An Architectural Overview of QNX®

Dan Hildebrand
Quantum Software Systems Ltd.
175 Terrence Matthews
Kanata, Ontario K2M 1W8
Canada
(613) 591-0931

danh@quantum.on.ca

*Abstract**

This paper presents an architectural overview of the QNX operating system. QNX is an OS that provides applications with a fully network- and multi-processor-distributed, realtime environment that delivers nearly the full, device-level performance of the underlying hardware. The OS architecture used to deliver this operating environment is that of a realtime microkernel surrounded by a collection of optional processes that provide POSIX- and UNIX-compatible system services. By including or excluding various resource managers at run time, QNX can be scaled down for ROM-based embedded systems, or scaled up to encompass hundreds of processors—either tightly or loosely connected by various LAN technologies. Conformance to POSIX standard 1003.1, draft standard 1003.2 (shell and utilities) and draft standard 1003.4 (realtime) is maintained transparently throughout the distributed environment.

*This paper appeared in the Proceedings of the Usenix Workshop on Micro-Kernels & Other Kernel Architectures, Seattle, April, 1992. ISBN 1-880446-42-1

Architecture: Past and Present

From its creation in 1982, the QNX architecture has been fundamentally similar to its current form—that of a very small microkernel (approximately 10K at that time) surrounded by a team of cooperating processes that provide higher-level OS services. To date, QNX has been used in nearly 200,000 systems, predominantly in applications where realtime performance, development flexibility, and network flexibility have been fundamental requirements. The large installed base has proven that microkernel technology is both commercially viable and suitable for mission-critical applications such as process control, medical instrumentation, and financial transaction processing. The performance needs of these applications have been a significant driving force in the evolution of QNX from version 1.00 up through version 3.15.

In 1989, the development of a POSIX-compliant version of QNX (4.0) began with the goals of maximizing the performance and flexibility delivered by the previous generation of the product. This new version was released in 1991. This paper will detail the features of the new architecture and discuss its strengths and limitations, as well as areas targeted for future development.

A True Microkernel

The QNX microkernel implements four services: interprocess communication, low-level network communication, process scheduling, and interrupt dispatching. There are 14 kernel calls associated with these services. In total, these functions occupy roughly 7K of code and provide the functionality and performance of a realtime executive (see Appendix A). Given the small kernel size, processors that provide a reasonable amount of on-chip cache can deliver excellent performance for applications that heavily use the services of the microkernel, since the microkernel and the system interrupt handlers can often fit comfortably within an on-chip CPU cache of 8K.

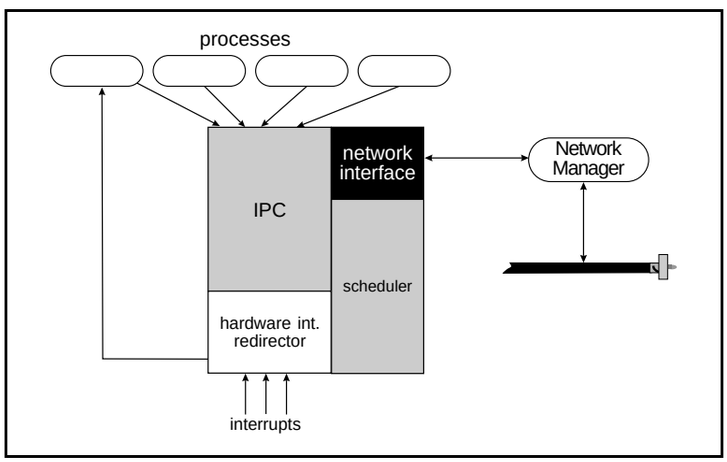


Figure 1. — The QNX Microkernel

The message-passing facilities provided by the microkernel are blocking versions of *Send*(), *Receive*(), and *Reply*(). To summarize, a process that does a *Send*() to another process will be blocked until the target process does a *Receive*(), processes the message, and does a *Reply*(). If a process executes a *Receive*() without a message pending, it will block until another process executes a *Send*(). Since these primitives copy directly from process to process without queuing, message delivery performance approaches the memory bandwidth of the underlying hardware. All system services are built upon these message-passing primitives. Variations on these IPC primitives (e.g. message queues) have been easily implemented as servers employing these lower-level services.

Performance of these alternate IPC servers is comparable and often superior to the performance of these services implemented within monolithic kernels (see Appendix B).

