QNX 4 OS

CE4218 Paper

Oliver Gerler
9954465
CE4218: Real-Time Systems
University of Limerick, Ireland

April 5, 2000

Abstract

The QNX Realtime Operating System is a commercially real-time, POSIX-certified, network-distributed OS that can easily be scaled from compact embedded systems up through vast networks running hundreds of processors. Consisting of a 8K microkernel and a team of modules, QNX provides priority-driven scheduling and responsive context switching. QNX supports multiple networks (such as Arcnet, Ethernet, Token Ring, FDDI) simultaneously. Modules are available for TCP/IP with NFS; the X Window System with Motif, data acquisition, and Photon microGUI, a full-featured embedded windowing system that takes up only 300K of RAM/ROM. As such, it is a working implementation of different scheduling algorithms and shows a different approach to the vast number of much diversifying problems, all solved by one OS.

1 Introduction

There are different ways for a Real-Time Operating System (RTOS) to come into existence. One is the result of an academic research, another is there because of need for a specific OS for a specific solution to a specific problem and some of them exist out of commercial interest.

QNX[1] is an RTOS out of the last type. It therefore is not freely available, but reports in magazines and journals are quite frequent. This is due to its wide use in many different kinds of applications, from small embedded systems with very limited ressources in hardware and computation power up to vast distributed entities confined to serve as computing and networking system for whole enterprises.

It is not a new OS. QNX was created in 1982, as stated by its creator Dan Hildebrand[2], it can said to be merely free of the bugs and problems new OSs have to deal with. It proved its robustness and usefulness in industrial applications and systems, ranging from small handheld applications to networks of interconnected nodes. By this it is not necessarilly limited to one such configuration, it can easily communicate with different instances of itself in differently equipped nodes. Synchronisation of a PDA (Personal Digital Assistant) with the host data base is so not more than connecting the device to the network and not even that if more sophisticated connections will be made possible, like IR- or RF-LANs.

1.1 Microkernel

The OS is centered around a microkernel, which only makes up 7KB[3], which will fit into a typical 8K on-chip processor cache, improving execution speeds no end. It is often said[4] that microkernels, by their nature message-passing and not based on kernel calls like monolithic OSs, introduce too much overhead due to this message passing and the necessary context switch following that. But if the time needed for switching can be made small enough to be insignificantly against the overall interval needed by the OS to perform the service request, this argument becomes invalid. Message-passing introduces not much overhead on itself at it can be conducted by simply passing a pointer to the other process or taking into account the statistical distribution of the length of the messages: most of them will be very small and given the rate at which processors can block-move memory, this overhead will not be significant, too.

This small microkernel implements four basic services, namely interprocess communication (IPC), interrupt dispatching, the network interface and scheduling. The beauty of QNX[5] is, that it is easyly expandable by further drivers, services, servers, applications and other processes, all of them in no hierarchical, but cooperating order and so making it easy

[1]See `http://www.qnx.com/`.

[2]See [2, p.2].

[3]See [2, p.2], [3, p.8] and several others.

[4]According to [3, p.7 - Microkernel Myths].

[5]The QNX RTOS exists in two different manners: the smaller one, called Neutrino, a subset of the larger one, QNX OS, with the only apparent difference that Neutrino has one scheduling algorithm less than the full flavored QNX OS.

to expand the functionality of the system. The unique approach of QNX to transparent computing is the ability to have processes launched across the network allowing for full inheritance of the environment. Also messaging to other processes is transparent and the process usually has no way of telling if the other process is running on the same node or not. This transparency is implemented into the extreme making it possible to move running applications from one node to another without the need of interrupting the execution of the code. This feature is possible due to the fact that new processes, regardless their "importance" to the node can be installed or removed at runtime without need to reboot or restart anything.

1.2 Features of an RTOS

QNX claims to be a Real-Time Operating System. There are several features and necessities to be taken into consideration, though.

An RTOS enables the user to be able to construct a real-time system. To quote a magazine[6]: "A specific RTOS can only allow you to develop a hard real-time system. But having such an RTOS will not prevent you from developing a system that does not meet deadlines."[7] The introduction of time into the definition of correctness of code execution moves the user of such a system into a position where he has to consider the environment of the system he achieves to construct. The timing demands of an industrial welding robot positioned at a crucial point along the assembly line are different from a handheld device that only has to react to user input in a reasonable small time.

To judge about a system and its real-time abilities, several criteria can be applied, some of them may be: asynchronous I/O, efficiency, multiprocessing ability, preemptive multitasking, critical code in RAM all time, existence of priority concepts, short and defined delay times, predictable thread synchronisation and mechanisms of priority and environment inheritance.[8] This means that a process has to be able to be put into a blocked mode when waiting for an I/O-device to finish and not using processor time by polling the device. The overhead of the OS to the total processing time is an architectural factor influencing the systems ability to make decisions before their deadlines. There must not be page swapping for essential code in a real-time system as this is heavily taking time and will very certainly exceed every deadline. The predictability of delay times introduced by scheduling and interrupt latency has to be well defined to know of worst case scenarios and being able to work with them.[9]

[6] *The Real-Time Magazine,* `http://www.realtime-magazine.com/`

[7] See [11].

