display can have a scrolling navigational map on a background layer and a web browser or other popup GUI element on a foreground layer.

The images on all the active layers of a display are combined, using alpha-blending, chroma-keying, or both, to produce the final image on the display.

Multimedia support

Many developers of Photon applications need to target resource-constrained embedded systems. While conventional multimedia solutions rely on powerful CPUs and graphics hardware, Photon's multimedia support is ideal for embedded systems equipped with "low-end" processors (e.g. Cyrix MediaGX 200) and limited system RAM (e.g. 16M).

Plugin architecture

Photon provides a suite of multimedia plugins that are as fast and light as possible, yet able to deliver the functionality of high-end systems. Using Photon's extensible plugin architecture and media API, developers can easily integrate their own multimedia components.

To enable embedded systems to play audio/video files without depending on disk-based access, Photon supports *streaming* technologies (MPEG-1 System Stream and MPEG-2 Program Stream).

Media Player

This Photon application forms the core of our multimedia support. Its interface is simple -- the user selects an audio or video file to play, and the Media Player "plays" it. But behind the scenes, the Media Player:

- identifies and loads/unloads the appropriate plugins
- initializes the plugins
- translates user input into plugin commands
- handles plugin-generated events (callbacks)
- identifies minor file types (registry bypass)
- supports playlists
- provides the GUI for QNX plugins
- provides OEM panes for custom plugins.

Media Player plugins

Currently, the multimedia plugins include:

- Audio CD Player -- a multitrack plugin that plays audio CDs
- MPEG Audio Player -- decodes MPEG-1 audio layers 1, 2, and 3 (MP3); also supports MPEG-2 low-bit rate streams and MPEG 2.5 audio.
- MPEG Audio/Video Player -- decodes MPEG-1 System and MPEG-2 Program streams
- MPEG Video Player -- decodes MPEG-1 video streams
- Soundfile Player -- plays simple audio files; supports Wave, AIFF, and other formats.



QNX Software Systems provides licenses for some, but not all, of the above technologies. For details, contact your sales representative.

Printing support

Photon provides built-in printing support for a variety of outputs, including:

- bitmap files
- PostScript
- Hewlett-Packard PCL
- Epson ESC/P2
- Epson IJS
- Canon
- Lexmark

Photon also comes with a print-selection widget/convenience dialog to make printing simpler within developers' own applications.

The Photon Window Manager

Adding a window manager to Photon creates a fully functional desktop-class GUI. The window manager is entirely optional and can be omitted for most classes of embedded systems. If present, the window manager allows the user to manipulate application windows by resizing, moving, and iconifying them.

The window manager is built on the concept of filtering events with additional regions placed behind application regions, upon which a title bar, resize handles, and other elements are drawn and interacted with. Since the replaceable window manager implements the actual "look and feel" of the environment, various UI flavors can be optionally installed.

Widget library

Photon provides a library of components known as *widgets* -- objects that can manage much of their on-screen behavior automatically, without explicit programming effort. As a result, a complete application can be quickly assembled by combining widgets in various ways and then attaching C code to the appropriate callbacks the widgets provide.

The Photon Application Builder (PhAB), which is included as part of the Photon development system, provides an extensive widget palette in its visual development environment.

Photon provides a wide range of widgets, which can be classified as follows:

- fundamental widgets (e.g. a button)
- container widgets (e.g. a window widget)

advanced widgets (e.g. an HTML display widget).

Driver development kits

As a product geared towards developers, Photon offers all the tools needed to build high-performance, accelerated graphics drivers that can be readily tailored for particular graphics cards and chipsets.

Developers will be able to create drivers that support advanced graphics techniques (e.g. alpha-blending or chroma-key substitution) through a software-only approach, a perfect fallback for "simple" hardware that doesn't directly handle such techniques.

The Photon graphics driver development kit provides full source code for several sample drivers as well as detailed instructions for handling the hardware-dependent issues involved in developing custom drivers.

Summary

Photon represents a new approach to GUI building -- using a microkernel and a team of cooperating processes, rather than the monolithic approach typified by other windowing systems. As a result, Photon exhibits a unique set of capabilities:

- Low memory requirements enable Photon to deliver a high level of windowing functionality to environments where only a graphics library might have been allowed within the memory constraints.
- Photon provides a very flexible, user-extensible architecture that allows developers to extend the GUI in directions unique to their applications.

With flexible cross-platform connectivity, Photon applications can be used from virtually any connected desktop environment.