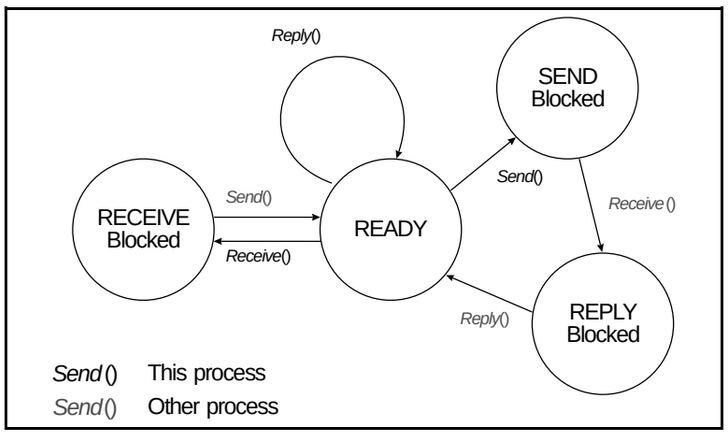


Figure 2. States involved in a typical send-receive-reply transaction.

Processes can request that messages be delivered in priority order (rather than in time order) and that process execution proceed at the priority of the highest-priority blocked process waiting for service. This message-driven priority mechanism neatly avoids the priority inversion problems that can result in fixed-priority message-passing systems. Server processes are forced to execute at the priority of the process they are serving, and yet automatically have their priority appropriately boosted when a higher-priority process blocks on the busy server. As a result, a low-priority process cannot preempt a higher-priority process by invoking the services of an even higher-priority server.

The messaging primitives support multi-part messaging, such that a message delivered from one process to another need not occupy a single, contiguous area in memory. Instead, both the sending and receiving processes can specify an MX table that indicates where the sending and receiving message fragments exist in memory. This allows 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, a three-part message will allow a header and two disjoint ranges within the ring buffer to be sent as a single, atomic message. The MX mapping applied to the message by the sender and the receiver need not be the same.

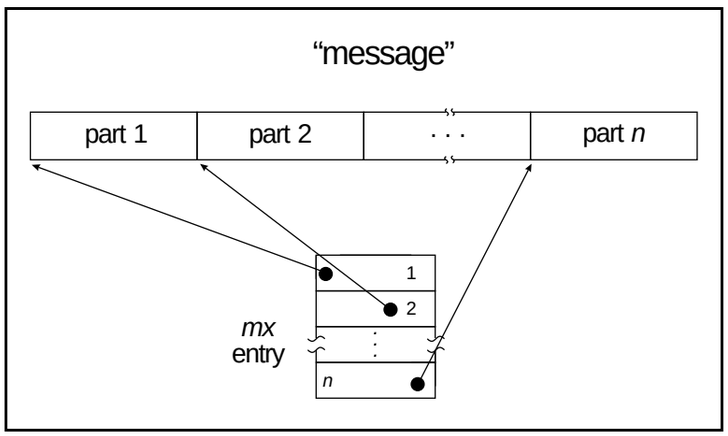


Figure 3. Multi-part messages can be specified with an MX control structure. The microkernel assembles these into a single data stream.

Low-level network communications are designed directly into the microkernel and are provided by an optional process known as the network manager (described later in this paper). When present, the network manager is directly connected into the microkernel and provides the microkernel with the facilities needed to move messages to and from other microkernels on the LAN. By providing network services at this fundamental level in the system, any services provided in higher architectural layers of the operating system are transparently accessible to any process, anywhere on the network. Architecturally speaking, this implementation is very lean, providing as efficient an interface as possible.

The process-scheduling primitives provided by QNX conform to the POSIX 1003.4 (real-time) draft specification. QNX provides fully preemptive, prioritized context switching with round-robin, FIFO, and adaptive scheduling. As the POSIX 1003.4 standard emerges from draft status, the microkernel and system processes will be evolved to match.

Resource Managers and Pathname Space Management

For the microkernel to deliver the functionality specified by POSIX standards and UNIX conventions, optional processes known as resource managers can be added. A minimal system, without a filesystem or device I/O system, can be built from a microkernel, a process manager, and a set of application processes.

The first, and only mandatory, resource manager is the process manager (**`Proc`**). **`Proc`** provides services such as process creation, process accounting, memory management, process environment inheritance (both locally and for network remote processes) and pathname space management. First-level pathname management is done by **`Proc`** because, unlike a monolithic-kernel system where the filesystem is always present, a filesystem is optional under QNX. Diskless or ROM-based systems may have no use for a filesystem, and so are not forced to use one.

Until resource managers begin execution, **`Proc`** "owns" the entire pathname space (the root and everything beneath it). Without any resource managers present to provide services, this is essentially an empty filesystem. **`Proc`** allows resource managers, through a standard API, to adopt a portion of the namespace (a "domain of authority") that they would like to administer. **`Proc`** is then responsible for maintaining a prefix tree to track the processes that own various portions of the pathname space.