[8] Taken from `http://www.elsoft.com.pl/definitions.html`, [1] and [11].

[9] For a negative example of a would-be real-time OS see [11], especially when it comes to dispatch levels and interrupt handlers, or [8] concerning several real-time extensions to NT. It also lacks of an open, standardized, useable API, which renders the word "expandibility" meaningless. Kernel faults and the trapping of them are a further problem.

2 Principles of Operation

QNX achieves its efficiency, modularity and simplicity through two fundamental principles: a microkernel and message-based interprocess communication. Because of the smallness of the kernel the scalability of the system is immense and only necessary and featured modules and processes will be included in the system, leading to a very adapted system built from standard or custom modules.

Processes in this system can be created from other processes with various degrees of inheritance, many of them optional to the user. The inherited parameters range from PID, open files, signals, environment variables, names or timers to priorities and scheduler policies.

2.1 IPC, Messages, Proxies and Signals

The communication of the processes with themselves or the kernel is conducted through messages acting as messengers for requests, replies or interrupts. As the microkernel is responsible for dispatching the messages, it is of no concern for the process how the message will reach its destination. At that point the transparency of QNX comes to attraction as the kernel passes the message, whatever its content, which has only meaning to sender and receiver, to the receiver. That will be a simple task if the receiving process is on the local machine, or the dispatching part of the kernel will consult the network interface part of the kernel to find the node the receiver runs on and passes the message on to that node.

This is implemented by so-called Virtual Circuits (VCs) connecting logically from one node to another. The endpoints to these VCs form virtual processes which perform as the local mirror of the other end of the connection, so for the processes sending and receiving messages it is all one huge space and for them no network exists.[10] The termination of a VC happens when it will become unable to reach a certain process it is connected to. This can be due to shutting down the node the process is running on, disconnecting the respective node from the network or termination of the prcoess itself. All of this will be properly recognized by QNX and acted on accordingly by gracefully terminating the VC if no other connections are performed through it, as only one VC connects one physical pair of nodes.

Processes can be in different states, regarding their sent or pending messages:

READY: The task is ready to run, it does not to wait for a message.

SEND-blocked: The task has sent a message and has to wait for the receiver to acknowledge its dispatching.

REPLY-blocked: The task has sent a message, which was acknowledged by the receiving task, but that task sent no reply, yet.

RECEIVE-blocked: The task wants to receive a message but there is no message pending at its port.

[10]These endpoints to the VCs form Virtual Process IDs (VIDs), which simulate the other process across the network to the lcoal process by taking over their Process ID (PID). For further informations regarding VCs and VIDs see [5, Interprocess communication].

These states now define the nature of the principal communication between tasks: each of them has to acknowledge to the other process.[11]

There are, however, methods in QNX to be able to send information to other processes non-blockingly, which is especially suited for event notification where no interaction with the receiver is necessary, for example interrupt notification to a certain process. Such a method is known as Proxy within the QNX terminology. These proxies basically work as device-in-between. Each time it is triggered by a task which knows of its existence, the proxy sends a predefined message to a predefined receiver.

The third method are Signals, which are widely known and used in unixoid systems and therefore quite standardized. QNX uses this system, too, but extending it in its functionality. So is it possible to block signals if execution of the corresponding signal handler is not useful or possible at a certain time. While executing a signal in the handler, this signal will automatically be blocked by QNX to prevent nested calls to the same handler, simplifying the design and proofability of correctness for the code.

These communication means are also used across the network using VCs and VIDs as described above. Additionally Virtual Proxies connect to the corresponding VID on the node where the real proxy is residing, enabling remote preocesses to trigger this proxy.

2.2 Network and Filesystem

The network module has not to be built into the operating system image. For small embedded systems there is often no need of networking facilities so they can just be left out. Or, as system requirements change, it can be started or stopped at any time, leaving room for other calculations to be done (which can be transferred to remote nodes afterwards).

It is in essence a process like any other process and communicates with the microkernel thorugh messages. Not only is it enhancing the message-passing IPC by propagating messages to remote machines, it also offers more advanced features like load balancing, fault tolerance and bridging.

Load balancing is a neat feature if there are more than one network connections from one node to another. One example are dial-in nodes connected by ISDN to a service provider. As the demand on data throughput increases, more connections over the ISDN can be initiated and the load can be balanced across those connections to increase the overall throughput of the network connection. It is also possible to have more than one physical network interface card (NIC) in the nodes, connecting them by parallel networks, which can be of different types (Ethernet, Arcnet, Tokenring, FDDI, . . .), too.

Fault tolerance will be possible if connections between nodes become redundant. The network module of QNX will be aware of that and will reroute messages if the primary connection breaks down.

Bridging means that there can be more distinct QNX-networks connected by non-QNX-networks. The two nodes adjacent to the other net-

[11] Advanced facilities of QNX are conditional reading of messages to prevent deadlocks and polling by immediate return from receive-blocked if no message is at the port, or reading or writing only parts of a message to be able to allocate memory dynamically according to the length of the message, or processing multipart messages.

work can bridge so that for all QNX nodes on both sides it will look like one space of resources and facilities.

QNX is implemented such that it is very robust against changing networks, unexpected shutting down of different nodes or routes, or leaving a file on a remote resource unvalidated after an unexpected loss of connection.

2.3 Scheduling

It is important to understand how the processes interact with each other to see how that influences the scheduling algorithms and the decision making.

The scheduler is one essential part of the microkernel and as such makes the decision on when to schedule which process. These decisions are based upon these criteria:[12]

Unblocking of a process: The process will be awaken by receiving a message it has waited for or an unexpected signal.

Expiring of the timeslice for a process: The process used up all its available processor time at this round and will be blocked.

Preemption: Occurrence of an interrupt or a prcoess at a higher interrupt.

The scheduling is based upon priorities. The next process to run is selected by choosing the one with the highest priority of which there are 32, with 10 being the standard priority set on initialization.

This will only work if only one process per priority is ready at that moment. For more than one process at a given priority, several algorithms of decision finding have been proposed, from simple First-Come-First-Serve (FCFS)[13] to complex Time-Controlled Dynamic Scheduling (TCDS)[14]. Especially if it comes to scheduling across networks, as will be of importance for Quality-of-Service (QoS)[15] applications, the algorithms to achieve proper scheduling become more and more complex (one approach to that topic could be ripple scheduling.[16]), the question arises where to start with execution of actual code and where to end using up time with finding an optimal scheduling algorithm and time-table. QNX avoids these problems by having a different approach in viewing resources across a network as being local to the local processes, therefore not taking into account the time it needs for them to be signalled or for them to receive data.

In QNX, however, one of three quite simple algorithms can be choosen. Any process on the system may run using one of these methods, as the choosing of the algorithm is no global feature but inherent only to a process alone. These three scheduling methods are:

[12]See [5, The Microkernel, Process Scheduling].

[13]The first task to enter the waiting loop is the first one to be executed, regardless if others have a more urgent deadline or any other criterium.

[14]Scheduling and execution are interleaved. See [10, p.287].

[15]See [10, p.188].

[16]See [10, p.192].

FIFO Scheduling: A process will continue to run until it makes a kernel call and hereby imposes a context switch or is preempted by a higher priority task. If more than one task is running in FIFO mode at the same priority, it can be useful for a mutual exclusion on shared resources. Even memory sharing without need of semaphoring will be possible in such a mode.

Round-Robin Scheduling: A process will continue to execute until it makes a kernel call, is preempted by a higher priority task or uses up his time slice.

Adaptive Scheduling: The principle of priority decaying[17] is applied to the processes depending on their blocking state. This is useful for keeping user-responses fast despite of heavy load of the processor due to computing intense background processes.

In QNX, as most transactions between processes will follow a server-client-model, these servers will more likely run at higher priorities than the clients requesting services from them. This request will be handled at the higher priority of the server, the client experiences a priority boosting during that time. If the server processes are running only for short times this will not be a problem, but to prevent problems arising from mixed priorities and therefore missed deadlines the priority of the server-task can be client-driven.

That means that the server task inherits the priority (but only the priority) from its client and behaves like the client prioritywise. A further problem arises from that feature: if a request arrives from a client with higher priority than the actual executing request inherited to the server, the new request would be changed to a lower priority as it should be, very likely following by missed deadlines. In order to be able to circumvent this, the new request may boost the priority of the server process again, to make sure the new request will be finished as soon as possible before continuing with the lower priority request.

2.4 Latency

Latency is the time between the occurrence of an external event and the start of the execution of the code responsible for that event. This time is to be minimized for a real-time system to be of any use.

In QNX two types of latency can occur:

Interrupt latency is the time from reception of an hardware interrupt to the execution of the responsible interrupt handler. As interrupts are fully enabled in QNX most of the time this value can be held very small. Only a few situations exist where there is need to disable the interrupts. The longest time of this disabling usually defines the worst-case scenario for interrupt latency.