When **Fsys** (the filesystem manager) and **Dev** (the device manager) are running, the prefix tree would look something like this:

/	*Disk-based filesystem (***Fsys**)
/dev	*Character device system (***Dev**)
/dev/hd0	*Raw disk volume (***Fsys**)
/dev/null	*Null device (***Dev**)

When a process opens a file, the *open*() library routine first sends the filename to **Proc** where the pathname is applied against the prefix tree in order to direct the *open*() to the appropriate resource manager. In the case of partially overlapping domains of authority, the longest match to the pathname would win. For example, if **/dev/tty0** were opened, the longest match would occur on **/dev**, causing the open to be directed to **Dev**. The pathname **/usr/fred** would match against **/**, directing the open to **Fsys**.

The process manager on each computer in the network maintains its own prefix tree and may present identical or different views of the network-wide pathname space to processes on each node. Pathnames that start with a **/** are applied against the prefix tree on that node. Network-unique names are also available to allow applications to specify the absolute location of resources within the network-wide pathname space. Through the use of prefix aliasing, portions of the namespace can be mapped to resource managers on other network nodes. For example, a diskless workstation that booted from the LAN and wished to have its filesystem root on another node could alias the root of its filesystem to a remote **Fsys** process.

With this alias in place, *open*() calls to **/dev** would still map to the local **Dev** process for control of local devices, but all *open*() calls for files would result in open messages being resolved by the prefix mapping table on the previously specified remote node (which would usually direct file opens to the **Fsys** process on that node). As a result, processes anywhere on the network can access all of the network filesystem resources within a single directory tree, connected to a common root. Alternatively, by using network-absolute pathnames, the network pathname space can also be manipulated as a collection of individual root filesystems.

By implementing individual domains of authority within the conventional filename space, portions of the overall functionality of the OS can be implemented in a runtime-optional manner. Since resource managers live outside the kernel space and are not fundamentally different from user processes, they can be added or removed dynamically, at runtime, without requiring that the kernel be relinked to contain different levels of functionality. This flexibility in sizing allows the OS to be easily scaled up or down, depending on application needs.

Although placing the services provided by resource managers outside the kernel would at first appear to be inefficient, the performance results given in Appendix B indicate that the context switch and IPC performance of the microkernel are more than adequate to keep up with the raw performance of the hardware.

The network transparency of this namespace allows remote execution of processes to be logically equivalent to execution on a local processor. The individually administered pathname spaces blend seamlessly, and there are no “surprises” in how the namespace behaves. Inheritance of the entire parent process environment, including open file descriptors, environment variables, and the current working directory, is done such that interprocessor communications and file I/O operate in accordance with the POSIX 1003.1 specification in spite of the network-distributed runtime context.

Fsys—The Filesystem Manager

Fsys is the resource manager that provides a POSIX-compliant filesystem for the QNX environment. It implements a disk structure that uses a bitmap for free space allocation, and a linked-list-of-extents approach to organizing the data on disk. This approach allows the system to deliver disk throughput at the application level that approaches the raw capacity of the hardware (see Appendix B). **Fsys** performs synchronous writes to disk for data structures critical to filesystem integrity, allowing the disk system to gracefully survive unexpected power outages. Special tags embedded in on-disk data structures allow the filesystem to be easily rebuilt in the event of catastrophic failures as well.

The multithreaded architecture of **Fsys** allows it to deal with multiple requests in parallel, such that ramdisk and cache I/O can occur while other threads are blocked, waiting for physical I/O to occur. This parallelism extends down into the driver as well—if a device can support multiple pending I/O requests, then the requests can be serviced by the driver in whatever order is appropriate.

Although an initial study of message-passing operating systems might suggest that a filesystem would need to copy data around more so than a monolithic kernel filesystem, the reality is that no additional copying is needed. The MX multipart messaging primitives allow **Fsys** to map the contiguous buffers specified by the *read*() and *write*() calls of the application into the non-contiguous cache blocks within **Fsys**. For a disk read, the disk driver reads from disk into multiple non-contiguous, LRU-allocated cache blocks. **Fsys** then invokes the MX facility within the kernel to atomically gather and copy the scattered blocks into the contiguous read buffer specified by the application. As a result, even though the filesystem exists within a message-passing, network-transparent environment, it exhibits the same amount of data copying that would occur with a filesystem implemented in a monolithic kernel.

The **Fsys** process can be started from the command line on a diskless, network-connected machine, and device drivers can then be dynamically attached to **Fsys**. In the event that it is no longer required, **Fsys** and its drivers can be removed from memory.