Scheduling latency occurs in the need of an more sophisticated process (than the usually quite small interrupt handlers) to run when there will be a proxy triggered by the interrupt handler. The scheduling latency in

[17]If a process uses up his timeslice its priority will be decreased (but only one level below the original priority). On blocking the priority immediatly rises again to its original value.

QNX is defined as the time it takes from triggering the proxy to the start of execution of the code for the driver process. This usually will be the time it takes to execute the context switch on the specific processor.[18]

Of course the interrupt latency T_{il} and scheduling latency T_{sl} heavily depend on the processor in use. Some figures[19] may illustrate that:

Processor	typ. T_{il}	typ. T_{sl}
Pentium 166	3.3 μs	4.7 μs
Pentium 100	4.4 μs	6.7 μs
486DX4 100	5.6 μs	11.1 μs
386 EX 33	22.5 μs	74.2 μs

The execution of the interrupted process can be delayed further by the ability of QNX to handle nested (or stacked) interrupts as they are not disabled within interrupt handlers. This allows for higher priority interrupts to preempt lower priority interrupts. Each of these interrupts can also cause proxies to be triggered which will be executed after the finishing of all interrupt handlers at their respective priority. Worst-time considerations for lower-level interrupts therefore have to take all the execution times of higher-level interrupts into account.

3 Conclusions

There are several criteria as stated in 1.2 an RTOS has to feature. QNX does feature all of them and more; it is a preemptive multitasking system with a very small microkernel which provides for fast context switches and small latencies. Due to this the efficiency of the OS is high. As all communication within the system is message based including several blocking features, threads and I/O operations are automatically synchronized.

The microkernel is the only part that needs to run on every node. Every other module of the system can be accessed from every other node due to the network process manager included in the kernel itself. It is therefore possible to have only one TCP/IP-stack running on one distinctive node for all QNX-nodes to be able to communicate with other networks.

QNX as an OS as such provides the user with a powerful instrument to construct real-time systems and can therefore be considered an RTOS, but that does not take the responsibility from the user to take care of the environment and external influences the system will be operating in as they define the real-time behaviour of the system in which the OS works.

It has to be mentioned that the features of QNX do not end by the few discussed here but extends to specific implementations for handheld[20] or mobile[21] applications as well as fully featured, still small,[22] windowing systems.

[18] Most interrupts will not trigger a proxy though, as their purposes are quite small and well defined (feeding an interface with data from a buffer, counting timing ticks or reacting to user input).

[19] From [5, Microkernel, A word about realtime performance].

[20] See [4].

[21] See [6].

[22] The full Photon microGUI occupies only 500K of memory, as stated in [7, Embedded GUI], but can still be scaled down to about 250K as some considerations in [3, p.17] show.

A Acronyms

FCFS First-Come First-Served

GUI Graphical User Interface

IPC InterProcess Communication

IR-LAN InfraRed Local Area Network

NIC Network Interface Card

PDA Personal Digital Assistant

PID Process IDentifier

QoS Quality of Service

RF-LAN RadioFrequency Local Area Network

RTOS Real-Time Operating System

TCDS Time-Controlled Dynamic Scheduling

TCP/IP Transport Control Protocol / Internet Protocol

VC Virtual Circuit

VID Virtual process IDentifier

References

[1] Gerler Oliver, *Lecture Notes to CE4218,* University of Limerick, 2000

[2] Hildebrand Dan, *An Architectural Overview of QNX,* Proceedings of the Usenix Workshop on Micro-Kernels & Other Kernel Architectures, Seattle, April, 1992, ISBN 1-880446-42-1

[3] Hildebrand Dan, *A Microkernel POSIX OS for Realtime Embedded Systems,* Embedded Computer Conference, Santa Clara, California, April, 1993

[4] Hildebrand Dan, *QNX®: Microkernel Technology for Open Systems Handheld Computing,* Pen & Portable Computing Conference and Exposition, Boston, MA, May, 1994

[5] *The System Architecture Guide to QNX*
`http://support.qnx.com/support/docs/qnx4/sysarch/`

[6] *QNX®Neutrino®RTOS for the MobileGT®Automotie Platform*
`www.qnx.com/products/mobileGT/index.html`

[7] Hildebrand Dan, *Adapting PC Technology for Internet Appliances*
`http://www.swd.de/documents/papers/pcadapt_e.html`

[8] Bergsma Greg, *Realtime Extensions to Windows NT*
`http://www.swd.de/documents/papers/nt_e.html`

[9] Theaker Colin J., Brookes Graham R., *A Practical Course on Operating Systems,* The MacMillan Press Ltd., 1983, ISBN 0-333-34678-5

[10] Rajkumar R. (ed.), *Operating Systems And Services,* Kluwer Academic Publishers, 1999, ISBN 0-7923-8548-9

[11] *Windows NT as Real-Time OS?*
`http://www.realtime-info.be/encyc/magazine/97q2/winntasrtos.htm`