Dev—The Device Manager

The device manager (**Dev**) provides POSIX-compliant device control with some extensions suitable for realtime communications. In a manner similar to **Fsys**, **Dev** can be dynamically started and its device drivers attached and then later removed from memory if no longer needed.

Dev can handle baud rates up to 115 Kbaud on modest hardware with non-intelligent UART devices because of the low interrupt latency provided by the microkernel. With the addition of intelligent communication boards, a high-bandwidth, multiline communications server can be configured.

Use of the MX messaging primitives allows **Dev** to *Receive*() the *write*() done by an application to a device directly into a ring buffer managed by an interrupt handler. With the MX table appropriately defined, the data received can be laid directly into the ring buffer managed by **Dev**. Since writing to a ring buffer can require that the data be mapped into physically disjoint (but logically contiguous) memory regions, an MX table with three entries can describe the header and the two physically disjoint sections of the ring buffer. For the *read*() case, the data flow from a device goes directly from the driver into the ring buffer, and from the ring buffer into the application's *read*() buffer without redundant copying to build contiguous messages.

Device Driver Support

Rather than insist that device-driver interrupt handlers live only in the kernel space, QNX provides a system call that allows user processes to connect a handler within a sufficiently privileged user process to a particular interrupt vector within the kernel. The connected handler can then be called by the kernel in response to physical interrupts. By existing within the user process, the handler has full access to the address space of the process for the purpose of responding to the interrupt. Once the handler has run, it can either wake up the process it shares code with or simply return to the kernel. The device drivers for **`Dev`** take advantage of this behavior by using the individual interrupts to accumulate characters within a **`Dev`** managed buffer, waking **`Dev`** only when a previously defined "significant event" has occurred (such as a terminal character count, end-of-line condition, or timeout).

With interrupt handlers existing outside the kernel in this manner, the user can dynamically add and remove interrupt handlers (and the device drivers that contain them) from a running system. The first-level interrupt handling done by the kernel also takes care of nested and shared interrupts without imposing the hardware-dependant details and complexities on user-written interrupt handlers. External interrupt handler support for the microkernel is fundamental to allowing a resource manager to match the level of performance that resource management within a monolithic kernel could provide.

Ease of Extension

A fundamental advantage to having device drivers exist within user-level processes is that developing extensions to the OS is not functionally different from developing user-level processes. In fact, the development approach used in-house at Quantum is to execute experimental resource managers under the control of the full-screen, source-level debugger, so that debugging OS services like a new **`Fsys`** process can be done without having to resort to low functionality tools such as the kernel debuggers typically used to debug kernel-linked extensions for monolithic kernel operating systems. Since resource managers and device drivers can be started and removed at will, the laborious process of relinking a kernel and rebooting to test the new kernel becomes entirely unnecessary.

As an example of how easily extensible the QNX system is, services such as a **`/proc`** resource manager similar to that described in [Pike 90] have been implemented by applications-level programmers (not kernel architects!) with only a few hours of effort and less than 200 lines of easily understood C source. In effect, the **`/proc`** resource manager packages up a system resource (the list of active processes in the system) and then presents it to the system as files and directories that can be manipulated within **`/proc`** pathname space.

As a more complex example, a client-side network filesystem cache manager similar to that described by [Presotto 91] has been implemented with approximately a week of effort. This cache keeps copies of recently accessed file blocks at the client-side for a node accessing a network remote Fsys. At *open*(), the cache verifies that the remote file has not changed (which would invalidate the locally cached data) and provides the locally cached data as a performance enhancement. By providing a local disk file for this client-side cache to "spill" into, a system networked over a slow serial link can still provide reasonable network-remote filesystem performance. This server represented only 1000 lines of source, and again, was fully within the reach of an application-level programmer using standard system libraries.

Finally, guest filesystems can be implemented as a resource manager that uses the raw disk block I/O services of **`Fsys`** to present a guest filesystem as a subtree within the root filesystem. One example is **`Dosfsys`**, a PC-DOS filesystem. **`Dosfsys`** adopts the **`/dos`**

pathname as its domain of authority and then presents to the system a series of directories of the form **/dos/a**, **/dos/b**, etc. These directories map onto corresponding PC-DOS media and the **Dosfsys** process manipulates the raw blocks on these volumes as indicated by the I/O requests that enter **Dosfsys**. File manipulations that can be mapped onto the underlying filesystem are supported, while others—such as *link*()—return the appropriate error status when attempted.

Network Services—FLEET™ Networking Technology

Fault-tolerant
Load-balancing
Efficient
Extensible
Transparent

As mentioned previously, the network manager (**Net**) is directly connected into the microkernel. When the microkernel is invoked to pass a message from a local process to a process on another node, it enqueues a pointer to the message through this private interface to **Net**. Similarly, **Net** can receive messages from other microkernels and give those messages to the local microkernel. Essentially, the network managers on the network merge the many remote microkernels into a single microkernel. Since all system services—including process creation, debugging, file and device I/O—are accomplished via message passing through the microkernel, the result is a network of machines that behave like a single computer. Any services provided in higher architectural layers of the operating system are transparently accessible to all processes on the network. This is in marked contrast to a TCP/IP services suite, which provides only very explicit sets of services—typically terminal sessions and file access. By comparison, this connected microkernel architecture allows the command:

```
ls /usr/danh | grep abc | wc
```

to be run such that every process will run on a different processor on the network, while the network-inherited file descriptors provided by **Proc** cause the pipes to connect and forward data over the network. The transparency of this environment also facilitates the implementation of distributed applications. For example, the development of a network-distributed team **make** utility was accomplished with only a man-week of effort, starting from a conventional, non-distributed **make**.

Just as **Fsys** and **Dev** can be started and stopped from the command line, each having a family of drivers, **Net** also has a family of drivers and supports the connection of multiple network drivers to **Net**. If **Net** discovers that more than one of the net drivers provides connectivity to the same node, it will load-balance the traffic between the drivers. The load-balancing uses an algorithm based on media transmission rate and queue depth. Command-line options are available to manually coerce network traffic as desired.

Use of multiple network paths between nodes provides better throughput and fault-tolerance by adding extra network links between network nodes. Application-level changes are not needed to take advantage of this fault-tolerance, since the support exists locally within the network manager.

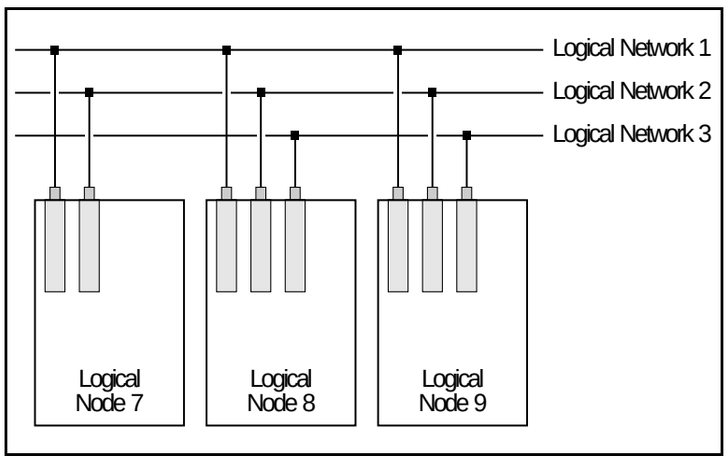


Figure 4. Multiple physical networks happily coexist via logical networks.

Inexpensive serial links can be used as a fall-back network link in case the main LAN fails. By running the serial link at a high baud rate (**Dev** is capable of 115 Kbaud), by doing data compression, and by enabling client-side filesystem caching, serial network performance can be very snappy.

This facility may also be used to resolve the LAN congestion problems that result when two file servers on a LAN experience a high volume of point-to-point traffic. With the FLEET approach, a private network link can be added to join the two servers, moving the point-to-point traffic off the main LAN and onto the private link. If the two servers are physically adjacent, unconventional LAN technologies such as point-to-point SCSI or bus-to-bus DMA become viable options. This approach can be used to implement CPU/Fileserver groups much as described in [Presotto 91].

The FLEET approach also allows a system backplane bus to be used to construct a multiprocessor system by building a processor board that uses the backplane bus as a VLAN (Very Local Area Network). Each processor board would run a QNX OS consisting of a microkernel, **Proc**, **Net**, and a **Net.vlan** driver. One of the processors could run **Net** with both a **Net.vlan** driver and a **Net.ethernet** driver to access an external ethernet LAN. By adding additional hardware to these processors, and the appropriate **Dev** or **Fsys** processes, they become distributed I/O processors. Currently, a test implementation of this architecture using a Microchannel bus for the VLAN is under joint development with Aox Incorporated. The Microchannel burst mode and multimaster bus arbitration will perform very well as a VLAN. In effect, each node on an Ethernet could contain a VLAN of additional processors within the chassis. This lets the team of processes that normally run on each Ethernet node to redistribute and run on a team of processors within that node. The compute servers described by [Tanenbaum 89] can be readily implemented with this hardware.

For embedded applications, a minimal QNX system can be put into less than 100K of ROM (microkernel, **Proc**, and some applications). With the addition of a **Net** process and a **Net** driver (approx. 35K), the embedded system could then be connected to a larger network, becoming a seamless extension of the larger LAN. This would allow the embedded system to access databases, graphical user interfaces, LAN gateways, and other services. In spite of the limited functionality of the embedded system, the network link out to the LAN provides access to the entire LAN's resources for the processes running on the embedded system. The embedded system could also boot from the LAN, further reducing its ROM require-

ments. Because the system debugging services are implemented through standard messages to the **Proc** process on the node running the application being debugged, applications on the embedded system can be debugged from any other node on the LAN.

To host standard transport protocols, a Clarkson-compatible raw packet delivery service provided by **Net** and the network drivers is available. With a protocol stack implemented in this manner, non-QNX machines on the same physical LAN can communicate through the protocol stack to access the services of the multi-processor QNX LAN. The QNX environment would appear to the outside world (e.g. TCP/IP) as a single, multiprocessor machine.

Currently, FLEET does not support network bridging. This requires that communication between nodes not connected to a common network will need to make use of intermediate agent processes to pass messages from LAN to LAN. Research is in progress to define a routing process running as an adjunct to **Net** to perform this function.

Maintainability

A fundamental problem with the maintenance of a monolithic kernel operating system is that all of the kernel code runs in a common, shared address space. The danger that one portion of the kernel might corrupt the data space of another is very real, and must be considered every time new drivers are linked into the kernel. The approach taken by QNX is to explicitly define the interfaces between the components that make up the OS, such that each resource manager, just like user processes, runs in its own memory-protected space, and all communication between the OS modules is through standard system IPC services. As a result, errors introduced by one resource manager will be constrained to that subsystem and will not corrupt other, unrelated resource managers in the system.

Given that new resource managers and device drivers can be debugged and profiled using the same tools as would be used on user processes, system development becomes at least as well instrumented as application development. This is very important, as it allows much greater freedom to experiment with new approaches to implementing OS subsystems without incurring the tremendous effort of debugging a kernel with limited tools.

The architecture also demonstrates a relative simplicity and ease of implementation that allows the maintenance of existing code to be manageable, and the addition of new features to be a task with a reasonable scope. The following table presents the source line count and code size for the various modules that make up the QNX system (all of the source line counts in this paper were generated by counting the semicolons in the C source files).

	Lines of Source	Code Size
Microkernel	605	7K
Proc	3924	52K
Fsys	4457	57K
Fsys.ahascsi	596	11K
Dev	2204	23K
Dev.con	1885	19K
Net	1142	18K
Net.ether	1117	17K
	15930 lines	204 Kbytes

This source line count compares favorably to [Pike 90], although the design goals for the two systems are somewhat different.

Future Directions

Now that QNX is UNIX source code compatible, development of binary compatibility is under way. The combination of source and binary compatibility (ABI) will allow existing UNIX applications to be hosted on a QNX runtime platform and benefit from network-transparent distributed processing and enhanced system performance.

The emergence of commercially successful symmetric shared-memory multiprocessor machines has also raised the issue of multiprocessor support in the QNX microkernel. Given that the microkernel is less than 7K in size, the complexities of multiprocessor support can be constrained to a well-defined portion of the system and will result in a robust implementation. Since the resource manager processes that provide the remainder of the operating system services are multithreaded independent processes, they will inherit the multiprocessor support provided by the microkernel without modification, and the individual components of the operating system will then achieve true concurrency.

Performance

A necessary challenge that QNX had to meet was the performance needs of a customer base primarily concerned with realtime applications. Although an elegant OS architecture is a joy to work with, "academic elegance" will not necessarily create a commercially successful operating system—it must also provide performance better than traditional monolithic kernel operating systems. A design goal of many of the current microkernel operating systems has been to attempt to match the performance of monolithic kernel systems [Guillemont 91]. Given that current monolithic systems, such as UNIX SVR4, fail to deliver the full performance of the hardware (Appendix B), matching only this level of performance will fail to provide consumers of operating system technology with a significant advantage of using a microkernel-based system. Much as with RISC processors, until a new technology can deliver a clear performance advantage, it will remain little more than an architectural detail to an end-user, and not a factor to influence buying decisions.

For QNX to deliver the full performance of the hardware to the application level (and to exceed the performance of monolithic kernel operating systems), a number of architectural innovations were developed. A necessary precondition for these enhancements was that realtime system performance not be compromised. Even though the increased generality of the message-passing model might at first study indicate that it has more overhead than the monolithic kernel, there are a number of architectural ideas that correct this misconception. Two concepts that contributed significantly to overall system performance were the support for interrupt handlers directly within resource managers, and the multipart messaging primitives.

Appendix B contains the performance results for both a DELL UNIX SVR4 v2.1 implementation and a QNX 4.1 implementation performing similar operations. DELL UNIX was chosen for its reputation as a well-performing port of the SVR4 product to the Intel 80x86 architecture. In the first section the timing for a typical UNIX kernel call (**`umask`**) is compared, but under QNX, *umask*() is actually implemented as a message to **`Proc`**. The QNX system call rate for *umask*() is roughly one third that of UNIX, given that these calls represent a different sequence of execution for QNX than for UNIX. The sequence executed is as follows:

1) The test program calls *Send*() in the kernel

2) The kernel schedules **`Proc`** to run

3) A context switch to **`Proc`**

4) **`Proc`** returns from the *Receive*() call it was blocked on

5) **`Proc`** processes the request

6) **`Proc`** calls *Reply*() in the kernel

7) The kernel schedules the test program to run

8) The test program returns from the *Send*() kernel call

In effect, QNX is doing two kernel calls, doing two message passes, executing the scheduling code twice, and performing two context switches in roughly the time it takes the UNIX system to perform three kernel calls. During this process, there are two points at which other processes can be scheduled, rather than only at system timer intervals. It might appear that an obvious optimization would be to add more kernel calls to the microkernel. However, note that these operations are not those that typically form the bottleneck for system-level performance. Another advantage for these calls to remain as messages to **`Proc`** is that they are network-transparent and can be invoked from any processor on the network.

The *Yield*() call is a true kernel call under QNX, and the results in Appendix B show the kernel call rate to be more than three times that of the UNIX kernel. Implementing the other calls measured in this report in the microkernel would have resulted in much faster kernel call times, but since these are not a performance bottleneck for the system, there is no pressing need to enlarge the kernel to accommodate them. Additionally, the greater the complexity of the microkernel, the slower the more important kernel calls will become, until the microkernel has grown back into a monolithic kernel, with all the limitations this implies.

At the system performance level, for IPC, pipe I/O, and disk I/O, we see that QNX outperformed the UNIX system by a substantial margin. In fact, the QNX system was able to deliver virtually all of the raw device throughput to the application, while the SVR4 system fell far short. For disk I/O, QNX was substantially faster than SVR4. As faster peripheral devices appear, the ability to deliver the full performance of that hardware will make possible a class of applications that the kernel overhead of UNIX will not be able to accommodate without much larger investments in processor power. In the network case, the QNX **`Net`** process and its drivers deliver very nearly the entire cable bandwidth to the application, even with only moderately powerful machines.

Conclusion

With the experience gained from implementing and analyzing the QNX microkernel architecture, it is clear that a microkernel system can both outperform and provide greater functionality than a monolithic kernel system while still providing a compatible API for application programs. Existing application source code continues to work unchanged, yet the development of OS extensions becomes much easier. The flexibility of the OS platform also paves the way for greater variety and easier experimentation with alternative operating system features as well. Much as innovations in RISC processor architectures have generated a flurry of new performance capabilities in computer hardware, the microkernel OS architecture will generate a renaissance of new performance and functionality standards in operating system technology.

References

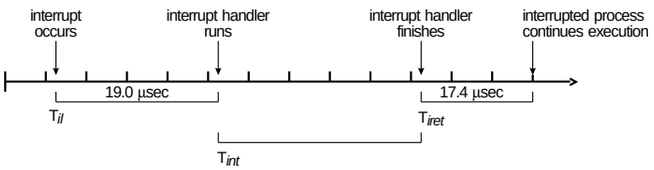
[Guillemont 91] M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. *A Second-Generation Micro-Kernel Based UNIX: Lessons in Performance and Compatibility*. Proceedings of the Usenix Winter'91 Conference, Dallas, January 21-25, 1991, pp. 13-22

[Pike 90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. *Plan 9 from Bell Labs. Proceedings of the Summer 1990 UKUUG Conference, London, July, 1990, pp. 1-9*

[Presotto 91] Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey. *Plan 9, A Distributed System*. Proceedings of the Spring 1991 EurOpen Conference, Tromsö, May, 1991, pp. 43-50

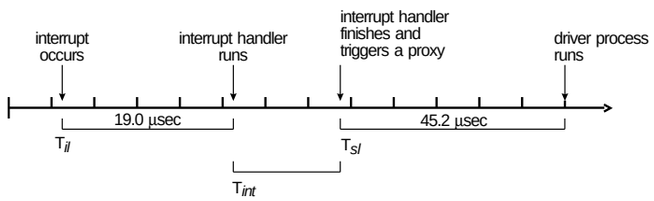
[Tanenbaum 89] Andrew Tanenbaum, Rob van Renesse, and Hans van Staveren. *A Retrospective and Evaluation of the Amoeba Distributed Operating System*. Technical Report, Vrije University, Amsterdam, October, 1989, pp. 27

Appendix A



T_{il} interrupt latency
T_{int} interrupt processing time
T_{iret} interrupt termination time

Interrupt handler simply terminates. Times are for a 20 MHz 386 processor in protected mode.



T_{il} interrupt latency
T_{int} interrupt processing time
T_{sl} scheduling latency

Interrupt handler terminatesand triggers a proxy. Times are for a 20 MHz 386 processor in protected mode.

The interrupt latency (T_{il}) in the above diagram represents the minimum latency—that which occurs when interrupts were fully enabled at the time the interrupt occurred. Worst-case interrupt latency will be this time plus the longest time in which QNX, or the running QNX process, disables CPU interrupts.

Interrupt and Process Latency

Processor	Typical Interrupt Latency (T_{il})	Interrupt Termination time (T_{iret})	Scheduling Latency (T_{sl})	Context Switch
33 Mhz 486	6 µsec	5 µsec	14 µsec	17 µsec
25 Mhz 486	8 µsec	7 µsec	18 µsec	22 µsec
33 Mhz 386	11 µsec	10 µsec	27 µsec	33 µsec
20 Mhz 386	19 µsec	17 µsec	45 µsec	55 µsec
16 Mhz 386SX	32 µsec	29 µsec	77 µsec	94 µsec
8 Mhz 286	65 µsec	59 µsec	163 µsec	188 µsec

Appendix B

System performance numbers comparing QNX4.1 to SVR4 UNIX.

Hardware Environment:

Processor:	Intel 80486 at 33 MHz, ISA bus
Cache:	8K on chip, 0K off chip
RAM:	16 Megabytes
Disk:	1.2 Gigabyte Micropolis SCSI Disk
Controller:	Adaptec 1542B

Software Environment:

A default installation of QNX4.1 with the pipe manager was used for the QNX benchmarks.

A default installation of DELL SVR4 v2.1 UNIX was used for the UNIX benchmarks.

Both QNX and DELL UNIX were run in multiuser mode. The QNX system used a fixed-size 2M cache and the DELL system used the default SVR4 caching algorithms.

Results:

Kernel Call:

		QNX	UNIX	Ratio
umask	*umask*() system call (umasks/sec)	10560	28743	0.37
Yield	*Yield*() system call (yields/sec)	99760	n/a	3.49 ①
message	message passing (msgs/second)	26296	1887	13.94 ②

① Since the *Yield*() call is defined in POSIX 1003.4 and is not supported under DELL UNIX, we will assume that if it was supported, the UNIX kernel would be able to perform it as quickly as the *umask*() call. Making this assumption allows us to compute a comparison ratio. The *Yield*() kernel call under QNX is implemented in a manner roughly comparable to the *umask*() kernel call in UNIX and serves well for comparison of kernel entry overhead.

② QNX is using its native *Send*()/*Receive*()/*Reply*() messaging primitives while UNIX is using its standard message-passing facilities.

Pipe I/O:

Block size	QNX	UNIX	Ratio
1024 bytes (bytes/second)	1948398	916598	2.13
16384 bytes (bytes/second)	3886920	2114722	1.84

Sequential file I/O:

Write a 16M file, and then read it, using 8192-byte *read*() and *write*() calls. Both the UNIX UFS and S5-1K filesystems were tested.

	QNX	UNIX UFS	Ratio
Read (bytes/second)	1430404	289811	4.94
Write (bytes/second)	777875	262513	2.96

	QNX	UNIX S5-1K	Ratio
Read (bytes/second)	1430404	175200	8.16
Write (bytes/second)	777875	60068	12.95

QNX Network Throughput:

Measured as the data transfer rate from a user process on one node to a user process on a second node across a private, two-node network. Each node is a 33 MHz 386.

Arcnet theoretical maximum:	200,800 bytes/second	2.5 Mbits/second
Single Arcnet:	190,000 bytes/second	95% efficient
Dual Arcnet:	380,000 bytes/second	95% efficient
Ethernet theoretical maximum:	1,185,840 bytes/second	10 Mbits/second
Ethernet:	960,000 K/second	81% efficient

Readers familiar with the transfer rates seen with NFS on an Ethernet with this class of processor will appreciate these performance numbers.